package fr.jmdoudoux.test;
import javax.swing.JFrame;
/**
* Ma fenetre
* @autho
*/
public cl
privat
public Window() {}
/**
* Fabrique de
* @return une
*/
public static Window creerWindow() {
Window window = new Window();
return window;
}
/**
* Point d'entrée de l'application
* @param argv
*/
public static void main(String[] argv) {
Window fenetre = creerWindow();
fenetre.setVisible(true);

Développons en Java

par Jean-Michel DOUDOUX

Partie 1 : Les bases du langage Java
Partie 2 : Les API de base
Partie 3 : Les API avancées
Partie 4 : La programmation parallèle et concurrente
Partie 5 : Le développement des interfaces graphiques
Partie 6 : L'utilisation de documents XML et JSON
Partie 7 : L'accès aux bases de données
Partie 8 : La machine virtuelle Java (JVM)
Partie 9 : Le développement d'applications d'entreprises
Partie 10 : Le développement d'applications web
Partie 11 : Le développement d'applications RIA / RDA
Partie 12 : Le développement d'applications avec Spring
Partie 13 : Les outils pour le développement
Partie 14 : La conception et le développement d'applications
Partie 15 : Les tests automatisés
Partie 16 : Les outils de profiling et monitoring
Partie 17 : Java et le monde informatique
Partie 18 : Le développement d'applications mobiles

Développons en Java v2.40

Jean Michel DOUDOUX

Table des matières

Développons en Java..........1

Préambule..........2
A propos de ce document..........2
Remerciements..........5
Notes de licence..........5
Marques déposées..........5
Historique des versions..........5

Partie 1 : Les bases du langage Java..........10

1. Présentation de Java..........11
1.1. Les caractéristiques..........11
1.2. Les logos de Java..........12
1.3. Un rapide historique de Java..........13
1.4. Les différentes éditions et versions de Java..........14
1.4.1. Les évolutions des plates-formes Java..........15
1.4.2. Les différentes versions de Java..........16
1.4.3. La gestion des évolutions dans la plateforme Java via des JEPs..........17
1.4.4. Java 1.0..........19
1.4.5. Java 1.1..........19
1.4.6. Java 1.2 (nom de code Playground)..........19
1.4.7. J2SE 1.3 (nom de code Kestrel)..........20
1.4.8. J2SE 1.4 (nom de code Merlin)..........20
1.4.9. J2SE 5.0 (nom de code Tiger)..........20
1.4.10. Java SE 6 (nom de code Mustang)..........21
1.4.10.1. Les évolutions de Java 6..........22
1.4.10.2. Java 6 update..........25
1.4.11. Java SE 7 (nom de code Dolphin)..........33
1.4.11.1. Les JSR de Java 7..........34
1.4.11.2. Java 7 update..........36
1.4.12. Java SE 8 (nom de code Spider)..........43
1.4.12.1. Les expressions Lambdas et les Streams..........44
1.4.12.2. Les autres évolutions dans les API et le langage..........44
1.4.12.3. Les évolutions dans la plate-forme..........45
1.4.12.4. Les profiles de Java SE..........46
1.4.12.5. Java 8 update..........48
1.4.13. Java SE 9..........56
1.4.13.1. Java 9 update..........58
1.4.14. Java SE 10..........58
1.4.14.1. Java 10 update..........59
1.4.15. Java SE 11..........59
1.4.15.1. Java 11 update..........60
1.4.16. Java SE 12..........63
1.4.16.1. Java 12 update..........63
1.4.17. Java SE 13..........64
1.4.17.1. Java 13 update..........64
1.4.18. Java SE 14..........64
1.4.18.1. Java 14 update..........65
1.4.19. Java SE 15..........65
1.4.19.1. Java 15 update..........66
1.4.20. Java SE 16..........66
1.4.20.1. Java 16 update..........67
1.4.21. Java SE 17..........67
1.4.21.1. Java 17 update..........68
1.4.22. Java SE 18..........69
1.4.22.1. Java 18 update..........69
1.4.23. Java SE 19..........70
1.4.23.1. Java 19 update..........70
1.4.24. Java SE 20..........70

Table des matières

1. Présentation de Java

1.4.24.1. Java 20 update....71
1.4.25. Java SE 21....71
1.4.25.1. Java 21 update....71
1.4.26. Le résumé des différentes versions....72
1.4.27. Le support des différentes versions....72
1.5. Un rapide tour d'horizon des API et de quelques outils....73
1.6. Les différences entre Java et JavaScript....74
1.7. L'installation du JDK....75
1.7.1. L'installation de la version 1.3 du JDK de Sun sous Windows 9x....75
1.7.2. L'installation de la documentation de Java 1.3 sous Windows....77
1.7.3. La configuration des variables système sous Windows 9x....78
1.7.4. Les éléments du JDK 1.3 sous Windows....79
1.7.5. L'installation de la version 1.4.2 du JDK de Sun sous Windows....79
1.7.6. L'installation de la version 1.5 du JDK de Sun sous Windows....80
1.7.7. Installation JDK 1.4.2 sous Linux Mandrake 10....81
1.7.8. L'installation de la version 1.8 du JDK d'Oracle sous Windows....83
1.7.9. Installation du JDK sur Ubuntu 20.10 64 bits sur Raspberry Pi 4....85
1.7.9.1. Pré-installation....85
1.7.9.2. Installer le JRE par défaut....85
1.7.9.3. Installer le JDK par défaut....86
1.7.9.4. Installation d'une autre version....88
1.7.9.5. L'installation d'un JDK 8....88
1.7.9.6. La variable d'environnement JAVA_HOME....89
1.7.9.7. La configuration de la version utilisée....90

2. Les notions et techniques de base en Java....94
2.1. Les concepts de base....94
2.1.1. La compilation et l'exécution....94
2.1.2. Les packages....97
2.1.3. Le déploiement sous la forme d'un jar....99
2.1.4. Le classpath....100
2.1.4.1. La définition du classpath pour exécuter une application....102
2.1.4.2. La définition du classpath pour exécuter une application avec la variable CLASSPATH....103
2.1.4.3. La définition du classpath pour exécuter une application utilisant une ou plusieurs bibliothèques....104
2.1.4.4. La définition du classpath pour exécuter une application packagée en jar....105
2.2. L'exécution d'une applet....107
2.3. L'utilisation de fonctionnalités non standards....107
2.3.1. Les fonctionnalités en preview....108
2.3.1.1. La définition de fonctionnalités en preview....108
2.3.1.2. Les APIs en relation avec une fonctionnalité en preview....110
2.3.1.3. L'utilisation de fonctionnalités en preview....110
2.3.2. Les modules en incubation (incubator modules)....116
2.3.3. Les fonctionnalités expérimentales....117
2.4. Les builds early access....117

3. La syntaxe et les éléments de bases de Java....118
3.1. Les règles de base....118
3.2. Les mots réservés du langage Java....119
3.3. Les identifiants....123
3.4. Les commentaires....124
3.5. La déclaration et l'utilisation de variables....125
3.5.1. La déclaration de variables....125
3.5.2. Les types élémentaires....126
3.5.3. Le format des types élémentaires....126
3.5.4. L'initialisation des variables....127
3.5.5. L'affectation....128
3.5.6. Les entiers exprimés en binaire (Binary Literals)....128
3.5.7. Utilisation des underscores dans les entiers littéraux....129

Table des matières

3. La syntaxe et les éléments de bases de Java

3.6. La déclaration de variables locales avec l'inférence de type....129
3.6.1. L'instruction var....131
3.6.2. Les restrictions d'utilisation de l'instruction var....132
3.6.3. Les avantages et les inconvénients....135
3.6.3.1. La facilitation de la déclaration de variables locales....136
3.6.3.2. La lisibilité....136
3.6.4. Quelques mises en garde sur l'utilisation de var....137
3.6.5. Les messages d'erreur courants lors d'une utilisation incorrecte de var....140
3.6.6. Les comparaisons....142
3.7. Les opérations arithmétiques....142
3.7.1. L'arithmétique entière....143
3.7.2. L'arithmétique en virgule flottante....143
3.7.3. L'incrémentation et la décrémentation....144
3.8. La priorité des opérateurs....145
3.9. Les structures de contrôles....145
3.9.1. Les boucles....146
3.9.1.1. Les boucles while....146
3.9.1.2. Les boucles do ... while....147
3.9.1.3. Les boucles for....147
3.9.1.4. Les boucles for améliorées....148
3.9.2. Les branchements conditionnels....150
3.9.2.1. L'instruction if....150
3.9.2.2. L'opérateur ternaire....151
3.9.2.3. L'instruction switch....151
3.9.3. Les débranchements....152
3.9.3.1. L'instruction break....153
3.9.3.2. L'instruction continue....153
3.9.3.3. L'utilisation de débranchement avec étiquette....155
3.9.3.4. L'instruction return....157
3.10. Les évolutions de l'instruction switch dans Java 14....158
3.10.1. La syntaxe de l'instruction switch avec l'opérateur arrow....159
3.10.2. L'utilisation de l'instruction switch comme une expression....160
3.10.2.1. L'instruction yield pour retourner une valeur....160
3.10.2.2. Les contraintes....161
3.10.3. L'utilisation de valeurs multiples dans une clause case....162
3.10.4. Les 4 formes de l'utilisation de switch....162
3.10.4.1. L'utilisation de switch comme structure de contrôle avec la syntaxe classique....163
3.10.4.2. L'utilisation de switch comme expression avec la syntaxe classique....165
3.10.4.3. L'utilisation de switch comme expression avec l'opérateur arrow....166
3.10.4.4. L'utilisation de switch comme structure de contrôle avec l'opérateur arrow....168
3.10.4.5. Le résumé des différentes formes d'utilisation....169
3.10.5. Les situations illicites....170
3.10.5.1. Des types différents retournés par les cases d'un switch comme expression....170
3.10.5.2. Dans un switch comme structure de contrôle, on ne peut pas retourner de valeur....171
3.10.5.3. Dans un switch comme expression, il faut retourner une valeur après l'opérateur ->....172
3.10.5.4. Dans un switch comme traitement, l'opérateur -> ne peut pas retourner de valeur....172
3.10.5.5. Toutes les valeurs possibles doivent être prises en compte dans un switch utilisé comme expression....173
3.10.5.6. Il n'est pas possible de mixer les deux syntaxes d'une clause case....174
3.10.5.7. L'utilisation de l'instruction continue dans un switch....174
3.10.5.8. L'utilisation de l'instruction break dans un switch....177
3.10.5.9. L'utilisation de return dans un switch comme expression....180
3.11. Les tableaux....181
3.11.1. La déclaration et l'allocation des tableaux....181
3.11.2. L'initialisation explicite d'un tableau....182
3.11.3. Le parcours d'un tableau....182
3.12. Les conversions de types....183
3.12.1. La conversion d'un entier int en chaîne de caractères String....183
3.12.2. La conversion d'une chaîne de caractères String en entier int....183

Table des matières

3. La syntaxe et les éléments de bases de Java

3.12.3. La conversion d'un entier int en entier long....184
3.13. L'autoboxing et l'unboxing....184

4. La programmation orientée objet....185
4.1. Le concept de classe....185
4.1.1. La syntaxe de déclaration d'une classe....186
4.2. Les objets....186
4.2.1. La création d'un objet : instancier une classe....186
4.2.2. La durée de vie d'un objet....187
4.2.3. Les références et la comparaison d'objets....187
4.2.4. Le littéral null....188
4.2.5. Les variables de classes....188
4.2.6. La variable this....188
4.2.7. L'opérateur instanceof....189
4.3. Les modificateurs d'accès....189
4.3.1. Les mots clés qui gèrent la visibilité des entités....190
4.3.2. Le mot clé static....190
4.3.3. Le mot clé final....191
4.3.4. Le mot clé abstract....192
4.3.5. Le mot clé synchronized....192
4.3.6. Le mot clé volatile....193
4.3.7. Le mot clé native....193
4.4. Les propriétés ou attributs....193
4.4.1. Les variables d'instances....193
4.4.2. Les variables de classes....193
4.4.3. Les constantes....193
4.5. Les méthodes....194
4.5.1. La syntaxe de la déclaration....194
4.5.2. La transmission de paramètres....195
4.5.3. L'émission de messages....196
4.5.4. L'enchaînement de références à des variables et à des méthodes....196
4.5.5. Les arguments variables (varargs)....196
4.5.6. La surcharge de méthodes....197
4.5.7. Les constructeurs....198
4.5.8. Les accesseurs....199
4.6. L'héritage....199
4.6.1. Le principe de l'héritage....200
4.6.2. La mise en oeuvre de l'héritage....200
4.6.3. L'accès aux propriétés héritées....200
4.6.4. La redéfinition d'une méthode héritée....200
4.6.5. Le polymorphisme....201
4.6.6. Le transtypage induit par l'héritage facilite le polymorphisme....201
4.6.7. Les interfaces et l'héritage multiple....201
4.6.7.1. Les méthodes par défaut....203
4.6.7.2. Les interfaces locales....210
4.6.8. L'héritage de méthodes statiques....212
4.6.9. Des conseils sur l'héritage....215
4.7. Les packages....215
4.7.1. La définition d'un package....216
4.7.2. Les importations....216
4.7.3. Les importations statiques....217
4.7.4. La collision de classes....218
4.7.5. Les packages et l'environnement système....218
4.8. Les classes internes....218
4.8.1. Les classes internes non statiques....220
4.8.2. Les classes internes locales....224
4.8.3. Les classes internes anonymes....227
4.8.4. Les classes internes statiques....227
4.8.5. Les membres static dans une classe interne....228

Table des matières

4. La programmation orientée objet

4.9. Les types scellés 230

4.9.1. L'historique de la fonctionnalité 231

4.9.2. Les possibilités pour limiter les classes d'une hiérarchie avant les types scellés 231

4.9.3. Le rôle des types scellés 232

4.9.4. La définition d'une classe scellée 233

4.9.5. Les contraintes sur les classes filles 233

4.9.5.1. Les modificateurs final, sealed ou non-sealed 234

4.9.5.2. La déclaration implicite des classes filles 235

4.9.5.3. La localisation des classes filles 237

4.9.6. Les interfaces scellées 237

4.9.7. Les types scellés et les records 238

4.9.8. Les vérifications à l'exécution 238

4.9.9. Les situations illicites 239

4.9.10. L'opérateur instanceof et les classes scellées 241

4.10. La gestion dynamique des objets 243

5. Les génériques (generics) 244

5.1. Le besoin des génériques 246

5.1.1. L'apport des génériques 248

5.1.2. Les génériques et l'API Collection 248

5.2. La définition des concepts 249

5.3. L'utilisation de types génériques 250

5.3.1. L'opérateur diamant 251

5.3.2. Le type brut (raw type) 253

5.4. La définition de types génériques 256

5.4.1. La portée des paramètres de type 257

5.4.2. Les conventions de nommage sur les noms des types 257

5.4.3. L'utilisation de plusieurs paramètres de types 257

5.4.4. Les interfaces génériques 258

5.4.5. Les tableaux de génériques 258

5.5. La pollution du heap (Heap Pollution) 259

5.6. La mise en oeuvre des génériques 261

5.6.1. Les génériques et l'héritage 262

5.6.2. Les classes génériques et le sous-typage 263

5.6.3. Le transtypage des instances génériques 265

5.7. Les méthodes et les constructeurs génériques 266

5.8. Les paramètres de type bornés (bounded type parameters) 268

5.9. Les paramètres de type avec wildcard 270

5.9.1. Le besoin des wildcards avec les génériques 271

5.9.2. L'utilisation de wildcards dans les types paramétrés 272

5.9.2.1. Les types paramétrés avec wildcard non bornés (Unbounded wildcard parameterized type) 273

5.9.2.2. Les types paramétrés avec wildcard borné (Bounded wildcard parameterized type) 277

5.9.2.3. Les wildcards avec bornes supérieures (Upper Bounded Wildcards) 278

5.9.2.4. Les wildcards avec bornes inférieures (Lower Bounded Wildcards) 282

5.9.2.5. Comment choisir entre borne inférieure et supérieure 285

5.9.2.6. Les wildcards et le sous-typage 286

5.9.2.7. La capture des wildcards et les méthodes helper 288

5.10. Les bornes multiples (Multiple Bounds) avec l'intersection de types 295

5.11. L'effacement de type (type erasure) 299

5.11.1. Le choix de l'effacement de type 299

5.11.2. La mise en oeuvre de l'effacement de type 299

5.11.2.1. L'effacement de type pour un type paramétré inconnu 300

5.11.2.2. L'effacement de type pour un type paramétré avec borne supérieure 302

5.11.2.3. Les méthodes pont (bridge method) 304

5.12. La différence entre l'utilisation d'un argument de type et un wildcard 308

5.13. Les conséquences et les effets de bord de l'effacement de type 309

5.13.1. Les types génériques et les casts 309

5.13.2. Le type des instances génériques 310

Table des matières

5. Les génériques (generics)

5.13.3. Les génériques et les méthodes surchargées....311
5.13.4. Les génériques et l'opérateur instanceof....311
5.13.5. Les surcharges avec un type générique et un type Object....313
5.13.6. La création d'une instance de type générique....313
5.13.7. La création d'une instance d'un tableau de type générique....315
5.13.8. Les varargs et les génériques....315
5.13.9. Les collisions de méthodes liées à l'effacement de type....317
5.13.10. Le contournement lors de l'utilisation d'un type brut d'une classe générique....318
5.14. Les restrictions dans l'utilisation des génériques....320
5.14.1. L'implémentation plusieurs fois de la même interface générique....320
5.14.2. Les génériques ne supportent pas le sous-typage....320
5.14.3. Les types génériques et les membres statiques....322
5.14.4. Les génériques et les types primitifs....322
5.14.5. Une exception ne peut pas être générique....323
5.14.6. Une annotation ne peut pas être générique....324
5.14.7. Une énumération ne peut pas être générique....324

6. Les chaînes de caractères....325
6.1. Les chaînes de caractères littérales....326
6.1.1. Les chaînes de caractères littérales mono ligne....326
6.1.2. Les séquences d'échappement....327
6.2. La classe java.lang.String....328
6.2.1. Les constructeurs....328
6.2.2. Les méthodes....329
6.2.3. La classe String est immuable....333
6.2.4. L'encapsulation des caractères des chaînes de caractères....333
6.3. La création d'un objet de type String....334
6.3.1. L'utilisation de la syntaxe littérale....334
6.3.2. L'utilisation d'un constructeur....334
6.3.3. Syntaxe littérale vs utilisation d'un constructeur....334
6.4. Les opérations sur les chaînes de caractères....335
6.4.1. Le test de l'égalité de deux chaînes de caractères....335
6.4.1.1. equals() vs ==....336
6.4.2. La comparaison de chaînes de caractère....337
6.4.3. Le formatage d'une chaîne de caractères....339
6.4.3.1. Le format de la chaîne de formatage....340
6.4.3.2. Les conversions....340
6.4.3.3. Les conversions de données temporelles....341
6.4.3.4. Les flags....344
6.4.3.5. La taille (Width)....345
6.4.3.6. La précision (Precision)....345
6.4.3.7. Les index des arguments....346
6.4.4. La conversion en minuscules ou en majuscules....346
6.4.5. L'obtention d'une sous-chaîne....347
6.4.6. L'obtention de la taille d'une chaîne....347
6.4.7. Le découpage d'une chaîne de caractères....347
6.4.8. La vérification du contenu d'une chaîne....347
6.4.9. L'obtention de la position d'un caractère ou d'une chaîne....348
6.4.10. L'obtention d'un caractère de la chaîne....350
6.4.11. La suppression des espaces de début et de fin....350
6.4.12. Tester si une chaîne est vide....352
6.4.13. Tester si une chaîne est vide ou ne contient que des espaces....353
6.4.14. Le remplacement de caractères ou de sous-chaînes....354
6.5. La conversion de et vers une chaîne de caractères....355
6.5.1. La conversion d'une chaîne en tableau d'octets....355
6.5.2. La conversion d'une chaîne en tableau de caractères....355
6.5.3. La conversion d'un type primitif en chaîne....356
6.5.4. La conversion d'un objet en chaîne de caractères....356
6.5.5. La conversion d'une chaîne en entier....356

Table des matières

6. Les chaînes de caractères

6.6. La concaténation de chaînes de caractères....357
6.6.1. La concaténation avec l'opérateur +....357
6.6.2. La concaténation avec la méthode concat()....358
6.6.3. La classe StringBuffer....358
6.6.4. La classe StringBuilder....361
6.6.5. Différence entre StringBuilder et StringBuffer....363
6.6.6. La classe StringJoiner....363
6.6.7. La méthode join() de la classe String....364
6.7. La classe StringTokenizer....365
6.8. Les chaînes de caractères et la sécurité....366
6.9. Le stockage en mémoire....366
6.9.1. Le pool de String....367
6.10. Les blocs de texte (Text Blocks)....368
6.10.1. La situation historique....368
6.10.2. Les solutions proposées....369
6.10.3. La description des blocs de texte....369
6.10.4. Les avantages....371
6.10.5. La syntaxe....371
6.10.6. La gestion de l'indentation accessoire et essentielle....373
6.10.7. L'utilisation des séquences d'échappement....375
6.10.7.1. Les séquences d'échappement ajoutées en Java 14....377
6.10.8. Les traitements par le compilateur....378
6.10.8.1. L'uniformisation des retours chariots....379
6.10.8.2. La gestion des espaces pour l'indentation et en fin de ligne....379
6.10.8.3. Le traitement des séquences d'échappement....381
6.10.8.4. Le résultat de la transformation....382
6.10.9. La concaténation et le formatage de blocs de texte....382
6.10.10. Les méthodes dans la classe String....383
6.10.10.1. La méthode translateEscapes....384
6.10.10.2. La méthode formatted....384
6.10.10.3. La méthode stripIndent....385
6.10.11. Les cas d'utilisation....385
6.10.12. Les bonnes pratiques....386

7. Les packages de bases....388
7.1. Les packages selon la version du JDK....388
7.2. Le package java.lang....395
7.2.1. La classe Object....395
7.2.1.1. La méthode getClass()....395
7.2.1.2. La méthode toString()....396
7.2.1.3. La méthode equals()....396
7.2.1.4. La méthode finalize()....396
7.2.1.5. La méthode clone()....397
7.2.2. Les classes de manipulations de chaînes de caractères....397
7.2.3. Les wrappers....397
7.2.4. La classe System....398
7.2.4.1. L'utilisation des flux d'entrée/sortie standard....398
7.2.4.2. Les variables d'environnement et les propriétés du système....400
7.2.5. Les classes Runtime et Process....402
7.3. La présentation rapide du package awt java....405
7.4. La présentation rapide du package java.io....405
7.5. Le package java.util....405
7.5.1. La classe Random....405
7.5.2. Les classes Date et Calendar....406
7.5.3. La classe SimpleDateFormat....407
7.5.4. La classe Vector....407
7.5.5. La classe Hashtable....408
7.5.6. L'interface Enumeration....409
7.5.7. La manipulation d'archives zip....410

Table des matières

7. Les packages de bases

7.5.8. Les expressions régulières....413
7.5.8.1. Les motifs....413
7.5.8.2. La classe Pattern....415
7.5.8.3. La classe Matcher....415
7.5.9. La classe Formatter....417
7.5.10. La classe Scanner....418
7.6. La présentation rapide du package java.net....419
7.7. La présentation rapide du package java.applet....419

8. Les fonctions mathématiques....420

8.1. Les variables de classe....420
8.2. Les fonctions trigonométriques....421
8.3. Les fonctions de comparaisons....421
8.4. Les arrondis....421
8.4.1. La méthode round(n)....421
8.4.2. La méthode rint(double)....422
8.4.3. La méthode floor(double)....422
8.4.4. La méthode ceil(double)....422
8.4.5. La méthode abs(x)....423
8.5. La méthode IEEEremainder(double, double)....423
8.6. Les Exponentielles et puissances....423
8.6.1. La méthode pow(double, double)....423
8.6.2. La méthode sqrt(double)....424
8.6.3. La méthode exp(double)....424
8.6.4. La méthode log(double)....424
8.7. La génération de nombres aléatoires....424
8.8. La classe BigDecimal....425
8.9. La précision des calculs en virgule flottante....430
8.9.1. Les calculs en virgule flottante stricte....430
8.9.2. Le mot clé réservé strictfp....431
8.9.2.1. L'utilisation de strictfp sur des classes....432
8.9.2.2. L'utilisation de strictfp sur des méthodes non abstraites....433
8.9.2.3. L'utilisation de strictfp sur des interfaces....433
8.9.2.4. Les cas d'utilisation invalides de strictfp....434
8.9.2.5. L'utilisation de strictfp à partir de Java 17....435

9. La gestion des exceptions....436

9.1. Les mots clés try, catch et finally....437
9.2. La classe Throwable....438
9.3. Les classes Exception, RunTimeException et Error....439
9.4. Les exceptions personnalisées....439
9.5. Les exceptions chaînées....440
9.6. L'utilisation des exceptions....441
9.7. L'instruction try-with-resources....442
9.8. Des types plus précis lorsqu'une exception est relevée dans une clause catch....450
9.9. Multiples exceptions dans une clause catch....453

10. Les énumérations (type enum)....455

10.0.1. La définition d'une énumération....456
10.0.2. L'utilisation d'une énumération....457
10.0.3. L'enrichissement de l'énumération....458
10.0.4. La personnalisation de chaque élément....461
10.0.5. Les limitations dans la mise en oeuvre des énumérations....463
10.1. Les énumérations locales....463

11. Les annotations....466

11.1. La présentation des annotations....466
11.2. La mise en oeuvre des annotations....467
11.3. L'utilisation des annotations....468

Table des matières

11. Les annotations

11.3.1. La documentation....469
11.3.2. L'utilisation par le compilateur....469
11.3.3. La génération de code....469
11.3.4. La génération de fichiers....469
11.3.5. Les API qui utilisent les annotations....469
11.4. Les annotations standard....470
11.4.1. L'annotation @Deprecated....470
11.4.2. L'annotation @Override....471
11.4.3. L'annotation @SuppressWarning....472
11.4.3.1. L'utilisation de l'annotation @SuppressWarnings....472
11.4.3.2. Les attributs de l'annotation @SuppressWarnings....473
11.4.3.3. Les options du compilateur pour afficher les warnings....474
11.4.3.4. Les options du compilateur d'OpenJDK pour afficher les warnings....475
11.4.3.5. Les bonnes pratiques d'utilisation....505
11.5. Les annotations communes (Common Annotations)....506
11.5.1. L'annotation @Generated....506
11.5.2. Les annotations @Resource et @Resources....507
11.5.3. Les annotations @PostConstruct et @PreDestroy....507
11.6. Les annotations personnalisées....508
11.6.1. La définition d'une annotation....508
11.6.2. Les annotations pour les annotations....510
11.6.2.1. L'annotation @Target....510
11.6.2.2. L'annotation @Retention....511
11.6.2.3. L'annotation @Documented....511
11.6.2.4. L'annotation @Inherited....512
11.7. L'exploitation des annotations....512
11.7.1. L'exploitation des annotations dans un Doclet....512
11.7.2. L'exploitation des annotations avec l'outil Apt....513
11.7.3. L'exploitation des annotations par introspection....519
11.7.4. L'exploitation des annotations par le compilateur Java....521
11.8. L'API Pluggable Annotation Processing....521
11.8.1. Les processeurs d'annotations....522
11.8.2. L'utilisation des processeurs par le compilateur....523
11.8.3. La création de nouveaux fichiers....524
11.9. Les ressources relatives aux annotations....525

12. Les expressions lambda....526
12.1. L'historique des lambdas pour Java....527
12.2. Les expressions lambda....528
12.2.1. La syntaxe d'une expression lambda....529
12.2.1.1. Les paramètres d'une expression lambda....529
12.2.1.2. Le corps d'une expression lambda....532
12.2.2. Des exemples d'expressions lambda....533
12.2.3. La portée des variables....534
12.2.4. L'utilisation d'une expression lambda....537
12.3. Les références de méthodes....539
12.3.1. La référence à une méthode statique....541
12.3.2. La référence à une méthode d'une instance....544
12.3.3. La référence à une méthode d'une instance arbitraire d'un type....545
12.3.4. La référence à un constructeur....547
12.4. Les interfaces fonctionnelles....549
12.4.1. L'annotation @FunctionalInterface....550
12.4.2. La définition d'une interface fonctionnelle....550
12.4.3. L'utilisation d'une interface fonctionnelle....552
12.4.4. Les interfaces fonctionnelles du package java.util.function....556
12.4.4.1. Les interfaces fonctionnelles de types Consumer....558
12.4.4.2. Les interfaces fonctionnelles de type Function....560
12.4.4.3. Les interfaces fonctionnelles de type Predicate....572
12.4.4.4. Les interfaces fonctionnelles de type Supplier....576

Table des matières

13. Les records..........579
13.1. L'introduction aux records..........580
13.1.1. L'historique..........580
13.1.2. Le problème de la verbosité..........581
13.1.3. Présentation des records..........582
13.2. La définition d'un record..........584
13.2.1. Le record minimaliste..........584
13.2.2. L'identifiant restreint record..........585
13.2.3. L'ajout des composants..........585
13.2.4. L'implémentation d'interfaces..........587
13.2.5. L'utilisation d'annotations avec les records..........587
13.2.6. Le support des génériques..........588
13.2.7. L'immutabilité..........589
13.2.8. La redéfinition de membres..........591
13.2.8.1. Le constructeur généré et sa redéfinition..........591
13.2.8.2. La redéfinition d'un accesseur..........593
13.2.9. L'ajout de membres à un record..........593
13.2.9.1. L'implémentation de constructeurs..........594
13.2.9.2. L'ajout de membres statiques..........595
13.2.9.3. Les méthodes d'instances..........596
13.3. La mise en oeuvre des records..........596
13.3.1. Les records comme membre imbriqué..........596
13.3.2. Les records locaux..........597
13.3.3. Les limitations et les incompatibilités..........598
13.3.4. La sérialisation des records..........599
13.3.5. L'introspection sur les records..........602

Partie 2 : Les API de base..........604

14. Les collections..........605
14.1. Présentation du framework collection..........605
14.2. Les interfaces des collections..........608
14.2.1. L'interface Collection..........609
14.2.2. L'interface Iterator..........611
14.3. Les collections de type List : les listes..........611
14.3.1. L'interface List..........612
14.3.2. La classe Vector..........614
14.3.3. La classe ArrayList..........615
14.3.4. Les listes chaînées : la classe LinkedList..........618
14.3.5. L'interface ListIterator..........620
14.3.6. La classe CopyOnWriteArrayList..........620
14.3.7. Le choix d'une implémentation de type List..........622
14.4. Les collections de type Set : les ensembles..........623
14.4.1. L'interface Set..........624
14.4.2. L'interface SortedSet..........625
14.4.3. L'interface NavigableSet..........626
14.4.4. La classe HashSet..........628
14.4.5. La classe TreeSet..........629
14.4.6. La classe ConcurrentSkipListSet..........631
14.4.7. La classe CopyOnWriteArraySet..........633
14.4.8. Le choix d'une implémentation de type Set..........634
14.5. Les collections de type Map : les associations de type clé/valeur..........634
14.5.1. L'interface Map..........636
14.5.2. L'interface SortedMap..........637
14.5.3. La classe Hashtable..........638
14.5.4. La classe HashMap..........639
14.5.5. La classe LinkedHashMap..........643
14.5.6. La classe TreeMap..........645
14.5.7. La classe WeakHashMap..........647
14.5.8. La classe EnumMap..........649

Table des matières

14. Les collections
14.5.9. La classe IdentityHashMap....650
14.5.10. L'interface NavigableMap....652
14.5.11. L'interface ConcurrentNavigableMap....653
14.5.12. La classe ConcurrentSkipListMap....654
14.5.13. L'interface ConcurrentMap....656
14.5.14. La classe ConcurrentHashMap....656
14.5.15. Le choix d'une implémentation de type Map....659
14.6. Les collections de type Queue : les files....660
14.6.1. L'interface Queue....661
14.6.2. La classe AbstractQueue....661
14.6.3. L'interface TransferQueue....662
14.6.4. La classe LinkedTransferQueue....662
14.6.5. La classe PriorityQueue....664
14.6.6. La classe ConcurrentLinkedQueue....665
14.6.7. Les files utilisables par les deux bouts (Deque)....666
14.6.7.1. L'interface java.util.Deque....666
14.6.7.2. La classe ArrayDeque....668
14.6.8. Les files d'attente....668
14.6.8.1. L'interface java.util.concurrent.BlockingQueue....669
14.6.8.2. La classe java.util.concurrent.ArrayBlockingQueue....671
14.6.8.3. La classe java.util.concurrent.LinkedBlockingQueue....672
14.6.8.4. La classe java.util.concurent.PriorityBlockingQueue....673
14.6.8.5. La classe java.util.concurrent.DelayQueue....676
14.6.8.6. La classe java.util.concurrrent.SynchronousQueue....679
14.6.8.7. L'interface java.util.concurrent.BlockingDeque....680
14.6.8.8. La classe LinkedBlockingDeque....681
14.6.9. Le choix d'une implémentation de type Queue....682
14.7. Le tri des collections....683
14.7.1. L'interface Comparable....683
14.7.2. L'interface Comparator....684
14.8. Les algorithmes....684
14.9. Les exceptions du framework....685

15. Les flux d'entrée/sortie....686
15.1. La présentation des flux....686
15.2. Les classes de gestion des flux....686
15.3. Les flux de caractères....688
15.3.1. La classe Reader....689
15.3.2. La classe Writer....690
15.3.3. Les flux de caractères avec un fichier....690
15.3.3.1. Les flux de caractères en lecture sur un fichier....690
15.3.3.2. Les flux de caractères en écriture sur un fichier....690
15.3.4. Les flux de caractères tamponnés avec un fichier.....691
15.3.4.1. Les flux de caractères tamponnés en lecture avec un fichier....691
15.3.4.2. Les flux de caractères tamponnés en écriture avec un fichier....692
15.3.4.3. La classe PrintWriter....693
15.4. Les flux d'octets....694
15.4.1. Les flux d'octets avec un fichier.....695
15.4.1.1. Les flux d'octets en lecture sur un fichier....695
15.4.1.2. Les flux d'octets en écriture sur un fichier....696
15.4.2. Les flux d'octets tamponnés avec un fichier.....697
15.5. La classe File....698
15.6. Les fichiers à accès direct....700
15.7. La classe java.io.Console....701

16. NIO 2....703
16.1. Les entrées/sorties avec Java....704
16.2. Les principales classes et interfaces....704
16.3. L'interface Path....705

Table des matières

16. NIO 2

16.3.1. L'obtention d'une instance de type Path....706
16.3.2. L'obtention d'éléments du chemin....706
16.3.3. La manipulation d'un chemin....708
16.3.4. La comparaison de chemins....709
16.3.5. La conversion d'un chemin....710
16.4. Glob....711
16.5. La classe Files....713
16.5.1. Les vérifications sur un fichier ou un répertoire....713
16.5.2. La création d'un fichier ou d'un répertoire....714
16.5.3. La copie d'un fichier ou d'un répertoire....717
16.5.4. Le déplacement d'un fichier ou d'un répertoire....718
16.5.5. La suppression d'un fichier ou d'un répertoire....721
16.5.6. L'obtention du type de fichier....721
16.6. Le parcours du contenu de répertoires....723
16.6.1. Le parcours d'un répertoire....723
16.6.2. Le parcours d'une hiérarchie de répertoires....726
16.6.3. Les opérations récursives....728
16.7. L'utilisation de systèmes de gestion de fichiers....729
16.7.1. La classe FileSystems....729
16.7.2. La classe FileSystem....729
16.7.3. La création d'une implémentation de FileSystem....730
16.7.4. Une implémentation de FileSystem pour les fichiers Zip....730
16.8. La lecture et l'écriture dans un fichier....732
16.8.1. Les options d'ouverture d'un fichier....733
16.8.2. La lecture et l'écriture de l'intégralité d'un fichier....733
16.8.3. La lecture et l'écriture bufférisées d'un fichier....734
16.8.4. La lecture et l'écriture d'un flux d'octets....735
16.8.5. La lecture et l'écriture d'un fichier avec un channel....736
16.9. Les liens et les liens symboliques....738
16.9.1. La création d'un lien physique....738
16.9.2. La création d'un lien symbolique....739
16.9.3. L'utilisation des liens et des liens symboliques....739
16.10. La gestion des attributs....740
16.10.1. La gestion individuelle des attributs....740
16.10.2. La gestion de plusieurs attributs....741
16.10.3. L'utilisation des vues....742
16.10.4. La gestion des permissions DOS....743
16.10.5. La gestion des permissions Posix....744
16.11. La gestion des unités de stockages....746
16.12. Les notifications de changements dans un répertoire....747
16.12.1. La surveillance d'un répertoire....748
16.12.2. L'obtention des événements....749
16.12.3. Le traitement des événements....750
16.12.4. Un exemple complet....751
16.12.5. L'utilisation et les limites de l'API WatchService....753
16.13. La gestion des erreurs et la libération des ressources....753
16.14. L'interopérabilité avec le code existant....755
16.14.1. L'équivalence des fonctionnalités entre java.io et NIO2....756

17. La sérialisation....758
17.1. La sérialisation standard....759
17.1.1. La sérialisation binaire....759
17.1.1.1. L'interface Serializable....761
17.1.1.2. La classe ObjectOutputStream....762
17.1.1.3. La classe ObjectInputStream....766
17.1.1.4. Le mot clé transient....769
17.1.1.5. La gestion des versions d'une classe sérialisable....772
17.1.1.6. La compatibilité des versions de classes sérialisées....776
17.1.1.7. Des points particuliers....777

Table des matières

17. La sérialisation

17.1.1.8. La vérification d'un objet désérialisé....780
17.1.1.9. Les exceptions liées à la sérialisation....782
17.1.2. La sérialisation personnalisée....782
17.1.2.1. La définition des champs à sérialiser....782
17.1.2.2. Les méthodes writeObject() et readObject()....784
17.1.2.3. La méthode readObjectNoData()....790
17.1.2.4. Les méthodes writeReplace() et readResolve()....792
17.1.2.5. L'interface Externalizable....794
17.1.2.6. Les différences entre Serializable et Externalizable....801
17.2. La documentation d'une classe sérialisable....802
17.3. La sérialisation et la sécurité....803
17.4. La sérialisation en XML....804
17.4.1. La classe XMLEncoder....804
17.4.2. La classe XMLDecoder....806
17.4.3. La personnalisation de la sérialisation....807
17.4.3.1. La classe PersistenceDelegate....808
17.4.3.2. La classe DefaultPersistenceDelegate....808
17.4.3.3. Empêcher la sérialisation d'un attribut....809
17.4.3.4. Sérialiser des attributs sans accesseur/modificateur standard....811
17.4.3.5. Sérialiser une classe sans constructeur par défaut....813
17.4.3.6. Les arguments du constructeur sont des champs de la classe....816
17.4.3.7. Les arguments du constructeur ne sont pas des champs de la classe....817
17.4.3.8. La gestion des exceptions....818

18. L'interaction avec le réseau....821
18.1. L'introduction aux concepts liés au réseau....821
18.2. Les adresses internet....822
18.2.1. La classe InetAddress....822
18.3. L'accès aux ressources avec une URL....823
18.3.1. La classe URL....824
18.3.2. La classe URLConnection....824
18.3.3. La classe URLEncoder....825
18.3.4. La classe HttpURLConnection....826
18.4. L'utilisation du protocole TCP....827
18.4.1. La classe ServerSocket....827
18.4.2. La classe Socket....829
18.5. L'utilisation du protocole UDP....830
18.5.1. La classe DatagramSocket....830
18.5.2. La classe DatagramPacket....831
18.5.3. Un exemple de serveur et de client....831
18.6. Les exceptions liées au réseau....833
18.7. Les interfaces de connexions au réseau....833

19. L'internationalisation....835
19.1. Les objets de type Locale....835
19.1.1. La création d'un objet Locale....835
19.1.2. L'obtention de la liste des Locales disponibles....836
19.1.3. L'utilisation d'un objet Locale....837
19.2. La classe ResourceBundle....837
19.2.1. La création d'un objet ResourceBundle....837
19.2.2. Les sous-classes de ResourceBundle....837
19.2.2.1. L'utilisation de PropertyResourceBundle....838
19.2.2.2. L'utilisation de ListResourceBundle....838
19.2.3. L'obtention d'un texte d'un objet ResourceBundle....839
19.3. Un guide pour réaliser la localisation....839
19.3.1. L'utilisation d'un ResourceBundle avec un fichier propriétés....839
19.3.2. Des exemples de classes utilisant PropertiesResourceBundle....840
19.3.3. L'utilisation de la classe ListResourceBundle....841
19.3.4. Des exemples de classes utilisant ListResourceBundle....842

Table des matières

19. L'internationalisation
19.3.5. La création de sa propre classe fille de ResourceBundle....844

20. Les composants Java beans....848
20.1. La présentation des Java beans....848
20.2. Les propriétés....849
20.2.1. Les propriétés simples....849
20.2.2. Les propriétés indexées (indexed properties)....850
20.2.3. Les propriétés liées (Bound properties)....850
20.2.4. Les propriétés liées avec contraintes (Constrained properties)....852
20.3. Les méthodes....854
20.4. Les événements....854
20.5. L'introspection....854
20.5.1. Les modèles (design patterns)....855
20.5.2. La classe BeanInfo....855
20.6. Paramétrage du bean (Customization)....857
20.7. La persistance....857
20.8. La diffusion sous forme de jar....857

21. Le logging....859
21.1. La présentation du logging....859
21.1.1. Des recommandations lors de la mise en oeuvre....860
21.1.2. Les différents frameworks....860
21.2. Log4j....861
21.2.1. Les premiers pas....862
21.2.1.1. L'installation....862
21.2.1.2. Les principes de mise en oeuvre....863
21.2.1.3. Un exemple de mise en oeuvre....863
21.2.2. La gestion des logs avec les versions antérieures à la 1.2....864
21.2.2.1. Les niveaux de gravités : la classe Priority....864
21.2.2.2. La classe Category....865
21.2.2.3. La hiérarchie dans les catégories....866
21.2.3. La gestion des logs à partir de la version 1.2....866
21.2.3.1. Les niveaux de gravité : la classe Level....867
21.2.3.2. La classe Logger....867
21.2.3.3. La migration de Log4j antérieure à 1.2 vers 1.2....869
21.2.4. Les Appender....869
21.2.4.1. AsyncAppender....871
21.2.4.2. JDBCAppender....872
21.2.4.3. JMSAppender....872
21.2.4.4. LF5Appender....872
21.2.4.5. NTEventLogAppender....873
21.2.4.6. NullAppender....873
21.2.4.7. SMTPAppender....873
21.2.4.8. SocketAppender....873
21.2.4.9. SocketHubAppender....875
21.2.4.10. SyslogAppender....875
21.2.4.11. TelnetAppender....875
21.2.4.12. WriterAppender....875
21.2.4.13. ConsoleAppender....876
21.2.4.14. FileAppender....876
21.2.4.15. DailyRollingFileAppender....877
21.2.4.16. RollingFileAppender....878
21.2.4.17. ExternalyRolledFileAppender....879
21.2.5. Les layouts....880
21.2.5.1. SimpleLayout....880
21.2.5.2. HTMLLayout....881
21.2.5.3. XMLLayout....882
21.2.5.4. PatternLayout....882
21.2.6. L'externalisation de la configuration....884

Table des matières

21. Le logging

21.2.6.1. Les principes généraux....885
21.2.6.2. Le chargement explicite d'une configuration....886
21.2.6.3. Les formats des fichiers de configuration....888
21.2.6.4. La configuration par fichier properties....889
21.2.6.5. La configuration par un fichier XML....891
21.2.6.6. log4j.xml versus log4j.properties....897
21.2.6.7. La conversion du format properties en format XML....897
21.2.7. La mise en oeuvre avancée....898
21.2.7.1. La lecture des logs....898
21.2.7.2. Les variables d'environnement....900
21.2.7.3. L'internationalisation des messages....900
21.2.7.4. L'initialisation de Log4j dans une webapp....901
21.2.7.5. La modification dynamique de la configuration....901
21.2.7.6. NDC/MDC....901
21.2.8. Des best practices....904
21.2.8.1. Le choix du niveau de gravité des messages....904
21.2.8.2. L'amélioration des performances....904
21.2.8.3. D'autres recommandations....905
21.3. L'API logging....905
21.3.1. La classe LogManager....906
21.3.2. La classe Logger....907
21.3.3. La classe Level....908
21.3.4. La classe LogRecord....909
21.3.5. La classe Handler....910
21.3.5.1. La classe FileHandler....911
21.3.6. L'interface Filter....912
21.3.7. La classe Formatter....913
21.3.8. Le fichier de configuration....913
21.4. Jakarta Commons Logging (JCL)....914
21.5. D'autres API de logging....914

22. L'API Stream....916
22.1. Le besoin de l'API Stream....918
22.1.1. L'API Stream....920
22.1.2. Le rôle d'un Stream....921
22.1.3. Les concepts mis en oeuvre par les Streams....923
22.1.3.1. Le mode de fonctionnement d'un Stream....923
22.1.3.2. Les opérations pour définir les traitements d'un Stream....924
22.1.4. La différence entre une collection et un Stream....926
22.1.5. L'obtention d'un Stream....927
22.1.5.1. La création d'un Stream à partir de ses fabriques....928
22.1.5.2. L'obtention d'un Stream à partir d'une collection....929
22.2. Le pipeline d'opérations d'un Stream....930
22.3. Les opérations intermédiaires....933
22.3.1. Les méthodes map(), mapToInt(), mapToLong et mapToDouble()....934
22.3.2. La méthode flatMap()....937
22.3.3. La méthode filter()....939
22.3.4. La méthode distinct()....940
22.3.5. La méthode limit()....943
22.3.6. La méthode skip()....943
22.3.7. La méthode sorted()....944
22.3.8. La méthode peek()....945
22.3.9. Les méthodes qui modifient le comportement du Stream....945
22.4. Les opérations terminales....947
22.4.1. Les méthodes forEach() et forEachOrdered()....950
22.4.2. La méthode collect()....954
22.4.3. Les méthodes findFirst() et findAny(....958
22.4.4. Les méthodes xxxMatch()....959
22.4.5. La méthode count()....960

Table des matières

22. L'API Stream

22.4.6. La méthode reduce....960
22.4.7. Les méthodes min() et max()....966
22.4.8. La méthode toArray()....967
22.4.9. La méthode iterator....968
22.5. Les Collectors....969
22.5.1. L'interface Collector....969
22.5.2. La classe Collectors....971
22.5.2.1. Les fabriques pour des Collector vers des collections....974
22.5.2.2. La fabrique pour des Collector qui exécutent une action complémentaire....977
22.5.2.3. Les fabriques qui renvoient des Collector pour réaliser une agrégation....977
22.5.2.4. Les fabriques qui renvoient des Collectors pour effectuer des opérations numériques....979
22.5.2.5. Les fabriques qui renvoient des Collectors pour effectuer des groupements....981
22.5.2.6. Les fabriques qui renvoient des Collectors pour effectuer des transformations....982
22.5.3. La composition de Collectors....983
22.5.4. L'implémentation d'un Collector....985
22.5.5. Les fabriques of() pour créer des instances de Collector....985
22.6. Les Streams pour des données primitives....986
22.6.1. Les interfaces IntStream, LongStream et DoubleStream....988
22.7. L'utilisation des Streams avec les opérations I/O....992
22.7.0.1. La création d'un Stream à partir d'un fichier texte....992
22.7.0.2. La création d'un Stream à partir du contenu d'un répertoire....993
22.8. Le traitement des opérations en parallèle....995
22.8.1. La mise en oeuvre des Streams parallèles....995
22.8.2. Le fonctionnement interne d'une Stream....996
22.8.2.1. L'interface Spliterator pour l'obtention des données par la source....996
22.8.2.2. L'utilisation du framework Fork/Join....998
22.8.3. Le mode d'exécution des Streams....1000
22.8.4. Le comportement de certaines opérations en parallèle....1002
22.8.5. L'impact de l'état ordonné ou non des éléments....1003
22.8.6. Les performances des traitements en parallèle....1004
22.8.7. Des recommandations pour les Streams parallèles....1005
22.8.7.1. L'utilisation de traitements stateless....1005
22.8.7.2. Les effets de bord....1006
22.9. Les optimisations réalisées par l'API Stream....1006
22.9.1. Les optimisations liées aux opérations de type short circuiting....1007
22.9.2. L'ordre des opérations d'un Stream....1008
22.10. Les Streams infinis....1011
22.11. Le débogage d'un Stream....1013
22.12. Les limitations de l'API Stream....1014
22.12.1. Un Stream n'est pas réutilisable....1014
22.13. Quelques recommandations sur l'utilisation de l'API Stream....1015
22.13.1. L'utilisation à mauvais escient de l'API Stream....1015
22.13.1.1. Le parcours des éléments....1016
22.13.1.2. Le remplacement des boucles for par un Stream....1016
22.13.1.3. La conversion d'une collection....1018
22.13.1.4. La recherche du plus grand élément d'une collection....1019
22.13.1.5. La détermination du nombre d'éléments d'une collection....1020
22.13.1.6. La vérification de la présence d'un élément dans une collection....1020
22.13.2. L'utilisation de l'API Stream pour rechercher des éléments....1021
22.13.2.1. La recherche de la présence d'un élément....1021
22.13.2.2. La recherche du plus petit élément....1022
22.13.2.3. La recherche du plus grand élément....1022
22.13.3. La création de Streams....1023
22.13.3.1. La création d'un Stream vide....1023
22.13.3.2. La création d'un Stream avec un seul élément....1024
22.13.3.3. La création d'un Stream à partir d'un tableau....1024
22.13.3.4. Le traitement d'une plage d'éléments d'un tableau....1025
22.13.4. L'utilisation non requise d'un Collector....1025
22.13.5. Le traitement de valeurs numériques....1026

Table des matières

22. L'API Stream

22.13.6. Ne compter les éléments que si c'est nécessaire....1027
22.13.6.1. La détermination du nombre d'éléments de sous-ensembles....1027
22.13.6.2. La vérification qu'au moins un élément satisfasse une condition....1028
22.13.6.3. La vérification qu'aucun élément ne satisfasse une condition....1029
22.13.6.4. La vérification qu'au moins N éléments satisfassent une condition....1030

23. Les expressions régulières....1031
23.1. Le package java.util.regex....1032
23.1.1. La classe Pattern....1032
23.1.1.1. L'obtention d'une instance de type Pattern....1033
23.1.1.2. L'obtention d'une instance de type Matcher....1034
23.1.1.3. La méthode matches()....1034
23.1.1.4. Le découpage d'une chaîne selon un motif....1036
23.1.2. La classe Matcher....1037
23.1.2.1. Les méthodes pour obtenir des index....1037
23.1.2.2. Les méthodes pour la recherche de correspondance....1038
23.1.2.3. Les méthodes concernant les groupes....1041
23.1.2.4. Les méthodes de remplacement....1041
23.1.3. La classe PatternSyntaxException....1042
23.2. La mise en oeuvre des expressions régulières....1043
23.2.1. L'échappement de caractères....1045
23.2.2. Les terminaisons de ligne (Line terminators)....1047
23.2.3. Une simple chaîne littérale....1047
23.2.4. Les métacaractères (Meta Characters)....1049
23.2.5. Les classes de caractères (Character Classes)....1049
23.2.5.1. Les ensembles de caractères....1049
23.2.5.2. L'opérateur OR....1052
23.2.5.3. L'opérateur NOR....1053
23.2.5.4. Une plage....1053
23.2.5.5. L'opérateur union....1055
23.2.5.6. L'opérateur intersection....1056
23.2.5.7. La soustraction de classe de caractères (Subtraction Class)....1056
23.2.5.8. Les classes de caractères prédéfinies (Predefined Character Classes)....1057
23.2.5.9. Les classes de caractères POSIX....1061
23.2.5.10. Les classes de java.lang.Character....1062
23.2.5.11. Les classes pour Unicode....1063
23.2.6. Les quantificateurs (Quantifiers)....1065
23.2.7. Les groupes de capture (capturing groups)....1072
23.2.7.1. La syntaxe de définition de groupes....1072
23.2.7.2. L'utilisation des méthodes de la classe Matcher pour les groupes....1072
23.2.7.3. La numérotation des groupes....1074
23.2.7.4. Le nommage d'un groupe....1075
23.2.7.5. Les références arrières (back reference)....1077
23.2.7.6. Les groupes non capturants....1079
23.2.7.7. Les groupes atomiques....1080
23.2.8. Les assertions lookaround....1082
23.2.8.1. Les assertions Lookahead....1083
23.2.8.2. Les assertions lookbehind....1089
23.2.9. Les limites de correspondance (Boundary Matchers)....1095
23.2.10. Les modes utilisables....1100
23.2.11. Le support d'Unicode....1109
23.3. Les remplacements de texte....1110
23.3.1. Les remplacements avec la classe Matcher....1110
23.3.1.1. Les méthodes replaceFirst() et replaceAll()....1110
23.3.1.2. Les méthodes appendReplacement() et appendTail()....1112
23.4. L'utilisation d'expressions régulières dans les méthodes de la classe String....1113

Table des matières

Partie 3 : Les API avancées.....**1116**

24. La gestion dynamique des objets et l'introspection.....**1117**
24.1. La classe Class.....1118
24.1.1. L'obtention d'un objet de type Class.....1118
24.1.1.1. La détermination de la classe d'un objet.....1118
24.1.1.2. L'obtention d'un objet Class à partir d'un nom de classe.....1118
24.1.1.3. Une troisième façon d'obtenir un objet Class.....1119
24.1.2. Les méthodes de la classe Class.....1119
24.2. La recherche des informations sur une classe.....1120
24.2.1. La recherche de la classe mère d'une classe.....1120
24.2.2. La recherche des modificateurs d'une classe.....1120
24.2.3. La recherche des interfaces implémentées par une classe.....1121
24.2.4. La recherche des champs publics.....1121
24.2.5. La recherche des paramètres d'une méthode ou d'un constructeur.....1122
24.2.6. La recherche des constructeurs de la classe.....1123
24.2.7. La recherche des méthodes publiques.....1124
24.2.8. La recherche de toutes les méthodes.....1124
24.2.9. La recherche des getters et des setters.....1125
24.2.10. Le support des types scellés.....1126
24.3. La définition dynamique d'objets.....1126
24.3.1. La création d'objets grâce à la classe Class.....1126
24.3.2. La création d'objets grâce à la classe Constructor.....1128
24.4. L'invocation dynamique d'une méthode.....1129
24.4.1. La passage de paramètre à la méthode invoquée.....1131
24.4.2. La gestion d'une exception levée par la méthode invoquée.....1133
24.4.3. L'invocation d'une méthode statique.....1134
24.4.4. L'accès aux méthodes privées.....1134
24.4.5. L'invocation dynamique d'une méthode avec type generic.....1135
24.5. L'API Reflection et le SecurityManager.....1137
24.6. L'utilisation de l'API Reflection sur les annotations.....1139
24.6.1. Les annotations sur une classe.....1139
24.6.2. Les annotations sur une méthode.....1140
24.6.3. Les annotations sur un paramètre d'une méthode.....1141
24.6.4. Les annotations sur un champ.....1143

25. L'appel de méthodes distantes : RMI.....**1145**
25.1. La présentation et l'architecture de RMI.....1145
25.2. Les différentes étapes pour créer un objet distant et l'appeler avec RMI.....1145
25.3. Le développement coté serveur.....1146
25.3.1. La définition d'une interface qui contient les méthodes de l'objet distant.....1146
25.3.2. L'écriture d'une classe qui implémente cette interface.....1146
25.3.3. L'écriture d'une classe pour instancier l'objet et l'enregistrer dans le registre.....1147
25.3.3.1. La mise en place d'un security manager.....1147
25.3.3.2. L'instanciation d'un objet de la classe distante.....1148
25.3.3.3. L'enregistrement dans le registre de noms RMI.....1148
25.3.3.4. Le lancement dynamique du registre de noms RMI.....1148
25.4. Le développement coté client.....1149
25.4.1. La mise en place d'un security manager.....1149
25.4.2. L'obtention d'une référence sur l'objet distant à partir de son nom.....1150
25.4.3. L'appel de la méthode à partir de la référence sur l'objet distant.....1150
25.4.4. L'appel d'une méthode distante dans une applet.....1151
25.5. La génération de la classe stub.....1151
25.6. La mise en oeuvre des objets RMI.....1152
25.6.1. Le lancement du registre RMI.....1152
25.6.2. L'instanciation et l'enregistrement de l'objet distant.....1152
25.6.3. Le lancement de l'application cliente.....1152

Table des matières

26. La sécurité 1155

26.1. La sécurité dans les spécifications du langage 1155

26.1.1. Les contrôles lors de la compilation 1156

26.1.2. Les contrôles lors de l'exécution 1156

26.2. Le contrôle des droits d'une application 1156

26.2.1. Le modèle de sécurité de Java 1.0 1156

26.2.2. Le modèle de sécurité de Java 1.1 1156

26.2.3. Le modèle Java 1.2 1157

26.3. La cryptographie 1157

26.3.1. Le modèle symétrique 1159

26.3.2. Le modèle asymétrique 1160

26.4. JCA (Java Cryptography Architecture) et JCE (Java Cryptography Extension) 1162

26.5. JSSE (Java Secure Sockets Extension) 1163

26.6. JAAS (Java Authentication and Authorization Service) 1163

27. JCA (Java Cryptography Architecture) 1164

27.1. L'architecture de JCA 1165

27.2. Les classes et interfaces de JCA 1165

27.3. Les fournisseurs d'implémentations 1166

27.4. La classe java.security.Provider 1167

27.5. La classe java.security.Security 1168

27.6. La classe java.security.MessageDigest 1170

27.7. Les classes DigestInputStream et DigestOuputStream 1172

27.8. La classe java.security.Signature 1174

27.9. La classe java.security.KeyStore 1176

27.10. Les interfaces de type java.security.Key 1178

27.11. La classe java.security.KeyPair 1179

27.12. La classe java.security.KeyPairGenerator 1179

27.13. La classe java.security.KeyFactory 1181

27.14. La classe java.security.SecureRandom 1183

27.15. La classe java.security.AlgorithmParameters 1185

27.16. La classe java.security.AlgorithmParameterGenerator 1186

27.17. La classe java.security.cert.CertificateFactory 1187

27.18. L'interface java.security.spec.KeySpec et ses implémentations 1189

27.19. La classe java.security.spec.EncodedKeySpec et ses sous-classes 1190

27.19.1. La classe java.security.spec.PKCS8EncodedKeySpec 1191

27.20. L'interface java.security.spec.AlgorithmParameterSpec 1192

27.20.1. La classe java.security.spec.DSAParameterSpec 1192

28. JCE (Java Cryptography Extension) 1193

28.1. La classe javax.crypto.KeyGenerator 1195

28.2. La classe javax.crypto.SecretKeyFactory 1198

28.3. La classe javax.crypto.Cipher 1199

28.3.1. La création d'une instance de type Cipher 1200

28.3.2. L'initialisation de l'instance de type Cipher 1201

28.3.3. Le chiffrement et le déchiffrement de données 1202

28.3.4. Les modes wrap et unwrap 1204

28.3.5. Les paramètres des algorithmes de chiffrement 1205

28.3.6. Des exemples d'utilisation d'algorithmes de chiffrement 1206

28.4. Les classes javax.crypto.CipherInputStream et javax.crypto.CipherOutputStream 1208

28.5. La classe javax.crypto.SealedObject 1210

28.6. La classe javax.crypto.Mac 1211

29. JNI (Java Native Interface) 1216

29.1. La déclaration et l'utilisation d'une méthode native 1216

29.2. La génération du fichier d'en-tête 1217

29.3. L'écriture du code natif en C 1219

29.4. Le passage de paramètres et le renvoi d'une valeur (type primitif) 1220

29.5. Le passage de paramètres et le renvoi d'une valeur (type objet) 1221

Table des matières

30. JNDI (Java Naming and Directory Interface)....1224
- 30.1. La présentation de JNDI....1225
  - 30.1.1. Les services de nommage....1226
  - 30.1.2. Les annuaires....1226
  - 30.1.3. Le contexte....1226
- 30.2. La mise en oeuvre de l'API JNDI....1227
  - 30.2.1. L'interface Name....1227
  - 30.2.2. L'interface Context et la classe IntitialContext....1227
- 30.3. L'utilisation d'un service de nommage....1228
  - 30.3.1. L'obtention d'un objet....1229
  - 30.3.2. Le stockage d'un objet....1229
- 30.4. L'utilisation avec un DNS....1229
- 30.5. L'utilisation du File System Context Provider....1230
- 30.6. LDAP....1231
  - 30.6.1. L'outil OpenLDAP....1233
  - 30.6.2. LDAPBrowser....1234
  - 30.6.3. LDIF....1237
- 30.7. L'utilisation avec un annuaire LDAP....1237
  - 30.7.1. L'interface DirContext....1237
  - 30.7.2. La classe InitialDirContext....1238
  - 30.7.3. Les attributs....1239
  - 30.7.4. L'utilisation d'objets Java....1240
  - 30.7.5. Le stockage d'objets Java....1240
  - 30.7.6. L'obtention d'un objet Java....1242
  - 30.7.7. La modification d'un objet....1243
  - 30.7.8. La suppression d'un objet....1248
  - 30.7.9. La recherche d'associations....1249
  - 30.7.10. La recherche dans un annuaire LDAP....1250
- 30.8. JNDI et J2EE/Java EE....1254

31. Le scripting....1256
- 31.1. L'API Scripting....1256
  - 31.1.1. La mise en oeuvre de l'API....1257
  - 31.1.2. Ajouter d'autres moteurs de scripting....1258
  - 31.1.3. L'évaluation d'un script....1259
  - 31.1.4. L'interface Compilable....1262
  - 31.1.5. L'interface Invocable....1263
  - 31.1.6. La commande jrunscript....1264

32. JMX (Java Management Extensions)....1266
- 32.1. La présentation de JMX....1267
- 32.2. L'architecture de JMX....1268
- 32.3. Un premier exemple....1270
  - 32.3.1. La définition de l'interface et des classes du MBean....1270
  - 32.3.2. L'exécution de l'application....1271
- 32.4. La couche instrumentation : les MBeans....1274
  - 32.4.1. Les MBeans....1274
  - 32.4.2. Les différents types de MBeans....1275
  - 32.4.3. Les MBeans dans l'architecture JMX....1276
  - 32.4.4. Le nom des MBeans....1276
  - 32.4.5. Les types de données dans les MBeans....1277
    - 32.4.5.1. Les types de données complexes....1278
- 32.5. Les MBeans standard....1278
  - 32.5.1. La définition de l'interface d'un MBean standard....1278
  - 32.5.2. L'implémentation du MBean Standard....1280
  - 32.5.3. L'utilisation d'un MBean....1280
- 32.6. La couche agent....1281
  - 32.6.1. Le rôle d'un agent JMX....1281
  - 32.6.2. Le serveur de MBeans (MBean Server)....1282
  - 32.6.3. Le Mbean de type MBeanServerDelegate....1282

Table des matières

32. JMX (Java Management Extensions)

32.6.3.1. L'enregistrement d'un MBean dans le serveur de MBeans....1282
32.6.3.2. L'interface MBeanRegistration....1283
32.6.3.3. La suppression d'un MBean du serveur de MBeans....1283
32.6.4. La communication avec la couche agent....1283
32.6.5. Le développement d'un agent JMX....1284
32.6.5.1. L'instanciation d'un serveur de MBeans....1284
32.6.5.2. L'instanciation et l'enregistrement d'un MBean dans le serveur....1285
32.6.5.3. L'ajout d'un connecteur ou d'un adaptateur de protocoles....1286
32.6.5.4. L'utilisation d'un service de l'agent....1286
32.7. Les services d'un agent JMX....1286
32.7.1. Le service de type M-Let....1286
32.7.1.1. Le format du fichier de définitions....1287
32.7.1.2. L'instanciation et l'utilisation d'un service M-Let dans un agent....1288
32.7.1.3. Un exemple de mise en oeuvre du service M-Let....1288
32.7.2. Le service de type Timer....1290
32.7.2.1. Les fonctionnalités du service Timer....1291
32.7.2.2. L'ajout d'une définition de notifications....1291
32.7.2.3. Un exemple de mise en oeuvre du service Timer....1292
32.7.3. Le service de type Monitor....1293
32.7.4. Le service de type Relation....1293
32.8. La couche services distribués....1293
32.8.1. L'interface MBeanServerConnection....1294
32.8.2. Les connecteurs et les adaptateurs de protocoles....1294
32.8.2.1. Les connecteurs....1295
32.8.2.2. Les adaptateurs de protocoles....1295
32.8.3. L'utilisation du connecteur RMI....1295
32.8.4. L'utilisation du connecteur utilisant le protocole JMXMP....1299
32.8.5. L'utilisation de l'adaptateur de protocole HTML....1302
32.8.6. L'invocation d'un MBean par un proxy....1306
32.8.7. La recherche et la découverte des agents JMX....1308
32.8.7.1. Par le Service Location Protocol (SLP)....1308
32.8.7.2. Par la technologie Jini....1308
32.8.7.3. Par un annuaire et la technologie JNDI....1308
32.9. Les notifications....1309
32.9.1. L'interface NotificationBroadcaster....1309
32.9.2. L'interface NotificationEmitter....1309
32.9.3. La classe NotificationBroadcasterSupport....1309
32.9.4. La classe javax.management.Notification....1309
32.9.5. Un exemple de notifications....1310
32.9.6. L'abonnement aux notifications par un client JMX....1312
32.10. Les Dynamic MBeans....1316
32.10.1. L'interface DynamicMBean....1316
32.10.2. Les métadonnées d'un Dynamic MBean....1317
32.10.2.1. La classe MBeanInfo....1317
32.10.2.2. La classe MBeanFeatureInfo....1318
32.10.2.3. La classe MBeanAttributeInfo....1318
32.10.2.4. La classe MBeanParameterInfo....1318
32.10.2.5. La classe MBeanConstructorInfo....1319
32.10.2.6. La classe MBeanOperationInfo....1319
32.10.2.7. La classe MBeanNotificationInfo....1320
32.10.3. La définition d'un MBean Dynamic....1320
32.10.4. La classe StandardMBean....1324
32.11. Les Model MBeans....1325
32.11.1. L'interface ModelMBean et la classe RequiredModelMBean....1325
32.11.2. La description des fonctionnalités exposées....1326
32.11.3. Un exemple de mise en oeuvre....1327
32.11.4. Les fonctionnalités optionnelles des Model MBeans....1330
32.11.5. Les différences entre un Dynamic MBean et un Model MBean....1330
32.12. Les Open MBeans....1331

Table des matières

32. JMX (Java Management Extensions)
- 32.12.1. La mise en oeuvre d'un Open MBean.....1331
- 32.12.2. Les types de données utilisables dans les Open MBeans.....1332
- 32.12.2.1. Les Open Types.....1332
- 32.12.2.2. La classe CompositeType et l'interface CompositeData.....1332
- 32.12.2.3. La classe TabularType et l'interface TabularData.....1333
- 32.12.3. Un exemple d'utilisation d'un Open MBean.....1333
- 32.12.4. Les avantages et les inconvénients des Open MBeans.....1334
- 32.13. Les MXBeans.....1334
- 32.13.1. La définition d'un MXBean.....1334
- 32.13.2. L'écriture d'un type personnalisé utilisé par le MXBean.....1335
- 32.13.3. La mise en oeuvre d'un MXBean.....1336
- 32.14. L'interface PersistentMBean.....1337
- 32.15. Le monitoring d'une JVM.....1338
- 32.15.1. L'interface ClassLoadingMXBean.....1339
- 32.15.2. L'interface CompilationMXBean.....1339
- 32.15.3. L'interface GarbageCollectorMXBean.....1340
- 32.15.4. L'interface MemoryManagerMXBean.....1341
- 32.15.5. L'interface MemoryMXBean.....1342
- 32.15.6. L'interface MemoryPoolMXBean.....1344
- 32.15.7. L'interface OperatingSystemMXBean.....1345
- 32.15.8. L'interface RuntimeMXBean.....1346
- 32.15.9. L'interface ThreadMXBean.....1347
- 32.15.10. La sécurisation des accès à l'agent.....1349
- 32.16. Des recommandations pour l'utilisation de JMX.....1349
- 32.17. Des ressources.....1349

33. L'API Service Provider (SPI).....1351
- 33.1. Introduction.....1351
- 33.1.1. La différence entre une API et une SPI.....1351
- 33.1.2. Un framework pour service provider.....1352
- 33.1.2.1. Un exemple d'implémentation basique.....1352
- 33.1.3. Le chargement dynamique de classes.....1354
- 33.1.4. L'utilisation par Java SE et Java EE.....1355
- 33.2. La mise en oeuvre.....1356
- 33.2.1. La définition d'un service.....1356
- 33.2.2. L'implémentation d'un service.....1357
- 33.2.3. Le fichier de configuration du Provider.....1357
- 33.3. La consommation d'un service.....1358
- 33.3.1. Le déploiement d'un service dans le classpath.....1358
- 33.3.2. La classe ServiceLoader.....1358
- 33.3.2.1. L'obtention d'un ServiceLoader.....1359
- 33.3.2.2. La recherche des implémentations par le ServiceLoader.....1360
- 33.3.2.3. L'obtention des implémentations par le ServiceLoader.....1360
- 33.3.2.4. L'utilisation d'un cache par le ServiceLoader.....1362
- 33.3.3. Les exceptions lors de l'utilisation de la classe ServiceLoader.....1362
- 33.3.4. Les limitations de la classe ServiceLoader.....1363
- 33.4. Un exemple complet.....1363
- 33.5. Les services de JPMS.....1366
- 33.5.1. Le module qui contient l'interface du service.....1366
- 33.5.2. Un fournisseur de service dans un module.....1366
- 33.5.2.1. L'implémentation du service dans le module.....1367
- 33.5.2.2. La déclaration dans le descripteur de module.....1367
- 33.5.3. La consommation de services d'un module.....1368
- 33.5.3.1. La déclaration dans le descripteur de module.....1369
- 33.5.3.2. L'utilisation de la classe ServiceLoader.....1369
- 33.5.3.3. Faciliter la consommation d'un service.....1369
- 33.5.4. La résolution de services.....1370
- 33.5.5. Les options de jlink pour résoudre les services.....1370

Table des matières

Partie 4 : Le système de modules....1372

34. Le système de modules de la plateforme Java....1373
34.1. La modularisation....1374
34.2. Les difficultés pouvant être résolues par les modules....1375
34.2.1. La taille croissante des API du JRE....1375
34.2.2. Les limitations du format jar....1375
34.2.3. Les problématiques liées au classpath....1376
34.2.3.1. JAR/Classpath Hell....1376
34.2.3.2. Un classpath pour le compilateur et un autre pour la JVM....1377
34.2.3.3. Les collisions de versions....1377
34.2.3.4. Le temps de démarrage d'une JVM....1377
34.2.4. L'impossibilité de définir les dépendances....1378
34.2.5. Pas d'encapsulation dans un jar ou entre les jars....1378
34.2.6. La sécurité....1379
34.3. Java Plateform Module System (JPMS)....1379
34.3.1. Les buts de JPMS....1379
34.3.2. Une configuration plus fiable (reliable configuration)....1380
34.3.3. L'encapsulation forte (strong encapsulation)....1380
34.3.4. L'évolutivité de la plateforme....1380
34.4. L'implémentation du système de modules....1381
34.4.1. Le projet Jigsaw....1381
34.4.2. Les JEP utilisées pour intégrer le système de modules....1381
34.4.3. La modularisation du JDK....1382
34.4.4. Les modules dépréciés de Java 9....1383
34.4.5. Les modules du JDK....1383

35. Les modules....1387
35.1. Le contenu d'un module....1388
35.2. Le code source d'un module....1388
35.3. Le descripteur de module....1389
35.3.1. Le nommage d'un module....1390
35.3.2. Le descripteur de module : le fichier module-info.java....1391
35.3.3. L'export des packages....1393
35.3.4. La déclaration des dépendances....1394
35.3.4.1. Les dépendances explicites....1395
35.3.4.2. Les dépendances transitives....1396
35.3.4.3. Les patterns utilisables avec la lisibilité implicite....1399
35.3.4.4. Récapitulatif de la lisibilité....1400
35.3.5. Les dépendances optionnelles....1401
35.3.6. L'utilisation de l'API Reflection et l'accès aux ressources....1401
35.4. Les règles d'accès....1403
35.5. La qualité des descripteurs de module....1404

Partie 5 : La programmation parallèle et concurrente....1406

36. Le multitâche....1407

37. Les threads....1408
37.1. L'interface Runnable....1409
37.2. La classe Thread....1409
37.3. Le cycle de vie d'un thread....1411
37.3.1. La création d'un thread....1412
37.3.2. L'arrêt d'un thread....1414
37.4. Les démons (daemon threads)....1415
37.5. Les groupes de threads....1415
37.6. L'obtention d'informations sur un thread....1421
37.6.1. L'état d'un thread....1421
37.6.2. L'obtention du thread courant....1421
37.7. La manipulation des threads....1422

Table des matières

37. Les threads

37.7.1. La mise en sommeil d'un thread pour une certaine durée....1422
37.7.2. L'attente de la fin de l'exécution d'un thread....1423
37.7.3. La modification de la priorité d'un thread....1424
37.7.4. Laisser aux autres threads plus de chance de s'exécuter....1425
37.7.5. L'interruption d'un thread....1426
37.7.6. L'exception InterruptedException....1430
37.8. Les messages de synchronisation entre threads....1431
37.9. Les restrictions sur les threads....1433
37.10. Les threads et les classloaders....1434
37.11. Les threads et la gestion des exceptions....1435
37.11.1. L'exception ThreadDeath....1437
37.12. Les piles....1438
37.12.1. Les threads et la mémoire....1439
37.12.2. L'obtention d'informations sur la pile....1441
37.12.3. L'escape analysis....1441
37.12.4. Les restrictions d'accès sur les threads et les groupes de threads....1447

38. L'association de données à des threads....1451

38.1. La classe ThreadLocal....1452
38.1.1. L'utilisation de la classe ThreadLocal....1452
38.1.2. Le fonctionnement interne d'un ThreadLocal....1456
38.1.3. ThreadLocal et fuite de mémoire....1458
38.1.3.1. Les fuites de mémoires dans un serveur d'applications....1458
38.1.3.2. Les fuites de mémoires dans un Executor....1459
38.1.4. Bonnes pratiques pour l'utilisation d'un ThreadLocal....1461
38.2. La classe InheritableThreadLocal....1463
38.3. La classe ThreadLocalRandom....1465

39. Le framework Executor....1467

39.1. L'interface Executor....1467
39.1.1. L'interface ExecutorService....1468
39.1.2. L'interface ScheduledExecutorService....1469
39.2. Les pools de threads....1469
39.2.1. La classe ThreadPoolExecutor....1470
39.2.2. La classe Executors....1477
39.3. L'interface java.util.concurrent.Callable....1479
39.4. L'interface java.util.concurrent.Future....1480
39.5. L'interface java.util.concurrent.CompletionService....1484

40. La gestion des accès concurrents....1488

40.1. Le mot clé volatile....1489
40.2. Les races conditions....1491
40.2.1. La détection des race conditions....1494
40.3. La synchronisation avec les verrous....1495
40.3.1. Les verrous avec des moniteurs....1495
40.3.1.0.1. Le mot clé synchronized....1495
40.3.1.1. Les moniteurs....1496
40.3.1.2. Les contraintes d'utilisation....1498
40.3.1.3. Des recommandations d'utilisation....1498
40.3.1.4. Les avantages et les inconvénients....1500
40.3.2. Les classes Lock et Condition....1500
40.3.2.1. L'interface Lock....1500
40.3.2.2. La classe ReentrantLock....1502
40.3.2.3. L'interface Condition....1506
40.3.2.4. L'interface ReadWriteLock et la classe ReentrantReadWriteLock....1508
40.4. Les opérations atomiques....1510
40.4.1. La classe AtomicBoolean....1514
40.4.2. La classe AtomicInteger....1515
40.4.3. La classe AtomicLong....1517

Table des matières

40. La gestion des accès concurrents
40.4.4. Les classes AtomicIntegerArray, AtomicLongArray et AtomicReferenceArray<E>....1519
40.4.4.1. La classe AtomicIntegerArray....1519
40.4.4.2. La classe AtomicLongArray....1520
40.4.4.3. La classe AtomicReferenceArray<E>....1521
40.4.5. Les classes AtomicReference, AtomicMarkableReference et AtomicStampedReference....1522
40.4.5.1. La classe AtomicReference<V>....1522
40.4.5.2. La classe AtomicMarkableReference....1523
40.4.5.3. La classe AtomicStampedReference....1524
40.4.6. Les classes AtomicIntegerFieldUpdater, AtomicLongFieldUpdater, AtomicReferenceFieldUpdater....1526
40.4.7. Les classes DoubleAccumulator et LongAccumulator....1526
40.4.8. Les classes DoubleAdder et LongAdder....1526
40.5. L'immutabilité et la copie défensive....1526
40.5.1. L'immutabilité....1527
40.5.2. La copie défensive....1527

Partie 6 : Le développement des interfaces graphiques....1533

41. Le graphisme....1534
41.1. Les opérations sur le contexte graphique....1534
41.1.1. Le tracé de formes géométriques....1534
41.1.2. Le tracé de texte....1535
41.1.3. L'utilisation des fontes....1535
41.1.4. La gestion de la couleur....1536
41.1.5. Le chevauchement de figures graphiques....1536
41.1.6. L'effacement d'une aire....1536
41.1.7. La copie d'une aire rectangulaire....1536

42. Les éléments d'interfaces graphiques de l'AWT....1537
42.1. Les composants graphiques....1538
42.1.1. Les étiquettes....1538
42.1.2. Les boutons....1539
42.1.3. Les panneaux....1539
42.1.4. Les listes déroulantes (combobox)....1539
42.1.5. La classe TextComponent....1541
42.1.6. Les champs de texte....1541
42.1.7. Les zones de texte multilignes....1542
42.1.8. Les listes....1544
42.1.9. Les cases à cocher....1547
42.1.10. Les boutons radio....1548
42.1.11. Les barres de défilement....1548
42.1.12. La classe Canvas....1550
42.2. La classe Component....1550
42.3. Les conteneurs....1552
42.3.1. Le conteneur Panel....1552
42.3.2. Le conteneur Window....1553
42.3.3. Le conteneur Frame....1553
42.3.4. Le conteneur Dialog....1554
42.4. Les menus....1555
42.4.1. Les méthodes de la classe MenuBar....1557
42.4.2. Les méthodes de la classe Menu....1557
42.4.3. Les méthodes de la classe MenuItem....1558
42.4.4. Les méthodes de la classe CheckboxMenuItem....1558
42.5. La classe java.awt.Desktop....1558

43. La création d'interfaces graphiques avec AWT....1560
43.1. Le dimensionnement des composants....1560
43.2. Le positionnement des composants....1561
43.2.1. La mise en page par flot (FlowLayout)....1562

Table des matières

43. La création d'interfaces graphiques avec AWT

43.2.2. La mise en page bordure (BorderLayout)....1563
43.2.3. La mise en page de type carte (CardLayout)....1564
43.2.4. La mise en page GridLayout....1565
43.2.5. La mise en page GridBagLayout....1567
43.3. La création de nouveaux composants à partir de Panel....1568
43.4. L'activation ou la désactivation des composants....1569

44. L'interception des actions de l'utilisateur....1570
44.1. L'interception des actions de l'utilisateur avec Java version 1.0....1570
44.2. L'interception des actions de l'utilisateur avec Java version 1.1....1570
44.2.1. L'interface ItemListener....1572
44.2.2. L'interface TextListener....1573
44.2.3. L'interface MouseMotionListener....1574
44.2.4. L'interface MouseListener....1574
44.2.5. L'interface WindowListener....1575
44.2.6. Les différentes implémentations des Listeners....1576
44.2.6.1. Une classe implémentant elle même le listener....1576
44.2.6.2. Une classe indépendante implémentant le listener....1577
44.2.6.3. Une classe interne....1578
44.2.6.4. Une classe interne anonyme....1578
44.2.7. Résumé....1579

45. Le développement d'interfaces graphiques avec SWING....1580
45.1. La présentation de Swing....1580
45.2. Les packages Swing....1581
45.3. Un exemple de fenêtre autonome....1581
45.4. Les composants Swing....1582
45.4.1. La classe JFrame....1582
45.4.1.1. Le comportement par défaut à la fermeture....1585
45.4.1.2. La personnalisation de l'icône....1586
45.4.1.3. Centrer une JFrame à l'écran....1586
45.4.1.4. Les événements associées à un JFrame....1587
45.4.2. Les étiquettes : la classe JLabel....1587
45.4.3. Les panneaux : la classe JPanel....1590
45.5. Les boutons....1590
45.5.1. La classe AbstractButton....1590
45.5.2. La classe JButton....1592
45.5.3. La classe JToggleButton....1593
45.5.4. La classe ButtonGroup....1593
45.5.5. Les cases à cocher : la classe JCheckBox....1594
45.5.6. Les boutons radio : la classe JRadioButton....1595
45.6. Les composants de saisie de texte....1599
45.6.1. La classe JTextComponent....1599
45.6.2. La classe JTextField....1600
45.6.3. La classe JPasswordField....1600
45.6.4. La classe JFormattedTextField....1602
45.6.5. La classe JEditorPane....1602
45.6.6. La classe JTextPane....1603
45.6.7. La classe JTextArea....1603
45.7. Les onglets....1604
45.8. Le composant JTree....1606
45.8.1. La création d'une instance de la classe JTree....1606
45.8.2. La gestion des données de l'arbre....1609
45.8.2.1. L'interface TreeNode....1610
45.8.2.2. L'interface MutableTreeNode....1610
45.8.2.3. La classe DefaultMutableTreeNode....1611
45.8.3. La modification du contenu de l'arbre....1612
45.8.3.1. Les modifications des noeuds fils....1612
45.8.3.2. Les événements émis par le modèle....1613

Table des matières

45. Le développement d'interfaces graphiques avec SWING

45.8.3.3. L'édition d'un noeud....1614
45.8.3.4. Les éditeurs personnalisés....1614
45.8.3.5. La définition des noeuds éditables....1615
45.8.4. La mise en oeuvre d'actions sur l'arbre....1616
45.8.4.1. Etendre ou refermer un noeud....1616
45.8.4.2. La détermination du noeud sélectionné....1617
45.8.4.3. Le parcours des noeuds de l'arbre....1617
45.8.5. La gestion des événements....1618
45.8.5.1. La classe TreePath....1619
45.8.5.2. La gestion de la sélection d'un noeud....1620
45.8.5.3. Les événements lies à la sélection de noeuds....1621
45.8.5.4. Les événements lorsqu'un noeud est étendu ou refermé....1623
45.8.5.5. Le contrôle des actions pour étendre ou refermer un noeud....1624
45.8.6. La personnalisation du rendu....1624
45.8.6.1. Personnaliser le rendu des noeuds....1625
45.8.6.2. Les bulles d'aides (Tooltips)....1627
45.9. Les menus....1628
45.9.1. La classe JMenuBar....1631
45.9.2. La classe JMenuItem....1633
45.9.3. La classe JPopupMenu....1633
45.9.4. La classe JMenu....1635
45.9.5. La classe JCheckBoxMenuItem....1636
45.9.6. La classe JRadioButtonMenuItem....1636
45.9.7. La classe JSeparator....1636
45.10. L'affichage d'une image dans une application....1637

46. Le développement d'interfaces graphiques avec SWT....1642
46.1. La présentation de SWT....1642
46.2. Un exemple très simple....1644
46.3. La classe SWT....1644
46.4. L'objet Display....1645
46.5. L'objet Shell....1645
46.6. Les composants....1647
46.6.1. La classe Control....1647
46.6.2. Les contrôles de base....1647
46.6.2.1. La classe Button....1647
46.6.2.2. La classe Label....1648
46.6.2.3. La classe Text....1649
46.6.3. Les contrôles de type liste....1650
46.6.3.1. La classe Combo....1650
46.6.3.2. La classe List....1650
46.6.4. Les contrôles pour les menus....1651
46.6.4.1. La classe Menu....1651
46.6.4.2. La classe MenuItem....1651
46.6.5. Les contrôles de sélection ou d'affichage d'une valeur....1653
46.6.5.1. La classe ProgressBar....1653
46.6.5.2. La classe Scale....1653
46.6.5.3. La classe Slider....1654
46.6.6. Les contrôles de type « onglets »....1655
46.6.6.1. La classe TabFolder....1655
46.6.6.2. La classe TabItem....1655
46.6.7. Les contrôles de type « tableau »....1656
46.6.7.1. La classe Table....1656
46.6.7.2. La classe TableColumn....1657
46.6.7.3. La classe TableItem....1658
46.6.8. Les contrôles de type « arbre »....1658
46.6.8.1. La classe Tree....1658
46.6.8.2. La classe TreeItem....1659
46.6.9. La classe ScrollBar....1660

Table des matières

46. Le développement d'interfaces graphiques avec SWT

46.6.10. Les contrôles pour le graphisme....1660
46.6.10.1. La classe Canvas....1660
46.6.10.2. La classe GC....1660
46.6.10.3. La classe Color....1661
46.6.10.4. La classe Font....1662
46.6.10.5. La classe Image....1663
46.7. Les conteneurs....1664
46.7.1. Les conteneurs de base....1664
46.7.1.1. La classe Composite....1664
46.7.1.2. La classe Group....1665
46.7.2. Les contrôles de type « barre d'outils »....1666
46.7.2.1. La classe ToolBar....1666
46.7.2.2. La classe ToolItem....1667
46.7.2.3. Les classes CoolBar et Cooltem....1669
46.8. La gestion des erreurs....1670
46.9. Le positionnement des contrôles....1670
46.9.1. Le positionnement absolu....1670
46.9.2. Le positionnement relatif avec les LayoutManager....1670
46.9.2.1. FillLayout....1671
46.9.2.2. RowLayout....1671
46.9.2.3. GridLayout....1673
46.9.2.4. FormLayout....1675
46.10. La gestion des événements....1675
46.10.1. L'interface KeyListener....1677
46.10.2. L'interface MouseListener....1678
46.10.3. L'interface MouseMoveListener....1679
46.10.4. L'interface MouseTrackListener....1680
46.10.5. L'interface ModifyListener....1681
46.10.6. L'interface VerifyText()....1681
46.10.7. L'interface FocusListener....1682
46.10.8. L'interface TraverseListener....1683
46.10.9. L'interface PaintListener....1684
46.11. Les boîtes de dialogue....1685
46.11.1. Les boîtes de dialogue prédéfinies....1685
46.11.1.1. La classe MessageBox....1685
46.11.1.2. La classe ColorDialog....1686
46.11.1.3. La classe FontDialog....1687
46.11.1.4. La classe FileDialog....1688
46.11.1.5. La classe DirectoryDialog....1688
46.11.1.6. La classe PrintDialog....1689
46.11.2. Les boîtes de dialogue personnalisées....1690

47. JFace....1692
47.1. La présentation de JFace....1692
47.2. La structure générale d'une application....1693
47.3. Les boites de dialogue....1694
47.3.1. L'affichage des messages d'erreur....1694
47.3.2. L'affichage des messages d'information à l'utilisateur....1695
47.3.3. La saisie d'une valeur par l'utilisateur....1697
47.3.4. La boîte de dialogue pour afficher la progression d'un traitement....1698

Partie 7 : L'utilisation de documents XML et JSON....1702

48. Java et XML....1704
48.1. La présentation de XML....1704
48.2. Les règles pour formater un document XML....1705
48.3. La DTD (Document Type Definition)....1705
48.4. Les parseurs....1705
48.5. La génération de données au format XML....1706

Table des matières

48. Java et XML

48.6. JAXP : Java API for XML Parsing....1707
48.6.1. JAXP 1.1....1707
48.6.2. L'utilisation de JAXP avec un parseur de type SAX....1708
48.7. Jaxen....1709

49. SAX (Simple API for XML)....1710
49.1. L'utilisation de SAX de type 1....1710
49.2. L'utilisation de SAX de type 2....1717

50. DOM (Document Object Model)....1719
50.1. Les interfaces du DOM....1720
50.1.1. L'interface Node....1720
50.1.2. L'interface NodeList....1721
50.1.3. L'interface Document....1721
50.1.4. L'interface Element....1721
50.1.5. L'interface CharacterData....1722
50.1.6. L'interface Attr....1722
50.1.7. L'interface Comment....1722
50.1.8. L'interface Text....1722
50.2. L'obtention d'un arbre DOM....1723
50.3. Le parcours d'un arbre DOM....1723
50.3.1. L'interface DocumentTraversal....1724
50.3.2. L'interface NodeIterator....1724
50.3.3. L'interface TreeWalker....1725
50.3.4. Le filtrage lors du parcours....1726
50.4. La modification d'un arbre DOM....1727
50.4.1. La création d'un document....1727
50.4.2. L'ajout d'un élément....1728
50.5. L'envoi d'un arbre DOM dans un flux....1729
50.5.1. Un exemple avec Xerces....1729

51. XSLT (Extensible Stylesheet Language Transformations)....1731
51.1. XPath....1731
51.2. La syntaxe de XSLT....1732
51.3. Un exemple avec Internet Explorer....1733
51.4. Un exemple avec Xalan 2....1733

52. Les modèles de documents....1735
52.1. L'API JDOM....1735
52.1.1. L'historique de JDOM....1735
52.1.2. La présentation de JDOM....1735
52.1.3. Les fonctionnalités et les caractéristiques....1736
52.1.4. L'installation de JDOM....1737
52.1.4.1. L'installation de JDOM version 1 bêta 7 sous Windows....1737
52.1.4.2. L'installation de la version 1.x....1738
52.1.5. Les différentes entités de JDOM....1738
52.1.5.1. La classe Document....1739
52.1.5.2. La classe DocType....1740
52.1.5.3. La classe Element....1742
52.1.5.4. La classe Attribut....1744
52.1.5.5. La classe Text....1749
52.1.5.6. La classe Comment....1751
52.1.5.7. La classe Namespace....1752
52.1.5.8. La classe CData....1755
52.1.5.9. La classe ProcessingInstruction....1756
52.1.6. La création d'un document....1758
52.1.6.1. La création d'un nouveau document....1759
52.1.6.2. L'obtention d'une instance de Document à partir d'un document XML....1760
52.1.6.3. La création d'éléments....1763

Table des matières

52. Les modèles de documents

52.1.6.4. L'ajout d'éléments fils....1764
52.1.7. L'arborescence d'éléments....1765
52.1.7.1. Le parcours des éléments....1766
52.1.7.2. L'accès direct à un élément fils....1767
52.1.7.3. Le parcours de toute l'arborescence d'un document....1770
52.1.7.4. Les éléments parents....1773
52.1.8. La modification d'un document....1773
52.1.8.1. L'obtention du texte d'un élément....1777
52.1.8.2. La modification du texte d'un élément....1779
52.1.8.3. L'obtention du texte d'un élément fils....1781
52.1.8.4. L'ajout et la suppression des fils....1782
52.1.8.5. Le déplacement d'un ou des éléments....1784
52.1.8.6. La duplication d'un élément....1785
52.1.9. L'utilisation de filtres....1787
52.1.10. L'exportation d'un document....1790
52.1.10.1. L'exportation dans un flux....1791
52.1.10.2. L'exportation dans un arbre DOM....1795
52.1.10.3. L'exportation en SAX....1795
52.1.11. L'utilisation de XSLT....1797
52.1.12. L'utilisation de XPath....1800
52.1.13. L'intégration à Java....1802
52.1.14. Les contraintes de la mise en oeuvre de JDOM....1803
52.2. dom4j....1803
52.2.1. L'installation de dom4j....1803
52.2.2. La création d'un document....1803
52.2.3. Le parcours d'un document....1804
52.2.4. La modification d'un document XML....1805
52.2.5. La création d'un nouveau document XML....1805
52.2.6. L'exportation d'un document....1806

53. JAXB (Java Architecture for XML Binding)....1809
53.1. JAXB 1.0....1809
53.1.1. La génération des classes....1810
53.1.2. L'API JAXB....1812
53.1.3. L'utilisation des classes générées et de l'API....1812
53.1.4. La création d'un nouveau document XML....1813
53.1.5. La génération d'un document XML....1814
53.2. JAXB 2.0....1815
53.2.1. L'obtention de JAXB 2.0....1818
53.2.2. La mise en oeuvre de JAXB 2.0....1818
53.2.3. La génération des classes à partir d'un schéma....1819
53.2.4. La commande xjc....1820
53.2.5. Les classes générées....1820
53.2.6. L'utilisation de l'API JAXB 2.0....1822
53.2.6.1. Le mapping d'un document XML à des objets (unmarshalling)....1822
53.2.6.2. La création d'un document XML à partir d'objets....1824
53.2.6.3. La création d'un document en utilisant des classes annotées....1825
53.2.6.4. La création d'un document en utilisant les classes générées à partir d'un schéma....1828
53.2.7. La configuration de la liaison XML / Objets....1829
53.2.7.1. L'annotation du schéma XML....1829
53.2.7.2. Les annotations de JAXB....1831
53.2.7.3. La génération d'un schéma à partir de classes compilées....1835

54. StAX (Streaming Api for XML)....1836
54.1. La présentation de StAX....1836
54.2. Les deux API de StAX....1837
54.3. Les fabriques....1838
54.4. Le traitement d'un document XML avec l'API du type curseur....1840
54.5. Le traitement d'un document XML avec l'API du type itérateur....1847

Table des matières

54. StAX (Streaming Api for XML)

54.6. La mise en oeuvre des filtres....1850
54.7. L'écriture un document XML avec l'API de type curseur....1853
54.8. L'écriture un document XML avec l'API de type itérateur....1857
54.9. La comparaison entre SAX, DOM et StAX....1860

55. JSON....1863

55.1. La syntaxe de JSON....1864
55.2. L'utilisation de JSON....1865
55.3. JSON ou XML....1866

56. Gson....1869

56.1. La classe Gson....1870
56.2. La sérialisation....1870
56.3. La désérialisation....1873
56.4. La personnalisation de la sérialisation/désérialisation....1877
56.4.1. La classe GsonBuilder....1877
56.4.2. L'utilisation de Serializer et TypeAdapter....1879
56.4.2.1. L'interface JsonSerializer....1880
56.4.2.2. L'interface JsonDeserializer....1881
56.4.2.3. La classe TypeAdapter....1882
56.4.3. L'interface InstanceCreator....1884
56.4.4. Le formatage de la représentation JSON....1884
56.4.5. Le support des valeurs null....1885
56.4.6. L'exclusion de champs....1885
56.4.6.1. L'exclusion de champs sur la base de modificateurs....1886
56.4.6.2. Les stratégies d'exclusion personnalisées....1886
56.4.7. Le nommage des éléments....1888
56.5. Les annotations de Gson....1889
56.5.1. La personnalisation forcée des noms des champs....1889
56.5.2. La désignation des champs à prendre en compte....1890
56.5.3. La gestion des versions....1892
56.6. L'API Streaming....1894
56.6.1. L'énumération JsonToken....1894
56.6.2. Le parcours d'un document JSON avec la classe JsonReader....1895
56.6.2.1. Le traitement d'un objet....1896
56.6.2.2. Le traitement d'un tableau....1897
56.6.2.3. Le traitement d'éléments composites....1898
56.6.3. La création d'un document JSON avec la classe JsonWriter....1901
56.6.3.1. L'ajout d'un objet....1902
56.6.3.2. L'ajout d'un tableau....1904
56.6.3.3. L'ajout d'éléments composites....1905
56.7. Mixer l'utilisation du model objets et de l'API Streaming....1905
56.7.1. Mixer l'utilisation pour analyser un document....1905
56.7.2. Mixer l'utilisation pour générer un document....1907
56.8. Les concepts avancés....1908
56.8.1. La sérialisation/désérialisation de types generic....1908
56.8.2. La sérialisation/désérialisation d'une collection contenant différents types....1910
56.8.3. La désérialisation quand un même objet est référencé plusieurs fois....1911

57. JSON-P (Java API for JSON Processing)....1916

57.1. La classe Json....1917
57.2. L'API Streaming....1917
57.2.1. L'interface JsonParser....1918
57.2.2. L'interface JsonGenerator....1920
57.3. L'API Object Model....1921
57.3.1. Les classes qui encapsulent un élément d'un document Json....1922
57.3.1.1. L'interface JsonValue....1922
57.3.1.2. L'interface JsonNumber....1922
57.3.1.3. L'interface JsonString....1923

Table des matières

57. JSON-P (Java API for JSON Processing)

57.3.1.4. L'interface JsonStructure....1923
57.3.1.5. L'interface JsonObject....1923
57.3.1.6. L'interface JsonArray....1924
57.3.2. L'interface JsonReader....1925
57.3.3. Le parcours du modèle objet....1926
57.3.4. L'interface JsonArrayBuilder....1927
57.3.5. L'interface JsonObjectBuilder....1928
57.3.6. L'interface JsonWriter....1929
57.3.7. Les interfaces JsonXXXFactory....1930

58. JSON-B (Java API for JSON Binding)....1932

58.1. La sérialisation/désérialisation d'un objet....1932
58.2. Le moteur JSON-B....1934
58.3. Le mapping par défaut....1935
58.3.1. Le mapping par défaut des types communs....1936
58.3.2. Le mapping par défaut d'un type quelconque....1939
58.3.3. Le mapping par défaut des tableaux et des collections....1941
58.3.4. Le mapping par défaut des classes de JSON-P....1943
58.4. La configuration du moteur JSON-B....1943
58.4.1. La classe JsonbConfig....1944
58.4.2. Le formatage du résultat de la sérialisation....1945
58.5. La personnalisation du mapping....1945
58.5.1. Le nom d'une propriété....1945
58.5.2. La stratégie de nommage des propriétés....1947
58.5.3. Les stratégies de gestion de l'ordre des propriétés....1948
58.5.4. Ignorer une propriété....1948
58.5.5. La visibilité des propriétés....1950
58.5.6. La gestion des valeurs null....1951
58.5.7. La personnalisation de la création d'instanciation....1953
58.5.8. Le format des dates et des nombres....1955
58.5.9. Binary Data Encoding....1958
58.5.10. Strict I-JSON support....1958
58.5.11. Les adaptateurs....1959
58.5.12. Les Serializers/Deserializers....1961

Partie 8 : L'accès aux bases de données....1966

59. La persistance des objets....1967

59.1. La correspondance entre les modèles relationnel et objet....1967
59.2. L'évolution des solutions de persistance avec Java....1968
59.3. Le mapping O/R (objet/relationnel)....1968
59.3.1. Le mapping de l'héritage de classes....1968
59.3.1.1. Une table par hiérarchie de classes....1969
59.3.1.2. Une table par sous-classe....1969
59.3.1.3. Une table par classe concrète....1970
59.3.2. Le choix d'une solution de mapping O/R....1970
59.3.3. Les difficultés lors de la mise en place d'un outil de mapping O/R....1971
59.4. L'architecture et la persistance de données....1971
59.4.1. La couche de persistance....1972
59.4.2. Les opérations de type CRUD....1972
59.4.3. Le modèle de conception DAO (Data Access Object)....1972
59.5. Les différentes solutions....1973
59.6. Les API standards....1973
59.6.1. JDBC....1973
59.6.2. JDO 1.0....1973
59.6.3. JD0 2.0....1974
59.6.4. EJB 2.0....1974
59.6.5. Java Persistence API et EJB 3.0....1974
59.7. Les frameworks open source....1975

Table des matières

59. La persistance des objets

59.7.1. iBatis....1975
59.7.2. MyBatis....1975
59.7.3. Hibernate....1975
59.7.4. EclipseLink....1975
59.7.5. Castor....1976
59.7.6. Apache Torque....1976
59.7.7. TopLink....1976
59.7.8. Apache OJB....1976
59.7.9. Apache Cayenne....1976
59.8. L'utilisation de procédures stockées....1976

60. JDBC....1978

60.1. Les outils nécessaires pour utiliser JDBC....1979
60.2. Les types de pilotes JDBC....1979
60.2.1. L'enregistrement d'une base de données dans ODBC sous Windows 9x ou XP....1980
60.3. La présentation des classes et interfaces de base de l'API JDBC....1981
60.4. La connexion à une base de données....1982
60.4.1. Le chargement explicite du pilote avant Java 6....1982
60.4.2. Le chargement du pilote par le DriverManager....1983
60.4.3. L'établissement de la connexion....1983
60.5. L'accès à la base de données....1986
60.5.1. L'exécution de requêtes SQL....1986
60.5.2. Le parcours des enregistrements obtenus en retour....1988
60.5.3. Un exemple complet de mise à jour et de sélection sur une table....1989
60.6. L'obtention d'informations sur la base de données....1990
60.6.1. L'interface ResultSetMetaData....1991
60.6.2. L'interface DatabaseMetaData....1991
60.7. L'utilisation d'un objet de type PreparedStatement....1992
60.8. L'utilisation des transactions....1993
60.9. Les procédures stockées....1994
60.10. Le traitement des erreurs JDBC....1994
60.11. Les évolutions de l'API JDBC....1995
60.11.1. JDBC 2.0....1995
60.11.1.1. Les fonctionnalités de l'objet ResultSet....1996
60.11.1.2. Les mises à jour de masse (Batch Updates)....1998
60.11.1.3. Le package javax.sql....1998
60.11.1.4. L'interface DataSource....1999
60.11.1.5. Les pools de connexions....2000
60.11.1.6. Les transactions distribuées....2000
60.11.1.7. L'API RowSet....2000
60.11.2. JDBC 3.0....2017
60.11.2.1. Le nommage des paramètres d'un objet de type CallableStatement....2017
60.11.2.2. Les types java.sql.Types.DATALINK et java.sql.Types.BOOLEAN....2018
60.11.2.3. L'obtention des valeurs générées automatiquement lors d'une insertion....2018
60.11.2.4. Le support des points de sauvegarde (savepoint)....2019
60.11.2.5. Le pool d'objets PreparedStatements....2021
60.11.2.6. La définition de propriétés pour les pools de connexions....2021
60.11.2.7. L'ajout de metadata pour obtenir la liste des types de données supportés....2021
60.11.2.8. L'utilisation de plusieurs ResultSets retournés par un CallableStatement....2022
60.11.2.9. Préciser si un ResultSet doit être maintenu ouvert ou fermé à la fin d'une transaction....2022
60.11.2.10. La mise à jour des données de type BLOB, CLOB, REF et ARRAY....2022
60.11.3. JDBC 4.0....2024
60.11.3.1. Le chargement automatique des implémentations de Driver....2024
60.11.3.2. Le support du type ROWID de SQL....2025
60.11.3.3. Les améliorations dans la gestion des exceptions....2025
60.11.4. Le support du type XML de SQL....2026
60.11.5. Le support des types National Character de SQL....2028
60.11.6. Des améliorations dans la prise en charge des objets de grande taille....2029
60.11.7. Les nouvelles fonctionnalités de l'API JDBC 4.1....2029

Table des matières

60. JDBC

60.11.7.1. L'utilisation des ressources JDBC dans un try-with-resources....2029
60.11.7.2. L'API RowSet 1.1 : la création d'instance de type RowSet avec des fabriques....2029
60.11.7.3. Mapping supplémentaires des objets Java aux types JDBC dans l'interface CallableStatement....2030
60.11.7.4. Les améliorations dans les méthodes valueOf() des classes Date et Timestamp....2030
60.11.7.5. Des changements dans l'interface Connection....2031
60.11.7.6. L'ajout d'une fonction de limitation des lignes retournées....2031
60.11.8. Les nouvelles fonctionnalités de l'API JDBC 4.2....2031
60.11.8.1. L'ajout du support du type REF_CURSOR....2032
60.11.8.2. L'ajout de l'interface java.sql.DriverAction....2032
60.11.8.3. L'ajout de l'interface java.sql.SQLType....2032
60.11.8.4. L'ajout de l'énumération java.sql.JDBCType....2033
60.11.8.5. Le support de très nombreuses mises à jour exécutées....2033
60.12. MySQL et Java....2033
60.12.1. Les opérations de base avec MySQL....2033
60.12.2. L'utilisation de MySQL avec JDBC....2035
60.13. L'amélioration des performances avec JDBC....2036
60.13.1. Le choix du pilote JDBC à utiliser....2036
60.13.2. La mise en oeuvre de bonnes practiques....2037
60.13.3. L'utilisation des connexions et des Statements....2037
60.13.4. L'utilisation d'un pool de connexions....2037
60.13.5. La configuration et l'utilisation des ResultSets en fonction des besoins....2038
60.13.6. L'utilisation des PreparedStatement....2038
60.13.7. La maximisation des traitements effectués par la base de données :....2038
60.13.8. L'exécution de plusieurs requêtes en mode batch....2038
60.13.9. Prêter une attention particulière aux transactions....2039
60.13.10. L'utilisation des fonctionnalités de JDBC 3.0....2039
60.13.11. Les optimisations sur la base de données....2039
60.13.12. L'utilisation d'un cache....2040

61. JDO (Java Data Object)....2041
61.1. La présentation de JDO....2041
61.2. Un exemple avec Lido....2042
61.2.1. La création de la classe qui va encapsuler les données....2043
61.2.2. La création de l'objet qui va assurer les actions sur les données....2044
61.2.3. La compilation....2044
61.2.4. La définition d'un fichier metadata....2044
61.2.5. L'enrichissement des classes contenant des données....2045
61.2.6. La définition du schéma de la base de données....2045
61.2.7. L'exécution de l'exemple....2047
61.3. L'API JDO....2048
61.3.1. L'interface PersistenceManager....2048
61.3.2. L'interface PersistenceManagerFactory....2048
61.3.3. L'interface PersistenceCapable....2049
61.3.4. L'interface Query....2049
61.3.5. L'interface Transaction....2049
61.3.6. L'interface Extent....2049
61.3.7. La classe JDOHelper....2050
61.4. La mise en oeuvre....2050
61.4.1. La définition d'une classe qui va encapsuler les données....2050
61.4.2. La définition d'une classe qui va utiliser les données....2051
61.4.3. La compilation des classes....2051
61.4.4. La définition d'un fichier de description....2051
61.4.5. L'enrichissement de la classe qui va contenir les données....2051
61.5. Le parcours de toutes les occurrences....2052
61.6. La mise en oeuvre de requêtes....2053

Table des matières

62. Hibernate.....2055
62.1. La création d'une classe qui va encapsuler les données.....2056
62.2. La création d'un fichier de correspondance.....2057
62.3. Les propriétés de configuration.....2059
62.4. L'utilisation d'Hibernate.....2061
62.5. La persistance d'une nouvelle occurrence.....2062
62.6. L'obtention d'une occurrence à partir de son identifiant.....2063
62.7. L'obtention de données.....2064
62.7.1. Le langage de requête HQL.....2064
62.7.1.1. La syntaxe de HQL.....2065
62.7.1.2. La mise en oeuvre de HQL.....2066
62.7.2. L'API Criteria.....2070
62.7.2.1. L'utilité de HQL et de l'API Criteria.....2071
62.7.2.2. L'interface Criteria.....2073
62.7.2.3. L'interface Criterion.....2074
62.7.2.4. Les restrictions et les expressions.....2075
62.7.2.5. Les projections et les aggregations.....2078
62.7.2.6. La classe Property.....2080
62.7.2.7. Le tri des résultats.....2081
62.7.2.8. La jointure de tables.....2082
62.7.2.9. La création de critères à partir de données.....2084
62.7.2.10. Le choix entre HQL et l'API Criteria.....2085
62.8. La mise à jour d'une occurrence.....2085
62.9. La suppression d'une ou plusieurs occurrences.....2085
62.10. Les relations.....2086
62.10.1. Les relations un / un.....2086
62.10.1.1. Le mapping avec un Component.....2087
62.10.1.2. Le mapping avec une relation One-to-One avec clé primaire partagée.....2094
62.10.1.3. Le mapping avec une relation One-to-One avec clé étrangère.....2103
62.11. Le mapping de l'héritage de classes.....2111
62.11.1. XML.....2114
62.11.1.1. XML, une table par hiérarchie de classes.....2115
62.11.1.2. XML, une table par sous-classe.....2119
62.11.1.3. XML, une table par classe concrète.....2123
62.11.1.4. XML, une table par sous-classe avec discriminant.....2126
62.11.2. Annotations.....2131
62.11.2.1. Annotations, une table par hiérarchie de classes (SINGLE_TABLE).....2132
62.11.2.2. Annotations, une table par classe concrète (TABLE_PER_CLASS).....2135
62.11.2.3. Annotations, une table par sous-classe (JOINED).....2139
62.12. Les caches d'Hibernate.....2143
62.12.1. Des recommandations pour l'utilisation des caches.....2145
62.12.2. Les différents caches d'Hibernate.....2145
62.12.2.1. La base des exemples de cette section.....2146
62.12.2.2. Le cache de premier niveau.....2147
62.12.2.3. Le cache de second niveau.....2148
62.12.3. La configuration du cache de second niveau.....2150
62.12.3.1. Les différents caches supportés par Hibernate.....2150
62.12.3.2. La configuration du cache de second niveau.....2151
62.12.3.3. La configuration du cache Ehcache.....2152
62.12.4. Les stratégies d'usage transactionnel du cache.....2155
62.12.4.1. La stratégie read-only.....2156
62.12.4.2. La stratégie read-write.....2156
62.12.4.3. La stratégie nonstrict-read-write.....2156
62.12.4.4. La stratégie transactional.....2157
62.12.4.5. Le support des stratégies par les différents caches.....2158
62.12.5. Le cache des entités.....2159
62.12.6. Le cache des associations many.....2165
62.12.7. Le cache des requêtes.....2171
62.12.8. La gestion du cache de second niveau.....2177
62.12.9. Le monitoring de l'utilisation du cache.....2179

Table des matières

62. Hibernate

62.12.9.1. L'activation et l'obtention de données statistiques....2179
62.13. Les outils de génération de code....2184

63. JPA (Java Persistence API)....2187
63.1. L'installation de l'implémentation de référence....2187
63.2. Les entités....2188
63.2.1. Le mapping entre le bean entité et la table....2188
63.2.2. Le mapping de propriétés complexes....2197
63.2.3. Le mapping d'une entité sur plusieurs tables....2199
63.2.4. L'utilisation d'objets embarqués dans les entités....2202
63.3. Le fichier de configuration du mapping....2204
63.4. L'utilisation du bean entité....2204
63.4.1. L'utilisation du bean entité....2205
63.4.2. L'EntityManager....2205
63.4.2.1. L'obtention d'une instance de la classe EntityManager....2205
63.4.2.2. L'utilisation de la classe EntityManager....2206
63.4.2.3. L'utilisation de la classe EntityManager pour la création d'une occurrence....2206
63.4.2.4. L'utilisation de la classe EntityManager pour rechercher des occurrences....2207
63.4.2.5. L'utilisation de la classe EntityManager pour rechercher des données par requête....2208
63.4.2.6. L'utilisation de la classe EntityManager pour modifier une occurrence....2209
63.4.2.7. L'utilisation de la classe EntityManager pour fusionner des données....2210
63.4.2.8. L'utilisation de la classe EntityManager pour supprimer une occurrence....2211
63.4.2.9. L'utilisation de la classe EntityManager pour rafraîchir les données d'une occurrence....2212
63.5. Le fichier persistence.xml....2212
63.6. La gestion des transactions hors Java EE....2213
63.7. La gestion des relations entre tables dans le mapping....2214
63.8. Le mapping de l'héritage de classes....2215
63.8.1. Les annotations....2215
63.8.1.1. L'annotation @MappedSuperclass....2215
63.8.1.2. L'annotation @Inheritance....2217
63.8.1.3. L'annotation @DiscriminatorColumn....2217
63.8.1.4. L'annotation @DiscriminatorValue....2218
63.8.2. Des exemples de mises en oeuvre des stratégies....2218
63.8.3. La stratégie une table par hiérarchie de classes (SINGLE_TABLE)....2219
63.8.4. La stratégie : une classe par classe concrète (TABLE_PER_CLASS)....2222
63.8.5. La stratégie une table par sous-classe (JOINED)....2225
63.9. Les callbacks d'événements....2227

Partie 9 : La machine virtuelle Java (JVM)....2230

64. La JVM (Java Virtual Machine)....2231
64.1. La mémoire de la JVM....2232
64.1.1. Le Java Memory Model....2232
64.1.2. Les différentes zones de la mémoire....2232
64.1.2.1. La Pile (Stack)....2233
64.1.2.2. Le tas (Heap)....2233
64.1.2.3. La zone de mémoire "Method area"....2233
64.1.2.4. La zone de mémoire "Code Cache"....2233
64.2. Le cycle de vie d'une classe dans la JVM....2233
64.2.1. Le chargement des classes....2234
64.2.1.1. La recherche des fichiers .class....2235
64.2.1.2. Le chargement du bytecode....2236
64.2.2. La liaison de la classe....2236
64.2.3. L'initialisation de la classe....2237
64.2.4. Le chargement des classes et la police de sécurité....2238
64.3. Les ClassLoaders....2238
64.3.1. Le mode de fonctionnement d'un ClassLoader....2239
64.3.2. La délégation du chargement d'une classe....2242
64.3.2.1. L'écriture d'un classloader personnalisé....2243

Table des matières

64. La JVM (Java Virtual Machine)
64.4. Le bytecode....2244
64.4.1. La version du bytecode....2245
64.4.2. L'outil Jclasslib bytecode viewer....2246
64.4.3. Le jeu d'instructions de la JVM....2247
64.4.4. Le format des fichiers .class....2247
64.5. Le compilateur JIT....2248
64.6. Les paramètres de la JVM HotSpot....2248
64.7. Les interactions de la machine virtuelle avec des outils externes....2253
64.7.1. L'API Java Virtual Machine Debug Interface (JVMDI)....2253
64.7.2. L'API Java Virtual Machine Profiler Interface (JVMPI)....2254
64.7.3. L'API Java Virtual Machine Tools Interface (JVMTI)....2254
64.7.4. L'architecture Java Platform Debugger Architecture (JPDA)....2254
64.7.5. Des outils de profiling....2255
64.8. Service Provider Interface (SPI)....2255
64.8.1. La mise en oeuvre du SPI....2255
64.8.2. La classe ServiceLoader....2256
64.9. Les JVM 32 et 64 bits....2258
64.9.1. L'avantage des architectures 64 bits....2258
64.9.2. JLS et JVM 64 bits....2259
64.9.3. L'introduction de la fonctionnalité compressed OOPS (Ordinary Object Pointers)....2259
64.9.4. Les limitations et les contraintes avec une JVM 32 et 64 bits....2260
64.9.4.1. Les limites de la taille du heap d'une JVM 32 bits....2260
64.9.4.2. L'utilisation d'une JVM 64 bits....2260
64.9.4.3. Les contraintes liées à l'utilisation de bibliothèques natives....2262
64.9.5. Le choix entre une JVM 32 ou 64 bits....2262
64.9.6. Déterminer si la JVM est 32 ou 64 bits....2262

65. La gestion de la mémoire dans la JVM HotSpot....2264
65.1. Le ramasse-miettes (Garbage Collector ou GC)....2264
65.1.1. Le rôle du ramasse-miettes....2265
65.1.2. Les différents concepts des algorithmes du ramasse-miettes....2265
65.1.3. L'utilisation de générations....2266
65.1.4. Les limitations du ramasse-miettes....2267
65.1.5. Les facteurs de performance du ramasse-miettes....2268
65.2. Le fonctionnement du ramasse-miettes de la JVM Hotspot....2268
65.2.1. Les trois générations utilisées....2269
65.2.2. Les algorithmes d'implémentation du GC....2271
65.2.2.1. Le Serial Collector....2271
65.2.2.2. Le Parallel Collector ou Throughput Collector....2273
65.2.2.3. Le Parallel Compacting Collector....2275
65.2.2.4. Le Concurrent Mark-Sweep (CMS) Collector....2276
65.2.3. L'auto sélection des paramètres du GC par la JVM Hotspot....2280
65.2.4. La sélection explicite d'un algorithme pour le GC....2281
65.2.5. Demander l'exécution d'une collection majeure....2282
65.2.6. La méthode finalize()....2282
65.3. Le paramétrage du ramasse-miettes de la JVM HotSpot....2283
65.3.1. Les options pour configurer le ramasse-miettes....2283
65.3.2. La configuration de la JVM pour améliorer les performances du GC....2285
65.4. Le monitoring de l'activité du ramasse-miettes....2287
65.5. Les différents types de référence....2287
65.6. L'obtention d'informations sur la mémoire de la JVM....2287
65.7. Les fuites de mémoire (Memory leak)....2287
65.7.1. Diagnostiquer une fuite de mémoire....2288
65.8. Les exceptions liées à un manque de mémoire....2289
65.8.1. L'exception de type StackOverFlowError....2290
65.8.2. L'exception de type OutOfMemoryError....2290
65.8.2.1. L'exception OutOfMemoryError : Java heap space....2290
65.8.2.2. L'exception OutOfMemoryError : PermGen space....2291
65.8.2.3. L'exception OutOfMemoryError : Requested array size exceeds VM limit....2291

Table des matières

66. La JVM HotSpot dans un conteneur Docker....2292
66.1. La limitation des ressources d'un conteneur Docker....2292
66.1.1. La limitation en mémoire....2293
66.1.2. La limitation en CPU....2295
66.1.2.1. CPU Shares....2296
66.1.2.2. CPU Period/Quota....2297
66.1.2.3. CPU Sets....2297
66.1.2.4. Le choix du type de limitation....2298
66.2. L'utilisation et la configuration des ressources utilisables par une JVM....2298
66.2.1. Les ergonomics de la JVM....2299
66.2.2. L'utilisation du nombre de CPU par la JVM....2299
66.2.3. La gestion de la mémoire....2300
66.3. Le support de Docker par la JVM....2302
66.3.1. Java SE 8 < u131....2302
66.3.2. Les évolutions dans Java SE 9 et Java SE 8u131....2303
66.3.3. Les évolutions dans Java SE 10 et Java SE 8u191....2306
66.3.4. Les évolutions dans Java SE 11....2308
66.3.5. L'influence des ergonomics....2308
66.4. Le support de cgroups v2....2310

67. La décompilation et l'obfuscation....2311
67.1. Décompiler du bytecode....2311
67.1.1. JAD : the fast Java Decompiler....2312
67.1.2. La mise en oeuvre et les limites de la décompilation....2312
67.2. Obfusquer le bytecode....2316
67.2.1. Le mode de fonctionnement de l'obfuscation....2317
67.2.2. Un exemple de mise en oeuvre avec ProGuard....2317
67.2.3. Les problèmes possibles lors de l'obfuscation....2322
67.2.4. L'utilisation d'un ClassLoader dédié....2322

68. Programmation orientée aspects (AOP)....2323
68.1. Le besoin d'un autre modèle de programmation....2323
68.2. Les concepts de l'AOP....2324
68.3. La mise en oeuvre de l'AOP....2325
68.4. Les avantages et les inconvénients....2326
68.5. Des exemples d'utilisation....2326
68.6. Des implémentations pour la plate-forme Java....2327

69. Terracotta....2328
69.1. La présentation de Terrocatta....2330
69.1.1. Le mode de fonctionnement....2330
69.1.2. La gestion des objets par le cluster....2331
69.1.3. Les avantages de Terracotta....2332
69.1.4. L'architecture d'un cluster Terracotta....2332
69.2. Les concepts utilisés....2333
69.2.1. Les racines (root)....2334
69.2.2. Les transactions....2334
69.2.3. Les verrous (locks)....2335
69.2.4. L'instrumentation des classes....2336
69.2.5. L'invocation distribuée de méthodes....2336
69.2.6. Le ramasse-miettes distribué....2337
69.3. La mise en oeuvre des fonctionnalités....2337
69.3.1. L'installation....2337
69.3.2. Les modules d'intégration....2338
69.3.3. Les scripts de commande....2338
69.3.4. Les limitations....2339
69.4. Les cas d'utilisation....2340
69.4.1. La réplication de sessions HTTP....2340
69.4.2. Un cache distribué....2341
69.4.3. La répartition de charge de traitements....2341

Table des matières

69. Terracotta

69.4.4. Le partage de données entre applications 2341
69.5. Quelques exemples de mise en oeuvre 2341
69.5.1. Un exemple simple avec une application standalone 2342
69.5.2. Un second exemple avec une application standalone 2343
69.5.3. Un exemple avec une application web 2345
69.5.4. Le partage de données entre deux applications 2347
69.6. La console développeur 2350
69.7. Le fichier de configuration 2351
69.7.1. La configuration de la partie system 2351
69.7.2. La configuration de la partie serveur 2352
69.7.3. La configuration de la partie cliente 2353
69.7.4. La configuration de la partie applicative 2353
69.7.4.1. La définition des racines 2354
69.7.4.2. La définition des verrous 2354
69.7.4.3. La définition des classes à instrumenter 2355
69.7.4.4. La définition des champs qui ne doivent pas être gérés 2356
69.7.4.5. La définition des méthodes dont l'invocation doit être distribuée 2356
69.7.4.6. La définition des webapps dont la session doit être gérée 2356
69.8. La fiabilisation du cluster 2356
69.8.1. La configuration minimale 2357
69.8.2. La configuration pour la fiabilité 2358
69.8.3. La configuration pour une haute disponibilité 2359
69.8.4. La configuration pour une haute disponibilité et la fiabilité 2361
69.8.5. La configuration pour un environnement de production 2362
69.8.6. La configuration pour la montée en charge 2363
69.9. Quelques recommandations 2363

Partie 10 : Le développement d'applications d'entreprises 2364

70. J2EE / Java EE 2365
70.1. La présentation de J2EE 2365
70.2. Les API de J2EE / Java EE 2366
70.3. L'environnement d'exécution des applications J2EE 2368
70.3.1. Les conteneurs 2368
70.3.2. Le conteneur web 2369
70.3.3. Le conteneur d'EJB 2369
70.3.4. Les services proposés par la plate-forme J2EE 2369
70.4. L'assemblage et le déploiement d'applications J2EE 2369
70.4.1. Le contenu et l'organisation d'un fichier EAR 2369
70.4.2. La création d'un fichier EAR 2370
70.4.3. Les limitations des fichiers EAR 2370
70.5. J2EE 1.4 SDK 2370
70.5.1. L'installation de l'implémentation de référence sous Windows 2371
70.5.2. Le démarrage et l'arrêt du serveur 2372
70.5.3. L'outil asadmin 2373
70.5.4. Le déploiement d'applications 2374
70.5.5. La console d'administration 2374
70.6. La présentation de Java EE 5.0 2374
70.6.1. La simplification des développements 2376
70.6.2. La version 3.0 des EJB 2376
70.6.3. Un accès facilité aux ressources grâce à l'injection de dépendance 2377
70.6.4. JPA : le nouvelle API de persistance 2378
70.6.5. Des services web plus simples à écrire 2379
70.6.6. Le développement d'applications Web 2380
70.6.7. Les autres fonctionnalités 2380
70.6.8. L'installation du SDK Java EE 5 sous Windows 2380
70.7. La présentation de Java EE 6 2382
70.7.1. Les spécifications de Java EE 6 2382
70.7.2. Une plate-forme plus légère 2383

Table des matières

70. J2EE / Java EE

70.7.2.1. La notion de profile....2384
70.7.2.2. La notion de pruning....2384
70.7.3. Les évolutions dans les spécifications existantes....2385
70.7.3.1. Servlet 3.0....2385
70.7.3.2. JSF 2.0....2385
70.7.3.3. Les EJB 3.1....2386
70.7.3.4. JPA 2.0....2386
70.7.3.5. JAX-WS 2.2....2386
70.7.3.6. Interceptors 1.1....2386
70.7.4. Les nouvelles API....2387
70.7.4.1. JAX-RS 1.1....2387
70.7.4.2. Contexte and Dependency Injection (WebBeans) 1.0....2387
70.7.4.3. Dependency Injection 1.0....2388
70.7.4.4. Bean Validation 1.0....2388
70.7.4.5. Managed Beans 1.0....2388
70.8. La présentation de Java EE 7....2388
70.8.1. Java API for JSON Processing 1.0 (JSR 353)....2389
70.8.2. Java API for WebSocket 1.0 (JSR 356)....2390
70.8.3. Batch Applications for the Java Platform 1.0 (JSR 352)....2391
70.8.4. Concurrency utilities for Java EE (JSR 236)....2393
70.8.5. Java Message Service 2.0 (JSR 343)....2395
70.8.6. BeanValidation 1.1 (JSR 349)....2397
70.8.7. Java API for RESTful Web Services 2.0 (JSR 339)....2398
70.8.8. Servlets 3.1 (JSR 340)....2398
70.8.9. JPA 2.1 (JSR 338)....2398
70.8.10. JTA 1.2....2399
70.9. La présentation de Java EE 8/Jakarta EE 8....2399

71. JavaMail....2401
71.1. Le téléchargement et l'installation....2401
71.2. Les principaux protocoles....2402
71.2.1. Le protocole SMTP....2402
71.2.2. Le protocole POP....2402
71.2.3. Le protocole IMAP....2402
71.2.4. Le protocole NNTP....2402
71.3. Les principales classes et interfaces de l'API JavaMail....2402
71.3.1. La classe Session....2403
71.3.2. Les classes Address, InternetAddress et NewsAddress....2403
71.3.3. L'interface Part....2404
71.3.4. La classe Message....2404
71.3.5. Les classes Flags et Flag....2406
71.3.6. La classe Transport....2407
71.3.7. La classe Store....2407
71.3.8. La classe Folder....2407
71.3.9. Les propriétés d'environnement....2408
71.3.10. La classe Authenticator....2408
71.4. L'envoi d'un e-mail par SMTP....2409
71.5. La récupération des messages d'un serveur POP3....2409
71.6. Les fichiers de configuration....2410
71.6.1. Les fichiers javamail.providers et javamail.default.providers....2410
71.6.2. Les fichiers javamail.address.map et javamail.default.address.map....2410

72. JMS (Java Message Service)....2412
72.1. La présentation de JMS....2412
72.2. Les services de messages....2413
72.3. Le package javax.jms....2414
72.3.1. La fabrique de connexions....2414
72.3.2. L'interface Connection....2414
72.3.3. L'interface Session....2415

Table des matières

72. JMS (Java Message Service)
72.3.4. Les messages....2415
72.3.4.1. L'en-tête....2416
72.3.4.2. Les propriétés....2416
72.3.4.3. Le corps du message....2416
72.3.5. L'envoi de messages....2417
72.3.6. La réception de messages....2417
72.4. L'utilisation du mode point à point (queue)....2418
72.4.1. La création d'une fabrique de connexions : QueueConnectionFactory....2418
72.4.2. L'interface QueueConnection....2418
72.4.3. La session : l'interface QueueSession....2419
72.4.4. L'interface Queue....2419
72.4.5. La création d'un message....2419
72.4.6. L'envoi de messages : l'interface QueueSender....2419
72.4.7. La réception de messages : l'interface QueueReceiver....2420
72.4.7.1. La réception dans le mode synchrone....2420
72.4.7.2. La réception dans le mode asynchrone....2420
72.4.7.3. La sélection de messages....2421
72.5. L'utilisation du mode publication/abonnement (publish/subscribe)....2421
72.5.1. La création d'une fabrique de connexions : TopicConnectionFactory....2421
72.5.2. L'interface TopicConnection....2421
72.5.3. La session : l'interface TopicSession....2422
72.5.4. L'interface Topic....2422
72.5.5. La création d'un message....2422
72.5.6. L'émission de messages : l'interface TopicPublisher....2422
72.5.7. La réception de messages : l'interface TopicSubscriber....2422
72.6. La gestion des erreurs....2423
72.6.1. Les exceptions de JMS....2423
72.6.2. L'interface ExceptionListener....2423
72.7. JMS 1.1....2424
72.7.1. L'utilisation de l'API JMS 1.0 et 1.1....2425
72.7.2. L'interface ConnectionFactory....2425
72.7.3. L'interface Connection....2426
72.7.4. L'interface Session....2426
72.7.5. L'interface Destination....2428
72.7.6. L'interface MessageProducer....2428
72.7.7. L'interface MessageConsumer....2428
72.7.7.1. La réception synchrone de messages....2428
72.7.7.2. La réception asynchrone de messages....2429
72.7.8. Le filtrage des messages....2429
72.7.8.1. La définition du filtre....2430
72.7.9. Des exemples de mise en oeuvre....2431
72.8. Les ressources relatives à JMS....2433

73. Les EJB (Entreprise Java Bean)....2434
73.1. La présentation des EJB....2435
73.1.1. Les différents types d'EJB....2435
73.1.2. Le développement d'un EJB....2436
73.1.3. L'interface remote....2436
73.1.4. L'interface home....2437
73.2. Les EJB session....2438
73.2.1. Les EJB session sans état....2439
73.2.2. Les EJB session avec état....2440
73.3. Les EJB entité....2440
73.4. Les outils pour développer et mettre en oeuvre des EJB....2441
73.4.1. Les outils de développement....2441
73.4.2. Les conteneurs d'EJB....2441
73.5. Le déploiement des EJB....2441
73.5.1. Le descripteur de déploiement....2441
73.5.2. La mise en package des beans....2441

Table des matières

73. Les EJB (Entreprise Java Bean)

73.6. L'appel d'un EJB par un client....2441
73.6.1. Un exemple d'appel d'un EJB session....2442
73.7. Les EJB orientés messages....2442

74. Les EJB 3....2443

74.1. L'historique des EJB....2444
74.2. Les nouveaux concepts et fonctionnalités utilisés....2444
74.2.1. L'utilisation de POJO et POJI....2445
74.2.2. L'utilisation des annotations....2445
74.2.3. L'injection de dépendances....2447
74.2.4. La configuration par défaut....2447
74.2.5. Les intercepteurs....2447
74.3. EJB 2.x vs EJB 3.0....2447
74.4. Les conventions de nommage....2448
74.5. Les EJB de type Session....2448
74.5.1. L'interface distante et/ou locale....2449
74.5.2. Les beans de type Stateless....2449
74.5.3. Les beans de type Stateful....2451
74.5.4. L'invocation d'un EJB Session par un service web....2451
74.5.5. L'utilisation des exceptions....2452
74.6. Les EJB de type Entity....2452
74.6.1. La création d'un bean Entity....2453
74.6.2. La persistance des entités....2454
74.6.3. La création d'un EJB Session pour manipuler le bean Entity....2455
74.7. Un exemple simple complet....2455
74.7.1. La création de l'entité....2456
74.7.2. La création de la façade....2457
74.7.3. La création du service web....2458
74.8. L'utilisation des EJB par un client....2459
74.8.1. Pour un client de type application standalone....2459
74.8.2. Pour un client de type module Application Client Java EE....2460
74.9. L'injection de dépendances....2460
74.9.1. L'annotation @javax.ejb.EJB....2460
74.9.2. L'annotation @javax.annotation.Resource....2461
74.9.3. Les annotations @javax.annotation.Resources et @javax.ejb.EJBs....2462
74.9.4. L'annotation @javax.xml.ws.WebServiceRef....2462
74.10. Les intercepteurs....2462
74.10.1. Le développement d'un intercepteur....2462
74.10.1.1. L'interface InvocationContext....2463
74.10.1.2. La définition d'un intercepteur lié aux méthodes métiers....2463
74.10.1.3. La définition d'un intercepteur lié au cycle de vie....2464
74.10.1.4. La mise en oeuvre d'une classe d'un intercepteur....2464
74.10.2. Les intercepteurs par défaut....2465
74.10.3. Les annotations des intercepteurs....2466
74.10.3.1. L'annotation @javax.annotation.PostContruct....2466
74.10.3.2. L'annotation @javax.annotation.PreDestroy....2466
74.10.3.3. L'annotation @javax.interceptor.AroundInvoke....2467
74.10.3.4. L'annotation @javax.interceptor.ExcludeClassInterceptors....2467
74.10.3.5. L'annotation @javax.interceptor.ExcludeDefaultInterceptors....2467
74.10.3.6. L'annotation @javax.interceptor.Interceptors....2467
74.10.4. L'utilisation d'un intercepteur....2468
74.11. Les EJB de type MessageDriven....2469
74.11.1. L'annotation @ javax.ejb.MessageDriven....2469
74.11.2. L'annotation @javax.ejb.ActivationConfigProperty....2470
74.11.3. Un exemple d'EJB de type MDB....2470
74.12. Le packaging des EJB....2474
74.13. Les transactions....2474
74.13.1. La mise en oeuvre des transactions dans les EJB....2474
74.13.2. La définition de transactions....2475

Table des matières

74. Les EJB 3

74.13.2.1. La définition du mode de gestion des transactions dans un EJB....2475
74.13.2.2. La définition de transactions avec l'annotation @TransactionAttribute....2475
74.13.2.3. La définition de transactions dans le descripteur de déploiement....2476
74.13.2.4. Des recommandations sur la mise en oeuvre des transactions....2476
74.14. La mise en oeuvre de la sécurité....2477
74.14.1. L'authentification et l'identification de l'utilisateur....2477
74.14.2. La définition des restrictions....2477
74.14.2.1. La définition des restrictions avec les annotations....2477
74.14.2.2. La définition des restrictions avec le descripteur de déploiement....2478
74.14.3. Les annotations pour la sécurité....2478
74.14.3.1. javax.annotation.security.DeclareRoles....2478
74.14.3.2. javax.annotation.security.DenyAll....2479
74.14.3.3. javax.annotation.security.PermitAll....2479
74.14.3.4. javax.annotation.security.RolesAllowed....2479
74.14.3.5. javax.annotation.security.RunAs....2479
74.14.4. La mise en oeuvre de la sécurité par programmation....2479

75. Les EJB 3.1....2481

75.1. Les interfaces locales sont optionnelles....2481
75.2. Les EJB Singleton....2483
75.3. EJB Lite....2486
75.4. La simplification du packaging....2490
75.5. Les améliorations du service Timer....2493
75.5.1. La définition d'un timer....2493
75.5.2. L'annotation @Schedule....2495
75.5.3. La persistance des timers....2497
75.5.4. L'interface Timer....2498
75.5.5. L'interface TimerService....2499
75.6. La standardisation des noms JNDI....2501
75.7. L'invocation asynchrone des EJB session....2502
75.7.1. L'annotation @Asynchronous....2502
75.7.2. L'invocation d'une méthode asynchrone....2504
75.8. L'invocation d'un EJB hors du conteneur....2507

76. Les services web de type Soap....2508

76.1. La présentation des services web....2509
76.1.1. La définition d'un service web....2509
76.1.2. Les différentes utilisations....2510
76.2. Les standards....2510
76.2.1. SOAP....2511
76.2.1.1. La structure des messages SOAP....2512
76.2.1.2. L'encodage des messages SOAP....2513
76.2.1.3. Les différentes versions de SOAP....2513
76.2.2. WSDL....2514
76.2.2.1. Le format général d'un WSDL....2514
76.2.2.2. L'élément Types....2517
76.2.2.3. L'élément Message....2517
76.2.2.4. L'élément PortType/Interface....2517
76.2.2.5. L'élément Binding....2518
76.2.2.6. L'élément Service....2519
76.2.3. Les registres et les services de recherche....2520
76.2.3.1. UDDI....2520
76.2.3.2. Ebxml....2520
76.3. Les différents formats de services web SOAP....2521
76.3.1. Le format RPC Encoding....2523
76.3.2. Le format RPC Literal....2525
76.3.3. Le format Document encoding....2527
76.3.4. Le format Document Literal....2527
76.3.5. Le format Document Literal wrapped....2530

Table des matières

76. Les services web de type Soap
76.3.6. Le choix du format à utiliser....2532
76.3.6.1. L'utilisation de document/literal....2532
76.3.6.2. L'utilisation de Document/Literal Wrapped....2533
76.3.6.3. L'utilisation de RPC/Literal....2533
76.3.6.4. L'utilisation de RPC/Encoded....2533
76.4. Des conseils pour la mise en oeuvre....2534
76.4.1. Les étapes de la mise en oeuvre....2534
76.4.2. Quelques recommandations....2535
76.4.3. Les problèmes liés à SOAP....2535
76.5. Les API Java pour les services web....2536
76.5.1. JAX-RPC....2536
76.5.1.1. La mise en oeuvre côté serveur....2537
76.5.1.2. La mise en oeuvre côté client....2538
76.5.2. JAXM....2539
76.5.3. JAXR....2539
76.5.4. SAAJ....2539
76.5.5. JAX-WS....2540
76.5.5.1. La mise en oeuvre de JAX-WS....2540
76.5.5.2. La production de service web avec JAX-WS....2541
76.5.5.3. La consommation de services web avec JAX-WS....2545
76.5.5.4. Les handlers....2545
76.5.6. La JSR 181 (Web Services Metadata for the Java Platform)....2545
76.5.6.1. Les annotations définies....2546
76.5.6.2. javax.jws.WebService....2547
76.5.6.3. javax.jws.WebMethod....2547
76.5.6.4. javax.jws.OneWay....2548
76.5.6.5. javax.jws.WebParam....2548
76.5.6.6. javax.jws.WebResult....2549
76.5.6.7. javax.jws.soap.SOAPBinding....2549
76.5.6.8. javax.jws.HandlerChain....2550
76.5.6.9. javax.jws.soap.SOAPMessageHandlers....2550
76.5.7. La JSR 224 (JAX-WS 2.0 Annotations)....2550
76.5.7.1. javax.xml.ws.BindingType....2550
76.5.7.2. javax.xml.ws.RequestWrapper....2550
76.5.7.3. javax.xml.ws.ResponseWrapper....2551
76.5.7.4. javax.xml.ws.ServiceMode....2551
76.5.7.5. javax.xml.ws.WebFault....2551
76.5.7.6. javax.xml.ws.WebEndpoint....2552
76.5.7.7. javax.xml.ws.WebServiceclient....2552
76.5.7.8. javax.xml.ws.WebServiceProvider....2552
76.5.7.9. javax.xml.ws.WebServiceRef....2552
76.6. Les implémentations des services web....2553
76.6.1. Axis 1.0....2553
76.6.1.1. Installation....2555
76.6.1.2. La mise en oeuvre côté serveur....2556
76.6.1.3. Mise en oeuvre côté client....2557
76.6.1.4. L'outil TCPMonitor....2559
76.6.2. Apache Axis 2....2560
76.6.3. Xfire....2560
76.6.4. Apache CXF....2560
76.6.5. JWSDP (Java Web Service Developer Pack)....2561
76.6.5.1. L'installation du JWSDP 1.1....2561
76.6.5.2. L'exécution du serveur....2562
76.6.5.3. L'exécution d'un des exemples....2563
76.6.6. Java EE 5....2564
76.6.7. Java SE 6....2565
76.6.8. Le projet Metro et WSIT....2567
76.7. Inclure des pièces jointes dans SOAP....2567
76.8. WS-I....2567

Table des matières

76. Les services web de type Soap

76.8.1. WS-I Basic Profile....2568
76.9. Les autres spécifications....2568

77. Les WebSockets....2569
77.1. Les limitations du protocole HTTP....2569
77.2. La spécification du protocole WebSocket....2571
77.3. La connexion à une WebSocket....2572
77.4. La mise en oeuvre des WebSockets....2573

78. L'API WebSocket....2574
78.1. Les principales classes et interfaces....2575
78.1.1. L'interface javax.websocket.Session....2575
78.1.2. Les interfaces RemoteEndpoint....2577
78.1.3. Les interfaces MessageHandler....2579
78.2. Le développement d'un endpoint....2581
78.2.1. Le développement d'un endpoint avec les annotations....2581
78.2.1.1. L'annotation @ServerEndpoint....2582
78.2.1.2. L'annotation @OnMessage....2583
78.2.1.3. L'annotation @OnOpen....2584
78.2.1.4. L'annotation @OnClose....2584
78.2.1.5. L'annotation @OnError....2584
78.2.1.6. L'annotation @ClientEndpoint....2585
78.2.2. Le développement d'un endpoint sans annotations....2585
78.2.2.1. La développement d'un endpoint serveur....2586
78.2.2.2. Le développement d'un endpoint client....2587
78.2.3. La configuration des endpoints sans annotations....2591
78.2.3.1. L'interface EndpointConfig....2591
78.2.3.2. La configuration des endpoints serveur....2591
78.2.3.3. La configuration d'un endpoint client....2594
78.3. Les encodeurs et les décodeurs....2595
78.3.1. Les encodeurs....2596
78.3.2. Les décodeurs....2597
78.3.3. L'enregistrement des encodeurs et des décodeurs....2598
78.4. Le débogage des WebSockets....2598
78.5. Des exemples d'utilisation....2600
78.5.1. Un premier cas simple....2600
78.5.2. La mise à jour périodique d'un graphique....2602
78.6. L'utilisation d'implémentations....2604
78.6.1. L'utilisation de Tyrus....2604
78.6.1.1. L'utilisation de Tyrus côté client....2604
78.6.1.2. L'utilisation de Tyrus côté serveur....2606
78.6.2. L'utilisation de Javascript côté client....2608

Partie 11 : Le développement d'applications web....2612

79. Les servlets....2613
79.1. La présentation des servlets....2613
79.1.1. Le fonctionnement d'une servlet (cas d'utilisation de http)....2614
79.1.2. Les outils nécessaires pour développer des servlets....2614
79.1.3. Le rôle du conteneur web....2615
79.1.4. Les différences entre les servlets et les CGI....2615
79.2. L'API servlet....2615
79.2.1. L'interface Servlet....2616
79.2.2. La requête et la réponse....2617
79.2.3. Un exemple de servlet....2617
79.3. Le protocole HTTP....2618
79.4. Les servlets http....2619
79.4.1. La méthode init()....2620
79.4.2. L'analyse de la requête....2620

Table des matières

79. Les servlets

79.4.3. La méthode doGet() ... 2621
79.4.4. La méthode doPost() ... 2621
79.4.5. La génération de la réponse ... 2622
79.4.6. Un exemple de servlet HTTP très simple ... 2623
79.5. Les informations sur l'environnement d'exécution des servlets ... 2624
79.5.1. Les paramètres d'initialisation ... 2624
79.5.2. L'objet ServletContext ... 2625
79.5.3. Les informations contenues dans la requête ... 2626
79.6. L'utilisation des cookies ... 2627
79.6.1. La classe Cookie ... 2628
79.6.2. L'enregistrement et la lecture d'un cookie ... 2628
79.7. Le partage d'informations entre plusieurs échanges HTTP ... 2628
79.8. Packager une application web ... 2629
79.8.1. La structure d'un fichier .war ... 2629
79.8.2. Le fichier web.xml ... 2630
79.8.3. Le déploiement d'une application web ... 2632
79.9. L'utilisation de Log4J dans une servlet ... 2632

80. Les JSP (Java Server Pages) ... 2635
80.1. La présentation des JSP ... 2635
80.1.1. Le choix entre JSP et Servlets ... 2636
80.1.2. Les JSP et les technologies concurrentes ... 2636
80.2. Les outils nécessaires ... 2637
80.2.1. L'outil JavaServer Web Development Kit (JSWDK) sous Windows ... 2638
80.2.2. Le serveur Tomcat ... 2639
80.3. Le code HTML ... 2639
80.4. Les Tags JSP ... 2639
80.4.1. Les tags de directives <%@ ... %> ... 2639
80.4.1.1. La directive page ... 2640
80.4.1.2. La directive include ... 2641
80.4.1.3. La directive taglib ... 2642
80.4.2. Les tags de scripting ... 2642
80.4.2.1. Le tag de déclarations <%! ... %> ... 2642
80.4.2.2. Le tag d'expressions <%= ... %> ... 2643
80.4.2.3. Les variables implicites ... 2644
80.4.2.4. Le tag des scriptlets <% ... %> ... 2644
80.4.3. Les tags de commentaires ... 2645
80.4.3.1. Les commentaires HTML <!-- ... --> ... 2645
80.4.3.2. Les commentaires cachés <%-- ... --%> ... 2646
80.4.4. Les tags d'actions ... 2646
80.4.4.1. Le tag <jsp:useBean> ... 2646
80.4.4.2. Le tag <jsp:setProperty > ... 2649
80.4.4.3. Le tag <jsp:getProperty> ... 2650
80.4.4.4. Le tag de redirection <jsp:forward> ... 2651
80.4.4.5. Le tag <jsp:include> ... 2652
80.4.4.6. Le tag <jsp:plugin> ... 2652
80.5. Un exemple très simple ... 2653
80.6. La gestion des erreurs ... 2654
80.6.1. La définition d'une page d'erreur ... 2654
80.7. Les bibliothèques de tags personnalisés (custom taglibs) ... 2654
80.7.1. La présentation des tags personnalisés ... 2655
80.7.2. Les handlers de tags ... 2655
80.7.3. L'interface Tag ... 2656
80.7.4. L'accès aux variables implicites de la JSP ... 2657
80.7.5. Les deux types de handlers ... 2657
80.7.5.1. Les handlers de tags sans corps ... 2657
80.7.5.2. Les handlers de tags avec corps ... 2658
80.7.6. Les paramètres d'un tag ... 2658
80.7.7. La définition du fichier de description de la bibliothèque de tags (TLD) ... 2658

Table des matières

80. Les JSP (Java Server Pages)
80.7.8. L'utilisation d'une bibliothèque de tags....2660
80.7.8.1. L'utilisation dans le code source d'une JSP....2660
80.7.8.2. Le déploiement d'une bibliothèque....2662
80.7.9. Le déploiement et les tests dans Tomcat....2662
80.7.10. Les bibliothèques de tags existantes....2663
80.7.10.1. Struts....2663
80.7.10.2. JSP Standard Tag Library (JSTL)....2663
80.7.10.3. Apache Taglibs (Jakarta Taglibs)....2663

81. JSTL (Java server page Standard Tag Library)....2664
81.1. Un exemple simple....2665
81.2. Le langage EL (Expression Language)....2666
81.3. La bibliothèque Core....2668
81.3.1. Le tag set....2668
81.3.2. Le tag out....2669
81.3.3. Le tag remove....2670
81.3.4. Le tag catch....2670
81.3.5. Le tag if....2671
81.3.6. Le tag choose....2672
81.3.7. Le tag forEach....2672
81.3.8. Le tag forTokens....2674
81.3.9. Le tag import....2675
81.3.10. Le tag redirect....2676
81.3.11. Le tag url....2676
81.4. La bibliothèque XML....2676
81.4.1. Le tag parse....2677
81.4.2. Le tag set....2678
81.4.3. Le tag out....2678
81.4.4. Le tag if....2679
81.4.5. Le tag choose....2679
81.4.6. Le tag forEach....2679
81.4.7. Le tag transform....2680
81.5. La bibliothèque I18n....2680
81.5.1. Le tag bundle....2681
81.5.2. Le tag setBundle....2682
81.5.3. Le tag message....2682
81.5.4. Le tag setLocale....2683
81.5.5. Le tag formatNumber....2683
81.5.6. Le tag parseNumber....2684
81.5.7. Le tag formatDate....2684
81.5.8. Le tag parseDate....2684
81.5.9. Le tag setTimeZone....2685
81.5.10. Le tag timeZone....2685
81.6. La bibliothèque Database....2685
81.6.1. Le tag setDataSource....2686
81.6.2. Le tag query....2686
81.6.3. Le tag transaction....2688
81.6.4. Le tag update....2688

82. Struts....2689
82.1. L'installation et la mise en oeuvre....2690
82.1.1. Un exemple très simple....2692
82.2. Le développement des vues....2696
82.2.1. Les objets de type ActionForm....2697
82.2.2. Les objets de type DynaActionForm....2698
82.3. La configuration de Struts....2699
82.3.1. Le fichier struts-config.xml....2699
82.3.2. La classe ActionMapping....2701
82.3.3. Le développement de la partie contrôleur....2701

Table des matières

82. Struts

82.3.4. La servlet de type ActionServlet....2702
82.3.5. La classe Action....2702
82.3.6. La classe DispatchAction....2705
82.3.7. La classe LookupDispatchAction....2707
82.3.8. La classe ForwardAction....2710
82.4. Les bibliothèques de tags personnalisés....2710
82.4.1. La bibliothèque de tags HTML....2711
82.4.1.1. Le tag <html:html>....2712
82.4.1.2. Le tag <html:form>....2712
82.4.1.3. Le tag <html:button>....2713
82.4.1.4. Le tag <html:cancel>....2713
82.4.1.5. Le tag <html:submit>....2714
82.4.1.6. Le tag <html:radio>....2714
82.4.1.7. Le tag <html:checkbox>....2714
82.4.2. La bibliothèque de tags Bean....2714
82.4.2.1. Le tag <bean:cookie>....2715
82.4.2.2. Le tag <bean:define>....2715
82.4.2.3. Le tag <bean:header>....2716
82.4.2.4. Le tag <bean:include>....2716
82.4.2.5. Le tag <bean:message>....2716
82.4.2.6. Le tag <bean:page>....2717
82.4.2.7. Le tag <bean:param>....2717
82.4.2.8. Le tag <bean:resource>....2717
82.4.2.9. Le tag <bean:size>....2717
82.4.2.10. Le tag <bean:struts>....2717
82.4.2.11. Le tag <bean:write>....2718
82.4.3. La bibliothèque de tags Logic....2719
82.4.3.1. Les tags <logic:empty> et <logic:notEmpty>....2720
82.4.3.2. Les tags <logic:equal> et <logic:notEqual>....2720
82.4.3.3. Les tags <logic:lessEqual>, <logic:lessThan>, <logic:greaterEqual>, et <logic:greaterThan>....2721
82.4.3.4. Les tags <logic:match> et <logic:notMatch>....2721
82.4.3.5. Les tags <logic:present> et <logic:notPresent>....2722
82.4.3.6. Le tag <logic:forward>....2722
82.4.3.7. Le tag <logic:redirect>....2722
82.4.3.8. Le tag <logic:iterate>....2723
82.5. La validation de données....2723
82.5.1. La classe ActionError....2723
82.5.2. La classe ActionErrors....2724
82.5.3. L'affichage des messages d'erreur....2725
82.5.4. Les classes ActionMessage et ActionMessages....2728
82.5.5. L'affichage des messages....2728

83. JSF (Java Server Faces)....2729
83.1. La présentation de JSF....2729
83.2. Le cycle de vie d'une requête....2731
83.3. Les implémentations....2731
83.3.1. L'implémentation de référence....2731
83.3.2. MyFaces....2732
83.4. Le contenu d'une application....2733
83.5. La configuration de l'application....2734
83.5.1. Le fichier web.xml....2734
83.5.2. Le fichier faces-config.xml....2736
83.6. Les beans....2737
83.6.1. Les beans managés (managed bean)....2737
83.6.2. Les expressions de liaison de données d'un bean....2738
83.6.3. Les Backing beans....2740
83.7. Les composants pour les interfaces graphiques....2741
83.7.1. Le modèle de rendu des composants....2742

Table des matières

83. JSF (Java Server Faces)

83.7.2. L'utilisation de JSF dans une JSP....2742
83.8. La bibliothèque de tags Core....2743
83.8.1. Le tag <selectItem>....2743
83.8.2. Le tag <selectItems>....2744
83.8.3. Le tag <verbatim>....2745
83.8.4. Le tag <attribute>....2746
83.8.5. Le tag <facet>....2746
83.9. La bibliothèque de tags Html....2746
83.9.1. Les attributs communs....2747
83.9.2. Le tag <form>....2750
83.9.3. Les tags <inputText>, <inputTextarea>, <inputSecret>....2750
83.9.4. Le tag <ouputText> et <outputFormat>....2751
83.9.5. Le tag <graphicImage>....2752
83.9.6. Le tag <inputHidden>....2753
83.9.7. Le tag <commandButton> et <commandLink>....2753
83.9.8. Le tag <ouputLink>....2755
83.9.9. Les tags <selectBooleanCheckbox> et <selectManyCheckbox>....2755
83.9.10. Le tag <selectOneRadio>....2757
83.9.11. Le tag <selectOneListbox>....2759
83.9.12. Le tag <selectManyListbox>....2760
83.9.13. Le tag <selectOneMenu>....2762
83.9.14. Le tag <selectManyMenu>....2762
83.9.15. Les tags <message> et <messages>....2763
83.9.16. Le tag <panelGroup>....2764
83.9.17. Le tag <panelGrid>....2764
83.9.18. Le tag <dataTable>....2765
83.10. La gestion et le stockage des données....2771
83.11. La conversion des données....2771
83.11.1. Le tag <convertNumber>....2772
83.11.2. Le tag <convertDateTime>....2773
83.11.3. L'affichage des erreurs de conversions....2774
83.11.4. L'écriture de convertisseurs personnalisés....2775
83.12. La validation des données....2775
83.12.1. Les classes de validation standard....2775
83.12.2. Contourner la validation....2776
83.12.3. L'écriture de classes de validation personnalisées....2777
83.12.4. La validation à l'aide de bean....2779
83.12.5. La validation entre plusieurs composants....2780
83.12.6. L'écriture de tags pour un convertisseur ou un validateur de données....2782
83.12.6.1. L'écriture d'un tag personnalisé pour un convertisseur....2783
83.12.6.2. L'écriture d'un tag personnalisé pour un validateur....2783
83.13. La sauvegarde et la restauration de l'état....2783
83.14. Le système de navigation....2784
83.15. La gestion des événements....2785
83.15.1. Les événements liés à des changements de valeur....2786
83.15.2. Les événements liés à des actions....2788
83.15.3. L'attribut immediate....2789
83.15.4. Les événements liés au cycle de vie....2790
83.16. Le déploiement d'une application....2791
83.17. Un exemple d'application simple....2792
83.18. L'internationalisation....2795
83.19. Les points faibles de JSF....2799

84. D'autres frameworks pour les applications web....2801
84.1. Les frameworks pour les applications web....2801
84.1.1. Tapestry....2801
84.1.2. Spring MVC....2801
84.1.3. Play Framework....2801
84.1.4. Wicket....2802

Table des matières

84. D'autres frameworks pour les applications web

84.1.5. ZK....2802

84.2. Les moteurs de templates....2802

84.2.1. WebMacro....2803

84.2.2. FreeMarker....2803

84.2.3. Velocity....2803

84.2.4. StringTemplate....2803

Partie 12 : Le développement d'applications RIA / RDA....2804

85. Les applications riches de type RIA et RDA....2805

85.1. Les applications de type RIA....2806

85.2. Les applications de type RDA....2806

85.3. Les contraintes....2806

85.4. Les solutions RIA....2807

85.4.1. Les solutions RIA reposant sur Java....2807

85.4.1.1. Java FX....2807

85.4.1.2. Google GWT....2808

85.4.1.3. ZK....2808

85.4.1.4. Echo....2808

85.4.1.5. Apache Wicket....2809

85.4.1.6. Les composants JSF....2809

85.4.1.7. Tibco General Interface....2809

85.4.1.8. Eclipse RAP....2809

85.4.2. Les autres solutions RIA....2810

85.4.2.1. Adobe/Apache Flex....2810

85.4.2.2. Microsoft Silverlight....2810

85.4.2.3. Google Gears....2810

85.4.3. Une comparaison entre GWT et Flex....2811

85.5. Les solutions RDA....2811

85.5.1. Adobe AIR....2811

85.5.2. Eclipse RCP....2811

85.5.3. Netbeans RCP....2812

86. Les applets....2813

86.1. L'intégration d'applets dans une page HTML....2813

86.2. Les méthodes des applets....2814

86.2.1. La méthode init()....2814

86.2.2. La méthode start()....2814

86.2.3. La méthode stop()....2815

86.2.4. La méthode destroy()....2815

86.2.5. La méthode update()....2815

86.2.6. La méthode paint()....2815

86.2.7. Les méthodes size() et getSize()....2816

86.2.8. Les méthodes getCodeBase() et getDocumentBase()....2816

86.2.9. La méthode showStatus()....2816

86.2.10. La méthode getAppletInfo()....2817

86.2.11. La méthode getParameterInfo()....2817

86.2.12. La méthode getGraphics()....2817

86.2.13. La méthode getAppletContext()....2817

86.2.14. La méthode setStub()....2817

86.3. Les interfaces utiles pour les applets....2817

86.3.1. L'interface Runnable....2818

86.3.2. L'interface ActionListener....2818

86.3.3. L'interface MouseListener pour répondre à un clic de souris....2818

86.4. La transmission de paramètres à une applet....2819

86.5. Les applets et le multimédia....2819

86.5.1. L'insertion d'images....2819

86.5.2. L'utilisation des capacités audio....2820

86.5.3. L'animation d'un logo....2821

Table des matières

86. Les applets
86.6. Une applet pouvant s'exécuter comme une application....2822
86.7. Les droits des applets....2823

87. Java Web Start (JWS)....2824
87.1. La création du package de l'application....2825
87.2. La signature d'un fichier jar....2825
87.3. Le fichier JNLP....2826
87.4. La configuration du serveur web....2827
87.5. Le fichier HTML....2827
87.6. Le test de l'application....2828
87.7. L'utilisation du gestionnaire d'applications....2830
87.7.1. Le lancement d'une application....2831
87.7.2. L'affichage de la console....2832
87.7.3. Consigner les traces d'exécution dans un fichier de log....2832
87.8. L'API de Java Web Start....2832

88. AJAX....2833
88.1. La présentation d'AJAX....2834
88.2. Le détail du mode de fonctionnement....2836
88.3. Un exemple simple....2838
88.3.1. L'application de tests....2838
88.3.2. La prise en compte de l'événement déclencheur....2840
88.3.3. La création d'un objet de type XMLHttpRequest pour appeler la servlet....2840
88.3.4. L'exécution des traitements et le renvoi de la réponse par la servlet....2842
88.3.5. L'exploitation de la réponse....2843
88.3.6. L'exécution de l'application....2844
88.4. Des frameworks pour mettre en oeuvre AJAX....2845
88.4.1. Direct Web Remoting (DWR)....2845
88.4.1.1. Un exemple de mise en oeuvre de DWR....2846
88.4.1.2. Le fichier DWR.xml....2849
88.4.1.3. Les scripts engine.js et util.js....2852
88.4.1.4. Les scripts client générés....2853
88.4.1.5. Un exemple pour obtenir le contenu d'une page....2854
88.4.1.6. Un exemple pour valider des données....2855
88.4.1.7. Un exemple pour remplir dynamiquement une liste déroulante....2856
88.4.1.8. Un exemple pour afficher dynamiquement des informations....2858
88.4.1.9. Un exemple pour mettre à jour des données....2860
88.4.1.10. Un exemple pour remplir dynamiquement un tableau de données....2862

89. GWT (Google Web Toolkit)....2866
89.1. La présentation de GWT....2867
89.1.1. L'installation de GWT....2868
89.1.2. GWT version 1.6....2869
89.1.2.1. La nouvelle structure pour les projets....2869
89.1.2.2. Un nouveau système de gestion des événements....2870
89.1.2.3. De nouveaux composants....2871
89.1.3. GWT version 1.7....2871
89.2. La création d'une application....2871
89.2.1. L'application générée....2873
89.2.1.1. Le fichier MonApp.html....2874
89.2.1.2. Le fichier MonApp.gwt.xml....2874
89.2.1.3. Le fichier MonApp.java....2875
89.3. Les modes d'exécution....2876
89.3.1. Le mode hôte (hosted mode)....2876
89.3.2. Le mode web (web mode)....2877
89.4. Les éléments de GWT....2878
89.4.1. Le compilateur....2879
89.4.2. JRE Emulation Library....2879
89.4.3. Les modules....2880

Table des matières

89. GWT (Google Web Toolkit)

89.4.4. Les limitations....2881
89.5. L'interface graphique des applications GWT....2881
89.6. La personnalisation de l'interface....2882
89.7. Les composants (widgets)....2884
89.7.1. Les composants pour afficher des éléments....2885
89.7.1.1. Le composant Image....2885
89.7.1.2. Le composant Label....2886
89.7.1.3. Le composant DialogBox....2887
89.7.2. Les composants cliquables....2888
89.7.2.1. La classe Button....2888
89.7.2.2. La classe PushButton....2888
89.7.2.3. La classe ToggleButton....2889
89.7.2.4. La classe CheckBox....2889
89.7.2.5. La classe RadioButton....2889
89.7.2.6. Le composant HyperLink....2891
89.7.3. Les composants de saisie de texte....2891
89.7.3.1. Le composant TextBoxBase....2891
89.7.3.2. Le composant PasswordTextBox....2891
89.7.3.3. Le composant TextArea....2892
89.7.3.4. Le composant TextBox....2892
89.7.3.5. Le composant RichTextArea....2893
89.7.4. Les composants de sélection de données....2895
89.7.4.1. Le composant ListBox....2895
89.7.4.2. Le composant SuggestBox....2898
89.7.4.3. Le composant DateBox....2899
89.7.4.4. Le composant DatePicker....2899
89.7.5. Les composants HTML....2899
89.7.5.1. Le composant Frame....2900
89.7.5.2. Le composant HTML....2900
89.7.5.3. FileUpload....2900
89.7.5.4. Hidden....2900
89.7.6. Le composant Tree....2901
89.7.7. Les menus....2905
89.7.8. Le composant TabBar....2907
89.8. Les panneaux (panels)....2909
89.8.1. La classe Panel....2910
89.8.2. La classe RootPanel....2911
89.8.3. La classe SimplePanel....2911
89.8.4. La classe ComplexPanel....2912
89.8.5. La classe FlowPanel....2912
89.8.6. La classe DeckPanel....2913
89.8.7. La classe TabPanel....2914
89.8.8. La classe FocusPanel....2914
89.8.9. La classe HTMLPanel....2914
89.8.10. La classe FormPanel....2915
89.8.11. La classe CellPanel....2915
89.8.12. La classe DockPanel....2915
89.8.13. La classe HorizontalPanel....2916
89.8.14. La classe VerticalPanel....2916
89.8.15. La classe CaptionPanel....2917
89.8.16. La classe PopupPanel....2917
89.8.17. La classe DialogBox....2919
89.8.18. La classe DisclosurePanel....2919
89.8.19. La classe AbsolutePanel....2919
89.8.20. La classe StackPanel....2920
89.8.21. La classe ScrollPanel....2920
89.8.22. La classe FlexTable....2921
89.8.23. La classe Frame....2922
89.8.24. La classe Grid....2922

Table des matières

89. GWT (Google Web Toolkit)

89.8.25. La classe HorizontalSplitPanel 2923
89.8.26. La classe VerticalSplitPanel 2924
89.8.27. La classe HTMLTable 2924
89.8.28. La classe LazyPanel 2924
89.9. La création d'éléments réutilisables 2925
89.9.1. La création de composants personnalisés 2925
89.9.2. La création de modules réutilisables 2925
89.10. Les événements 2925
89.11. JSNI 2926
89.12. La configuration et l'internationalisation 2928
89.12.1. La configuration 2928
89.12.2. L'internationalisation 2929
89.13. L'appel de procédures distantes (Remote Procedure Call) 2931
89.13.1. GWT-RPC 2931
89.13.1.1. Une mise oeuvre avec un exemple simple 2931
89.13.1.2. La transmission d'objets lors des appels aux services 2938
89.13.1.3. L'invocation périodique d'un service 2941
89.13.2. L'objet RequestBuilder 2943
89.13.3. JavaScript Object Notation (JSON) 2943
89.14. La manipulation des documents XML 2943
89.15. La gestion de l'historique sur le navigateur 2944
89.16. Les tests unitaires 2944
89.17. Le déploiement d'une application 2946
89.18. Des composants tiers 2946
89.18.1. GWT-Dnd 2946
89.18.2. MyGWT 2946
89.18.3. GWT-Ext 2946
89.18.3.1. Installation et configuration 2947
89.18.3.2. La classe Panel 2947
89.18.3.3. La classe GridPanel 2948
89.19. Les ressources relatives à GWT 2951

Partie 13 : Le développement d'applications avec Spring 2952

90. Spring 2953
90.1. Le but et les fonctionnalités proposées par Spring 2953
90.2. L'historique de Spring 2954
90.3. Spring Framework 2955
90.4. Les projets du portfolio Spring 2956
90.5. Les avantages et les inconvénients de Spring 2957
90.6. Spring et Java EE 2958

91. Spring Core 2959
91.1. Les fondements de Spring 2959
91.1.1. L'inversion de contrôle 2959
91.1.2. La programmation orientée aspects (AOP) 2960
91.1.3. Les beans Spring 2960
91.2. Le conteneur Spring 2960
91.2.1. L'interface BeanFactory 2961
91.2.2. L'interface ApplicationContext 2961
91.2.3. Obtenir une instance du conteneur 2962
91.2.3.1. Obtenir une instance du conteneur de type BeanFactory 2962
91.2.3.2. Obtenir une instance du conteneur de type ApplicationContext 2962
91.2.4. Obtenir une instance d'un bean par le conteneur 2964
91.3. Le fichier de configuration 2964
91.3.1. La définition d'un objet dans le fichier de configuration 2965
91.3.2. La portée des beans (scope) 2966
91.3.3. Les callbacks liés au cycle de vie des beans 2967
91.3.4. L'initialisation tardive 2967

Table des matières

91. Spring Core

91.3.5. L'utilisation de valeurs issues d'un fichier de propriétés....2968
91.3.6. Les espaces de nommage....2968
91.3.6.1. L'espace de nommage beans....2968
91.3.6.2. L'espace de nommage P....2970
91.3.6.3. L'espace de nommage jee....2971
91.3.6.4. L'espace de nommage lang....2972
91.3.6.5. L'espace de nommage context....2973
91.3.6.6. L'espace de nommage util....2975
91.4. L'injection de dépendances....2978
91.4.1. L'injection de dépendances par le constructeur....2979
91.4.2. L'injection de dépendances par un setter....2980
91.4.3. Le passage des valeurs pour les paramètres....2980
91.4.4. Le choix entre injection par constructeur ou par setter....2981
91.4.5. L'autowiring....2981
91.5. Spring Expression Langage (SpEL)....2982
91.5.1. La mise en oeuvre de SpEL dans la définition du contexte....2982
91.5.2. L'utilisation de l'API....2983
91.5.3. Des exemples de mise en oeuvre....2985
91.5.4. La syntaxe de SpEL....2987
91.5.4.1. Les types de base....2987
91.5.4.2. L'utilisation d'objets....2987
91.5.4.3. Les opérateurs....2988
91.5.4.4. L'utilisation de types....2989
91.5.4.5. L'invocation d'un constructeur ou d'une méthode....2990
91.5.4.6. L'utilisation d'expressions régulières....2990
91.5.4.7. La manipulation de collections....2991
91.6. La configuration en utilisant les annotations....2993
91.6.1. L'annotation @Scope....2993
91.6.2. L'injection de dépendances avec les annotations....2994
91.6.2.1. L'annotation @Required....2995
91.6.2.2. L'annotation @Autowired....2995
91.6.2.3. L'annotation @Qualifier....2999
91.6.2.4. L'annotation @Resource....3004
91.6.2.5. L'annotation @Configurable....3005
91.6.2.6. Exemple d'injection de dépendance avec @Configurable....3008
91.6.2.7. Les annotations relatives à la gestion du cycle de vie....3009
91.6.2.8. L'annotation @PostConstruct....3009
91.6.2.9. L'annotation @PreDestroy....3010
91.6.3. Les annotations concernant les stéréotypes....3010
91.6.3.1. Les stéréotypes Spring....3010
91.6.3.2. La recherche des composants....3011
91.6.3.3. L'annotation @Component....3012
91.6.3.4. L'annotation @Repository....3012
91.6.3.5. L'annotation @Service....3012
91.6.3.6. L'annotation @Controller....3013
91.6.4. Le remplacement de la configuration par des annotations....3013
91.6.5. Le support de la JSR 330....3013
91.6.5.1. L'annotation @Inject....3014
91.6.5.2. L'annotation @Qualifier....3014
91.6.5.3. L'annotation @Named....3018
91.6.5.4. Le choix entre les annotations de Spring et celles de la JSR 330....3019
91.6.5.5. Le remplacement des annotations de Spring par celles de la JSR 330....3020
91.6.6. La configuration grâce aux annotations....3020
91.6.6.1. L'annotation @Configuration....3022
91.6.6.2. L'annotation @Bean....3022
91.6.6.3. L'annotation @DependsOn....3023
91.6.6.4. L'annotation @Primary....3024
91.6.6.5. L'annotation @Lazy....3024
91.6.6.6. L'annotation @Import....3024

Table des matières

91. Spring Core

91.6.6.7. L'annotation @ImportResource ... 3024
91.6.6.8. L'annotation @Value ... 3024
91.7. Le scheduling ... 3024
91.7.1. La définition dans le fichier de configuration du context ... 3025
91.7.2. La définition grâce aux annotations ... 3026
91.7.2.1. La définition grâce à l'annotation @Scheduled ... 3027
91.7.3. L'invocation de méthodes de manière asynchrone ... 3028

92. La mise en oeuvre de l'AOP avec Spring ... 3032
92.1. Spring AOP ... 3033
92.1.1. Les différents types d'advices ... 3033
92.1.2. Spring AOP sans les annotations AspectJ ... 3034
92.1.2.1. La définition de l'aspect ... 3034
92.1.2.2. La déclaration de l'aspect ... 3035
92.1.2.3. Le namespace aop ... 3036
92.1.2.4. Une autre implémentation de l'aspect ... 3041
92.1.2.5. La gestion de l'ordre des aspects ... 3043
92.1.3. Spring AOP avec les annotations AspectJ ... 3045
92.1.3.1. La gestion de l'ordre des aspects ... 3049
92.2. AspectJ ... 3053
92.2.1. AspectJ avec LTW (Load Time Weaving) ... 3053
92.3. Spring AOP et AspectJ ... 3055
92.4. L'utilisation des namespaces ... 3059
92.4.1. L'utilisation du tag <context:load-time-weaver> ... 3059

93. La gestion des transactions avec Spring ... 3061
93.1. La gestion des transactions par Spring ... 3061
93.2. La propagation des transactions ... 3062
93.3. L'utilisation des transactions de manière déclarative ... 3062
93.3.1. La déclaration des transactions dans la configuration du contexte ... 3063
93.3.2. Un exemple de déclaration de transactions dans la configuration ... 3065
93.4. La déclaration des transactions avec des annotations ... 3069
93.4.1. L'utilisation de l'annotation @Transactional ... 3070
93.4.2. Le support de @Transactional par AspectJ ... 3072
93.4.3. Un exemple de déclaration des transactions avec des annotations ... 3072
93.5. La gestion du rollback des transactions ... 3076
93.5.1. La gestion du rollback dans la configuration ... 3076
93.5.2. La gestion du rollback via l'API ... 3077
93.5.3. La gestion du rollback avec les annotations ... 3077
93.6. La mise en oeuvre d'aspects sur une méthode transactionnelle ... 3077
93.7. L'utilisation des transactions via l'API ... 3080
93.7.1. L'utilisation de la classe TransactionTemplate ... 3080
93.7.2. L'utilisation directe d'un PlatformTransactionManager ... 3083
93.8. L'utilisation d'un gestionnaire de transactions reposant sur JTA ... 3084

94. Spring et JMS ... 3085
94.1. Les packages de Spring JMS ... 3085
94.2. La classe JmsTemplate : le template JMS de Spring ... 3086
94.2.1. L'envoi d'un message avec JmsTemplate ... 3086
94.2.2. La réception d'un message avec JmsTemplate ... 3087
94.2.3. La mise en oeuvre dans une application ... 3087
94.2.4. La classe CachingConnectionFactory ... 3089
94.3. La réception asynchrone de messages ... 3091
94.3.1. La classe DefaultMessageListenerContainer ... 3092
94.3.2. L'amélioration des performances de la consommation des messages ... 3094
94.4. L'espace de nommage jms ... 3096

Table des matières

95. Spring et JMX......3098
95.1. L'enregistrement d'un bean en tant que MBean......3098
95.1.1. La classe MBeanExporter......3099
95.1.2. La création d'un MBeanServer......3101
95.1.3. L'accès distant au serveur de MBeans......3102
95.1.4. Les listeners d'un Exporter......3104
95.2. Le nommage des MBeans......3105
95.2.1. Les stratégies de nommage des MBeans......3105
95.2.1.1. L'interface ObjectNamingStrategy......3105
95.2.1.2. La classe IdentityNamingStrategy......3105
95.2.1.3. La classe KeyNamingStrategy......3106
95.2.1.4. La classe MetadataNamingStrategy......3107
95.2.2. L'interface SelfNaming......3108
95.3. Les Assembler......3108
95.3.1. La classe MethodNameBasedMBeanInfoAssembler......3108
95.3.2. La classe MethodExclusionMBeanInfoAssembler......3110
95.3.3. La classe InterfaceBasedMBeanInfoAssembler......3111
95.3.4. La classe MetadataBasedMBeanInfoAssembler......3114
95.3.4.1. L'utilisation d'attributs comme métadonnées......3114
95.3.4.2. L'utilisation d'annotations comme métadonnées......3117
95.4. L'utilisation des annotations......3117
95.4.1. L'annotation @ManagedResource......3119
95.4.2. L'annotation @ManagedAttribute......3121
95.4.3. L'annotation @ManagedOperation......3122
95.4.4. Les annotations @ManagedOperationParameters et @ManagedOperationParameter......3123
95.4.5. L'annotation @ManagedMetric......3123
95.4.6. Les annotations @ManagedNotifications et @ManagedNotification......3125
95.4.7. L'utilisation du tag <mbean-server>......3126
95.4.8. L'utilisation du tag <mbean-export>......3126
95.5. Le développement d'un client JMX......3127
95.5.1. La connexion à un serveur de MBeans......3127
95.5.2. L'utilisation d'un proxy......3129
95.6. Les notifications......3131
95.6.1. L'émission de notifications......3131
95.6.2. La réception de notifications......3132

Partie 14 : Les outils pour le développement......3134

96. Les outils du J.D.K.......3136
96.1. Le compilateur javac......3136
96.1.1. La syntaxe de javac......3136
96.1.2. Les options de javac......3137
96.1.3. Les principales erreurs de compilation......3137
96.2. L'interpréteur java/javaw......3144
96.2.1. La syntaxe de l'outil java......3144
96.2.2. Les options de l'outil java......3145
96.3. L'outil jar......3145
96.3.1. L'intérêt du format jar......3145
96.3.2. La syntaxe de l'outil jar......3146
96.3.3. La création d'une archive jar......3147
96.3.4. Lister le contenu d'une archive jar......3147
96.3.5. L'extraction du contenu d'une archive jar......3148
96.3.6. L'utilisation des archives jar......3148
96.3.7. Le fichier manifest......3149
96.3.8. La signature d'une archive jar......3149
96.4. L'outil appletviewer pour tester des applets......3150
96.5. L'outil javadoc pour générer la documentation technique......3150
96.6. L'outil Java Check Update pour mettre à jour Java......3153
96.7. La base de données Java DB......3155
96.8. L'outil JConsole......3159

Table des matières

97. JavaDoc......3163
97.1. La mise en oeuvre......3163
97.2. Les tags définis par javadoc......3165
97.2.1. Le tag @author......3166
97.2.2. Le tag @deprecated......3166
97.2.3. Le tag @exception et @throws......3167
97.2.4. Le tag @param......3168
97.2.5. Le tag @return......3168
97.2.6. Le tag @see......3169
97.2.7. Le tag @since......3170
97.2.8. Le tag @version......3171
97.2.9. Le tag {@link}......3171
97.2.10. Le tag {@value}......3171
97.2.11. Le tag {@literal}......3172
97.2.12. Le tag {@linkplain}......3172
97.2.13. Le tag {@inheritDoc}......3172
97.2.14. Le tag {@docRoot}......3172
97.2.15. Le tag {@code}......3173
97.2.16. Le tag @snippet......3173
97.2.16.1. Les attributs......3174
97.2.16.2. Les fragments en ligne......3175
97.2.16.3. Les régions......3176
97.2.16.4. Les fragments externes......3177
97.2.16.5. Les fragments externes non Java......3180
97.2.16.6. Les fragments hybrides......3181
97.2.16.7. Les tags de marquage......3183
97.3. Un exemple......3191
97.4. Les fichiers pour enrichir la documentation des packages......3192
97.5. La documentation générée......3192

98. JShell......3199
98.1. Les outils de type REPL......3199
98.1.1. L'utilité d'un outil de type REPL......3200
98.1.2. REPL vs IDE......3200
98.2. Introduction à JShell......3200
98.2.1. Les intérêts à utiliser JShell......3200
98.2.2. L'implémentation de JShell......3201
98.2.3. Lancer et arrêter JShell......3201
98.2.4. La mise en oeuvre de JShell......3203
98.3. La saisie d'éléments dans JShell......3203
98.3.1. Les fonctionnalités de saisie......3204
98.3.2. Le résultat de l'évaluation d'un fragment......3204
98.4. Les fragments......3205
98.4.1. Les variables et les expressions......3206
98.4.2. Les instructions de contrôle de flux......3208
98.4.3. La définition et l'invocation de méthodes......3209
98.4.4. La création de classes et d'instances......3210
98.5. Les commandes de JShell......3211
98.5.1. La commande /help......3213
98.5.2. La commande /exit......3213
98.5.3. La commande /list......3214
98.5.4. La commande /history......3215
98.5.5. La commande /imports......3215
98.5.6. La commande /vars......3216
98.5.7. La commande /methods......3216
98.5.8. La commande /types......3217
98.5.9. La commande /save......3217
98.5.10. La commande /open......3218
98.5.11. La commande /edit......3218
98.5.12. La commande /drop......3221

Table des matières

98. JShell

98.5.13. La commande /<id>, /! et /-<n> ...3222
98.5.14. La commande /set ...3222
98.5.14.1. L'option editor ...3223
98.5.14.2. L'option start ...3223
98.5.15. L'achèvement des commandes ...3224
98.5.16. L'abréviation des commandes ...3228
98.6. L'édition ...3228
98.6.1. La navigation dans la ligne courante ...3229
98.6.2. La navigation dans l'historique ...3229
98.6.3. La modification de la ligne courante ...3230
98.6.4. La recherche et les autres fonctionnalités ...3230
98.6.5. La définition et l'utilisation d'une macro ...3231
98.6.6. L'utilisation d'un éditeur ...3231
98.7. Les fonctionnalités de l'environnement ...3232
98.7.1. L'exploration des API et l'achèvement de code ...3232
98.7.2. Les références à venir (Forward references) ...3234
98.7.3. La redéclaration d'une variable ou d'une méthode ...3235
98.7.4. Les exceptions de type checked ...3236
98.7.5. L'ajout d'un import ...3238
98.7.6. La création d'une variable à partir d'une expression ...3238
98.8. Les scripts ...3239
98.8.1. Les scripts de démarrage ...3239
98.8.2. La création d'un script ...3242
98.8.3. Le chargement d'un script ...3243
98.9. Les modes de feedback ...3243
98.9.1. L'utilisation d'un mode de feedback ...3244
98.9.2. La définition d'un mode de feedback ...3245
98.9.2.1. La définition des prompts ...3245
98.9.2.2. La troncature des valeurs affichées ...3247
98.9.2.3. Le format du retour de l'évaluation ...3248
98.10. L'utilisation de classes externes ...3249
98.10.1. La configuration du classpath ...3249
98.10.2. La configuration du modulepath ...3250
98.11. Les options de JShell ...3250
98.12. JShell API ...3251

99. Les outils libres et commerciaux ...3254
99.1. Les environnements de développement intégrés (IDE) ...3255
99.1.1. Eclipse ...3255
99.1.2. Netbeans ...3257
99.1.3. IntelliJ IDEA ...3259
99.1.4. Oracle JDeveloper ...3260
99.1.5. IBM Rational Application Developer for WebSphere Software ...3262
99.1.6. IBM Rational Team Concert ...3263
99.1.7. MyEclipse ...3263
99.1.8. IBM Websphere Studio Application Developper ...3264
99.1.9. Embarcadero (Borland/CodeGear) JBuilder ...3264
99.1.10. Sun Java Studio Creator ...3265
99.1.11. JCreator ...3266
99.1.12. BEA Workshop ...3266
99.1.13. IBM Visual Age for Java ...3267
99.1.14. Webgain Visual Café ...3267
99.2. Les serveurs d'application ...3267
99.2.1. JBoss Application Server ...3267
99.2.2. JBoss Wildfly ...3268
99.2.3. JOnAs ...3269
99.2.4. GlassFish ...3269
99.2.5. Apache TomEE ...3270
99.2.6. IBM Websphere Application Server ...3271

Table des matières

99. Les outils libres et commerciaux
99.2.7. BEA/Oracle Weblogic....3272
99.2.8. Oracle Application Server....3273
99.2.9. Macromedia JRun....3273
99.3. Les conteneurs web....3273
99.3.1. Apache Tomcat....3273
99.3.2. Caucho Resin....3274
99.3.3. Enhydra....3274
99.4. Les conteneurs d'EJB....3274
99.4.1. OpenEJB....3274
99.5. Les outils divers....3274
99.5.1. Jikes....3274
99.5.2. GNU Compiler for Java....3275
99.5.3. Artistic Style....3277
99.6. Les MOM....3278
99.6.1. Apache ActiveMQ....3278
99.6.2. OpenJMS....3279
99.6.3. Joram....3281
99.6.4. OSMQ....3281
99.7. Les outils concernant les bases de données....3281
99.7.1. Derby....3281
99.7.2. SQuirrel-SQL....3281
99.8. Les outils de modélisation UML....3282
99.8.1. Argo UML....3282
99.8.2. Poseidon for UML....3282
99.8.3. StarUML....3282

100. Ant....3284
100.1. L'installation de l'outil Ant....3285
100.1.1. L'installation sous Windows....3285
100.2. L'exécution de l'outil Ant....3285
100.3. Le fichier build.xml....3286
100.3.1. Le projet....3286
100.3.2. Les commentaires....3287
100.3.3. Les propriétés....3287
100.3.4. Les ensembles de fichiers....3288
100.3.5. Les ensembles de motifs....3288
100.3.6. Les listes de fichiers....3289
100.3.7. Les éléments de chemins....3289
100.3.8. Les cibles....3289
100.4. Les tâches (task)....3290
100.4.1. echo....3291
100.4.2. mkdir....3292
100.4.3. delete....3293
100.4.4. copy....3293
100.4.5. tstamp....3294
100.4.6. java....3295
100.4.7. javac....3295
100.4.8. javadoc....3296
100.4.9. jar....3297

101. Maven....3299
101.1. Les différentes versions de Maven....3299
101.2. Maven 1....3300
101.2.1. L'installation....3300
101.2.2. Les plugins....3301
101.2.3. Le fichier project.xml....3301
101.2.4. L'exécution de Maven....3302
101.2.5. La génération du site du projet....3303
101.2.6. La compilation du projet....3305

Table des matières

101. Maven

101.3. Maven 2 3306
101.3.1. L'installation et la configuration 3306
101.3.2. Les concepts 3307
101.3.2.1. Les artéfacts 3307
101.3.2.2. Convention plutôt que configuration 3307
101.3.2.3. Le cycle de vie et les phases 3308
101.3.2.4. Les archétypes 3309
101.3.2.5. La gestion des dépendances 3309
101.3.2.6. Les dépôts (repositories) 3310
101.3.2.7. La portée des dépendances 3311
101.3.2.8. La transitivité des dépendances 3311
101.3.2.9. La communication sur le projet 3312
101.3.2.10. Les plugins 3312
101.3.3. La création d'un nouveau projet 3315
101.3.4. Le fichier POM 3315
101.3.4.1. L'utilisation et la configuration des plugins 3316
101.3.4.2. Les métadonnées du projet 3317
101.3.5. Le cycle de vie d'un projet 3318
101.3.5.1. L'ajout d'un goal à une phase 3320
101.3.6. La configuration générale de Maven 3321
101.3.6.1. Le fichier settings.xml 3321
101.3.6.2. L'utilisation d'un miroir du dépôt central 3323
101.3.7. La configuration du projet 3324
101.3.7.1. La déclaration des dépendances 3324
101.3.7.2. L'utilisation de profiles 3326
101.3.7.3. L'utilisation des propriétés 3331
101.3.7.4. L'ajout et l'exclusion de ressources dans l'artéfact 3332
101.3.7.5. La compilation d'un projet pour une version particulière de Java 3334
101.3.7.6. La gestion des versions des dépendances dans un POM parent 3334
101.3.7.7. La définition d'un fichier manifest particulier dans un jar 3336
101.3.7.8. L'utilisation de valeurs de propriétés dans les ressources 3337
101.3.7.9. La génération d'un fichier jar contenant les sources du projet 3339
101.3.8. L'exécution de commandes 3340
101.3.8.1. Un résumé des principales commandes 3340
101.3.8.2. Les options de la commande mvn 3341
101.3.8.3. Le nettoyage d'un projet 3341
101.3.8.4. La compilation du code source 3342
101.3.8.5. La compilation du code source des tests et leur exécution 3342
101.3.8.6. La création de l'artéfact 3343
101.3.8.7. L'installation de l'artéfact dans le dépôt local 3343
101.3.8.8. Le déploiement d'un artéfact dans un dépôt distant 3343
101.3.8.9. La génération de la Javadoc 3344
101.3.9. L'utilisation de projets multi-modules 3345

102. Tomcat 3350
102.1. L'historique des versions 3350
102.2. L'installation 3352
102.2.1. L'installation de Tomcat 3.1 sous Windows 98 3352
102.2.2. L'installation de Tomcat 4.0 sur Windows 98 3353
102.2.3. L'installation de Tomcat 5.0 sur Windows 3354
102.2.4. L'installation de Tomcat 5.5 sous Windows avec l'installer 3355
102.2.5. L'installation Tomcat 6.0 sous Windows avec l'installer 3359
102.2.6. La structure des répertoires 3359
102.2.6.1. La structure des répertoires de Tomcat 4 3359
102.2.6.2. La structure des répertoires de Tomcat 5 3359
102.2.6.3. La structure des répertoires de Tomcat 6 3360
102.3. L'exécution de Tomcat 3360
102.3.1. L'exécution sous Windows de Tomcat 4.0 3360
102.3.2. L'exécution sous Windows de Tomcat 5.0 3360

Table des matières

102. Tomcat

102.3.3. La vérification de l'exécution....3363
102.4. L'architecture....3364
102.4.1. Les connecteurs....3365
102.4.2. Les services....3366
102.5. La configuration....3366
102.5.1. Le fichier server.xml....3366
102.5.1.1. Le fichier server.xml avec Tomcat 5....3367
102.5.1.2. Les valves....3369
102.5.2. La gestion des rôles....3369
102.6. L'outil Tomcat Administration Tool....3369
102.6.0.1. La gestion des utilisateurs, des rôles et des groupes....3372
102.6.1. La création d'une DataSource dans Tomcat....3373
102.7. Le déploiement des applications WEB....3374
102.7.1. Déployer une application web avec Tomcat 5....3374
102.7.1.1. Déployer une application au lancement de Tomcat....3374
102.7.1.2. Déployer une application sur Tomcat en cours d'exécution....3375
102.7.1.3. L'utilisation d'un contexte....3375
102.7.1.4. Déployer une application avec le Tomcat Manager....3375
102.7.1.5. Déployer une application avec les tâches Ant du Manager....3375
102.7.1.6. Déployer une application avec le TCD....3375
102.8. Tomcat pour le développeur....3375
102.8.1. Accéder à une ressource par son url....3375
102.8.2. La structure d'une application web et format war....3376
102.8.3. La configuration d'un contexte....3377
102.8.3.1. La configuration d'un contexte avec Tomcat 5....3377
102.8.4. L'invocation dynamique de servlets....3377
102.8.5. Les bibliothèques partagées....3379
102.8.5.1. Les bibliothèques partagées sous Tomcat 5....3379
102.9. Le gestionnaire d'applications (Tomcat manager)....3380
102.9.1. L'utilisation de l'interface graphique....3380
102.9.1.1. Le déploiement d'une application....3383
102.9.1.2. La gestion des applications....3384
102.9.2. L'utilisation des commandes par requêtes HTTP....3385
102.9.2.1. La commande list....3385
102.9.2.2. La commande serverinfo....3386
102.9.2.3. La commande reload....3386
102.9.2.4. La commande resources....3386
102.9.2.5. La commande roles....3387
102.9.2.6. La commande sessions....3387
102.9.2.7. La commande stop....3387
102.9.2.8. La commande start....3388
102.9.2.9. La commande undeploy....3388
102.9.2.10. La commande deploy....3389
102.9.3. L'utilisation du manager avec des tâches Ant....3389
102.9.4. L'utilisation de la servlet JMXProxy....3391
102.10. L'outil Tomcat Client Deployer....3392
102.11. Les optimisations....3393
102.12. La sécurisation du serveur....3393

103. Des outils open source pour faciliter le développement....3394
103.1. CheckStyle....3394
103.1.1. L'installation....3394
103.1.2. L'utilisation avec Ant....3395
103.1.3. L'utilisation en ligne de commandes....3398
103.2. Jalopy....3398
103.2.1. L'utilisation avec Ant....3399
103.2.2. Les conventions....3400

Table des matières

Partie 15 : La conception et le développement des applications....3403

104. Java et UML....3405
104.1. La présentation d'UML....3405
104.2. Les commentaires....3406
104.3. Les cas d'utilisations (use cases)....3406
104.4. Le diagramme de séquence....3407
104.5. Le diagramme de collaboration....3408
104.6. Le diagramme d'états-transitions....3409
104.7. Le diagramme d'activités....3410
104.8. Le diagramme de classes....3410
104.8.1. Les attributs d'une classe....3410
104.8.2. Les méthodes d'une classe....3411
104.8.3. L'implémentation d'une interface....3412
104.8.4. La relation d'héritage....3412
104.9. Le diagramme d'objets....3412
104.10. Le diagramme de composants....3412
104.11. Le diagramme de déploiement....3413

105. Les motifs de conception (design patterns)....3414
105.1. Les modèles de création....3414
105.1.1. Fabrique (Factory)....3415
105.1.2. Fabrique abstraite (abstract Factory)....3418
105.1.3. Monteur (Builder)....3421
105.1.4. Prototype (Prototype)....3422
105.1.5. Singleton (Singleton)....3422
105.2. Les modèles de structuration....3425
105.2.1. Façade (Facade)....3425
105.2.2. Décorateur (Decorator)....3430
105.3. Les modèles de comportement....3434

106. Des normes de développement....3435
106.1. Les fichiers....3435
106.1.1. Les packages....3435
106.1.2. Les noms de fichiers....3436
106.1.3. Le contenu des fichiers sources....3436
106.1.4. Les commentaires de début de fichier....3436
106.1.5. Les clauses concernant les packages.....3436
106.1.6. La déclaration des classes et des interfaces....3437
106.2. La documentation du code....3437
106.2.1. Les commentaires de documentation....3437
106.2.1.1. L'utilisation des commentaires de documentation....3437
106.2.1.2. Les commentaires pour une classe ou une interface....3438
106.2.1.3. Les commentaires pour une méthode....3438
106.2.2. Les commentaires de traitements....3439
106.2.2.1. Les commentaires sur une ligne....3439
106.2.2.2. Les commentaires sur une portion de ligne....3439
106.2.2.3. Les commentaires multi-lignes....3439
106.2.2.4. Les commentaires de fin de ligne....3440
106.3. Les déclarations....3440
106.3.1. La déclaration des variables....3440
106.3.2. La déclaration des classes et des méthodes....3441
106.3.3. La déclaration des constructeurs....3442
106.3.4. Les conventions de nommage des entités....3443
106.4. Les séparateurs....3444
106.4.1. L'indentation....3444
106.4.2. Les lignes blanches....3444
106.4.3. Les espaces....3444
106.4.4. La coupure de lignes....3445
106.5. Les traitements....3445

Table des matières

106. Des normes de développement

106.5.1. Les instructions composées....3445
106.5.2. L'instruction return....3446
106.5.3. L'instruction if....3446
106.5.4. L'instruction for....3446
106.5.5. L'instruction while....3446
106.5.6. L'instruction do-while....3447
106.5.7. L'instruction switch....3447
106.5.8. Les instructions try-catch....3447
106.6. Les règles de programmation....3447
106.6.1. Le respect des règles d'encapsulation....3448
106.6.2. Les références aux variables et méthodes de classes....3448
106.6.3. Les constantes....3448
106.6.4. L'assignement des variables....3448
106.6.5. L'usage des parenthèses....3449
106.6.6. La valeur de retour....3449
106.6.7. La codification de la condition dans l'opérateur ternaire ? :....3449
106.6.8. La déclaration d'un tableau....3449

107. Les techniques de développement spécifiques à Java....3450

107.1. L'écriture d'une classe dont les instances seront immuables....3450
107.2. La redéfinition des méthodes equals() et hashCode()....3454
107.2.1. Les contraintes pour redéfinir equals() et hashCode()....3455
107.2.2. La méthode equals()....3455
107.2.2.1. L'implémentation par défaut de la méthode equals()....3456
107.2.2.2. La redéfinition de la méthode equals()....3457
107.2.2.3. Les contraintes et quelques recommandations....3458
107.2.3. La méthode hashCode()....3460
107.2.3.1. L'implémentation par défaut....3460
107.2.3.2. La redéfinition de la méthode hashCode()....3461
107.2.3.3. Les contraintes et les recommandations....3461
107.2.4. Des exemples de redéfinition des méthodes hashCode() et equals()....3464
107.2.4.1. L'utilisation d'un IDE....3464
107.2.4.2. L'utilisation des helpers de Commons Lang....3465
107.2.5. L'intérêt de redéfinir les méthodes hashCode() et equals()....3466
107.2.5.1. L'utilisation par certaines collections....3467
107.2.5.2. Les performances en définissant correctement la méthode hashCode()....3469
107.2.6. Des implémentations particulières des méthodes hashCode() et equals()....3472
107.2.6.1. Les méthodes hashCode() et equals() dans le JDK....3473
107.2.6.2. Les méthodes equals() et hashCode() dans une classe fille....3473
107.2.6.3. La redéfinition des méthodes equals() et hashCode() pour des entités....3477
107.3. Le clonage d'un objet....3477
107.3.1. L'interface Cloneable....3478
107.3.2. Le clonage par copie de surface....3478
107.3.2.1. La redéfinition de la méthode clone()....3479
107.3.2.2. L'implémentation personnalisée de la méthode clone()....3482
107.3.3. Le clonage par copie profonde....3482
107.3.4. Le clonage en utilisant la sérialisation....3484
107.3.5. Le clonage de tableaux....3486
107.3.6. La duplication d'un objet en utilisant un constructeur ou une fabrique....3487
107.3.6.1. La duplication d'un objet en utilisant un constructeur....3487
107.3.6.2. La duplication d'un objet en utilisant une fabrique....3488
107.3.7. La duplication en utilisant des bibliothèques tierces....3489
107.3.7.1. La duplication en utilisant Apache Commons....3489
107.3.7.2. La duplication en utilisant Kryo....3489
107.3.7.3. La duplication en utilisant XStream....3490

108. L'encodage des caractères....3493

108.1. L'utilisation des caractères dans la JVM....3493
108.1.1. Le stockage des caractères dans la JVM....3493

Table des matières

108. L'encodage des caractères

108.1.2. L'encodage des caractères par défaut....3494

108.2. Les jeux d'encodages de caractères....3494

108.3. Unicode....3494

108.3.1. L'encodage des caractères Unicode....3494

108.3.2. Le marqueur optionnel BOM....3495

108.4. L'encodage de caractères....3496

108.4.1. Les classes du package java.lang....3497

108.4.2. Les classes du package java.io....3497

108.4.3. Le package java.nio....3498

108.5. L'encodage du code source....3498

108.6. L'encodage de caractères avec différentes technologies....3499

108.6.1. L'encodage de caractères dans les fichiers....3499

108.6.2. L'encodage de caractères dans une application web....3500

108.6.3. L'encodage de caractères avec JDBC....3500

109. Les frameworks....3501

109.1. La présentation des concepts....3501

109.1.1. La définition d'un framework....3501

109.1.2. L'utilité de mettre en oeuvre des frameworks....3502

109.1.3. Les différentes catégories de framework....3503

109.1.4. Les socles techniques....3503

109.2. Les frameworks pour les applications web....3504

109.3. L'architecture pour les applications web....3504

109.3.1. Le modèle MVC....3504

109.4. Le modèle MVC type 1....3505

109.5. Le modèle MVC de type 2....3505

109.5.1. Les différents types de framework web....3506

109.5.2. Des frameworks pour le développement web....3507

109.5.2.1. Apache Struts....3507

109.5.2.2. Spring MVC....3508

109.5.2.3. Tapestry....3508

109.5.2.4. Java Server Faces....3508

109.5.2.5. Struts 2....3509

109.5.2.6. Struts Shale....3509

109.5.2.7. Expresso....3509

109.5.2.8. Jena....3510

109.5.2.9. Turbine....3510

109.5.2.10. Wicket....3510

109.6. Les frameworks de mapping Objet/Relationel....3510

109.7. Les frameworks de logging....3510

110. La génération de documents....3511

110.1. Apache POI....3511

110.1.1. POI-HSSF....3512

110.1.1.1. L'API de type usermodel....3512

110.1.1.2. L'API de type eventusermodel....3529

110.2. iText....3529

110.2.1. Un exemple très simple....3529

110.2.2. L'API de iText....3530

110.2.3. La création d'un document....3530

110.2.3.1. La classe Document....3531

110.2.3.2. Les objets de type DocWriter....3534

110.2.4. L'ajout de contenu au document....3536

110.2.4.1. Les polices de caractères....3536

110.2.4.2. Le classe Chunk....3540

110.2.4.3. La classe Phrase....3542

110.2.4.4. La classe Paragraph....3544

110.2.4.5. La classe Chapter....3546

110.2.4.6. La classe Section....3546

Table des matières

110. La génération de documents
110.2.4.7. La création d'une nouvelle page 3547
110.2.4.8. La classe Anchor 3549
110.2.4.9. Les classes List et ListItem 3549
110.2.4.10. La classe Table 3551
110.2.5. Des fonctionnalités avancées 3554
110.2.5.1. Insérer une image 3554

111. La validation des données 3556
111.1. Quelques recommandations sur la validation des données 3557
111.2. L'API Bean Validation (JSR 303) 3557
111.2.1. La présentation de l'API 3558
111.2.1.1. Les objectifs de l'API 3558
111.2.1.2. Les éléments et concepts utilisés par l'API 3558
111.2.1.3. Les contraintes et leur validation avec l'API 3559
111.2.1.4. La mise en oeuvre générale de l'API 3559
111.2.1.5. Un exemple simple de mise en oeuvre 3560
111.2.2. La déclaration des contraintes 3561
111.2.2.1. La déclaration des contraintes sur les champs 3562
111.2.2.2. La déclaration des contraintes sur les propriétés 3563
111.2.2.3. La déclaration des contraintes sur une classe 3564
111.2.2.4. L'héritage de contraintes 3564
111.2.2.5. Les contraintes de validation d'un ensemble d'objets 3565
111.2.3. La validation des contraintes 3567
111.2.3.1. L'obtention d'un valideur 3568
111.2.3.2. L'interface Validator 3569
111.2.3.3. L'utilisation d'un valideur 3569
111.2.3.4. L'interface ConstraintViolation 3571
111.2.3.5. La mise en oeuvre des groupes 3571
111.2.3.6. Définir et utiliser un groupe implicite 3574
111.2.3.7. La définition de l'ordre des validations 3576
111.2.3.8. La redéfinition du groupe par défaut 3576
111.2.4. Les contraintes standard 3576
111.2.4.1. L'annotation @Null 3578
111.2.4.2. L'annotation @NotNull 3578
111.2.4.3. L'annotation @AssertTrue 3578
111.2.4.4. L'annotation @AssertFalse 3580
111.2.4.5. L'annotation @Min 3580
111.2.4.6. L'annotation @Max 3581
111.2.4.7. L'annotation @DecimalMin 3581
111.2.4.8. L'annotation @DecimalMax 3582
111.2.4.9. L'annotation @Size 3583
111.2.4.10. L'annotation @Digits 3584
111.2.4.11. L'annotation @Past 3584
111.2.4.12. L'annotation @Future 3585
111.2.4.13. L'annotation @Pattern 3585
111.2.5. Le développement de contraintes personnalisées 3586
111.2.5.1. La création de l'annotation 3586
111.2.5.2. La création de la classe de validation 3590
111.2.5.3. Le message d'erreur 3593
111.2.5.4. L'utilisation d'une contrainte 3594
111.2.5.5. Application multiple d'une contrainte 3595
111.2.6. Les contraintes composées 3596
111.2.7. L'interpolation des messages 3599
111.2.7.1. L'algorithme d'une interpolation par défaut 3600
111.2.7.2. Le développement d'un MessageInterpolator spécifique 3600
111.2.8. Bootstrapping 3601
111.2.8.1. L'utilisation du Java Service Provider 3601
111.2.8.2. L'utilisation de la classe Validation 3601
111.2.8.3. Les interfaces ValidationProvider et ValidationProviderResolver 3603

Table des matières

111. La validation des données
111.2.8.4. L'interface MessageInterpolator....3604
111.2.8.5. L'interface TraversableResolver....3604
111.2.8.6. L'interface ConstraintValidatorFactory....3605
111.2.8.7. L'interface ValidatorFactory....3605
111.2.8.8. L'interface Configuration....3605
111.2.8.9. Le fichier de configuration META-INF/validation.xml....3607
111.2.9. La définition de contraintes dans un fichier XML....3607
111.2.10. L'API de recherche des contraintes....3607
111.2.10.1. L'interface ElementDescriptor....3608
111.2.10.2. L'interface ConstraintFinder....3608
111.2.10.3. L'interface BeanDescriptor....3608
111.2.10.4. L'interface PropertyDescriptor....3609
111.2.10.5. L'interface ConstraintDescriptor....3609
111.2.10.6. Un exemple de mise en oeuvre....3609
111.2.11. La validation des paramètres et de la valeur de retour d'une méthode....3610
111.2.12. L'implémentation de référence : Hibernate Validator....3611
111.2.13. Les avantages et les inconvénients....3611
111.3. D'autres frameworks pour la validation des données....3612

112. L'utilisation des dates....3613
112.1. Les classes standard du JDK pour manipuler des dates....3614
112.1.1. La classe java.util.Date....3614
112.1.2. La classe java.util.Calendar....3615
112.1.3. La classe java.util.GregorianCalendar....3615
112.1.4. Les classes java.util.TimeZone et java.util.SimpleTimeZone....3615
112.1.5. La classe java.text.DateFormat....3616
112.1.6. La classe java.util.SimpleDateFormat....3616
112.1.7. Les classes java.sql.Date, java.sql.Time, java.sql.TimeStamp....3622
112.2. Des exemples de manipulations de dates....3622
112.3. La classe SimpleDateFormat....3624
112.3.1. L'utilisation de la classe SimpleDateFormat....3624
112.3.2. Les points faibles de la classe SimpleDateFormat....3626
112.4. Joda Time....3630
112.4.1. Les principales classes de JodaTime....3631
112.4.2. Le concept d'Instant....3631
112.4.2.1. L'interface ReadableInstant....3631
112.4.2.2. La classe DateTime....3632
112.4.3. Le concept de Partial....3639
112.4.4. Les concepts d'intervalle, de durée et de période....3639
112.4.4.1. La classe Interval....3639
112.4.4.2. La classe Period....3640
112.4.4.3. La classe Duration....3643
112.4.5. Les calendriers et les fuseaux horaires....3643
112.4.5.1. La classe Chronology....3643
112.4.5.2. La classe DateTimeZone....3644
112.4.5.3. Le système calendaire ISO8601....3645
112.4.5.4. La calendrier Bouddhiste....3645
112.4.5.5. Le calendrier Copte....3646
112.4.5.6. Le calendrier Ethiopien....3646
112.4.5.7. Le calendrier Grégorien....3646
112.4.5.8. Le système calendaire Grégorien/Julien....3647
112.4.5.9. Le calendrier Islamique....3647
112.4.5.10. Le calendrier Julien....3648
112.4.6. La manipulation des dates....3648
112.4.7. L'intéropabilité avec les classes du JDK....3650
112.4.8. Le formattage des dates....3651
112.4.9. D'autres fonctionnalités de Joda Time....3652
112.4.9.1. La modification de l'heure de la JVM....3653
112.4.9.2. Les objets mutables....3653

Table des matières

112. L'utilisation des dates

112.5. La classe FastDateFormat du projet Apache commons.lang....3654
112.6. L'API Date and Time....3655
112.6.1. Le besoin d'une nouvelle API....3656
112.6.2. La JSR 310....3657
112.6.3. Les interfaces et classes de bas niveau de l'API....3659
112.6.3.1. L'interface TemporalAccessor....3660
112.6.3.2. L'interface Temporal....3660
112.6.3.3. L'interface TemporalAmount....3661
112.6.3.4. L'interface TemporalField....3662
112.6.3.5. L'énumération ChronoField....3662
112.6.3.6. L'interface TemporalUnit....3664
112.6.3.7. L 'énumération ChronoUnit....3664
112.6.4. Les classes spécifiques aux dates....3665
112.6.4.1. L'énumération DayOfWeek....3665
112.6.4.2. L'énumération Month....3667
112.6.4.3. La classe YearMonth....3668
112.6.4.4. La classe MonthDay....3670
112.6.4.5. La classe Year....3672
112.6.5. La gestion du temps machine....3674
112.6.5.1. La classe Instant....3674
112.6.5.2. La classe Duration....3677
112.6.6. La gestion du temps humain....3680
112.6.6.1. La classe LocalDate....3681
112.6.6.2. La classe LocalDateTime....3685
112.6.6.3. La classe LocalTime....3689
112.6.6.4. La classe Period....3692
112.6.7. Les fuseaux et les décalages horaires....3695
112.6.7.1. La classe ZoneId....3696
112.6.7.2. La classe ZoneOffset....3697
112.6.7.3. La classe ZonedDateTime....3698
112.6.7.4. La classe OffsetDateTime....3702
112.6.7.5. La classe OffsetTime....3703
112.6.8. L'analyse et le formatage....3706
112.6.8.1. La classe DateTimeFormatter....3706
112.6.8.2. La classe DateTimeFormatterBuilder....3713
112.6.9. Les calendriers....3714
112.6.10. Les autres classes de l'API....3715
112.6.10.1. La classe Clock....3715
112.6.10.2. L'interface TemporalAdjuster et la classe TemporalAdjusters....3717
112.6.10.3. L'interface TemporalQuery et la classe TemporalQueries....3721
112.6.11. L'utilisation de l'API Date-Time....3725
112.6.11.1. Le choix du type d'objet temporel à utiliser....3725
112.6.11.2. Les exceptions de l'API....3726
112.6.11.3. L'intégration avec le code avant Java 8....3727

113. La planification de tâches....3729
113.1. La planification de tâches avec l'API du JDK....3729
113.1.1. Les classes Timer et TimerTask....3729
113.1.1.1. La classe java.util.TimerTask....3730
113.1.1.2. La classe java.util.Timer....3730
113.1.2. Le ScheduledExecutorService....3734
113.1.2.1. L'interface java.util.concurrent.ScheduledExecutorService....3734
113.1.2.2. L'interface java.util.concurrent.ScheduledFuture....3735
113.1.2.3. L'utilisation d'un ScheduledExecutorService....3735
113.2. Quartz....3738
113.2.1. Un exemple simple....3739
113.2.2. Les principales classes et interfaces....3740
113.2.3. Le scheduler....3742
113.2.4. Les Jobs et les Triggers....3744

Table des matières

113. La planification de tâches

113.2.5. Les jobs....3744
113.2.5.1. Les jobs fournis par Quartz....3748
113.2.6. Les triggers....3750
113.2.6.1. SimpleTrigger....3752
113.2.6.2. CronTrigger....3757
113.2.6.3. CalendarIntervalTrigger....3761
113.2.6.4. DailyTimeIntervalTrigger....3763
113.2.6.5. L'exclusion de périodes dans la planification....3766
113.2.6.6. La classe TriggerUtils....3771
113.2.7. La gestion des jobs et des triggers....3772
113.2.8. Les listeners....3778
113.2.8.1. JobListener....3779
113.2.8.2. TriggerListener....3781
113.2.8.3. SchedulerListener....3783
113.2.9. Les JobStores....3785
113.2.9.1. Le RAMJobStore....3785
113.2.9.2. Le JDBCJobStore....3785
113.2.10. La configuration....3785
113.2.10.1. La configuration globale....3786
113.2.10.2. La configuration de listeners globaux....3787
113.2.10.3. La configuration des exécutions concurrentes des jobs....3788
113.2.10.4. Le stockage des entités de planification....3789
113.2.10.5. L'utilisation du moteur en mode client/serveur....3793
113.2.10.6. La configuration des plugins....3796
113.2.11. L'utilisation en cluster....3798

114. Des bibliothèques open source....3799

114.1. JFreeChart....3799
114.2. Beanshell....3803
114.3. Apache Commons....3804
114.4. JGoodies....3804
114.5. Apache Lucene....3804

115. Apache Commons....3805

115.1. Apache Commons Configuration....3806
115.1.1. L'interface Configuration....3807
115.1.2. Les différents types de configuration....3808
115.1.2.1. Les configurations reposant sur des fichiers....3808
115.1.2.2. Les configurations dans la base de données....3816
115.1.2.3. Les configurations dans une instance de type Map....3817
115.1.2.4. JNDI....3817
115.1.2.5. Les variables systèmes....3818
115.1.3. Les beans dynamiques....3818
115.1.4. Les configurations composites....3818
115.1.5. Les configurations hiérarchiques....3820
115.1.6. La classe CombinedConfiguration....3826
115.1.7. Les configurations dynamiques....3828
115.1.7.1. La classe ConfigurationFactory....3830
115.1.7.2. La classe DefaultConfigurationBuilder....3831
115.1.8. Les sous-ensembles d'une configuration....3831
115.1.9. Les événements sur la configuration....3832
115.1.10. La classe ConfigurationUtils....3833
115.2. Apache Commons CLI....3834
115.2.1. La définition des options....3835
115.2.2. L'analyse des options fournies en paramètres....3838
115.2.3. L'obtention des options et de leurs valeurs....3840
115.2.4. L'affichage d'une aide sur les options....3841

Table des matières

Partie 16 : Les tests automatisés......3842

116. Les frameworks de tests......3843
116.1. Les tests unitaires......3843
116.1.1. L'utilité des tests unitaires automatisés......3844
116.1.2. Quelques principes pour mettre en oeuvre des tests unitaires......3845
116.1.3. Les difficultés lors de la mise en oeuvre de tests unitaires......3846
116.1.4. Des best practices......3847
116.2. Les frameworks et outils de tests......3848
116.2.1. Les frameworks pour les tests unitaires......3848
116.2.2. Les frameworks pour le mocking......3848
116.2.3. Les extensions de JUnit......3848
116.2.4. Les outils de tests de charge......3849
116.2.5. Les outils d'analyse de couverture de tests......3849

117. JUnit......3850
117.1. Un exemple très simple......3851
117.2. L'écriture des cas de tests......3852
117.2.1. La définition de la classe de tests......3852
117.2.2. La définition des cas de tests......3853
117.2.3. L'initialisation des cas de tests......3856
117.2.4. Le test de la levée d'exceptions......3858
117.2.5. L'héritage d'une classe de base......3860
117.3. L'exécution des tests......3860
117.3.1. L'exécution des tests dans la console......3861
117.3.2. L'exécution des tests dans une application graphique......3862
117.3.3. L'exécution d'une classe de tests 3864
117.3.4. L'exécution répétée d'un cas de tests......3864
117.3.5. L'exécution concurrente de tests......3864
117.4. Les suites de tests......3866
117.5. L'automatisation des tests avec Ant......3867
117.6. JUnit 4......3868
117.6.1. La définition d'une classe de tests......3868
117.6.2. La définition des cas de tests......3868
117.6.3. L'initialisation des cas de tests......3869
117.6.4. Le test de la levée d'exceptions......3870
117.6.5. L'exécution des tests......3870
117.6.6. Un exemple de migration de JUnit 3 vers JUnit 4......3871
117.6.7. La limitation du temps d'exécution d'un cas de tests......3872
117.6.8. Les tests paramétrés......3873
117.6.9. La rétro compatibilité......3873
117.6.10. L'organisation des tests......3874

118. JUnit 5......3877
118.1. L'architecture......3878
118.2. Les dépendances......3878
118.3. L'écriture de tests......3880
118.3.1. Les annotations......3880
118.4. L'écriture de tests standard......3881
118.4.1. La définition d'une méthode de test......3881
118.4.2. La définition d'un libellé......3882
118.4.3. Le cycle de vie des tests......3882
118.4.3.1. La définition de méthodes exécutées avant/après tous les tests......3884
118.4.3.2. La définition de méthodes exécutées avant/après chaque tests......3884
118.5. Les assertions......3884
118.5.1. L'assertion assertAll......3885
118.5.2. L'assertion assertArrayEquals......3886
118.5.3. Les assertions assertEquals et assertNotEquals......3887
118.5.4. Les assertions assertTrue et assertFalse......3888
118.5.5. L'assertion assertIterableEquals......3889

Table des matières

118. JUnit 5

118.5.6. L'assertion assertLinesMatch....3890
118.5.7. Les assertions assertNull et assertNotNull....3891
118.5.8. Les assertions assertSame et assertNotSame....3892
118.5.9. L'assertion assertThrows....3893
118.5.10. Les assertions assertTimeout et assertTimeoutPreemptively....3894
118.5.11. L'assertion fail....3895
118.5.12. L'utilisation d'assertions de bibliothèques tiers....3896
118.6. Les suppositions....3896
118.7. La désactivation de tests....3898
118.8. Les tags....3898
118.9. Le cycle de vie des instances de test....3899
118.10. Les tests imbriqués....3899
118.11. L'injection d'instances dans les constructeurs et les méthodes de tests....3902
118.12. Les tests répétés....3904
118.13. Les tests paramétrés....3905
118.13.1. Les sources des arguments....3906
118.13.1.1. L'annotation @ValueSource....3906
118.13.1.2. L'annotation @EnumSource....3906
118.13.1.3. L'annotation @MethodSource....3908
118.13.1.4. L'annotation @CsvSource....3909
118.13.1.5. L'annotation @CsvFileSource....3910
118.13.1.6. L'annotation @ArgumentsSource....3910
118.13.1.7. La conversion implicite des arguments....3911
118.13.1.8. La conversion implicite des arguments....3912
118.13.2. La personnalisation du libellé des tests....3913
118.14. Les tests dynamiques....3913
118.15. Les tests dans une interface....3915
118.16. Les suites de tests....3917
118.16.1. La création d'une suite de tests en précisant les packages....3917
118.16.2. Créer une suite de tests en précisant les classes de tests....3918
118.16.3. Les annotations @IncludePackages et @ExcludePackages....3919
118.16.4. Les annotations @IncludeClassNamePatterns et @ExcludeClassNamePatterns....3919
118.16.5. Les annotations @IncludeTags et @ExcludeTags....3920
118.17. La compatibilité....3920
118.17.1. La migration de JUnit 4 vers JUnit 5....3921
118.18. La comparaison entre JUnit 4 et JUnit 5....3921

119. Les objets de type mock....3923
119.1. Les doublures d'objets et les objets de type mock....3923
119.1.1. Les types d'objets mock....3924
119.1.2. Exemple d'utilisation dans les tests unitaires....3924
119.1.3. La mise en oeuvre des objets de type mock....3925
119.2. L'utilité des objets de type mock....3925
119.2.1. L'utilisation dans les tests unitaires....3925
119.2.2. L'utilisation dans les tests d'intégration....3926
119.2.3. La simulation de l'appel à des ressources....3926
119.2.4. La simulation du comportement de composants ayant des résultats variables....3926
119.2.5. La simulation des cas d'erreurs....3926
119.3. Les tests unitaires et les dépendances....3927
119.4. L'obligation d'avoir une bonne organisation du code....3927
119.4.1. Quelques recommandations....3927
119.4.2. Les dépendances et les tests unitaires....3928
119.4.3. Exemple de mise en oeuvre de l'injection de dépendances....3929
119.4.4. Limiter l'usage des singletons....3930
119.4.5. Encapsuler les ressources externes....3931
119.5. Les frameworks....3931
119.5.1. EasyMock....3932
119.5.1.1. La création d'objets mock....3934
119.5.1.2. La définition du comportement des objets mock....3934

Table des matières

119. Les objets de type mock

119.5.1.3. L'initialisation des objets mock.....3935
119.5.1.4. La vérification des invocations des objets mock.....3936
119.5.1.5. La vérification de l'ordre d'invocations des mocks.....3936
119.5.1.6. La gestion de plusieurs objets de type mock.....3938
119.5.1.7. Un exemple complexe.....3938
119.5.2. Mockito.....3938
119.5.3. JMock.....3938
119.5.4. MockRunner.....3939
119.6. Les inconvénients des objets de type mock.....3939

Partie 17 : Les outils de profiling et monitoring.....3940

120. Arthas.....3941
120.1. Installation.....3942
120.1.1. L'installation avec le fichier arthas-boot.jar.....3942
120.1.2. L'installation via les binaires sous Windows.....3943
120.1.3. L'installation via un script sous Linux et Mac.....3943
120.1.4. L'installation via des packages pour Debian et Fedora.....3945
120.1.5. La mise à jour.....3945
120.1.6. La désinstallation.....3946
120.2. Le démarrage.....3946
120.2.1. Le démarrage avec le jar arthas-boot.jar.....3946
120.2.2. Le démarrage avec le script as.sh/as.bat.....3948
120.2.3. L'utilisation de la console CLI.....3949
120.3. Les fonctionnalités.....3950
120.3.1. Les fonctionnalités de base.....3950
120.3.1.1. Quitter Arthas.....3950
120.3.1.2. La commande cat.....3951
120.3.1.3. La commande grep.....3951
120.3.1.4. La commande pwd.....3952
120.3.1.5. La commande options.....3952
120.3.1.6. La commande help.....3954
120.3.1.7. La commande cls.....3955
120.3.1.8. La commande session.....3955
120.3.1.9. La commande reset.....3955
120.3.1.10. La commande version.....3956
120.3.1.11. La commande history.....3956
120.3.1.12. La commande keymap.....3956
120.3.2. Les pipes.....3957
120.3.3. Les fonctionnalités concernant la JVM.....3957
120.3.3.1. La commande dashboard.....3958
120.3.3.2. La commande heapdump.....3959
120.3.3.3. La commande thread.....3959
120.3.3.4. La commande jvm.....3961
120.3.3.5. La commande sysprop.....3963
120.3.3.6. La commande sysenv.....3965
120.3.3.7. La commande vmoption.....3966
120.3.3.8. La commande mbean.....3966
120.3.3.9. La commande logger.....3968
120.3.4. Les fonctionnalités concernant les classes/classloaders.....3970
120.3.4.1. La commande getstatic.....3970
120.3.4.2. La commande dump.....3971
120.3.4.3. La commande sc.....3971
120.3.4.4. La commande sm.....3973
120.3.4.5. La commande classloader.....3974
120.3.4.6. La commande jad.....3976
120.3.4.7. La commande mc.....3978
120.3.4.8. La commande redefine.....3979
120.3.4.9. La commande ognl.....3980

Table des matières

120. Arthas

120.3.5. Les fonctionnalités de profiling....3981
120.3.5.1. La commande monitor....3981
120.3.5.2. La commande stack....3983
120.3.5.3. La commande trace....3985
120.3.5.4. La commande watch....3987
120.3.5.5. La commande tt....3991
120.3.5.6. La commande profiler....3994
120.4. L'exécution de commandes en asynchrone....3998
120.4.1. L'exécution de commandes en arrière-plan....3999
120.4.2. La redirection de la sortie....3999
120.4.3. Afficher la liste des tâches en arrière-plan....4000
120.4.4. L'arrêt d'une tâche en arrière-plan....4000
120.5. Les scripts batch....4001
120.6. La console web....4001

121. VisualVM....4003
121.1. L'utilisation de VisualVM....4003
121.2. Les plugins pour VisualVM....4005
121.3. L'utilisation de VisualVM....4006
121.4. La connexion à une JVM....4007
121.5. L'obtention d'informations....4009
121.5.1. La génération d'un thread dump....4009
121.5.2. La génération d'un heap dump....4009
121.5.3. Le parcours d'un heap dump....4010
121.5.4. L'onglet Overview....4012
121.5.5. L'onglet Monitor....4012
121.5.6. L'onglet Threads....4013
121.6. Le profilage d'une JVM....4015
121.7. La création d'un snapshot....4016
121.8. Le plugin VisualGC....4017

Partie 18 : Java et le monde informatique....4019

122. La communauté Java....4020
122.1. Le JCP....4020
122.2. Les ressources proposées par Oracle....4021
122.3. Oracle Technology Network....4021
122.4. La communauté Java.net....4021
122.5. Les JUG....4022
122.6. Les Cast Codeurs Podcast....4024
122.7. Parleys.com....4024
122.8. Les conférences Devoxx et Voxxed Days....4025
122.8.1. Devoxx Belgium (ex : JavaPolis)....4025
122.8.2. Devoxx France....4025
122.8.3. VoxxedDays Luxembourg....4026
122.8.4. VoxxedDays Microservices....4027
122.9. Les conférences....4027
122.9.1. JavaOne....4027
122.9.2. JCertif....4027
122.9.3. Mix-IT....4028
122.9.4. JUG Summer Camp....4028
122.9.5. Codeurs en Seine....4029
122.9.6. Breizhcamp....4029
122.9.7. RivieraDev....4029
122.9.8. Sunny Tech....4030
122.9.9. SnowCamp....4030
122.9.10. Touraine Tech....4030
122.9.11. Bdx.io....4030
122.9.12. SophiaConf....4031

Table des matières

122. La communauté Java

122.9.13. EclipseCon France....4031
122.9.14. Jazoon....4031
122.10. Les unconférences....4032
122.10.1. JChateau....4032
122.11. Webographie....4032
122.12. Les communautés open source....4033
122.12.1. Apache - Jakarta....4033
122.12.2. La fondation Eclipse....4033
122.12.3. Codehaus....4033
122.12.4. OW2....4033
122.12.5. JBoss....4034
122.12.6. Source Forge....4034

123. Les plates-formes Java et .Net....4035

123.1. La présentation des plates-formes Java et .Net....4036
123.1.1. Les plates-formes supportées....4036
123.1.2. Standardisation....4037
123.2. La compilation....4037
123.3. Les environnements d'exécution....4037
123.3.1. Les machines virtuelles....4037
123.3.2. Le ramasse-miettes....4038
123.4. Le déploiement des modules....4038
123.5. Les version des modules....4038
123.6. L'interopérabilité inter-language....4039
123.7. La décompilation....4039
123.8. Les API des deux plates-formes....4039
123.8.1. La correspondance des principales classes....4040
123.8.1.1. La correspondance des classes de bases....4041
123.8.1.2. La correspondance des classes utilitaires....4041
123.8.2. Les collections....4042
123.8.3. Les entrées/sorties....4042
123.8.3.1. La correspondance des classes pour gérer les entrées/sorties....4043
123.8.4. L'accès aux bases de données....4044
123.8.4.1. Les API de bas niveau....4044
123.8.4.2. Les frameworks de type ORM....4045
123.8.5. Les interfaces graphiques....4046
123.8.6. Le développement d'applications web....4047
123.8.6.1. Les APIs de bas niveau....4047
123.8.6.2. Les frameworks....4047
123.8.7. Le développement d'applications de type RIA....4047

124. Java et C#....4048

124.1. La syntaxe....4049
124.1.1. Les mots clés....4049
124.1.2. L'organisation des classes....4050
124.1.3. Les conventions de nommage....4051
124.1.4. Les types de données....4052
124.1.4.1. Les types primitifs....4052
124.1.4.2. Les types objets....4053
124.1.4.3. Les types valeur (ValueTypes et Structs)....4053
124.1.5. La déclaration de constantes....4053
124.1.6. Les instructions....4054
124.1.6.1. L'instruction switch....4054
124.1.6.2. L'instruction goto....4055
124.1.6.3. Le parcours des collections de données....4055
124.1.7. Les metadatas....4057
124.1.8. Les énumérations....4057
124.1.9. Les délégués....4058
124.1.10. Les événements....4058

Table des matières

124. Java et C#

124.1.11. Le contrôle sur le débordement d'un downcast....4059
124.1.12. Les directives de précompilation....4059
124.1.13. La méthode main()....4059
124.2. La programmation orientée objet....4060
124.2.1. Les interfaces....4061
124.2.2. Les modificateurs d'accès....4062
124.2.3. Les champs....4062
124.2.4. Les propriétés....4063
124.2.5. Les indexeurs....4064
124.2.6. Les constructeurs....4064
124.2.7. Les constructeurs statics....4067
124.2.8. Les destructeurs....4067
124.2.9. Le passage de paramètres....4068
124.2.10. Liste de paramètres multiples....4068
124.2.11. L'héritage....4069
124.2.12. Le polymorphisme....4070
124.2.13. Les generics....4075
124.2.14. Le boxing/unboxing....4076
124.2.15. La surcharge des opérateurs....4076
124.2.16. Les classes imbriquées....4076
124.2.17. Les classes anonymes internes (Anonymous Inner classes)....4076
124.2.18. L'import de classes....4076
124.2.19. Déterminer et tester le type d'un objet....4077
124.2.20. L'opérateur as....4077
124.3. Les chaînes de caractères....4077
124.4. Les tableaux....4078
124.5. Les indexeurs....4078
124.6. Les exceptions....4078
124.7. Le multitâche....4079
124.7.1. Les threads....4079
124.7.2. La synchronisation de portions de code....4079
124.7.3. Le mot clé volatile....4080
124.8. L'appel de code natif....4080
124.9. Les pointeurs....4081
124.10. La documentation automatique du code....4081
124.11. L'introspection/reflection....4081
124.12. La sérialisation....4081

Partie 19 : Le développement d'applications mobiles....4082

125. J2ME / Java ME....4083
125.1. L'historique de la plate-forme....4083
125.2. La présentation de J2ME / Java ME....4084
125.3. Les configurations....4085
125.4. Les profiles....4086
125.5. J2ME Wireless Toolkit 1.0.4....4087
125.5.1. L'installation du J2ME Wireless Toolkit 1.0.4....4087
125.5.2. Les premiers pas....4088
125.6. J2ME wireless toolkit 2.1....4090
125.6.1. L'installation du J2ME Wireless Toolkit 2.1....4090
125.6.2. Les premiers pas....4091

126. CLDC....4096
126.1. Le package java.lang....4096
126.2. Le package java.io....4097
126.3. Le package java.util....4098
126.4. Le package javax.microedition.io....4098

Table des matières

127. MIDP....4099

127.1. Les Midlets....4099

127.2. L'interface utilisateur....4100

127.2.1. La classe Display....4101

127.2.2. La classe TextBox....4102

127.2.3. La classe List....4103

127.2.4. La classe Form....4104

127.2.5. La classe Item....4104

127.2.6. La classe Alert....4106

127.3. La gestion des événements....4107

127.4. Le stockage et la gestion des données....4108

127.4.1. La classe RecordStore....4108

127.5. Les suites de midlets....4109

127.6. Packager une midlet....4109

127.6.1. Le fichier manifest....4110

127.7. MIDP for Palm O.S.....4110

127.7.1. L'installation....4110

127.7.2. La création d'un fichier .prc....4111

127.7.3. L'installation et l'exécution d'une application....4114

128. CDC....4115

129. Les profils du CDC....4116

129.1. Foundation profile....4116

129.2. Le Personal Basis Profile (PBP)....4117

129.3. Le Personal Profile (PP)....4118

130. Les autres technologies pour les applications mobiles....4119

130.1. KJava....4119

130.2. PDAP (PDA Profile)....4119

130.3. PersonalJava....4120

130.4. Embedded Java....4120

Partie 20 : Annexes....4121

Annexe A : GNU Free Documentation License....4121

Annexe B : Glossaire....4125

Développons en Java

Version 2.40

du 20/12/2023

par Jean-Michel DOUDOUX

XML
SAX MIDP JAXB DOM StAX SWT
JMX J2ME SWING Struts J2SE5 Tests XSLT
JSTL Scripting Terracotta Servlets JDBC RMI
JMS Multitache Decompilation JavaBeans Tomcat POO
Web Services JavaEE Obfuscation JSF Logging AWT
Introspection JavaWebStart Hibernate CLDC
.Net JNI Persistance
Outils Spring JAVA Serialisation Collections
Exceptions Frameworks
Internationalisation JavaME Executor CSharp
Lambda Design Patterns Ant Maven JPA
GWT Date Time Annotations Communaute Securite Ajax Web
Mock JavaMail Reseau Packages Applets Flux UML
JVM CDC J2EE JFace JavaDoc JUnit Soap
RIA EJB JNDI Java8 JSP RDA
JDO

Préambule

A propos de ce document

L'idée de départ de ce document était de prendre des notes relatives à mes premiers essais avec Java en 1996. Ces notes ont tellement grossi que j'ai décidé de les formaliser un peu plus et de les diffuser sur Internet d'abord sous la forme d'articles puis rassemblées pour former le présent ouvrage.

Aujourd'hui, celui-ci est composé de 20 grandes parties :

1. Les bases du langage Java
2. Les API de base
3. Les API avancées
4. Le système de modules
5. La programmation parallèle et concurrente
6. Le développement des interfaces graphiques
7. L'utilisation de documents XML et JSON
8. L'accès aux bases de données
9. La machine virtuelle Java (JVM)
10. Le développement d'applications d'entreprises
11. Le développement d'applications web
12. Le développement d'applications RIA / RDA
13. Le développement d'applications avec Spring
14. Les outils pour le développement
15. La conception et le développement des applications
16. Les tests automatisés
17. Les outils de profiling et monitoring
18. Java et le monde informatique
19. Le développement d'applications mobiles
20. Annexes

Chacune de ces parties est composée de plusieurs chapitres dont voici la liste complète :

- Préambule
- Présentation de Java
- Les notions et techniques de base en Java
- La syntaxe et les éléments de bases de Java
- POO avec Java
- Les génériques (generics)
- Les chaînes de caractères
- Les packages de base
- Les fonctions mathématiques
- La gestion des exceptions
- Les énumérations (type enum)
- Les annotations
- Les expressions lambda
- Les records
- Les collections
- Les flux d'entrée/sortie
- NIO 2
- La sérialisation
- L'interaction avec le réseau
- L'internationalisation
- Les composants Java beans
- Le logging
- L'API Stream
- Les expressions régulières
- La gestion dynamique des objets et l'introspection
- L'appel de méthodes distantes : RMI

- La sécurité
- JCA (Java Cryptography Architecture)
- JCE (Java Cryptography Extension)
- JNI (Java Native Interface)
- JNDI (Java Naming and Directory Interface)
- Le scripting
- JMX (Java Management Extensions)
- L'API Service Loader
- Le système de modules de la plateforme Java
- Les modules
- Le multitâche
- Les threads
- L'association de données à des threads
- Le framework Executor
- La gestion des accès concurrents
- Le graphisme en Java
- Les éléments d'interfaces graphiques de l'AWT
- La création d'interfaces graphiques avec AWT
- L'interception des actions de l'utilisateur
- Le développement d'interfaces graphiques avec SWING
- Le développement d'interfaces graphiques avec SWT
- JFace
- Java et XML
- SAX (Simple API for XML)
- DOM (Document Object Model)
- XSLT (Extensible Stylesheet Language Transformations)
- Les modèles de documents
- JAXB (Java Architecture for XML Binding)
- StAX (Streaming Api for XML)
- Json
- Gson
- JSON-P
- JSON-B
- La persistance des objets
- JDBC
- JDO (Java Data Object)
- Hibernate
- JPA (Java Persistence API)
- La JVM (Java Virtual Machine)
- La gestion de la mémoire dans la JVM HotSpot
- La JVM HotSpot dans un conteneur Docker
- La décompilation et l'obfuscation
- Programmation orientée aspects (AOP)
- Terracotta
- Java Entreprise Edition
- JavaMail
- JMS (Java Message Service)
- Les EJB (Entreprise Java Bean)
- Les EJB 3
- Les EJB 3.1
- Les services web de type Soap
- Les Websockets
- L'API Websocket
- Les servlets
- Les JSP (Java Server Pages)
- JSTL (Java server page Standard Tag Library)
- Struts
- JSF (Java Server Faces)
- D'autres frameworks pour les applications web
- Les applications riches de type RIA et RDA
- Les applets
- Java Web Start (JWS)

- Ajax
- GWT (Google Web Toolkit)
- Spring
- Spring Core
- Spring et AOP
- La gestion des transactions avec Spring
- Spring et JMS
- Spring et JMX
- Les outils du J.D.K.
- JavaDoc
- JShell
- Les outils libres et commerciaux
- Ant
- Maven
- Tomcat
- Des outils open source
- Java et UML
- Les motifs de conception (design patterns)
- Des normes de développement
- Les techniques de développement spécifiques à Java
- L'encodage des caractères
- Les frameworks
- La génération de documents
- La validation des données
- L'utilisation des dates
- La planification de tâches
- Des bibliothèques open source
- Apache Commons
- Les frameworks de tests
- JUnit
- JUnit 5
- Les objets de type mock
- Arthas
- VisualVM
- La communauté Java
- Les plates-formes Java et .Net
- Java et C#
- J2ME/JavaME
- CLDC
- MIDP
- CDC
- Les profils du CDC
- Les autres technologies

Je souhaiterais l'enrichir pour qu'il couvre un maximum de sujets autour du développement avec les technologies relatives à Java. Ce souhait est ambitieux car les API de Java et open source sont très riches et ne cessent de s'enrichir au fil des versions et des années.

Dans chaque chapitre, les classes et les membres des classes décrits ne le sont que partiellement : le but n'est pas de remplacer la documentation d'une API mais de faciliter ses premières mises en oeuvre. Ainsi pour une description complète de chaque classe, il faut consulter la documentation fournie par Sun/Oracle au format HTML pour les API du JDK et la documentation fournie par les fournisseurs respectifs des API tiers.

Je suis ouvert à toutes réactions ou suggestions concernant ce document notamment le signalement des erreurs, les points à éclaircir, les sujets à ajouter, etc. ... N'hésitez pas à me contacter : jean-michel.doudoux@wanadoo.fr

La dernière version publiée de ce document est disponible aux formats HTML et PDF sur mon site personnel :
https://www.jmdoudoux.fr/java/

Il est aussi disponible en mirroir sur le site developpez.com à l'url :
https://jmdoudoux.developpez.com/cours/developpons/java/

Ce manuel est fourni en l'état, sans aucune garantie. L'auteur ne peut être tenu pour responsable des éventuels dommages causés par l'utilisation des informations fournies dans ce document.

La version PDF de ce document est réalisée grâce à l'outil freeware HTMLDOC version 1.9.17.

La version sur mon site perso utilise deux outils open source :

- PrismJS : pour afficher et appliquer une coloration syntaxique des exemples (https://prismjs.com/)
- JavaScript Tree Menu : pour afficher l'arborescence des chapitres et sections, facilitant ainsi la navigation dans ce document

Remerciements

Je souhaite remercier les personnes qui m'ont apporté leur soutien au travers de courriers électroniques de remerciements, de félicitations ou d'encouragements.

Je tiens aussi particulièrement à exprimer ma gratitude aux personnes qui m'ont fait part de correctifs ou d'idées d'évolutions : ainsi pour leurs actions, je veux particulièrement remercier Vincent Brabant, Thierry Durand, David Riou et surtout François Vancata.

Notes de licence

Marques déposées

Sun, Sun Microsystems, le logo Sun et Java sont des marques déposées de Sun Microsystems Inc jusqu'en décembre 2009 puis d'Oracle à partir de janvier 2010.

Les autres marques et les noms de produits cités dans ce document sont la propriété de leur éditeur respectif.

Historique des versions

Version	Date	Evolutions
0.10	15/01/2001	brouillon : 1ere version diffusée sur le web.
0.20	11/03/2001	ajout des chapitres JSP et sérialisation, des informations sur le JDK et son installation, corrections diverses.
0.30	10/05/2001	ajout des chapitres flux, beans et outils du JDK, corrections diverses.
0.40	10/11/2001	réorganisation des chapitres et remise en page du document au format HTML (1 page par chapitre) pour faciliter la maintenance ajout des chapitres : collections, XML, JMS, début des chapitres Swing et EJB

		séparation du chapitre AWT en trois chapitres.
0.50	31/04/2002	séparation du document en trois parties ajout des chapitres : logging, JNDI, Java mail, services web, outils du JDK, outils lires et commerciaux, Java et UML, motifs de conception compléments ajoutés aux chapitres : JDBC, Javadoc, interaction avec le réseau, Java et XML, bibliothèques de classes
0.60	23/12/2002	ajout des chapitres : JSTL, JDO, Ant, les frameworks ajout des sections : Java et MySQL, les classes internes, les expressions régulières, dom4j compléments ajoutés aux chapitres : JNDI, design patterns, J2EE, EJB
0.65	05/04/2003	ajout d'un index sous la forme d'un arbre hiérarchique affiché dans un frame de la version HTML ajout des sections : DOM, JAXB, bibliothèques de tags personnalisés, package .war compléments ajoutés aux chapitres : EJB, réseau, services web
0.70	05/07/2003	ajout de la partie sur le développement d'applications mobiles contenant les chapitres : J2ME, CLDC, MIDP, CDC, Personal Profile, les autres technologies ajout des chapitres : le multitâche, les frameworks de tests, la sécurité, les frameworks pour les app web compléments ajoutés aux chapitres : JDBC, JSP, servlets, interaction avec le réseau application d'une feuille de styles CSS pour la version HTML corrections et ajouts divers (652 pages)
0.75	21/03/2004	ajout des chapitres : le développement d'interfaces avec SWT, Java Web Start, JNI compléments ajoutés aux chapitres : GCJ, JDO, nombreuses corrections et ajouts divers notamment dans les premiers chapitres (737 pages)
0.80 0.80.1 0.80.2	29/06/2004 15/10/2004 02/11/2004	ajout des chapitres : le JDK 1.5, des bibliothèques open source, des outils open source, Maven et d'autres solutions de mapping objet-relationnel ajout des sections : Installation J2SE 1.4.2 sous Windows, J2EE 1.4 SDK, J2ME WTK 2.1 compléments ajoutés aux chapitres : Ant, JDBC, Swing, Java et UML, MIDP, J2ME, JSP, JDO nombreuses corrections et ajouts divers (831 pages)
0.85	27/11/2005	ajout du chapitre : Java Server Faces ajout des sections : Java updates, le composant JTree nombreuses corrections et ajouts divers (922 pages)
0.90	11/09/2006	ajout des chapitres : Ajax, Frameworks, Struts compléments ajoutés aux chapitres : Javadoc, JNDI, Design pattern (façade, fabrique, fabrique abstraite), JFace nombreuses corrections et ajouts divers (1092 pages)
0.95	18/11/2007	ajout des parties : utilisation de documents XML, l'Accès aux bases de données, développement d'applications web, concevoir et développer des applications

0.95.1	12/06/2008	ajout des chapitres : Scripting, persistance des objets, StAX, JPA, Tomcat
0.95.2	01/11/2008	compléments ajoutés aux chapitres : Java SE 6, JAXB 2.0, Java DB, Java EE 5, JMS 1.1, OpenJMS, les menus avec Swing, le design pattern décorateur, Rowset nombreuses corrections et ajouts divers (1305 pages)
1.00	16/03/2009	ajout des parties : la JVM et le développement d'applications RIA/RDA ajout des chapitres : annotations, décompilation et obfuscation, génération de documents, GWT, JVM, la gestion de la mémoire, la communauté Java, les applications RIA/RDA réécriture complète des chapitres : les modèles de documents (JDOM), les techniques de bases, Logging (log4j), JUnit compléments ajoutés aux chapitres : la gestion des exceptions, JDBC (performances), les fonctions mathématiques (BigDecimal), JDBC 3.0, les framework de tests, Apache POI, iText nombreuses corrections et ajouts divers (1672 pages)
1.10	04/08/2009	ajout des chapitres : JMX, EJB 3 et l'encodage des caractères compléments ajoutés aux chapitres : J2ME /Java ME corrections et ajouts divers (1820 pages)
1.20	29/10/2009	ajout du chapitre : les objets de type mock ajout de la section : Java 6 Update compléments ajoutés aux chapitres : les frameworks de tests, JUnit, GWT et RIA nombreuses corrections et ajouts divers (1888 pages)
1.30	27/03/2010	ajout des chapitres : la validation de données, EJB 3.1 ajout de la section : Java EE 6 réécriture du chapitre : les services web de type Soap nombreuses corrections et ajouts divers (2035 pages)
1.40	08/08/2010	ajout du chapitre : Terracotta ajout de la section : Hibernate (les relations 1/1) réécriture de la section : les énumérations nombreuses corrections et ajouts divers (2105 pages)
1.50	30/12/2010	ajout des chapitres : l'utilisation des dates, les plate-formes Java et .Net et Java et C# ajout de la section : l'API Criteria d'Hibernate réécriture de la section : le design pattern Singleton nombreuses corrections et ajouts divers (2198 pages)
1.60	11/09/2011	ajout de la partie : le développement d'applications avec Spring ajout des chapitres : Spring, Spring Core, la gestion des transactions avec Spring, Spring et JMS ajout de la section : Hibernate HQL, Service Provider Interface nombreuses corrections et ajouts divers (2330 pages)

1.70	05/03/2012	ajout des chapitres : Java 7 le projet Coin, les techniques de développement spécifiques à Java, la mise en oeuvre de l'AOP avec Spring mise à jour du chapitre : RMI compléments ajoutés relatifs à Java 7 très nombreuses corrections et ajouts divers (2402 pages, 100 chapitres)
1.80	14/10/2012	ajout du chapitre : NIO 2 ajout des sections : les caches d'Hibernate, JRadioButton mise à jour de la section : la création dynamique d'objets nombreuses corrections et ajouts divers (2507 pages, 101 chapitres)
1.90	25/02/2013	ajout du chapitre : Spring et JMX mise à jour du chapitre : Logging ajout des sections : Joda Time, VisualVM et JVM 32/64 bits très nombreuses corrections et ajouts divers (2599 pages, 102 chapitres)
2.00	19/05/2014	La version du quinzième anniversaire ajout des chapitres : JSON, Json-P, Gson, Websockets, l'API Websocket, JCA, JCE, AOP, Apache Commons réécriture des chapitres : les collections, la sérialisation ajout des sections : logging (NDC/MDC), parcours du DOM, ActiveMQBrowser, Maven 2, Java SE 8 très nombreuses corrections et ajouts divers (3004 pages, 111 chapitres)
2.10	19/03/2016	ajout du chapitre : la planification de tâches, les expressions lambdas, les threads, l'association de données à des threads, le framework executor, la gestion de la concurrence ajout des sections : l'invocation dynamique d'une méthode, l'API Reflection et le SecurityManager, l'utilisation de l'API Reflection sur les annotations, le clonage d'un objet, l'API Date-Time, le mapping de l'héritage de classes (ORM, JPA, Hibernate), les méthodes par défaut, l'héritage de méthodes statiques, les outils de génération de code Hibernate Corrections et ajouts divers (3413 pages, 117 chapitres)
2.20	06/01/2019	ajout des chapitres : JUnit 5, JSON-B, l'API Stream Corrections et ajouts divers (3623 pages, 120 chapitres)
2.30	15/06/2022	ajout des chapitres : JShell, l'API Service Loader, Arthas, les records, le système de module, les modules réécriture des chapitres : les chaines de caractères, les génériques très nombreuses corrections et ajouts divers (4001 pages, 128 chapitres)
2.40	20/12/2023	ajout des chapitres : les expressions régulières, la JVM Hotspot dans un conteneur Docker ajout des sections : les types scellés, la déclaration de variables locales avec l'inférence de type, les évolutions de l'instruction switch dans Java 14, les warnings du compilateur (lossy-conversions, missing-explicit-ctor, strictfp, synchronization, text-blocks), la version du bytecode, la gestion des évolutions des JEPs, l'utilisation de fonctionnalités non standards, les builds early access, la précision des calculs en virgule flottante

		réécriture des sections : les structures de controles, les mot réservés Complément dans le chapitre : JDBC migration vers HTMLDOC v 1.9.17 très nombreuses corrections et ajouts divers (4207 pages, 130 chapitres)

Partie 1 : Les bases du langage Java

Partie 1 :

Les bases
du langage
Java

Cette première partie est chargée de présenter les bases du langage Java. Elle comporte les chapitres suivants :

- Présentation de Java : introduit le langage Java en présentant les différentes éditions et versions du JDK, les caractéristiques du langage et décrit l'installation du JDK
- Les notions et techniques de base en Java : présente rapidement quelques notions de base et comment compiler et exécuter une application
- La syntaxe et les éléments de bases de Java : explore les éléments du langage d'un point de vue syntaxique
- La programmation orientée objet : explore comment Java permet d'utiliser la programmation orientée objet
- Les génériques (generics) : détaille la définition et l'utilisation de types génériques
- Les chaînes de caractères : détaille l'utilisation des chaînes de caractères
- Les packages de bases : propose une présentation rapide des principales API fournies avec le JDK
- Les fonctions mathématiques : indique comment utiliser les fonctions mathématiques
- La gestion des exceptions : explore la faculté qu'a Java de traiter et gérer les anomalies qui surviennent lors de l'exécution du code
- Les énumérations (type enum) : détaille les nouvelles fonctionnalités du langage de la version 1.5
- Les annotations : présente les annotations qui sont des métadonnées insérées dans le code source et leurs mises en oeuvre.
- Les expressions lambda : détaille l'utilisation des expressions lambda et des références de méthodes et constructeurs.
- Les records : décrit la déclaration et l'utilisation de classes records

1. Présentation de Java

Chapitre 1

Niveau : 🟩□□□□□ Fondamental

Java est un langage de programmation à usage général, évolué et orienté objet dont la syntaxe est proche du C. Ses caractéristiques ainsi que la richesse de son écosystème et de sa communauté lui ont permis d'être très largement utilisé pour le développement d'applications de types très disparates. Java est notamment largement utilisé pour le développement d'applications d'entreprises et mobiles.

Quelques chiffres et faits à propos de Java en 2011 :

- 97% des machines d'entreprises ont une JVM installée
- Java est téléchargé plus d'un milliards de fois chaque année
- Il y a plus de 9 millions de développeurs Java dans le monde
- Java est un des langages les plus utilisés dans le monde
- Tous les lecteurs de Blue-Ray utilisent Java
- Plus de 3 milliards d'appareils mobiles peuvent mettre en oeuvre Java
- Plus de 1,4 milliards de cartes à puce utilisant Java sont produites chaque année

En 2020, Java fête ses 25 ans.

Ce chapitre contient plusieurs sections :

♦ Les caractéristiques
♦ Les logos de Java
♦ Un rapide historique de Java
♦ Les différentes éditions et versions de Java
♦ Un rapide tour d'horizon des API et de quelques outils
♦ Les différences entre Java et JavaScript
♦ L'installation du JDK

1.1. Les caractéristiques

Java possède un certain nombre de caractéristiques qui ont largement contribué à son énorme succès :

Java est interprété	le source est compilé en pseudo code ou bytecode puis exécuté par un interpréteur Java : la Java Virtual Machine (JVM). Ce concept est à la base du slogan de Sun pour Java : WORA (Write Once, Run Anywhere : écrire une fois, exécuter partout). En effet, le bytecode, s'il ne contient pas de code spécifique à une plate-forme particulière peut être exécuté et obtenir quasiment les mêmes résultats sur toutes les machines disposant d'une JVM.
Java est portable : il est indépendant de toute plate-forme	il n'y a pas de compilation spécifique pour chaque plate forme. Le code reste indépendant de la machine sur laquelle il s'exécute. Il est possible d'exécuter des programmes Java sur tous les environnements qui possèdent une Java Virtual Machine. Cette indépendance est assurée au niveau du code source grâce à Unicode et au niveau du bytecode.

Java est orienté objet.	comme la plupart des langages récents, Java est orienté objet. Chaque fichier source contient la définition d'une ou plusieurs classes qui sont utilisées les unes avec les autres pour former une application. Java n'est pas complètement objet car il définit des types primitifs (entier, caractère, flottant, booléen,...).
Java est simple	le choix de ses auteurs a été d'abandonner des éléments mal compris ou mal exploités des autres langages tels que la notion de pointeurs (pour éviter les incidents en manipulant directement la mémoire), l'héritage multiple et la surcharge des opérateurs, ...
Java est fortement typé	toutes les variables sont typées et il n'existe pas de conversion automatique qui risquerait une perte de données. Si une telle conversion doit être réalisée, le développeur doit obligatoirement utiliser un cast ou une méthode statique fournie en standard pour la réaliser.
Java assure la gestion de la mémoire	l'allocation de la mémoire pour un objet est automatique à sa création et Java récupère automatiquement la mémoire inutilisée grâce au garbage collector qui restitue les zones de mémoire laissées libres suite à la destruction des objets.
Java est sûr	la sécurité fait partie intégrante du système d'exécution et du compilateur. Un programme Java planté ne menace pas le système d'exploitation. Il ne peut pas y avoir d'accès direct à la mémoire. L'accès au disque dur est réglementé dans une applet. Les applets fonctionnant sur le Web sont soumises aux restrictions suivantes dans la version 1.0 de Java : • aucun programme ne peut ouvrir, lire, écrire ou effacer un fichier sur le système de l'utilisateur • aucun programme ne peut lancer un autre programme sur le système de l'utilisateur • toute fenêtre créée par le programme est clairement identifiée comme étant une fenêtre Java, ce qui interdit par exemple la création d'une fausse fenêtre demandant un mot de passe • les programmes ne peuvent pas se connecter à d'autres sites Web que celui dont ils proviennent.
Java est économe	le pseudo code a une taille relativement petite car les bibliothèques de classes requises ne sont liées qu'à l'exécution.
Java est multitâche	il permet l'utilisation de threads qui sont des unités d'exécutions isolées. La JVM, elle même, utilise plusieurs threads.

Il existe 2 types de programmes avec la version standard de Java : les applets et les applications. Une application autonome (stand alone program) est une application qui s'exécute sous le contrôle direct du système d'exploitation. Une applet est une application qui est chargée par un navigateur et qui est exécutée sous le contrôle d'un plug in de ce dernier.

Les principales différences entre une applet et une application sont :

- les applets n'ont pas de méthode main() : la méthode main() est appelée par la machine virtuelle pour exécuter une application.
- les applets ne peuvent pas être testées avec l'interpréteur. Elles doivent être testées avec l'applet viewer ou doivent être intégrées à une page HTML, elle même visualisée avec un navigateur disposant d'un plug in Java, .

Remarque : l'API Applet est deprecated en Java 9.

1.2. Les logos de Java

Java a connu deux logos au cours de son histoire.

de 1996 à 2003

à partir de 2003

1.3. Un rapide historique de Java

Depuis sa première diffusion publique le 23 mai 1995, le langage et les plateformes Java ont été marqués par de nombreux événements dont les principaux sont :

Année	Evénements
1995	mai : premier lancement commercial du JDK 1.0
1996	janvier : JDK 1.0.1 septembre : lancement du JDC
1997	Java Card 2.0 février : JDK 1.1
1998	décembre : lancement de J2SE 1.2 et du JCP Personal Java 1.0
1999	décembre : lancement J2EE 1.2
2000	mai : J2SE 1.3
2001	J2EE 1.3
2002	février : J2SE 1.4
2003	J2EE 1.4
2004	septembre : J2SE 5.0
2005	Lancement du programme Java Champion
2006	mai : Java EE 5 décembre : Java SE 6.0
2007	Duke, la mascotte de Java est sous la licence Free BSD
2008	décembre : Java FX 1.0
2009	février : JavaFX 1.1 juin : JavaFX 1.2 décembre : Java EE 6
2010	janvier : rachat de Sun Microsystems par Oracle avril : JavaFX 1.3

2011	juillet : Java SE 7 octobre : JavaFX 2.0
2012	août : JavaFX 2.2
2013	juin : Java EE 7
2014	mars : Java SE 8, JavaFX 8
2017	septembre Java SE 9, Java EE 8
2018	mars : Java SE 10 septembre : Java SE 11
2019	mars : Java SE 12 septembre : Java SE 13, Jakarta EE 8
2020	mars : Java SE 14 septembre : Java SE 15 décembre : Jakarta EE 9
2021	mars : Java SE 16 mai : Jakarta EE 9.1 septembre : Java SE 17
2022	mars : Java SE 18, Jakarta EE 10 septembre : Java SE 19
2023	mars : Java SE 20 septembre : Java SE 21

1.4. Les différentes éditions et versions de Java

Sun puis Oracle ont toujours fourni gratuitement un ensemble d'outils et d'API pour permettre le développement de programmes avec Java. Ce kit, nommé JDK, est librement téléchargeable sur le site web d'Oracle :

https://www.oracle.com/java/technologies/

Le JRE (Java Runtime Environment) contient uniquement l'environnement d'exécution de programmes Java. Le JDK contient lui-même le JRE. Le JRE seul doit être installé sur les machines où des applications Java doivent être exécutées.

Depuis sa version 1.2, Java a été renommé Java 2. Les numéros de version 1.2 et 2 désignent donc la même version. Le JDK a été renommé J2SDK (Java 2 Software Development Kit) mais la dénomination JDK reste encore largement utilisée, à tel point que la dénomination JDK est reprise dans la version 5.0. Le JRE a été renommé J2RE (Java 2 Runtime Environment).

Trois plate-formes d'exécution (ou éditions) Java sont définies pour des cibles distinctes selon les besoins des applications à développer :

- Java Standard Edition (J2SE / Java SE) : environnement d'exécution et ensemble complet d'API pour des applications de type desktop. Cette plate-forme sert de base en tout ou partie aux autres plate-formes
- Java Enterprise Edition (J2EE / Java EE) : environnement d'exécution reposant intégralement sur Java SE pour le développement d'applications d'entreprises
- Java Micro Edition (J2ME / Java ME) : environnement d'exécution et API pour le développement d'applications sur appareils mobiles et embarqués dont les capacités ne permettent pas la mise en oeuvre de Java SE

La séparation en trois plate-formes permet au développeur de mieux cibler l'environnement d'exécution et de faire évoluer les plate-formes de façon plus indépendante.

Avec différentes éditions, les types d'applications qui peuvent être développées en Java sont nombreux et variés :

- Applications desktop
- Applications web : servlets/JSP, portlets, applets
- Applications pour appareil mobile (MIDP) : midlets

- Applications pour appareil embarqué (CDC) : Xlets
- Applications pour carte à puce (Javacard) : applets Javacard
- Applications temps réel

Sun fournit le JDK, à partir de la version 1.2, pour les plate-formes Windows, Solaris et Linux.

La version 1.3 de Java est désignée sous le nom Java 2 version 1.3.

La version 1.5 de Java est désignée officiellement sous le nom J2SE version 5.0.

La version 1.6 de Java est désignée officiellement sous le nom Java SE version 6.

La neuvième version de Java est numéroté Java 9 en remplacement du traditionnel 1.9.

La documentation au format HTML des API de Java est fournie séparément. Malgré sa taille, cette documentation est indispensable pour obtenir des informations complètes sur toutes les classes fournies. Le tableau ci-dessous résume la taille des différents composants selon leur version pour la plate-forme Windows.

Version	JDK Oracle		JRE Oracle		Documentation	
	compressé	installé	compressé	installé	compressé	installé
1.0						
1.1	8,6 Mo		12 Mo		16 Mo	83 Mo
1.2	20 Mo					
1.3	30 Mo	53 Mo	7 Mo	35 Mo	21 Mo	106 Mo
1.4	47 Mo	59 Mo		40 Mo	30 Mo	144 Mo
5.0	44 Mo		14 Mo		43,5 Mo	223 Mo
6	73 Mo		15,5 Mo		56 Mo	
7	140 Mo	219 Mo	42 Mo	106 Mo		263 Mo
8	151 Mo	365 Mo	46 Mo	136 Mo	90 Mo	345 Mo
9		498 Mo		214 Mo	68 Mo	393 Mo
10	399 Mo	461 Mo	100 Mo	226 Mo	66 Mo	378 Mo
11	171 Mo	278 Mo	-	-	50 Mo	290 Mo
12	179 Mo	307 Mo	-	-	49 Mo	264 Mo
13	187 Mo	304 Mo	-	-	48 Mo	266 Mo
14	164 Mo				50 Mo	
15	179 Mo				48 Mo	
16	170 Mo				49 Mo	
17	173 Mo				50 Mo	
18	173 Mo				50 Mo	
19	186 Mo				50 Mo	
20	188 Mo				50 Mo	
21	191 Mo				50 Mo	

1.4.1. Les évolutions des plates-formes Java

Les technologies Java évoluent au travers du JCP (Java Community Process). Le JCP est une organisation communautaire ouverte qui utilise des processus établis pour définir ou réviser les spécifications des technologies Java.

Les membres du JCP sont des personnes individuelles ou des membres d'organisations communautaires ou de sociétés commerciales qui tendent à mettre en adéquation la technologie Java avec les besoins du marché.

Bien que le JCP soit une organisation communautaire ouverte, Oracle (depuis son rachat de Sun Microsystems) est le détenteur des marques déposées autour de la technologie Java et l'autorité suprême concernant les plates-formes Java.

Des membres du JCP qui souhaitent enrichir la plate-forme Java doivent faire une proposition formalisée sous la forme d'une JSR (Java Specification Request). Chaque JSR suit un processus qui définit son cycle de vie autour de plusieurs étapes clés : drafts, review et approval.

Chaque JSR est sous la responsabilité d'un leader et traitée par un groupe d'experts.

Il est possible de souscrire à la liste de diffusion du JCP à l'url : https://jcp.org/en/participation/mail

Cette liste de diffusion permet d'être informé sur les évolutions des JSR et des procédures du JCP et de participer à des revues publiques ou de fournir des commentaires.

Le site du JCP propose une liste des JSR par plates-formes ou technologies :

- Java SE : https://jcp.org/en/jsr/tech?listBy=2&listByType=platform
- Java EE : https://jcp.org/en/jsr/tech?listBy=3&listByType=platform
- Java ME : https://jcp.org/en/jsr/tech?listBy=1&listByType=platform
- OSS : https://jcp.org/en/jsr/tech?listBy=3&listByType=tech
- JAIN : https://jcp.org/en/jsr/tech?listBy=2&listByType=tech
- XML : https://jcp.org/en/jsr/tech?listBy=1&listByType=tech

Une fois validée, chaque JSR doit proposer une spécification, une implémentation de référence (Reference Implementation) et un technology compatibility kit (TCK).

1.4.2. Les différentes versions de Java

Chaque version de la plate-forme Java possède un numéro de version et un nom de projet.

A partir de la version 5, la plate-forme possède deux numéros de version :

- Un numéro de version interne : exemple 1.5.0
- Un numéro de version externe : exemple 5.0

A partir de la version 9, la plate-forme ne possède plus qu'un numéro correspondant au numéro majeure.

Le nom de projet des versions majeures fait fréquemment référence à des oiseaux ou des mammifères. Le nom de projet des versions mineures concerne des insectes.

Les versions majeures de Java SE supérieures à la version 8, n'ont plus de nom de code.

Version	Nom du projet	Date de diffusion
JDK 1.0	Oak	Mai 1995
JDK 1.1		Février 1997
JDK 1.1.4	Sparkler	Septembre 1997
JDK 1.1.5	Pumpkin	Décembre 1997
JDK 1.1.6	Abigail	Avril 1998
JDK 1.1.7	Brutus	Septembre 1998
JDK 1.1.8	Chelsea	Avril 1999
J2SE 1.2	Playground	Décembre 1998

J2SE 1.2.1		Mars 1999
J2SE 1.2.2	Cricket	Juillet 1999
J2SE 1.3	Kestrel	Mai 2000
J2SE 1.3.1	Ladybird	Mai 2001
J2SE 1.4.0	Merlin	Février 2002
J2SE 1.4.1	Hopper	Septembre 2002
J2SE 1.4.2	Mantis	Juin 2003
J2SE 5.0 (1.5)	Tiger	Septembre 2004
Java SE 6.0 (1.6)	Mustang	Décembre 2006
Java SE 7 (1.7)	Dolphin	Juillet 2011
Java SE 8	Spider	Mars 2014
Java SE 9		Septembre 2017
Java SE 10		Mars 2018
Java SE 11		Septembre 2018
Java SE 12		Mars 2019
Java SE 13		Septembre 2019
Java SE 14		Mars 2020
Java SE 15		Septembre 2020
Java SE 16		Mars 2021
Java SE 17		Septembre 2021
Java SE 18		Mars 2022
Java SE 19		Septembre 2022
Java SE 20		Mars 2023
Java SE 21		Septembre 2023

1.4.3. La gestion des évolutions dans la plateforme Java via des JEPs

Le projet OpenJDK utilise le processus reposant sur des JDK Enhancement Proposals (JEPs) pour gérer le développement des fonctionnalités : proposer, discuter et spécifier de nouvelles fonctionnalités, améliorations ou modifications majeures à apporter à la plateforme Java.

Le processus des JEPs permet de faire évoluer la plateforme Java et de garantir que les nouvelles fonctionnalités répondent aux besoins grâce à une participation de la communauté Java.

La plupart des nouvelles fonctionnalités d'OpenJDK sont proposées sous la forme de JEPs. Une JEP peut concerner différentes fonctionnalités :

- La syntaxe du langage
- L'API de Java SE (module java.* ou jdk.incubator.* à partir de Java 9)
- L'API du JDK (module jdk.* à partir de Java 9)
- Un outil du JDK
- Une fonctionnalité de la JVM Hotspot

Les JEP sont aussi utilisées pour déprécier, améliorer ou retirer des fonctionnalités.

La JEP 0 contient une liste de toutes les JEPs : https://openjdk.org/jeps/0

La rédaction d'une JEP est un processus itératif qui peut faire évoluer le contenu du document en fonction des échanges et des retours. Son contenu peut être modifié tout au long du processus afin qu'une fois achevé, la JEP puisse servir de documentation officielle sur ce qui a été fait.

Les JEPs sont rédigées en Markdown.

Toutes les JEPs sont décrites dans un document qui respecte un modèle (template), défini dans la JEP 2. Certaines sections sont obligatoires (REQUIRED) ou optionnelles. L'ordre des sections doit être respecté. Il n'est pas possible d'ajouter des sections supplémentaires : seules celles définies dans le modèle peuvent être utilisées.

Les sections du modèle sont :

- Summary : (obligatoire) proposer un résumé court de la proposition, en une ou plusieurs phrases au maximum. Ce résumé est repris dans les listes de fonctionnalités, les JSR et d'autres documents
- History : fournir des informations sur l'historique de la fonctionnalité, notamment si elle a déjà été introduite dans des JEPs précédentes ainsi que les évolutions proposées dans la JEP par rapport à la précédente
- Goals : préciser les objectifs de la proposition
- Non-Goals : préciser les objectifs n'entrant pas dans le champ d'application de la proposition
- Success Metrics : préciser si le succès de la proposition peut être évalué par des mesures spécifiques et des objectifs associés
- Motivation : préciser ce qui motive la proposition (Pourquoi cela doit être effectué ? Quels sont les avantages ? Qui le demande ? ...)
- Description : (obligatoire) décrire en détail la proposition. Cette section peut évoluer au fil du temps, au fur et à mesure que le travail progresse, pour devenir à la fin la description de haut niveau du résultat final
- Alternatives : décrire si d'autres approches ou technologies ont été envisagées et si oui, décrire et expliquer pourquoi elles n'ont pas été retenues
- Testing : décrire les tests nécessaires pour valider la proposition
- Risks and Assumptions : décrire les risques ou les hypothèses qui doivent être pris en compte dans le cadre de cette proposition
- Dependencies : décrire les dépendances par rapport à d'autres JEPs, à des composants, à des produits ou à d'autres choses.

Chaque fonctionnalité importante du langage Java, de la JVM et de l'API Java SE commence par la rédaction d'une JEP qui passe par différentes phases de candidature et d'approbation avant d'être intégrée dans une version du JDK. Les travaux sur une JEP font évoluer son statut selon différents états.

Les trois premiers états d'une JEP sont :

- Draft : état initial dans lequel la JEP est rédigée, examinée, révisée et approuvée
- Submitted : le responsable de la JEP déclare que la JEP est prête à être évaluée
- Candidate : le lead OpenJDK ajoute la JEP à la roadmap du JDK

Une JEP à l'état Candidate n'est qu'une proposition méritant d'être examinée : il n'y a aucun engagement à ce qu'elle soit livrée dans une version particulière d'OpenJDK. A l'état Candidate, la JEP reçoit un numéro distinct.

À partir de ce moment, une JEP passera le plus souvent par plusieurs états :

- Proposed to Target : une proposition d'intégration dans une version d'OpenJDK
- Targeted : la JEP est ciblée pour une version d'OpenJDK
- Integrated : après l'intégration du code et des tests unitaires dans la base de code principale de la version d'OpenJDK cible
- Complete : après que les tests fonctionnels, de performance et de conformité ont été effectués et que la fonctionnalité a atteint le point où seules des corrections de bogues sont attendues
- Closed/Delivered : lorsque la release est publiée

Une release, dans son ensemble, n'est considérée comme complète qu'une fois que toutes ses JEP relatives à des fonctionnalités ont atteint l'état Complete.

Il arrive parfois que le traitement d'une JEP soit interrompu entre l'état Draft et Closed/Delivered grâce à différents états :

- Closed/Rejected : pour rejeter la JEP à l'état Submitted ou Candidate car elle ne vaut pas la peine d'être poursuivie ou qu'il est peu probable qu'elle soit ciblée et qu'il ne vaut pas la peine d'être maintenue dans la roadmap

- Closed/Withdrawn : pour retirer la JEP à l'état Draft, Submitted, Candidate ou Proposed-to-Target car plus aucun travail ne soit réalisé dessus
- Proposed to Drop : pour retirer la JEP de la version ciblée d'OpenJDK. Si le retrait est validé, la JEP repasse à l'état précédent ou à l'état Candidate

1.4.4. Java 1.0

Cette première version est lancée officiellement en mai 1995.

Elle se compose de 8 packages :

- java.lang : classes de bases
- java.util : utilitaires
- java.applet : applets
- java.awt : interface graphique portable
- java.awt.image : manipulation d'images
- java.awt.peer : interaction avec le système d'exploitation pour l'affichage de composants graphiques
- java.io : gestion des entrées/sorties grâce aux flux
- java.net : gestion des communications à travers le réseau

1.4.5. Java 1.1

Cette version du JDK est annoncée officiellement en mars 1997. Elle apporte de nombreuses améliorations et d'importantes fonctionnalités nouvelles dont :

- les Java beans
- les fichiers JAR
- RMI pour les objets distribués
- la sérialisation
- l'introspection
- JDBC pour l'accès aux données
- les classes internes (inner class)
- l'internationalisation
- un nouveau modèle de sécurité permettant notamment de signer les applets
- un nouveau modèle de gestion des événements
- JNI pour l'appel de méthodes natives
- ...

1.4.6. Java 1.2 (nom de code Playground)

Cette version du JDK est lancée fin 1998. Elle apporte de nombreuses améliorations et d'importantes fonctionnalités nouvelles dont :

- un nouveau modèle de sécurité basé sur les policy
- les JFC sont incluses dans le JDK (Swing, Java2D, accessibility, drag & drop ...)
- JDBC 2.0
- les collections
- support de CORBA
- un compilateur JIT est inclus dans le JDK
- de nouveaux formats audio sont supportés
- ...

Java 2 se décline en 3 éditions différentes qui regroupent des APIs par domaine d'applications :

- Java 2 Micro Edition (J2ME) : contient le nécessaire pour développer des applications capables de fonctionner dans des environnements limités tels que les assistants personnels (PDA), les téléphones portables ou les systèmes de navigation embarqués
- Java 2 Standard Edition (J2SE) : contient le nécessaire pour développer des applications et des applets. Cette édition reprend le JDK 1.0 et 1.1.
- Java 2 Enterprise Edition (J2EE) : contient un ensemble de plusieurs API permettant le développement d'applications destinées aux entreprises tel que JDBC pour l'accès aux bases de données, EJB pour développer des composants orientés métiers, Servlet / JSP pour générer des pages HTML dynamiques, ... Cette édition nécessite le J2SE pour fonctionner.

Le but de ces trois éditions est de proposer une solution reposant sur Java quelque soit le type de développement à réaliser.

1.4.7. J2SE 1.3 (nom de code Kestrel)

Cette version du JDK est lancée en mai 2000. Elle apporte de nombreuses améliorations notamment sur les performances et des fonctionnalités nouvelles dont :

- JNDI est inclus dans le JDK
- hotspot est inclus dans la JVM
- JPDA
- ...

La rapidité d'exécution a été grandement améliorée dans cette version.

1.4.8. J2SE 1.4 (nom de code Merlin)

Cette version du JDK, lancée début 2002, est issue des travaux de la JSR 59. Elle apporte de nombreuses améliorations notamment sur les performances et des fonctionnalités nouvelles dont :

- JAXP est inclus dans le JDK pour le support de XML
- JDBC version 3.0
- new I/O API pour compléter la gestion des entrées/sorties de manière non bloquante
- logging API pour la gestion des logs applicatives
- une API pour utiliser les expressions régulières
- une API pour gérer les préférences utilisateurs
- JAAS est inclus dans le JDK pour l'authentification
- un ensemble d'API pour utiliser la cryptographie
- les exceptions chainées
- l'outil Java WebStart
- ...

Cette version ajoute un nouveau mot clé au langage pour utiliser les assertions : assert.

1.4.9. J2SE 5.0 (nom de code Tiger)

La version 1.5 de J2SE est spécifiée par le JCP sous la JSR 176. Elle intègre un certain nombre de JSR dans le but de simplifier les développements en Java.

Ces évolutions sont réparties dans une quinzaine de JSR qui sont intégrées dans la version 1.5 de Java.

JSR-003	JMX Management API
JSR-013	Decimal Arithmetic
JSR-014	Generic Types

JSR-028	SASL
JSR-114	JDBC API Rowsets
JSR-133	New Memory Model and thread
JSR-163	Profiling API
JSR-166	Concurrency Utilities
JSR-174	Monitoring and Management for the JVM
JSR-175	Metadata facility
JSR-199	Compiler APIs
JSR-200	Network Transfer Format for Java Archives
JSR-201	Four Language Updates
JSR-204	Unicode Surrogates
JSR-206	JAXP 1.3

La version 1.5 de Java est désignée officiellement sous le nom J2SE version 5.0.

La version 1.5 de Java apporte de nombreuses évolutions qui peuvent être classées dans deux catégories :

- Les évolutions sur la syntaxe du langage
- Les évolutions sur les API : mises à jour d'API existantes, intégration d'API dans le SDK

Depuis sa première version et jusqu'à sa version 1.5, le langage Java lui-même n'a que très peu évolué : la version 1.1 a ajouté les classes internes et la version 1.4 les assertions.

Les évolutions de ces différentes versions concernaient donc essentiellement les API de la bibliothèque standard (core) de Java.

La version 1.5 peut être considérée comme une petite révolution pour Java car elle apporte énormément d'améliorations sur le langage :

- L'autoboxing et l'unboxing
- Les importations statiques
- Les annotations ou métadonnées (Meta Data)
- Les arguments variables (varargs)
- Les generics
- Les boucles for évoluées pour le parcours des Iterator et des tableaux
- Les énumérations (type enum)

Toutes ces évolutions sont déjà présentes dans différents autres langages notamment C#.

Le but principal de ces ajouts est de faciliter le développement d'applications avec Java en simplifiant l'écriture et la lecture du code.

La technologie Pack200 permet de compresser les fichiers .jar pour obtenir un réduction du volume pouvant atteindre 60%.

1.4.10. Java SE 6 (nom de code Mustang)

Cette version est spécifiée par le JCP sous la JSR 270 et développée sous le nom de code Mustang.

Elle intègre un changement de dénomination et de numérotation : la plate-forme J2SE est renommée en Java SE, SE signifiant toujours Standard Edition.

1.4.10.1. Les évolutions de Java 6

Cette version inclut plusieurs JSR :

JSR 105	XML Digital Signature APIs
JSR 173	Streaming API for XML
JSR 181	Web Services Metadata for Java Platform
JSR 199	Java Compiler API
JSR 202	Java Class File Specification Update
JSR 221	JDBC 4.0 API Specification
JSR 222	Java Architecture for XML Binding (JAXB) 2.0
JSR 223	Scripting for the Java Platform
JSR 224	Java API for XML-Based Web Services (JAX-WS) 2.0
JSR 250	Common Annotations for the Java Platform
JSR 269	Pluggable Annotation Processing API

Elle apporte donc plusieurs améliorations :

L'amélioration du support XML

Java 6.0 s'est enrichie avec de nombreuses nouvelles fonctionnalités concernant XML :

- JAXP 1.4
- JSR 173 Streaming API for XML (JSR 173)
- JSR 222 Java Architecture for XML Binding (JAXB) 2.0 : évolution de JAXB qui utilise maintenant les schémas XML
- Le support du type de données XML de la norme SQL 2003 dans JDBC

JDBC 4.0

Cette nouvelle version de l'API JDBC est le fruit des travaux de la JSR 221. Elle apporte de nombreuses évolutions :

- chargement automatique des pilotes JDBC
- Support du type SQL ROWID et XML
- Amélioration du support des champs BLOB et CLOB
- L'exception SQLTransaction possède deux classes filles SQLTransientException et SQLNonTransientException

Le support des services web

Les services web font leur apparition dans la version SE de Java : précédemment ils n'étaient intégrés que dans la version EE. Plusieurs JSR sont ajoutés pour supporter les services web dans la plate-forme :

- JSR 224 Java API for XML based Web Services (JAX-WS) 2.0 : facilite le développement de services web et propose un support des standards SOAP 1.2, WSDL 2.0 et WS-I Basic Profile 1.1
- JSR 181 Web Services MetaData : définit un ensemble d'annotations pour faciliter le développement de services web
- JSR 105 XML Digital Signature API : la package javax.xmlcrypto contient une API qui implémente la spécification Digital Signature du W3C. Ceci permet de proposer une solution pour sécuriser les services web

Le support des moteurs de scripts

L'API Scripting propose un standard pour l'utilisation d'outils de scripting. Cette API a été développée sous la JSR 223. La plate-forme intègre Rhino un moteur de scripting Javascript

L'amélioration de l'intégration dans le système d'exploitation sous-jacent

- La classe java.awt.SystemTray permet d'interagir avec la barre d'outils système (System Tray)
- La classe java.awt.Desktop qui permet des interactions avec le système d'exploitation (exécution de fonctionnalités de base sur des documents selon leur type)
- Nouveaux modes pour gérer la modalité d'une fenêtre
- Amélioration dans Swing
- Amélioration des look and feel Windows et GTK
- la classe javax.awt.SplashScreen qui propose un support des splashscreens
- Ajout du layout javax.swing.GroupLayout
- Filtres et tris des données dans le composant Jtable
- La classe SwingWorker qui facilite la mise en oeuvre de threads dans Swing
- Utilisation du double buffering pour l'affichage

Les améliorations dans l'API Collection

- 5 nouvelles interfaces : Deque (queue à double sens), BlockingDeque, NavigableSet (étend SortedSet), NavigableMap (étend SortedMap) et ConcurrentNavigableMap
- ArrayDeque : implémentation de Deque utilisant un tableau
- ConcurrentSkipListSet : implémentation de NavigableSet
- ConcurrentSkipListMap : implémentation de ConcurrentNavigableMap
- LinkedBlockingDeque : implémentation de BlockingDeque

L'améliorations dans l'API IO

- Modification des attributs d'un fichier grâce aux méthodes setReadable(), setWritable() et setExecutable() de la classe File

Java Compiler API

Cette API est le résultat des travaux de la JSR 199 et a pour but de proposer une utilisation directe du compilateur Java. Cette API est utilisable à partir du package javax.tools

Pluggable Annotation-Processing API

Cette API est le résultat des travaux de la JSR 269 et permet un traitement des annotations à la compilation. Cette API est utilisable à partir du package javax.annotation.processing

Common Annotations

Cette API est le résultat des travaux de la JSR 250 et définit plusieurs nouvelles annotations standards.
@javax.annotation.Generated : permet de marquer une classe, une méthode ou un champ comme étant généré par un outil

@javax.annotation.PostConstruct : méthode exécutée après la fin de l'injection de dépendance
@javax.annotation.PreDestroy : méthode de type callback appelée juste avant d'être supprimée par le conteneur
@javax.annotation.Resource : permet de déclarer une référence vers une ressource
@javax.annotation.Resources : conteneur pour la déclaration de plusieurs ressources

Java Class File Specification

Issue des travaux de la JSR 202, cette spécification fait évoluer le format du fichier .class résultant de la compilation.

La vérification d'un fichier .class exécute un algorithme complexe et coûteux en ressources et en temps d'exécution pour valider un fichier .class.

La JSR 202, reprend une technique développée pour le profile CLDC de J2ME nommée split vérification qui décompose la vérification d'un fichier .class en deux étapes :

- la première étape réalisée lors de la création du fichier .class ajoute des attributs qui seront utilisés par la seconde étape
- la seconde étape est réalisée à l'exécution en utilisant les attributs

Le temps de chargement du fichier .class est ainsi réduit.

Le Framework JavaBeans Activation

Le Framework JavaBeans Activation a été intégré en standard dans la plate-forme Java SE 6. Ce framework historiquement fourni séparément permet de gérer les types mimes et était généralement utilisé avec l'API JavaMail. Ce framework permet d'associer des actions à des types mimes.

La liste des nouveaux packages de Java 6 comprend :

java.text.spi	
java.util.spi	
javax.activation	Activation Framework
javax.annotation	Traitement des annotations
javax.jws	Support des services web
javax.jws.soap	support SOAP
javax.lang.model.*	
javax.script	Support des moteurs de scripting
javax.tools	Accès à certains outils notamment le compilateur
javax.xml.bind.*	JAXB
javax.xml.crypto.*	Cryptographie avec XML
javax.xml.soap	Support des messages SOAP
javax.xml.stream.*	API Stax
javax.xml.ws.*	API JAX-WS

Une base de données nommée JavaDB est ajoutée au JDK 6.0 : c'est une version de la base de données Apache Derby.

1.4.10.2. Java 6 update

En attendant la version 7, Sun puis Oracle ont proposé plusieurs mises à jour de la plate-forme Java SE. Ces mises à jour concernent :

- des corrections de bugs et ou de sécurité
- des évolutions ou des ajouts dans les outils du JDK et du JRE notamment la machine virtuelle HotSpot

Deux de ces mises à jour sont particulièrement importantes : update 10 et 14.

Java 6 update 1

Cette mise à jour contient des corrections de bugs.

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/6u1.html

Java 6 update 2

Cette mise à jour contient des corrections de bugs et une nouvelle version de la base de données embarquée Java DB.

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/6u2.html

Java 6 update 3

Cette mise à jour contient des corrections de bugs.

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u3.html

Java 6 update 4

Cette mise à jour contient des corrections de bugs et la version 10.3 de Java DB.

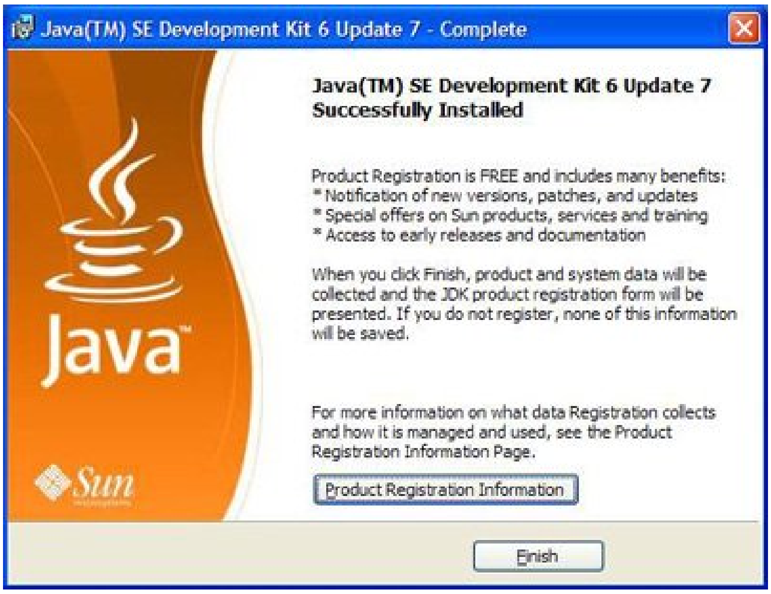
La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u4.html

Java 6 update 5

Cette mise à jour contient des corrections de bugs et la possibilité d'enregistrer le JDK.

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u5.html

Depuis la version 6u5 de Java, le programme d'installation du JDK propose à la fin la possibilité d'enregistrer le JDK.


Java(TM) SE Development Kit 6 Update 7 - Complete

Java(TM) SE Development Kit 6 Update 7 Successfully Installed

Product Registration is FREE and includes many benefits:
* Notification of new versions, patches, and updates
* Special offers on Sun products, services and training
* Access to early releases and documentation

When you click Finish, product and system data will be collected and the JDK product registration form will be presented. If you do not register, none of this information will be saved.

For more information on what data Registration collects and how it is managed and used, see the Product Registration Information Page.

Il suffit de cliquer sur le bouton « Product Registration Information » pour obtenir des informations sur le processus d'enregistrement du produit.

Lors du clic sur le bouton « Finish », le processus d'enregistrement collecte les informations sur le JDK installé et sur le système hôte. Ces informations sont envoyées via une connexion http sécurisée sur le serveur Sun Connection.

Le navigateur s'ouvre sur la page d'enregistrement du JDK.

Il faut utiliser son compte SDN (Sun Developer Network) pour se logguer et afficher la page « Thank You ».

Il est possible d'enregistrer son JDK en ouvrant la page register.html située dans le répertoire d'installation du JDK.

En plus du JDK, plusieurs autres produits de Sun Connection peuvent être enregistrés comme GlassFish, Netbeans, ...

Sun Connection propose un service gratuit nommé Inventory Channel qui permet de gérer ses produits enregistrés.

Java 6 update 6

Cette mise à jour contient des corrections de bugs.

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u6.html

Java 6 update 7

Cette mise à jour contient des corrections de bugs et l'outil Java Visual VM.

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u7.html

Le numéro de version interne complet est build 1.6.0_07-b06. Le numéro de version externe est 6u7.

Exemple :

```
C:\>java -version
java version "1.6.0_07"
Java(TM) SE Runtime Environment (build 1.6.0_07-b06)
Java HotSpot(TM) Client VM (build 10.0-b23, mixed mode, sharing)
```

Java 6 update 10

Cette mise à jour qui a porté le nom Java Update N ou Consumer JRE est très importante car elle apporte de grandes évolutions notamment pour le support des applications de type RIA.

Elle contient :

- Un nouveau plug-in pour l'exécution des applets dans les navigateurs
- Nimbus, un nouveau L&F pour Swing
- Java kernel
- Java Quick Starter (JQS)
- des corrections de bugs
- la version 10.4 Java DB
- support par défaut de Direct3D 9 sous Windows.

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u10.html

Le plug-in pour les navigateurs a été entièrement réécrit notamment pour permettre une meilleure exécution des applets, des applications Java Web Start et des applications RIA en Java FX.

Java 6 update 11

Cette mise à jour ne contient que des corrections de bugs et des patchs de sécurité.

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u11.html

La version externe est Java 6u11, le numéro de build est 1.6.0_11-b03.

Java 6 update 12

Cette mise à jour contient

- Support de Windows Server 2008
- Amélioration des performances graphiques et de démarrage des applications Java Web Start
- des corrections de bugs.

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u12.html

La version externe est Java 6u12, le numéro de build est 1.6.0_12-b04.

Java 6 update 13

Cette mise à jour contient des corrections de bugs.

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u13.html

La version externe est Java 6u13, le numéro de build est 1.6.0_13-b03.

L'installeur propose par défaut l'installation de la barre d'outils de Yahoo.

Java 6 update 14

Cette mise à jour contient

- La version 14 de la JVM HotSpot
- G1, le nouveau ramasse miette

- Les versions 2.1.6 de JAX-WS et 2.1.10 de JAX-B
- La version 10.4.2.1 de Java DB
- Des mises à jour dans Visual VM
- Blacklist des jars signés qui contiennent une faille de sécurité
- des corrections de bugs.

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u14.html

La version externe est Java 6u14, le numéro de build est 1.6.0_14-b08.

Java 6 update 15

Cette mise à jour ne contient que des corrections de bugs et patchs de sécurité.

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u15.html

La version externe est Java 6u15, le numéro de build est 1.6.0_15-b03.

Java 6 update 16

Cette mise à jour ne contient que des corrections de bugs.

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u16.html

La version externe est Java 6u16, le numéro de build est 1.6.0_16-b01.

Java 6 update 17

Cette mise à jour ne contient que des corrections de bugs.

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u17.html

La version externe est Java 6u17, le numéro de build est 1.6.0_17-b04.

Java 6 update 18

Cette mise à jour propose le support de Windows 7, Ubuntu 8.04 et Red Hat Enterprise Linux 5.3

Elle inclut la version 1.2 de Visual VM, la version 10.5.3.0 de Java DB, la version 16.0 de la machine virtuelle HotSpot, une mise à jour de Java Web Start.

Cette mise à jour contient aussi des corrections de bugs et une amélioration des performances.

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u18.html

La version externe est Java 6u18, le numéro de build est 1.6.0_18-b07.

Java 6 update 19

Cette mise à jour ne contient que des corrections de bugs.

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u19.html

La version externe est Java 6u19, le numéro de build est 1.6.0_19-b04.

Java 6 update 20

Cette mise à jour ne contient que des corrections de bugs.

L'attribut codebase de JNLP est obligatoire.

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u20.html

La version externe est Java 6u20, le numéro de build est 1.6.0_20-b02.

Java 6 update 21

Cette mise à jour propose le support de Oracle Enterprise Linux 4.8, 5.4 et 5.5, Red Hat Enterprise Linux 5.5 et 5.4.

Elle inclut la version 1.2.2 de Visual VM et la version 17.0 de la machine virtuelle HotSpot.

La propriété Compagny Name est modifiée : "Sun Microsystem" est remplacé par "Oracle" dans la version 1.6.0_21_b6. Comme cette modification a posé des soucis pour le lancement d'Eclipse, la modification a été annulée dans la version 1.6.0_21_b7 pour Windows.

Cette mise à jour contient aussi des corrections de bugs.

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u21.html

La version externe est Java 6u21, le numéro de build est 1.6.0_21-b02.

Java 6 update 22

Cette mise à jour contient des corrections de bugs et patchs de sécurité.

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u22.html

La version externe est Java 6u22, le numéro de build est 1.6.0_22-b04.

Java 6 update 23

Cette mise à jour contient aussi des corrections de bugs.

Elle inclut la version 1.3.1 de Visual VM et la version 19.0 de la machine virtuelle HotSpot.

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u23.html

La version externe est Java 6u23, le numéro de build est 1.6.0_23-b05.

Java 6 update 24

Cette mise à jour contient des corrections de bugs, des patchs de sécurité, une amélioration des performances, la version 20 de la JVM HotSpot et le support pour les navigateurs IE9, Firefox 4 et Chrome 10.

La version 20 de la JVM Hotspot propose plusieurs améliorations :

l'option -XX:+TieredCompilation permet d'activer la tiered compilation dans le mode server

amélioration des informations de diagnostique

L'option -XX:+AggressiveOpts permet d'améliorer les performances de la classe BigDecimal

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u24.html

La version externe est Java 6u24, le numéro de build est 1.6.0_24-b07.

Java 6 update 25

Cette mise à jour contient des corrections de bugs et patchs de sécurité. L'outil Java DB est mis à jour en version 10.6.2.1.

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u25.html

La version externe est Java 6u25, le numéro de build est 1.6.0_25-b06.

Java 6 update 26

Cette mise à jour contient des corrections de bugs et patchs de sécurité. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u26.html

La version externe est Java 6u26, le numéro de build est 1.6.0_26-b03.

Java 6 update 27

Cette mise à jour contient des corrections de bugs : cette version est certifiée pour une utilisation avec Firefox 5, Oracle Linux 5.6 et Red Hat Enterprise Linux 6.0.

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u27.html

La version externe est Java 6u27, le numéro de build est 1.6.0_27-b07.

La version Java 6 update 28 n'a jamais été publiée.

Java 6 update 29

Cette mise à jour contient des corrections de bugs et patchs de sécurité. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u29.html

La version externe est Java 6u29, le numéro de build est 1.6.0_29-b11.

Java 6 update 30

Cette mise à jour contient des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u30.html

Elle inclut le support pour Red Hat Enterprise Linux 6, la version 1.3.2 de Visual VM et l'implémentation du synthetizer open source Gervill, fournie aussi dans Java 7 qui est activable en utilisant l'option -Dsun.sound.useNewAudioEngine=true.

La version externe est Java 6u30, le numéro de build est 1.6.0_30-b12.

Java 6 update 31

Cette mise à jour contient 14 patchs de sécurité. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u31.html

La version externe est Java 6u31, le numéro de build est 1.6.0_31-b4.

Java 6 update 32

Cette mise à jour contient des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u32.html

La version externe est Java 6u32, le numéro de build est 1.6.0_32-b05.

Java 6 update 33

Cette mise à jour contient des corrections de bugs et des patches de sécurité. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u33.html

La version externe est Java 6u33, le numéro de build est 1.6.0_33-b04.

Java 6 update 34

Cette mise à jour contient des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u34.html

La version externe est Java 6u34, le numéro de build est 1.6.0_34-b04.

Java 6 update 35

Cette mise à jour contient un patch de sécurité critique (Oracle Security Alert CVE-2012-4686). La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u35.html

La version externe est Java 6u35, le numéro de build est 1.6.0_35-b10.

Java 6 update 37

Cette mise à jour contient des patches de sécurité. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u37.html

La version externe est Java 6u37, le numéro de build est 1.6.0_37-b06.

Java 6 update 38

Cette mise à jour contient des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u38.html

La version externe est Java 6u38, le numéro de build est 1.6.0_38-b05.

Java 6 update 39

Cette mise à jour contient des patches de sécurité. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u39.html

La version externe est Java 6u39, le numéro de build est 1.6.0_39-b04.

Java 6 update 41

Cette mise à jour contient des patches de sécurité. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u41.html

La version externe est Java 6u41, le numéro de build est 1.6.0_41-b02.

Java 6 update 43

Cette mise à jour contient des patches de sécurité. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u43.html

Cette version est certifiée pour une utilisation avec Windows Server 2012 (64-bit). Cette mise à jour a été annoncée comme étant la dernière diffusée gratuitement : le support est toujours possible en souscrivant au programme payant Java SE Support d'Oracle.

La version externe est Java 6u43, le numéro de build est 1.6.0_43-b01.

Java 6 update 45

Cette mise à jour contient des patches de sécurité. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/javase-6u45.html

La valeur par défaut de la propriété java.rmi.server.useCodebaseOnly depuis cette version est à true.

Sous Windows, la manière d'analyser la commande fournie en paramètre à la méthode exec() de la classe Runtime a été modifiée notamment si le chemin contient un espace.

La version externe est Java 6u45, le numéro de build est 1.6.0_45-b06. Elle est diffusée le 16 avril 2013.

Les mises à jour suivantes ne sont pas diffusées publiquement mais uniquement grâce au programme de support payant de Java SE.

1.4.11. Java SE 7 (nom de code Dolphin)

Java SE 7 est issu des travaux commencés par Sun et poursuivis par Oracle, des travaux du JCP et des travaux d'implémentation du projet open source OpenJDK.

La version 7 de Java est une version évolutive qui propose quelques fonctionnalités intéressantes mais surtout c'est une version qui réouvre la voie aux évolutions du langage et de la plate-forme Java après plus de quatre années d'attente depuis la version 6.

Java SE 7 propose des évolutions sur toute la plate-forme :

- Le langage Java évolue grâce au projet Coin
- Les API du JDK
- La JVM

Java SE 7 assure une forte compatibilité ascendante avec les versions antérieures de la plate-forme car les nouvelles fonctionnalités sont des ajouts. Ceci permet de préserver les compétences des développeurs et les investissements qui ont pu être faits dans la technologie Java.

Il y a cependant quelques petites contraintes dont il faut tenir compte lors d'une migration. Celles-ci sont consultables à l'url :
https://www.oracle.com/java/technologies/compatibility.html

Lors de sa sortie, Java SE 7 est déjà utilisable avec les principaux IDE du marché : Netbeans 7.0, Eclipse Indigo (avec un plug-in dédié en attendant son inclusion par défaut dans l'outil à partir de la version 7.0.1) et Intellij IDEA 10.5.

Java SE 7 est la première version majeure diffusée depuis le rachat de Sun par Oracle.

La base de Java SE 7 est le projet open source OpenJDK : c'est la première fois qu'une version de Java SE repose majoritairement sur une implémentation open source et des contributions externes comme le framework fork/join.

Avant la création du projet Open JDK, certaines fonctionnalités de la plate-forme Java étaient des implémentations propriétaires voire commerciales notamment le système de rendu des polices, le système de gestion des sons, certains algorithmes d'encryptage, ... toutes ces implémentations ont dû être remplacées par des solutions open source qui soient compatibles avec la licence GPL.

Ainsi, Java SE 7 utilise un nouveau synthétiseur MIDI open source en remplacement du synthétiseur propriétaire existant : le Gervill Software Synthesizer. Ce nouveau système est beaucoup plus performant que son prédécesseur.

Java SE version 7 a été spécifié par le JCP sous la JSR 336. Cette version se focalise sur quatre thèmes :

- Productivité : gestion automatique des ressources de type I/O, le projet Coin
- Performance : nouvelle API de Gestion de la concurrence
- Universalité : nouvelle instruction dans le bytecode et nouvelle API d'invocation de méthodes (invoke dymamic)
- Intégration : NIO 2

Java SE 7 intègre plusieurs spécifications :

- JSR 203 : More New I/O APIs for the Java Platform ("NIO.2"), API I/O asynchrone permettant aussi une utilisation avancée du système de fichiers (permissions, attributs de fichier, parcours de répertoires, événements sur un répertoire, ...)
- JSR 292 : Invoke Dynamic qui a pour but de faciliter la mise en oeuvre dans la JVM de langages typés dynamiquement, comme Groovy, permettant une amélioration de leurs performances
- JSR 334 : Small Enhancements to the Java Programming Language (OpenJDK Project Coin), quelques petites améliorations dans la syntaxe du langage : utilisation des strings dans l'instruction switch, gestion automatique des ressources dans l'instruction try, l'opérateur diamant, amélioration de la lisibilité des valeurs numériques, support des exceptions multiples dans une clause catch.
- JSR 166y : Fork/Join framework est un framework qui a pour but de faciliter le découpage et l'exécution de traitements sous la forme de tâches exécutées en parallèle grâce à une utilisation de plusieurs processeurs.

Java SE 7 intègre aussi plusieurs améliorations et évolutions :

- Support d'Unicode version 6.0
- Support de TLS 1.2
- Implémentation de l'algorithme Elliptic-Curve Cryptography (ECC)
- Mise à jour de JDBC version 4.1 et RowSet version 1.1
- Le look and feel Nimbus dans Swing
- Le composant javax.swing.JLayer issu des travaux du composant JXLayer du projet SwingLabs
- Mises à jour des API relatives à XML : JAXP 1.4, JAXB 2.2a et JAX-WS 2.2 correspondant à des révisions des JSR 206 (Java API for XML Processing (JAXP)) , JSR 222: (Java Architecture for XML Binding (JAXB)) et JSR 224 (Java API for XML-Based Web Services (JAX-WS))
- Amélioration du monitoring avec des évolutions des MBeans : OperatingSystemMXBean concernant la charge CPU globale du système et celle de la JVM et GarbageCollectorMXBean pour envoyer des notifications lors de l'exécution du ramasse-miettes
- Amélioration de l'internationalisation : la classe java.util.Locale propose un support des normes IETF BCP 47 (Tags for Identifying Languages) et UTR 35 (Local Data Markup Language)
- Ajout du support des protocoles SCTP (Stream Control Transmission Protocol) et SDP (Sockets Direct Protocol) pour Solaris et Linux
- Transparence et forme libre pour les fenêtres Swing
- La méthode close() de la classe UrlClassLoader permet de libérer les ressources chargées
- La classe Objects
- Une nouvelle feuille de style plus moderne pour la Javadoc
- ...

Oracle a commencé une fusion de ses JVM JRockit et HotSpot : la JVM de Java 7 possède la première version de cette fusion. Les prochaines mises à jour de Java 7 devraient fournir le travail de cette fusion : parvenir à la suppression de la PermGen et contenir certaines fonctionnalités de JRockit.

Plusieurs fonctionnalités initialement prévues pour Java SE 7 sont reportées dans Java SE 8 :

- La modularité (Project Jigsaw)
- Les closures (Project Lambda)
- L'amélioration des annotations
- La JSR-296 : Swing Framework

La sortie de Java SE 7 a eu lieu de 28 juillet 2011.

Lors de cette sortie, Java 7 contient un bug dans les optimisations faites par le compilateur Hotspot qui compile de façon erronée certaines boucles. Ce bug peut se traduire par un plantage de la JVM ou des résultats erronés ce qui est plus grave encore. Certains projets open source notamment Lucene et Solr sont affectés par ce bug.

Une solution de contournement de ce bug est de désactiver l'optimisation des boucles en utilisant l'option -XX:-UseLoopPredicate de la JVM.

Ce bug peut aussi survenir avec Java SE 6 dans une JVM Sun où les options -XX:+OptimizeStringConcat ou -XX:+AggressiveOpts sont utilisées.

1.4.11.1. Les JSR de Java 7

Cette version inclut plusieurs JSR :

JSR 166y	Fork / Join
JSR 203	NIO 2
JSR 292	Invoke dynamic
JSR 334	Project Coin

Elle apporte donc plusieurs améliorations :

Le projet Coin

Java 7.0 propose quelques évolutions syntaxiques. Le projet Coin propose des améliorations au langage Java pour augmenter la productivité des développeurs et simplifier certaines tâches de programmation courantes. Il s'agit d'une part de réduire la quantité de code nécessaire et d'autre part de rendre ce code plus facile à lire en utilisant une syntaxe plus claire.

Le projet Coin a été développé par le JCP sous la JSR 334: "Small Enhancements to the Java Programming Language".

Le projet Coin a été développé et implémenté dans le cadre de l'open JDK, tout d'abord sous la forme d'un appel à contribution d'idées de la part de la communauté. Toutes les idées retenues ne sont pas proposées dans Java SE 7, certaines seront implémentées dans Java SE 8. Une première partie du projet Coin est incluse dans Java SE 7 l'autre partie sera intégrée dans Java SE 8.

Les fonctionnalités du projet Coin incluses dans Java 7 peuvent être regroupées en trois parties :

1) Simplifier l'utilisation des generics

- l'opérateur diamant
- la suppression possible des avertissements lors de l'utilisation des varargs

2) Simplifier la gestion des erreurs

- la prise en compte de plusieurs exceptions dans la clause catch
- l'opérateur try-with-resources

3) Simplifier l'écriture du code

- l'utilisation des d'objets de type String dans l'opérateur switch
- faciliter la lecture des valeurs littérales

Le but du projet Coin est de proposer quelques améliorations au niveau du langage Java :

- Les entiers exprimés en binaire (Binary Literals) : les types entiers (byte, short, int, et long) peuvent être exprimés dans le système binaire en utilisant le préfixe 0b ou 0B

Exemple (code Java 7) :

```
    int valeurInt = 0b1000;
```

- Utilisation des underscores dans les entiers littéraux : il est possible d'utiliser le caractère tiret bas (underscore) entre les chiffres qui composent un entier littéral. Ceci permet de faire des groupes de chiffres pour par exemple séparer les milliers, les millions, les milliards, ... afin d'améliorer la lisibilité du code

Exemple (code Java 7) :

```
    int maValeur = 123_1456_789;
```

- Utilisation d'objets de type Strings dans l'instruction Switch : il est possible d'utiliser un objet de type String dans l'expression d'une instruction Switch
- L'opérateur diamant (diamond operator) : lors de l'instanciation d'un type utilisant les generic, il n'est plus obligatoire de repréciser le type generics au niveau du constructeur mais de simplement utiliser l'opérateur diamant « <> » tant que le compilateur est en mesure de déterminer le type generic
- Le mot clé try-with-resources : il permet de déclarer une ou plusieurs ressources. Une ressource est un objet qui a besoin d'être fermé lorsqu'il n'est plus utilisé. Le mot clé try-with-resources garantit que chaque ressource sera fermée lorsqu'elle n'est plus utilisée. Une ressource et un objet qui implémente l'interface java.lang.AutoCloseable. Plusieurs classes du JDK implémentent l'interface AutoClosable : java.io.InputStream, OutputStream, Reader, Writer, java.sql.Connection, Statement et ResultSet. Il est donc possible d'utiliser une instance de ces interfaces avec le mot clé try-with-resources.
- Il est possible de capturer plusieurs exceptions dans une même clause catch.
- Le compilateur de Java 7 détermine plus précisément les exceptions qui peuvent être levées dans le bloc try. Il est capable de vérifier la clause throws lorsque ces exceptions sont propagées dans un catch et ce,

indépendamment du type utilisé pour les capturer.
- Il est possible de demander au compilateur de ne plus émettre de warnings lors de l'utilisation de varargs generic en utilisant l'option -Xlint:varargs ou les annotations @SafeVarags ou @SuppressWarnings({"unchecked", "varargs"})

NIO 2

Cette nouvelle version de l'API NIO apporte de nombreuses fonctionnalités :

- l'API FileSystem : l'accès et la manipulation du système de fichiers, le support des métadonnées, les liens symboliques, le parcours des répertoires, les notifications des changements dans un répertoire, ...
- Des mises à jour des API existantes
- Les canaux asynchrones (Asynchronous I/O)
- . . .

Invoke dynamic

Développé dans le projet Da Vinci, Invoke dynamic a pour but de faciliter l'exécution de code issu de langages dynamiques dans la JVM : pour cela un nouvel opérateur (invokedynamic) a été ajouté dans le bytecode et une nouvelle API permet d'utiliser le chargement d'une classe avec un AnonymousClassLoader et une nouvelle manière d'invoquer dynamiquement une méthode,

Fork / Join

Ce framework facilite la parallélisation de tâches en exploitant les capacités multi-processeurs des machines. Le principe est de diviser les traitements en tâches (fork), exécutées en parallèle et éventuellement de nouveau les diviser, puis d'agréger les résultats (join).

1.4.11.2. Java 7 update

Oracle a proposé plusieurs mises à jour de la plate-forme Java SE 7. Ces mises à jour concernent :

- des corrections de bugs et ou de sécurité
- des évolutions ou des ajouts dans les outils du JDK et du JRE notamment la machine virtuelle HotSpot

Oracle applique à Java sa politique Critical Patch Updates (CPU) comme mécanisme primaire de diffusion des patches de sécurité : les dates de diffusion de ces CPU sont prédéfinies à une fréquence de quatre par an qui correspondent au mardi le plus proches du 17 des mois de janvier, avril, juillet et octobre. Des patches de sécurités peuvent être diffusés en dehors de ces dates notamment s'ils concernent des Security Alerts.

Java 7 update 1 CPU

Cette mise à jour contient une vingtaine de patchs de sécurité et quelques corrections de bugs. Il contient notamment un correctif dans le compilateur JIT de HotSpot qui provoque une erreur de l'optimisation de certaines boucles.

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/7u1-relnotes.html

La liste des patches de sécurité est consultable à l'url :
https://www.oracle.com/security-alerts/javacpuoct2011.html

La version externe est Java 7u1, le numéro de build est 1.7.0_1-b08. Elle est diffusée le 18 octobre 2011.

Cette version inclut la version 1.7R3 du moteur JavaScript Rhino de la fondation Mozilla.

Java 7 update 2

Cette mise à jour contient des corrections de bugs.

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/7u2-relnotes.html

La version externe est Java 7u2, le numéro de build est 1.7.0_2-b13. Elle est diffusée le 12 décembre 2011.

Cette version inclut :

- La version 22 de la machine virtuelle HotSpot
- Le support de Solaris 11 et Firefox 5, 6, 7 et 8
- Le SDK de JavaFX est inclus dans le JDK
- Des mises à jour dans les processus de déploiement d'applications par le réseau

Les démos et les exemples ne sont plus fournis avec le JDK mais sont fournis séparément.

Java 7 update 3 CPU

Cette mise à jour contient des patchs de sécurité. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/7u3-relnotes.html

La version externe est Java 7u3, le numéro de build est 1.7.0_3-b4. Elle est diffusée le 14 février 2012.

Java 7 update 4

Cette mise à jour contient des corrections de bugs.

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/7u4-relnotes.html

La version externe est Java 7u4, le numéro de build est 1.7.0_4-b20. Elle est diffusée le 26 avril 2012.

Cette version inclut :

- Le support du JDK pour Mac OS X
- La version 23 de la JVM HotSpot
- Le support du ramasse-miettes G1 (Garbage First)
- La version 1.4.6 de JAXP qui corrige quelques bugs et améliore les performances
- La version 10.8.2.2 de Java DB corrige quelques bugs

Java 7u4 propose une version 64 bits du JDK pour Mac OS X version Lion ou ultérieure.

La version 23 de la JVM HotSpot contient une partie des travaux de convergence avec la JVM JRockit.

Le ramasse-miettes G1 (Garbage First) est supporté : il est particulièrement adapté pour les JVM utilisant un gros heap.

Java 7 update 5 CPU

Cette mise à jour contient des corrections de bugs et des patches de sécurité.

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/7u5-relnotes.html

La version externe est Java 7u5, le numéro de build est 1.7.0_5-b06. Elle est diffusée le 12 juin 2012.

Java 7 update 6

Cette mise à jour contient des corrections de bugs.

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/7u6-relnotes.html

La version externe est Java 7u6, le numéro de build est 1.7.0_6-b24. Elle est diffusée le 14 aout 2012.

Cette version inclut :

- Java FX est installé en même temps que Java SE, dans le même package et plus dans un package d'installation séparé
- Le JDK et le JRE sont disponibles sur Mac OS X 10.7.3 (Lion) et supérieur. Le JDK est disponible pour Linux sur processeurs ARM v6 et v7
- La technologie Java Access Bridge est incluse dans cette version mais désactivée par défaut
- Une nouvelle fonction de hachage pour les clés de type String des collections de type map permet d'améliorer les performances. Par défaut, cette nouvelle fonction est désactivée car elle peut avoir des effets de bord sur l'ordre des itérations sur les clés ou les valeurs : pour l'activer, il faut affecter une valeur différente de -1 à la propriété système jdk.map.althashing.threshold. Cette valeur correspond à la taille de la collection à partir de laquelle la nouvelle fonction de hachage est utilisée. La valeur recommandée est 512

Java 7 update 7

Cette mise à jour contient des patches de sécurité critiques (Oracle Security Alert CVE-2012-4681).

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/7u7-relnotes.html

La version externe est Java 7u7, le numéro de build est 1.7.0_7-b10 (1.7.0_7-b11 pour Windows). Elle est diffusée le 30 aout 2012.

Java 7 update 9 CPU

Cette mise à jour contient des corrections de bugs et des patches de sécurité.

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/7u9-relnotes.html

La version externe est Java 7u9, le numéro de build est 1.7.0_9-b05. Elle est diffusée le 16 octobre 2012.

Java 7 update 10

Cette mise à jour contient des corrections de bugs et des améliorations de sécurité.

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/7u10-relnotes.html

La version externe est Java 7u10, le numéro de build est 1.7.0_10-b18. Elle est diffusée le 11 décembre 2012.

Cette version ajoute MAC OS X 10.8, Windows Server 2012 (64 bits) et Windows 8 Desktop à la liste des plate-formes certifiées.

Java 7 update 11 CPU

Cette mise à jour contient des patchs de sécurité critiques (Oracle Security Alert CVE-2013-0422).

La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/7u11-relnotes.html

La version externe est Java 7u11, le numéro de build est 1.7.0_11-b21. Elle est diffusée le 13 janvier 2013.

Java 7 update 13 CPU

Cette mise à jour contient des patches de sécurité. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/7u13-relnotes.html

La version externe est Java 7u13, le numéro de build est 1.7.0_43-b20. Elle est diffusée le 1er février 2013.

Java 7 update 15 CPU

Cette mise à jour contient des patches de sécurité. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/7u15-relnotes.html

Attention : comme Java 6 a atteint sa fin de vie et qu'Oracle ne fournira plus d'update publique gratuit pour Java 6, cette mise à jour désinstalle automatiquement Java 6.

La version externe est Java 7u15, le numéro de build est 1.7.0_15-b03. Elle est diffusée le 19 février 2013.

Java 7 update 17

Cette mise à jour contient des patches de sécurité critiques (Oracle Security Alert CVE-2013-1493). La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/7u17-relnotes.html

La version externe est Java 7u17, le numéro de build est 1.7.0_17-b02. Elle est diffusée le 4 mars 2013.

Java 7 update 21 CPU

Cette mise à jour contient des patches de sécurité. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/7u21-relnotes.html

La configuration de la sécurité du panneau de control Java a été modifiée : les niveaux low et custom ont été retirés.

Il est recommandé de signer toutes les applications : les applications non signées ou self-signed pourraient ne plus être supportées dans les prochaines mises à jour du JDK.

Une boîte de dialogue fournissant des informations de sécurités est affichée à l'utilisateur avant l'exécution d'une application qui utilise des composants signés et non signés.

Cette version du JDK est disponible pour Linux sur microprocesseur ARM (v6 et v7) avec certaines fonctionnalités qui ne sont pas supportées : Java WebStart, Java Plug-In, le ramasse-miettes Garbage First (G1), JavaFX SDK et Runtime. Le support de Java sur ARM ne concerne que GNOME Desktop Environment version 1:2.30+7.

La valeur par défaut de la propriété java.rmi.server.useCodebaseOnly depuis cette version est à true.

Sous Windows, la manière d'analyser la commande fournie en paramètre à la méthode exec() de la classe Runtime a été modifiée notamment si le chemin contient un espace.

La version externe est Java 7u21, le numéro de build est 1.7.0_21-b11 (1.7.0_21-b12 pour Mac OS X). Elle est diffusée le 16 avril 2013.

Java 7 update 25 CPU

Cette mise à jour contient des patches de sécurité. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/7u25-relnotes.html

La version externe est Java 7u25, le numéro de build est 1.7.0_25-b15 (1.7.0_25-b17 pour Windows). Elle est diffusée le 18 juin 2013.

Java 7 update 40

Cette mise à jour contient des améliorations relatives à la sécurité, le support de l'affichage Retina sur Mac OS X et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/7u40-relnotes.html

La version externe est Java 7u40, le numéro de build est 1.7.0_40-b43. Elle est diffusée le 10 septembre 2013.

L'outil Java Mission Control (JMC) est fourni avec le JDK.

La version 2.2.40 de JavaFX est incluse avec cette version.

Java 7 update 45 CPU

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de 51 bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/7u45-relnotes.html

La version externe est Java 7u45, le numéro de build est 1.7.0_45-b18. Elle est diffusée le 15 octobre 2013.

La version 2.2.51 de JavaFX est incluse avec cette version.

Cette version est disponible pour Linux ARM.

Java 7 update 51

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de 36 bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/7u51-relnotes.html

La version externe est Java 7u51, le numéro de build est 1.7.0_51-b13. Elle est diffusée le 11 janvier 2014.

La version 2.2.45 de JavaFX est incluse avec cette version.

Java 7 update 55 CPU

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de 37 bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/7u55-relnotes.html

La version externe est Java 7u55, le numéro de build est 1.7.0_55-b13. Elle est diffusée le 15 avril 2014.

Il est possible de désactiver l'installation d'outils tiers sponsorisés lors de l'installation de Java 32bits sous Windows en utilisant l'option SPONSORS=0 dans la ligne de commande.

Exemple :

```
c:\>jre-7u55-windows-i586-iftw.exe SPONSORS=0
```

Cette désactivation sera maintenue lors des mises à jour suivantes.

La version 2.2.55 de JavaFX est incluse avec cette version.

Java 7 update 60

Cette mise à jour contient des améliorations et des corrections de 130 bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/7u60-relnotes.html

La version externe est Java 7u60, le numéro de build est 1.7.0_60-b19. Elle est diffusée le 28 mai 2014.

La version 2.2.60 de JavaFX et 5.3 de Java Mission Control sont incluses avec cette version.

Java 7 update 65

Cette mise à jour contient des améliorations et des corrections de 18 bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/7u65-relnotes.html

La version externe est Java 7u65, le numéro de build est 1.7.0_65-b20. Elle est diffusée le 15 juillet 2014.

La version 2.2.65 de JavaFX est incluse avec cette version.

Il est possible de désactiver l'installation d'outils tiers sponsorisés lors de l'installation via le panneau de configuration Java (Java Control Panel).

Java 7 update 67

Cette mise à jour contient des améliorations et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/7u67-relnotes.html

La version externe est Java 7u67, le numéro de build est 1.7.0_67-b01. Elle est diffusée le 4 août 2014.

La version 2.2.67 de JavaFX est incluse avec cette version.

Java 7 update 71 CPU

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de 16 bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/7u71-relnotes.html

La version externe est Java 7u71, le numéro de build est 1.7.0_71-b14. Elle est diffusée le 14 octobre 2014.

La version 2.2.71 de JavaFX est incluse avec cette version.

Oracle recommande de désactiver le support du protocole SSL V3 à cause de la faille SSL V3.0 "Poodle" Vulnerability - CVE-2014-3566.

Java 7 update 72

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de 36 bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/7u72-relnotes.html

La version externe est Java 7u71, le numéro de build est 1.7.0_72-b14. Elle est diffusée le 14 octobre 2014.

La version 2.2.72 de JavaFX est incluse avec cette version.

Java 7 update 75 CPU

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de 12 bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/7u75-relnotes.html

La version externe est Java 7u75, le numéro de build est 1.7.0_75-b13. Elle est diffusée le 19 janvier 2015.

La version 2.2.75 de JavaFX est incluse avec cette version.

Le protocole SSL V3 est désactivé par défaut et il est supprimé du panneau de contrôle.

Java 7 update 76 CPU

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de 96 bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/7u76-relnotes.html

La version externe est Java 7u76, le numéro de build est 1.7.0_76-b13. Elle est diffusée le 19 janvier 2015.

La version 2.2.76 de JavaFX est incluse avec cette version.

Java 7 update 79 CPU

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de 21 bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/7u79-relnotes.html

La version externe est Java 7u79, le numéro de build est 1.7.0_79-b15. Elle est diffusée le 14 avril 2015.

La version 2.2.79 de JavaFX est incluse avec cette version.

Java 7 update 80 CPU

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/7u80-relnotes.html

La version externe est Java 7u80, le numéro de build est 1.7.0_80-b15. Elle est diffusée le 14 avril 2015.

La version 2.2.80 de JavaFX est incluse avec cette version.

Les mises à jour suivantes ne sont pas diffusées publiquement mais uniquement grâce au programme de support payant de Java SE.

1.4.12. Java SE 8 (nom de code Spider)

Java SE 8 est issu des travaux d'Oracle, du JCP et des travaux d'implémentation du projet open source OpenJDK.

Les spécifications de Java SE 8 sont définit dans la JSR 337.

Java SE 8 a été diffusé le 18 mars 2014.

Les travaux de développement du JDK sur les fonctionnalités de l'implémentation de référence sont organisés en JEP (Java Enhancement Proposals).

Une cinquantaine de JEP ont été incluses dans Java 8 notamment :

- JEP 101: Generalized Target-Type Inference
- JEP 103 : Parallel array sorting. Cette fonctionnalité ajoute des méthodes à la classe java.util.Arrays pour utiliser le framework Fork/Join pour permettre de trier un tableau en parallèle
- JEP 104 : annotations on Java Type.
- JEP 106 : add Javadoc to javax.tools. Cette fonctionnalité permet d'utiliser javadoc via l'API javax.tools
- JEP 107 : Bulk data operations for collections. Cette fonctionnalité ajoute des opérations de masse sur des données à l'API Collections (filter/map/reduce). Ces opérations peuvent être mono ou multithreads
- JEP 109 : enhance code libraries with lambda. Cette fonctionnalité utilise les lambdas dans certaines API de la bibliothèque core de Java (JSR 335)
- JEP 114 : TLS server name indication (SNI) extension
- JEP 118 : Language Access to Parameter Names at Runtime. Cette fonctionnalité fournit un mécanisme pour retrouver le nom des paramètres d'une méthode ou d'un constructeur à l'exécution en utilisant l'introspection
- JEP 120 : Repeating annotations
- JEP 122: Remove the permanent generation
- JEP 123 : configurable secure random-number generation
- JEP 124 : enhance the certificate revocation checking API
- JEP 126 : Lambda Expression & Virtual Extension Methods
- JEP 128 : BCP 47 locale matching
- JEP 129 : NSA Suite B cryptographic algorithms
- JEP 133 : Unicode 6.2. Cette fonctionnalité propose le support d'Unicode 6.2 dans la plate-forme
- JEP 135 : Base 64 encoding & decoding. Cette fonctionnalité permet de définir une API standard pour l'encodage et le décodage en base 64
- JEP 140 : Limited doPrivileged
- JEP 150 : date & time API. Cette fonctionnalité implémente la nouvelle API de gestion des dates/heures (JSR 310)
- JEP 153: Launch Java FX applications (direct launching of JavaFX application JARs)
- JEP 155 : concurrency updates
- JEP 161 : Platform compact profiles (JSR 3 MR et 160 MR). Cette fonctionnalité permet de définir des profiles de la plate-forme Java SE qui en sont un sous-ensemble de l'API Core
- JEP 162 : prepare for modularization (JSR 173 MR et 206 MR)
- JEP 170 : JDBC 4.2 (JSR 114 MR et 221 MR)
- JEP 178: Statically-linked JNI libraries
- JEP 184 : HTTP URL permissions. Cette fonctionnalité ajoute une permission pour utiliser une URL plutôt qu'une adresse IP
- JEP 185 : restrict fetching for external XML resources (JSR 206 MR)

Les principales API ajoutées sont :

- JSR 308 (JEP104) : Annotations on Java Types
- JSR 310 (JEP 150) : Date and Time API
- JSR 335 (JEP126) : le projet Lambda dont le but est d'intégrer les expressions lambda, appelées closure dans d'autres langages
- JSR 223 (JEP 174) : le projet Nashorn dont le but est de proposer un moteur d'exécution Javascript. La commande jjs permet d'invoquer ce moteur en ligne de commande.

Les spécifications de certaines API ont été modifiées dans une Maintenance Release dédiée :

- JSR 003: Java Management Extensions
- JSR 114: JDBC Rowsets

- JSR 160: JMX Remote API
- JSR 173: Streaming API for XML
- JSR 199: Java Compiler API (Add Javadoc to javax.tools)
- JSR 206: Java API for XML Processing :
- JSR 221: JDBC 4.0
- JSR 269: Pluggable Annotation-Processing API pour ajouter les repeating annotations

1.4.12.1. Les expressions Lambdas et les Streams

Les expressions Lambda permettent de mettre en oeuvre une forme de programmation fonctionnelle. Une expression Lambda permet de passer les traitements contenus dans l'expression en paramètre d'une méthode.

L'API Stream, contenue dans le package java.util.stream, permet de mettre en oeuvre des opérations fonctionnelles sur un flux d'éléments. Les éléments traités par un Stream ont pour origine une source. Une source contient (par exemple une collection ou un tableau) ou génère (par exemple un générateur de nombres aléatoires ou de valeurs incrémentées) les éléments à traiter.

Les traitements sont composés d'une aggrégation d'opérations. Les opérations proposées par l'API permettent de réaliser différentes actions sur les données à traiter. Généralement ces opérations attendent en paramètre une interface fonctionnelle qu'il est possible d'exprimer avec une expression Lambda ou une référence de méthode.

L'API Stream propose plusieurs opérations qui sont regroupées sous la forme d'un pipeline :

- composé de zéro, une ou plusieurs opérations intermédiaires
- une seule opération terminale

Dans ce pipeline, les éléments en sortie d'une opération sont utilisés en entrée de l'opération suivante.

Les opérations intermédiaires sont toutes lazy : elles n'exécutent aucun traitement tant que l'opération terminale n'est pas invoquée. Une opération intermédiaire renvoie toujours un nouveau Stream qui contient les éléments à traiter par l'opération suivante. Elle ne doit pas nécessairement traiter tous ses éléments d'entrée avant de produire un ou des éléments de sortie consommable par l'opération suivante.

Une opération intermédiaire peut être :

- stateless : chaque élément peut être traité de manière indépendante les uns des autres
- stateful : les traitements peuvent avoir besoin de conserver tout ou partie des éléments précédents pour traiter l'élément courant (par pour un tri ou l'élimination des doublons)

Les opérations terminales permettent de produire un résultat ou éventuellement des effets de bord à partir des éléments qui lui sont fournis. Un effet de bords résulte de l'exécution d'un bloc de code. Généralement les effets de bords ne sont pas recommandés car ils peuvent engendrer des problèmes hazardeux notamment lors de l'utilisation en concurrence. Il est cependant possible que ces effets de bord consistent simplement à afficher des messages dans la console ou les logs pour déboguer.

Un Stream peut facilement exécuter ses traitement de manière séquentielle ou parallèle. En parallèle, la gestion des éventuels accès concurrents sont cependant à la charge du développeur.

1.4.12.2. Les autres évolutions dans les API et le langage

Plusieurs améliorations ont été apportées au langage :

- Il est possible de définir des méthodes statiques dans une interface
- es méthodes par défaut (default method) permettent de définir des méthodes, ainsi que leurs traitements dans une interface. Ceci permet à une classe qui implémente la méthode sans la redéfinir d'avoir le comportement de la méthode définie dans l'interface. Elles permettent donc d'ajouter de nouvelles fonctionnalités aux interfaces en maintenant la compatibilité binaire avec les classes plus anciennes qui implémentent ces interfaces. Cela permet d'avoir une forme limitée d'héritage multiple dans les interfaces

- Les Repeating Annotations permettent d'utiliser plusieurs fois la même annotation sur une même entité
- Les Type Annotations permettent d'utiliser une annotation partout où un type est utilisé : ceci permet à des outils de faire plus de vérifications sur les types dans le code
- L'inférence de type a été améliorée

Plusieurs améliorations ont été apportées à certaines API :

- Support d'Unicode 6.2
- Encodage et décodage en Base64 fournis en standard
- Ajout de la classe java.net.URLPermission
- l'API Reflection permet d'obtenir le nom des paramètres d'une méthode ou d'un constructeur

La version 4.2 de JDBC propose quelques fonctionnalités nouvelles et la suppression du pont JDBC-ODBC.

Plusieurs amélioration relatives à la sécurité ont été apportées notamment :

- TLS version 1.2 est utilisé par défaut côté client
- implémentation du mode AEAD pour algorithmes de chiffrement
- support de la fonction de hachage SHA-224
- support des algorithmes PBEWithSHA256AndAES_128 et PBEWithSHA512AndAES_256, des paires de clés DSA 2048 bits et des algorithmes de signature SHA224withDSA et SHA256withDSA par l'implémentation SunJCE

Plusieurs fonctionnalités ont été ajoutées pour faciliter la gestion de la concurrence :

- la classe ConcurrentHashMap a été intégralement réécrite et une trentaine de méthodes ont été ajoutées pour permettre le support de l'API Stream et les expressions Lambda
- plusieurs classes (CompletableFuture<T>, ConcurrentHashMap.KeySetView<K,V>, CountedCompleter<T> et CompletionException) et interfaces (CompletableFuture.AsynchronousCompletionTask et CompletionStage<T>) ont été ajoutées dans le package java.util.concurrent
- plusieurs classes ont été ajoutées dans le package java.util.concurrent.atomic : DoubleAccumulator, DoubleAdder, LongAccumulator et LongAdder
- la classe java.util.concurrent.locks.StampedLock permet de poser des verrous selon trois modes pour contrôler les accès en lecture/écriture
- la classe java.util.concurrent.ForkJoinPool propose un support pour un pool commun

1.4.12.3. Les évolutions dans la plate-forme

Plusieurs évolutions ont été apportées à la machine virtuelle HotSpot :

- suppression de la Permanent Generation (PermGen)
- support dans le bytecode de l'invocation des méthodes par défaut
- support des fonctionnalités relatives à AES sur le matériel Intel en utilisant les options UseAES et UseAESIntrinsics

Le JDK 8 inclut la version 5.3 de Java Mission Control et la version 10.10 de Java DB.

Plusieurs outils ont été ajoutés ou modifiés :

- jjs pour invoquer le moteur javascript Nashorn
- l'outil java peut exécuter des applications JavaFX
- jdeps
- l'option -parameters de l'outil javac permet de demander au compilateur de stocker les informations permettant de retrouver les paramètres par leur nom en utilisant l'API Reflection.

1.4.12.4. Les profiles de Java SE

Initialement, Java 8 devait proposer un système de gestion des modules qui permettant de modulariser la plate-forme Java elle-même. Cependant cette fonctionnalité a été reportée à la prochaine version de Java. Pour permettre la future modularisation de la plate-forme, certaines méthodes ont été déclarées deprecated à cause de leurs dépendances. Elles devraient être supprimée dans la prochaine version de Java.

En attendant, pour permettre de déployer Java sur certaines plate-formes, Java 8 introduit le concept de Java SE Profile qui est un sous-ensemble des API de la plate-forme Java SE.

Le but des profiles est de permettre de préparer la migration des applications écrites en utilisant le CDC de Java ME vers un profile de Java SE. La puissance des appareils mobiles augmentant, permet de préparer la convergence du CDC vers Java SE.

Un profile respecte plusieurs contraintes :

- un profile ne concerne pas la JVM ni les outils
- un profile ne concerne pas le langage tel que définit dans la JLS
- un profile est défini comme une liste de packages dont toutes les classes et interfaces doivent être celles incluses dans le package du même nom dans le Java SE (quelques exceptions existent pour des raisons de dépendances)
- un profile peut être basé sur un autre profile : dans ce cas, le profile doit contenir tous les packages du profile de base
- le plus petit profile doit obligatoirement contenir les packages contenant les classes et interfaces utilisées dans la JLS

L'intégralité des API de la plate-forme Java SE n'est pas considérée comme un profile.

Les Compact Profiles définissent des sous-ensembles de la plate-forme Java qui permettent d'exécuter sur des appareils possédant des ressources limitées des applications ne requérant pas toutes les fonctionnalités.

Trois profiles sont définis par les spécifications pour la plate-forme Java SE :

- compact1 : contient un ensemble minimum de packages plus quelques packages additionnels (java.util.logging, javax.net.ssl et javax.script). Hormis certains packages spécifiques à Java ME, son but est de remplacer le CDC 1.1 et Foundation Profile 1.1
- compact2 : contient des packages relatifs à RMI, JDBC et XML
- compact3 : doit permettre de développer des applications sans interfaces graphiques (AWT, Swing), services web (JAX-WS, SAAJ, JAXB) ni CORBA.

Ils ne concernent que les API utilisables mais ne concernent pas la JVM ou les outils.

Chaque profil contient les API des profils de numéro inférieur : par exemple, compact2 est un sur-ensemble de compact1 et l'API complète de Java SE est un sur-ensemble du profil de compact3. Le tableau suivant présente la composition des API de chaque ensemble :

Toutes les API	Beans	JNI	JAX-WS
	Preferences	Accessibility	IDL
	RMI-IIOP	CORBA	Print Service
	Sound	Swing	Java 2D
	AWT	Drag and Drop	Input Methods
	Image I/O		
compact3	Security	JMX	
	XML JAXP	Management	Instrumentation
compact2	JDBC	RMI	XML JAXP
compact1	Core (java.lang.*)	Security	Serialization
	Networking	Ref Objects	Regular Expressions

	Date and Time	Input/Output	Collections
	Logging	Concurrency	Reflection
	JAR	ZIP	Versioning
	Internationalization	JNDI	Override Mechanism
	Extension Mechanism	Scripting	

Le tableau ci-dessous précise les packages contenus dans chaque profil.

compact1	**compact2**	**compact3**
java.io	java.rmi	java.lang.instrument
java.lang	java.rmi.activation	java.lang.management
java.lang.annotation	java.rmi.dgc	java.security.acl
java.lang.invoke	java.rmi.registry	java.util.prefs
java.lang.ref	java.rmi.server	javax.annotation.processing
java.lang.reflect	java.sql	javax.lang.model
java.math	javax.rmi.ssl	javax.lang.model.element
java.net	javax.sql	javax.lang.model.type
java.nio	javax.transaction	javax.lang.model.util
java.nio.channels	javax.transaction.xa	javax.management
java.nio.channels.spi	javax.xml	javax.management.loading
java.nio.charset	javax.xml.datatype	javax.management.modelmbean
java.nio.charset.spi	javax.xml.namespace	javax.management.monitor
java.nio.file	javax.xml.parsers	javax.management.openmbean
java.nio.file.attribute	javax.xml.stream	javax.management.relation
java.nio.file.spi	javax.xml.stream.events	javax.management.remote
java.security	javax.xml.stream.util	javax.management.remote.rmi
java.security.cert	javax.xml.transform	javax.management.timer
java.security.interfaces	javax.xml.transform.dom	javax.naming
java.security.spec	javax.xml.transform.sax	javax.naming.directory
java.text	javax.xml.transform.stax	javax.naming.event
java.text.spi	javax.xml.transform.stream	javax.naming.ldap
java.time	javax.xml.validation	javax.naming.spi
java.time.chrono	javax.xml.xpath	javax.security.auth.kerberos
java.time.format	org.w3c.dom	javax.security.sasl
java.time.temporal	org.w3c.dom.bootstrap	javax.sql.rowset
java.time.zone	org.w3c.dom.events	javax.sql.rowset.serial
java.util	org.w3c.dom.ls	javax.sql.rowset.spi
java.util.concurrent	org.xml.sax	javax.tools
java.util.concurrent.atomic	org.xml.sax.ext	javax.xml.crypto

java.util.concurrent.locks	org.xml.sax.helpers	javax.xml.crypto.dom
java.util.function		javax.xml.crypto.dsig
java.util.jar		javax.xml.crypto.dsig.dom
java.util.logging		javax.xml.crypto.dsig.keyinfo
java.util.regex		javax.xml.crypto.dsig.spec
java.util.spi		org.ietf.jgss
java.util.stream		
java.util.zip		
javax.crypto		
javax.crypto.interfaces		
javax.crypto.spec		
javax.net		
javax.net.ssl		
javax.script		
javax.security.auth		
javax.security.auth.callback		
javax.security.auth.login		
javax.security.auth.spi		
javax.security.auth.x500		
javax.security.cert		

La documentation Javadoc précise à quel profile une classe ou une interface appartient.

Une implémentation des spécifications de Java n'a pas l'obligation d'implémenter ses trois profiles.

L'option -profile du compilateur javac permet de demander la compilation pour le profile précisé en paramètre.

1.4.12.5. Java 8 update

Java 8 update 5 CPU

Cette mise à jour contient des améliorations relatives à la sécurité. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u5-relnotes.html

La version externe est Java 8u5, le numéro de build est 1.8.0_5-b13. Elle est diffusée le 11 janvier 2014.

Java 8 update 11 CPU

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de 18 bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u11-relnotes.html

La version externe est Java 8u11, le numéro de build est 1.8.0_11-b12.

L'outil jdeps est ajouté dans le JDK.

Il est possible de désactiver l'installation d'outils tiers sponsorisés lors de l'installation via le panneau de configuration Java (Java Control Panel).

Java 8 update 20 CPU

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u20-relnotes.html

La version externe est Java 8u20, le numéro de build est 1.8.0_20-b62.

La version 5.4 de Java Mission Control sont incluses avec cette version.

Java 8 update 25 CPU

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u25-relnotes.html

La version externe est Java 8u25, le numéro de build est 1.8.0_25-b17 (1.8.0_25-b18 sous Windows).

Oracle recommande de désactiver le support du protocole SSL V3.

Java 8 update 31 CPU

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de 26 bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u31-relnotes.html

La version externe est Java 8u31, le numéro de build est 1.8.0_31-b13.

Le protocole SSL V3 est désactivé par défaut et retiré du panneau de configuration.

Java 8 update 40

Cette mise à jour contient des améliorations et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u40-relnotes.html

La version externe est Java 8u40, le numéro de build est 1.8.0_40-b26.

Les mécanismes endorsed et extension sont deprecated et pourront être retiré dans les versions futures. L'option XX:+CheckEndorsedAndExtDirs permet de vérifier si la JVM est configurée pour utiliser un de ces mécanismes.

Des optimisations ont été apportées au projet Nashorn et certains paramètres par défaut du ramasse-miettes G1 ont été modifiés pour limiter les full GC.

Java 8 update 45 CPU

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u45-relnotes.html

La version externe est Java 8u45, le numéro de build est 1.8.0_45-b15.

Java 8 update 51 CPU

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u51-relnotes.html

La version externe est Java 8u51, le numéro de build est 1.8.0_51-b16.

Java 8 update 60

Cette mise à jour contient des améliorations et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u60-relnotes.html

La version externe est Java 8u60, le numéro de build est 1.8.0_60-b27.

Plusieurs améliorations ont été apportées au moteur Javascript Nashorn et à la version 1.2 du Deployment Rule Set.

Le keystore de type JKS peut accéder à des fichiers au format PKCS12.

Les algorithmes de chiffrement de TLS reposant sur RC4 sont désactivés par défaut car ils sont considérés comme non sûre.

Les méthodes monitorEnter(), monitorExit() et tryMonitorEnter() de la classe sun.misc.Unsafe sont deprecated.

Java 8 update 65

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u65-relnotes.html

La version externe est Java 8u65, le numéro de build est 1.8.0_65-b17.

Java 8 update 66

Cette mise à jour contient des améliorations et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u66-relnotes.html

La version externe est Java 8u66, le numéro de build est 1.8.0_66-b17.

Java 8 update 71

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u71-relnotes.html

La version externe est Java 8u71, le numéro de build est 1.8.0_71-b15.

Java 8 update 73

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u73-relnotes.html

La version externe est Java 8u73, le numéro de build est 1.8.0_73-b02.

Java 8 update 77

Cette mise à jour contient des améliorations relatives à la sécurité. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u77-relnotes.html

La version externe est Java 8u77, le numéro de build est 1.8.0_77-b03.

Java 8 update 91

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u91-relnotes.html

La version externe est Java 8u91, le numéro de build est 1.8.0_91-b14 pour Windows et 1.8.0_91-b15 pour les autres systèmes d'exploitation.

De nouveaux certificats racine sont ajoutés.

Java 8 update 101 (juillet 2016)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u101-relnotes.html

La version externe est Java 8u101, le numéro de build est 1.8.0_101-b13.

De nouveaux certificats racine sont ajoutés.

Java 8 update 111 (octobre 2016)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u111-relnotes.html

La version externe est Java 8u111, le numéro de build est 1.8.0_111-b14.

Java 8 update 121 (janvier 2017)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u121-relnotes.html

La version externe est Java 8u121, le numéro de build est 1.8.0_121-b13.

Java 8 update 131 (avril 2017)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u131-relnotes.html

La version externe est Java 8u131, le numéro de build est 1.8.0_131-b11.

Java 8 update 141 (juillet 2017)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u141-relnotes.html

La version externe est Java 8u141, le numéro de build est 1.8.0_141-b15.

Java 8 update 144 (juillet 2017)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u144-relnotes.html

La version externe est Java 8u144, le numéro de build est 1.8.0_144-b01.

Java 8 update 151 (octobre 2017)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u151-relnotes.html

La version externe est Java 8u151, le numéro de build est 1.8.0_151-b12.

Java 8 update 161 (janvier 2018)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u161-relnotes.html

La version externe est Java 8u161, le numéro de build est 1.8.0_161-b12.

Java 8 update 171 (avril 2018)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u171-relnotes.html

La version externe est Java 8u171, le numéro de build est 1.8.0_171-b11.

Java 8 update 181 (juillet 2018)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u181-relnotes.html

La version externe est Java 8u181, le numéro de build est 1.8.0_181-b13.

La base de données Java DB n'est plus incluse dans le JDK : elle peut être téléchargée indépendamment sur le site d'Apache Derby.

Java 8 update 191 (octobre 2018)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u191-relnotes.html

La version externe est Java 8u191, le numéro de build est 1.8.0_191-b12.

La version de gcc utilisée pour compiler les versions Linux x86/x64 du JDK est 7.3 en remplacement de la version 4.3.

Les algorithmes TLS reposant sur DES sont désactivés et plusieurs certificats racines qui ne sont plus utilisés sont retirés.

Le support de Docker a été amélioré grâce au report de 3 fonctionnalités de Java 10 :

- JDK-8146115 : une nouvelle option (UseContainerSupport) est utilisable sur Linux uniquement et est activée par défaut. Elle permet d'améliorer la détection de l'éxécution de la JVM dans un conteneur Docker et l'obtention d'informations à partir du conteneur (nombre de CPU et quantité de mémoire) et non plus de l'hôte. C'est d'autant plus important que ces ressources peuvent être limitée au démarrage du conteneur et qu'elles sont utilisées par défaut pour configurer certaines caractéristques de la JVM (taille maximum du heap, nombres de threads pour certains pools (GC et Fork/Join), ...). Une nouvelle option (-XX:ActiveProcessorCount=count) permet de préciser le nombre de CPU utilisable par la JVM dans le conteneur

- JDK-8186248 : trois nouvelles options (-XX:InitialRAMPercentage, -XX:MaxRAMPercentage et -XX:MinRAMPercentage) permettent un contrôle plus fin de la taille du heap en l'exprimant en pourcentage de taille de la mémoire assignée au conteneur. Elles remplacent les trois anciennes options (-XX:InitialRAMFraction, -XX:MaxRAMFraction et -XX:MinRAMFraction) qui permettent d'exprimer la quantité de mémoire sous la forme d'une fraction de la mémoire du conteneur
- JDK-8179498 : corrige bug dans le mécanisme de liaison d'un processus hôte à un processus Java exécuté dans un conteneur Docker

Java 8 update 201 (janvier 2019)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u201-relnotes.html

La version externe est Java 8u201, le numéro de build est 1.8.0_201-b09.

Java 8 update 211 (avril 2019)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u211-relnotes.html

Cette version propose un support pour la nouvelle ère du calendrier japonais nommé Reiwa.

La version externe est Java 8u211, le numéro de build est 1.8.0_211-b12.

Java 8 update 221 (juillet 2019)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u221-relnotes.html

La JVM détecte correctement la version Server 2019 de Windows.

La version externe est Java 8u221, le numéro de build est 1.8.0_221-b11.

Java 8 update 231 (octobre 2019)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u231-relnotes.html

4 nouveaux événements de JFR relatifs à la sécurité ont été ajoutés.

La version externe est Java 8u231, le numéro de build est 1.8.0_231-b11.

Java 8 update 241 (janvier 2020)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u241-relnotes.html

La version externe est Java 8u241, le numéro de build est 1.8.0_241-b07.

Java 8 update 251 (avril 2020)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u251-relnotes.html

La version externe est Java 8u251, le numéro de build est 1.8.0_251-b08.

Java 8 update 261 (juillet 2020)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u261-relnotes.html

Le build sous Windows utilise Visual Studio 2017 ce qui change les dll utilisées par le JDK.

Un support de TLS 1.3 est inclus.

Java Mission Control n'est plus fourni avec la JDK.

Les méthodes getFreePhysicalMemorySize(), getTotalPhysicalMemorySize(), getFreeSwapSpaceSize(), getTotalSwapSpaceSize(), getSystemCpuLoad()de l'OperatingSystemMXBean retourne des informtions tenant compte des limitations appliquées lors de l'exécution de la JVM dans un conteneur.

La version externe est Java 8u261, le numéro de build est 1.8.0_261-b12.

Java 8 update 271 (octobre 2020)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u271-relnotes.html

Le plug-in Java utilisant le connecteur NPAPI a été rétiré sur les plate-formes Linux, Solaris et MacOS car tous les navigateurs ont retirés l'utilisation de ce connecteur.

La version externe est Java 8u271, le numéro de build est 1.8.0_271-b09.

Java 8 update 281 (janvier 2021)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u281-relnotes.html

La version externe est Java 8u281, le numéro de build est 1.8.0_281-b09.

Java 8 update 291 (avril 2021)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u291-relnotes.html

TLS 1.0 et 1.1 sont désactivés par défaut. Pour les réactiver, il faut supprimer "TLSv1" et/ou "TLSv1.1" de la propriété jdk.tls.disabledAlgorithms dans le fichier de configuration java.security.

La version externe est Java 8u291, le numéro de build est 1.8.0_291-b10.

Java 8 update 301 (juillet 2021)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u301-relnotes.html

Les algorithmes de chiffrement par défaut utilisés dans un keystore PKCS #12 ont été mis à jour pour utiliser des algorithmes plus forts basés sur AES-256 et SHA-256.

La version externe est Java 8u301, le numéro de build est 1.8.0_301-b09.

Java 8 update 311 (octobre 2021)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u311-relnotes.html

La version externe est Java 8u311, le numéro de build est 1.8.0_311-b11.

Java 8 update 321 (janvier 2022)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u321-relnotes.html

La version externe est Java 8u321, le numéro de build est 1.8.0_321-b07.

Java 8 update 331 (avril 2022)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u331-relnotes.html

La version externe est Java 8u331, le numéro de build est 1.8.0_331-b09.

Java 8 update 341 (juillet 2022)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u341-relnotes.html

La version externe est Java 8u341, le numéro de build est 1.8.0_341-b10.

TLSv1.3 est activé par défaut cîté client

Java 8 update 351 (octobre 2022)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u351-relnotes.html

La version externe est Java 8u351, le numéro de build est 1.8.0_351-b10.

Java 8 update 361 (janvier 2023)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u361-relnotes.html

La version externe est Java 8u361, le numéro de build est 1.8.0_361-b09.

L'outil VisualVM n'est plus fourni avec le JDK : il doit être téléchargé séparemment à l'url https://visualvm.github.io.

Java 8 update 371 (avril 2023)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u371-relnotes.html

La version externe est Java 8u371, le numéro de build est 1.8.0_371-b11.

Java 8 update 381 (juillet 2023)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u381-relnotes.html

La version externe est Java 8u381, le numéro de build est 1.8.0_381-b09.

Elle comprend plusieurs améliorations et corrections visant à améliorer le support des cgroup v1 et v2 dans les conteneurs.

Java 8 update 391 (octobre 2023)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/8u391-relnotes.html

La version externe est Java 8u391, le numéro de build est 1.8.0_391-b13.

Un nouvel événement JFR a été ajouté pour enregistrer les détails des invocations à java.security.Provider.getService(String type, String algorithm).

À partir de cette version, les valeurs "allow" et "disallow" de la propriété système java.security.manager sont ignorées.

1.4.13. Java SE 9

Java SE 9 est issu des travaux d'Oracle, du JCP et des travaux d'implémentation du projet open source OpenJDK.

Les spécifications de Java SE 9 sont définit dans la JSR 337.

De nombreuses JEP ont été incluses dans Java 9 :

Numéro	Titre	Numéro	Titre
102	Process API Updates	252	Use CLDR Locale Data by Default
110	HTTP 2 Client	253	Prepare JavaFX UI Controls & CSS APIs for Modularization
143	Improve Contended Locking	254	Compact Strings
158	Unified JVM Logging	255	Merge Selected Xerces 2.11.0 Updates into JAXP

165	Compiler Control	256	BeanInfo Annotations
193	Variable Handles	257	Update JavaFX/Media to Newer Version of GStreamer
197	Segmented Code Cache	258	HarfBuzz Font-Layout Engine
199	Smart Java Compilation, Phase Two	259	Stack-Walking API
200	The Modular JDK	260	Encapsulate Most Internal APIs
201	Modular Source Code	261	Module System
211	Elide Deprecation Warnings on Import Statements	262	TIFF Image I/O
212	Resolve Lint and Doclint Warnings	263	HiDPI Graphics on Windows and Linux
213	Milling Project Coin	264	Platform Logging API and Service
214	Remove GC Combinations Deprecated in JDK 8	265	Marlin Graphics Renderer
215	Tiered Attribution for javac	266	More Concurrency Updates
216	Process Import Statements Correctly	267	Unicode 8.0
217	Annotations Pipeline 2.0	268	XML Catalogs
219	Datagram Transport Layer Security (DTLS)	269	Convenience Factory Methods for Collections
220	Modular Run-Time Images	270	Reserved Stack Areas for Critical Sections
221	Simplified Doclet API	271	Unified GC Logging
222	jshell: The Java Shell (Read-Eval-Print Loop)	272	Platform-Specific Desktop Features
223	New Version-String Scheme	273	DRBG-Based SecureRandom Implementations
224	HTML5 Javadoc	274	Enhanced Method Handles
225	Javadoc Search	275	Modular Java Application Packaging
226	UTF-8 Property Files	276	Dynamic Linking of Language-Defined Object Models
227	Unicode 7.0	277	Enhanced Deprecation
228	Add More Diagnostic Commands	278	Additional Tests for Humongous Objects in G1
229	Create PKCS12 Keystores by Default	279	Improve Test-Failure Troubleshooting
231	Remove Launch-Time JRE Version Selection	280	Indify String Concatenation
232	Improve Secure Application Performance	281	HotSpot C++ Unit-Test Framework
233	Generate Run-Time Compiler Tests Automatically	282	jlink: The Java Linker
235	Test Class-File Attributes Generated by javac	283	Enable GTK 3 on Linux
236	Parser API for Nashorn	284	New HotSpot Build System
237	Linux/AArch64 Port	285	Spin-Wait Hints
238	Multi-Release JAR Files	287	SHA-3 Hash Algorithms
240	Remove the JVM TI hprof Agent	288	Disable SHA-1 Certificates
241	Remove the jhat Tool	289	Deprecate the Applet API
243	Java-Level JVM Compiler Interface	290	Filter Incoming Serialization Data
244	TLS Application-Layer Protocol Negotiation Extension	291	Deprecate the Concurrent Mark Sweep (CMS) Garbage Collector
245	Validate JVM Command-Line Flag Arguments	292	

			Implement Selected ECMAScript 6 Features in Nashorn
246	Leverage CPU Instructions for GHASH and RSA	294	Linux/s390x Port
247	Compile for Older Platform Versions	295	Ahead-of-Time Compilation
248	Make G1 the Default Garbage Collector	297	Unified arm32/arm64 Port
249	OCSP Stapling for TLS	298	Remove Demos and Samples
250	Store Interned Strings in CDS Archives	299	Reorganize Documentation
251	Multi-Resolution Images		

Les principales API ajoutées sont :

- JSR 308 (JEP 104) : Annotations on Java Types
- JSR 310 (JEP 150) : Date and Time API
- JSR 335 (JEP 126) : le projet Lambda dont le but est d'intégrer les expressions lambda, appelées closure dans d'autres langages
- JSR 223 (JEP 174) : le projet Nashorn dont le but est de proposer un moteur d'exécution Javascript. La commande jjs permet d'invoquer ce moteur en ligne de commande.

Java introduit le système de modules pour la plate-forme Java (JPMS : Java Plateform Modules system) issu des travaux du projet Jigsaw. Il est à l'origine de nombreuses controverses.

1.4.13.1. Java 9 update

Java 9.0.1

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/9-0-1-relnotes.html

La version externe est Java 9.0.1, le numéro de build est 9.0.1+11.

Java 9.0.4

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/9-0-4-relnotes.html

La version externe est Java 9.0.4, le numéro de build est 9.0.4+11.

1.4.14. Java SE 10

Java SE 10 est issu des travaux d'Oracle, du JCP et des travaux d'implémentation du projet open source OpenJDK.

Les spécifications de Java SE 10 sont définit dans la JSR 383.

Une douzaine de JEP ont été incluses dans Java 10 :

Numéro	Titre	Numéro	Titre
286	Local-Variable Type Inference	313	Remove the Native-Header Generation Tool (javah)

296	Consolidate the JDK Forest into a Single Repository	314	Additional Unicode Language-Tag Extensions
304	Garbage-Collector Interface	316	Heap Allocation on Alternative Memory Devices
307	Parallel Full GC for G1	317	Experimental Java-Based JIT Compiler
310	Application Class-Data Sharing	319	Root Certificates
312	Thread-Local Handshakes	322	Time-Based Release Versioning

Java 10 intègre une évolution dans le langage : la possibilité de définir des variables locales dont le type est inféré par le compilateur en utilisant l'instruction var.

1.4.14.1. Java 10 update

Java 10.0.1

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/10-0-1-relnotes.html

La version externe est Java 10.0.1, le numéro de build est 10.0.1+10.

Java 10.0.2

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/10-0-2-relnotes.html

La version externe est Java 10.0.2, le numéro de build est 10.0.2+11.

1.4.15. Java SE 11

Java SE 11 est issu des travaux d'Oracle, du JCP et des travaux d'implémentation du projet open source OpenJDK.

Les spécifications de Java SE 11 sont définit dans la JSR 384.

17 JEP ont été incluses dans Java 11 :

Numéro	Titre	Numéro	Titre
181	Nest-Based Access Control	328	Flight Recorder
309	Dynamic Class-File Constants	329	ChaCha20 and Poly1305 Cryptographic Algorithms
315	Improve Aarch64 Intrinsics	330	Launch Single-File Source-Code Programs
318	Epsilon : A No-Op Garbage Collector	331	Low-Overhead Heap Profiling
320	Remove the Java EE and CORBA Modules	332	Transport Layer Security (TLS) 1.3
321	HTTP Client (Standard)	333	ZGC : A Scalable Low-Latency Garbage Collector (Experimental)
323	Local-Variable Syntax for Lambda Parameters	335	Deprecate the Nashorn JavaScript Engine
324	Key Agreement with Curve25519 and Curve448	336	Deprecate the Pack200 Tools and API

327	Unicode 10		

Java SE 11 est une version particulière désignée comme LTS (Long Time Support) : il est possible d'obtenir un support commercial sur cette version auprès de différents fournisseurs.

1.4.15.1. Java 11 update

Java 11.0.1 (octobre 2018)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/11-0-1-relnotes.html

La version externe est Java 11.0.1, le numéro de build est 11.0.1+13.

Java 11.0.2 (janvier 2019)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/11-0-2-relnotes.html

La version externe est Java 11.0.2, le numéro de build est 11.0.2+9.

Swing propose un support de GTK 3.20.

Java 11.0.3 (avril 2019)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/11-0-3-relnotes.html

Cette version propose un support pour la nouvelle ère du calendrier japonais nommé Reiwa.

La version externe est Java 11.0.3, le numéro de build est 11.0.3+12.

Java 11.0.4 (juillet 2019)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/11-0-4-relnotes.html

la JVM détecte correctement la version Server 2019 de Windows.

La version externe est Java 11.0.4, le numéro de build est 11.0.4+10.

Java 11.0.5 (octobre 2019)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/11-0-5-relnotes.html

4 nouveaux événements de JFR relatifs à la sécurité ont été ajoutés.

La version externe est Java 11.0.5, le numéro de build est 11.0.5+10.

Java 11.0.6 (janvier 2020)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/11-0-6-relnotes.html

La version externe est Java 11.0.6, le numéro de build est 11.0.6+8.

Java 11.0.7 (avril 2020)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/11-0-7-relnotes.html

La version externe est Java 11.0.7, le numéro de build est 11.0.7+8.

Java 11.0.8 (juillet 2020)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/11-0-8-relnotes.html

Les méthodes getFreePhysicalMemorySize(), getTotalPhysicalMemorySize(), getFreeSwapSpaceSize(), getTotalSwapSpaceSize(), getSystemCpuLoad()de l'OperatingSystemMXBean retourne des informtions tenant compte des limitations appliquées lors de l'exécution de la JVM dans un conteneur.

La version externe est Java 11.0.8, le numéro de build est 11.0.8+10.

Java 11.0.9 (octobre 2020)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/11-0-9-relnotes.html

La version externe est Java 11.0.9, le numéro de build est 11.0.9+7.

Java 11.0.10 (janvier 2021)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/11-0-10-relnotes.html

La version externe est Java 11.0.10, le numéro de build est 11.0.10+8.

Java 11.0.11 (avril 2021)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/11-0-11-relnotes.html

Les options --print-module-deps, --list-deps, et --list-reduce-deps de jdeps ont été améliorées pour prendre en compte les dépendancces transitives par défaut.

La version externe est Java 11.0.11, le numéro de build est 11.0.11+9.

Java 11.0.12 (juillet 2021)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/11-0-12-relnotes.html

Les algorithmes de chiffrement par défaut utilisés dans un keystore PKCS #12 ont été mis à jour pour utiliser des algorithmes plus forts basés sur AES-256 et SHA-256.

La version externe est Java 11.0.12, le numéro de build est 11.0.12+8.

Java 11.0.13 (octobre 2021)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/11-0-13-relnotes.html

Le compilateur JIT expérimental Graal écrit en Java (JEP 317) a été retiré de la JVM. L'outil expérimental de compilation Java Ahead-of-Time jaotc a aussi été retiré.

La version externe est Java 11.0.13, le numéro de build est 11.0.13+10.

Java 11.0.14 (janvier 2022)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/11-0-14-relnotes.html

La version externe est Java 11.0.14, le numéro de build est 11.0.14+8.

Java 11.0.15 (avril 2022)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/11-0-15-relnotes.html

La version externe est Java 11.0.15, le numéro de build est 11.0.15+8.

Java 11.0.16 (juillet 2022)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/11-0-16-relnotes.html

La version externe est Java 11.0.16, le numéro de build est 11.0.16+11.

Java 11.0.17 (octobre 2022)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/11-0-17-relnotes.html

La version externe est Java 11.0.17, le numéro de build est 11.0.15+10.

Java 11.0.18 (janvier 2023)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/11-0-18-relnotes.html

La version externe est Java 11.0.18, le numéro de build est 11.0.18+9.

Java 11.0.19 (avril 2023)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/11-0-19-relnotes.html

La version externe est Java 11.0.19, le numéro de build est 11.0.19+9.

Java 11.0.20 (juillet 2023)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/11-0-20-relnotes.html

La version externe est Java 11.0.20, le numéro de build est 11.0.20+9.

Java 11.0.21 (octobre 2023)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/11-0-21-relnotes.html

La version externe est Java 11.0.21, le numéro de build est 11.0.21+9.

1.4.16. Java SE 12

Java SE 12 est issu des travaux d'Oracle, du JCP et des travaux d'implémentation du projet open source OpenJDK.

Les spécifications de Java SE 12 sont définit dans la JSR 386.

Huit JEP ont été incluses dans Java 12 :

Numéro	Titre	Numéro	Titre
189	Shenandoah: A Low-Pause-Time Garbage Collector (Experimental)	340	One AArch64 Port, Not Two
230	Microbenchmark Suite	341	Default CDS Archives
325	Switch Expressions (Preview)	344	Abortable Mixed Collections for G1
334	JVM Constants API	346	Promptly Return Unused Committed Memory from G1

Java 12 intègre en mode preview une évolution dans l'instruction switch et un nouveau ramasse-miettes expérimental.

1.4.16.1. Java 12 update

Java 12.0.1 (avril 219)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/12-0-1-relnotes.html

La version externe est Java 12.0.1, le numéro de build est 12.0.1+12.

Java 12.0.2 (juillet 2019)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/12-0-2-relnotes.html

La version externe est Java 12.0.2, le numéro de build est 12.0.2+10.

1.4.17. Java SE 13

Java SE 13 est issu des travaux d'Oracle, du JCP et des travaux d'implémentation du projet open source OpenJDK.

Les spécifications de Java SE 13 sont définit dans la JSR 388.

Cinq JEP ont été incluses dans Java 13 :

Numéro	Titre	Numéro	Titre
350	Dynamic CDS Archives	354	Switch Expressions (Preview)
351	ZGC: Uncommit Unused Memory	355	Text Blocks (Preview)
353	Reimplement the Legacy Socket API		

Java 13 propose en mode preview deux évolutions dans le langage : l'utilisation de yield dans l'instruction switch et les Text Blocks.

1.4.17.1. Java 13 update

Java 13.0.1 (octobre 2019)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/13-0-1-relnotes.html

La version externe est Java 13.0.1, le numéro de build est 13.0.1+09.

Java 13.0.2 (janvier 2020)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/13-0-2-relnotes.html

La version externe est Java 13.0.2, le numéro de build est 13.0.2+8.

1.4.18. Java SE 14

Java SE 14 est issu des travaux d'Oracle, du JCP et des travaux d'implémentation du projet open source OpenJDK.

Les spécifications de Java SE 14 sont définit dans la JSR 389.

Seize JEP ont été incluses dans Java 14 :

Numéro	Titre	Numéro	Titre
305	Pattern Matching for instanceof (Preview)	362	Deprecate the Solaris and SPARC Ports
362	Deprecate the Solaris and SPARC Ports	363	Remove the Concurrent Mark Sweep (CMS) Garbage Collector
343	Packaging Tool (Incubator)	364	ZGC on macOS
345	NUMA-Aware Memory Allocation for G1	365	ZGC on Windows
349	JFR Event Streaming	366	Deprecate the ParallelScavenge + SerialOld GC Combination
352	Non-Volatile Mapped Byte Buffers	367	Remove the Pack200 Tools and API
358	Helpful NullPointerExceptions	368	Text Blocks (Second Preview)
359	Records (Preview)	370	Foreign-Memory Access API (Incubator)

Java 14 intègre en standard les évolutions dans l'instruction switch.

Java 14 propose en mode preview trois évolutions dans le langage : les Texts blocks, les Records, le pattern matching pour l'instruction instanceof.

Il y a aussi des changements dans les ramasse-miettes : CMS est retiré et ZGC est proposé en expérimental sous Windows et MacOS.

1.4.18.1. Java 14 update

Java 14.0.1 (avril 2020)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/14-0-1-relnotes.html

La version externe est Java 14.0.1, le numéro de build est 14.0.1+07.

Java 14.0.2 (juillet 2020)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/14-0-2-relnotes.html

La version externe est Java 14.0.2, le numéro de build est 14.0.2+12.

1.4.19. Java SE 15

Java SE 15 est issu des travaux d'Oracle, du JCP et des travaux d'implémentation du projet open source OpenJDK.

Les spécifications de Java SE 15 sont définit dans la JSR 390.

Quatorze JEP ont été incluses dans Java 15 :

Numéro	Titre	Numéro	Titre
339	Edwards-Curve Digital Signature Algorithm (EdDSA)	377	ZGC: A Scalable Low-Latency Garbage Collector
360	Sealed Classes (Preview)	378	Text Blocks

371	Hidden Classes	379	Shenandoah: A Low-Pause-Time Garbage Collector
372	Remove the Nashorn JavaScript Engine	381	Remove the Solaris and SPARC Ports
373	Reimplement the Legacy DatagramSocket API	383	Foreign-Memory Access API (Second Incubator)
374	Disable and Deprecate Biased Locking	384	Records (Second Preview)
375	Pattern Matching for instanceof (Second Preview)	385	Deprecate RMI Activation for Removal

1.4.19.1. Java 15 update

Java 15.0.1 (octobre 2020)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/15-0-1-relnotes.html

La version externe est Java 15.0.1, le numéro de build est 15.0.1+09.

Java 15.0.2 (janvier 2021)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/15-0-2-relnotes.html

La version externe est Java 15.0.2, le numéro de build est 15.0.2+7.

1.4.20. Java SE 16

Java SE 16 est issu des travaux d'Oracle, du JCP et des travaux d'implémentation du projet open source OpenJDK.

Les spécifications de Java SE 16 sont définit dans la JSR 391.

Dix sept JEP ont été incluses dans Java 16 :

Numéro	Titre	Numéro	Titre
338	Vector API (Incubator)	388	Windows/AArch64 Port
347	Enable C++14 Language Features	389	Foreign Linker API (Incubator)
357	Migrate from Mercurial to Git	390	Warnings for Value-Based Classes
369	Migrate to GitHub	392	Packaging Tool
376	ZGC: Concurrent Thread-Stack Processing	393	Foreign-Memory Access API (Third Incubator)
380	Unix-Domain Socket Channels	394	Pattern Matching for instanceof
386	Alpine Linux Port	395	Records
387	Elastic Metaspace	396	Strongly Encapsulate JDK Internals by Default
397	Sealed Classes (Second Preview)		

En Java, il est possible de créer des classes locales, qui sont des classes définies dans une méthode.

Historiquement, les interfaces et les énumérations locales sont proscrites en raison de préoccupations sémantiques : les énumérations imbriquées et les interfaces imbriquées sont implicitement statiques, donc les énumérations locales et les interfaces locales devraient être implicitement statiques aussi. Cependant, les déclarations locales dans le langage Java (variables locales, classes locales) ne sont jamais statiques par défaut. Cette sémantique a évolué avec l'introduction des record locaux, en permettant à une déclaration locale d'être statique, ouvrant la porte aux énumérations locales et aux interfaces locales.

La JEP 395 incluse dans Java 16 introduit dans le langage Java la possibilité d'utiliser des interfaces et des énumérations locales.

Les énumérations imbriquées et les interfaces imbriquées sont déjà implicitement statiques. Par souci de cohérence, les énumérations locales et des interfaces locales sont également implicitement statiques.

Les interfaces et énumérations locales ne peuvent pas capturer les variables non static du contexte englobant comme les paramètres de la méthode.

1.4.20.1. Java 16 update

Java 16.0.1 (avril 2021)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/16-0-1-relnotes.html

La version externe est Java 16.0.1, le numéro de build est 16.0.1+9.

Java 16.0.2 (juillet 2021)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/16-0-2-relnotes.html

La version externe est Java 16.0.2, le numéro de build est 16.0.2+7.

1.4.21. Java SE 17

Java SE 17 est issu des travaux d'Oracle, du JCP et des travaux d'implémentation du projet open source OpenJDK.

Les spécifications de Java SE 17 sont définit dans la JSR 392.

Quatorze JEP ont été incluses dans Java 17 :

Numéro	Titre	Numéro	Titre
306	Restore Always-Strict Floating-Point Semantics	407	Remove RMI Activation
356	Enhanced Pseudo-Random Number Generators	409	Sealed Classes
382	New macOS Rendering Pipeline	410	Remove the Experimental AOT and JIT Compiler
391	macOS/AArch64 Port	411	Deprecate the Security Manager for Removal
398	Deprecate the Applet API for Removal	412	Foreign Function & Memory API (Incubator)
403	Strongly Encapsulate JDK Internals	414	Vector API (Second Incubator)
406	Pattern Matching for switch (Preview)	415	Context-Specific Deserialization Filters

Java SE 17 est une version particulière désignée comme LTS (Long Time Support) : il est possible d'obtenir un support commercial sur cette version auprès de différents fournisseurs.

1.4.21.1. Java 17 update

Java 17.0.1 (octobre 2021)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/17-0-1-relnotes.html

La version externe est Java 17.0.1, le numéro de build est 17.0.1+12.

Java 17.0.2 (janvier 2022)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/17-0-2-relnotes.html

La version externe est Java 17.0.2, le numéro de build est 17.0.2+8.

Java 17.0.3 (avril 2022)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/17-0-3-relnotes.html

La version externe est Java 17.0.3, le numéro de build est 17.0.3+8.

Java 17.0.4 (juillet 2022)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/17-0-4-relnotes.html

La version externe est Java 17.0.4, le numéro de build est 17.0.4+11.

À partir de cette version du JDK, par défaut, la JVM Hotspot ne prend plus en compte le paramètre "cpu.shares" de cgroups lorsqu'elle détermine le nombre de threads à utiliser par les différents pools de threads. L'option de ligne de commande -XX:+UseContainerCpuShares peut être utilisée pour revenir au comportement précédent. Cette option est dépréciée et pourrait être supprimée dans une prochaine version du JDK.

Java 17.0.5 (octobre 2022)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/17-0-5-relnotes.html

La version externe est Java 17.0.5, le numéro de build est 17.0.5+9.

Java 17.0.6 (janvier 2023)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/17-0-6-relnotes.html

La version externe est Java 17.0.6, le numéro de build est 17.0.6+9.

Java 17.0.7 (avril 2023)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/17-0-7-relnotes.html

La version externe est Java 17.0.7, le numéro de build est 17.0.7+8.

Java 17.0.8 (juillet 2023)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/17-0-8-relnotes.html

La version externe est Java 17.0.8, le numéro de build est 17.0.8+9.

Java 17.0.9 (octobre 2023)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/17-0-9-relnotes.html

La version externe est Java 17.0.9, le numéro de build est 17.0.9+11.

1.4.22. Java SE 18

Java SE 18 est issu des travaux d'Oracle, du JCP et des travaux d'implémentation du projet open source OpenJDK.

Les spécifications de Java SE 18 sont définit dans la JSR 393.

Neuf JEP sont été incluses dans Java 18 :

Numéro	Titre	Numéro	Titre
400	UTF-8 by Default	418	Internet-Address Resolution SPI
408	Simple Web Server	419	Foreign Function & Memory API (Second Incubator)
413	Code Snippets in Java API Documentation	420	Pattern Matching for switch (Second Preview)
416	Reimplement Core Reflection with Method Handles	421	Deprecate Finalization for Removal
417	Vector API (Third Incubator)		

1.4.22.1. Java 18 update

Java 18.0.1 (mai 2022)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/18-0-1-relnotes.html

La version externe est Java 18.0.1, le numéro de build est 18.0.1+10.

Java 18.0.2 (août 2022)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/18-0-2-relnotes.html

La version externe est Java 18.0.2, le numéro de build est 18.0.2+9.

1.4.23. Java SE 19

Java SE 19 est issu des travaux d'Oracle, du JCP et des travaux d'implémentation du projet open source OpenJDK.

Les spécifications de Java SE 19 sont définit dans la JSR 394.

Sept JEP sont été incluses dans Java 19 :

Numéro	Titre	Numéro	Titre
405	Record Patterns (Preview)	426	Vector API (Fourth Incubator)
422	Linux/RISC-V Port	427	Pattern Matching for switch (Third Preview)
424	Foreign Function & Memory API (Preview)	428	Structured Concurrency (Incubator)
425	Virtual Threads (Preview)		

1.4.23.1. Java 19 update

Java 19.0.1 (octobre 2022)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/19-0-1-relnotes.html

La version externe est Java 19.0.1, le numéro de build est 19.0.1+10.

Java 19.0.2 (janvier 2023)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/19-0-2-relnotes.html

La version externe est Java 19.0.2, le numéro de build est 19.0.2+7.

1.4.24. Java SE 20

Java SE 20 est issu des travaux d'Oracle, du JCP et des travaux d'implémentation du projet open source OpenJDK.

Les spécifications de Java SE 20 sont définit dans la JSR 395.

Sept JEP sont été incluses dans Java 20 :

Numéro	Titre	Numéro	Titre
429	Scoped Values (Incubator)	436	Virtual Threads (Second Preview)
432	Record Patterns (Second Preview)	437	Structured Concurrency (Second Incubator)
433	Pattern Matching for switch (Fourth Preview)	438	Vector API (Fifth Incubator)

434	Foreign Function & Memory API (Second Preview)		

1.4.24.1. Java 20 update

Java 20.0.1 (avril 2023)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/20-0-1-relnotes.html

La version externe est Java 20.0.1, le numéro de build est 20.0.1+9.

Java 20.0.2 (juillet 2023)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/20-0-2-relnotes.html

La version externe est Java 20.0.2, le numéro de build est 20.0.2+9.

1.4.25. Java SE 21

Java SE 21 est issu des travaux d'Oracle, du JCP et des travaux d'implémentation du projet open source OpenJDK.

Les spécifications de Java SE 21 sont définit dans la JSR 396.

Quinze JEP sont été incluses dans Java 21 :

Numéro	Titre	Numéro	Titre
430	String Templates (Preview)	445	Unnamed Classes and Instance Main Methods (Preview)
431	Sequenced Collections	446	Scoped Values (Preview)
439	Generational ZGC	448	Vector API (Sixth Incubator)
440	Record Patterns	449	Deprecate the Windows 32-bit x86 Port for Removal
441	Pattern Matching for switch	451	Prepare to Disallow the Dynamic Loading of Agents
442	Foreign Function & Memory API (Third Preview)	452	Key Encapsulation Mechanism API
443	Unnamed Patterns and Variables (Preview)	453	Structured Concurrency (Preview)
444	Virtual Threads		

1.4.25.1. Java 21 update

Java 21.0.1 (ocotbre 2023)

Cette mise à jour contient des améliorations relatives à la sécurité et des corrections de bugs. La liste complète est consultable à l'url : https://www.oracle.com/java/technologies/javase/21-0-1-relnotes.html

La version externe est Java 21.0.1, le numéro de build est 21.0.1+12.

1.4.26. Le résumé des différentes versions

Au fur et à mesure des nouvelles versions de Java, le nombre de packages, de classes de modules (à partir de Java 9) fluctue :

Plateforme	Version	Nombre de packages	Nombre de classes	Nombre de modules
Java	1.0	8	201	
	1.1	23	503	
	1.2	59	1520	
J2SE	1.3	76	1840	
	1.4	135	2990	
	5.0	166	3280	
Java SE	6	202	3780	
	7	209	4024	
	8	217	4240	
	9	315	6005	98
	10	314	6002	99
	11	223	4410	71
	12	225	4432	71
	13	223	4403	71
	14	225	4419	73
	15	223	4265	73
	16	224	4373	71
	17	224	4371	71
	18	224	4405	71
	19	225	4419	71
	20	225	4429	71
	21	223	4443	70

1.4.27. Le support des différentes versions

Chaque version de Java possède une durée de support durant laquelle Sun/Oracle propose des mises à jour de la plate-forme concernant la correction de bugs et de failles de sécurité.

La date de fin de ce support est définie par une date de fin de vie (End Of Life).

Version	Date de diffusion (GA)	Date de fin de vie (EOL) par Oracle
J2SE 1.4	Février 2002	Octobre 2008
J2SE 5.0	Mai 2004	Octobre 2009
Java SE 6	Décembre 2006	Initialement prévue en juillet 2012, repoussée à novembre 2012
Java SE 7	Juillet 2011	Avril 2015
Java SE 8	Mars 2014	

		Initialement prévue en mars 2017, repoussée à janvier 2019 puis à décembre 2030
Java SE 9	Septembre 2017	Mars 2018
Java SE 10	Mars 2018	Septembre 2018
Java SE 11	Septembre 2018	Septembre 2023, repoussée à septembre 2026
Java SE 12	Mars 2019	Septembre 2019
Java SE 13	Septembre 2019	Mars 2020
Java SE 14	Mars 2020	Septembre 2020
Java SE 15	Septembre 2020	Mars 2021
Java SE 16	Mars 2021	Septembre 2021
Java SE 17	Septembre 2021	Septembre 2026
Java SE 18	Mars 2022	Septembre 2022
Java SE 19	Septembre 2022	Mars 2023
Java SE 20	Mars 2023	Septembre 2023
Java SE 21	Septembre 2023	Septembre 2028

Il est cependant possible d'obtenir une prolongation en souscrivant à un support payant auprès d'Oracle ou d'autres fournisseurs de JDK.

Les dates fournies concernent Oracle : d'autres fournisseurs peuvent proposer des dates de fin de support différents.

1.5. Un rapide tour d'horizon des API et de quelques outils

La communauté Java est très productive car elle regroupe :

- Sun MicroSystems, le fondateur de Java et Oracle depuis son acquisition de Sun MicroSystems
- le JCP (Java Community Process) : c'est le processus de traitement des évolutions de Java. Chaque évolution est traitée dans une JSR (Java Specification Request) par un groupe de travail constitué de différents acteurs du monde Java
- des acteurs commerciaux dont tous les plus grands acteurs du monde informatique excepté Microsoft
- la communauté libre qui produit un très grand nombre d'API et d'outils pour Java
- Les JUGs

Ainsi l'ensemble des API et des outils utilisables est énorme et évolue très rapidement. Les tableaux ci-dessous tentent de recenser les principaux par thème.

J2SE 1.4

Java Bean	RMI	IO	Applet
Reflexion	Collection	Logging	AWT
Net (réseau)	Preferences	Security	JFC
Internationalisation	Exp régulière		Swing

Les outils du JDK de Sun/Oracle

Jar	Javadoc	Java Web Start	JWSDK

Les outils libres (les plus connus)

Apache Tomcat	Apache Ant	JBoss	Apache Axis
Eclipse	NetBeans	Apache Maven	

Les autres API

Données	Web	Entreprise	XML	Divers
JDBC	Servlets	Java Mail	JAXP	JAI
JDO	JSP	JNDI	SAX	JAAS
JPA	JSTL	EJB	DOM	JCA
	Java Server Faces	JMS	JAXB	JCE
		JMX	Stax	Java Help
		JTA	Services Web	JMF
		RMI-IIOP	JAXM	JSSE
		Java IDL	JAXR	Java speech
		JINI	JAX-RPC	Java 3D
		JXTA	SAAJ	
			JAX-WS	

Les API de la communauté open source

Données	Web	Entreprise	XML	Divers
OJB	Jakarta Struts	Spring	Apache Xerces	Apache Log4j
Castor	Webmacro	Apache Axis	Apache Xalan	JUnit
Hibernate	Expresso	Seams	JDOM	Mockito
Ibatis/MyBatis	Barracuda		DOM4J	EasyMock
	Tapestry			
	Wicket			
	GWT			

1.6. Les différences entre Java et JavaScript

Il ne faut pas confondre Java et JavaScript. JavaScript est un langage développé par Netscape Communications.

La syntaxe des deux langages est très proche car ils dérivent tous les deux du C++.

Il existe de nombreuses différences entre les deux langages :

	Java	Javascript

Auteur	Développé par Sun Microsystems	Développé par Netscape Communications
Format	Compilé sous forme de bytecode	Interprété
Stockage	Applet téléchargé comme un élément de la page web	Source inséré dans la page web
Utilisation	Utilisable pour développer tous les types d'applications	Utilisable pour "dynamiser" les pages web
Exécution	Exécuté dans la JVM	Exécuté par un moteur dans le navigateur
POO	Orienté objets reposant sur des classes	Orienté objets reposant sur le prototypage
Typage	Fortement typé	Pas de contrôle de type
Complexité du code	Code relativement complexe	Code simple

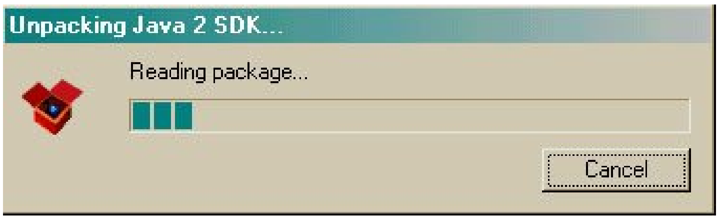
1.7. L'installation du JDK

Le JDK et la documentation sont librement téléchargeables sur le site d'Oracle : https://www.oracle.com/java/technologies/

1.7.1. L'installation de la version 1.3 du JDK de Sun sous Windows 9x

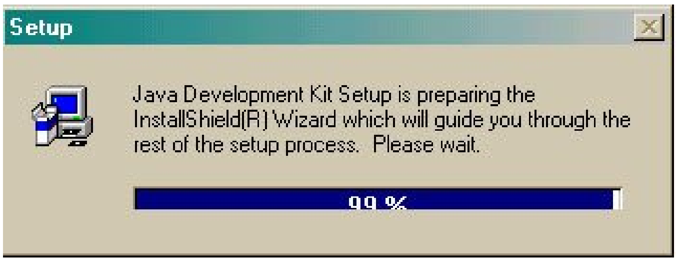
Pour installer le JDK 1.3 sous Windows 9x, il suffit de télécharger et d'exécuter le programme : j2sdk1_3_0-win.exe

Le programme commence par désarchiver les composants.


Unpacking Java 2 SDK...

Reading package...

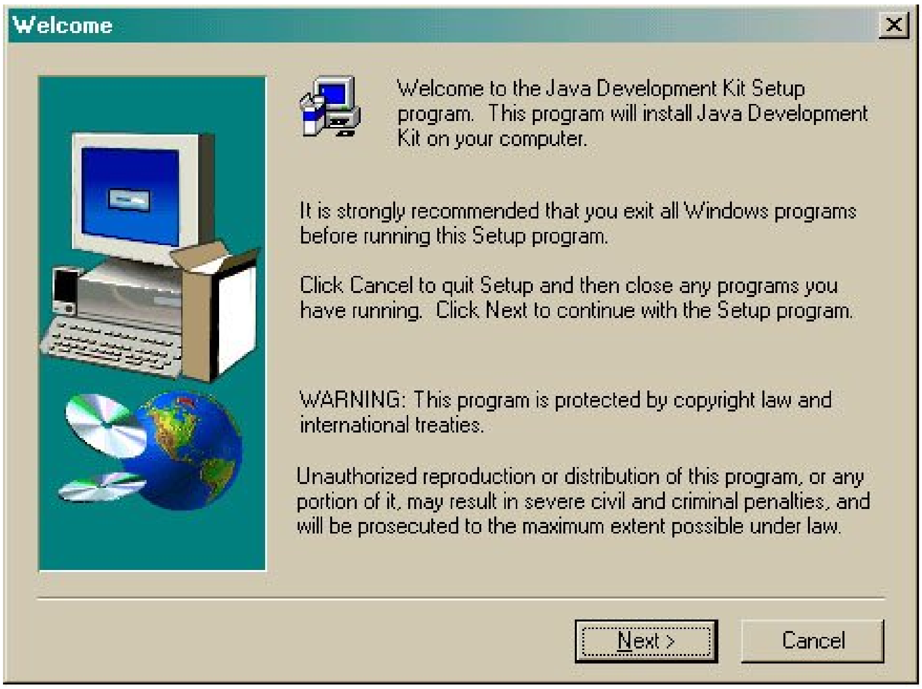
Le programme utilise InstallShield pour guider et réaliser l'installation.


Setup

Java Development Kit Setup is preparing the InstallShield(R) Wizard which will guide you through the rest of the setup process. Please wait.

99 %

L'installation vous souhaite la bienvenue et vous donne quelques informations d'usage.



Welcome

Welcome to the Java Development Kit Setup program. This program will install Java Development Kit on your computer.

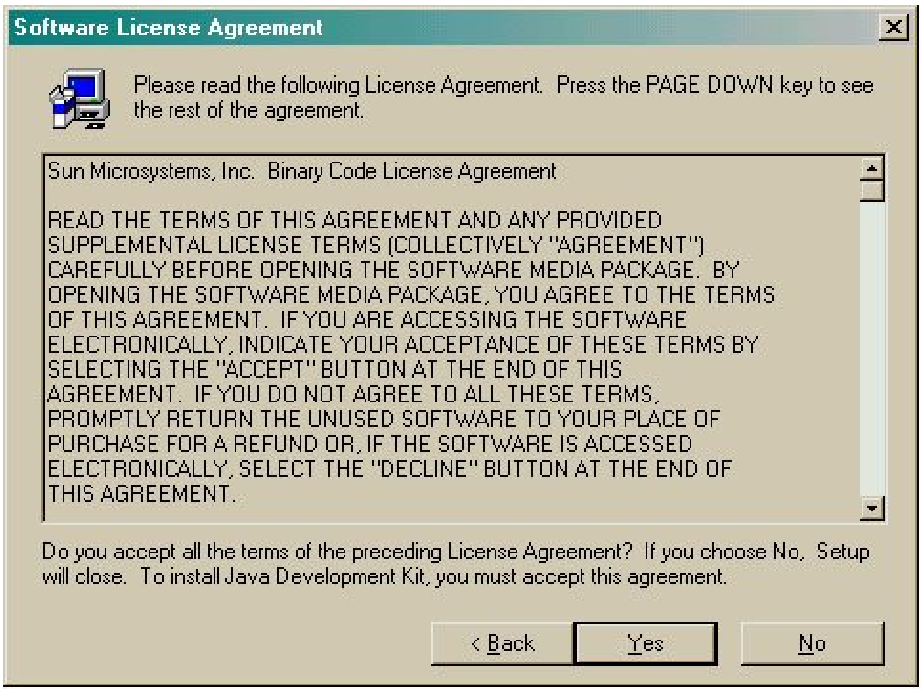
It is strongly recommended that you exit all Windows programs before running this Setup program.

Click Cancel to quit Setup and then close any programs you have running. Click Next to continue with the Setup program.

WARNING: This program is protected by copyright law and international treaties.

Unauthorized reproduction or distribution of this program, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted to the maximum extent possible under law.

L'installation vous demande ensuite de lire et d'approuver les termes de la licence d'utilisation.



Software License Agreement

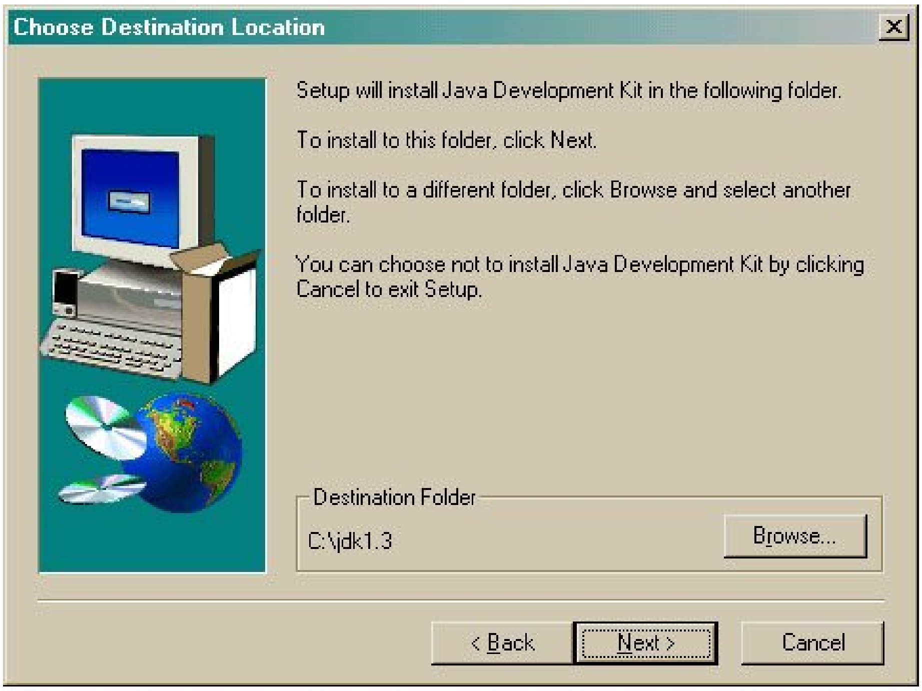
Please read the following License Agreement. Press the PAGE DOWN key to see the rest of the agreement.

Sun Microsystems, Inc. Binary Code License Agreement

READ THE TERMS OF THIS AGREEMENT AND ANY PROVIDED SUPPLEMENTAL LICENSE TERMS (COLLECTIVELY "AGREEMENT") CAREFULLY BEFORE OPENING THE SOFTWARE MEDIA PACKAGE. BY OPENING THE SOFTWARE MEDIA PACKAGE, YOU AGREE TO THE TERMS OF THIS AGREEMENT. IF YOU ARE ACCESSING THE SOFTWARE ELECTRONICALLY, INDICATE YOUR ACCEPTANCE OF THESE TERMS BY SELECTING THE "ACCEPT" BUTTON AT THE END OF THIS AGREEMENT. IF YOU DO NOT AGREE TO ALL THESE TERMS, PROMPTLY RETURN THE UNUSED SOFTWARE TO YOUR PLACE OF PURCHASE FOR A REFUND OR, IF THE SOFTWARE IS ACCESSED ELECTRONICALLY, SELECT THE "DECLINE" BUTTON AT THE END OF THIS AGREEMENT.

Do you accept all the terms of the preceding License Agreement? If you choose No, Setup will close. To install Java Development Kit, you must accept this agreement.

L'installation vous demande le répertoire dans lequel le JDK va être installé. Le répertoire proposé par défaut est pertinent car il est simple.


Choose Destination Location

Setup will install Java Development Kit in the following folder.

To install to this folder, click Next.

To install to a different folder, click Browse and select another folder.

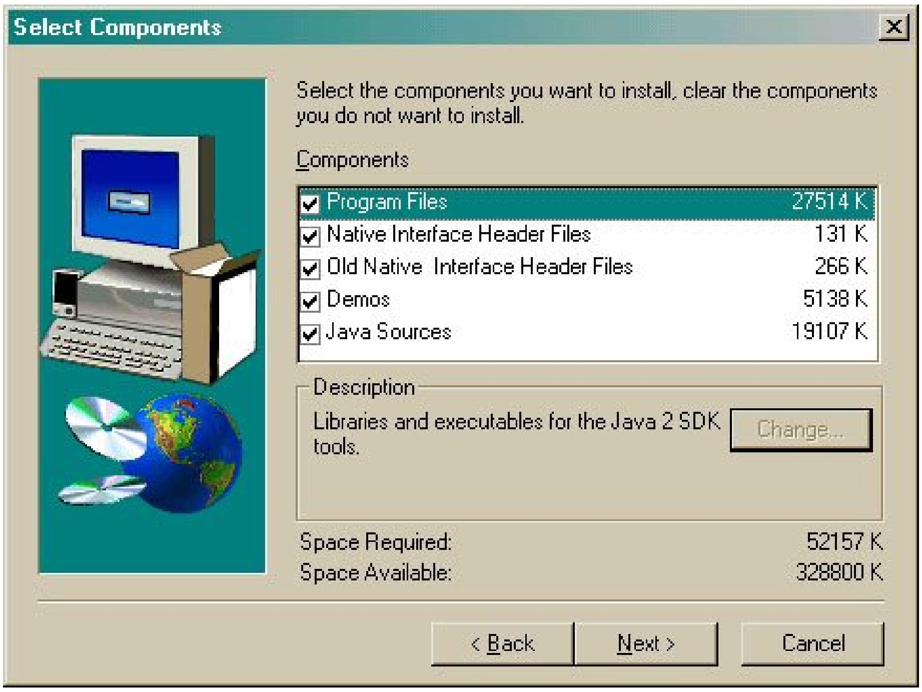
You can choose not to install Java Development Kit by clicking Cancel to exit Setup.

Destination Folder

C:\jdk1.3

L'installation vous demande les composants à installer :

- Program Files est obligatoire pour une première installation
- Les interfaces natives ne sont utiles que pour réaliser des appels de code natif dans les programmes Java
- Les démos sont utiles car elles fournissent quelques exemples
- les sources contiennent les sources de la plupart des classes Java écrites en Java. Attention à l'espace disque nécessaire à cet élément


Select Components

Select the components you want to install, clear the components you do not want to install.

Components

☑ Program Files	27514 K
☑ Native Interface Header Files	131 K
☑ Old Native Interface Header Files	266 K
☑ Demos	5138 K
☑ Java Sources	19107 K

Description

Libraries and executables for the Java 2 SDK tools.

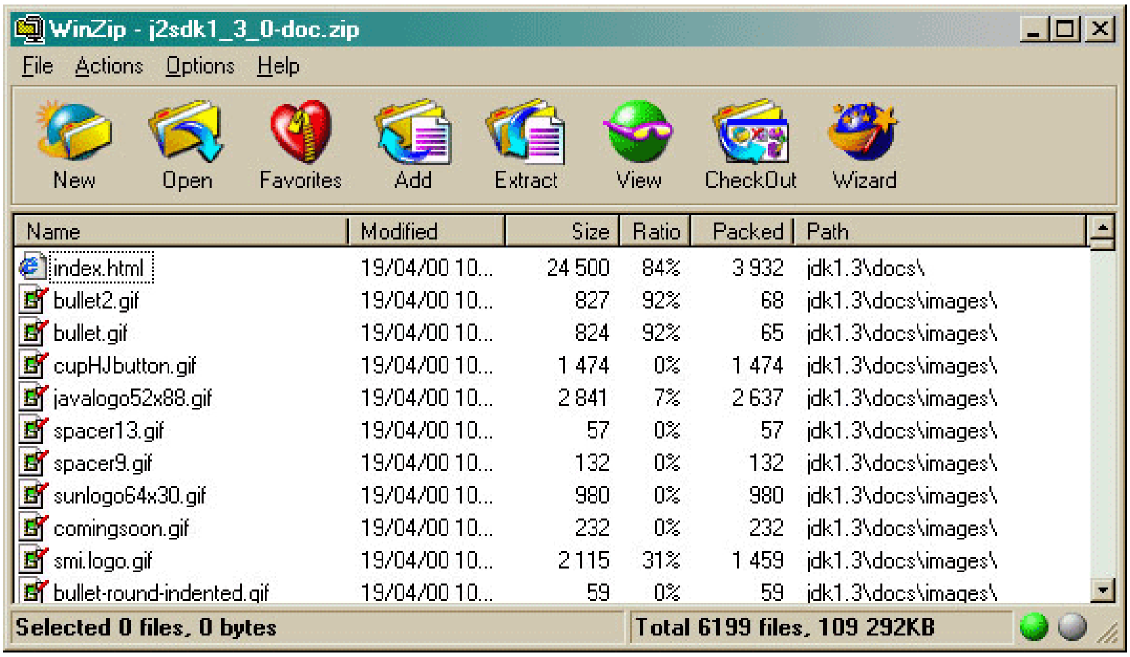
Space Required: 52157 K

Space Available: 328800 K

L'installation se poursuit par la copie des fichiers et la configuration du JRE.

1.7.2. L'installation de la documentation de Java 1.3 sous Windows

L'archive contient la documentation sous forme d'arborescence dont la racine est jdk1.3\docs.

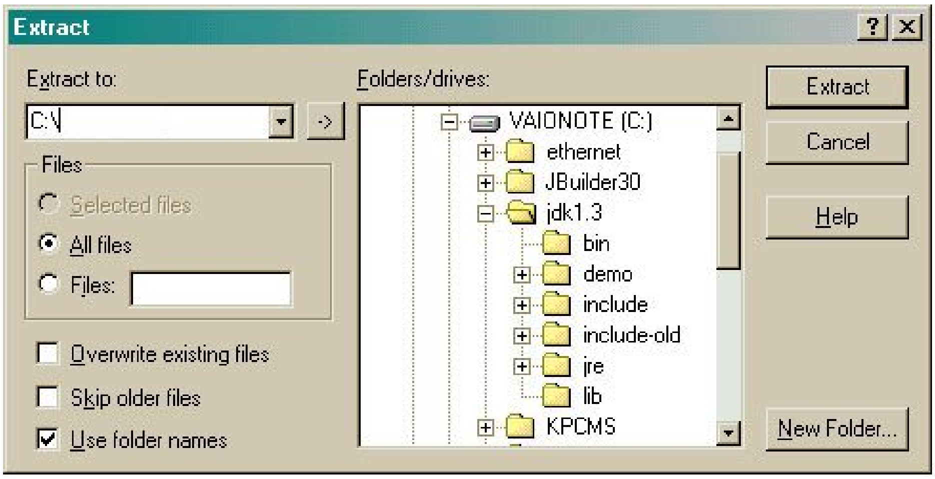


Name	Modified	Size	Ratio	Packed	Path
index.html	19/04/00 10...	24 500	84%	3 932	jdk1.3\docs\
bullet2.gif	19/04/00 10...	827	92%	68	jdk1.3\docs\images\
bullet.gif	19/04/00 10...	824	92%	65	jdk1.3\docs\images\
cupHJbutton.gif	19/04/00 10...	1 474	0%	1 474	jdk1.3\docs\images\
javalogo52x88.gif	19/04/00 10...	2 841	7%	2 637	jdk1.3\docs\images\
spacer13.gif	19/04/00 10...	57	0%	57	jdk1.3\docs\images\
spacer9.gif	19/04/00 10...	132	0%	132	jdk1.3\docs\images\
sunlogo64x30.gif	19/04/00 10...	980	0%	980	jdk1.3\docs\images\
comingsoon.gif	19/04/00 10...	232	0%	232	jdk1.3\docs\images\
smi.logo.gif	19/04/00 10...	2 115	31%	1 459	jdk1.3\docs\images\
bullet-round-indented.gif	19/04/00 10...	59	0%	59	jdk1.3\docs\images\

Selected 0 files, 0 bytes

Total 6199 files, 109 292KB

Si le répertoire par défaut a été utilisé lors de l'installation, il suffit de décompresser l'archive à la racine du disque C:\.



Extract to: C:\

Files

○ Selected files

● All files

○ Files:

□ Overwrite existing files

□ Skip older files

☑ Use folder names

Folders/drives:

VAIONOTE (C:)
ethernet
JBuilder30
jdk1.3
bin
demo
include
include-old
jre
lib
KPCMS

Il peut être pratique de désarchiver le fichier dans un sous-répertoire, ce qui permet de réunir plusieurs versions de la documentation.

1.7.3. La configuration des variables système sous Windows 9x

Pour un bon fonctionnement du JDK, il est recommandé de paramétrer correctement deux variables systèmes : la variable PATH qui définit les chemins de recherche des exécutables et la variable CLASSPATH qui définit les chemins de rechercher des classes et bibliothèques Java.

Pour configurer la variable PATH, il suffit d'ajouter à la fin du fichier autoexec.bat :

Exemple :

```
SET PATH=%PATH%;C:\JDK1.3\BIN
```

Attention : si une version antérieure du JDK était déjà présente, la variable PATH doit déjà contenir un chemin vers les utilitaires du JDK. Il faut alors modifier ce chemin sinon c'est l'ancienne version qui sera utilisée. Pour vérifier la version du JDK utilisée, il suffit de saisir la commande java -version dans une fenêtre DOS.

La variable CLASSPATH est aussi définie dans le fichier autoexec.bat. Il suffit d'ajouter une ligne ou de modifier la ligne existante définissant cette variable.

Exemple :

```
SET CLASSPATH=C:\JAVA\DEV;
```

Dans un environnement de développement, il est pratique d'ajouter le . qui désigne le répertoire courant dans le CLASSPATH surtout lorsque l'on n'utilise pas d'outils de type IDE. Attention toutefois, cette pratique est fortement déconseillée dans un environnement de production pour ne pas poser de problèmes de sécurité.

Il faudra ajouter par la suite les chemins d'accès aux différents packages requis par les développements afin de les faciliter.

Pour que ces modifications prennent effet dans le système, il faut redémarrer Windows ou exécuter ces deux instructions sur une ligne de commande DOS.

1.7.4. Les éléments du JDK 1.3 sous Windows

Le répertoire dans lequel a été installé le JDK contient plusieurs répertoires. Les répertoires donnés ci-après sont ceux utilisés en ayant gardé le répertoire par défaut lors de l'installation.

Répertoire	Contenu
C:\jdk1.3	Le répertoire d'installation contient deux fichiers intéressants : le fichier readme.html qui fournit quelques informations, des liens web et le fichier src.jar qui contient le source Java de nombreuses classes. Ce dernier fichier n'est présent que si l'option correspondante a été cochée lors de l'installation.
C:\jdk1.3\bin	Ce répertoire contient les exécutables : le compilateur javac, l'interpréteur java, le débuggeur jdb et d'une façon générale tous les outils du JDK.
C:\jdk1.3\demo	Ce répertoire n'est présent que si l'option nécessaire a été cochée lors de l'installation. Il contient des applications et des applets avec leur code source.
C:\jdk1.3\docs	Ce répertoire n'est présent que si la documentation a été décompressée.
C:\jdk1.3\include et C:\jdk1.3\include-old	Ces répertoires ne sont présents que si les options nécessaires ont été cochées lors de l'installation. Il contient des fichiers d'en-tête C (fichier avec l'extension .H) qui permettent de faire interagir du code Java avec du code natif
C:\jdk1.3\jre	Ce répertoire contient le JRE : il regroupe le nécessaire à l'exécution des applications notamment le fichier rt.jar qui regroupe les API. Depuis la version 1.3, le JRE contient deux machines virtuelles : la JVM classique et la JVM utilisant la technologie Hot spot. Cette dernière est bien plus rapide et c'est elle qui est utilisée par défaut. Les éléments qui composent le JRE sont séparés dans les répertoires bin et lib selon leur nature.
C:\jdk1.3\lib	Ce répertoire ne contient plus que quelques bibliothèques notamment le fichier tools.jar. Avec le JDK 1.1 ce répertoire contenait le fichier de la bibliothèque standard. Ce fichier est maintenant dans le répertoire JRE.

1.7.5. L'installation de la version 1.4.2 du JDK de Sun sous Windows

Télécharger sur le site java.sun.com et exécuter le fichier j2sdk-1_4_2_03-windows-i586-p.exe.

Un assistant permet de configurer l'installation au travers de plusieurs étapes :

- La page d'acceptation de la licence (« Licence agreement ») s'affiche
- Lire la licence et si vous l'acceptez, cliquer sur le bouton radio « I accept the terms in the licence agreement », puis cliquez sur le bouton « Next »
- La page de sélection des composants à installer (« Custom setup ») s'affiche, modifiez les composants à installer si nécessaire puis cliquez sur le bouton « Next »
- La page de sélection des plug in pour navigateur (« Browser registration ») permet de sélectionner les navigateurs pour lesquels le plug in Java sera installé, sélectionner ou non le ou les navigateurs détectés, puis cliquez sur le bouton « Install »
- L'installation s'opère en fonction des informations fournies précédemment
- La page de fin s'affiche, cliquez sur le bouton « Finish »

Même si ce n'est pas obligatoire pour fonctionner, il est particulièrement utile de configurer deux variables systèmes : PATH et CLASSPATH.

Dans la variable PATH, il est pratique de rajouter le chemin du répertoire bin du JDK installé pour éviter à chaque appel des commandes du JDK d'avoir à saisir leur chemin absolu.

Dans la variable CLASSPATH, il est pratique de rajouter les répertoires et les fichiers .jar qui peuvent être nécessaires lors des phases de compilation ou d'exécution, pour éviter d'avoir à les préciser à chaque fois.

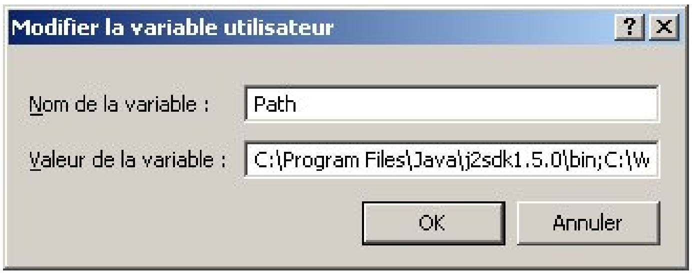
1.7.6. L'installation de la version 1.5 du JDK de Sun sous Windows

Il faut télécharger sur le site de Sun et exécuter le fichier j2sdk-1_5_0-windows-i586.exe

Un assistant guide l'utilisateur pour l'installation de l'outil.

- Sur la page « Licence Agreement », il faut lire la licence et si vous l'acceptez, cochez le bouton radio « I accept the terms in the licence agreement » et cliquez sur le bouton « Next »
- Sur la page « Custom Setup », il est possible de sélectionner/désélectionner les éléments à installer. Cliquez simplement sur le bouton « Next ».
- La page « Browser registration » permet de sélectionner les plugins des navigateurs qui seront installés. Cliquez sur le bouton « Install »
- Les fichiers sont copiés.
- La page « InstallShield Wizard Completed » s'affichage à la fin de l'installation. Cliquez sur « Finish ».

Pour faciliter l'utilisation des outils du J2SE SDK, il faut ajouter le chemin du répertoire bin contenant ces outils dans la variable Path du système.


Modifier la variable utilisateur

Nom de la variable : Path

Valeur de la variable : C:\Program Files\Java\j2sdk1.5.0\bin;C:\W

Il est aussi utile de définir la variable d'environnement JAVA_HOME avec comme valeur le chemin d'installation du SDK.

1.7.7. Installation JDK 1.4.2 sous Linux Mandrake 10

La première chose est de décompresser le fichier téléchargé sur le site de Sun en exécutant le fichier dans un shell.

Exemple :

```
[java@localhost tmp]$ sh j2sdk-1_4_2_06-linux-i586-rpm.bin
 Sun Microsystems, Inc.
 Binary Code License Agreement for the

 JAVATM 2 SOFTWARE DEVELOPMENT KIT (J2SDK), STANDARD
 EDITION, VERSION 1.4.2_X

 SUN MICROSYSTEMS, INC. ("SUN") IS WILLING TO LICENSE THE
 SOFTWARE IDENTIFIED BELOW TO YOU ONLY UPON THE CONDITION
 THAT YOU ACCEPT ALL OF THE TERMS CONTAINED IN THIS BINARY
 CODE LICENSE AGREEMENT AND SUPPLEMENTAL LICENSE TERMS
 (COLLECTIVELY "AGREEMENT"). PLEASE READ THE AGREEMENT
 ...

 Do you agree to the above license terms? [yes or no]
 yes
 Unpacking...
 Checksumming...
 0
 0
 Extracting...
 UnZipSFX 5.40 of 28 November 1998, by Info-ZIP (Zip-Bugs@lists.wku.edu).
 inflating: j2sdk-1_4_2_06-linux-i586.rpm
 Done.
 [java@localhost tmp]$
```

La décompression crée un fichier j2sdk-1_4_2_06-linux-i586.rpm. Pour installer ce package, il est nécessaire d'être root sinon son installation est impossible.

Exemple :

```
[java@localhost eclipse3]$ rpm -ivh j2sdk-1_4_2_06-linux-i586.rpm
erreur: cannot open lock file ///var/lib/rpm/RPMLOCK in exclusive mode
```

```
erreur: impossible d'ouvrir la base de données Package dans /var/lib/rpm

[java@localhost eclipse3]$ su root
Password:
[root@localhost eclipse3]# rpm -ivh j2sdk-1_4_2_06-linux-i586.rpm
Préparation... ########################################### [100%]
 1:j2sdk ########################################### [100%]
[root@localhost eclipse3]#
```

Le JDK a été installé dans le répertoire /usr/java/j2sdk1.4.2_06

Pour permettre l'utilisation par tous les utilisateurs du système, le plus simple est de créer un fichier de configuration dans le répertoire /etc/profile.d

Créez un fichier java.sh

Exemple :

```
[root@localhost root]# cat java.sh
export JAVA_HOME="/usr/java/j2sdk1.4.2_06"
export PATH=$PATH:$JAVA_HOME/bin
```

Modifiez ses droits pour permettre son exécution

Exemple :

```
[root@localhost root]# chmod 777 java.sh
[root@localhost root]# source java.sh
```

Si kaffe est déjà installé sur le système il est préférable de mettre le chemin vers le JDK en tête de la variable PATH

Exemple :

```
[root@localhost root]# java
usage: kaffe [-options] class
Options are:
 -help Print this message
 -version Print version number
 -fullversion Print verbose version info
 -ss <size> Maximum native stack size
[root@localhost root]# cat java.sh
export JAVA_HOME="/usr/java/j2sdk1.4.2_06"
export PATH=$JAVA_HOME/bin:$PATH
```

Pour rendre cette modification permanente, il faut copier le fichier java.sh dans le répertoire /etc/profile.d

Exemple :

```
[root@localhost root]# cp java.sh /etc/profile.d
```

Ainsi tous les utilisateurs qui ouvriront une nouvelle console Bash auront ces variables d'environnements positionnées pour utiliser les outils du JDK.

Exemple :

```
[java@localhost java]$ echo $JAVA_HOME
/usr/java/j2sdk1.4.2_06
[java@localhost java]$ java -version
java version "1.4.2_06"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2_06-b03)
Java HotSpot(TM) Client VM (build 1.4.2_06-b03, mixed mode)
```

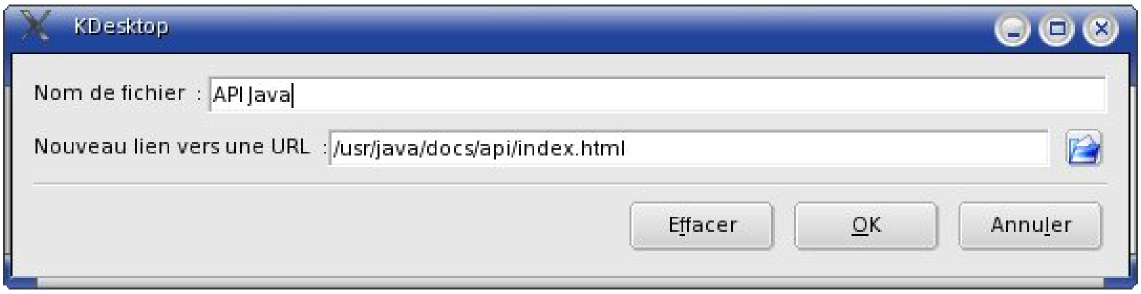
```
[java@localhost java]$
```

L'installation de la documentation se fait en décompressant l'archive dans un répertoire du système par exemple /usr/java.

Exemple :

```
[root@localhost local]# mv j2sdk-1_4_2-doc.zip /usr/java
[root@localhost java]# ll
total 33636
drwxr-xr-x 8 root root 4096 oct 16 22:18 j2sdk1.4.2_06/
-rwxr--r-- 1 root root 34397778 oct 18 23:39 j2sdk-1_4_2-doc.zip*
[root@localhost java]# unzip -q j2sdk-1_4_2-doc.zip
[root@localhost java]# ll
total 33640
drwxrwxr-x 8 root root 4096 aoû 15 2003 docs/
drwxr-xr-x 8 root root 4096 oct 16 22:18 j2sdk1.4.2_06/
-rwxr--r-- 1 root root 34397778 oct 18 23:39 j2sdk-1_4_2-doc.zip*
[root@localhost java]# rm j2sdk-1_4_2-doc.zip
rm: détruire fichier régulier `j2sdk-1_4_2-doc.zip'? o
[root@localhost java]# ll
total 8
drwxrwxr-x 8 root root 4096 aoû 15 2003 docs/
drwxr-xr-x 8 root root 4096 oct 16 22:18 j2sdk1.4.2_06/
[root@localhost java]#
```

Il est possible pour un utilisateur de créer un raccourci sur le bureau KDE en utilisant le menu contextuel créer un « nouveau/fichier/lien vers une url ... »



Nom de fichier : API Java

Nouveau lien vers une URL : /usr/java/docs/api/index.html

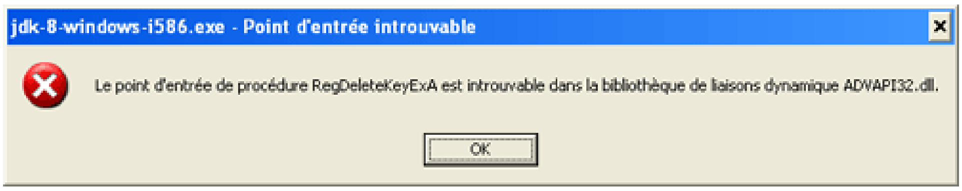
Un double-clic sur la nouvelle icône permet d'ouvrir directement Konqueror avec l'aide en ligne de l'API.

1.7.8. L'installation de la version 1.8 du JDK d'Oracle sous Windows

Télécharger et exécuter le fichier jdk-8uxxx-windows-i586.exe (ou 8uxxx est la version) à l'url https://www.oracle.com/java/technologies/downloads/.

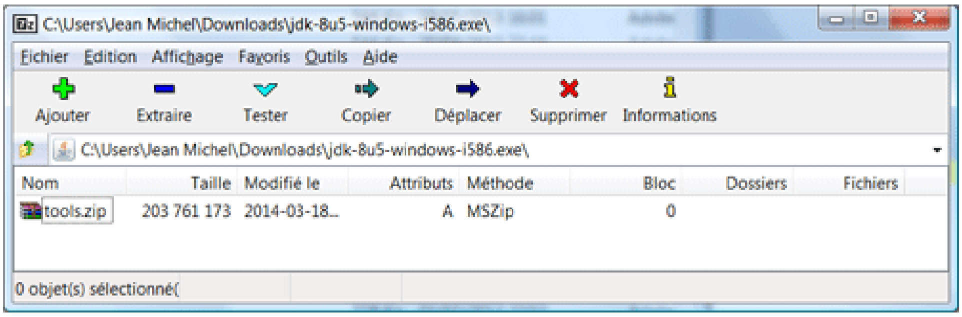
Attention : Windows XP n'est plus supporté par Java 8. Le programme d'installation échoue lors de son exécution sous Windows XP.

Java 8 peut être utilisé sous Windows XP mais le programme d'installation n'est pas utilisable sur cette version du système d'exploitation qui n'est plus supportée par Microsoft. Il est cependant possible de l'installer manuellement.



Le point d'entrée de procédure RegDeleteKeyExA est introuvable dans la bibliothèque de liaisons dynamique ADVAPI32.dll.

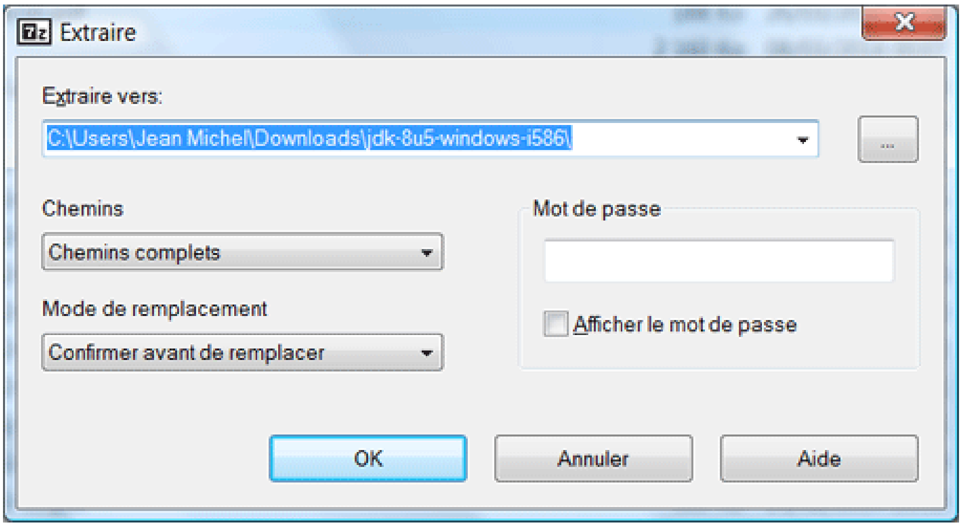
Il faut utiliser l'option 7-zip/extraire les fichiers sur le fichier jdk-8u5-windows-i586.exe


C:\Users\Jean Michel\Downloads\jdk-8u5-windows-i586.exe\

C:\Users\Jean Michel\Downloads\jdk-8u5-windows-i586.exe\

Nom	Taille	Modifié le	Attributs	Méthode	Bloc	Dossiers	Fichiers
tools.zip	203 761 173	2014-03-18...	A	MSZip	0		

0 objet(s) sélectionné(


Extraire

Extraire vers:
C:\Users\Jean Michel\Downloads\jdk-8u5-windows-i586\

Chemins
Chemins complets

Mode de remplacement
Confirmer avant de remplacer

Mot de passe

□ Afficher le mot de passe

Le répertoire décompressé jdk-8u5-windows-i586 contient le fichier tools.zip. Il suffit d'extraire les fichiers de l'archive tools.zip dans un nouveau répertoire par exemple C:\Program Files\Java\jdk1.8.0_5\.

Exemple :

```
C:\Program Files\Java\jdk1.8.0_5\jre\lib>echo %PATH%
C:\Program Files\Java\jdk1.8.0_5\bin;C:\java\apache-maven-3.1.1\bin;C:\WINDOWS\system32;
C:\WINDOWS
C:\Program Files\Java\jdk1.8.0_5\jre\lib>echo %JAVA_HOME%
C:\Program Files\Java\jdk1.8.0_5
C:\Program Files\Java\jdk1.8.0_5\jre\lib>"%JAVA_HOME%\bin\unpack200" -r charsets.pack
 charsets.jar
C:\Program Files\Java\jdk1.8.0_5\jre\lib>"%JAVA_HOME%\bin\unpack200" -r deploy.pack deploy.jar
C:\Program Files\Java\jdk1.8.0_5\jre\lib>"%JAVA_HOME%\bin\unpack200" -r jsse.pack jsse.jar
C:\Program Files\Java\jdk1.8.0_5\jre\lib>"%JAVA_HOME%\bin\unpack200" -r plugin.pack plugin.jar
C:\Program Files\Java\jdk1.8.0_5\jre\lib>"%JAVA_HOME%\bin\unpack200" -r rt.packrt.jar
C:\Program Files\Java\jdk1.8.0_5\jre\lib>java -version
 java version "1.8.0_05"
 Java(TM) SE Runtime Environment (build 1.8.0_05-b13)
 Java HotSpot(TM) Client VM (build 25.5-b02, mixed mode)
C:\Program Files\Java\jdk1.8.0_5\jre\lib>cd ext
C:\Program Files\Java\jdk1.8.0_5\jre\lib\ext>"%JAVA_HOME%\bin\unpack200" -r jfxrt.pack
 jfxrt.jar
C:\Documents and Settings\admin>cd "C:\Program Files\Java\jdk1.8.0_5\jre\lib\ext"
C:\Program Files\Java\jdk1.8.0_5\jre\lib\ext>dir *.pack
 Le volume dans le lecteur C n'a pas de nom.
 Le numéro de série du volume est 1C28-9F3C
 Répertoire de C:\Program Files\Java\jdk1.8.0_5\jre\lib\ext
 18/03/2014 03:15 4 926 375 jfxrt.pack
 18/03/2014 03:15 1 339 473 localedata.pack
 2 fichier(s) 6 265 848 octets
 0 Rép(s) 24 532 004 864 octets libres
C:\Program Files\Java\jdk1.8.0_5\jre\lib\ext>"%JAVA_HOME%\bin\unpack200" -r jfxrt.pack
 jfxrt.jar
C:\Program Files\Java\jdk1.8.0_5\jre\lib\ext>"%JAVA_HOME%\bin\unpack200" -r localedata.pack
 localedata.jar
C:\Program Files\Java\jdk1.8.0_5\jre\lib\ext>cd "C:\Program Files\Java\jdk1.8.0_5\lib"
C:\Program Files\Java\jdk1.8.0_5\lib>"%JAVA_HOME%\bin\unpack200" -r tools.pack tools.jar
```

```
C:\Program Files\Java\jdk1.8.0_5\jre\lib\ext>cd "C:\Program Files\Java\jdk1.8.0_5\lib"
C:\Program Files\Java\jdk1.8.0_5\lib>"%JAVA_HOME%\bin\unpack200" -r tools.pack tools.jar
```

Pour utiliser certains outils comme Netbeans 8.0, il est nécessaire de convertir les autres fichiers, notamment jfxrt.pack et tools.pack,en fichier jar.

Attention : cette solution installe le JDK mais n'installe pas les plugins dans les navigateurs.

1.7.9. Installation du JDK sur Ubuntu 20.10 64 bits sur Raspberry Pi 4

L'installation d'une distribution Java sur un Raspberry Pi 4 peut se faire de différente manière. Cela dépend de la distribution Java choisie, car plusieurs distribution sont disponibles sur ARM 32 et 64 bits. Le plus simple est d'installer les distributions proposées dans le gestionnaire de packages apt.

1.7.9.1. Pré-installation

Pour vérifier si Java est présent, il faut faut exécuter la commande :

Exemple :

```
jm@rpi4-ubuntu:~$ java -version
Command 'java' not found, but can be installed with:
sudo apt install default-jre              #  version 2:1.11-72, or
sudo apt install openjdk-11-jre-headless  #  version 11.0.9.1+1-0ubuntu1~20.10
sudo apt install openjdk-8-jre-headless   #  version 8u275-b01-0ubuntu1~20.10
sudo apt install openjdk-13-jre-headless  #  version 13.0.4+8-1
sudo apt install openjdk-14-jre-headless  #  version 14.0.2+12-1
sudo apt install openjdk-15-jre-headless  #  version 15+36-1
```

La première étape consiste à mettre à jour l'index des packages :

Exemple :

```
jm@rpi4-ubuntu:~$ sudo apt update
[sudo] password for jm:
```

1.7.9.2. Installer le JRE par défaut

La version par défaut au moment de l'exécution de la commande est un OpenJDK 11. Le JRE par défaut s'installe en installant le package default-jre.

Exemple :

```
jm@rpi4-ubuntu:~$ sudo apt install default-jre
Lecture des listes de paquets... Fait
Construction de l'arbre des dépendances
Lecture des informations d'état... Fait
Les paquets suivants ont été installés automatiquement et ne sont plus nécessaires :
cryptsetup-bin dctrl-tools dmeventd dmraid dpkg-repack efibootmgr gir1.2-timezonemap-1.0 gir1.
2-xkl-1.0 grub-common grub-efi-arm64 grub-efi-arm64-bin grub-efi-arm64-signed grub2-common kpa
rtx kpartx-boot libdebian-installer4 libdevmapper-event1.02.1 libdmraid1.0.0.rc16 liblvm2cmd2.
03 libreadline5 libtimezonemap-data libtimezonemap1 lvm2 os-prober python3-icu python3-pam rda
te thin-provisioning-tools
Veuillez utiliser « sudo apt autoremove » pour les supprimer.
Les paquets supplémentaires suivants seront installés :
ca-certificates-java default-jre-headless fonts-dejavu-extra java-common libatk-wrapper-java
libatk-wrapper-java-jni openjdk-11-jre openjdk-11-jre-headless
Paquets suggérés :
fonts-ipafont-gothic fonts-ipafont-mincho fonts-wqy-microhei | fonts-wqy-zenhei
```

```
Les NOUVEAUX paquets suivants seront installés :
ca-certificates-java default-jre default-jre-headless fonts-dejavu-extra java-common libatk-wr
apper-java libatk-wrapper-java-jni openjdk-11-jre openjdk-11-jre-headless
0 mis à jour, 9 nouvellement installés, 0 à enlever et 0 non mis à jour.
Il est nécessaire de prendre 39,0 Mo dans les archives.
Après cette opération, 177 Mo d'espace disque supplémentaires seront utilisés.
Souhaitez-vous continuer ? [O/n] O
Réception de :1 http://ports.ubuntu.com/ubuntu-ports groovy/main arm64 java-common all 0.72
[6?816 B]
Réception de :2 http://ports.ubuntu.com/ubuntu-ports groovy-updates/main arm64 openjdk-11-jre-
headless arm64 11.0.9.1+1-0ubuntu1~20.10 [36,9 MB]
Réception de :3 http://ports.ubuntu.com/ubuntu-ports groovy/main arm64 default-jre-headless ar
m64 2:1.11-72 [3?192 B]
...
Traitement des actions différées (« triggers ») pour fontconfig (2.13.1-2ubuntu3) ...
Traitement des actions différées (« triggers ») pour desktop-file-utils (0.24-1ubuntu4) ...
Traitement des actions différées (« triggers ») pour mime-support (3.64ubuntu1) ...
Traitement des actions différées (« triggers ») pour hicolor-icon-theme (0.17-2) ...
Traitement des actions différées (« triggers ») pour gnome-menus (3.36.0-1ubuntu1) ...
Traitement des actions différées (« triggers ») pour man-db (2.9.3-2) ...
Traitement des actions différées (« triggers ») pour ca-certificates (20201027ubuntu0.20.10.1)
Updating certificates in /etc/ssl/certs...
0 added, 0 removed; done.
Running hooks in /etc/ca-certificates/update.d...

done.
Done.
```

La vérification de la version installée se fait en executant la commande :

Exemple :

```
jm@rpi4-ubuntu:~$ java -version
openjdk version "11.0.9.1" 2020-11-04
OpenJDK Runtime Environment (build 11.0.9.1+1-Ubuntu-0ubuntu1.20.10)
OpenJDK 64-Bit Server VM (build 11.0.9.1+1-Ubuntu-0ubuntu1.20.10, mixed mode)
jm@rpi4-ubuntu:~$
```

1.7.9.3. Installer le JDK par défaut

Le JDK par défaut s'installe en installant le package default-jdk.

Exemple :

```
jm@rpi4-ubuntu:~$ sudo apt install default-jdk
Lecture des listes de paquets... Fait
Construction de l'arbre des dépendances
Lecture des informations d'état... Fait
Les paquets suivants ont été installés automatiquement et ne sont plus nécessaires :
cryptsetup-bin dctrl-tools dmeventd dmraid dpkg-repack efibootmgr gir1.2-timezonemap-1.0 gir1.
2-xkl-1.0 grub-common grub-efi-arm64 grub-efi-arm64-bin grub-efi-arm64-signed grub2-common kpa
rtx kpartx-boot libdebian-installer4 libdevmapper-event1.02.1 libdmraid1.0.0.rc16 liblvm2cmd2.
03 libreadline5 libtimezonemap-data libtimezonemap1 lvm2 os-prober python3-icu python3-pam rda
te thin-provisioning-tools
Veuillez utiliser « sudo apt autoremove » pour les supprimer.
Les paquets supplémentaires suivants seront installés :
default-jdk-headless libice-dev libpthread-stubs0-dev libsm-dev libx11-dev libxau-dev libxcb1-
dev libxdmcp-dev libxt-dev openjdk-11-jdk openjdk-11-jdk-headless x11proto-core-dev x11proto-d
ev xorg-sgml-doctools xtrans-dev
Paquets suggérés :
libice-doc libsm-doc libx11-doc libxcb-doc libxt-doc openjdk-11-demo openjdk-11-source visualvm
Les NOUVEAUX paquets suivants seront installés :
default-jdk default-jdk-headless libice-dev libpthread-stubs0-dev libsm-dev libx11-dev libxau-
dev libxcb1-dev libxdmcp-dev libxt-dev openjdk-11-jdk openjdk-11-jdk-headless x11proto-core-de
v x11proto-dev xorg-sgml-doctools xtrans-dev
0 mis à jour, 16 nouvellement installés, 0 à enlever et 0 non mis à jour.
Il est nécessaire de prendre 219 Mo dans les archives.
Après cette opération, 234 Mo d'espace disque supplémentaires seront utilisés.
Souhaitez-vous continuer ? [O/n] O
```

```
Réception de :1 http://ports.ubuntu.com/ubuntu-ports groovy-updates/main arm64 openjdk-11-jdk-
headless arm64 11.0.9.1+1-0ubuntu1~20.10 [215 MB]
Réception de :2 http://ports.ubuntu.com/ubuntu-ports groovy/main arm64 default-jdk-headless ar
m64 2:1.11-72 [1?140 B]
Réception de :3 http://ports.ubuntu.com/ubuntu-ports groovy-updates/main arm64 openjdk-11-jdk
arm64 11.0.9.1+1-0ubuntu1~20.10 [2?048 kB]
...
update-alternatives: utilisation de « /usr/lib/jvm/java-11-openjdk-arm64/bin/jar » pour fourni
r « /usr/bin/jar » (jar) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-11-openjdk-arm64/bin/jarsigner » pour
fournir « /usr/bin/jarsigner » (jarsigner) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-11-openjdk-arm64/bin/javac » pour four
nir « /usr/bin/javac » (javac) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-11-openjdk-arm64/bin/javadoc » pour fo
urnir « /usr/bin/javadoc » (javadoc) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-11-openjdk-arm64/bin/javap » pour four
nir « /usr/bin/javap » (javap) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-11-openjdk-arm64/bin/jcmd » pour fourn
ir « /usr/bin/jcmd » (jcmd) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-11-openjdk-arm64/bin/jdb » pour fourni
r « /usr/bin/jdb » (jdb) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-11-openjdk-arm64/bin/jdeprscan » pour
fournir « /usr/bin/jdeprscan » (jdeprscan) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-11-openjdk-arm64/bin/jdeps » pour four
nir « /usr/bin/jdeps » (jdeps) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-11-openjdk-arm64/bin/jimage » pour fou
rnir « /usr/bin/jimage » (jimage) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-11-openjdk-arm64/bin/jinfo » pour four
nir « /usr/bin/jinfo » (jinfo) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-11-openjdk-arm64/bin/jlink » pour four
nir « /usr/bin/jlink » (jlink) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-11-openjdk-arm64/bin/jmap » pour fourn
ir « /usr/bin/jmap » (jmap) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-11-openjdk-arm64/bin/jmod » pour fourn
ir « /usr/bin/jmod » (jmod) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-11-openjdk-arm64/bin/jps » pour fourni
r « /usr/bin/jps » (jps) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-11-openjdk-arm64/bin/jrunscript » pour
 fournir « /usr/bin/jrunscript » (jrunscript) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-11-openjdk-arm64/bin/jshell » pour fou
rnir « /usr/bin/jshell » (jshell) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-11-openjdk-arm64/bin/jstack » pour fou
rnir « /usr/bin/jstack » (jstack) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-11-openjdk-arm64/bin/jstat » pour four
nir « /usr/bin/jstat » (jstat) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-11-openjdk-arm64/bin/jstatd » pour fou
rnir « /usr/bin/jstatd » (jstatd) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-11-openjdk-arm64/bin/rmic » pour fourn
ir « /usr/bin/rmic » (rmic) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-11-openjdk-arm64/bin/serialver » pour
fournir « /usr/bin/serialver » (serialver) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-11-openjdk-arm64/bin/jaotc » pour four
nir « /usr/bin/jaotc » (jaotc) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-11-openjdk-arm64/bin/jhsdb » pour four
nir « /usr/bin/jhsdb » (jhsdb) en mode automatique
Paramétrage de libpthread-stubs0-dev:arm64 (0.4-1) ...
Paramétrage de xtrans-dev (1.4.0-1) ...
Paramétrage de default-jdk-headless (2:1.11-72) ...
Paramétrage de openjdk-11-jdk:arm64 (11.0.9.1+1-0ubuntu1~20.10) ...
update-alternatives: utilisation de « /usr/lib/jvm/java-11-openjdk-arm64/bin/jconsole » pour f
ournir « /usr/bin/jconsole » (jconsole) en mode automatique
Paramétrage de xorg-sgml-doctools (1:1.11-1) ...
Paramétrage de default-jdk (2:1.11-72) ...
Traitement des actions différées (« triggers ») pour man-db (2.9.3-2) ...
Traitement des actions différées (« triggers ») pour sgml-base (1.30) ...
Paramétrage de x11proto-dev (2020.1-1) ...
Paramétrage de libxau-dev:arm64 (1:1.0.9-0ubuntu1) ...
Paramétrage de libice-dev:arm64 (2:1.0.10-1) ...
Paramétrage de libsm-dev:arm64 (2:1.2.3-1) ...
Paramétrage de libxdmcp-dev:arm64 (1:1.1.3-0ubuntu1) ...
Paramétrage de x11proto-core-dev (2020.1-1) ...
Paramétrage de libxcb1-dev:arm64 (1.14-2) ...
Paramétrage de libx11-dev:arm64 (2:1.6.12-1) ...
Paramétrage de libxt-dev:arm64 (1:1.2.0-1) ...
```

```
jm@rpi4-ubuntu:~$
```

La vérification de la version du JDK installée se fait en executant la commande :

Exemple :

```
jm@rpi4-ubuntu:~$ javac -version
javac 11.0.9.1
jm@rpi4-ubuntu:~$
```

1.7.9.4. Installation d'une autre version

Par défaut pour une Ubuntu 20.10, le repository standard propose plusieurs versions d'OpenJDK.

Pour obtenir la liste des version disponibles, il faut faire une recherche sur openjdk dans les packages.

Exemple :

```
jm@rpi4-ubuntu:~$ apt search openjdk
Pour installer une version parmis celles de la liste, il suffit d'exécuter la commande
 pour le package de la version concernée
jm@rpi4-ubuntu:~$ sudo apt install openjdk-8-jdk
jm@rpi4-ubuntu:~$ sudo apt install openjdk-11-jdk
jm@rpi4-ubuntu:~$ sudo apt install openjdk-13-jdk
jm@rpi4-ubuntu:~$ sudo apt install openjdk-14-jdk
```

1.7.9.5. L'installation d'un JDK 8

L'installation d'un OpenJDK version 8 se fait en installant le package openjdk-8-jdk

Exemple :

```
jm@rpi4-ubuntu:~$ sudo apt-get install openjdk-8-jdk
Lecture des listes de paquets... Fait
Construction de l'arbre des dépendances
Lecture des informations d'état... Fait
Les paquets suivants ont été installés automatiquement et ne sont plus nécessaires :
cryptsetup-bin dctrl-tools dmeventd dmraid dpkg-repack efibootmgr gir1.2-timezonemap-1.0 gir1.
2-xkl-1.0 grub-common grub-efi-arm64 grub-efi-arm64-bin grub-efi-arm64-signed grub2-common kpa
rtx kpartx-boot libdebian-installer4 libdevmapper-event1.02.1 libdmraid1.0.0.rc16 liblvm2cmd2.
03 libreadline5 libtimezonemap-data libtimezonemap1 lvm2 os-prober python3-icu python3-pam rda
te thin-provisioning-tools
Veuillez utiliser « sudo apt autoremove » pour les supprimer.
Les paquets supplémentaires suivants seront installés :
openjdk-8-jdk-headless openjdk-8-jre openjdk-8-jre-headless
Paquets suggérés :
openjdk-8-demo openjdk-8-source visualvm icedtea-8-plugin fonts-ipafont-gothic fonts-ipafont-m
incho fonts-wqy-microhei fonts-wqy-zenhei
Les NOUVEAUX paquets suivants seront installés :
openjdk-8-jdk openjdk-8-jdk-headless openjdk-8-jre openjdk-8-jre-headless
0 mis à jour, 4 nouvellement installés, 0 à enlever et 6 non mis à jour.
Il est nécessaire de prendre 40,3 Mo dans les archives.
Après cette opération, 147 Mo d'espace disque supplémentaires seront utilisés.
Souhaitez-vous continuer ? [O/n] O
Réception de :1 http://ports.ubuntu.com/ubuntu-ports groovy-updates/universe arm64 openjdk-8-j
re-headless arm64 8u275-b01-0ubuntu1~20.10 [27,8 MB]
Réception de :2 http://ports.ubuntu.com/ubuntu-ports groovy-updates/universe arm64 openjdk-8-j
re arm64 8u275-b01-0ubuntu1~20.10 [65,5 kB]
Réception de :3 http://ports.ubuntu.com/ubuntu-ports groovy-updates/universe arm64 openjdk-8-j
dk-headless arm64 8u275-b01-0ubuntu1~20.10 [8?291 kB]
Réception de :4 http://ports.ubuntu.com/ubuntu-ports groovy-updates/universe arm64 openjdk-8-j
dk arm64 8u275-b01-0ubuntu1~20.10 [4?116 kB]
40,3 Mo réceptionnés en 7s (5?409 ko/s)
Sélection du paquet openjdk-8-jre-headless:arm64 précédemment désélectionné.
(Lecture de la base de données... 169868 fichiers et répertoires déjà installés.)
Préparation du dépaquetage de .../openjdk-8-jre-headless_8u275-b01-0ubuntu1~20.10_arm64.deb ...
```

```
Dépaquetage de openjdk-8-jre-headless:arm64 (8u275-b01-0ubuntu1~20.10) ...
Sélection du paquet openjdk-8-jre:arm64 précédemment désélectionné.
Préparation du dépaquetage de .../openjdk-8-jre_8u275-b01-0ubuntu1~20.10_arm64.deb ...
Dépaquetage de openjdk-8-jre:arm64 (8u275-b01-0ubuntu1~20.10) ...
Sélection du paquet openjdk-8-jdk-headless:arm64 précédemment désélectionné.
Préparation du dépaquetage de .../openjdk-8-jdk-headless_8u275-b01-0ubuntu1~20.10_arm64.deb ...
Dépaquetage de openjdk-8-jdk-headless:arm64 (8u275-b01-0ubuntu1~20.10) ...
Sélection du paquet openjdk-8-jdk:arm64 précédemment désélectionné.
Préparation du dépaquetage de .../openjdk-8-jdk_8u275-b01-0ubuntu1~20.10_arm64.deb ...
Dépaquetage de openjdk-8-jdk:arm64 (8u275-b01-0ubuntu1~20.10) ...
Paramétrage de openjdk-8-jre-headless:arm64 (8u275-b01-0ubuntu1~20.10) ...
update-alternatives: utilisation de « /usr/lib/jvm/java-8-openjdk-arm64/jre/bin/orbd » pour fo
urnir « /usr/bin/orbd » (orbd) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-8-openjdk-arm64/jre/bin/servertool » p
our fournir « /usr/bin/servertool » (servertool) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-8-openjdk-arm64/jre/bin/tnameserv » po
ur fournir « /usr/bin/tnameserv » (tnameserv) en mode automatique
Paramétrage de openjdk-8-jre:arm64 (8u275-b01-0ubuntu1~20.10) ...
update-alternatives: utilisation de « /usr/lib/jvm/java-8-openjdk-arm64/jre/bin/policytool » p
our fournir « /usr/bin/policytool » (policytool) en mode automatique
Paramétrage de openjdk-8-jdk-headless:arm64 (8u275-b01-0ubuntu1~20.10) ...
update-alternatives: utilisation de « /usr/lib/jvm/java-8-openjdk-arm64/bin/clhsdb » pour four
nir « /usr/bin/clhsdb » (clhsdb) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-8-openjdk-arm64/bin/extcheck » pour fo
urnir « /usr/bin/extcheck » (extcheck) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-8-openjdk-arm64/bin/hsdb » pour fourni
r « /usr/bin/hsdb » (hsdb) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-8-openjdk-arm64/bin/idlj » pour fourni
r « /usr/bin/idlj » (idlj) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-8-openjdk-arm64/bin/javah » pour fourn
ir « /usr/bin/javah » (javah) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-8-openjdk-arm64/bin/jhat » pour fourni
r « /usr/bin/jhat » (jhat) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-8-openjdk-arm64/bin/jsadebugd » pour f
ournir « /usr/bin/jsadebugd » (jsadebugd) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-8-openjdk-arm64/bin/native2ascii » pou
r fournir « /usr/bin/native2ascii » (native2ascii) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-8-openjdk-arm64/bin/schemagen » pour f
ournir « /usr/bin/schemagen » (schemagen) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-8-openjdk-arm64/bin/wsgen » pour fourn
ir « /usr/bin/wsgen » (wsgen) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-8-openjdk-arm64/bin/wsimport » pour fo
urnir « /usr/bin/wsimport » (wsimport) en mode automatique
update-alternatives: utilisation de « /usr/lib/jvm/java-8-openjdk-arm64/bin/xjc » pour fournir
 « /usr/bin/xjc » (xjc) en mode automatique
Paramétrage de openjdk-8-jdk:arm64 (8u275-b01-0ubuntu1~20.10) ...
update-alternatives: utilisation de « /usr/lib/jvm/java-8-openjdk-arm64/bin/appletviewer » pou
r fournir « /usr/bin/appletviewer » (appletviewer) en mode automatique
Traitement des actions différées (« triggers ») pour mime-support (3.64ubuntu1) ...
Traitement des actions différées (« triggers ») pour hicolor-icon-theme (0.17-2) ...
Traitement des actions différées (« triggers ») pour gnome-menus (3.36.0-1ubuntu1) ...
Traitement des actions différées (« triggers ») pour libc-bin (2.32-0ubuntu3) ...
Traitement des actions différées (« triggers ») pour desktop-file-utils (0.24-1ubuntu4) ...
jm@rpi4-ubuntu:~$
```

1.7.9.6. La variable d'environnement JAVA_HOME

Certaines applications recherchent la variable d'environnement JAVA_HOME pour connaître la localisation du JDK.

Exemple :

```
jm@rpi4-ubuntu:~$ export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-arm64
jm@rpi4-ubuntu:~$ echo $JAVA_HOME
/usr/lib/jvm/java-8-openjdk-arm64
jm@rpi4-ubuntu:~$
```

Il est possible de configurer la variable d'environnement JAVA_HOME avec un chemin en dur au démarrage du système en modifiant le fichier /etc/environment

Exemple :

```
jm@rpi4-ubuntu:~$ sudo nano /etc/environment
```

Il suffit de rajouter la ligne

Exemple :

```
JAVA_HOME="/usr/lib/jvm/java-11-openjdk-amd64"
Il faut enregistrer les modifications et appliquer les modifications dans la session courante
jm@rpi4-ubuntu:~$ source /etc/environment
jm@rpi4-ubuntu:~$
```

Il est possible de définir la variable en obtenant dynamiquement le chemin de la commande java. C'est pratique après avoir changer les alternatives pour les commandes du JDK ou du JRE.

Exemple :

```
jm@rpi4-ubuntu:~$ export JAVA_HOME=$(dirname $(dirname $(readlink -f $(which java))))
jm@rpi4-ubuntu:~$ echo $JAVA_HOME
/usr/lib/jvm/java-11-openjdk-arm64
```

Il est possible de définir la variable JAVA_HOME avec comme valeur un chemin en dur dans le fichier */etc/*environment pour définir la valeur par défaut.

Il est possible de définir dynamiquement dans le fichier ~/.bashrc en lui ajoutant la ligne

Exemple :

```
export JAVA_HOME=$(dirname $(dirname $(readlink -f $(which java))))
```

1.7.9.7. La configuration de la version utilisée

Plusieurs sont utilisables pour configurer la version de Java à utiliser par défaut.

1.7.9.7.1. La commande update-alternatives

Les alternatives sont une solution proposée par les systèmes reposant sur une Debian pour plusieurs programmes qui remplissent des fonctions identiques ou similaires soient listés comme des implémentations alternatives qui sont installées simultanément avec une implémentation particulière désignée comme implémentation par défaut.

Les alternatives sont gérées avec la commande update-alternatives

Elle permet de sélectionner une alternative (une version particulière) pour une commande. La commande est alors un alias vers la commande correspondante de la version souhaitée.

Exemple :

```
jm@rpi4-ubuntu:~$ java -version
openjdk version "1.8.0_275"
OpenJDK Runtime Environment (build 1.8.0_275-8u275-b01-0ubuntu1~20.10-b01)
OpenJDK 64-Bit Server VM (build 25.275-b01, mixed mode)
jm@rpi4-ubuntu:~$ javac -version
javac 1.8.0_275
jm@rpi4-ubuntu:~$ sudo update-alternatives --config java
Il existe 3 choix pour l'alternative java (qui fournit /usr/bin/java).
Sélection   Chemin                                      Priorité  ï¿½tat
------------------------------------------------------------
0            /usr/lib/jvm/java-15-openjdk-arm64/bin/java      1511      mode automatique
1            /usr/lib/jvm/java-11-openjdk-arm64/bin/java      1111      mode manuel
2            /usr/lib/jvm/java-15-openjdk-arm64/bin/java      1511      mode manuel
* 3            /usr/lib/jvm/java-8-openjdk-arm64/jre/bin/java   1081      mode manuel
```

```
Appuyez sur <Entrée> pour conserver la valeur par défaut[*] ou choisissez le numéro sélectionné
:2
update-alternatives: utilisation de « /usr/lib/jvm/java-15-openjdk-arm64/bin/java » pour fourn
ir « /usr/bin/java » (java) en mode manuel
jm@rpi4-ubuntu:~$ java -version
openjdk version "15" 2020-09-15
OpenJDK Runtime Environment (build 15+36-Ubuntu-1)
OpenJDK 64-Bit Server VM (build 15+36-Ubuntu-1, mixed mode, sharing)
jm@rpi4-ubuntu:~$ javac -version
javac 1.8.0_275
```

Il faut donc l'utiliser sur chaque commande concernée, notamment java et javac et éventuellement les autres commandes utilisée du JRE ou du JDK : javadoc, jlink, jarsigner, ...

Si une seule version de Java est installée, il n'y a rien à faire

Exemple :

```
jm@rpi4-ubuntu:~$ sudo update-alternatives --config java
Il n'existe qu'une « alternative » dans le groupe de liens java (qui fournit /usr/bin/java) :
/usr/lib/jvm/java-11-openjdk-arm64/bin/java
Rien à configurer.
jm@rpi4-ubuntu:~$
```

1.7.9.7.2. La commande update-java-alternatives

Pour obtenir les différentes installations, il faut utiliser l'option -l ou --list

Exemple :

```
jm@rpi4-ubuntu:~$ update-java-alternatives --list
java-1.11.0-openjdk-arm64      1111       /usr/lib/jvm/java-1.11.0-openjdk-arm64
java-1.15.0-openjdk-arm64      1511       /usr/lib/jvm/java-1.15.0-openjdk-arm64
java-1.8.0-openjdk-arm64       1081       /usr/lib/jvm/java-1.8.0-openjdk-arm64
jm@rpi4-ubuntu:~$ update-java-alternatives -l
java-1.11.0-openjdk-arm64      1111       /usr/lib/jvm/java-1.11.0-openjdk-arm64
java-1.15.0-openjdk-arm64      1511       /usr/lib/jvm/java-1.15.0-openjdk-arm64
java-1.8.0-openjdk-arm64       1081       /usr/lib/jvm/java-1.8.0-openjdk-arm64
```

Pour définir la version par défaut à utiliser, il faut utiliser l'option -s suivi du nom de la version souhaitée (celle qui apprarait en premier dans la liste). La commande doit être exécutée avec les privilèges root

Exemple :

```
jm@rpi4-ubuntu:~$ update-java-alternatives -s java-1.8.0-openjdk-arm64
update-java-alternatives: no root privileges
jm@rpi4-ubuntu:~$ sudo update-java-alternatives -s java-1.8.0-openjdk-arm64
[sudo] password for jm:
update-alternatives: erreur: pas d'alternatives pour mozilla-javaplugin.so
update-java-alternatives: plugin alternative does not exist: /usr/lib/jvm/java-8-openjdk-arm64
/jre/lib/aarch64/IcedTeaPlugin.so
jm@rpi4-ubuntu:~$ java -version
openjdk version "1.8.0_275"
OpenJDK Runtime Environment (build 1.8.0_275-8u275-b01-0ubuntu1~20.10-b01)
OpenJDK 64-Bit Server VM (build 25.275-b01, mixed mode)
jm@rpi4-ubuntu:~$ javac -version
javac 1.8.0_275
```

L'avantage de cette commande est de changer les alternatives pour les commandes en une seule instructions

Exemple :

```
jm@rpi4-ubuntu:~$ java -version
openjdk version "15" 2020-09-15
```

```
OpenJDK Runtime Environment (build 15+36-Ubuntu-1)
OpenJDK 64-Bit Server VM (build 15+36-Ubuntu-1, mixed mode, sharing)
jm@rpi4-ubuntu:~$ javac -version
javac 1.8.0_275
jm@rpi4-ubuntu:~$ update-java-alternatives -s java-1.11.0-openjdk-arm64
update-java-alternatives: no root privileges
jm@rpi4-ubuntu:~$ sudo update-java-alternatives -s java-1.11.0-openjdk-arm64
update-alternatives: erreur: pas d'alternatives pour mozilla-javaplugin.so
jm@rpi4-ubuntu:~$ java -version
openjdk version "11.0.9.1" 2020-11-04
OpenJDK Runtime Environment (build 11.0.9.1+1-Ubuntu-0ubuntu1.20.10)
OpenJDK 64-Bit Server VM (build 11.0.9.1+1-Ubuntu-0ubuntu1.20.10, mixed mode)
jm@rpi4-ubuntu:~$ javac -version
javac 11.0.9.1
```

1.7.9.7.3. L'utilisation d'un script pour changer de version

Il est possible de définir un script pour chaque version souhaitée qui change les alternatives et définit la variable JAVA_HOME

Exemple /opt/java/utils/java8.sh

Exemple :

```
sudo update-java-alternatives -s java-1.8.0-openjdk-arm64
export JAVA_HOME=$(dirname $(dirname $(readlink -f $(which java))))
export PATH=$PATH:$JAVA_HOME
java -version
```

Exemple /opt/java/utils/java11.sh

Exemple :

```
 sudo update-java-alternatives -s java-1.11.0-openjdk-arm64
export JAVA_HOME=$(dirname $(dirname $(readlink -f $(which java))))
export PATH=$PATH:$JAVA_HOME
java -version
```

Ces fichiers doivent être créer avec les privilèges root. Il est possible de définir des alias dans le fichier ~/.bash_aliases

Exemple :

```
alias java8='source /opt/java/utils/java8.sh'
alias java11='source /opt/java/utils/java11.sh'
```

Pour prendre en compte les modifications, il faut rédemmarer sa session ou exécuter la commande

Exemple :

```
jm@rpi4-ubuntu:~$ source ~/.bashrc
jm@rpi4-ubuntu:~$
```

Il suffit alors d'invoquer le script relatif à la version à utiliser

Exemple :

```
jm@rpi4-ubuntu:~$ java8
update-alternatives: erreur: pas d'alternatives pour mozilla-javaplugin.so
update-java-alternatives: plugin alternative does not exist: /usr/lib/jvm/java-8-openjdk-arm64
/jre/lib/aarch64/IcedTeaPlugin.so
openjdk version "1.8.0_275"
```

```
OpenJDK Runtime Environment (build 1.8.0_275-8u275-b01-0ubuntu1~20.10-b01)
OpenJDK 64-Bit Server VM (build 25.275-b01, mixed mode)
jm@rpi4-ubuntu:~$ echo $JAVA_HOME
/usr/lib/jvm/java-8-openjdk-arm64/jre
jm@rpi4-ubuntu:~$ java11
update-alternatives: erreur: pas d'alternatives pour mozilla-javaplugin.so
openjdk version "11.0.9.1" 2020-11-04
OpenJDK Runtime Environment (build 11.0.9.1+1-Ubuntu-0ubuntu1.20.10)
OpenJDK 64-Bit Server VM (build 11.0.9.1+1-Ubuntu-0ubuntu1.20.10, mixed mode)
jm@rpi4-ubuntu:~$ echo $JAVA_HOME
/usr/lib/jvm/java-11-openjdk-arm64
jm@rpi4-ubuntu:~$
```

2. Les notions et techniques de base en Java

Chapitre 2

Niveau : 🟩□□□□□ Fondamental

Ce chapitre présente quelques concepts de base utilisés en Java relatifs à la compilation et l'exécution d'applications, notamment, les notions de classpath, de packages et d'archives de déploiement jar.

Ce chapitre contient plusieurs sections :

- Les concepts de base
- L'exécution d'une applet : cette section présente l'exécution d'un programme et d'une applet.
- L'utilisation de fonctionnalités non standards
- Les builds early access

2.1. Les concepts de base

La plate-forme Java utilise quelques notions de base lors de sa mise en oeuvre, notamment :

- La compilation du code source dans un langage indépendant de la plate-forme d'exécution : le bytecode
- l'exécution du bytecode par une machine virtuelle nommée JVM (Java Virtual Machine)
- la notion de package qui permet d'organiser les classes
- le classpath qui permet de préciser au compilateur et à la JVM où elle peut trouver les classes requises par l'application
- le packaging des classes compilées dans une archive de déploiement nommée jar (Java ARchive)

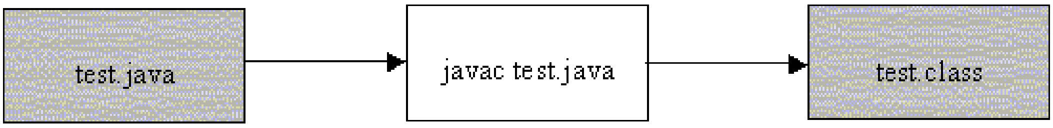
2.1.1. La compilation et l'exécution

Un programme Java est composé d'un ou plus généralement plusieurs fichiers source. N'importe quel éditeur de texte peut être utilisé pour éditer un fichier source Java.

Ces fichiers source possèdent l'extension .java. Ils peuvent contenir une ou plusieurs classes ou interfaces mais il ne peut y avoir qu'une seule classe ou interface déclarée publique par fichier. Le nom de ce fichier source doit obligatoirement correspondre à la casse près au nom de cette entité publique suivi de l'extension .java

Il est nécessaire de compiler le source pour le transformer en J-code ou bytecode Java qui sera lui exécuté par la machine virtuelle. Pour être compilé, le programme doit être enregistré au format de caractères Unicode : une conversion automatique est faite par le JDK si nécessaire.

Un compilateur Java, par exemple l'outil javac fourni avec le JDK est utilisé pour compiler chaque fichier source en fichier de classe possédant l'extension .class. Cette compilation génère pour chaque fichier source un ou plusieurs fichiers .class qui contiennent du bytecode.



Exemple :

```
public class MaClasse {

  public static void main(String[] args) {
    System.out.println("Bonjour");
  }
}
```

Résultat :

```
C:\TEMP>javac MaClasse.java

C:\TEMP>dir MaClas*
 Volume in drive C has no label.
 Volume Serial Number is 1E06-2R43

 Directory of C:\TEMP

31/07/2007  13:34               417 MaClasse.class
31/07/2007  13:34               117 MaClasse.java
```

Le compilateur génère autant de fichiers .class que de classes et interfaces définies dans chaque fichier source.

Exemple :

```
public class MaClasse {

  public static void main(String[] args) {
    System.out.println("Bonjour");
  }
}

class MonAutreClasse {

  public static void afficher(String message) {
    System.out.println(message);
  }
}
```

Résultat :

```
C:\TEMP>dir *.class
 Volume in drive C has no label.
 Volume Serial Number is 1E06-2R43

 Directory of C:\TEMP

31/07/2007  13:40               417 MaClasse.class
31/07/2007  13:40               388 MonAutreClasse.class
```

Pour exécuter une application, la classe servant de point d'entrée doit obligatoirement contenir une méthode ayant la signature public static void main(String[] args). Il est alors possible de fournir cette classe à la JVM qui va charger le ou les fichiers .class utiles à l'application et exécuter le code.

Exemple :

```
C:\TEMP>java MaClasse
Bonjour
```

Pour les classes anonymes, le compilateur génère un nom de fichier constitué du nom de la classe englobante suffixé par $ et un numéro séquentiel.

Exemple :

```
import javax.swing.JFrame;
import java.awt.event.*;

public class MonApplication {

  public static void main(String[] args) {
    MaFenetre f = new MaFenetre();
    f.afficher();
  }
}

class MaFenetre {

  JFrame mainFrame = null;

  public MaFenetre() {

    mainFrame = new JFrame();
    mainFrame.setTitle("Mon application");

    mainFrame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent ev) {
        System.exit(0);
      }
    });

    mainFrame.setSize(320, 240);
  }

  public void afficher() {
    mainFrame.setVisible(true);
  }
}
```

Résultat :

```
C:\TEMP>javac MonApplication.java

C:\TEMP>dir *.class
 Volume in drive C has no label.
 Volume Serial Number is 1E06-2R43

 Directory of C:\TEMP

31/07/2007  13:50               494 MaFenetre$1.class
31/07/2007  13:50               687 MaFenetre.class
31/07/2007  13:50               334 MonApplication.class
```

Une classe anonyme peut elle-même définir une classe : dans ce cas le nom du fichier de classe sera celui de la classe anonyme suffixé par le caractère $ et le nom de la classe

Exemple :

```
import javax.swing.JFrame;
import java.awt.event.*;

public class MonApplication {

  public static void main(String[] args) {
    MaFenetre f = new MaFenetre();
    f.afficher();
  }
}

class MaFenetre {
  JFrame mainFrame = null;

  public MaFenetre() {

    mainFrame = new JFrame();
    mainFrame.setTitle("Mon application");

    mainFrame.addWindowListener(new WindowAdapter() {

      class MonAutreClasse {

        public void afficher(String message) {
          System.out.println(message);
        }
      }

      public void windowClosing(WindowEvent ev) {
        System.exit(0);
      }
    });

    mainFrame.setSize(320, 240);
  }

  public void afficher() {
    mainFrame.setVisible(true);
  }
}
```

Résultat :

```
C:\TEMP>javac MonApplication.java

C:\TEMP>dir *.class
 Volume in drive C has no label.
 Volume Serial Number is 1E06-2R43

 Directory of C:\TEMP

31/07/2007  13:53               549 MaFenetre$1$MonAutreClasse.class
31/07/2007  13:53               555 MaFenetre$1.class
31/07/2007  13:53               687 MaFenetre.class
31/07/2007  13:53               334 MonApplication.class
```

2.1.2. Les packages

Les fichiers sources peuvent être organisés en packages. Les packages définissent une hiérarchie de noms, chaque nom étant séparé par le caractère point. Le nom d'un package est lié à une arborescence de sous-répertoires correspondant à ce nom.

Ceci permet de structurer les sources d'une application car une application peut rapidement contenir plusieurs centaines voire milliers de fichiers source. Les packages permettent aussi d'assurer l'unicité d'une classe grâce à son nom pleinement qualifié (nom du package suivi du caractère «.» suivi du nom de la classe).

L'API Java est organisée en packages répartis en trois grands ensembles :

- Packages standards : ce sont les sous-packages du package java
- Packages d'extensions : ce sont les sous-packages du package javax
- Packages tiers : ces packages concernant notamment Corba et XML

Les principaux packages standards de Java 6 sont :

java.applet	Création d'applets
java.awt	Création d'interfaces graphiques avec AWT

java.io	Accès aux flux entrants et sortants
java.lang	Classes et interfaces fondamentales
java.math	Opérations mathématiques
java.net	Accès aux réseaux
java.nio	API NIO
java.rmi	API RMI (invocation de méthodes distantes)
java.security	Mise en oeuvre de fonctionnalités concernant la sécurité
java.sql	API JDBC (accès aux bases de données)
java.util	Utilitaires (collections, internationalisation, logging, expressions régulières,...).

Les principaux packages d'extensions de Java 6 sont :

javax.crypto	Cryptographie
javax.jws	Services web
javax.management	API JMX
javax.naming	API JNDI (Accès aux annuaires)
javax.rmi	RMI-IIOP
javax.script	API Scripting
javax.security	Authentification et habilitations
javax.swing	API Swing pour le développement d'interfaces graphiques
javax.tools	API pour l'accès à certains outils comme le compilateur par exemple
javax.xml.bind	API JAXB pour la mapping objet/XML
javax.xml.soap	Création de messages SOAP
javax.xml.stream	API StAX (traitement de documents XML)
javax.xml.transform	Transformation de documents XML
javax.xml.validation	Validation de documents XML
javax.xml.ws	API JAX-WS (service web)

Les principaux packages tiers de Java 6 sont :

org.omg.CORBA	Mise en oeuvre de CORBA
org.w3c.dom	Traitement de documents XML avec DOM
org.xml.sax	Traitement de documents XML avec SAX

Le package est précisé dans le fichier source grâce à l'instruction package. Le fichier doit donc, dans ce cas, être stocké dans une arborescence de répertoires qui correspond au nom du package.

Exemple :

```
package fr.jmdoudoux.dej;

public class MaCLasseTest {

  public static void main() {
      System.out.println("Bonjour");
  }

}
```

Si les sources de l'application sont dans le répertoire C:\Documents and Settings\jm\workspace\Tests, alors le fichier MaCLasseTest.java doit être dans le répertoire C:\Documents and Settings\jm\workspace\Tests\com\jmdoudoux\test.

Si aucun package n'est précisé, alors c'est le package par défaut (correspondant au répertoire courant) qui est utilisé. Ce n'est pas une bonne pratique d'utiliser le package par défaut sauf pour des tests.

Dans le code source, pour éviter d'avoir à utiliser les noms pleinement qualifiés des classes, il est possible d'utiliser l'instruction import suivi d'un nom de package suivi d'un caractère «.» et du nom d'une classe ou du caractère «*»

Exemple :

```
import javax.swing.JFrame;
import java.awt.event.*;
```

Remarque : par défaut le package java.lang est toujours importé par le compilateur.

2.1.3. Le déploiement sous la forme d'un jar

Il est possible de créer une enveloppe qui va contenir tous les fichiers d'une application Java ou une portion de cette application dans un fichier .jar (Java archive). Ceci inclus : l'arborescence des packages, les fichiers .class, les fichiers de ressources (images, configuration, ...), ... Un fichier .jar est physiquement une archive de type Zip qui contient tous ces éléments.

L'outil jar fourni avec le jdk permet de manipuler les fichiers jar.

Exemple :

```
C:\TEMP>jar cvf MonApplication.jar *.class
manifest ajoutÚ
ajout : MaFenetre$1$MonAutreClasse.class (entrÚe = 549) (sortie = 361) (34% comp
ressÚs)
ajout : MaFenetre$1.class (entrÚe = 555) (sortie = 368) (33% compressÚs)
ajout : MaFenetre.class (entrÚe = 687) (sortie = 467) (32% compressÚs)
ajout : MonApplication.class (entrÚe = 334) (sortie = 251) (24% compressÚs)
```

Le fichier .jar peut alors être diffusé et exécuté s'il contient au moins une classe avec une méthode main().

Exemple : déplacement du jar pour être sûre qu'il n'utilise pas de classe du répertoire et exécution

```
C:\TEMP>copy MonApplication.jar ..
        1 file(s) copied.

C:\TEMP>cd ..

C:\>java -cp MonApplication.jar MonApplication
```

Remarque : un fichier .jar peut contenir plusieurs packages.

Le fichier jar peut inclure un fichier manifest qui permet de préciser des informations d'exécution sur le fichier jar (classe principale à exécuter, classpath, ...) : ceci permet d'exécuter directement l'application en double-cliquant sur le fichier .jar.

2.1.4. Le classpath

A l'exécution, la JVM et les outils du JDK recherchent les classes requises dans :

- Les classes de la plate-forme Java (stockées dans le fichier rt.jar)
- Les classes d'extension de la plate-forme Java
- Le classpath

Important : il n'est pas recommandé d'ajouter des classes ou des bibliothèques dans les sous-répertoires du JDK.

La notion de classpath est importante car elle est toujours utilisée quel que soit l'emploi qui est fait de Java (ligne de commandes, IDE, script Ant, ...). Le classpath est sûrement la notion de base qui pose le plus de problèmes aux développeurs inexpérimentés en Java mais sa compréhension est absolument nécessaire.

Le classpath permet de préciser au compilateur et à la JVM où ils peuvent trouver les classes dont ils ont besoin pour la compilation et l'exécution d'une application. C'est un ensemble de chemins vers des répertoires ou des fichiers .jar dans lequel l'environnement d'exécution Java recherche les classes (celles de l'application mais aussi celles de tiers) et éventuellement des fichiers de ressources utiles à l'exécution de l'application. Ces classes ne concernent pas celles fournies par l'environnement d'exécution incluses dans le fichier rt.jar qui est implicitement utilisé par l'environnement.

Le classpath est constitué de chemins vers des répertoires et/ou des archives sous la forme de fichiers .jar ou .zip. Chaque élément du classpath peut donc être :

- Pour des fichiers .class : le répertoire qui contient l'arborescence des sous-répertoires des packages ou les fichiers .class (si ceux-ci sont dans le package par défaut)
- Pour des fichiers .jar ou .zip : le chemin vers chacun des fichiers

Les éléments du classpath qui ne sont pas des répertoires ou des fichiers .jar ou .zip sont ignorés.

Ces chemins peuvent être absolus ou relatifs. Chaque chemin est séparé par un caractère spécifique au système d'exploitation utilisé : point-virgule sous Windows et deux-points sous Unix par exemple.

Exemple sous Windows :

```
.;C:\java\tests\bin;C:\java\lib\log4j-1.2.11.jar;"C:\Program Files\tests\tests.jar"
```

Dans cet exemple, le classpath est composé de quatre entités :

- le répertoire courant
- le répertoire C:\java\tests\bin
- le fichier C:\java\lib\log4j-1.2.11.jar
- le fichier C:\Program Files\tests\tests.jar qui est entouré par des caractères " parce qu'il y a un espace dans son chemin

Remarque : sous Windows, il est possible d'utiliser le caractère / ou \ comme séparateur d'arborescence de répertoires.

Par défaut, si aucun classpath n'est défini, le classpath est composé uniquement du répertoire courant. Une redéfinition du classpath (avec l'option -classpath ou -cp ou la variable d'environnement système CLASSPATH) inhibe cette valeur par défaut.

La recherche d'une classe se fait dans l'ordre des différents chemins du classpath : cet ordre est donc important surtout si une bibliothèque est précisée dans deux chemins. Dans ce cas, c'est le premier trouvé dans l'ordre précisé qui sera utilisé, ce qui peut être à l'origine de problèmes.

Le classpath peut être défini à plusieurs niveaux :

1. Au niveau global : il faut utiliser la variable d'environnement système CLASSPATH

Exemple sous Windows

Il faut utiliser la commande set pour définir la variable d'environnement CLASSPATH. Le séparateur entre chaque élément du classpath est le caractère point-virgule. Il ne faut pas mettre d'espace de part et d'autre du signe égal.

Exemple :

```
set CLASSPATH=C:\java\classes;C:\java\lib;C:\java\lib\mysql.jar;.
```

Sous Windows 9x : il est possible d'ajouter une ligne définissant la variable d'environnement dans le fichier autoexec.bat :

Exemple :

```
set CLASSPATH=.;c:\java\lib\mysql.jar;%CLASSPATH%
```

Sous Windows NT/2000/XP : il faut lancer l'application démarrer/paramètre/panneau de configuration/système, ouvrir l'onglet "avancé" et cliquer sur le bouton "Variables d'environnement". Il faut ajouter ou modifier la variable CLASSPATH avec comme valeur les différents éléments du classpath séparés chacun par un caractère point virgule.

Exemple sous Unix (interpréteur bash) :

Exemple :

```
CLASSPATH=.:./lib/log4j-1.2.11.jar
export CLASSPATH;
```

2. Au niveau spécifique : en utilisant l'option -classpath ou -cp du compilateur javac et de la machine virtuelle
3. Au niveau d'un script de lancement : cela permet de définir la variable d'environnement CLASSPATH uniquement pour le contexte d'exécution du script

L'utilisation de la variable système CLASSPATH est pratique car elle évite d'avoir à définir le classpath pour compiler ou exécuter mais c'est une mauvaise pratique car cela peut engendrer des problèmes :

- peut vite devenir un casse-tête lorsque le nombre d'applications augmente
- ce n'est pas portable d'une machine à une autre
- peut engendrer des conflits de versions entre applications ou entre bibliothèques

Si la JVM ou le compilateur n'arrive pas à trouver une classe dans le classpath, une exception de type java.lang.ClassNotFoundException à la compilation ou java.lang.NoClassDefFoundError à l'exécution est levée.

Exemple :

```
package fr.jmdoudoux.dej;

public class MaCLasseTest {

  public static void main() {
      System.out.println("Bonjour");
  }

}
```

Le fichier MaCLassTest.class issu de la compilation est stocké dans le répertoire C:\Documents and Settings\jm\workspace\Tests\com\jmdoudoux\test

En débutant en Java, il est fréquent de se placer dans le répertoire qui contient le fichier .class et de lancer la JVM.

Exemple :

```
C:\Documents and Settings\jmd\workspace\Tests\com\jmdoudoux\test>java MaCLasseTe
st
```

```
Exception in thread "main" java.lang.NoClassDefFoundError: MaCLasseTest (wrong n
ame: com/jmdoudoux/test/MaCLasseTest)
        at java.lang.ClassLoader.defineClass1(Native Method)
        at java.lang.ClassLoader.defineClass(Unknown Source)
        at java.security.SecureClassLoader.defineClass(Unknown Source)
        at java.net.URLClassLoader.defineClass(Unknown Source)
        at java.net.URLClassLoader.access$000(Unknown Source)
        at java.net.URLClassLoader$1.run(Unknown Source)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClassInternal(Unknown Source)
```

Cela ne fonctionne pas car la JVM cherche à partir du répertoire courant (défini dans le classpath par défaut) une classe qui soit définie dans le package par défaut (aucun nom de package précisé). Hors dans l'exemple, la classe est définie dans le package fr.jmdoudoux.dej.

Une autre erreur assez fréquente est de se déplacer dans le répertoire qui contient le premier répertoire du package

Exemple :

```
C:\Documents and Settings\jm\workspace\Tests\fr\jmdoudoux\test>cd ../../..

C:\Documents and Settings\jm\workspace\Tests>java MaCLasseTest
Exception in thread "main" java.lang.NoClassDefFoundError: MaCLasseTest
```

Dans ce cas, cela ne fonctionne pas car le nom de la classe n'est pas pleinement qualifié

Exemple :

```
C:\Documents and Settings\jmd\workspace\Tests>java fr.jmdoudoux.dej.MaCL
asseTest
Bonjour
```

En précisant le nom pleinement qualifié de la classe, l'application est exécutée.

Si le classpath est redéfini, il ne faut pas oublier d'ajouter le répertoire courant au besoin en utilisant le caractère point. Cette pratique n'est cependant pas recommandée.

Exemple :

```
C:\Documents and Settings\jmd\workspace\Tests>java -cp test.jar fr.jmdoudouxde
j.MaCLasseTest
Exception in thread "main" java.lang.NoClassDefFoundError: com/jmdoudoux/test/Ma
CLasseTest

C:\Documents and Settings\jmd\workspace\Tests>java -cp test.jar;. fr.jmdoudoux
test.MaCLasseTest
Bonjour
```

Les IDE fournissent tous des facilités pour gérer le classpath. Cependant en débutant, il est préférable d'utiliser les outils en ligne de commande pour bien comprendre le fonctionnement du classpath.

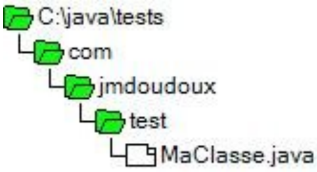
2.1.4.1. La définition du classpath pour exécuter une application

Dans cette section, une application est contenue dans le répertoire c:\java\tests. Elle est composée de la classe fr.jmdoudoux.dej.MaClasse.java.

Exemple :

```
package fr.jmdoudoux.dej;

public class MaClasse {

  public static void main(
      String[] args) {
    System.out.println("Bonjour");
  }

}
```

La structure des répertoires et fichiers de l'application est la suivante :



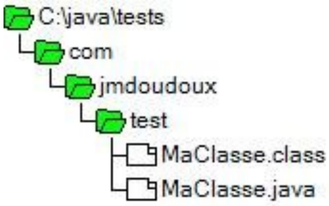
```
C:\java\tests
└com
  └jmdoudoux
    └test
      └MaClasse.java
```

Pour compiler la classe MaClasse, il faut utiliser la commande :

Exemple :

```
C:\java\tests>javac com/jmdoudoux/test/MaClasse.java
```

Le fichier MaClasse.class est créé



```
C:\java\tests
└com
  └jmdoudoux
    └test
      ├MaClasse.class
      └MaClasse.java
```

Pour exécuter la classe, il faut utiliser la commande

Exemple :

```
C:\java\tests>java fr.jmdoudoux.dej.MaClasse
Bonjour
```

Remarque : il est inutile de spécifier le classpath puisque celui-ci n'est composé que du répertoire courant qui correspond au classpath par défaut.

Il est cependant possible de le préciser explicitement

Exemple :

```
C:\java\tests>java -cp . fr.jmdoudoux.dej.MaClasse
Bonjour

C:\java\tests>java -cp c:/java/tests fr.jmdoudoux.dej.MaClasse
Bonjour
```

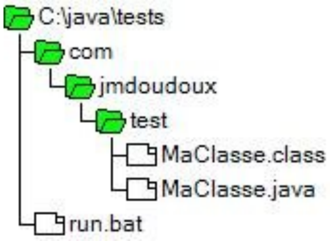
2.1.4.2. La définition du classpath pour exécuter une application avec la variable CLASSPATH

Il est possible de définir le classpath en utilisant la variable d'environnement système CLASSPATH.

Exemple : le fichier run.bat

```
@echo off
set CLASSPATH=c:/java/tests
javac com/jmdoudoux/test/MaClasse.java
java fr.jmdoudoux.dej.MaClasse
```

Ce script redéfinit la variable CLASSPATH, exécute le compilateur javac et l'interpréteur java pour exécuter la classe. Ces deux commandes utilisent la variable CLASSPATH.



Exemple :

```
C:\java\tests>run.bat
Bonjour
```

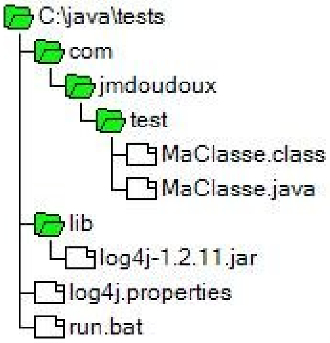
2.1.4.3. La définition du classpath pour exécuter une application utilisant une ou plusieurs bibliothèques

L'exemple de cette section va utiliser la bibliothèque log4j.

Exemple :

```
package fr.jmdoudoux.dej;

import org.apache.log4j.*;

public class MaClasse {
  static Logger logger = Logger.getLogger(MaClasse.class);

  public static void main(
      String[] args) {
    PropertyConfigurator.configure("log4j.properties");

    logger.info("Bonjour");
  }

}
```

Le fichier jar de log4j est stocké dans le sous-répertoire lib. Le fichier de configuration log4.properties est dans le répertoire principal de l'application puisqu'il est inclus dans le classpath



Il est nécessaire de préciser dans le classpath le répertoire tests et le fichier jar de log4j.

Exemple :

```
C:\java\tests>javac -cp c:/java/tests;c:/java/tests/lib/log4j-1.2.11.jar fr/jmd
oudoux/dej/MaClasse.java
C:\java\tests>java -cp c:/java/tests;c:/java/tests/lib/log4j-1.2.11.jar fr.jmdo
udoux.dej.MaClasse
[main] INFO fr.jmdoudoux.dej.MaClasse - Bonjour
```

Il est aussi possible d'utiliser la variable d'environnement système classpath.

2.1.4.4. La définition du classpath pour exécuter une application packagée en jar

Il est possible de préciser les bibliothèques requises dans le fichier manifest du fichier jar.

La propriété JAR-class-path va étendre le classpath mais uniquement pour les classes chargées à partir du jar. Les classes incluses dans le JAR-class-path sont chargées comme si elles étaient incluses dans le jar.

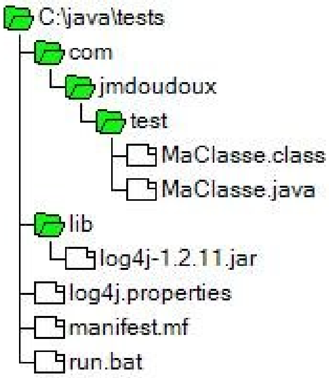
Exemple : le fichier manifest.mf

```
Main-Class: fr.jmdoudoux.dej.MaClasse
Class-Path: lib/log4j-1.2.11.jar
```

La clé Class-Path permet de définir le classpath utilisé lors de l'exécution.

Remarques importantes : Il faut obligatoirement que le fichier manifest se termine par une ligne vide. Pour préciser plusieurs entités dans le classpath, il faut les séparer par un caractère espace.

La structure des répertoires et des fichiers est la suivante :

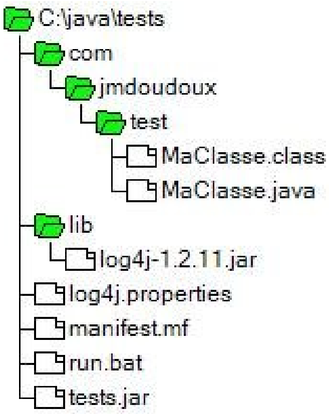


Pour créer l'archive jar, il faut utiliser l'outil jar en précisant les options de création, le nom du fichier .jar, le fichier manifest et les entités à inclure dans le fichier jar.

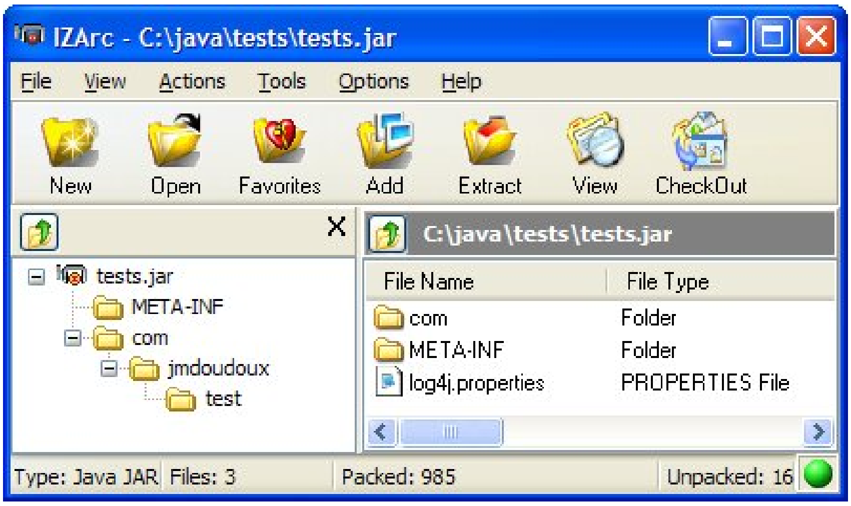
Exemple :

```
C:\java\tests>jar cfm tests.jar manifest.mf com log4j.properties
```

Le fichier jar est créé



L'archive jar ne contient pas le sous-répertoire lib, donc il n'inclut pas la bibliothèque requise.



File Name	File Type
com	Folder
META-INF	Folder
log4j.properties	PROPERTIES File

Type: Java JAR Files: 3 Packed: 985 Unpacked: 16

Pour exécuter l'application, il suffit d'utiliser l'interpréteur java avec l'option -jar

Exemple :

```
C:\java\tests>java -jar tests.jar
[main] INFO fr.jmdoudoux.dej.MaClasse - Bonjour
```

Attention : les entités précisées dans le classpath du fichier manifest doivent exister pour permettre l'exécution de l'application.

Exemple :

```
C:\java\tests>rename lib libx

C:\java\tests>java -jar tests.jar
Exception in thread "main" java.lang.NoClassDefFoundError: org/apache/log4j/Logg
er
        at fr.jmdoudoux.dej.MaClasse.<clinit>(MaClasse.java:6)
Caused by: java.lang.ClassNotFoundException: org.apache.log4j.Logger
        at java.net.URLClassLoader$1.run(Unknown Source)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClassInternal(Unknown Source)
        ... 1 more
```

2.2. L'exécution d'une applet

Il suffit de créer une page HTML pouvant être très simple :

Exemple :

```
<HTML>
<TITLE> test applet Java </TITLE>
<BODY>
<APPLET code="NomFichier.class" width="270" height="200">
</APPLET>
</BODY>
</HTML>
```

Il faut ensuite visualiser la page créée dans l'appletviewer ou dans un navigateur 32 bits compatible avec la version de Java dans laquelle l'applet est écrite.

2.3. L'utilisation de fonctionnalités non standards

Après Java 9, une nouvelle version majeure de Java est publiée tous les six mois. Ce nouveau rythme a un impact sur l'ensemble de l'écosystème et en particulier le délai pour livrer de nouvelles fonctionnalités dans le langage et dans les API. Avec une durée de six mois entre deux versions majeures du JDK, il est possible de proposer des fonctionnalités pour évaluation, de recueillir des commentaires et des retours suite à leurs utilisations, de les améliorer selon ceux qui sont fournis et finalement les rendre standard.

Ainsi certaines fonctionnalités du JDK sont proposées à la communauté pour permettre de fournir du feedback sur leurs mises en oeuvre.

Les nouvelles fonctionnalités non finales peuvent être livrées dans trois catégories :

- Preview : pour les nouvelles fonctionnalités de la plate-forme Java entièrement spécifiées et implémentées mais non encore finalisées. Introduit en Java 12
- Incubation : pour des modules concernant de nouvelles API ou outils du JDK. Introduit en Java 9
- Expérimental : pour des nouvelles fonctionnalités de la JVM Hotspot

Exemple de fonctionnalité en preview : les switch expressions

- Introduite en preview en Java 12
- Proposée pour une seconde preview en Java 13
- Promue standard et permanente en Java 14

Exemple de fonctionnalité expérimentale : le ramasse-miettes ZGC

- Introduite en expérimental en Java 11 pour Linux
- Portage sur macOS et Windows en Java 14
- Promue standard et permanent en Java 15

Des mesures de protection sont mise en oeuvre pour empêcher l'utiliser accidentellement de fonctionnalités non finales. C'est nécessaire car une fonctionnalité non définitive peut être différente lorsqu'elle devient définitive et permanente dans une version ultérieure de Java. D'autant que seules les fonctionnalités définitives et permanentes sont soumises aux règles strictes de rétrocompatibilité de Java.

Pour éviter toute utilisation involontaire, ces fonctionnalités sont désactivées par défaut :

- Les fonctionnalités en preview et expérimentales sont désactivées par défaut et la documentation du JDK avertit sans équivoque les développeurs de la nature non définitive de ces fonctionnalités et de toutes les API qui leur sont associées
- Les modules en incubation fournis par le JDK ne sont pas accessibles par défaut
- Les fonctions expérimentales sont désactivées par défaut

2.3.1. Les fonctionnalités en preview

Une fonctionnalité en preview est une nouvelle fonctionnalité du langage Java, de la machine virtuelle Java ou de l'API Java SE dont la conception, la spécification et l'implémentation sont complètes, mais qui n'est pas encore finale et permanente, ce qui signifie que la fonctionnalité peut évoluer avant d'être incluse dans la plate-forme Java de manière définitive et permanente ou même être retirée dans une version future du JDK. Les fonctionnalités en preview ne sont pas des versions bêta : elles sont publiées dans un état stable pour évaluation.

Elle est mise à disposition dans une version du JDK pour permettre aux développeurs de la tester et de l'évaluer en situation réelle afin de fournir des retours et des commentaires. Ceux-ci pourront éventuellement être pris en compte pour améliorer la fonctionnalité dans une future version du JDK. Le but d'une fonctionnalité en preview est donc de permettre aux développeurs de l'essayer et de fournir des retours et commentaires sur sa mise en oeuvre.

Ces retours d'informations et les commentaires recueillis sont évalués afin de pouvoir être éventuellement pris en compte pour apporter d'éventuels ajustements dans la release suivante en preview ou avant de devenir permanente et standard. En conséquence, la fonctionnalité peut se voir accorder le statut final et permanent (avec ou sans améliorations), ou être proposée pour une nouvelle période en preview (avec ou sans améliorations), ou même être supprimée.

Ces fonctionnalités sont publiées pour expérimentation, mais protégées contre une utilisation accidentelle avec des erreurs à la compilation et à l'exécution sans activation explicite des fonctionnalités en preview.

A partir de Java 12, chaque version de Java propose une ou plusieurs fonctionnalités en preview. La première fonctionnalité introduite en preview concerne les switch expressions livrées en preview en Java 12 (JEP 325) et Java 13 (JEP 354) avant de devenir standard en Java 14 (JEP 361).

Plusieurs projets livrent des fonctionnalités en preview notamment le projet Amber qui concerne des évolutions dans la syntaxe du langage Java. Il livre ses fonctionnalités au fur et à mesure, la plupart en preview avec plusieurs releases en preview, généralement au moins 2 (switch expressions, blocs de texte, records, pattern matching pour l'instruction instanceof, les classes scellées, ...).

Il est important de se rappeler qu'une fonctionnalité en preview sera dans la version suivante de Java :

- Soit proposée pour une nouvelle période en preview avec ou sans modification
- Soit promue comme fonctionnalité finale et permanente avec ou sans modification
- Soit supprimée purement et simplement

2.3.1.1. La définition de fonctionnalités en preview

Il n'est pas obligatoire ni nécessaire que toutes les nouvelles fonctionnalités du langage, de la JVM et des API soient disponibles initialement en tant que fonctionnalités en preview.

Le rôle et le cycle de vie des fonctionnalités en preview sont définis dans la JEP 12.

La JEP 12 a été introduite dans le JDK 12 pour proposer initialement deux types de fonctionnalités en preview dans la plateforme Java SE :

- Les fonctionnalités du langage en preview
- Les fonctionnalités de la JVM en preview

Dans le JDK 13, la JEP 12 a été améliorée pour reconnaître que les fonctionnalités du langage en preview sont parfois co-développées avec de nouvelles API. Cela a conduit à une taxonomie des API co-développées : "essential", "reflective" et "convenient".

Dans le JDK 15, la JEP 12 a été améliorée pour permettre un troisième type de fonctionnalité en preview dans la plate-forme Java SE : les API en preview. Les API en preview englobent l'idée d'API co-développées avec les fonctionnalités du langage en preview et permettent en outre la définition d'API Java SE non liées à d'autres fonctionnalités en preview. Cela a conduit à une taxonomie d'API en preview "essential", "reflective", "convenient" et "standalone".

Une fonctionnalité en preview peut donc être :

- Soit une nouvelle fonctionnalité du langage Java (preview language feature)
- Soit une nouvelle fonctionnalité de la JVM (preview VM feature)
- Soit un nouveau module, package, classe, interface, enum, record, méthode, constructeur, champ dans les packages java.* ou javax.* (preview API)

La conception, la spécification et l'implémentation sont achevées lors de leur diffusion. Elles permettent de proposer une période d'exposition et d'évaluation à grande échelle avant d'obtenir un statut définitif et permanent dans la plate-forme Java SE ou d'être affinées selon les retours obtenus ou même supprimées.

Les fonctionnalités en preview permettent de bénéficier d'une période d'exposition après que leurs spécifications et leurs implémentations soient stables. Elles permettent aux développeurs de les expérimenter et de fournir des retours et commentaires avant qu'elles n'atteignent un statut final et permanent dans la plate-forme Java SE.

Au final, la fonctionnalité se verra accorder un statut final et permanent (avec ou sans améliorations) ou sera supprimée.

Une fonctionnalité en preview doit respecter plusieurs critères :

- Ne pas être expérimentale : une fonction en preview ne doit pas être expérimentale, risquée, incomplète ou instable
- Etre de bonne qualité : une fonctionnalité en preview doit présenter le même niveau de qualité qu'une fonctionnalité finale et permanente de la plate-forme Java SE

Chaque fonctionnalité en preview est décrite dans une JDK Enhancement Proposal (JEP) qui définit sa portée et fournit sa description. Généralement, les JEP qui concernent une fonctionnalité en preview contiennent dans leur titre le nom de la fonctionnalité et « (Preview) / (Second Preview) / (Third Preview) / ...».

Toutes les fonctionnalités en preview doivent être désactivées par défaut dans une implémentation de Java SE et une implémentation doit offrir un mécanisme pour activer toutes les fonctionnalités en preview. Une implémentation ne doit pas permettre d'activer individuellement les fonctionnalités en preview, puisque toutes les fonctionnalités en preview ont le même statut dans la plate-forme Java SE.

Il n'y a pas de contraintes sur la forme d'une fonctionnalité en preview :

- Une fonctionnalité du langage en preview peut ajouter des formes de déclarations, d'instructions, d'expressions et de littéraux à la syntaxe du langage Java. Elle peut modifier la sémantique statique (typage) des déclarations, instructions, expressions et littéraux préexistants ; et elle peut modifier la sémantique dynamique (exécution) des instructions et expressions préexistants
- Une API en preview peut ajouter des membres publics aux classes et interfaces des packages java.* et javax.*, ajouter des classes et interfaces publiques aux packages java.* et javax.*, ajouter et exporter des package dans les espaces de nommage java.* et javax.* et ajouter des modules dans l'espace de nommage java.*. Une API en preview peut modifier les spécifications narratives (mais pas les signatures) des méthodes, champs, classes, interfaces, packages et modules existants. Une API en preview réside généralement dans le module java.base, mais elle peut également ou exclusivement résider dans d'autres modules java.*, y compris ceux introduits uniquement pour l'API en preview. Une API en preview peut concerner des API non Java, des protocoles ou des API des packages com.sun.*
- Une fonctionnalité de la VM en preview peut ajouter et modifier des éléments du format de fichier .class, modifier les règles de chargement, de chaînage et d'initialisation des classes et étendre et modifier le jeu d'instructions (opcodes) du bytecode. Les fonctionnalités de la VM en preview sont différentes des fonctionnalités de bas niveau de l'implémentation de la JVM de HotSpot qui sont configurées via java -X ou java -XX. Les fonctionnalités de la VM en preview sont également différentes des fonctionnalités "expérimentales" de la JVM HotSpot, qui sont les premières versions des fonctionnalités de bas niveau qui doivent être explicitement débloquées dans HotSpot au moment de l'exécution en utilisant l'option -XX:+UnlockExperimentalVMOptions

En Java 12 et 13, les API en preview sont marquées avec le mécanisme de dépréciation standard. Par conséquent, les API en preview ont été dépréciées à la naissance en utilisant l'annotation @Deprecated(forRemoval=true, since=...) lors de leur introduction. Cependant, cette approche basée sur la dépréciation a finalement été abandonnée car il était déroutant de voir un élément d'API introduit dans la même version de Java SE qu'il a été déprécié. Par exemple voir @since 13 et @Deprecated(forRemoval=true, since="13") sur le même élément d'une API en preview.

A partir de Java 14, les fonctionnalités en preview dans le JDK sont annotées avec @jdk.internal.javac.PreviewFeature. Elle indique que l'élément annoté est associé à une fonctionnalité en preview. Celle-ci est précisée via l'attribut feature de type jdk.internal.javac.PreviewFeature.Feature qui est une énumération dont les valeurs évoluent à chaque version du JDK.

Cette annotation interne est exploitée dans le compilateur javac notamment pour afficher un avertissement lors de l'utilisation d'une telle fonctionnalité.

Exemple :

```
C:\java>javac --enable-preview -source 14 -Xlint:preview HelloBlocTexte.java
HelloBlocTexte.java:5: warning: [preview] text blocks are a preview feature and may be
 removed in a future release.
    System.out.println("""
                       ^
1 warning
```

Les éléments d'une API en preview doivent utiliser une balise @since dans le commentaire javadoc de l'élément, indiquant la version à laquelle l'annotation @preview a été ajoutée pour la première fois. Si l'élément de l'API est finalement rendu définitive et permanente dans Java SE version N, la balise @since doit être modifiée pour indiquer la version N.

2.3.1.2. Les APIs en relation avec une fonctionnalité en preview

La plupart des API en preview seront autonomes, sans lien avec les fonctionnalités du langage ou de la VM en preview.

Il arrive parfois qu'une API soit liée à une fonctionnalité en preview. Certaines API en preview peuvent donc être codéveloppées avec des fonctionnalités du langage ou de la VM en preview. Il existe trois types d'API en preview codéveloppées :

- Les API essentielles (Essential APIs) sont celles qui sont nécessaires pour que la fonctionnalité en preview soit utilisable. Une telle API se trouve dans un package java.*. Par exemple, l'instruction for améliorée utilise java.lang.Iterable et l'instruction try-with-resources utilise java.lang.AutoCloseable. Initalement, elles étaient annotées avec @Deprecated(forRemoval=true) et leur documentation souligne qu'elles n'existent que grâce à une fonctionnalité en preview, qu'elles peuvent changer avec elle et qu'elles ne doivent être utilisées qu'en conjonction avec elle. Si la fonctionnalité est finalisée, la dépréciation et la partie "preview" de la documentation sont supprimées. Si la fonctionnalité est supprimée, l'API le sera également.
- Les API réflectives (Reflective APIs) sont celles qui exposent des fonctionnalités du langage ou de la VM en preview dans l'API Reflection, l'API Method Handle, l'API Language Model, l'API Annotation Processing, l'API Compiler ou l'API Compiler Tree. Une API en preview réflective se trouve dans les packages java.* ou javax.* ou com.sun.source.tree.*. Leurs annotations, leur documentation et leur cycle de vie sont les mêmes que pour les API essentielles.
- Les API pratiques (Convenient APIs) sont celles qui sont en lien avec une fonctionnalité en preview, mais ne sont pas essentielles pour celle-ci et peuvent être utilisées sans elle. Elles ne sont ni annotées ni documentées de manière spécifique et leur cycle de vie n'est pas lié à la fonctionnalité.

Une API en preview réflective doit annotée sa déclaration avec @jdk.internal.javac.PreviewFeature(..., reflective=true). Cela entraîne l'ajout par l'outil javadoc d'un message différent à l'élément que lorsque reflective=false.

La JSR d'un JDK fournit une liste définitive des éléments d'API en preview de la version.

La javadoc propose une page nommée preview-list qui énumère tous les éléments du JDK proposés en preview.

2.3.1.3. L'utilisation de fonctionnalités en preview

Comme les fonctionnalités en preview n'ont pas atteint un statut final et permanent dans la plate-forme Java SE, elles ne sont pas disponibles par défaut ni à la compilation ni à l'exécution. Pour pouvoir les utiliser, il faut explicitement les activer à la compilation et à l'exécution pour s'assurer qu'elles sont utilisées en âme et conscience.

Les fonctionnalités en preview sont spécifiques à une version particulière de Java SE. Elles nécessitent l'utilisation d'options spéciales à la compilation et à l'exécution.

Important : il faut obligatoirement que le compilateur et la JVM utilisée pour du code utilisant des fonctionnalités en preview soit de la même version du JDK.

Il est important de ne pas oublier que les fonctionnalités en preview sont susceptibles d'être modifiées et qu'elles sont essentiellement destinées à être testées pour fournir des retours.

Il ne faut pas distribuer d'artefacts qui nécessitent des fonctionnalités non finales : il ne faut pas diffuser de code compilé qui utilise des fonctionnalités en preview. Ces fonctionnalités ne sont compatibles qu'avec la version de Java pour laquelle elles ont été compilées.

2.3.1.3.1. L'activation des fonctionnalités en preview

Les fonctionnalités en preview sont désactivées par défaut : pour utiliser des fonctionnalités en preview, il faut explicitement les activer. L'activation des fonctionnalités en preview se fait en utilisant l'option --enable-preview lors de l'utilisation des commandes java, javac, javadoc et shell du JDK. Il faut obligatoirement utiliser l'option --enable-preview avec ces outils du JDK pour leur permettre d'utiliser des fonctionnalités en preview.

Les fonctionnalités en preview ne peuvent pas être activées individuellement : l'option --enable-preview active toutes les fonctionnalités en preview pour le JDK utilisé.

Avec Maven, il faut ajouter des options aux plug-ins Compiler, Surefire et Failsafe.

Exemple :

```
<plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
               <release>13</release>
               <compilerArgs>
                       --enable-preview
               </compilerArgs>
        </configuration>
</plugin>
<plugin>
        <artifactId>maven-surefire-plugin</artifactId>
        <configuration>
               <argLine>--enable-preview</argLine>
        </configuration>
</plugin>
<plugin>
        <artifactId>maven-failsafe-plugin</artifactId>
        <configuration>
               <argLine>--enable-preview</argLine>
        </configuration>
</plugin>
```

Avec Gradle, il faut ajouter des options au compilateur et à l'exécution des tests.

Exemple :

```
compileJava {
  options.compilerArgs += ["--enable-preview"]
}

test {
  jvmArgs '--enable-preview'
}
```

Avec IntelliJ IDEA, il faut aller dans « Project Settings / Project » et utiliser la liste déroulante de « Language level » pour sélectionner la version du code source du projet. Pour utiliser une version dans laquelle les fonctionnalités sont utilisables, il faut sélectionner la version suivie de « (Preview) »

Avec Eclipse, dans « Java compiler » des propriétés du projet, il faut cocher « Enable preview features ».

2.3.1.3.2. L'utilisation de fonctionnalités en preview lors de la compilation

Lors de la compilation avec les fonctionnalités en preview désactivées, une erreur de compilation est émise si le code utilise une fonctionnalité du langage ou une API en preview. Une erreur est justifiée parce qu'une version ultérieure pourrait supprimer la fonctionnalité ou modifier son comportement de manière incompatible.

Exemple :

```
C:\java>javac MonApp.java
MonApp.java:5: error: text blocks are a preview feature and are disabled by default.
    String message = """
                     ^
  (use --enable-preview to enable text blocks)
1 error
```

Pour compiler du code source avec javac qui utilise les fonctionnalités en preview de la version N du JDK, il faut utiliser la commande javac de la version N du JDK avec l'option de ligne de commande --enable-preview en conjonction avec l'option de ligne de commande --release N ou -source N.

Lors de la compilation du code source, le compilateur considère par défaut que la version du code source est la même que la sienne. Par exemple, en compilant avec un compilateur javac d'un jdk 11, il permettra d'utiliser les fonctionnalités du langage en Java 11.

Il est possible de définir une version du code source différente de celle du compilateur en utilisant les options -source ou --release. L'option --release introduite en Java 9, combine en une seule option -source, -target et -bootclasspath.

L'utilisation des options -source ou --release est facultative pour compiler du code n'utilisant pas de fonctionnalité en preview mais l'une ou l'autre est obligatoire pour compiler du code utilisant des fonctionnalités en preview.

Exemple :

```
C:\java>javac --enable-preview TestTextBlock.java
error: --enable-preview must be used with either -source or --release
```

La version précisée doit obligatoirement correspondre à la version du JDK contenant le compilateur utilisé.

Exemple :

```
C:\java>javac -version
javac 13

C:\java>javac --enable-preview --release 12 TestTextBlock.java
error: invalid source release 12 with --enable-preview
  (preview language features are only supported for release 13)
```

Cela oblige le développeur à indiquer explicitement que les fonctionnalités en preview utilisées dans le code correspondent à la version indiquée et cela force le compilateur à vérifier. La signification de l'option --enable-preview change d'un JDK à l'autre. Demander au développeur d'indiquer un numéro de version concret avec --release ou -source, c'est s'attendre à ce que le code qui s'appuie sur les fonctionnalités en preview du JDK N soient liés à cette version et ne puissent pas être compilées avec un JDK N+1.

Seul un compilateur du JDK version N peut prendre en charge les fonctionnalités en preview définies par Java SE version N. On ne peut pas s'attendre à ce qu'un compilateur dans le JDK version N+1 prenne en charge les fonctionnalités en preview définies par Java SE version N, car elles peuvent avoir été modifiées ou abandonnées depuis Java SE version N.

Le compilateur javac d'un JDK version N ne compile pas du code source qui utilise des fonctionnalités du langage en preview ou des API en preview d'autres versions de Java SE, ou pour compiler du code source par rapport à des fichiers .class qui dépendent de fonctionnalités de la VM en preview d'autres versions de Java SE.

Si le code se compile correctement, le compilateur javac d'un JDK N affiche un message s'il détecte l'utilisation d'une fonctionnalité du langage en preview de Java SE N dans le code source. Ce message ne peut pas être désactivé en utilisant @SuppressWarnings dans le code source, parce que les développeurs doivent être conscients de leur dépendance à une fonctionnalité du langage en preview de Java SE N. Cette fonctionnalité peut être changée ou être retirée dans la version N+1 de Java SE.

Lors de la compilation avec les fonctionnalités en preview activées, la JLS à partir de Java 15 recommande fortement que le compilateur affiche un message d'avertissement. Le compilateur javac le propose sous la forme d'une note.

Exemple :

```
C:\java>javac --enable-preview --release 19 MonApp.java
Note: HelloApp.java uses preview features of Java SE 19.
Note: Recompile with -Xlint:preview for details.
```

Comme indiqué dans la JEP 12, il n'est pas possible de supprimer les messages "use of preview features" du compilateur. Ce message affiché par le compilateur n'est pas un avertissement : il ne peut donc pas être désactivé avec @SuppressWarnings("...").

Lors de la compilation avec des fonctionnalités en preview activées, toute référence dans le code source à un élément d'une API essentielle associé à une fonctionnalité en preview doit émettre un avertissement.

Exemple :

```
C:\java>javac --enable-preview -source 14 -Xlint:preview MonApp.java
MonApp.java:5: warning: [preview] text blocks are a preview feature and may be
 removed in a future release.
    String message = """
                     ^
1 warning
```

Cet avertissement peut être supprimé avec l'annotation @SuppressWarnings("preview"), contrairement à la note affichée par le compilateur javac lorsqu'il détecte l'utilisation d'une fonctionnalité en preview dans le code source.

Lors de la compilation, si le code utilise une fonctionnalité du langage en preview ou une API en preview alors le compilateur ajoute des informations dans le fichier .class généré : la major_version correspond à la version du byte code pour le JDK utilisé et la minor_version dont les 16 bits sont définis avec la valeur 65535.

2.3.1.3.3. L'utilisation de fonctionnalités en preview à l'exécution

A l'exécution, il faut aussi utiliser l'option --enable-preview de la JVM pour exécuter du bytecode qui contient des fonctionnalités en preview. Pour exécuter une application qui utilise des fonctionnalités en preview de la version N du JDK, il faut utiliser la commande java du JDK N avec l'option --enable-preview.

Exemple :

```
C:\java>java --enable-preview TestTextBlock
```

Si ce n'est pas le cas, la JVM s'arrête à avec une exception

Exemple :

```
C:\java>java TestTextBlock
Error: LinkageError occurred while loading main class TestTextBlock
        java.lang.UnsupportedClassVersionError: Preview features are not enabled for
 TestTextBlock (class file version 57.65535). Try running with '--enable-preview'
```

Il en est de même lors de l'exécution directe d'un unique fichier .java à la JVM depuis Java 11. Si le code utilise une fonctionnalité en preview sans l'option --enable-preview.

Exemple :

```
C:\java>java TestTextBlock.java
TestTextBlock.java:5: error: text blocks are a preview feature and are disabled by default.
    String message = """
                     ^
  (use --enable-preview to enable text blocks)
1 error
error: compilation failed
```

L'option --enable-preview de la JVM permet d'activer les fonctionnalités en preview de la version de Java correspond au JDK utilisé. Cela permet l'exécution de tout fichier .class qui dépend de ces fonctionnalités en preview, soit parce qu'il repose directement sur les fonctionnalités de la VM en preview, soit parce qu'il a été compilé à partir d'un code source qui utilise les fonctionnalités du langage en preview ou des API en preview.

La JVM ne permet pas de charger une classe utilisant une fonctionnalité en preview compilée avec une version différente du JDK de la JVM.

L'option -Xlog:class+preview de la JVM permet d'afficher les classes chargées qui dépendent de fonctionnalités en preview.

Exemple :

```
C:\java>java --enable-preview -Xlog:class+preview HelloForeign
[0.073s][info][class,preview] Loading class HelloForeign that depends on
 preview features (class file version 63.65535)
```

2.3.1.3.4. La version doit être la même pour le compilateur et la JVM

Lors de l'utilisation de fonctionnalités en preview, il faut utiliser une JVM de la même version majeure de Java que celle du compilateur utilisé.

Exemple :

```
C:\java>javac -version
javac 13

C:\java>javac --enable-preview --release 12 MonApp.java
error: invalid source release 12 with --enable-preview
  (preview language features are only supported for release 13)
```

Il n'est pas possible d'exécuter du bytecode utilisant des fonctionnalités en preview compilé avec une version antérieure. Dans l'exemple ci-dessous, le code est compilé avec un JDK 13 et exécuté avec un JDK 14 sans recompilation. Comme le code utilise des fonctionnalités en preview, une erreur est émise à l'exécution.

Exemple :

```
C:\java>java -version
openjdk version "14" 2020-03-17
OpenJDK Runtime Environment AdoptOpenJDK (build 14+36)
OpenJDK 64-Bit Server VM AdoptOpenJDK (build 14+36, mixed mode, sharing)

C:\java>java --enable-preview MonApp
Erreur : LinkageError lors du chargement de la classe principale MonApp
```

```
        java.lang.UnsupportedClassVersionError: MonApp (class file version 57.65535)
 was compiled with preview features that are unsupported. This version of the Java
 Runtime only recognizes preview features for class file version 58.65535
```

Cela permet d'éviter que des fonctionnalités en preview obsolètes soit utilisées avec une version de Java différente. Lors de la mise à jour à version majeure suivante de Java, il faut incrémenter la valeur de l'option --release ou -source et selon les évolutions de la fonctionnalité en preview :

- Si la fonctionnalité est finalisée sans changement alors il suffit de retirer --enable-preview
- Si la fonctionnalité est finalisée avec des changements alors il faut adapter le code et retirer --enable-preview
- Si la fonctionnalité est toujours en preview sans changement alors il n'y a rien à faire
- Si la fonctionnalité est toujours en preview avec des changements alors il faut adapter le code

Ainsi les fonctionnalités en preview obligent à s'assurer que le code est toujours à jour avec la version de Java utilisée.

2.3.1.3.5. Un exemple d'utilisation

Exemple avec une classe qui utilise un bloc de texte introduit en Java 13 en tant que fonctionnalité en preview.

Exemple (code Java 13) :

```
class HelloBlocTexte {
  public static void main(String[] args) {
    System.out.println("""
           Hello World""");
  }
}
```

Le code se compile avec un compilateur javac d'un JDK 13 avec les options requises dont un niveau de compatibilité du code source fixé à 13 et peut être exécuté qu'avec une JVM d'un JDK 13.

Résultat :

```
C:\java>javac -version
javac 13

C:\java>javac --enable-preview -source 13 HelloBlocTexte.java
Note: HelloBlocTexte.java uses preview language features.
Note: Recompile with -Xlint:preview for details.

C:\java>java --enable-preview HelloBlocTexte
Hello World
```

Ce code compilé en Java 13 avec des fonctionnalités en preview ne peut pas être exécuté avec une JVM d'un JDK 14. Une telle exécution lève une exception de type LinkageError.

Exemple :

```
C:\java>java -version
openjdk version "14" 2020-03-17
OpenJDK Runtime Environment AdoptOpenJDK (build 14+36)
OpenJDK 64-Bit Server VM AdoptOpenJDK (build 14+36, mixed mode, sharing)

C:\java>java --enable-preview HelloBlocTexte
Erreur : LinkageError lors du chargement de la classe principale HelloBlocTexte
        java.lang.UnsupportedClassVersionError: HelloBlocTexte (class file version 57.65535)
 was compiled with preview features that are unsupported. This version of the Java Runtime
 only recognizes preview features for class file version 58.65535
```

Il faut recompiler le code avec un compilateur javac d'un JDK 14. Une erreur de compilation est émise lors de la compilation de code utilisant une fonctionnalité en preview avec un niveau de compatibilité du code source différent de celui du JDK utilisé.

Exemple :

```
C:\java>javac --enable-preview -source 13 HelloBlocTexte.java
error: invalid source release 13 with --enable-preview
  (preview language features are only supported for release 14)
```

Il faut recompiler le code avec un compilateur javac du JDK 14 et un niveau de compatibilité du code source fixé à 14.

Exemple :

```
C:\java>javac --enable-preview -source 14 HelloBlocTexte.java
Note: HelloBlocTexte.java uses preview language features.
Note: Recompile with -Xlint:preview for details.

C:\java>java --enable-preview HelloBlocTexte
Hello World
```

2.3.2. Les modules en incubation (incubator modules)

La JEP 11 introduit la notion de modules en incubation pour permettre l'inclusion d'API et d'outils du JDK qui pourraient dans le futur, après améliorations et stabilisations, être inclus et pris en charge de manière standard dans la plate-forme Java SE ou dans le JDK.

Le premier module proposé en incubation a été l'API Client HTTP en Java 9 (JEP 110) et 10. Le module est devenu standard en Java 11 (JEP 321).

Java 14 a introduit en incubation l'outil de packaging jpackage (JEP 343) et l'API Foreign Memory Accesss (JEP 370)

Java 14 a introduit en incubation les API Vector (JEP 338) et Foreign Linker (JEP 389)

Java 17 a introduit en incubation l'API Foreign Function and Memory (JEP 412) qui est la fusion des API Foreign Linker et Foreign Memory Access.

Java 19 a introduit en incubation l'API Structured concurrency (JEP 428)

Module/version du JDK	9	10	14	15	16	17	18
jdk.incubator.httpclient	X	X					
jdk.incubator.jpackage			X	X			
jdk.incubator.foreign			X	X	X	X	X

Module/version du JDK	16	17	18	19	20	21	
jdk.incubator.vector	X	X	X	X	X	X	
jdk.incubator.concurrent				X	X	X	

Les modules en incubation sont protégés contre une utilisation accidentelle. Ils sont dans l'espace de nommage jdk.incubator. Ces modules fournis par le JDK ne sont pas accessibles par défaut. Il faut implicitement les ajouter en utilisant l'option --add-modules pour permettre leur résolution par une application même si elle n'utilise que le classpath.

Pour une application qui utilise le module path, il faut ajouter en tant que dépendances les modules en incubation.

2.3.3. Les fonctionnalités expérimentales

Une fonctionnalité expérimentale permet de proposer de nouvelles fonctionnalités non triviales dans la JVM Hotspot. Il n'y a pas de JEP qui décrivent comment sont mises en oeuvre les fonctionnalités expérimentales : les fonctionnalités expérimentales sont plus une convention qu'un processus formel.

Un exemple de fonctionnalités expérimentales est le ramasse-miettes ZGC, introduit en Java 11 uniquement sur Linux x64 (JEP 333). Après un portage sous Windows et Mac OSX et plusieurs améliorations, ZGC a été promu standard en Java 15 (JEP 377).

Les fonctions expérimentales sont des fonctionnalités de la JVM Hotspot qui sont désactivées par défaut. Pour pouvoir utiliser de telles fonctionnalités, il faut utiliser l'option -XX:+UnlockExperimentalVMOptions pour autoriser les fonctions expérimentales. Il est alors possible d'utiliser l'option de la fonctionnalité expérimentale.

2.4. Les builds early access

Pour certaines fonctionnalités très impactantes de certains projets d'OpenJDK de longues durées, des builds early access (EA) sont proposés. Ces projets mènent des recherches et des investigations dans des domaines particuliers afin d'améliorer certains aspects de la plate-forme Java.

Ils peuvent être livrés de manière itérative et incrémentale pour proposer différents prototypes et expérimentations de solutions potentielles qui pourront être améliorées ou abandonnées.

Le public cible de ces builds EA est composé d'utilisateurs avertis souhaitant tester les fonctionnalités proposées et fournir des retours et des commentaires sur leurs expérimentations. Ces builds ne sont soumis à aucune règle de compatibilité.

Une fois qu'une fonctionnalité donnée a atteint le niveau de stabilité et de qualité attendu, elle peut alors être proposée selon les mécanismes habituels, tels qu'une JEP.

Les builds early access du JDK sont téléchargeables à l'url : jdk.java.net.

3. La syntaxe et les éléments de bases de Java

Chapitre 3

Niveau : 🟩⬜⬜⬜⬜⬜ Fondamental

Ce chapitre détaille la syntaxe et les éléments de bases du langage Java. Cette syntaxe est largement inspirée de celle du langage C.

Ce chapitre contient plusieurs sections :

- Les règles de base : présente les règles syntaxiques de base de Java.
- Les mots réservés du langage Java : donne la liste des mots réservés du langage Java.
- Les identifiants : présente les règles de composition des identificateurs.
- Les commentaires : présente les différentes formes de commentaires de Java.
- La déclaration et l'utilisation de variables : présente la déclaration des variables, les types élémentaires, les formats des type élémentaires, l'initialisation des variables, l'affectation et les comparaisons.
- La déclaration de variables locales avec l'inférence de type
- Les opérations arithmétiques : présente les opérateurs arithmétiques sur les entiers et les flottants et les opérateurs d'incrémentation et de décrémentation.
- La priorité des opérateurs : présente la priorité des opérateurs.
- Les structures de contrôles : présente les instructions permettant la réalisation de boucles, de branchements conditionnels et de débranchements.
- Les évolutions de l'instruction switch dans Java 14
- Les tableaux : présente la déclaration, l'initialisation explicite et le parcours d'un tableau
- Les conversions de types : présente la conversion de types élémentaires.
- L'autoboxing et l'unboxing

3.1. Les règles de base

Java est sensible à la casse.

Les blocs de code sont encadrés par des accolades. Chaque instruction se termine par un caractère ';' (point-virgule).

Une instruction peut tenir sur plusieurs lignes :

Exemple :

```
char
code
=
'D';
```

L'indentation est ignorée du compilateur mais elle permet une meilleure compréhension du code par le programmeur.

3.2. Les mots réservés du langage Java

Java 9 définit 54 mots réservés qui ne peuvent pas être utilisés comme identifiant.

Historiquement, les mots réservés (reserved words) peuvent être regroupés en deux catégories :

- 51 mots clés (keywords) dont 3 ne sont pas utilisés
- 3 valeurs littérales (literals) : true, false et null

Les mots clés sont des mots réservés utilisés dans le code source Java en ayant un rôle particulier dans la syntaxe du code.

abstract	continue	for	new	switch
assert (Java 1.4)	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum (Java 5)	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp (Java 1.2 à 16)	volatile
const	float	native	super	while
_ (Java 9)				

Les mots clé const et goto sont réservés mais ne sont pas utilisés actuellement.

A partir de Java 9, le mot-clé _ (le caractère underscore) est non utilisé pour le moment mais il est réservé pour une éventuelle utilisation future dans les déclarations de certaines variables.

A partir de Java 17, le mot-clé réservé strictfp est obsolète et ne devrait plus être utilisé dans le code.

Plusieurs séquences sont fréquemment considérées à tort comme des mots clés du langage et sont en fait des valeurs litérales :

- true et false : sont des valeurs littérales booléennes
- null qui est une valeur littérale

Différents types de mots-clés sont définis dans le langage Java en fonction de la version utilisée :

- mot-clé réservé depuis Java 1.0
- mot-clé restreint (restricted keyword) à partir de Java 9
- identifiant restreint (restricted identifier) à partir de Java 10
- mot-clé contextuel (contextual keyword) qui remplace les notions antérieures d'identifiant restreint et de mot-clé restreint dans la JLS à partir de Java 17

Le langage Java définit plusieurs mots-clés contextuels (pour certains nommés initialement mots-clés restreints ou identifiants restreints) depuis Java 9 :

exports	module	non-sealed (Java 17)	open
opens	permits (Java 17)	provides	record (Java 16)
requires	sealed (Java 17)	to	transitive
uses	var (Java 10)	when (Java 20)	with

yield (Java 14)			

Le but de ses mots-clés contextuels est de pouvoir ajouter des mots-clés qui sont considérés comme tels dans certains contextes et dans les autres cas sont considérés comme des identifiants légaux pour maintenir la compatibilité.

Une séquence de caractères correspondant à un mot-clé contextuel n'est pas traitée comme un mot-clé si une partie quelconque de la séquence peut être combinée avec les caractères immédiatement précédents ou suivants pour former un jeton différent.

Deux mots-clés doivent être séparés par un caractère d'espacement ou un commentaire.

Version de Java	Evolution des mots-clés
Java 9	10 mots-clés restreints (restricted keywords) sont introduits : open, module, requires, transitive, exports, opens, to, uses, provides et with. Ils sont considérés comme des mots-clés uniquement dans le contexte de la définition d'un descripteur de module. Dans tous les autres contextes, ils sont considérés comme des identifiants, pour assurer la compatibilité avec le code écrit avant l'introduction des mots-clés restreints. Il y a une exception : immédiatement à droite de la séquence de caractères requires dans la déclaration d'un descripteur de module, la séquence de caractères transitive est considérée comme un mot-clé à moins qu'elle ne soit suivie d'un séparateur, auquel cas elle est considérée comme un identifiant.
Java 10	La séquence de caractère var est considérée comme un identifiant restreint : var est considéré comme un mot-clé lorsque qu'il est utilisé pour la déclaration d'une variable locale en demandant au compilateur d'inférer le type en fonction de la valeur d'initialisation.
Java 11	var est aussi considéré comme un mot-clé lorsque qu'il est utilisé pour la déclaration un paramètre d'une expression lambda en demandant au compilateur d'inférer le type en fonction de la signature de l'unique méthode de l'interface fonctionnelle.
Java 14	La séquence de caractère yield est considérée comme un identifiant restreint : yield est considéré comme un mot-clé lorsque qu'il est utilisé dans une instruction swith comme une expression.
Java 15	La séquence de caractère record est considérée comme un identifiant restreint : record est considéré comme un mot-clé lorsque qu'il est utilisé pour déclarer un type qui encapsule des données de manière immuable.
Java 17	La notion de mot-clé contextuel (contextual keyword) remplace les notions antérieures d'identifiant restreint (restricted identifier) et de mot-clé restreint (restricted keyword) dans la JLS. Les mots-clés contextuels sealed, non-sealed et permits sont ajoutés. A noter que le mot-clé contextuel non-sealed est le premier mot clé à utiliser un nom composé avec un trait d'union. L'utilisation d'un nom composé avec un trait d'union permet d'ajouter de nouveaux mot clés tout en garantissant la compatibilité car un identifiant valide en Java ne peut pas contenir un trait d'union.

Le nom d'un type (type identifier) ne peut pas être permits, record, sealed, var ou yield.

Exemple :

```
public class var {
}
```

Résultat :

```
C:\java>javac -version
javac 10.0.2

C:\java>javac var.java
```

```
var.java:1: error: 'var' not allowed here
public class var {
             ^
  as of release 10, 'var' is a restricted local variable type and cannot be used for type
declarations
1 error

C:\java>
```

Exemple :

```
public class yield {
}
```

En Java 13, le compilateur émet un avertissement si un type se nomme yield.

Résultat :

```
C:\java>javac -version
javac 13

C:\java>javac yield.java
yield.java:1: warning: 'yield' may become a restricted type name in a future release and
may be unusable for type declarations or as the element type of an array
public class yield {
             ^
1 warning

C:\java>
```

En Java 13, le compilateur émet un avertissement lors de l'utilisation de yield comme identifiant.

Résultat :

```
C:\java>javac -version
javac 13

C:\java>javac TestYield.java
TestYield.java:7: warning: 'yield' may become a restricted identifier in a future release
    yield();
    ^
  (to invoke a method called yield, qualify the yield with a receiver or type name)
1 warning

C:\java>
```

A partir de Java 14, le compilateur émet une erreur si un type se nomme yield.

Résultat :

```
C:\java>javac -version
javac 14

C:\java>javac yield.java
yield.java:1: error: 'yield' not allowed here
public class yield {
             ^
  as of release 13, 'yield' is a restricted type name and cannot be used for type declarations
1 error

C:\java>
```

Jusqu'à Java 12, il est possible d'invoquer une méthode nommée yield sans la qualifier.

Exemple :

```
public class TestYield {

  public void yield() {
  }

  public void traiter() {
    yield();
  }
}
```

Résultat :

```
C:\java>javac -version
javac 12

C:\java>javac TestYield.java

C:\java>
```

A partir de Java 14, le compilateur émet une erreur car il faut obligatoirement qualifier l'invocation d'une méthode nommée yield() de façon à se distinguer d'une utilisation de yield comme instruction.

Résultat :

```
C:\java>javac -version
javac 14

C:\java>javac TestYield.java
TestYield.java:7: error: invalid use of a restricted identifier 'yield'
    yield();
    ^
  (to invoke a method called yield, qualify the yield with a receiver or type name)
1 error

C:\java>
```

record a une signification particulière dans une déclaration d'une classe record.

Jusqu'à Java 12, un type peut se nommer record.

Exemple (code Java 12) :

```
public class record {
}
```

En Java 14 et 15, le compilateur émet un avertissement si un type se nomme record.

Résultat :

```
C:\java>javac -version
javac 14

C:\java>javac record.java
record.java:1: warning: 'record' may become a restricted type name in a future release and
 may be unusable for type declarations or as the element type of an array
public class record {
             ^
1 warning

C:\java>
```

A partir de Java 16, le compilateur émet une erreur si un type se nomme record.

Résultat :

```
C:\java>javac -version
javac 16.0.1

C:\java>javac record.java
record.java:1: error: 'record' not allowed here
public class record {
             ^
  as of release 14, 'record' is a restricted type name and cannot be used for type declarations
1 error

C:\java>
```

3.3. Les identifiants

Les identifiants (identifiers) sont utilisés pour nommer des éléments dans du code source Java notamment les variables, les classes, les méthodes, les paramètres, les packages, les interfaces, les énumérations, les étiquettes, les modules, ... Les identifiants permettent aux développeurs d'utiliser un élément ailleurs dans le code.

Exemple :

```
public class MaClasse {
  public static void main(String[] args) {
    int valeur = 10;
  }
}
```

Dans l'exemple ci-dessus, plusieurs identifiants sont utilisés dans le code source :

- MaClasse : est le nom d'une classe
- main : est le nom d'une méthode
- String : est le nom d'une classe du JDK
- args : est le nom d'un paramètre
- valeur : est le nom d'une variable locale

Les spécifications de Java imposent des règles strictes pour définir un identifiant valide.

Un identifiant est une séquence d'un ou plusieurs caractères (lettres et chiffres) dont le premier est obligatoirement une lettre.

La première lettre est un caractère Unicode pour laquelle la valeur passée en paramètre de la méthode isJavaIdentifierStart() de la classe Character renvoie true. Le premier caractère ne peut donc pas être un chiffre.

Une lettre peut être entre-autre :

- A-Z (\u0041-\u005a)
- a-z (\u0061-\u007a)
- $ (\u0024)
- _ (\u005f)
- Les autres caractères Unicode valides sont ceux dont la valeur passée en paramètre de la méthode isJavaIdentifierPart() de la classe Character renvoie true.

La quasi-totalité des caractères Unicode peuvent être utilisés ce qui permet d'écrire des identifiants dans toutes les langues. Cependant plusieurs caractères ne peuvent pas être utilisés dans un identifiant : c'est notamment le cas des caractères qui ont une utilité dans le langage comme par exemple un espace, une tabulation, les caractères #, @, !, -, *, + , /, %, (,), ...

Les identifiants Java sont sensibles à la casse.

La taille d'un identifiant n'est pas limitée mais pour des raisons de lisibilité, il est préférable qu'ils ne soient pas trop petit ni trop grand.

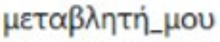
Remarque : depuis Java 9, l'identifiant composé uniquement d'un caractère underscore n'est plus valide car _ est devenu un mot clé du langage.

Un identifiant ne peut pas être :

- un mot clé du langage
- true ou false qui sont des booléens littéraux
- null qui est une valeur littérale

Les identifiants utilisés pour un type doivent respecter les règles générales de définition d'un identifiant. Depuis Java 10, l'identifiant d'un type ne peut pas être var qui est utilisé de manière particulière dans le contexte de la définition d'une variable locale.

Exemple d'identifiants valides :

_ (jusqu'à Java 8)	a	a1	_$	mavariable
MaVariable	MAVARIABLE	maVariable	ma_variable	mavariable1
ma1variable	mavariable_	_mavariable	ma$variable1	_maVariable
$maVariable	ma_variable_initialisée	 μεταβλητή_μου		
variable_avec_un_nom_tres_long_avec_chaque_mot_separe_par_un_underscore				

Exemple d'identifiants invalides :

Identifiant invalide	Raison pour laquelle il est invalide
1aabbb	Commence par un chiffre
aa-bb	Contient un caractère moins
aa bb	Contient un espace
_	Le caractère underscore est devenu un mot clé en Java 9
aa*bb	Contient un caractère *
()_	Contient des parenthèses
true	Est une valeur littérale réservée
new	Est un mot clé réservé du langage
aa+bb	Contient un caractère +
aa&bb	Contient un caractère &
aa@bb	Contient un caractère @
aa#bb	Contient un caractère #

Remarque : il est important de prêter une attention particulière lors de l'utilisation d'un identifiant pour que celui-ci soit suffisamment significatif dans la compréhension de ce qu'il représente

Il est aussi recommandé de respecter des normes de développement

3.4. Les commentaires

Ils ne sont pas pris en compte par le compilateur donc ils ne sont pas inclus dans le pseudo code. Ils ne se terminent pas par un caractère ";".

Il existe trois types de commentaire en Java :

Type de commentaires	Exemple
commentaire abrégé	// commentaire sur une seule ligne int N=1; // déclaration du compteur
commentaire multiligne	/* commentaires ligne 1 commentaires ligne 2 */
commentaire de documentation automatique	/** * commentaire de la méthode * @param val la valeur à traiter * @since 1.0 * @return la valeur de retour * @deprecated Utiliser la nouvelle méthode XXX */

3.5. La déclaration et l'utilisation de variables

Une variable, identifié par un nom, permet d'accéder à une valeur. En Java, une variable est obligatoirement typée statiquement à sa définition : une fois la variable définie, son type ne peut pas changer.

3.5.1. La déclaration de variables

Une variable possède un nom, un type et une valeur. La déclaration d'une variable doit donc contenir deux choses : un nom et le type de données qu'elle peut contenir. Une variable est utilisable dans le bloc où elle est définie.

La déclaration d'une variable permet de réserver la mémoire pour en stocker la valeur.

Le type d'une variable peut être :

- soit un type élémentaire dit aussi type primitif déclaré sous la forme type_élémentaire variable;
- soit une classe déclarée sous la forme classe variable ;

Exemple :

```
long nombre;
int compteur;
String chaine;
```

Rappel : les noms de variables en Java peuvent commencer par une lettre, par le caractère de soulignement ou par le signe dollar. Le reste du nom peut comporter des lettres ou des nombres mais jamais d'espace.

Il est possible de définir plusieurs variables de même type en séparant chacune d'elles par une virgule.

Exemple :

```
int jour, mois, annee ;
```

Java est un langage à typage rigoureux qui ne possède pas de transtypage automatique lorsque ce transtypage risque de conduire à une perte d'information. A partir de Java 5, l'autoboxing permet la conversion automatique d'un type primitif vers son wrapper correspondant. Le mécanisme inverse est nommé unboxing.

Pour les objets, il est nécessaire en plus de la déclaration de la variable de créer un objet avant de pouvoir l'utiliser. Il faut réserver de la mémoire pour la création d'un objet (remarque : un tableau est un objet en Java) avec l'instruction new. La libération de la mémoire se fait automatiquement grâce au garbage collector.

Exemple :

```
MaClasse instance; // déclaration de l'objet

instance = new MaClasse(); // création de l'objet

OU MaClasse instance = new MaClasse(); // déclaration et création de l'objet
```

Exemple :

```
int[] nombre = new int[10];
```

Il est possible en une seule instruction de faire la déclaration et l'affectation d'une valeur à une variable ou plusieurs variables.

Exemple :

```
int i=3 , j=4 ;
```

3.5.2. Les types élémentaires

Les types élémentaires ont une taille identique quel que soit la plate-forme d'exécution : c'est un des éléments qui permet à Java d'être indépendant de la plate-forme sur laquelle le code s'exécute.

Type	Désignation	Longueur	Valeurs	Commentaires
boolean	valeur logique : true ou false	1 bit	true ou false	pas de conversion possible vers un autre type
byte	octet signé	8 bits	-128 à 127	
short	entier court signé	16 bits	-32768 à 32767	
char	caractère Unicode	16 bits	\u0000 à \uFFFF	entouré de cotes simples dans du code Java
int	entier signé	32 bits	-2147483648 à 2147483647	
float	virgule flottante simple précision (IEEE754)	32 bits	1.401e-045 à 3.40282e+038	
double	virgule flottante double précision (IEEE754)	64 bits	2.22507e-308 à 1.79769e+308	
long	entier long	64 bits	-9223372036854775808 à 9223372036854775807	

Les types élémentaires commencent tous par une minuscule.

3.5.3. Le format des types élémentaires

Le format des nombres entiers :

Il existe plusieurs formats pour les nombres entiers : les types byte, short, int et long peuvent être codés en décimal, hexadécimal ou octal. Pour un nombre hexadécimal, il suffit de préfixer sa valeur par 0x. Pour un nombre octal, le nombre doit commencer par un zéro. Le suffixe l ou L permet de spécifier que c'est un entier long.

Le format des nombres décimaux :

Il existe plusieurs formats pour les nombres décimaux. Les types float et double stockent des nombres flottants : pour être reconnus comme tels ils doivent posséder soit un point, un exposant ou l'un des suffixes f, F, d, D. Il est possible de préciser des nombres qui n'ont pas le partie entière ou pas de partie décimale.

Exemple :

```
float pi = 3.141f;
double valeur = 3d;
float flottant1 = +.1f , flottant2 = 1e10f;
```

Par défaut un littéral représentant une valeur décimale est de type double : pour définir un littéral représentant une valeur décimale de type float il faut le suffixer par la lettre f ou F.

Attention :

```
float pi = 3.141; // erreur à la compilation
float pi = 3.141f; // compilation sans erreur
```

Exemple :

```
double valeur = 1.1;
```

Le format des caractères :

Un caractère est codé sur 16 bits car il est conforme à la norme Unicode. Il doit être entouré par des apostrophes. Une valeur de type char peut être considérée comme un entier non négatif de 0 à 65535. Cependant la conversion implicite par affectation n'est pas possible.

Exemple :

```
/* test sur les caractères */
class test1 {
    public static void main (String args[]) {
        char code = 'D';
        int index = code - 'A';
        System.out.println("index = " + index);
    }
}
```

3.5.4. L'initialisation des variables

Exemple :

```
int nombre; // déclaration
nombre = 100; //initialisation
OU int nombre = 100; //déclaration et initialisation
```

En Java, toute variable appartenant à un objet (définie comme étant un attribut de l'objet) est initialisée avec une valeur par défaut en accord avec son type au moment de la création. Cette initialisation ne s'applique pas aux variables locales des méthodes de la classe.

Les valeurs par défaut lors de l'initialisation automatique des variables d'instances sont :

Type	Valeur par défaut
boolean	false
byte, short, int, long	0
float, double	0.0
char	\u000

classe	null

Remarque : Dans une applet, il est préférable de faire les déclarations et initialisations dans la méthode init().

3.5.5. L'affectation

le signe = est l'opérateur d'affectation et s'utilise avec une expression de la forme variable = expression. L'opération d'affectation est associative de droite à gauche : il renvoie la valeur affectée ce qui permet d'écrire :

```
x = y = z = 0;
```

Il existe des opérateurs qui permettent de simplifier l'écriture d'une opération d'affectation associée à un opérateur mathématique :

Opérateur	Exemple	Signification
`=`	`a=10`	équivalent à : a = 10
`+=`	`a+=10`	équivalent à : a = a + 10
`-=`	`a-=10`	équivalent à : a = a - 10
`*=`	`a*=10`	équivalent à : a = a * 10
`/=`	`a/=10`	équivalent à : a = a / 10
`%=`	`a%=10`	reste de la division
`^=`	`a^=10`	équivalent à : a = a ^ 10
`<<=`	`a<<=10`	équivalent à : a = a << 10 a est complété par des zéros à droite
`>>=`	`a>>=10`	équivalent à : a = a >> 10 a est complété par des zéros à gauche
`>>>=`	`a>>>=10`	équivalent à : a = a >>> 10 décalage à gauche non signé

Attention : Lors d'une opération sur des opérandes de types différents, le compilateur détermine le type du résultat en prenant le type le plus précis des opérandes. Par exemple, une multiplication d'une variable de type float avec une variable de type double donne un résultat de type double. Lors d'une opération entre un opérande entier et un flottant, le résultat est du type de l'opérande flottant.

3.5.6. Les entiers exprimés en binaire (Binary Literals)

Avec Java 7, la valeur des types entiers (byte, short, int, et long) peut être exprimée dans le système binaire en utilisant le préfixe 0b ou 0B

Exemple (code Java 7) :

```
public static void testEntierBinaire() {
  byte valeurByte = (byte) 0b00010001;
  System.out.println("valeurByte = " + valeurByte);
  valeurByte = (byte) 0B10001;
  System.out.println("valeurByte = " + valeurByte);
  valeurByte = (byte) 0B11101111;
  System.out.println("valeurByte = " + valeurByte);
  short valeurShort = (short) 0b1001110111101;
  System.out.println("valeurShort = " + valeurShort);
  int valeurInt = 0b1000;
  System.out.println("valeurInt = " + valeurInt);
  valeurInt = 0b10011101000101101001101010000101;
  System.out.println("valeurInt = " + valeurInt);
  long valeurLong =
      0b010000101000101101000010100010110100001010001011010000101000101L;
  System.out.println("valeurLong = " + valeurLong);
  }
```

3.5.7. Utilisation des underscores dans les entiers littéraux

Il n'est pas facile de lire un nombre qui compte de nombreux chiffres : dès que le nombre de chiffres dépasse 9 ou 10 la lecture n'est plus triviale, ce qui peut engendrer des erreurs.

A partir de Java 7, il est possible d'utiliser un ou plusieurs caractères tiret bas (underscore) entre les chiffres qui composent un entier littéral. Ceci permet de faire des groupes de chiffres pour par exemple séparer les milliers, les millions, les milliards, ... afin d'améliorer la lisibilité du code.

Exemple (code Java 7) :

```
    int maValeur = 123_1456_789;
    maValeur = 4_3;
    maValeur = 4___3;
    maValeur = 0x4_3;
    maValeur = 0_43;
    maValeur = 04_3;
    maValeur = 0b1001110_10001011_01001101_01000101;
    long creditCardNumber = 1234_5678_9012_3456L;
    long numeroSecuriteSociale = 1_75_02_31_235_897L;
    long octetsDebutFichierClass = 0xCAFE_BABE;
    long maxLong = 0x7fff_ffff_ffff_ffffL;
    float pi = 3.141_593f;
```

Un nombre quelconque de caractères de soulignement (underscore) peut apparaître n'importe où entre les chiffres d'un littéral numérique. Le caractère underscore doit être placé uniquement entre deux chiffres. Il n'est donc pas possible de l'utiliser :

- Au début ou à la fin d'un nombre
- Avant ou après le point de séparation de la partie décimale d'un nombre flottant
- Avant les suffixes F ou L

Exemple (code Java 7) :

```
  // toutes ces expressions provoquent une erreur de compilation
  int maValeur = _43;
  int maValeur = 43_;
  int x5 = 0_x43;
  int x6 = 0x_43;
  int x8 = 0x43_;
  float pi1 = 3_.141593F;
  float pi2 = 3._141593F;
  long numeroSecuriteSociale = 1750231235897_L;
```

Le caractère underscore ne modifie pas la valeur mais facilite simplement sa lecture.

3.6. La déclaration de variables locales avec l'inférence de type

Globalement Java a la réputation d'être assez verbeux, en particulier par rapport à des langages plus récents : c'est une critique commune du langage par les développeurs Java novices et expérimentés.

Java est un langage typé ce qui permet au compilateur de détecter des erreurs liées à une mauvaise utilisation de type mais cela oblige à ajouter du code notamment lors de la définition de variables locales.

Historiquement, Java a toujours exigé que les déclarations de variables locales incluent explicitement le type. Bien que les types explicites puissent être très utiles, parfois ils ne sont pas très importants. Historiquement pour définir une variable et lui affecter une nouvelle instance, il est nécessaire de préciser le type à la déclaration (à gauche de l'opérateur

=) et de créer l'instance (à droite de l'opérateur =).

Exemple :

```
MaClasse mc = new MaClasse();
```

Cela implique une redondance du type. C'est aussi le cas lors de la déclaration et l'initialisation d'une variable de type primitif ou String : le type doit être explicitement précisé mais il est redondant avec la valeur affectée à la variable.

Jusqu'à Java 9 inclus, la déclaration de toutes variables impose de préciser explicitement le type.

Exemple :

```
String salutation = "Bonjour";
```

Sur cette simple déclaration, le fait de préciser explicitement le type est redondant avec la valeur littérale d'initialisation qui implique obligatoire une variable de type String.

Pour une variable locale qui est un objet, cela implique généralement d'utiliser deux fois le type dans le code de déclaration de la variable :

- une première fois pour indique le type de la variable
- une seconde pour invoquer le constructeur

Exemple :

```
ArrayList<String> liste = new ArrayList<String>();
```

Si la variable est de type primitif et qu'elle est initialisée, le type utilisé dans la déclaration est redondant avec la valeur.

Exemple :

```
int valeur = 10;
```

Java 10 a introduit une nouvelle fonctionnalité dans le langage Java appelée inférence de type pour variable locale via la JEP 286. Le but de la JEP 286 est de permettre au compilateur d'inférer le type des variables locales qui sont initialisées avec une valeur. Le compilateur pourra alors déterminer le type de la variable et dispenser le développeur de l'indiquer explicitement. Elle permet de réduire la quantité de code Java à produire pour un traitement courant : la déclaration de variables locales en introduisant un sucre syntaxique.

L'objectif est de réduire la verbosité de Java, comme le propose déjà de nombreux autres langages modernes typés comme Scala, C#, Go ou Kotlin qui offrent déjà l'inférence de type pour la définition de variables locales et permettent ainsi de déclarer des variables locales sans préciser leur type. Comme dans ces langages, il est possible d'utiliser un instruction, var en Java, pour déclarer une variable locale dont le type est inféré à partir de la valeur qui lui est affectée.

De nombreux langages qui utilisent l'instruction var pour déclarer une variable propose aussi un autre mot clé comme val ou let pour définir une variable immuable. Java 10 ne propose que le mot clé var : il faut donc le compléter avec le mot clé final pour obtenir une déclaration d'une variable dont la valeur est immuable.

Le choix a été fait de ne pas utiliser une instruction val pour définir une variable immuable. L'immutabilité est importante mais son intérêt est très limité pour des variables locales. De plus, Java 8 a introduit la notion de variable effectivement finale.

Cette fonctionnalité va permettre de réduire la quantité de code requise pour déclarer une variable locale en Java depuis sa création. Tant que le compilateur peut déduire le type à partir de la valeur d'initialisation, il n'est plus obligatoire de définir explicitement le type d'une variable locale mais de le remplacer par l'instruction var.

Lors du traitement de var, le compilateur utilise le type de la valeur d'initialisation de la variable pour définir le type de la variable et l'utiliser dans le bytecode.

Lors de la déclaration d'une variable locale initialisée, il est possible d'utiliser var à la place du type ou du type primitif pour indiquer au compilateur d'inférer le type de la variable. Cela n'est possible que pour des variables qui sont initialisée afin de permettre au compilateur de déterminer le type en fonction de la valeur d'initialisation.

Ainsi le type d'une variable locale pourra disparaitre dans le code source mais Java reste un langage typé et c'est simplement le compilateur qui est en charge de déterminer le type et de l'utiliser dans le bytecode.

Cela ne concerne que la déclaration de variables locales :

- uniquement celles avec une valeur d'initialisation qui permettra au compilateur d'inférer le type selon la valeur
- les variables dans les boucles for et les try with resources
- la variable contenant l'élément courant dans les boucles for améliorées

Cela n'est pas possible dans les autres cas : paramètres de méthodes et constructeurs, type de retour, champs, instruction catch, ...

L'instruction var ne rend pas le langage Java typé dynamiquement : Java reste un langage statiquement typé. Une fois le type inféré par le compilateur, celui-ci est ajouté dans le bytecode et il n'est pas possible d'affecter une valeur non adéquate au type inféré à la variable.

3.6.1. L'instruction var

Java 10 propose l'instruction var pour définir une variable locale : l'instruction var remplace le type dans le code. Le compilateur va alors inférer le type de la variable sous réserve que la variable soit initialisée avec une valeur qui permette au compilateur de déterminer le type.

Exemple (code Java 10) :

```
var salutation = "Bonjour";
```

Pour les variables locales initialisées explicitement, il est donc possible de remplacer le type de la variable par l'instruction var. Cette nouvelle syntaxe réduit la verbosité associée à la déclaration d'une variable locale en Java, tout en maintenant la sécurité du typage statique : elle permet la déclaration de variables sans avoir à préciser explicitement le type.

Exemple

Code	Type inféré pour la variable
var i = 2 ;	int
var i = 2L ;	long
var liste = new ArrayList<String>();	ArrayList<String>
var stream = liste.stream();	Stream(<String>)
var chemin = Paths.get("fichier.txt");	Path
var contenu = Files.readAllBytes(chemin);	byte[]
var references = new HashMap<Integer, String>();	HashMap<Integer, String>

L'utilisation d'une boucle requiert fréquemment une variable locale : l'instruction var peut aussi être utilisée pour définir une telle variable.

Exemple (code Java 10) :

```
  public static void main(String[] args) {
    for (var arg : args) {
      System.out.println(arg);
    }

    for (var i = 0; i < args.length; i++) {
      var arg = args[i];
      System.out.println(arg);
    }
  }
```

L'identifiant var n'est pas un mot clé : il n'est pas ajouté à la liste des mots réservés du langage Java. L'identifiant var est un identifiant restreint : il est reconnu comme un mot clé qu'aux endroits où il peut être interprété comme tel dans le contexte de la déclaration d'une variable locale. Ainsi même si ce n'est pas recommandé, il est possible de déclarer une variable locale nommée var.

Exemple (code Java 10) :

```
var var = 2;
```

Ainsi la majorité du code existant qui utilise var comme identifiant se compile correctement en Java 10. Comme var n'est pas un mot clé, il est toujours possible de nommer une variable, un champ, un paramètre, une méthode ou un package avec var.

Par contre, un type nommé var va poser des soucis : est-ce que var est l'instruction ou le type ? Ce cas devrait être rare puisque la convention de nommage très largement utilisée préconise de commencer le nom d'un type par une majuscule.

Il n'est pas possible d'utiliser var comme nom d'un type (une classe, une interface, une énumération, un record) : cette limitation est ajoutée dans les spécifications du langage Java.

3.6.2. Les restrictions d'utilisation de l'instruction var

L'utilisation de l'instruction var doit respecter plusieurs contraintes qui seront vérifiées lors de la compilation.

L'utilisation de l'identifiant var pour inférer le type d'une variable locale n'est utilisable que dans certaines circonstances. Il n'est pas possible d'utiliser l'inférence de type sur la déclaration de variables locales :

- qui ne sont pas initialisées (la variable locale doit être initialisée explicitement)
- qui déclarent plusieurs variables
- qui déclarent un tableau avec une initialisation
- qui déclarent une variable avec la valeur null : l'instance sera affectée ultérieurement et il n'est pas possible de présumer son type

L'instruction var n'est utilisable que pour définir des variables locales : elle ne peut donc être utilisée que dans le corps d'une méthode ou d'un bloc de code ou d'une expression Lambda.

Exemple (code Java 10) :

```
Consumer<String> c = s -> {var i = 0; System.out.println(i); };
```

Il n'est pas possible d'utiliser l'instruction var dans un autre contexte : il n'est donc par exemple pas possible de l'utiliser avec des paramètres de méthodes, des valeurs de retour de méthodes, des champs ...

L'instruction var n'est utilisable que pour définir une variable locale : dans les autres cas, il est toujours obligatoire de préciser explicitement le type.

Résultat :

```
jshell> private var getValeur() { return 10; }
|  Error:
|  'var' is not allowed here
|  private var getValeur() { return 10; }
|          ^-^
```

Il n'est pas possible d'utiliser l'instruction var pour indiquer le type de retour d'une méthode.

Exemple (code Java 10) :

```
  var getLibelle() {       // erreur de compilation
    return "Description";
  }
```

Il n'est pas possible d'utiliser var pour définir un paramètre d'une méthode.

Exemple (code Java 10) :

```
  void saluer(var nom) {     // erreur de compilation
    System.out.println("Bonjour "+nom);
  }
```

Le choix de limiter l'utilisation de l'instruction var uniquement à la déclaration de variables locales est dictée pour maintenir l'inférence à côté de la déclaration et ainsi limiter les erreurs pouvant être induites plus loin dans le code.

Le compilateur doit être en mesure d'inférer le type : cela impose que la variable soit initialisée avec une valeur qui permettre au compilateur de déterminer le type inféré.

Résultat :

```
jshell> var valeur;
|  Error:
|  cannot infer type for local variable valeur
|    (cannot use 'var' on variable without initializer)
|  var valeur;
|  ^---------^
```

La variable doit être initialisée à sa déclaration pour permettre au compilateur d'inférer le type par rapport à la valeur assignée.

Exemple (code Java 10) :

```
    var salutation;            // erreur de compilation
    salutation = "Bonjour";
```

La compilation du code ci-dessus provoque une erreur « cannot infer type for local variable salutation ».

Il n'est pas possible d'utiliser l'instruction var avec une variable initialisée avec la valeur null.

Résultat :

```
jshell> var titres = null;
|  Error:
|  cannot infer type for local variable titres
|    (variable initializer is 'null')
|  var titres = null;
|  ^----------------^
```

Il n'est pas possible d'utiliser l'instruction var pour déclarer plusieurs variables en même temps, même si le type inféré par le compilateur est le même pour toutes les variables.

Résultat :

```
jshell> var x, y = 0;
|  Error:
|  'var' is not allowed in a compound declaration
|  var x, y = 0;
|      ^

jshell> var x = 0, y = 0;
|  Error:
|  'var' is not allowed in a compound declaration
|  var x = 0, y = 0;
|      ^
```

Il faut utiliser une instruction var dédiée pour chaque variable.

Résultat :

```
jshell> var x=0; var y=0;
x ==> 0
y ==> 0
```

Il n'est pas possible d'utiliser l'instruction var pour une variable qui n'a pas encore de type cible dû au fait qu'il n'est pas encore inféré (poly expressions) tels que les expressions Lambda, les références de méthodes ou l'initialisation d'un tableau avec des valeurs.

Il n'est pas possible d'utiliser l'instruction var avec une initialisation de tableau.

Résultat :

```
jshell> var chaines = {"e1","e2"};
|  Error:
|  cannot infer type for local variable chaines
|    (array initializer needs an explicit target-type)
|  var chaines = {"e1","e2"};
|  ^-----------------------^
```

Il faut dans ce cas obligatoirement invoquer le constructeur pour créer une instance du tableau qui sera initialisée avec les valeurs fournies.

Résultat :

```
jshell> var chaines = new String[] {"e1","e2"}
chaines ==> String[2] { "e1", "e2" }
|  created variable chaines : String[]
```

Ainsi, il n'est pas possible d'initialiser une variable déclarée avec var et initialisée avec une expression lambda ou une référence de méthode.

Exemple (code Java 10) :

```
    var salutation = () -> "Bonjour";    // erreur de compilation
```

Il n'est pas possible d'utiliser l'instruction var avec une expression Lambda comme valeur sans préciser explicitement l'interface fonctionnelle

Résultat :

```
jshell> var affichage = () -> { System.out.println("Bonjour"); };
|  Error:
|  cannot infer type for local variable affichage
|    (lambda expression needs an explicit target-type)
|  var affichage = () -> { System.out.println("Bonjour"); };
|  ^-------------------------------------------------------^
```

Il n'est pas possible d'utiliser l'instruction var avec une référence de méthode.

Résultat :

```
jshell>  var max = Math::max;
|  Error:
|  cannot infer type for local variable max
|    (method reference needs an explicit target-type)
|   var max = Math::max;
|   ^------------------^

jshell> var comparerStr = String::compareTo
|  Error:
|  cannot infer type for local variable comparerStr
|    (method reference needs an explicit target-type)
|  var comparerStr = String::compareTo;
|  ^----------------------------------^
```

Il faut obligatoirement caster l'expression lambda ou la référence de méthode avec l'interface fonctionnelle correspondante.

Résultat :

```
jshell> var additionner = (IntBinaryOperator) (a,b) -> a+b
additionner ==> $Lambda$21/762227630@4e7dc304
|  created variable additionner : IntBinaryOperator
```

Il est possible de modifier la valeur d'une variable définie avec var tant que la valeur est affectable au type inféré.

Exemple (code Java 10) :

```
    var salutation = "Bonjour" ;
    salutation = "Hello";
```

Java reste statiquement typé : une fois le type inféré par le compilateur, il est utilisé dans le byte code. Il n'est pas possible de modifier le type d'une variable ou de lui affecter une valeur qui ne corresponde pas à son type

Exemple (code Java 10) :

```
    var valeur = 0 ;
    valeur = "test";       // erreur de compilation
```

3.6.3. Les avantages et les inconvénients

L'inférence de type pour les variables locales avec l'instruction var est une fonctionnalité qui fait l'objet de controverses. Certains apprécient la concision de la déclaration des variables locales : le code est plus court grâce à l'élimination du type qui est généralement redondant. D'autres craignent que l'absence du type nuise à la lisibilité due à l'élimination du type.

Comme pour toutes les fonctionnalités, il doit être utilisé avec discernement. Il n'y a pas de règles absolues pour savoir quand elle devrait et ne devrait pas être utilisée.

La règle essentielle est que la lecture du code est plus importante que l'écriture du code.

3.6.3.1. La facilitation de la déclaration de variables locales

L'utilisation de l'instruction var est une fonctionnalité du langage Java dont le but est de faciliter la déclaration et l'initialisation de variables locales en laissant le compilateur inférer le type. C'est un sucre syntaxique qui délègue au compilateur la responsabilité de déterminer le type lors de la définition d'une variable locale en fonction de sa valeur d'initialisation. L'inférence de type pour les variables locales permet de réduire un peu la quantité de code nécessaire à la déclaration de variables locales.

L'instruction var réduit la quantité de code requise pour déclarer des variables locales en Java. C'est notamment le cas lorsque le type de la variable est long voir très long car il inclut par exemple plusieurs types génériques.

L'instruction var facilite la déclaration de variables intermédiaires et ainsi de permettre leur accès. Il est aussi fréquent de définir une variable qui est uniquement utilisé sur la ligne suivante pour définir une autre variable.

Exemple :

```
    URL url = new URL("http://www.google.com");
    URLConnection con = url.openConnection();
    Reader reader = new BufferedReader(new InputStreamReader(con.getInputStream()));
    reader.close();
```

L'utilisation de l'inférence pour les variables locales réduit la quantité de code à écrire pour obtenir le même résultat.

Exemple :

```
    var url = new URL("http://www.google.com")
    var con = url.openConnection();
    var reader = new BufferedReader(new InputStreamReader(con.getInputStream()));
    reader.close();
```

Cela réduit la quantité de code en évitant la redondance sur le type mais cela améliore aussi la lisibilité puisque toutes les variables sont alignées sur la même colonne.

Avec l'instruction var, il est beaucoup plus facile de déclarer des variables intermédiaires voire même de le faire à des endroits où la complexité nous aurait rebuté de le faire auparavant. C'est notamment le cas lors de l'utilisation d'expressions imbriquées ou chaînées.

3.6.3.2. La lisibilité

Comme avec d'autres langages qui proposent cette fonctionnalité, l'inférence du type de variables locales peut conduire à du code plus ou moins lisible : il est de la responsabilité du développeur d'utiliser cette fonctionnalité avec discernement.

L'omission d'un type explicite peut réduire le code nécessaire, mais seulement si son omission ne nuit pas à la compréhension. Le type n'est pas le seul moyen de fournir de l'information : il est aussi possible d'en fournir via le nom de la variable et la valeur d'initialisation.

L'instruction var permet de réduire la quantité de code lors de la déclaration d'une variable locale, mais sans y prêter attention son utilisation peut aussi en réduire la lisibilité et la compréhension.

Exemple (code Java 10) :

```
    var resultats = rechercherService.obtenirPremiers();
```

La lecture du code ci-dessus ne permet pas facilement de connaître le type de la variable inféré par le compilateur. Evidemment, les IDE fournissent des fonctionnalités pour afficher le type de la variable mais le code n'est pas toujours lu dans un IDE, par exemple comme dans une page sur le web ou dans des outils de revue de code.

C'est aussi vrai lors du remplacement d'un type explicite par l'utilisation d'une instruction var.

Exemple (code Java 8) :

```
    List<Resultat> r = rechercherService.obtenirPremiers();
```

Exemple (code Java 10) :

```
    var r = rechercherService.obtenirPremiers();
```

L'utilisation de noms de variable courts n'est généralement pas une bonne pratique mais dans ce cas, il est encore plus recommandé d'accorder une importance accrue dans le nommage des variables déclarées avec l'instruction var.

Dans certains cas, l'utilisation de l'instruction var ne permet que de réduire la quantité de code à produire. Cependant, elle peut permettre de rendre le code plus lisible notamment si le type de la variable est très long (par ce que son nom est très long et il utilise un ou plusieurs types génériques qui peuvent être imbriqués)

Exemple (code Java 10) :

```
Commande<Client, Produits<Map<Reference, Integer>>, TypeCommande> commande =
    traiterCommandeUrgente(client, articles);
```

Le fait de ne plus voire explicitement le type peut parfois être handicapant : bien sûr les IDE permettent facilement de connaitre le type de la variable mais sans IDE cela peut être plus difficile comme par exemple lors d'une revue de code ou de la lecture de code sur Internet.

Généralement l'absence de type réduit la compréhension du code notamment car la lecture du type facilite la compréhension du code notamment lors de l'utilisation de type ou de type complexe avec générique. Les IDE proposent des fonctionnalités pour afficher le type d'un élément du code source mais cela requiert d'avoir un IDE.

Par contre l'utilisation de l'instruction var améliore la lisibilité car les noms des variables déclarées peuvent être alignées, même si là aussi les IDE peuvent réaliser cet alignement.

Le type est important mais le nom d'une variable doit l'être aussi : il l'est d'autant plus avec l'utilisation de l'instruction var car le type n'est plus visible dans le code source.

Il est de la responsabilité des développeurs d'assurer un équilibre entre verbosité et clarté du code. Sans type visible et avec des noms de variables peu compréhensifs, le code peut devenir moins facile à comprendre.

Exemple (code Java 10) :

```
    // Personne p = service.get();
    var p = service.get();
```

Il est donc important de n'utiliser l'instruction var que lorsque cela rend le codeplus concis mais pas moins clair : elle ne devrait pas forcement être utilisé systématiquement. D'autant que l'utilisation de var peut encourager l'utilisation de variables locales et donc l'écriture de méthodes plus longues.

3.6.4. Quelques mises en garde sur l'utilisation de var

Le code Java reste statiquement typé mais avec l'utilisation de l'identifiant var, le développeur ne voit plus le type. C'est notamment le cas si la valeur de la variable locale est initialisée avec la valeur de retour d'une méthode.

Exemple (code Java 10) :

```
jshell> public int getValeur() { return 10; }
|  created method getValeur()
```

```
jshell> var valeur = getValeur()
valeur ==> 10
|  created variable valeur : int
```

Le compilateur infère le type le plus proche possible : le type déterminé lors de l'inférence n'est pas un super type.

L'exemple ci-dessous ne fonctionne pas car le type inféré n'est pas List<String> mais ArrayList<String>. Il n'est donc pas possible d'affecter une instance de type LinkedList<String> à la variable.

Résultat :

```
jshell> var liste = new ArrayList<String>();
liste ==> []

jshell> liste = new LinkedList<String>();
|  Error:
|  incompatible types: java.util.LinkedList<java.lang.String> cannot be converted
to java.util.ArrayList<java.lang.String>
|  liste = new LinkedList<String>();
|          ^----------------------^
```

Pour que cet exemple puisse fonctionner, il faut obligatoirement préciser explicitement le type de la variable et ne pas laisser le compilateur le déterminer par inférence.

Le type inféré est parfois non trivial.

Exemple (code Java 10) :

```
jshell> var liste = List.of(1.0, 1, "un")
liste ==> [1.0, 1, un]
|  created variable liste : List<Serializable&Comparable<? extends Serializable&Comparable<?>>>
```

Le type inféré par le compilateur est le résultat de l'intersection des types des valeurs fournies dans la collection. La complexité de l'intersection des types peut s'accroitre avec la version de Java utilisée.

Exemple (code Java 10) :

```
jshell> var liste = List.of(1, "un")
liste ==> [1, un]
|  created variable liste : List<Serializable&Comparable<? extends Serializable&Comparable<?>>>
```

Exemple (code Java 12) :

```
jshell> var liste = List.of(1, "un")
liste ==> [1, un]
|  created variable liste : List<Serializable&Comparable<? extends Serializable&Comparable<?>
&java.lang.constant.Constable&java.lang.constant.ConstantDesc>&java.lang.constant.Constable
&java.lang.constant.ConstantDesc>
```

Avant Java 10, le type d'une classe anonyme interne ne peut pas être déterminé pour être utilisé dans le code.

Dans l'exemple ci-dessous, la classe n'implémente pas un super type qui contienne la méthode à invoquer. Il est possible d'utiliser le type Object mais il ne sera pas possible d'invoquer la méthode afficher().

Exemple (code Java 1.1) :

```
(new Object() {
  void afficher() {
    System.out.println("test");
  }
}).afficher();
```

Avec cette syntaxe, le résultat de l'invocation du constructeur est chaîné à l'invocation de la méthode. Cette syntaxe n'est utilisable que si la classe ne possède qu'une seule méthode à invoquer.

Avec l'instruction var, il est possible de déclarer une variable dont le type est la classe anonyme et d'obtenir le type de la classe. La variable ayant le type réel, il est possible d'invoquer la méthode afficher().

En utilisant l'inférence de type avec l'instruction var, il est possible de définir une variable dont la valeur est une instance d'une classe anonyme interne.

Exemple (code Java 10) :

```
jshell> var objet = new Object() {
   ...> void afficher() {
   ...> System.out.println("test");
   ...> }
   ...> }
objet ==> $1@3fee9989
|  created variable objet : <anonymous class extending Object>

jshell> objet.afficher();
test
```

Comme la variable contient une référence sur l'instance du type de la classe interne, il est possible d'invoquer d'autres méthodes de l'instance.

Exemple (code Java 10) :

```
jshell> var objet = new Object() {
   ...> void afficher() {
   ...> System.out.println("afficher");
   ...> }
   ...> void calculer() {
   ...> System.out.println("calculer");
   ...> }
   ...> }
objet ==> $1@7085bdee

jshell> objet.afficher();
afficher

jshell> objet.calculer();
calculer
```

Lors de l'utilisation de l'instruction avec une variable initialisée avec l'invocation d'un constructeur d'une classe générique avec l'opérateur diamant, le type inféré sera paramétré avec Object.

Exemple (code Java 10) :

```
jshell> var references = new ArrayList<>();
references ==> []
|  created variable references : ArrayList<Object>
```

Cette situation survient notamment lors du remplacement du type explicite par une instruction var.

Exemple (code Java 10) :

```
jshell> List<String> references = new ArrayList<>();
references ==> []
|  created variable references : List<String>
```

Pour imposer le type paramétré inféré, il faut préciser l'argument de type lors de l'invocation du constructeur.

Exemple (code Java 10) :

```
jshell> var references = new ArrayList<String>();
references ==> []
|  created variable references : ArrayList<String>
```

Dans les exemples ci-dessus, il est important de noter que le type inféré n'est pas List<> mais ArrayList<>.

3.6.5. Les messages d'erreur courants lors d'une utilisation incorrecte de var

Différents messages d'erreur sont émis par le compilateur javac du JDK en cas de situation illicite dans l'utilisation de l'instruction var.

Lorsque la variable n'est pas initialisée :

Exemple (code Java 10) :

```
public class TestVar {

  public static void main(String[] args) {
    var valeur;
  }
}
```

Résultat :

```
C:\java>javac TestVar.java
TestVar.java:4: error: cannot infer type for local variable valeur
    var valeur;
        ^
  (cannot use 'var' on variable without initializer)
1 error
```

Lorsque la variable est initialisée avec null :

Exemple (code Java 10) :

```
public class TestVar {

  public static void main(String[] args) {
    var obj = null;
  }
}
```

Résultat :

```
C:\java>javac TestVar.java
TestVar.java:4: error: cannot infer type for local variable obj
    var obj = null;
        ^
  (variable initializer is 'null')
1 error
```

Lorsque plusieurs variables sont définies avec la même instruction var :

Exemple (code Java 10) :

```
public class TestVar {

  public static void main(String[] args) {
    var a = 1, b = 2;
  }
}
```

Résultat :

```
C:\java>javac TestVar.java
TestVar.java:4: error: 'var' is not allowed in a compound declaration
    var a = 1, b = 2;
        ^
1 error
```

Lorsque la variable est initialisée avec une expression Lambda sans la caster avec le type de son interface fonctionnelle correspondante :

Exemple (code Java 10) :

```
public class TestVar {

  public static void main(String[] args) {
    var additionner = (a,b) -> a + b;
  }
}
```

Résultat :

```
C:\java>javac TestVar.java
TestVar.java:4: error: cannot infer type for local variable additionner
    var additionner = (a,b) -> a + b;
        ^
  (lambda expression needs an explicit target-type)
1 error
```

Lorsque la variable est initialisée avec une référence de méthode sans la caster avec le type de son interface fonctionnelle correspondante :

Exemple (code Java 10) :

```
public class TestVar {

  public static void main(String[] args) {
    var afficher = System.out::println;
  }
}
```

Résultat :

```
C:\java>javac TestVar.java
TestVar.java:4: error: cannot infer type for local variable afficher
    var afficher = System.out::println;
        ^
  (method reference needs an explicit target-type)
1 error
```

Lorsque la variable est initialisée avec l'initialisation d'un tableau sans en préciser explicitement le type :

Exemple (code Java 10) :

```
public class TestVar {

  public static void main(String[] args) {
    var valeurs = {1, 2, 3};
  }
}
```

Résultat :

```
C:\java>javac TestVar.java
TestVar.java:4: error: cannot infer type for local variable valeurs
    var valeurs = {1, 2, 3};
        ^
  (array initializer needs an explicit target-type)
1 error
```

3.6.6. Les comparaisons

Java propose des opérateurs pour toutes les comparaisons :

Opérateur	Exemple	Signification
`>`	`a > 10`	strictement supérieur
`<`	`a < 10`	strictement inférieur
`>=`	`a >= 10`	supérieur ou égal
`<=`	`a <= 10`	inférieur ou égal
`==`	`a == 10`	Egalité
`!=`	`a != 10`	diffèrent de
`&`	`a & b`	ET binaire
`^`	`a ^ b`	OU exclusif binaire
`|`	`a | b`	OU binaire
`&&`	`a && b`	ET logique (pour expressions booléennes) : l'évaluation de l'expression cesse dès qu'elle devient fausse
`||`	`a || b`	OU logique (pour expressions booléennes) : l'évaluation de l'expression cesse dès qu'elle devient vraie
`? :`	`a ? b : c`	opérateur conditionnel : renvoie la valeur b ou c selon l'évaluation de l'expression a (si a alors b sinon c) : b et c doivent retourner le même type

Les opérateurs sont exécutés dans l'ordre suivant à l'intérieur d'une expression qui est analysée de gauche à droite:

- incréments et décréments
- multiplication, division et reste de division (modulo)
- addition et soustraction
- comparaison
- le signe = d'affectation d'une valeur à une variable

L'usage des parenthèses permet de modifier cet ordre de priorité.

3.7. Les opérations arithmétiques

Les opérateurs arithmétiques se notent + (addition), - (soustraction), * (multiplication), / (division) et % (reste de la division). Ils peuvent se combiner à l'opérateur d'affectation

Exemple :

```
nombre += 10;
```

3.7.1. L'arithmétique entière

Pour les types numériques entiers, Java met en oeuvre une sorte de mécanisme de conversion implicite vers le type int appelé promotion entière. Ce mécanisme fait partie des règles mises en place pour renforcer la sécurité du code.

Exemple :

```
short x= 5 , y = 15;
x = x + y ; //erreur à la compilation

Incompatible type for =. Explicit cast needed to convert int to short.
x = x + y ; //erreur à la compilation
^
1 error
```

Les opérandes et le résultat de l'opération sont convertis en type int. Le résultat est affecté dans un type short : il y a donc risque de perte d'informations et donc une erreur est émise à la compilation. Cette promotion évite un débordement de capacité sans que le programmeur soit pleinement conscient du risque : il est nécessaire, pour régler le problème, d'utiliser une conversion explicite ou cast.

Exemple :

```
x = (short) ( x + y );
```

Il est nécessaire de mettre l'opération entre parenthèses pour que ce soit son résultat qui soit converti car le cast a une priorité plus forte que les opérateurs arithmétiques.

La division par zéro pour les types entiers lève l'exception ArithmeticException.

Exemple :

```
/* test sur la division par zéro de nombres entiers */
class test3 {
  public static void main (String args[]) {
    int valeur=10;
    double resultat = valeur / 0;
    System.out.println("index = " + resultat);
  }
}
```

3.7.2. L'arithmétique en virgule flottante

Avec des valeurs float ou double, la division par zéro ne produit pas d'exception mais le résultat est indiqué par une valeur spéciale qui peut prendre trois états :

- indéfini : Float.NaN ou Double.NaN (not a number)
- indéfini positif : Float.POSITIVE_INFINITY ou Double.POSITIVE_INFINITY, + ∞
- indéfini négatif : Float.NEGATIVE_INFINITY ou Double.NEGATIVE_INFINITY, - ∞

Conformément à la norme IEEE754, ces valeurs spéciales représentent le résultat d'une expression invalide NaN, une valeur supérieure au plafond du type pour infini positif ou négatif.

X	Y	X / Y	X % Y
valeur finie	0	+ ∞	NaN
valeur finie	+/- ∞	0	x
0	0	NaN	NaN
+/- ∞	valeur finie	+/- ∞	NaN

+/- ∞	+/- ∞	NaN	NaN

Exemple :

```
/* test sur la division par zéro de nombres flottants */

class test2 {
  public static void main (String args[]) {
    float valeur=10f;
    double resultat = valeur / 0;
    System.out.println("index = " + resultat);
  }
}
```

3.7.3. L'incrémentation et la décrémentation

Les opérateurs d'incrémentation et de décrémentation sont : `n++ ++n n-- --n`

Si l'opérateur est placé avant la variable (préfixé), la modification de la valeur est immédiate sinon la modification n'a lieu qu'à l'issue de l'exécution de la ligne d'instruction (postfixé)

L'opérateur ++ renvoie la valeur avant incrémentation s'il est postfixé, après incrémentation s'il est préfixé.

Exemple :

```
System.out.println(x++);
// est équivalent à
System.out.println(x); x = x + 1;

System.out.println(++x);
// est équivalent à
x = x + 1; System.out.println(x);
```

Exemple :

```
/* Test sur les incrémentations préfixées et postfixées */

class test4 {
  public static void main (String args[]) {
    int n1=0;
    int n2=0;
    System.out.println("n1 = " + n1 + " n2 = " + n2);
    n1=n2++;
    System.out.println("n1 = " + n1 + " n2 = " + n2);
    n1=++n2;
    System.out.println("n1 = " + n1 + " n2 = " + n2);
    n1=n1++;        //attention
    System.out.println("n1 = " + n1 + " n2 = " + n2);
  }
}
```

Résultat :

```
int n1=0;
int n2=0; // n1=0 n2=0
n1=n2++; // n1=0 n2=1
n1=++n2; // n1=2 n2=2
n1=n1++; // attention : n1 ne change pas de valeur
```

3.8. La priorité des opérateurs

Java définit les priorités dans les opérateurs comme suit (du plus prioritaire au moins prioritaire)

les opérateurs postfix	expr++ expr--
les opérateurs unaires	++expr --expr +expr -expr ~ !
les opérateurs de multiplication, division et modulo	* / %
les opérateurs d'addition et soustraction	+ -
les opérateurs de décalage	<< >> >>>
les opérateurs de comparaison	< > <= >= instanceof
les opérateurs d'égalité	== !=
l'opérateur OU exclusif	^
l'opérateur ET	&
l'opérateur OU	|
l'opérateur ET logique	&&
l'opérateur OU logique	||
l'opérateur ternaire	? :
les opérateurs d'assignement	= += -= *= /= %= ^= |= <<= >>= >>>=
l'opérateur arrow	->

L'ordre d'interprétation des opérateurs peut être modifié par l'utilisation de parenthèses.

3.9. Les structures de contrôles

Comme la quasi-totalité des langages de développement orientés objets, Java propose un ensemble d'instructions qui permettent d'organiser et de structurer les traitements. L'usage de ces instructions est similaire à celui rencontré avec leur équivalent dans d'autres langages.

3.9.1. Les boucles

Les boucles permettent d'exécuter plusieurs fois un bloc de code selon une condition ou le parcours des éléments d'un tableau ou d'un Iterable. Java propose plusieurs types de boucles :

- while(condition)
- do ... while(condition)
- for
- for améliorée (depuis Java 1.5)

3.9.1.1. Les boucles while

Une boucle while permet d'exécuter l'instruction ou le bloc de code qui la suit tant que la condition booléenne est évaluée à vraie. La condition est évaluée en début de chaque itération. La syntaxe générale est de la forme :

```
while ( boolean ) {
  // code à exécuter dans la boucle
}
```

Exemple :

```
public class Boucle {

  public static void main(String[] args) {
    int i = 1;
    while (i <= 3) {
      System.out.println(i++);
    }
  }
}
```

Résultat :

```
1
2
3
```

Si avant l'instruction while, le booléen est false, alors le code de la boucle ne sera jamais exécuté.

Exemple :

```
public class Boucle {

  public static void main(String[] args) {
    System.out.println("debut");
    int i = 4;
    while (i <= 3) {
      System.out.println(i++);
    }
    System.out.println("fin");
  }
}
```

Résultat :

```
debut
fin
```

Attention : ne pas mettre de ; après les parenthèses de la condition sinon cela produira une boucle infinie qui consomme énormément de CPU uniquement pour tester la condition.

3.9.1.2. Les boucles do ... while

Une boucle do ... while permet d'exécuter l'instruction ou le bloc de code qui la suit tant que la condition booléenne est évaluée à vraie.

La syntaxe générale est de la forme :

```
do {
  // ...
} while ( boolean );
```

Exemple :

```
public class Boucle {

  public static void main(String[] args) {
    int i = 1;
    do {
      System.out.println(i++);
    } while (i <= 3);
  }
}
```

Résultat :

```
1
2
3
```

La condition est évaluée en fin de chaque itération : la boucle est donc exécutée au moins une fois quelle que soit la valeur du booléen lors de la première itération.

Exemple :

```
public class Boucle {

  public static void main(String[] args) {
    int i = 4;
    do {
      System.out.println(i++);
    } while (i <= 3);
  }
}
```

Résultat :

```
4
```

3.9.1.3. Les boucles for

La boucle for permet de réaliser une boucle dont la définition est composée de plusieurs parties optionnelles. La syntaxe générale est :

```
for ( initialisation; condition; modification ) {
    ...
}
```

Exemple :

```
public class Boucle {

  public static void main(String[] args) {
    int i = 0;
    for (i = 1; i < 4; i++) {
      System.out.println(i);
    }
  }
}
```

Attention : comme toutes les parties sont optionnelles, il est facile de faire une boucle infinie qui va consommer 100% de CPU jusqu'à ce que la JVM soit tuée.

Exemple :

```
for ( ; ; ) {
} // boucle infinie
```

L'initialisation, la condition et la modification de l'index sont optionnelles.

Dans l'initialisation, on peut déclarer une variable qui servira d'index. La variable incrémentée peut être définie dans la partie initialisation et qui sera dans ce cas locale à la boucle : elle ne sera donc accessible que dans le bloc de code de l'instruction for.

Exemple :

```
public class Boucle {

  public static void main(String[] args) {
    for (int i = 1; i < 4; i++) {
      System.out.println(i);
    }
  }
}
```

Il est possible d'inclure plusieurs traitements dans l'initialisation et la modification de la boucle : chacun des traitements doit être séparé par une virgule.

Exemple :

```
  for (i = 0 , j = 0 ; i * j < 1000; i++ , j+= 2) {
    // ...
  }
```

La condition peut ne pas porter sur l'index de la boucle :

Exemple :

```
  boolean trouve = false;
  for (int i = 0 ; !trouve ; i++ ) {
    if ( tableau[i] == 1 ) {
      trouve = true;
      // gestion de la fin du parcours du tableau
      // ...
    }
  }
```

3.9.1.4. Les boucles for améliorées

Java 1.5 propose une nouvelle syntaxe pour les boucles for : les boucles for améliorées pour faciliter le parcours intégral des implémentations de l'interface Iterator et des tableaux.

L'itération sur les éléments d'une collection est verbeuse avec la déclaration d'un objet de type Iterator.

Exemple (code Java 5.0) :

```
import java.util.*;

public class BoucleForAvecIterator {

  public static void main(String[] args) {
    List liste = new ArrayList();
    for(int i = 0; i < 10; i++) {
      liste.add(i);
    }

    for (Iterator iter = liste.iterator(); iter.hasNext(); ) {
      System.out.println(iter.next());
    }
  }
}
```

La nouvelle forme de l'instruction for, spécifiée dans la JSR 201, permet de simplifier l'écriture du code pour réaliser une telle itération et laisse le soin au compilateur de générer le bytecode nécessaire.

Exemple (code Java 5.0) :

```
import java.util.*;

public class TestFor {

  public static void main(String[] args) {
    List liste = new ArrayList();
    for(int i = 0; i < 10; i++) {
      liste.add(i);
    }

    for (Object element : liste) {
      System.out.println(element);
    }
  }
}
```

L'utilisation de la syntaxe de l'instruction for améliorée peut être renforcée en combinaison avec les génériques, ce qui évite l'utilisation d'un cast.

Exemple (code Java 5.0) :

```
import java.util.*;
import java.text.*;

public class BoucleForAvecGeneriques {

  public static void main(String[] args) {
    List<Date> liste = new ArrayList();
    for(int i = 0; i < 10; i++) {
      liste.add(new Date());
    }

    DateFormat df = DateFormat.getDateInstance();

    for (Date element : liste) {
      System.out.println(df.format(element));
    }
  }
}
```

La syntaxe de l'instruction for améliorée peut aussi être utilisée pour parcourir tous les éléments d'un tableau.

Exemple (code Java 5.0) :

```
import java.util.*;

public class BoucleForAvecTableau {

  public static void main(String[] args) {
    int[] tableau = {0,1,2,3,4,5,6,7,8,9};

    for (int element : tableau) {
      System.out.println(element);
    }
  }
}
```

Cela permet d'éviter la déclaration et la gestion dans le code d'une variable contenant l'index courant lors du parcours du tableau.

3.9.2. Les branchements conditionnels

Les instructions de branchements conditionnels permettent l'exécution d'un bloc de code selon le résultat de l'évaluation d'une condition.

3.9.2.1. L'instruction if

L'instruction if permet d'exécuter l'instruction ou le bloc de code qui la suit si l'évaluation de la condition booléenne est vraie.

Il est possible d'utiliser une instruction optionnelle else qui précise une instruction ou un bloc de code à exécuter si l'évaluation de la condition booléenne est fausse. La syntaxe générale est de la forme :

```
if (boolean) {
    ...
} else if (boolean) {
    ...
} else {
    ...
}
```

Exemple :

```
  public static void main(String[] args) {
    estPaire(2);
    estPaire(3);
  }

  public static void estPaire(int val) {
    if (val % 2 == 0) {
      System.out.println(val + " est paire");
    } else {
      System.out.println(val + " est impaire");
    }
  }
```

Résultat :

```
2 est paire
3 est impaire
```

3.9.2.2. L'opérateur ternaire

L'opérateur ternaire est un raccourci syntaxique pour une instruction if/else qui renvoie simplement une valeur selon l'évaluation de la condition.

La syntaxe générale est de la forme :

```
( condition ) ? valeur-vraie : valeur-fausse
```

Exemple :

```
if (niveau == 5) // équivalent à total = (niveau == 5) ? 10 : 5;
  total = 10;
else total = 5;
```

Il est possible d'imbriquer des opérateurs ternaires mais ce n'est pas recommandé car cela nuit à la lisibilité du code.

3.9.2.3. L'instruction switch

L'instruction switch permet d'exécuter du code selon l'évaluation de la valeur d'une expression. La syntaxe historique générale est de la forme :

```
switch (expression) {
  case valeur1 :
    instr11;
    instr12;
    break;
  case valeur2 :
  case valeur3 :
    ...
    break;
  case valeur4 :
    ...
    break;
  default :
    ...
}
```

Si une instruction case ne contient pas de break alors les traitements associés au case suivant sont exécutés : cette fonctionnalité se nomme le fall through. Cela permet d'exécuter les mêmes instructions pour plusieurs valeurs.

Avant Java 7, on ne peut utiliser l'instruction switch qu'avec des types primitifs d'une taille maximum de 32 bits (byte, short, int, char) ou une énumération. L'utilisation d'une chaîne de caractères dans une instruction switch provoquait une erreur à la compilation "Incompatible Types. Require int instead of String".

A partir de Java SE 7, il est possible d'utiliser un objet de type String dans l'expression fournie à l'instruction switch.

Exemple (code Java 7) :

```
public static Boolean getReponse(String reponse) {
  Boolean resultat = null;
  switch(reponse) {
    case "oui" :
    case "Oui" :
      resultat = true;
      break;
    case "non" :
    case "Non" :
      resultat = false;
      break;
    default:
      resultat = null;
      break;
  }
  return resultat;
  }
```

L'instruction switch compare la valeur de la chaîne de caractères avec la valeur fournie à chaque instruction case comme si elle utilisait la méthode String.equals(). Dans les faits, le compilateur utilise la méthode String.hashCode() pour faire la comparaison. Le compilateur va ainsi générer un code qui est plus optimisé que le code équivalent avec des instructions if/else.

Important : il est nécessaire de vérifier que la chaîne de caractères évaluée par l'instruction switch ne soit pas null sinon une exception de type NullPointerException est levée.

Le test réalisé par l'instruction switch est sensible à la casse : il faut donc en tenir compte si un test ne l'est pas.

Exemple (code Java 7) :

```
  public static Boolean getReponse(String reponse) {
    Boolean resultat = null;

    switch (reponse.toLowerCase()) {
      case "oui":
        resultat = true;
        break;
      case "non":
        resultat = false;
        break;
      default:
        resultat = null;
        break;
    }
    return resultat;
  }
```

A partir de Java 14, l'instruction switch a été revue en profondeur pour proposer 4 formes syntaxiques, une utilisation comme instruction ou comme une expression, la possibilité de gérer plusieurs valeurs dans un case. Ces nouvelles possibilités sont détaillées dans une section suivante.

Il est possible d'imbriquer des instructions switch même si cela n'est pas recommandé pour des raisons de lisibilité.

Pour certains cas d'usage, l'instruction switch peut être remplacée avantageusement par une utilisation du polymorphisme.

3.9.3. Les débranchements

Le langage Java propose plusieurs instructions pour réaliser des débranchements :

- break
- continue
- return

Les deux premières instructions peuvent être utilisées avec une étiquette.

Remarque : l'instruction goto est un mot clé réservé du langage Java mais non utilisé.

Attention : l'utilisation des débranchements est à utiliser avec parcimonie car elle peut contribuer à dégrader la lecture et la maintenabilité du code et induire des bugs subtils.

3.9.3.1. L'instruction break

L'instruction break permet de quitter immédiatement une boucle ou un branchement. Elle est utilisable dans tous les contrôles de flot.

La syntaxe est de la forme

```
break;
```

L'instruction break est utilisée pour mettre fin à une boucle dès qu'une instruction Break est exécutée dans une boucle, l'itération courante de la boucle s'arrête immédiatement et le contrôle revient à l'instruction suivant la boucle.

Exemple :

```
public class Boucle {

  public static void main(String[] args) {
    for (int i = 1; i < 5; i++) {
      if (i == 3) break;
      System.out.println(i);
    }
  }
}
```

Résultat :

```
1
2
```

Remarque : l'instruction break, lorsqu'elle est utilisée à l'intérieur d'un ensemble de boucles imbriquées, n'interrompt que la boucle la plus interne.

Exemple :

```
public class Boucle {

  public static void main(String[] args) {
    for (int i = 1; i < 3; i++) {
      for (int j = 1; j < 5; j++) {
        if (j == 3)
          break;
        System.out.println(i + " - " + j);
      }
    }
  }
}
```

Résultat :

```
1 - 1
1 - 2
2 - 1
2 - 2
```

3.9.3.2. L'instruction continue

L'instruction continue s'utilise dans une boucle pour passer directement à l'itération suivante.

La syntaxe est de la forme :

```
continue;
```

L'instruction continue peut-être utilisée à l'intérieur des structures de contrôle de type boucles. À l'intérieur de la boucle, lorsqu'une instruction continue est rencontrée, le contrôle saute directement au début de la boucle pour aller à l'itération suivante : cela met immédiatement fin à l'exécution des instructions de l'itération en cours.

L'instruction continue est utilisée pour sauter une condition particulière et poursuivre l'exécution à la prochaine itération.

L'instruction continue est utilisable avec toutes les types de boucle en Java :

- dans le cas d'une boucle for, le mot clé continue force le contrôle à sauter immédiatement à l'instruction d'incrémentation
- dans le cas d'une boucle while ou do-while, le contrôle saute immédiatement à l'expression booléenne
- dans le cas d'une boucle for améliorée, le contrôle saute à l'élément suivant

Exemple avec une boucle for

Exemple :

```
public class Boucle {

  public static void main(String[] args) {
    for (int i = 1; i < 5; i++) {
      if (i == 1 || i == 3) continue;
      System.out.println(i);
    }
  }
}
```

Résultat :

```
2
4
```

Exemple avec une boucle while

Exemple :

```
public class Boucle {

  public static void main(String[] args) {
    int i = 0;
    while (i < 4) {
      i++;
      if (i == 1 || i == 3) continue;
      System.out.println(i);
    }
  }
}
```

Résultat :

```
2
4
```

Exemple avec une boucle do-while

Exemple :

```
public class Boucle {

  public static void main(String[] args) {
    int i = 0;
    do {
      i++;
      if (i == 1 || i == 3) continue;
      System.out.println(i);
    } while (i < 4);
  }
}
```

Résultat :

```
2
4
```

Il est possible d'utiliser une instruction continue dans des boucles imbriquées. L'instruction continue permet d'ignorer le reste des traitements de l'itération en cours. Dans le cas de boucles imbriquées, elle passe à l'itération suivante de la boucle la plus interne.

Exemple :

```
public class Boucle {

  public static void main(String[] args) {
    for (int i = 1; i < 3; i++) {
      for (int j = 1; j < 5; j++) {
        if (j == 1 || j == 3)
          continue;
        System.out.println(i + " - " + j);
      }
    }
  }
}
```

Résultat :

```
1 - 2
1 - 4
2 - 2
2 - 4
```

Exemple avec une boucle for améliorée

Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.List;

public class Boucle {

  public static void main(String[] args) {
    List<Integer> liste = new ArrayList<Integer>();
    for(int i = 0; i < 5; i++) {
      liste.add(i);
    }

    for (Integer val : liste) {
      if (val % 2 == 0) continue;
      System.out.println(val);
    }
  }
}
```

Résultat :

```
1
3
```

3.9.3.3. L'utilisation de débranchement avec étiquette

break et continue peuvent s'exécuter avec des blocs nommés grâce à une étiquette.

Une étiquette est un nom suivi d'un caractère deux-points qui définit le début d'une instruction.

Remarque : l'utilisation dans le code des débranchements n'est pas recommandée.

Le langage Java permet d'exécuter des débranchements dans le code en utilisant des étiquettes (labels).

Il est possible de préciser une étiquette pour indiquer le point de retour lors de la fin du traitement déclenché par le break.

Une étiquette peut être utilisée pour identifier un bloc de code.

La syntaxe est de la forme :

```
etiquette:
  {
    instruction1;
    instruction2;
    ...
  }
```

Il est possible d'utiliser une instruction de débranchement break en la faisant suivre du nom d'une étiquette pour permettre un débranchement à la fin du bloc de code identifié par cette étiquette.

La syntaxe est de la forme :

```
break etiquette;
```

Exemple :

```
public class TestBreakAvecEtiquette {

  public static void main(String args[]) {
    boolean saut = true;

    bloc1: {
      System.out.println("debut bloc1");
      bloc2: {
        System.out.println("debut bloc2");

        bloc3: {
          System.out.println("debut bloc3");

          if (saut) {
            break bloc2;
          }
          System.out.println("fin bloc3");
        }
        System.out.println("fin bloc2");
      }
      System.out.println("fin bloc1");
    }
  }
}
```

Résultat :

```
debut bloc1
debut bloc2
debut bloc3
fin bloc1
```

Il est aussi possible de nommer une boucle à l'aide d'une étiquette pour permettre de l'interrompre même si cela est peu recommandé :

Exemple :

```
public class TestBreakAvecEtiquette {
  public static void main(String args[]) {
    int compteur = 0;
    boucle: while (compteur < 100) {

      for (int compte = 0; compte < 10; compte++) {
        compteur += compte;
        System.out.println("compteur = " + compteur);
        if (compteur > 20)
          break boucle;
      }
    }
  }
}
```

Résultat :

```
compteur = 0
compteur = 1
compteur = 3
compteur = 6
compteur = 10
compteur = 15
compteur = 21
```

3.9.3.4. L'instruction return

L'instruction return est utilisée pour sortir immédiatement de l'exécution d'une méthode et retourner le contrôle à l'appelant.

Elle peut être utilisée dans une méthode avec ou sans valeur de retour.

Pour les méthodes qui définissent un type de retour, l'instruction return doit être immédiatement suivie de la valeur de retour.

Exemple :

```
  public static long additionner(int a, int b) {
    return (long) a +b;
  }
```

Dans une telle méthode, toutes les branches de sortie de la méthode doivent avoir une instruction return qui fournit la valeur retournée pour la branche.

Exemple :

```
public class TestReturn {

  public static boolean estPair(int a) {
    if ( a % 2 == 0 ) return true;
  }
}
```

Résultat :

```
C:\java>javac TestReturn.java
TestReturn.java:5: error: missing return statement
  }
  ^
1 error
```

Pour régler l'erreur, il faut utiliser une instruction return dans la branche else de l'instruction if.

Exemple :

```
public class TestReturn {

  public static boolean estPair(int a) {
    if ( a % 2 == 0 ) return true;
        else return false;
  }
}
```

L'instruction return peut être utilisée à différents endroits dans une méthode, mais il faut toujours s'assurer qu'elle doit être la dernière instruction à être exécutée dans une méthode. Si ce n'est pas le cas, une erreur est émise par le compilateur.

L'instruction return ne doit pas nécessairement être la dernière instruction d'une méthode, mais elle doit être la dernière instruction à être exécutée dans une méthode.

Exemple :

```
public class TestReturn {

  public static long additionner(int a, int b) {
    return (long) a +b;
    System.out.println((long) a+b);
  }
}
```

Résultat :

```
C:\java>javac TestReturn.java
TestReturn.java:5: error: unreachable statement
    System.out.println((long) a+b);
    ^
TestReturn.java:6: error: missing return statement
  }
  ^
2 errors
```

Dans une méthode qui ne retourne pas de valeur, il est possible d'utiliser une instruction return pour arrêter immédiatement les traitements de la méthode.

Exemple :

```
public class TestReturn {

  public static void main(String[] args) {
    afficherPaireouImpaire(-1);
    afficherPaireouImpaire(2);
    afficherPaireouImpaire(3);
  }

  public static void afficherPaireouImpaire(int a) {
    if (a <= 0)
      return;
    if (a % 2 == 0)
      System.out.println(a + " est paire");
    else
      System.out.println(a + " est impaire");
  }
}
```

3.10. Les évolutions de l'instruction switch dans Java 14

L'instruction switch est présente dans le langage Java depuis la version 1.0. Sa syntaxe est inspirée de celle des langages C et C++ : elle permet d'exécuter du code selon la valeur qui lui est fournie.

Cependant, sa syntaxe n'est pas exempte de défaut, notamment :

- la sémantique par défaut (fall-through) offre une certaine flexibilité mais impose l'utilisation de l'instruction break à la fin des traitements à exécuter
- l'oubli accidentel d'une instruction break, entraine l'exécution du code de l'instruction case suivante. Ceci entraine fréquemment des bugs ou des erreurs
- une seule valeur n'est utilisable par instruction case

Historiquement et jusqu'à Java 12, le mot clé réservé switch ne pouvait être utilisé que comme une structure de contrôle qui ne peut qu'exécuter un ou plusieurs traitements selon l'évaluation d'une valeur. A partir de Java 12 en preview et Java 14 en standard, l'instruction switch peut maintenant être utilisée comme une expression permettant de retourner une valeur. Plusieurs changements ont également été apportés à la syntaxe des déclarations des cas dans l'instruction switch.

L'instruction switch a évolué tout d'abord dans Java 12 dans une première version preview puis dans une seconde version preview en Java 13 :

- en Java 12, via la JEP 325 : propose plusieurs évolutions dans l'instruction switch notamment une nouvelle syntaxe avec l'opérateur arrow, la possibilité d'utiliser un switch comme une expression et de mettre plusieurs valeurs dans un case
- en Java 13, via la JEP 354 : cette seconde preview introduit l'instruction yield en remplacement de l'instruction break pour retourner une valeur d'un switch utilisé comme une expression

Finalement en Java 14, via la JEP 361, ces évolutions deviennent des fonctionnalités standards en Java 14.

L'instruction switch peut dès lors être utilisée soit comme une structure du contrôle qui ne renvoie rien ou comme une expression qui renvoie une valeur.

Ces évolutions concernent la syntaxe et l'utilisation de l'instruction switch, dont voici un résumé :

- l'utilisation possible de deux variantes syntaxiques : la syntaxe historique "case L :" et une syntaxe sémantiquement plus courte, plus claire et moins sujette aux erreurs
- la syntaxe propose une nouvelle forme nommée « case L -> » : elle indique que seul le code à droite de l'opérateur -> (arrow) sera exécuté si la valeur correspond à la valeur de l'instruction case. La conséquence est que l'instruction break n'est plus utile : il n'y a plus de fall-through
- le support de valeurs multiples dans une clause case : quel que soit la syntaxe utilisée, il est possible de définir plusieurs valeurs dans une clause case, chacune séparée par une virgule
- l'utilisation de l'instruction switch comme expression : elle renvoie alors une valeur. Le résultat de l'expression est une valeur qui peut être assignée où utilisée partout ou une expression du type retourné peut être utilisée
- elle peut être utilisée comme une structure de contrôle ou une expression : les deux formes peuvent utiliser la syntaxe traditionnelle ou une syntaxe simplifiée
- le fall-through : il est toujours utilisé avec la syntaxe historique mais la nouvelle syntaxe ne le met pas en oeuvre
- l'
- le renvoie d'une valeur : une seule expression à droite du "case L ->" suffit. Un nouvel identifiant restreint "yield" est introduit pour retourner une valeur dans une instruction switch. Il doit être utilisé dans le cas où un bloc de code est nécessaire ou dans la syntaxe historique
- l'exhaustivité des cas : dans le cas d'une utilisation comme une expression, toutes les valeurs possibles doivent être prise en compte dans les cas, ce qui oblige généralement à utiliser l'instruction default

Les types utilisables comme valeur à tester par l'instruction switch sont toujours les mêmes : byte, short, int, char, leurs wrappers, String et les énumérations.

3.10.1. La syntaxe de l'instruction switch avec l'opérateur arrow

Historiquement dans une instruction switch, chaque cas est défini avec le mot clé réservé case suivi d'une unique valeur puis du caractère deux points (case X:)

En plus de la forme historique de l'instruction switch, une nouvelle forme simplifiée nommée "case L ->" est introduite pour définir les cas d'une instruction switch. La nouvelle syntaxe utilise le mot clé réservé case et l'opérateur arrow : case X -> où x est la valeur du cas.

Dans une instruction switch, le code à la droite de l'opérateur -> ne peut être qu'une expression ou un traitement unique, un bloc de code ou la levée d'une exception.

Si plusieurs instructions doivent être exécutées pour un même cas, il faut obligatoirement les regrouper dans un bloc de code. Etant des blocs dédiés, il est possible d'utiliser le même nom de variable dans chacun des blocs.

La nouvelle syntaxe utilisant l'opérateur arrow peut être utilisée dans un switch en tant qu'instruction ou comme expression.

Le grand intérêt de cette nouvelle syntaxe avec l'opérateur arrow est que si la valeur correspond à celle de la clause case, alors les traitements ou la valeur à sa droite est uniquement exécutée. Il n'y a donc jamais de débordement (fall through) sur la clause case suivante, donc pas besoin de mettre une instruction break avec cette syntaxe.

3.10.2. L'utilisation de l'instruction switch comme une expression

Fréquemment, l'instruction switch est utilisée pour définir la valeur d'une variable en fonction de la valeur d'une autre.

Une expression est une séquence composée de valeurs littérales, de variables, d'opérateurs et d'invocations de méthodes qui une fois évaluée produit une valeur unique.

Le langage Java permet de construire des expressions composées à partir de diverses expressions plus petites tant que le type de données requis par une partie de l'expression correspond au type de données de l'autre.

Comme avec l'opérateur ternaire qui est une expression équivalente à une instruction if/else, l'instruction switch est améliorée pour pouvoir être utilisée comme une expression. L'instruction switch est toujours utilisables comme un traitement mais elle est aussi utilisable comme une expression.

La syntaxe de l'instruction switch évolue pour permettre son utilisation en tant qu'expression : comme toute expression, elle peut être utilisée partout où une expression peut être attendue. Par exemple, comme valeur d'affectation à une variable, comme valeur d'une instruction return, ...

Cela signifie qu'il n'est plus nécessaire de déclarer d'abord une variable, puis de lui attribuer une valeur dans chaque branche. Combiné à l'utilisation de la syntaxe avec l'opérateur arrow le code devient plus simple.

Utilisée comme une expression, l'instruction switch doit retourner une valeur pour toutes les valeurs possibles du type de la variable passée en paramètre.

3.10.2.1. L'instruction yield pour retourner une valeur

Dans la JEP 325, c'était une instruction break suivie de la valeur qui est utilisée pour préciser une valeur de retour.

Les retours ont indiqué que l'utilisation de l'instruction break pour indiquer la valeur à retourner pourrait être source de confusion. D'autant que le langage Java permet déjà l'utilisation de break (et continue) avec une étiquette pour effectuer un débranchement similaire à une forme de goto.

Dans la JEP 354, Java 13 a introduit le nouveau mot-clé contextuel yield qui doit être utilisé pour retourner une valeur dans un switch comme une expression avec la syntaxe utilisant l'opérateur arrow et quelque chose qui n'est pas une simple expression. C'est le cas, si le traitement est un bloc de code.

L'utilisation de l'instruction break (avec ou sans étiquette) et yield réduit l'ambiguïté lors de l'utilisation d'un switch comme une structure de contrôle ou comme expression : l'instruction break ne peut être utilisée qu'avec un switch comme instruction et l'instruction yield ne peut être utilisée qu'avec un switch comme expression.

Important : yield n'est pas un mot clé réservé mais comme var c'est un identifiant restreint. yield peut être utilisé comme un identifiant mais a un rôle particulier dans l'instruction switch.

3.10.2.2. Les contraintes

Quant l'instruction switch est une expression, elle doit obligatoirement se terminer par un point-virgule si elle est à la fin d'une ligne de code. Ceci est différent de l'utilisation comme instruction. Le compilateur émet une erreur si une instruction switch utilisée comme expression ne se termine pas par un point-virgule en fin de ligne.

Exemple (code Java 14) :

```
public class TestSwitch {

  public static void main(String[] args) {

    int valeur = 2;
    String type = switch(valeur) {
      case 1  -> "un";
      case 2  -> "deux";
      default -> "hors limite";
    }
    System.out.println(type);
  }
}
```

Résultat :

```
C:\java>javac TestSwitch.java
TestSwitch.java:9: error: ';' expected
    }
     ^
1 error
```

3.10.2.2.1. Les Poly Expressions

L'utilisation d'une instruction switch comme une expression est une poly expression.

Cela signifie qu'elles n'ont pas de type définitif propre, mais peuvent être de plusieurs types. Les poly expressions les plus utilisées sont les lambdas : s -> s + " " qui peuvent être l'implémentation d'une Function ou d'un UnaryOperator avec des types génériques différents.

Si le type de la valeur de retour est précisé alors toutes les valeurs retournées de toutes les branches doivent respecter ce type.

Si le type de la valeur de retour n'est pas précisé, comme lors de l'utilisation de var, alors il est déterminé en trouvant le supertype le plus spécifique des types que les branches retournent.

3.10.2.2.2. L'exhaustivité

Lors de l'utilisation d'une instruction comme une structure de contrôle, il n'est pas obligatoire de prendre en compte toutes les valeurs possibles selon le type de la variable fournie à l'instruction switch.

Il est alors possible d'oublier involontairement de prendre en compte une ou plusieurs valeurs : cela ne va pas empêcher le code de se compiler mais le résultat obtenu ne sera peut-être pas celui attendu.

Cette problématique est aggravée avec l'utilisation de l'instruction switch comme une expression. Si l'exhaustivité n'était pas obligatoire, il faudrait renvoyer des valeurs par défaut sont le type de retour : cela conduirait sûrement à des erreurs subtiles et occasionnelles.

Pour éviter cela, le compilateur va vérifier que toutes les valeurs du type de la variable à évaluer sont prises en comptes dans les clauses case de manière explicite. Cela implique dans tous les cas, sauf celui d'une énumération où toutes les valeurs sont prises en compte, d'utiliser une clause default.

3.10.3. L'utilisation de valeurs multiples dans une clause case

Historiquement la clause case d'une instruction switch n'accepte qu'une seule valeur. Pour définir plusieurs valeurs, il est nécessaire d'utiliser le mécanisme fall-through.

Dans la version preview de Java 12, il est possible de préciser plusieurs valeurs dans une clause case en séparant chaque d'elle par une virgule. Les valeurs multiples sont utilisables dans la syntaxe traditionnelle et dans la syntaxe utilisant l'opérateur arrow.

Exemple (code Java 14) :

```
    int position = 1;
    switch (position) {
      case 1, 2, 3 -> System.out.println("Sur le podium");
      default      -> System.out.println("Hors podium");
    }
```

Cette possibilité est utilisable en remplacement de l'utilisation du fall-though puisqu'elle est aussi utilisable avec la syntaxe historique.

Exemple (code Java 14) :

```
    int position = 1;
    switch (position) {
      case 1, 2, 3 : System.out.println("Sur le podium"); break;
      default :      System.out.println("Hors podium");
    }
```

L'utilisation de valeurs multiples est aussi possible lorsque l'instruction switch est utilisée comme une expression.

Que cela soit avec la syntaxe historique :

Exemple (code Java 14) :

```
    int position = 1;
    boolean monteSurPodium = switch (position) {
      case 1, 2, 3 : yield true;
      default      : yield false;
    };
    System.out.println(monteSurPodium);
```

Ou que cela soit avec la nouvelle syntaxe :

Exemple (code Java 14) :

```
    int position = 1;
    boolean monteSurPodium = switch (position) {
      case 1, 2, 3 -> true;
      default      -> false;
    };
    System.out.println(monteSurPodium);
```

Avec la possibilité d'utiliser plusieurs valeurs dans un case, il ne sera probablement plus utile d'utiliser le fall-through.

3.10.4. Les 4 formes de l'utilisation de switch

L'instruction switch peut être utilisée comme :

- une structure de contrôle qui exécute un traitement selon une valeur

- mais aussi comme une expression qui renvoie une valeur

L'instruction switch propose aussi deux syntaxes :

- la syntaxe historique qui le caractère deux points
- une nouvelle syntaxe qui utilise l'opérateur arrow « -> »

Par combinaison des deux syntaxes et des deux cas d'utilisation, l'instruction switch possède 4 formes d'utilisation.

L'utilisation d'un « : » dans une clause case n'implique pas forcement que le switch est utilisé comme une structure de contrôle et l'utilisation d'un opérateur arrow n'implique par forcement que le switch est utilisé comme une expression.

3.10.4.1. L'utilisation de switch comme structure de contrôle avec la syntaxe classique

La syntaxe historique de l'instruction switch est toujours valide.

Avant Java 12, l'instruction switch est un traitement impératif qui conditionne l'exécution de code selon la valeur d'une variable. Historiquement inspirée dans l'instruction switch en C, sa syntaxe impose quelques contraintes et peut être à l'origine de bugs subtiles :

- un case ne peut avoir qu'une valeur
- fall-through par défaut : si l'instruction break n'est pas ajoutée à la fin d'un traitement d'un cas, le code du cas suivant est aussi exécuté et ainsi de suite jusqu'à la prochaine instruction break ou la fin du code l'instruction switch. C'est la manière d'exécuter un même traitement pour plusieurs valeurs
- la portée d'une variable définie dans une instruction switch est le bloc de l'instruction, de sorte qu'il n'est pas possible de réutiliser un nom de variable dans deux cas différents
- le cas par défaut, identifié avec l'instruction default est optionnel : il est alors possible de ne pas prendre en compte certains cas, volontairement ou involontairement

Un usage courant de l'instruction switch est de déterminer une valeur à partir d'une autre. Comme l'instruction switch ne peut pas renvoyer de valeur, il faut utiliser une variable intermédiaire qui doit être assignée à chaque cas.

Exemple :

```
  public static void main(String[] args) {
    FeuTricolore feu = VERT;
    String action;
    switch(feu) {
    case VERT :
      action = "Passage avec priorité à droite";
      break;
    case ORANGE :
      action = "Stop";
      break;
    case ROUGE :
      action = "Interdiction absolue de passer";
      break;
    default :
      action = "Passage avec priorité à droite";
    }
    System.out.println(action);
  }
```

Résultat :

```
Passage avec priorité à droite
```

Une possibilité pour pallier à certains problèmes est d'utiliser l'instruction switch dans une méthode : cela permet de retourner une valeur et d'éviter d'avoir à utiliser des breaks.

Exemple :

```
  public static String obtenirAction(FeuTricolore feu) {
    switch(feu) {
    case VERT :
      return "Poursuite de sa route avec priorité à droite";
    case ORANGE :
      return "Stop";
    case ROUGE :
      return "Interdiction absolue de passer";
    default :
      return "Priorité à droite";
    }
  }
```

Le fall-through permet d'exécuter le même traitement pour plusieurs valeurs qui doivent être précisés dans leur propre cas.

Exemple :

```
  public static void main(String[] args) {
    FeuTricolore feu = VERT;
    String action;
    switch(feu) {
    case ORANGE :
      action = "Stop";
      break;
    case ROUGE :
      action = "Interdiction absolue de passer";
      break;
    case VERT :
    default :
      action = "Passage avec priorité à droite";
    }
    System.out.println(action);
  }
```

Résultat :

```
Passage avec priorité à droite
```

Il peut aussi être problématique car il est facile d'oublier une instruction break.

Exemple :

```
  public static void main(String[] args) {
    FeuTricolore feu = VERT;
    String action;
    switch(feu) {
    case VERT :
      action = "Passage avec priorité à droite";
    case ORANGE :
      action = "Stop";
      break;
    case ROUGE :
      action = "Interdiction absolue de passer";
      break;
    default :
      action = "Passage avec priorité à droite";
    }
    System.out.println(action);
  }
```

Résultat :

```
Stop
```

Dans l'exemple ci-dessous l'absence de l'instruction break dans le cas VERT conduit à un résultat erroné.

3.10.4.2. L'utilisation de switch comme expression avec la syntaxe classique

Il est possible d'utiliser la syntaxe traditionnelle de l'instruction switch qui utilise le caractère deux point dans les clauses case dans une instruction switch utilisée comme une expression. Pour désigner la valeur retournée pour un cas, il faut utiliser l'instruction yield suivie de la valeur souhaitée.

Exemple (code Java 14) :

```
public class TestSwitch {

  public static void main(String[] args) {

    int valeur = 2;
    String libelle = switch(valeur) {
      case 1  : yield "un";
      case 2  : yield "deux";
      default : yield "hors limite";
    };
    System.out.println(libelle);
  }
}
```

Résultat :

```
deux
```

Une clause case peut contenir plusieurs instructions.

Exemple (code Java 14) :

```
    int    valeur = 2;
    String libelle   = switch (valeur) {
      case 1:
        System.out.println("cas 1");
        yield "un";
      case 2:
        System.out.println("cas2");
        yield "deux";
      default:
        yield "hors limite";
    };
    System.out.println(libelle);
```

Résultat :

```
cas 2
deux
```

Il est aussi possible d'utiliser un bloc de code dans une clause case.

Exemple (code Java 14) :

```
    int    valeur = 2;
    String libelle   = switch (valeur) {
      case 1: {
        System.out.println("cas 1");
        yield "un";
      }
      case 2: {
        System.out.println("cas 2");
        yield "deux";
      }
      default: {
        yield "hors limite";
      }
    };
    System.out.println(libelle);
```

Résultat :

```
cas 2
deux
```

L'utilisation d'un bloc de code permet comme tout bloc de code de regrouper plusieurs traitements qui seront exécutés si la valeur correspond au cas associé.

Il est possible de définir les mêmes variables dans plusieurs blocs puisqu'ils ne sont pas imbriqués.

Exemple (code Java 14) :

```
    int    valeur = 2;
    String libelle   = switch (valeur) {
      case 1: {
        String val = "un";
        yield val;
      }
      case 2: {
        String val = "deux";
        yield val;
      }
      default:
        yield "hors limite";
    };
    System.out.println(libelle);
```

Résultat :

```
deux
```

Attention, avec la syntaxe historique, le fall-through est toujours utilisé par défaut.

Exemple (code Java 14) :

```
    int    valeur = 2;
    String libelle   = switch (valeur) {
      case 1:
        System.out.println("cas 1");
        yield "un";
      case 2:
        System.out.println("cas 2");
      default:
        yield "hors limite";
    };
    System.out.println(libelle);
```

Résultat :

```
cas 2
hors limite
```

3.10.4.3. L'utilisation de switch comme expression avec l'opérateur arrow

L'utilisation de la syntaxe avec l'opérateur arrow facilite l'utilisation de l'instruction switch comme une expression : la valeur retournée est celle directement indiquée à la droite de l'opérateur arrow.

Exemple (code Java 14) :

```
    FeuTricolore feu    = VERT;
    String       action = switch (feu) {
      case VERT   -> "Passage avec priorité à droite";
      case ORANGE -> "Stop";
      case ROUGE  -> "Interdiction absolue de passer";
      default     -> "Passage avec priorité à droite";
    };
    System.out.println(action);
```

Résultat :

```
Passage avec priorité à droite
```

Attention, lorsque qu'une instruction switch est utilisée comme une expression, il faut obligatoirement terminer l'instruction par un point-virgule.

Exemple (code Java 14) :

```
public class TestSwitch {

  public static void main(String[] args) {
    FeuTricolore feu    = VERT;
    String       action = switch (feu) {
      case VERT   -> "Passage avec priorité à droite";
      case ORANGE -> "Stop";
      case ROUGE  -> "Interdiction absolue de passer";
      default     -> "Passage avec priorité à droite";
    }
    System.out.println(action);
  }
}
```

Résultat :

```
C:\java>javac TestSwitch.java
TestSwitch.java:11: error: ';' expected
    }
     ^
1 error
```

Une instruction switch utilisée comme une expression peut être utilisée partout où une expression du type retourné par le switch est attendue.

Exemple (code Java 14) :

```
    FeuTricolore feu = VERT;
    System.out.println(switch (feu) {
      case VERT -> "Passage avec priorité à droite";
      case ORANGE -> "Stop";
      case ROUGE -> "Interdiction absolue de passer";
      default -> "Passage avec priorité à droite";
    });
```

Résultat :

```
Passage avec priorité à droite
```

Lorsqu'une instruction switch est utilisée comme expression, elle doit toujours retourner une valeur ou lever une exception.

Exemple (code Java 14) :

```
    FeuTricolore feu = VERT;
    String action = switch (feu) {
      case VERT   -> "Passage avec priorité à droite";
      case ORANGE -> "Stop";
      case ROUGE  -> "Interdiction absolue de passer";
      default     -> throw new IllegalArgumentException("Feu tricolore défaillant");
    };
    System.out.println(action);
```

Si le code à exécuter contient plusieurs lignes, alors il faut utiliser un bloc de code. Dans ce cas, pour retourner une valeur, il faut utiliser une instruction yield.

Exemple (code Java 14) :

```
    FeuTricolore feu    = VERT;
    String       action = switch (feu) {
      case VERT   -> "Passage avec priorité à droite";
      case ORANGE -> "Stop";
      case ROUGE  -> "Interdiction absolue de passer";
      default     -> {
        System.out.println("cas par defaut");
        yield "Passage avec priorité à droite";
      }
    };
    System.out.println(action);
```

3.10.4.4. L'utilisation de switch comme structure de contrôle avec l'opérateur arrow

En utilisant la syntaxe avec l'opérateur arrow dans un switch comme structure de contrôle, il ne peut y avoir qu'une seule instruction.

Exemple (code Java 14) :

```
    int position = 1;
    switch (position) {
      case 1, 2, 3 -> System.out.println("Sur le podium");
      default      -> System.out.println("Hors podium");
    }
```

Résultat :

```
Sur le podium
```

Le compilateur émet une erreur si plusieurs instructions sont utilisées les unes à la suite des autres.

Exemple (code Java 14) :

```
    int position = 1;
    switch (position) {
      case 1, 2, 3 -> System.out.println("Sur le podium");
                      System.out.println("Avec medaille");
      default      -> System.out.println("Hors podium");
    }
```

Résultat :

```
C:\java> javac TestSwitch.java
TestSwitch.java:7: error: case, default, or '}' expected
                      System.out.println("Avec medaille");
                      ^
...
9 errors
```

Dans ce cas, il faut utiliser un bloc de code.

Exemple (code Java 14) :

```
    int position = 1;
    switch (position) {
      case 1, 2, 3 -> {
        System.out.println("Sur le podium");
        System.out.println("Avec medaille");
      }
      default      -> System.out.println("Hors podium");
    }
```

Dans le bloc de code, il est possible d'utiliser l'instruction break pour sortir du bloc et donc de l'instruction puisqu'il n'y a pas de fall-though.

Exemple (code Java 14) :

```
    int position = 1;
    switch (position) {
      case 1, 2, 3 -> {
        System.out.println("Sur le podium");
        System.out.println("Avec medaille");
        break;
      }
      default      -> System.out.println("Hors podium");
    }
```

En erreur est émise par le compilateur si des instructions suivent l'instruction break.

Exemple (code Java 14) :

```
    int position = 1;
    switch (position) {
      case 1, 2, 3 -> {
        System.out.println("Sur le podium");
        System.out.println("Avec medaille");
        break;
        System.out.println("");
      }
      default      -> System.out.println("Hors podium");
    }
```

Résultat :

```
C:\java> javac TestSwitch.java
TestSwitch.java:10: error: unreachable statement
        System.out.println("");
        ^
1 error
```

3.10.4.5. Le résumé des différentes formes d'utilisation

Il est possible de combiner les deux syntaxes de l'instruction switch pour une utilisation en tant que structure de contrôle ou comme expression : cela donne 4 formes d'utilisation. Des contraintes s'appliquent sur ces 4 formes ce qui rend leur utilisation plus complexe.

Le tableau ci-dessous résume les contraintes des différentes formes et cas d'utilisation :

	Quel que soit l'utilisation	Structure de contrôle	Expression
Quel que soit la syntaxe	Valeurs multiples possible dans les cases	default facultatif (pas d'exhaustivité) yield interdit	Exhaustivité obligatoire des valeurs à prendre en compte return interdit continue/break interdit

			L'instruction doit se terminer par un « ; » en fin de ligne
Syntaxe classique case X:	Fall-through par défaut Une seule portée pour les variables	Chaque case peut avoir zéro ou plusieurs instructions suivies selon les besoins d'un break return autorisé continue/break autorisé	yield pour retourner une valeur Chaque case peut avoir zéro ou plusieurs instructions mais doit finir par retourner une valeur avec yield ou lever une exception
Syntaxe avec l'opérateur arrow case X ->	Pas de fall-through Chaque case à sa propre portée	Chaque case doit avoir un traitement ou un bloc de code continue/break autorisé dans un bloc de code	Chaque case doit retourner une valeur yield pour retourner une valeur dans un bloc de code

3.10.5. Les situations illicites

Avec la nouvelle syntaxe et le nouveau rôle de l'instruction switch, plusieurs situations ne sont pas permises et provoqueront une erreur de la part du compilateur.

3.10.5.1. Des types différents retournés par les cases d'un switch comme expression

Lorsque le type de la variable d'affectation est déterminé, le compilateur vérifie que toutes les valeurs retournées par les clauses case sont bien du même type, quel que soit la syntaxe utilisée.

Exemple (code Java 14) :

```
public class TestSwitch {

  public static void main(String[] args) {
    int valeur = 2;
    String type = switch(valeur) {
      case 1  : yield "un";
      case 2  : yield 2;
      default : yield "hors limite";
    };
    System.out.println(type);
    System.out.println(type.getClass());
  }
}
```

Résultat :

```
C:\java> javac TestSwitch.java
TestSwitch.java:7: error: incompatible types: bad type in switch expression
      case 2  : yield 2;
                      ^
    int cannot be converted to String
1 error
```

Exemple (code Java 14) :

```
public class TestSwitch {
  public static void main(String[] args) {
    int valeur = 2;
    String type = switch(valeur) {
      case 1  -> "un";
      case 2  -> 2;
      default -> hors limite";
    };
    System.out.println(type);
    System.out.println(type.getClass());
  }
}
```

Résultat :

```
C:\java> javac TestSwitch.java
TestSwitch.java:7: error: incompatible types: bad type in switch expression
      case 2 -> 2;
                 ^
    int cannot be converted to String
1 error
```

Si le type de la variable n'est pas défini explicitement, une instruction switch peut retourner des valeurs de types différents. Dans cas, il faut impérativement déclarer la variable avec l'instruction var.

Exemple (code Java 14) :

```
public class TestSwitch {

  public static void main(String[] args) {
    int valeur = 2;
    var type = switch(valeur) {
      case 1  : yield "un";
      case 2  : yield 2;
      default : yield "hors limite";
    };
    System.out.println(type);
    System.out.println(type.getClass());
  }
}
```

Résultat :

```
C:\java> java TestSwitch
2
class java.lang.Integer
```

Attention : l'utilisation de cette fonctionnalité peut être source de difficultés et ne peut contourner les règles du typage statique de Java.

3.10.5.2. Dans un switch comme structure de contrôle, on ne peut pas retourner de valeur

Une erreur sera émise par le compilateur en cas de renvoie d'une valeur par l'instruction break dans un switch utilisée comme traitement.

Exemple (code Java 14) :

```
public class TestSwitch {

  public static void main(String[] args) {
    int valeur = 2;
    String type;
    switch(valeur) {
      case 1  : yield "un";
      case 2  : yield "deux";
      default : yield "hors limite";
    };
    System.out.println(type);
  }
}
```

Résultat :

```
C:\java> javac TestSwitch.java
TestSwitch.java:7: error: yield outside of switch expression
      case 1  : yield "un";
                ^
TestSwitch.java:8: error: yield outside of switch expression
      case 2  : yield "deux";
                ^
TestSwitch.java:9: error: yield outside of switch expression
      default : yield "hors limite";
                ^
3 errors
```

3.10.5.3. Dans un switch comme expression, il faut retourner une valeur après l'opérateur ->

Une erreur sera émise par le compilateur en cas de non renvoie d'une valeur par l'opérateur arrow « -> » dans une instruction switch utilisée comme une expression.

Exemple (code Java 14) :

```
public class TestSwitch {

  public static void main(String[] args) {
    int valeur = 2;
    String type = switch(valeur) {
      case 1  -> "un";
      case 2  -> "deux";
      default -> System.out.println("hors limite");
    };
    System.out.println(type);
  }
}
```

Résultat :

```
C:\java> javac TestSwitch.java
TestSwitch.java:8: error: incompatible types: bad type in switch expression
      default -> System.out.println("hors limite");
                                   ^
    void cannot be converted to String
1 error
```

3.10.5.4. Dans un switch comme traitement, l'opérateur -> ne peut pas retourner de valeur

Une erreur sera émise par le compilateur en cas d'utilisation du renvoie d'une valeur par l'opérateur arrow « -> » dans une instruction switch qui n'est pas utilisée comme une expression.

Exemple (code Java 14) :

```
public class TestSwitch {

  public static void main(String[] args) {
    int valeur = 2;
    String type;
    switch(valeur) {
      case 1  -> "un";
      case 2  -> "deux";
      default -> "hors limite";
    };
    System.out.println(type);
  }
}
```

Résultat :

```
C:\java> javac TestSwitch.java
TestSwitch.java:7: error: not a statement
      case 1  -> "un";
```

```
                 ^
TestSwitch.java:8: error: not a statement
      case 2  -> "deux";
                 ^
TestSwitch.java:9: error: not a statement
      default -> "hors limite";
                 ^
3 errors
```

3.10.5.5. Toutes les valeurs possibles doivent être prises en compte dans un switch utilisé comme expression

Toutes les valeurs possibles doivent être prises en compte explicitement et exhaustivement dans une instruction switch utilisée comme expression. Sauf si la valeur testée est une énumération et que toutes les valeurs sont utilisées dans une instruction case, sinon il faut utiliser une instruction default pour définir le cas par défaut qui concerne toutes les valeurs non explicitement définies dans un case.

Lors de l'utilisation d'une instruction switch comme une expression, le compilateur vérifie que toutes les valeurs possibles pour le type de la variable sont prises en compte dans une clause case. Si ce n'est pas le cas, le compilateur émet une erreur.

Exemple (code Java 14) :

```
public class TestSwitch {

  public static void main(String[] args) {
    int valeur = 2;
    String type = switch(valeur) {
      case 1 -> "un";
      case 2 -> "deux";
    };
    System.out.println(type);
  }
}
```

Résultat :

```
C:\java> javac TestSwitch.java
TestSwitch.java:5: error: the switch expression does not cover all possible input values
    String type = switch(valeur) {
                  ^
1 error
```

Prendre en compte toutes les valeurs possibles selon le type n'est pas toujours facile voire impossible : il est alors possible d'utiliser l'instruction default pour prendre en compte tous les autres cas.

Exemple (code Java 14) :

```
public class TestSwitch {

  public static void main(String[] args) {
    int valeur = 2;
    String type = switch(valeur) {
      case 1  -> "un";
      case 2  -> "deux";
      default -> "hors limite";
    };
    System.out.println(type);
  }
}
```

L'exhaustivité de la prise en compte des valeurs est obligatoire lors de l'utilisation de l'instruction switch comme une expression : cela concerne aussi l'utilisation comme une expression avec la syntaxe historique. D'ailleurs concrètement, cela signifie qu'il faut presque toujours définir une clause par défaut dans une instruction switch utilisée comme expression.

Exemple (code Java 14) :

```
public class TestSwitch {

  public static void main(String[] args) {
    int valeur = 2;
    String type = switch(valeur) {
      case 1  : yield "un";
      case 2  : yield "deux";
    };
    System.out.println(type);
  }
}
```

Résultat :

```
C:\java>javac TestSwitch.java
TestSwitch.java:7: error: the switch expression does not cover all possible input values
    String type = switch(valeur) {
                  ^
1 error
```

3.10.5.6. Il n'est pas possible de mixer les deux syntaxes d'une clause case

Une erreur sera émise par le compilateur en cas d'utilisation de la syntaxe historique utilisant « : » et la nouvelle syntaxe utilisant l'opérateur arrow « -> ».

Exemple (code Java 14) :

```
public class TestSwitch {

  public static void main(String[] args) {
    int valeur = 2;
    String type;
    switch(valeur) {
      case 1 -> type = "un";
      case 2  : type = "deux"; break;
      default : type = "hors limite"; break;
    };
    System.out.println(type);
  }
}
```

Résultat :

```
C:\java> javac TestSwitch.java
TestSwitch.java:8: error: different case kinds used in the switch
      case 2  : type = "deux"; break;
      ^
1 error
```

3.10.5.7. L'utilisation de l'instruction continue dans un switch

L'instruction continue peut être utilisée dans une instruction switch avec la syntaxe historique.

Exemple (code Java 14) :

```
public class TestSwitch {

  public static void main(String[] args) {
    for (int i = 1; i <= 5; i++) {
      System.out.println(i);
      switch (i) {
        case 1, 2, 3 :  System.out.println("Sur le podium");
        default : continue;
      }
    }
  }
}
```

Résultat :

```
C:\java>java TestSwitch
1
Sur le podium
2
Sur le podium
3
Sur le podium
4
5
```

Il n'est pas possible d'utiliser l'instruction continue dans un switch utilisé comme structure de contrôle utilisant la syntaxe avec l'opérateur arrow.

Exemple (code Java 14) :

```
public class TestSwitch {

  public static void main(String[] args) {
    for (int i = 1; i <= 5; i++) {
      System.out.println(i);
      switch (i) {
        case 1, 2, 3 -> System.out.println("Sur le podium");
        default -> continue;
      };
    }
  }
}
```

Résultat :

```
C:\java>javac TestSwitch.java
TestSwitch.java:10: error: unexpected statement in case, expected is an expression,
 a block or a throw statement
        default -> continue;
                   ^
1 error
```

Il est cependant possible d'utiliser l'instruction continue dans un bloc de code.

Exemple (code Java 14) :

```
public class TestSwitch {

  public static void main(String[] args) {
    for (int i = 1; i <= 5; i++) {
      System.out.println(i);
      switch (i) {
        case 1, 2, 3 -> System.out.println("Sur le podium");
        default -> { continue; }
      };
      System.out.println("suite");
    }
  }
}
```

Résultat :

```
C:\java>java
TestSwitch
1
Sur le podium
suite
```

```
2
Sur le podium
suite
3
Sur le podium
suite
4
5
```

Dans tous les cas, il n'est pas possible d'utiliser l'instruction continue lorsqu'une instruction switch est utilisée comme une expression.

Exemple (code Java 14) :

```java
public class TestSwitch {

  public static void main(String[] args) {
    for (int i = 1; i <= 5; i++) {
      System.out.println(i);
      String libelle = switch (i) {
        case 1, 2, 3 -> "Sur le podium";
        default -> continue;
      };
      System.out.println(libelle);
    }
  }
}
```

Résultat :

```
C:\java>javac TestSwitch.java
TestSwitch.java:10: error: illegal start of expression
        default -> continue;
                   ^
TestSwitch.java:10: error: case, default, or '}' expected
        default -> continue;
                           ^
2 errors
```

Ce n'est pas possible non plus dans un bloc de code.

Exemple (code Java 14) :

```java
public class TestSwitch {

  public static void main(String[] args) {
    for (int i = 1; i <= 5; i++) {
      System.out.println(i);
      String libelle = switch (i) {
        case 1, 2, 3 -> "Sur le podium";
        default -> { continue; }
      };
      System.out.println(libelle);
    }
  }
}
```

Résultat :

```
C:\java>javac TestSwitch.java
TestSwitch.java:10: error: attempt to continue out of a switch expression
        default -> { continue; }
                     ^
1 error
```

Il n'est pas possible non plus d'utiliser l'instruction continue dans un switch utilisé comme une expression avec la syntaxe historique.

Exemple (code Java 14) :

```
public class TestSwitch {

  public static void main(String[] args) {
    for (int i = 1; i <= 5; i++) {
      System.out.println(i);
      String libelle = switch (i) {
        case 1, 2, 3 :  yield "Sur le podium";
        default : continue;
      };
      System.out.println(libelle);
    }
  }
}
```

Résultat :

```
C:\java>javac TestSwitch.java
TestSwitch.java:10: error: attempt to continue out of a switch expression
        default -> { continue; }
                     ^
1 error
```

3.10.5.8. L'utilisation de l'instruction break dans un switch

Il n'est pas possible d'utiliser l'instruction continue dans un switch utilisé comme structure de contrôle utilisant la syntaxe avec l'opérateur arrow.

Exemple (code Java 14) :

```
public class TestSwitch {

  public static void main(String[] args) {
    for (int i = 1; i <= 5; i++) {
      System.out.println(i);
      switch (i) {
        case 1, 2, 3 -> System.out.println("Sur le podium");
        default -> break;
      };
    }
  }
}
```

Résultat :

```
C:\java>javac TestSwitch.java
TestSwitch.java:10: error: unexpected statement in case, expected is an expression,
 a block or a throw statement
        default -> break;
                   ^
1 error
```

Il est possible d'utiliser l'instruction break dans un bloc de code mais elle ne permet pas d'interrompre la boucle.

Exemple (code Java 14) :

```
public class TestSwitch {

  public static void main(String[] args) {
    for (int i = 1; i <= 5; i++) {
      System.out.println(i);
      switch (i) {
        case 1, 2, 3 -> System.out.println("Sur le podium");
        default -> { break; }
      };
      System.out.println("suite");
    }
    System.out.println("fin");
  }
}
```

Résultat :

```
C:\java>java
TestSwitch
Sur le podium
1
Sur le podium
2
Sur le podium
3
4
5
fin
```

Pour le faire, il est possible d'utiliser un break avec un étiquette dans un bloc de code. L'utilisation de cette forme de break n'est cependant pas recommandée.

Exemple (code Java 14) :

```
public class TestSwitch {

  public static void main(String[] args) {
    suite:
    for (int i = 1; i <= 5; i++) {
      System.out.println(i);
      switch (i) {
        case 1, 2, 3 -> System.out.println("Sur le podium");
        default -> { break suite; }
      };
      System.out.println("suite");
    }
    System.out.println("fin");
  }
}
```

Résultat :

```
C:\java>java TestSwitch
1
Sur le podium
suite
2
Sur le podium
suite
3
Sur le podium
suite
4
fin
```

Dans tous les cas, il n'est pas possible d'utiliser l'instruction break lorsqu'une instruction switch est utilisée comme une expression.

Exemple (code Java 14) :

```
public class TestSwitch {

  public static void main(String[] args) {
    for (int i = 1; i <= 5; i++) {
      System.out.println(i);
      String libelle = switch (i) {
        case 1, 2, 3 -> "Sur le podium";
        default -> break;
      };
          System.out.println(libelle);
    }
  }
}
```

Résultat :

```
C:\java>javac TestSwitch.java
TestSwitch.java:10: error: illegal start of expression
        default -> break;
                   ^
TestSwitch.java:10: error: case, default, or '}' expected
        default -> break;
                        ^
2 errors
```

Ce n'est pas possible non plus dans un bloc de code.

Exemple (code Java 14) :

```
public class TestSwitch {

  public static void main(String[] args) {
    for (int i = 1; i <= 5; i++) {
      System.out.println(i);
      String libelle = switch (i) {
        case 1, 2, 3 -> "Sur le podium";
        default -> { break; };
      };
      System.out.println(libelle);
    }
    System.out.println("fin");
  }
}
```

Résultat :

```
C:\java>javac TestSwitch.java
TestSwitch.java:10: error: case, default, or '}' expected
        default -> { break; };
                             ^
1 error
```

L'utilisation dans un bloc de code d'une instruction break avec une étiquette provoque la même erreur.

Il n'est pas possible non plus d'utiliser l'instruction break dans un switch utilisé comme une expression avec la syntaxe historique.

Exemple (code Java 14) :

```
public class TestSwitch {

  public static void main(String[] args) {

    for (int i = 1; i <= 5; i++) {
      System.out.println(i);
      String libelle = switch (i) {
        case 1, 2, 3 :  yield "Sur le podium";
        default : break;
      };
      System.out.println(libelle);
    }
  }
}
```

Résultat :

```
C:\java>javac TestSwitch.java
TestSwitch.java:10: error: attempt to break out of a switch expression
        default : break;
                  ^
1 error
```

La même erreur est émise avec un break avec etiquette.

Résultat :

```
C:\java>javac TestSwitch.java
TestSwitch.java:11: error: attempt to break out of a switch expression
        default : break suite;
                  ^
1 error
```

3.10.5.9. L'utilisation de return dans un switch comme expression

Il n'est pas possible d'utiliser l'instruction return dans un switch utilisé comme expression.

Ni avec la syntaxe historique

Exemple (code Java 14) :

```
public class TestSwitch {

  public static void main(String[] args) {
    int valeur = 2;
    String type = switch(valeur) {
      case 1  : yield "un";
      case 2  : return;
    };
    System.out.println(type);
  }
}
```

Résultat :

```
C:\java>javac TestSwitch.java
TestSwitch.java:9: error: attempt to return out of a switch expression
      case 2  : return;
                ^
1 error
```

Ni avec la syntaxe utilisant l'opérateur arrow.

Exemple (code Java 14) :

```
public class TestSwitch {

  public static void main(String[] args) {
    int valeur = 2;
    String type = switch(valeur) {
      case 1  -> "un";
      case 2  -> return;
    };
    System.out.println(type);
  }
}
```

Résultat :

```
C:\java>javac TestSwitch.java
TestSwitch.java:9: error: illegal start of expression
      case 2  -> return;
                 ^
TestSwitch.java:9: error: case, default, or '}' expected
      case 2 -> return;
                       ^
2 errors
```

3.11. Les tableaux

Les tableaux permettent de stocker un ensemble fini d'éléments d'un type particulier. L'accès à un élément particulier se fait grâce à son indice. Le premier élément d'un tableau possède l'indice 0.

Même si leur déclaration est spécifique, ce sont des objets : ils sont donc dérivés de la classe Object. Il est possible d'utiliser les méthodes héritées telles que equals() ou getClass().

3.11.1. La déclaration et l'allocation des tableaux

La déclaration d'un tableau à une dimension requiert un type, le nom de la variable permettant de faire référence au tableau et une paire de crochets. Java permet de placer les crochets sur le type ou sur le nom de la variable dans la déclaration.

L'allocation de la mémoire et donc l'obtention d'une référence pour accéder au tableau se fait en utilisation l'opérateur new, suivi d'une paire de crochets qui doit contenir le nombre maximum d'éléments que peut contenir le tableau.

La déclaration et l'allocation peut se faire sur une même ligne ou sur des lignes distinctes.

Exemple :

```
int tableau[] = new int[50]; // déclaration et allocation

// OU

int[] tableau = new int[50];

// OU

int tab[];         // déclaration
tab = new int[50]; // allocation
```

Pour passer un tableau à une méthode, il suffit de déclarer le paramètre dans la signature de la méthode

Exemple :

```
public void afficher(String texte[]){
  // ...
}
```

Les tableaux sont toujours transmis par référence puisque ce sont des objets.

Java ne supporte pas directement les tableaux à plusieurs dimensions : il faut déclarer un tableau de tableau en utilisant une paire de crochet pour chaque dimension.

Exemple :

```
float tableau[][] = new float[10][10];
```

La taille des tableaux de la seconde dimension peut ne pas être identique pour chaque occurrence.

Exemple :

```
int dim1[][] = new int[3][];
dim1[0]      = new int[4];
dim1[1]      = new int[9];
dim1[2]      = new int[2];
```

Chaque élément du tableau est initialisé selon son type par l'instruction new : 0 pour les numériques, '\0' pour les caractères, false pour les booléens et null pour les chaînes de caractères et les autres objets.

3.11.2. L'initialisation explicite d'un tableau

Exemple :

```
int tableau[5]    = {10, 20, 30, 40, 50};
int tableau[3][2] = {{5, 1}, {6, 2}, {7, 3}};
```

La taille du tableau n'est pas obligatoire si le tableau est initialisé à sa création.

Exemple :

```
int tableau[] = {10, 20, 30, 40, 50};
```

Le nombre d'éléments de chaque ligne peut ne pas être identique :

Exemple :

```
int[][] tabEntiers = {{1, 2, 3, 4, 5, 6},
                      {1, 2, 3, 4},
                      {1, 2, 3, 4, 5, 6, 7, 8, 9}};
```

3.11.3. Le parcours d'un tableau

La variable length retourne le nombre d'éléments du tableau. Il est alors possible d'utiliser une boucle pour itérer sur chacun des éléments du tableau.

Exemple :

```
for (int i = 0; i < tableau.length; i ++) {
  // ...
}
```

Un accès a un élément d'un tableau qui dépasse sa capacité, lève une exception du type java.lang.arrayIndexOutOfBoundsException.

A partir de Java 5, le parcours de l'intégralité des éléments d'un tableau peut être réalisé en utilisant la version améliorée de l'instruction for. La syntaxe générale est de la forme :

```
for ( type element_courant : type[] ) {
  // code à exécuter sur l'élément courant identifié par la variable element_courant
}
```

Exemple :

```
    String[] chaines = {"element1","element2","element3"};
    for (String chaine : chaines) {
      System.out.println(chaine);
    }
```

3.12. Les conversions de types

Lors de la déclaration, il est possible d'utiliser un cast :

Exemple :

```
int   entier   = 5;
float flottant = (float) entier;
```

La conversion peut entrainer une perte d'informations.

Il n'existe pas en Java de fonction pour convertir : les conversions de type se font par des méthodes. La bibliothèque de classes API fournit une série de classes qui contiennent des méthodes de manipulation et de conversion de types élémentaires.

Classe	Rôle
String	pour les chaînes de caractères Unicode
Integer	pour les valeurs entières (integer)
Long	pour les entiers longs signés (long)
Float	pour les nombres à virgule flottante (float)
Double	pour les nombres à virgule flottante en double précision (double)

Les classes portent le même nom que le type élémentaire sur lequel elles reposent avec la première lettre en majuscule.

Ces classes contiennent généralement plusieurs constructeurs ou des méthodes statiques pour obtenir des instances.

3.12.1. La conversion d'un entier int en chaîne de caractères String

Exemple :

```
int    i        = 10;
String montexte = new String();
montexte = montexte.valueOf(i);
```

Des surcharges de la méthode valueOf() sont également définies pour des arguments de type boolean, long, float, double et char

3.12.2. La conversion d'une chaîne de caractères String en entier int

Exemple :

```
String  montexte  = "10";
Integer monnombre = new Integer(montexte);
```

```
int     i         = monnombre.intValue(); // conversion d'Integer en int
```

3.12.3. La conversion d'un entier int en entier long

Exemple :

```
int     i         = 10;
Integer monnombre = new Integer(i);
long    j         = monnombre.longValue();
```

3.13. L'autoboxing et l'unboxing

L'autoboxing permet de transformer automatiquement une variable de type primitif en un objet du type du wrapper correspondant. L'unboxing est l'opération inverse. Cette nouvelle fonctionnalité est spécifiée dans la JSR 201 et intégrée dans Java 1.5.

Par exemple, jusqu'à la version 1.4 de Java pour ajouter des entiers dans une collection, il était nécessaire d'encapsuler chaque valeur dans un objet de type Integer.

Exemple :

```
import java.util.*;

public class TestAvantAutoboxing {

  public static void main(String[] args) {
    List liste = new ArrayList();
    Integer valeur = null;
    for(int i = 0; i < 10; i++) {
      valeur = new Integer(i);
      liste.add(valeur);
    }
  }
}
```

Avec Java 1.5, l'encapsulation de la valeur dans un objet n'est plus obligatoire car elle sera réalisée automatiquement par le compilateur.

Exemple (java 1.5) :

```
import java.util.*;

public class TestAutoboxing {

  public static void main(String[] args) {
    List liste = new ArrayList();
    for(int i = 0; i < 10; i++) {
      liste.add(i);
    }
  }
}
```

4. La programmation orientée objet

Chapitre 4

Niveau : 🟩🟩⬜⬜⬜⬜ Elémentaire

L'idée de base de la programmation orientée objet est de rassembler dans une même entité appelée objet les données et les traitements qui s'y appliquent.

Ce chapitre contient plusieurs sections :

- Le concept de classe : présente le concept et la syntaxe de la déclaration d'une classe
- Les objets : présente la création d'un objet, sa durée de vie, le clonage d'objets, les références et la comparaison d'objets, l'objet null, les variables de classes, la variable this et l'opérateur instanceof.
- Les modificateurs d'accès : présente les modificateurs d'accès des entités classes, méthodes et attributs ainsi que les mots clés qui permettent de qualifier ces entités
- Les propriétés ou attributs : présente les données d'une classe : les propriétés ou attributs
- Les méthodes : présente la déclaration d'une méthode, la transmission de paramètres, l'émission de messages, la surcharge, la signature d'une méthode et le polymorphisme et des méthodes particulières : les constructeurs, le destructeur et les accesseurs
- L'héritage : présente l'héritage : son principe, sa mise en oeuvre, ses conséquences. Il présente aussi la redéfinition d'une méthode héritée et les interfaces
- Les packages : présente la définition et l'utilisation des packages
- Les classes internes : présente une extension du langage Java qui permet de définir une classe dans une autre.
- Les types scellés
- La gestion dynamique des objets : présente rapidement la gestion dynamique des objets grâce à l'introspection

4.1. Le concept de classe

Une classe est le support de l'encapsulation : c'est un ensemble de données et de fonctions regroupées dans une même entité. Une classe est une description abstraite d'un objet. Les fonctions qui opèrent sur les données sont appelées des méthodes. Instancier une classe consiste à créer un objet sur son modèle. Entre classe et objet il y a, en quelque sorte, le même rapport qu'entre type et variable.

Java est un langage orienté objet : tout appartient à une classe sauf les variables de types primitives.

Pour accéder à une classe il faut en déclarer une instance de classe ou objet.

Une classe comporte sa déclaration, des variables et les définitions de ses méthodes.

Une classe se compose de deux parties : un en-tête et un corps. Le corps peut être divisé en 2 sections : la déclaration des données et des constantes et la définition des méthodes. Les méthodes et les données sont pourvues d'attributs de visibilité qui gèrent leur accessibilité par les composants hors de la classe.

4.1.1. La syntaxe de déclaration d'une classe

La syntaxe de déclaration d'une classe est la suivante :

modificateurs class nom_de_classe [extends classe_mere] [implements interfaces] { ... }

Les modificateurs de classe (ClassModifiers) sont :

Modificateur	Rôle
abstract	la classe contient une ou des méthodes abstraites, qui n'ont pas de définition explicite. Une classe déclarée abstract ne peut pas être instanciée : il faut définir une classe qui hérite de cette classe et qui implémente les méthodes nécessaires pour ne plus être abstraite.
final	la classe ne peut pas être modifiée, sa redéfinition grâce à l'héritage est interdite. Les classes déclarées final ne peuvent donc pas avoir de classes filles.
private	la classe n'est accessible qu'à partir du fichier où elle est définie
public	La classe est accessible partout

Les modificateurs abstract et final ainsi que public et private sont mutuellement exclusifs.

Marquer une classe comme final peut permettre au compilateur et à la JVM de réaliser quelques petites optimisations.

Le mot clé extends permet de spécifier une super-classe éventuelle : ce mot clé permet de préciser la classe mère dans une relation d'héritage.

Le mot clé implements permet de spécifier une ou des interfaces que la classe implémente. Cela permet de récupérer quelques avantages de l'héritage multiple.

L'ordre des méthodes dans une classe n'a pas d'importance. Si dans une classe, on rencontre d'abord la méthode A puis la méthode B, B peut être appelée sans problème dans A.

4.2. Les objets

Les objets contiennent des attributs et des méthodes. Les attributs sont des variables ou des objets nécessaires au fonctionnement de l'objet. En Java, une application est un objet. La classe est la description d'un objet. Un objet est une instance d'une classe. Pour chaque instance d'une classe, le code est le même, seules les données sont différentes à chaque objet.

4.2.1. La création d'un objet : instancier une classe

Il est nécessaire de définir la déclaration d'une variable ayant le type de l'objet désiré. La déclaration est de la forme nom_de_classe nom_de_variable

Exemple :

```
MaClasse m;
String chaine;
```

L'opérateur new se charge de créer une instance de la classe et de l'associer à la variable

Exemple :

```
m = new MaClasse();
```

Il est possible de tout réunir en une seule déclaration

Exemple :

```
MaClasse m = new MaClasse();
```

Chaque instance d'une classe nécessite sa propre variable. Plusieurs variables peuvent désigner un même objet.

En Java, tous les objets sont instanciés par allocation dynamique. Dans l'exemple, la variable m contient une référence sur l'objet instancié (contient l'adresse de l'objet qu'elle désigne : attention toutefois, il n'est pas possible de manipuler ou d'effectuer des opérations directement sur cette adresse comme en C).

Si m2 désigne un objet de type MaClasse, l'instruction m2 = m ne définit pas un nouvel objet mais m et m2 désignent tous les deux le même objet.

L'opérateur new est un opérateur de haute priorité qui permet d'instancier des objets et d'appeler une méthode particulière de cet objet : le constructeur. Il fait appel à la machine virtuelle pour obtenir l'espace mémoire nécessaire à la représentation de l'objet puis appelle le constructeur pour initialiser l'objet dans l'emplacement obtenu. Il renvoie une valeur qui référence l'objet instancié.

Si l'opérateur new n'obtient pas l'allocation mémoire nécessaire, il lève l'exception OutOfMemoryError.

4.2.2. La durée de vie d'un objet

Les objets ne sont pas des éléments statiques et leur durée de vie ne correspond pas forcément à la durée d'exécution du programme.

La durée de vie d'un objet passe par trois étapes :

- la déclaration de l'objet et l'instanciation grâce à l'opérateur new

  Exemple :

  ```
  nom_de_classe nom_d_objet = new nom_de_classe( ... );
  ```

- l'utilisation de l'objet en appelant ses méthodes
- la suppression de l'objet : elle est automatique en Java grâce à la machine virtuelle. La restitution de la mémoire inutilisée est prise en charge par le récupérateur de mémoire (garbage collector). Il n'existe pas d'instruction delete comme en C++.

4.2.3. Les références et la comparaison d'objets

Les variables de type objet que l'on déclare ne contiennent pas un objet mais une référence vers cet objet. Lorsque l'on écrit c1 = c2 (c1 et c2 sont des objets), on copie la référence de l'objet c2 dans c1. c1 et c2 font référence au même objet : ils pointent sur le même objet. L'opérateur == compare ces références. Deux objets avec des propriétés identiques sont deux objets distincts :

Exemple :

```
Rectangle r1 = new Rectangle(100,50);
Rectangle r2 = new Rectangle(100,50);
if (r1 == r1) { ... } // vrai
if (r1 == r2) { ... } // faux
```

Pour comparer l'égalité des variables de deux instances, il faut munir la classe d'une méthode à cet effet : la méthode equals() héritée de Object.

Pour s'assurer que deux objets sont de la même classe, il faut utiliser la méthode getClass() de la classe Object dont toutes les classes héritent.

Exemple :

```
(obj1.getClass().equals(obj2.getClass())
```

4.2.4. Le littéral null

Le littéral null est utilisable partout où il est possible d'utiliser une référenc à un objet. Il n'appartient pas à une classe mais il peut être utilisé à la place d'un objet de n'importe quelle type ou comme paramètre. null ne peut pas être utilisé comme un objet normal : il n'y a pas d'appel de méthodes et aucune classe ne peut en hériter.

Le fait d'affecter null une variable référençant un objet pourra permettre au ramasse-miettes de libérer la mémoire allouée à l'objet si aucune autre référence n'existe encore sur lui.

4.2.5. Les variables de classes

Elles ne sont définies qu'une seule fois quel que soit le nombre d'objets instanciés de la classe. Leur déclaration est accompagnée du mot clé static

Exemple :

```
public class MaClasse() {
    static int compteur = 0;
}
```

L'appartenance des variables de classe à une classe entière et non à un objet spécifique permet de remplacer le nom de la variable par le nom de la classe.

Exemple :

```
MaClasse m = new MaClasse();
int c1 = m.compteur;
int c2 = MaClasse.compteur;
// c1 et c2 possèdent la même valeur.
```

Ce type de variable est utile pour, par exemple, compter le nombre d'instanciations de la classe.

4.2.6. La variable this

Cette variable sert à référencer dans une méthode l'instance de l'objet en cours d'utilisation. this est un objet qui est égal à l'instance de l'objet dans lequel il est utilisé.

Exemple :

```
private int nombre;
public maclasse(int nombre) {
   nombre = nombre; // variable de classe = variable en paramètre du constructeur
}
```

Il est préférable de préfixer la variable d'instance par le mot clé this.

Exemple :

```
this.nombre = nombre;
```

Cette référence est habituellement implicite :

Exemple :

```
class MaClasse() {
  String chaine = " test " ;
  public String getChaine() { return chaine; }
  // est équivalent à public String getChaine() { return this.chaine; }
}
```

This est aussi utilisé quand l'objet doit appeler une méthode en se passant lui-même en paramètre de l'appel.

4.2.7. L'opérateur instanceof

L'opérateur instanceof permet de déterminer la classe de l'objet qui lui est passé en paramètre. La syntaxe est objet instanceof classe

Exemple :

```
void testClasse(Object o) {
   if (o instanceof MaClasse )
      System.out.println(" o est une instance de la classe MaClasse ");
   else System.out.println(" o n'est pas un objet de la classe MaClasse ");
}
```

Dans le cas ci-dessus, même si o est une instance de MaClasse, il n'est pas permis d'appeler une méthode de MaClasse car o est de type Objet.

Exemple :

```
void afficheChaine(Object o) {
    if (o instanceof MaClasse)
        System.out.println(o.getChaine());
        // erreur à la compil car la méthode getChaine()
        //n'est pas définie dans la classe Object
}
```

Pour résoudre le problème, il faut utiliser la technique du casting (conversion).

Exemple :

```
void afficheChaine(Object o) {
   if (o instanceof MaClasse) {
      MaClasse m = (MaClasse) o;
      System.out.println(m.getChaine());
      // OU System.out.println( ((MaClasse) o).getChaine() );
   }
}
```

4.3. Les modificateurs d'accès

Ils s'appliquent aux classes, aux méthodes et aux attributs.

Ils ne peuvent pas être utilisés pour qualifier des variables locales : seules les variables d'instances et de classes peuvent en profiter.

Ils assurent le contrôle des conditions d'héritage, d'accès aux éléments et de modification de données par les autres objets.

4.3.1. Les mots clés qui gèrent la visibilité des entités

De nombreux langages orientés objet introduisent des attributs de visibilité pour réglémenter l'accès aux classes et aux objets, aux méthodes et aux données.

En plus de la valeur par défaut, il existe 3 modificateurs explicites qui peuvent être utilisés pour définir les attributs de visibilité des entités (classes, méthodes ou attributs) : public, private et protected. Leur utilisation permet de définir des niveaux de protection différents (présentés dans un ordre croissant de niveau de protection offert) :

Modificateur	Rôle
public	Une variable, méthode ou classe déclarée public est visible par tous les autres objets. Depuis la version 1.0, une seule classe public est permise par fichier et son nom doit correspondre à celui du fichier. Dans la philosophie orientée objet aucune donnée d'une classe ne devrait être déclarée publique : il est préférable d'écrire des méthodes pour la consulter et la modifier
par défaut : package-private	Il n'existe pas de mot clé pour définir ce niveau, qui est le niveau par défaut lorsqu'aucun modificateur n'est précisé. Cette déclaration permet à une entité (classe, méthode ou variable) d'être visible par toutes les classes se trouvant dans le même package.
protected	Si une méthode ou une variable est déclarée protected, seules les méthodes présentes dans le même package que cette classe ou ses sous-classes pourront y accéder. On ne peut pas qualifier une classe avec protected.
private	C'est le niveau de protection le plus fort. Les composants ne sont visibles qu'à l'intérieur de la classe : ils ne peuvent être modifiés que par des méthodes définies dans la classe et prévues à cet effet. Les méthodes déclarées private ne peuvent pas être en même temps déclarées abstract car elles ne peuvent pas être redéfinies dans les classes filles.

Ces modificateurs d'accès sont mutuellement exclusifs.

4.3.2. Le mot clé static

Le mot clé static s'applique aux variables et aux méthodes.

Les variables d'instance sont des variables propres à un objet. Il est possible de définir une variable de classe qui est partagée entre toutes les instances d'une même classe : elle n'existe donc qu'une seule fois en mémoire. Une telle variable permet de stocker une constante ou une valeur modifiée tour à tour par les instances de la classe. Elle se définit avec le mot clé static.

Exemple :

```
public class Cercle {
  static float pi = 3.1416f;
  float rayon;
  public Cercle(float rayon) { this.rayon = rayon; }
  public float surface() { return rayon * rayon * pi;}
}
```

Il est aussi possible par exemple de mémoriser les valeurs min et max d'un ensemble d'objets de même classe.

Une méthode static est une méthode qui n'agit pas sur des variables d'instance mais uniquement sur des variables de classe. Ces méthodes peuvent être utilisées sans instancier un objet de la classe. Les méthodes ainsi définies peuvent être appelées avec la notation classe.methode() au lieu de objet.methode() : la première forme est fortement recommandée pour éviter toute confusion.

Il n'est pas possible d'appeler une méthode d'instance ou d'accéder à une variable d'instance à partir d'une méthode de classe statique.

4.3.3. Le mot clé final

Le mot clé final s'applique aux variables de classe ou d'instance ou locales, aux méthodes, aux paramètres d'une méthode et aux classes. Il permet de rendre l'entité sur laquelle il s'applique non modifiable une fois qu'elle est déclarée pour une méthode ou une classe et initialisée pour une variable.

Une variable qualifiée de final signifie que la valeur de la variable ne peut plus être modifiée une fois que celle-ci est initialisée.

Exemple :

```java
package fr.jmdoudoux.dej;

public class Constante2 {

  public final int constante;

  public Constante2() {
    this.constante = 10;
  }
}
```

Une fois la variable déclarée final initialisée, il n'est plus possible de modifier sa valeur. Une vérification est opérée par le compilateur.

Exemple :

```java
package fr.jmdoudoux.dej;

public class Constante1 {

  public static final int constante = 0;

  public Constante1() {
    this.constante = 10;
  }
}
```

Résultat :

```
C:\>javac Constante1.java
Constante1.java:6: cannot assign a value to final variable constante
    this.constante = 10;
    ^
1 error
```

Les constantes sont qualifiées avec les modificateurs final et static.

Exemple :

```java
public static final float PI = 3.141f;
```

Une méthode déclarée final ne peut pas être redéfinie dans une sous-classe. Une méthode possédant le modificateur final pourra être optimisée par le compilateur car il est garanti qu'elle ne sera pas sous-classée.

Lorsque le modificateur final est ajouté à une classe, il est interdit de créer une classe qui en hérite.

Pour une méthode ou une classe, on renonce à l'héritage mais ceci peut s'avérer nécessaire pour des questions de sécurité ou de performance. Le test de validité de l'appel d'une méthode est bien souvent repoussé à l'exécution, en fonction du type de l'objet appelé (c'est la notion de polymorphisme qui sera détaillée ultérieurement). Ces tests ont un coût en termes de performance.

Quatre types de variables sont implicitement déclarés final :

- un champ d'une interface
- une variable locale déclarée comme ressource d'une instruction try-with-resources
- un paramètre d'exception d'une clause multi-catch
- un champ correspondant à un composant d'un record

Remarque : un unique paramètre d'exception d'une clause catch n'est jamais déclaré final implicitement, mais peut être effectivement final.

4.3.4. Le mot clé abstract

Le mot clé abstract s'applique aux méthodes et aux classes.

Abstract indique que la classe ne pourra être instanciée telle quelle. De plus, toutes les méthodes de cette classe abstract ne sont pas implémentées et devront être redéfinies par des méthodes complètes dans ses sous-classes.

Abstract permet de créer une classe qui sera une sorte de moule. Toutes les classes dérivées pourront profiter des méthodes héritées et n'auront à implémenter que les méthodes déclarées abstract.

Exemple :

```
abstract class ClasseAbstraite {
  ClasseAbstraite() { ... //code du constructeur }
  void methode() { ... // code partagé par tous les descendants }
  abstract void methodeAbstraite();
}

class ClasseComplete extends ClasseAbstraite {
  ClasseComplete() { super(); ... }
  void methodeAbstraite() { ... // code de la méthode }
  // void methode est héritée
}
```

Une méthode abstraite est une méthode déclarée avec le modificateur abstract et sans corps. Elle correspond à une méthode dont on veut forcer l'implémentation dans une sous-classe. L'abstraction permet une validation du codage : une sous-classe sans le modificateur abstract et sans définition explicite d'une ou des méthodes abstraites génère une erreur de compilation.

Une classe est automatiquement abstraite dès qu'une de ses méthodes est déclarée abstraite. Il est possible de définir une classe abstraite sans méthodes abstraites.

4.3.5. Le mot clé synchronized

Il permet de gérer l'accès concurrent aux variables et méthodes lors de traitements de threads (exécution « simultanée » de plusieurs petites parties de code du programme)

4.3.6. Le mot clé volatile

Le mot clé volatile s'applique aux variables.

Il précise que la variable peut être changée par un périphérique ou de manière asynchrone. Cela indique au compilateur de ne pas stocker cette variable dans des registres. A chaque utilisation, sa valeur est lue et réécrite immédiatement si elle a changé.

4.3.7. Le mot clé native

Une méthode native est une méthode qui est implémentée dans un autre langage. L'utilisation de ce type de méthode limite la portabilité du code mais permet une vitesse d'exécution plus rapide.

4.4. Les propriétés ou attributs

Les données d'une classe sont contenues dans des variables nommées propriétés ou attributs. Ce sont des variables qui peuvent être des variables d'instances, des variables de classes ou des constantes.

4.4.1. Les variables d'instances

Une variable d'instance nécessite simplement une déclaration de la variable dans le corps de la classe.

Exemple :

```
public class MaClasse {
  public int valeur1 ;
  int valeur2 ;
  protected int valeur3 ;
  private int valeur4 ;
}
```

Chaque instance de la classe a accès à sa propre occurrence de la variable.

4.4.2. Les variables de classes

Les variables de classes sont définies avec le mot clé static

Exemple (code Java 1.1) :

```
public class MaClasse {
   static int compteur ;
}
```

Chaque instance de la classe partage la même variable.

4.4.3. Les constantes

Les constantes sont définies avec le mot clé final : leur valeur ne peut pas être modifiée une fois qu'elles sont initialisées.

Exemple (code Java 1.1) :

```
public class MaClasse {
   final double pi=3.14 ;
}
```

4.5. Les méthodes

Les méthodes sont des fonctions qui implémentent les traitements de la classe.

4.5.1. La syntaxe de la déclaration

La syntaxe de la déclaration d'une méthode est :

modificateurs type_retourné nom_méthode (arg1, ...) {... } // définition des variables locales et du bloc d'instructions

Le type retourné peut être élémentaire ou correspondre à un objet. Si la méthode ne retourne rien, alors on utilise void.

Le type et le nombre d'arguments déclarés doivent correspondre au type et au nombre d'arguments transmis. Il n'est pas possible d'indiquer des valeurs par défaut dans les paramètres. Les arguments sont passés par valeur : la méthode fait une copie de la variable qui lui est locale. Lorsqu'un objet est transmis comme argument à une méthode, cette dernière reçoit une référence qui désigne son emplacement mémoire d'origine et qui est une copie de la variable. Il est possible de modifier l'objet grâce à ses méthodes mais il n'est pas possible de remplacer la référence contenue dans la variable passée en paramètre : ce changement n'aura lieu que localement à la méthode.

Les modificateurs de méthodes sont :

Modificateur	Rôle
public	la méthode est accessible aux méthodes des autres classes
private	l'usage de la méthode est réservé aux autres méthodes de la même classe
protected	la méthode ne peut être invoquée que par des méthodes de la classe ou de ses sous-classes
final	la méthode ne peut être modifiée (redéfinition lors de l'héritage interdite)
static	la méthode appartient simultanément à tous les objets de la classe (comme une constante déclarée à l'intérieur de la classe). Il est inutile d'instancier la classe pour appeler la méthode mais la méthode ne peut pas manipuler de variable d'instance. Elle ne peut utiliser que des variables de classes.
synchronized	la méthode fait partie d'un thread. Lorsqu'elle est appelée, elle barre l'accès à son instance. L'instance est à nouveau libérée à la fin de son exécution.
native	le code source de la méthode est écrit dans un autre langage

Sans modificateur, la méthode peut être appelée par toutes autres méthodes des classes du package auquel appartient la classe.

La valeur de retour de la méthode doit être transmise par l'instruction return. Elle indique la valeur que prend la méthode et termine celle-ci : toutes les instructions qui suivent return sont donc ignorées.

Exemple :

```
int add(int a, int b) {
   return a + b;
}
```

Il est possible d'inclure une instruction return dans une méthode de type void : cela permet de quitter la méthode.

La méthode main() de la classe principale d'une application doit être déclarée de la façon suivante : public static void main (String args[]) { ... }

Exemple :

```
public class MonApp1 {

  public static void main(String[] args) {
    System.out.println("Bonjour");
  }
}
```

Cette déclaration de la méthode main() est imposée par la machine virtuelle pour reconnaitre le point d'entrée d'une application. Si la déclaration de la méthode main() diffère, une exception sera levée lors de la tentative d'exécution par la machine virtuelle.

Exemple :

```
public class MonApp2 {

  public static int main(String[] args) {
    System.out.println("Bonjour");
    return 0;
  }
}
```

Résultat :

```
C:\>javac MonApp2.java

C:\>java MonApp2
Exception in thread "main" java.lang.NoSuchMethodError: main
```

Si la méthode retourne un tableau alors les caractères [] peuvent être précisés après le type de retour ou après la liste des paramètres :

Exemple :

```
int[] getValeurs() { ... }
int getValeurs()[] { ... }
```

4.5.2. La transmission de paramètres

Lorsqu'un objet est passé en paramètre, ce n'est pas l'objet lui-même qui est passé mais une référence sur l'objet. La référence est bien transmise par valeur et ne peut pas être modifiée mais l'objet peut être modifié par un message (appel d'une méthode).

Pour transmettre des arguments par référence à une méthode, il faut les encapsuler dans un objet qui prévoit les méthodes nécessaires pour les mises à jour.

Si un objet o transmet sa variable d'instance v en paramètre à une méthode m, deux situations sont possibles :

- si v est une variable primitive alors elle est passée par valeur : il est impossible de la modifier dans m pour que v en retour contienne cette nouvelle valeur.
- si v est un objet alors m pourra modifier l'objet en utilisant une méthode de l'objet passé en paramètre.

4.5.3. L'émission de messages

Un message est émis lorsqu'on demande à un objet d'exécuter l'une de ses méthodes.

La syntaxe d'appel d'une méthode est : nom_objet.nom_méthode(parametre, ...) ;

Si la méthode appelée ne contient aucun paramètre, il faut laisser les parenthèses vides.

4.5.4. L'enchaînement de références à des variables et à des méthodes

Exemple :

```
System.out.println("bonjour");
```

Deux classes sont impliquées dans l'instruction : System et PrintStream. La classe System possède une variable nommée out qui est un objet de type PrintStream. Println() est une méthode de la classe PrintStream. L'instruction signifie : « utilise la méthode println() de la variable out de la classe System ».

4.5.5. Les arguments variables (varargs)

Depuis Java 1.5, les varargs, spécifiés dans la JSR 201, permettent de passer un nombre non défini d'arguments d'un même type à une méthode. Ceci va éviter de devoir encapsuler ces données dans une collection.

Elle utilise un notation pour préciser la répétition d'un type d'argument utilisant trois petits points : ...

Exemple (java 1.5) :

```
public class TestVarargs {

  public static void main(String[] args) {
    System.out.println("valeur 1 = " + additionner(1,2,3));
    System.out.println("valeur 2 = " + additionner(2,5,6,8,10));
  }

  public static int additionner(int ... valeurs) {
    int total = 0;

    for (int val : valeurs) {
      total += val;
        }

    return total;
  }
}
```

Résultat :

```
C:\tiger>java TestVarargs
valeur 1 = 6
valeur 2 = 31
```

L'utilisation de la notation ... permet le passage d'un nombre indéfini de paramètres du type précisé. Tous ces paramètres sont traités comme un tableau : il est d'ailleurs possible de fournir les valeurs sous la forme d'un tableau.

Exemple (java 1.5) :

```
public class TestVarargs2 {

  public static void main(String[] args) {
    int[] valeurs = {1,2,3,4};
    System.out.println("valeur 1 = " + additionner(valeurs));
  }

  public static int additionner(int ... valeurs) {
    int total = 0;

        for (int val : valeurs) {
      total += val;
        }

        return total;
  }
}
```

Résultat :

```
C:\tiger>java TestVarargs2
valeur 1 = 10
```

Il n'est cependant pas possible de mixer des éléments unitaires et un tableau dans la liste des éléments fournis en paramètres.

Exemple (java 1.5) :

```
public class TestVarargs3 {

  public static void main(String[] args) {
    int[] valeurs = {1,2,3,4};
    System.out.println("valeur 1 = " + additionner(5,6,7,valeurs));
  }

  public static int additionner(int ... valeurs) {
    int total = 0;

    for (int val : valeurs) {
      total += val;
    }

    return total;
  }
}
```

Résultat :

```
C:\tiger>javac -source 1.5 -target 1.5 TestVarargs3.java
TestVarargs3.java:7: additionner(int[]) in TestVarargs3 cannot be applied to (in
t,int,int,int[])
    System.out.println("valeur 1 = " + additionner(5,6,7,valeurs));
                                       ^
1 error
```

4.5.6. La surcharge de méthodes

La surcharge d'une méthode permet de définir plusieurs fois une même méthode avec des arguments différents. Le compilateur choisi la méthode qui doit être appelée en fonction du nombre et du type des arguments. Ceci permet de simplifier l'interface des classes vis à vis des autres classes.

Une méthode est surchargée lorsqu'elle exécute des actions différentes selon le type et le nombre de paramètres transmis.

Il est donc possible de donner le même nom à deux méthodes différentes à condition que les signatures de ces deux méthodes soient différentes. La signature d'une méthode comprend le nom de la classe, le nom de la méthode et les types des paramètres.

Exemple :

```
class affiche{
    public void afficheValeur(int i) {
        System.out.println(" nombre entier = " + i);
    }

    public void afficheValeur(float f) {
        System.out.println(" nombre flottant = " + f);
    }
}
```

Il n'est pas possible d'avoir deux méthodes de même nom dont tous les paramètres sont identiques et dont seul le type retourné diffère.

Exemple :

```
class Affiche{

    public float convert(int i){
         return((float) i);
    }

    public double convert(int i){
        return((double) i);
    }
}
```

Résultat :

```
C:\>javac Affiche.java
Affiche.java:5: Methods can't be redefined with a different return type: double
convert(int) was float convert(int)
public double convert(int i){
       ^
1 error
```

4.5.7. Les constructeurs

La déclaration d'un objet est suivie d'une sorte d'initialisation par le moyen d'une méthode particulière appelée constructeur pour que les variables aient une valeur de départ. Elle n'est systématiquement invoquée que lors de la création d'un objet.

Le constructeur suit la définition des autres méthodes excepté que son nom doit obligatoirement correspondre à celui de la classe et qu'il n'est pas typé, pas même void, donc il ne peut pas y avoir d'instruction return dans un constructeur. On peut surcharger un constructeur.

La définition d'un constructeur est facultative. Si aucun constructeur n'est explicitement défini dans la classe, le compilateur va créer un constructeur par défaut sans argument. Dès qu'un constructeur est explicitement défini, le compilateur considère que le programmeur prend en charge la création des constructeurs et que le mécanisme par défaut, qui correspond à un constructeur sans paramètres, n'est pas mis en oeuvre. Si on souhaite maintenir ce mécanisme, il faut définir explicitement un constructeur sans paramètres en plus des autres constructeurs.

Il existe plusieurs manières de définir un constructeur :

1. le constructeur simple : ce type de constructeur ne nécessite pas de définition explicite : son existence découle automatiquement de la définition de la classe.

   Exemple :

   ```
   public MaClasse() {}
   ```

2. le constructeur avec initialisation fixe : il permet de créer un constructeur par défaut

Exemple :

```
public MaClasse() {
    nombre = 5;
}
```

3. le constructeur avec initialisation des variables : pour spécifier les valeurs de données à initialiser on peut les passer en paramètres au constructeur

Exemple :

```
public MaClasse(int valeur) {
     nombre = valeur;
}
```

4.5.8. Les accesseurs

L'encapsulation permet de sécuriser l'accès aux données d'une classe. Ainsi, les données déclarées private à l'intérieur d'une classe ne peuvent être accédées et modifiées que par des méthodes définies dans la même classe. Si une autre classe veut accéder aux données de la classe, l'opération n'est possible que par l'intermédiaire d'une méthode de la classe prévue à cet effet. Ces appels de méthodes sont appelés « échanges de messages ».

Un accesseur est une méthode publique qui donne l'accès à une variable d'instance privée. Pour une variable d'instance, il peut ne pas y avoir d'accesseur, un seul accesseur en lecture ou un accesseur en lecture et un autre en écriture. Par convention, les accesseurs en lecture commencent par get et les accesseurs en écriture commencent par set.

Exemple :

```
private int valeur = 13;

public int getValeur(){
    return(valeur);
}

public void setValeur(int val) {
    valeur = val;
}
```

Pour un attribut de type booléen, il est possible de faire commencer l'accesseur en lecture par is au lieu de get.

4.6. L'héritage

L'héritage est un mécanisme qui facilite la réutilisation du code et la gestion de son évolution. Elle définit une relation entre deux classes :

- une classe mère ou super-classe
- une classe fille ou sous-classe qui hérite de sa classe mère

4.6.1. Le principe de l'héritage

Grâce à l'héritage, les objets d'une classe fille ont accès aux données et aux méthodes de la classe parente et peuvent les étendre. Les sous-classes peuvent redéfinir les variables et les méthodes héritées. Pour les variables, il suffit de les redéclarer sous le même nom avec un type différent. Les méthodes sont redéfinies avec le même nom, les mêmes types et le même nombre d'arguments, sinon il s'agit d'une surcharge.

L'héritage successif de classes permet de définir une hiérarchie de classe qui se compose de super-classes et de sous-classes. Une classe qui hérite d'une autre est une sous-classe et celle dont elle hérite est une super-classe. Une classe peut avoir plusieurs sous-classes. Une classe ne peut avoir qu'une seule classe mère : il n'y a pas d'héritage multiple en Java.

Object est la classe parente de toutes les classes en Java. Toutes les variables et méthodes contenues dans Object sont accessibles à partir de n'importe quelle classe car par héritages successifs toutes les classes héritent d'Object.

4.6.2. La mise en oeuvre de l'héritage

On utilise le mot clé extends pour indiquer qu'une classe hérite d'une autre. En l'absence de ce mot réservé associé à une classe, le compilateur considère la classe Object comme classe mère.

Exemple :

```
class Fille extends Mere { ... }
```

Pour invoquer une méthode d'une classe mère, il suffit d'indiquer la méthode préfixée par super. Pour appeler le constructeur de la classe mère, il suffit d'écrire super(paramètres) avec les paramètres adéquats.

Le lien entre une classe fille et une classe mère est géré par la plate-forme : une évolution des règles de gestion de la classe mère conduit à modifier automatiquement la classe fille dès que cette dernière est recompilée.

En Java, il est obligatoire dans un constructeur d'une classe fille de faire appel explicitement ou implicitement au constructeur de la classe mère.

4.6.3. L'accès aux propriétés héritées

Les variables et méthodes définies avec le modificateur d'accès public restent publiques à travers l'héritage et toutes les autres classes.

Une variable d'instance définie avec le modificateur private est bien héritée mais elle n'est pas accessible directement mais par les méthodes héritées.

Une variable définie avec le modificateur protected sera héritée dans toutes les classes filles qui pourront y accéder librement ainsi que les classes du même package.

4.6.4. La redéfinition d'une méthode héritée

La redéfinition d'une méthode héritée doit impérativement conserver la déclaration de la méthode parente (type et nombre de paramètres, la valeur de retour et les exceptions propagées doivent être identiques).

Si la signature de la méthode change, ce n'est plus une redéfinition mais une surcharge. Cette nouvelle méthode n'est pas héritée : la classe mère ne possède pas de méthode possédant cette signature.

4.6.5. Le polymorphisme

Le polymorphisme est la capacité, pour un même message de correspondre à plusieurs formes de traitements selon l'objet auquel ce message est adressé. La gestion du polymorphisme est assurée par la machine virtuelle dynamiquement à l'exécution.

4.6.6. Le transtypage induit par l'héritage facilite le polymorphisme

L'héritage définit un cast implicite de la classe fille vers la classe mère : on peut affecter à une référence d'une classe n'importe quel objet d'une de ses sous-classes.

Exemple : la classe Employe hérite de la classe Personne

```
    Personne p = new Personne ("Dupond", "Jean");
    Employe e = new Employe("Durand", "Julien", 10000);
    p = e ; // ok : Employe est une sous-classe de Personne
    Objet obj;
    obj = e ; // ok : Employe hérite de Personne qui elle même hérite de Object
```

Il est possible d'écrire le code suivant si Employe hérite de Personne

Exemple :

```
    Personne[] tab = new Personne[10];
    tab[0] = new Personne("Dupond","Jean");
    tab[1] = new Employe("Durand", "Julien", 10000);
```

Il est possible de surcharger une méthode héritée : la forme de la méthode à exécuter est choisie en fonction des paramètres associés à l'appel.

Compte tenu du principe de l'héritage, le temps d'exécution du programme et la taille du code source et de l'exécutable augmentent.

4.6.7. Les interfaces et l'héritage multiple

Avec l'héritage multiple, une classe peut hériter en même temps de plusieurs super-classes. Ce mécanisme n'existe pas en Java. Les interfaces permettent de mettre en oeuvre un mécanisme de remplacement.

Une interface est un ensemble de constantes et de déclarations de méthodes correspondant un peu à une classe abstraite. C'est une sorte de standard auquel une classe peut répondre. Tous les objets qui se conforment à cette interface (qui implémentent cette interface) possèdent les méthodes et les constantes déclarées dans celle-ci. Plusieurs interfaces peuvent être implémentées dans une même classe.

Les interfaces se déclarent avec le mot clé interface et sont intégrées aux autres classes avec le mot clé implements. Une interface est implicitement déclarée avec le modificateur abstract.

Déclaration d'une interface :

```
[public] interface nomInterface [extends nomInterface1, nomInterface2 ... ] {
    // insérer ici des méthodes ou des champs static
}
```

Implémentation d'une interface :

```
Modificateurs class nomClasse [extends superClasse]
   [implements nomInterface1, nomInterface 2, ...] {
    //insérer ici des méthodes et des champs

}
```

Exemple :

```
interface AfficheType {
    void afficherType();
}

class Personne implements AfficheType {

    public void afficherType() {
        System.out.println(" Je suis une personne ");
    }
}

class Voiture implements AfficheType {

    public void afficherType() {

        System.out.println(" Je suis une voiture ");
    }
}
```

Exemple : déclaration d'une interface à laquelle doit se conformer tout individus

```
interface Individu {
    String getNom();
    String getPrenom();
    Date getDateNaiss();
}
```

Toutes les méthodes d'une interface sont abstraites : elles sont implicitement déclarées comme telles.

Une interface peut être d'accès public ou package. Si elle est publique, toutes ses méthodes sont implicitement publiques même si elles ne sont pas déclarées avec le modificateur public. Si elle est d'accès package, il s'agit d'une interface d'implémentation pour les autres classes du package et ses méthodes ont le même accès package : elles sont accessibles à toutes les classes du packages.

Les seules variables que l'on peut définir dans une interface sont des variables de classe qui doivent être constantes : elles sont donc implicitement déclarées avec le modificateur static et final même si elles sont définies avec d'autres modificateurs.

Exemple :

```
public interface MonInterface {
  public int VALEUR=0;
  void maMethode();
}
```

Toute classe qui implémente cette interface doit au moins posséder les méthodes qui sont déclarées dans l'interface. L'interface ne fait que donner une liste de méthodes qui seront à définir dans les classes qui implémentent l'interface.

Les méthodes déclarées dans une interface publique sont implicitement publiques et elles sont héritées par toutes les classes qui implémentent cette interface. De telles classes doivent, pour être instanciables, définir toutes les méthodes héritées de l'interface.

Une classe peut implémenter une ou plusieurs interfaces tout en héritant de sa classe mère.

L'implémentation d'une interface définit un cast : l'implémentation d'une interface est une forme d'héritage. Comme pour l'héritage d'une classe, l'héritage d'une classe qui implémente une interface définit un cast implicite de la classe fille vers cette interface. Il est important de noter que dans ce cas il n'est possible de faire des appels qu'à des méthodes de l'interface. Pour utiliser des méthodes de l'objet, il faut définir un cast explicite : il est préférable de contrôler la classe de l'objet pour éviter une exception ClassCastException à l'exécution.

4.6.7.1. Les méthodes par défaut

Depuis les débuts de Java, il est possible d'utiliser l'héritage multiple avec les interfaces. Les méthodes par défaut de Java 8 permettent l'héritage multiple de comportement.

L'héritage d'interfaces est possible depuis la version 1.0 de Java. : une interface ne peut contenir que la déclaration de constantes et de méthodes mais elle ne contient pas leurs traitements. C'est l'héritage multiple de type (multiple inheritance of type).

Avec les méthodes par défaut de Java 8, Java introduit la possibilité d'héritage multiple de comportement (multiple inheritance of behaviour) mais ne permet toujours pas l'héritage multiple d'état (multiple inheritance of state) puisque seules les interfaces sont concernées par l'héritage multiple.

Les interfaces sont couplées avec les classes qui les implémentent : par exemple, il n'est pas possible d'ajouter une méthode à une interface sans devoir modifier les classes qui l'implémentent directement.

Avant Java 8, la modification d'une ou plusieurs méthodes d'une interface oblige à adapter en conséquence toutes les classes qui l'implémentent. La seule solution pour éviter cela aurait été de créer une nouvelle version de l'API et les deux versions auraient dues cohabiter, ce qui aurait impliqué des problèmes pour maintenir et utiliser l'API.

L'ajout des lambdas dans certaines classes de base du JDK, notamment le package java.util, aurait eu beaucoup d'impacts sans les méthodes par défaut. L'intérêt initial des méthodes par défaut est donc de maintenir la compatibilité ascendante des API.

A partir de Java 8, il est possible d'utiliser les méthodes par défaut (default method) dans une interface. Elles permettent de définir le comportement d'une méthode dans l'interface dans laquelle elle est définie. Si aucune implémentation de la méthode n'est fournie dans une classe qui implémente l'interface alors c'est le comportement défini dans l'interface qui sera utilisé.

Une méthode par défaut est déclarée en utilisant le mot clé default. Le corps de la méthode contient l'implémentation des traitements.

Exemple (code Java 8) :

```
public interface MonInterface {
  default void maMethode() {
    System.out.println("Implementation par defaut");
  }
}
```

Les méthodes par défaut devraient surtout être utilisées pour maintenir une compatibilité ascendante afin de permettre l'ajout d'une méthode à une interface existante sans avoir à modifier les classes qui l'implémentent.

Les méthodes par défaut peuvent aussi éviter d'avoir à implémenter une classe abstraite qui contient les traitements par défaut de méthodes héritées dans les classes filles concrètes. Ces implémentations peuvent directement être codées dans des méthodes par défaut de l'interface. Les méthodes par défaut ne remplacent cependant pas complètement les classes abstraites qui peuvent avoir des constructeurs et des membres sous la forme de variables d'instance ou de classe.

Remarque : bien qu'il soit possible en Java 8 de définir des méthodes static et par défaut dans une interface, ce n'est pas possible dans la définition du type d'une annotation.

L'héritage multiple de comportement permet par exemple que de mêmes méthodes par défaut, avec des signatures identiques, soient définies dans plusieurs interfaces héritées par une interface fille. Il est probable que chaque implémentation de ces méthodes soit différente : le compilateur a besoin de règles pour déterminer quelle implémentation il doit utiliser :

- la redéfinition d'une méthode par une classe ou une super-classe est toujours prioritaire par rapport à une méthode par défaut
- l'implémentation choisie est celle par défaut de l'interface la plus spécifique

Ainsi une interface peut être modifiée en ajoutant une méthode sans compromettre sa compatibilité ascendante sous réserve qu'elle implémente cette méthode en tant que méthode par défaut.

Les méthodes par défaut sont virtuelles comme toutes les autres méthodes mais elles proposent une implémentation par défaut qui sera invoquée si la classe implémentant l'interface ne redéfinit pas explicitement la méthode. Une classe qui implémente une interface n'a donc pas l'obligation de redéfinir une méthode par défaut. Si celle-ci n'est pas redéfinie alors c'est l'implémentation contenue dans l'interface qui est utilisée.

Exemple (code Java 8) :

```java
package fr.jmdoudoux.dej.java8;

public interface Service {

  default void afficherNom() {
    System.out.println("Nom du service : inconnu");
  }
}
```

Exemple (code Java 8) :

```java
package fr.jmdoudoux.dej.java8;

public class MonService implements Service {
}
```

Exemple (code Java 8) :

```java
package fr.jmdoudoux.dej.java8;

public class TestMethodeParDefaut {

  public static void main(String[] args) {
    Service service = new MonService(){};
    service.afficherNom();
  }
}
```

Résultat :

```
Nom du service : inconnu
```

Si la classe redéfinit la méthode alors c'est l'implémentation de la méthode qui est utilisée.

Exemple (code Java 8) :

```java
package fr.jmdoudoux.dej.java8;

public class MonService implements Service {

  @Override
  public void afficherNom() {
    System.out.println("Nom du service : mon service");
  }
}
```

Résultat :

```
Nom du service : mon service
```

Il est possible d'utiliser l'annotation @Override sur la méthode redéfinie qu'elle soit par défaut ou non.

Il est aussi possible de créer directement une instance d'une interface si toutes ses méthodes sont des méthodes par défaut.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8;

public class TestMethodeParDefaut {

  public static void main(String[] args) {
    Service service = new Service(){};
    service.afficherNom();
  }
}
```

Résultat :

```
Nom du service : inconnu
```

Une interface peut hériter d'une autre interface qui contient une méthode par défaut et peut redéfinir cette méthode.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8;

public interface ServiceSpecial extends Service {

  default void afficherNom() {
    System.out.println("Nom du service special : inconnu");
  }
}
```

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8;

public class MonService implements ServiceSpecial {
}
```

Résultat de l'exécution de la classe TestMethodeParDefaut :

```
Nom du service special : inconnu
```

L'interface fille peut redéfinir la méthode sans la déclarer par défaut : dans ce cas, elle est redéfinie comme étant abstraite.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8;

public interface ServiceSpecial extends Service {

  void afficherNom();
}
```

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8;

public class MonService implements ServiceSpecial {

}
```

Résultat :

```
C:\java\TestJava8\src>javac -cp . com/jmdoudoux/test/java8/MonService.java
com\jmdoudoux\test\java8\MonService.java:3: error: MonService is not abstract an
d does not override abstract method afficherNom() in ServiceSpecial
public class MonService implements ServiceSpecial {
       ^
1 error
```

Une classe peut implémenter deux interfaces qui définissent la méthode par défaut avec des implémentations par défaut différentes.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8;

public interface Groupe {
  default void afficherNom() {
    System.out.println("Nom du groupe : inconnu");
  }
}
```

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8;

public class MonService implements Service, Groupe {

}
```

Dans ce cas, le compilateur lève une erreur qui précise le nom de la méthode par défaut et les interfaces concernées car il ne peut pas décider quelle implémentation il doit utiliser.

Résultat :

```
C:\java\TestJava8\src>javac -cp . com/jmdoudoux/test/java8/MonService.java
com\jmdoudoux\test\java8\MonService.java:3: error: class MonService inherits unr
elated defaults for afficherNom() from types Service and Groupe
public class MonService implements Service, Groupe {
       ^
1 error
```

Pour régler le problème, la classe doit explicitement redéfinir la méthode.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8;

public class MonService implements Service, Groupe {

  @Override
  public void afficherNom() {
    System.out.println("Nom du service : mon service");
  }
}
```

La redéfinition de la méthode dans la classe peut explicitement invoquer la méthode par défaut d'une des interfaces en utilisant la syntaxe : nom du type de l'interface, point, super, point le nom de la méthode.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8;

public class MonService implements Service, Groupe {

  @Override
  public void afficherNom() {
    Groupe.super.afficherNom();
  }
}
```

Résultat de l'exécution de la classe TestMethodeParDefaut :

```
Nom du groupe : inconnu
```

Il est possible que deux interfaces définissent la même méthode par défaut avec des implémentations différentes.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8;

public interface Service {

  default void afficherNom() {
    System.out.println("Nom du service : inconnu");
  }
}
```

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8;

public interface ServiceEtendu {

  default void afficherNom() {
    System.out.println("Nom du service etendu : inconnu");
  }
}
```

Une troisième interface peut hériter de ces deux interfaces.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8;

public interface ServiceSpecial extends Service, ServiceEtendu {

}
```

Le compilateur génère une erreur car il n'est pas en mesure de déterminer laquelle des deux implémentations il doit utiliser.

Résultat :

```
C:\java\TestJava8\src>javac -cp . com/jmdoudoux/test/java8/ServiceSpecial.java
com\jmdoudoux\test\java8\ServiceSpecial.java:3: error: interface ServiceSpecial
inherits unrelated defaults for afficherNom() from types Service and ServiceEtte
ndu
public interface ServiceSpecial extends Service, ServiceEtendu {
       ^
1 error
```

Une classe peut implémenter deux interfaces :

- une qui définit une méthode par défaut
- une autre qui définit la même méthode mais abstraite

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8;

public interface Groupe {

  void afficherNom();
}
```

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8;

public class MonService implements Service, Groupe {

}
```

Dans ce cas, le compilateur lève une erreur car il ne peut pas décider de prendre la méthode par défaut.

Résultat :

```
C:\java\TestJava8\src>javac -cp . com/jmdoudoux/test/java8/MonService.java
com\jmdoudoux\test\java8\MonService.java:4: error: MonService is not abstract an
d does not override abstract method afficherNom() in Groupe
public class MonService implements Service, Groupe {
       ^
1 error
```

Pour régler le problème, la classe doit explicitement redéfinir la méthode, éventuellement en invoquant la méthode par défaut de l'interface.

Si aucune des interfaces ne propose de méthodes par défaut, alors il n'y a pas d'ambiguïté et cette situation est celle qui pouvait exister avant Java 8. Une classe qui implémente ces interfaces doit fournir une implémentation pour chacune des méthodes ou être déclarée abstraite.

Il est possible qu'une classe hérite d'une classe et implémente une interface avec une méthode par défaut qui est implémentée par la classe mère.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8;

public class MonService implements ServiceEtendu {
}
```

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8;

public class ServiceComptable extends MonService implements Service{
}
```

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8;


public class TestMethodeParDefaut {

  public static void main(String[] args) {
    Service service = new ServiceComptable();
    service.afficherNom();
  }
}
```

Résultat :

```
Nom du service etendu : inconnu
```

Dans ce cas, c'est la classe mère qui prévaut et la méthode par défaut de l'interface Service est ignorée par le compilateur. En application de la règle «l'implémentation d'une classe ou super-classe est prioritaire», c'est la méthode getName() héritée de la classe MonService qui est utilisée.

Les méthodes par défaut sont virtuelles comme toutes les méthodes en Java.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8;

public interface ServiceEtendu extends Service {

  @Override
  default void afficherNom() {
    System.out.println("Nom du service etendu : inconnu");
  }
}
```

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8;

public interface ServiceDedie extends Service {
}
```

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8;

public class MonService implements ServiceDedie, ServiceEtendu {
}
```

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8;

public class TestMethodeParDefaut {

  public static void main(String[] args) {
    MonService monService = new MonService();
    monService.afficherNom();

    Service service = new MonService();
    service.afficherNom();
  }
}
```

Résultat :

```
Nom du service etendu : inconnu
Nom du service etendu : inconnu
```

Comme la méthode par défaut n'est pas redéfinie, le compilateur applique les règles pour déterminer l'implémentation de la méthode par défaut à utiliser : dans le cas ci-dessus, c'est celle de l'interface ServiceEtendu qui est la plus spécifique car elle redéfinit la méthode héritée de l'interface Service. Peu importe le type de la variable, c'est l'implémentation de l'instance créée qui est utilisée.

La règle qui veut qu'une implémentation d'une classe prévale toujours sur une méthode par défaut implique plusieurs choses :

- elle assure la compatibilité avec les classes antérieures à Java 8 : l'ajout de méthodes par défaut n'a pas d'effet sur du code qui fonctionnait avant l'ajout des méthodes par défaut
- elle empêche de définir de manière utile les méthodes de la classe Object. Le compilateur lève une erreur si une méthode ayant la signature d'une des méthodes toString(), equals() ou hashCode() de la classe Object est définie comme étant par défaut dans une interface

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8;

public interface Service {

  default String toString() {
    return "";
  }

  default boolean equals(Object o) {
    return false;
  }

  default public int hashCode() {
    return 0;
  }
}
```

Résultat :

```
C:\java\TestJava8\src>jav
ac -cp . com/jmdoudoux/test/java8/Service.java
com\jmdoudoux\test\java8\Service.java:5: error: default method toString in inter
face Service overrides a member of java.lang.Object
  default String toString() {
                 ^
com\jmdoudoux\test\java8\Service.java:9: error: default method equals in interfa
ce Service overrides a member of java.lang.Object
  default boolean equals(Object o) {
                  ^
com\jmdoudoux\test\java8\Service.java:13: error: default method hashCode in inte
rface Service overrides a member of java.lang.Object
  default public int hashCode() {
                     ^
3 errors
```

4.6.7.2. Les interfaces locales

Jusqu'à Java 15, il n'est pas possible de définir des interfaces locales.

Exemple (code Java 15) :

```
public class TestInterfaceLocal {

  public void traiter() {

    interface MonInterface {};
  }
}
```

Résultat :

```
C:\java>javac -version
javac 15

C:\java>javac TestInterfaceLocal.java
TestInterfaceLocal.java:5: error: interface not allowed here
    interface MonInterface {};
    ^
1 error

C:\java>
```

Java 16 permet de définir des interfaces locales, qui ne pourront donc être utilisées que dans la classe où elles sont définies.

Exemple (code Java 16) :

```
public class InterfaceLocale {

  public void traiter() {

    interface MonInterface {

      public default void afficher() {
        System.out.println("Hello");
      }
    };
    (new MonInterface() {}).afficher();
  }
}
```

Les interfaces locales ne peuvent pas capturer les variables du contexte englobant comme les paramètres de la méthode par exemple.

Exemple (code Java 16) :

```
public class InterfaceLocale {

  public void traiter(int valeur) {

    interface MonInterface {

      public default void afficher() {
        System.out.println(valeur);
      }
    };
    (new MonInterface() {}).afficher();
  }
}
```

Résultat :

```
C:\java>javac InterfaceLocale.java
InterfaceLocale.java:8: error: non-static variable valeur cannot be referenced from a static
context
        System.out.println(valeur);
                           ^
1 error

C:\java>
```

Les interfaces locales peuvent capturer les variables static du contexte englobant.

Exemple (code Java 16) :

```
public class InterfaceLocale {

  static int valeur = 10;

  public void traiter() {

    interface MonInterface {

      public default void afficher() {
        System.out.println(valeur);
      }
    };
    (new MonInterface() {}).afficher();
  }
}
```

4.6.8. L'héritage de méthodes statiques

Toutes les méthodes, incluant les méthodes statiques, sont héritées d'une super-classe du moment qu'elles soient accessibles par la classe fille.

Exemple :

```
package fr.jmdoudoux.java;

public class MaClasseMere {

  public static void maMethode() {
    System.out.println("MaClasseMere");
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej;

public class MaClasseFille extends MaClasseMere {
}
```

Exemple :

```
package fr.jmdoudoux.dej;

public class TestHeritageStatic {

  public static void main(String[] args) {
    MaClasseMere.maMethode();
    MaClasseFille.maMethode();
  }
}
```

Résultat :

```
MaClasseMere
MaClasseMere
```

Dans le cas des méthodes statiques, il y a cependant une restriction qui interdit de redéfinir une méthode statique héritée. Pourtant, il est possible d'écrire :

Exemple :

```
package fr.jmdoudoux.dej;

public class MaClasseFille extends MaClasseMere {

  public static void maMethode() {
    System.out.println("MaClasseFille");
  }
}
```

Résultat de l'exécution de la méthode TestHeritageStatic :

```
MaClasseMere
MaClasseFille
```

Une méthode static ne peut pas être redéfinie (overriden) mais il est possible de définir une méthode dans la classe fille avec la même signature. Si une méthode statique définie dans une classe mère est définie de manière identique dans une classe fille, celle-ci n'est pas une redéfinition mais elle masque (hidden) la méthode de la classe mère.

Exemple :

```
package fr.jmdoudoux.dej;

public class MaClasseMere {

  public static void maMethode() {
    System.out.println("MaClasseMere");
  }

  public static void monAutreMethode() {
    maMethode();
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej;

public class MaClasseFille extends MaClasseMere {

  public static void maMethode() {
    System.out.println("MaClasseFille");
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej;

public class TestHeritageStatic {

  public static void main(String[] args) {
    MaClasseMere.maMethode();
    MaClasseMere.monAutreMethode();
    MaClasseFille.maMethode();
    MaClasseFille.monAutreMethode();
  }
}
```

Résultat :

```
MaClasseMere
MaClasseMere
MaClasseFille
MaClasseMere
```

Il n'est donc pas possible d'utiliser l'annotation @Override.

Exemple (code Java 6) :

```
package fr.jmdoudoux.dej;

public class MaClasseFille extends MaClasseMere {

  @Override
  public static void maMethode() {
    System.out.println("MaClasseFille");
  }
}
```

Résultat :

```
C:\java\Test\src\com\jmdoudoux\dej>javac
```

```
MaClasseFille.java
MaClasseFille.java:3:
error: cannot find symbol
public class MaClasseFille extends MaClasseMere {
                                   ^
  symbol: class MaClasseMere
MaClasseFille.java:5:
error: method does not override or implement a method from a supertype
  @Override
  ^
2 errors
```

Il n'est pas possible de redéfinir une méthode statique dans une classe fille si cette redéfinition n'est pas statique.

Exemple :

```
package fr.jmdoudoux.java;

public class MaClasseFille extends MaClasseMere {

  public void maMethode() {
    System.out.println("MaClasseFille");
  }
}
```

Résultat :

```
C:\Users\jm\workspace\Test\src>javac com/jmdoudoux/java/MaClasseFille.java
com\jmdoudoux\java\MaClasseFille.java:5:
error: maMethode() in MaClasseFille cannot override maMethode() in MaClasseMere
  public void maMethode() {
              ^
  overridden method is static
1 error
```

La redéfinition des méthodes d'instances implique une résolution à l'exécution. Les méthodes statiques sont des méthodes de classes : leurs résolutions sont toujours faites par le compilateur à la compilation. Il n'est donc pas possible d'utiliser le polymorphisme sur des méthodes statiques, celles-ci étant résolues par le compilateur.

Exemple :

```
package fr.jmdoudoux.java;

public class Test {

  public static void main(String[] args) {
    MaClasseMere mere = new MaClasseMere();
    mere.maMethode();
    MaClasseFille fille = new MaClasseFille();
    fille.maMethode();
    mere = new MaClasseFille();
    mere.maMethode();
  }
}
```

Résultat :

```
MaClasseMere
MaClasseFille
MaClasseMere
```

Ce comportement est dû au fait que la méthode n'est pas redéfinie mais masquée. Les accès aux méthodes statiques sont toujours résolus à la compilation : le compilateur utilise le type de la variable et pas le type de l'instance qui invoque la méthode. L'invocation d'une méthode statique à partir d'une instance est possible en Java mais le compilateur émet un avertissement pour préconiser l'utilisation de la classe pour invoquer la méthode et ainsi éviter toute confusion sur la méthode invoquée.

Le compilateur remplace l'instance par la classe de son type, ce qui permet à l'exemple ci-dessous de se compiler et de s'exécuter correctement.

Exemple :

```
package fr.jmdoudoux.java;

public class Test {

  public static void main(String[] args) {
    MaClasseMere mere = null;
    mere.maMethode();
  }
}
```

Résultat :

```
MaClasseMere
```

La redéfinition (overriding) est une fonctionnalité offerte par les langages de POO qui permet de mettre en ouvre une forme de polymorphisme. Une sous-classe fournit une implémentation dédiée d'une méthode héritée de sa super-classe : les signatures des deux méthodes doivent être les mêmes. Le choix de la méthode à exécuter est déterminé à l'exécution en fonction du type de l'objet qui l'invoque.

La surcharge (overload) est une fonctionnalité offerte par les langages de POO qui permet de mettre en oeuvre une forme de polymorphisme. Elle permet de définir différentes méthodes ayant le même nom avec le nombre et/ou le type des paramètres différent.

Le choix de la méthode à exécuter est déterminée statiquement par le compilateur en fonction des paramètres utilisés à l'invocation.

4.6.9. Des conseils sur l'héritage

Lors de la création d'une classe « mère » il faut tenir compte des points suivants :

- la définition des accès aux variables d'instances, très souvent privées, doit être réfléchie entre protected et private
- pour empêcher la redéfinition d'une méthode ou sa surcharge, il faut la déclarer avec le modificateur final

Lors de la création d'une classe fille, pour chaque méthode héritée qui n'est pas final, il faut envisager les cas suivants :

- la méthode héritée convient à la classe fille : on ne doit pas la redéfinir
- la méthode héritée convient mais partiellement du fait de la spécialisation apportée par la classe fille : il faut la redéfinir voire la surcharger. La plupart du temps une redéfinition commencera par appeler la méthode héritée (en utilisant le mot clé super) pour garantir l'évolution du code
- la méthode héritée ne convient pas : il faut redéfinir ou surcharger la méthode sans appeler la méthode héritée lors de la redéfinition.

4.7. Les packages

En Java, il existe un moyen de regrouper des classes voisines ou qui couvrent un même domaine : ce sont les packages.

4.7.1. La définition d'un package

Pour réaliser un package, on écrit un nombre quelconque de classes dans plusieurs fichiers d'un même répertoire et au début de chaque fichier on met la directive ci-dessous où nom-du-package doit être composé des répertoires séparés par un caractère point :

package nom-du-package;

La hiérarchie d'un package se retrouve dans l'arborescence du disque dur puisque chaque package est dans un répertoire nommé du nom du package.

D'une façon générale, l'instruction package associe toutes les classes qui sont définies dans un fichier source à un même package.

Le mot clé package doit être la première instruction dans un fichier source et il ne doit être présent qu'une seule fois dans le fichier source (une classe ne peut pas appartenir à plusieurs packages).

4.7.2. Les importations

Pour pouvoir utiliser un type en Java, il faut utiliser le nom pleinement qualifié du type qui inclue le nom du package avec la notation utilisant un point.

Afin de réduire la verbosité lors de l'utilisation de type, il est possible d'utiliser les imports. Ils utilisent le mot clé import qui définit un alias du nom pleinement qualifié d'un type vers simplement le nom du type.

Il y a deux manière d'utiliser les imports :

- préciser un nom de classe ou d'interface qui sera l'unique entité importée
- ou indiquer un package suivi d'un point et d'un caractère * indiquant toutes les classes et interfaces définies dans le package

Exemple	Rôle
`import nomPackage.*;`	toutes les classes du package sont importées
`import nomPackage.nomClasse;`	appel à une seule classe : l'avantage de cette notation est de réduire le temps de compilation

Attention : l'astérisque n'importe pas les sous-packages. Par exemple, il n'est pas possible d'écrire import java.*.

L'utilisation de joker dans les imports est un choix :

- ne pas les utiliser permet de déterminer précisément les classes utilisées
- l'utilisation de joker permet de réduire le nombre d'import

Généralement, les IDE proposent une fonctionnalité qui permet de réorganiser les imports en

- remplaçant les jokers par les classes utilisées
- triant les classes utilisées par ordre alphabétique

Les imports sont traités par le compilateur : le bytecode généré sera le même qu'une clause import utilise le nom pleinement qualifié d'une classe ou un nom comportant un joker.

Le bytecode ne contient pas les imports mais simplement le nom pleinement qualifiés des classes.

L'utilisation d'un joker dans un import n'a aucun impact sur les performances ou la consommation mémoire à l'exécution.

L'utilisation de joker dans les imports peut cependant induire une collision de classes si deux classes ayant le même nom dans deux packages différents sont importés en utilisant un joker.

L'utilisation de joker peut aussi induire de futurs problèmes de compilation si deux import utilisent un joker et qu'une classe ayant un nom existant dans un des packages est ajoutée dans l'autre package ultérieurement. La classe ne se compilera alors plus.

En précisant son nom complet, il est possible d'appeler une méthode d'une classe sans utiliser son importation :

nomPackage.nomClasse.nomméthode(arg1, arg2 ...)

Il existe plusieurs types de packages : le package par défaut (identifié par le point qui représente le répertoire courant et permet de localiser les classes qui ne sont pas associées à un package particulier), les packages standard qui sont empaquetés dans le fichier classes.zip (Java 1.0 et 1.1) et rt.jar (à partir de Java 1.2) et les packages personnels.

Le compilateur implémente automatiquement une instruction import lors de la compilation d'un programme Java même si elle ne figure pas explicitement au début du programme : import java.lang.*;. Ce package contient entre autres les classes de base de tous les objets Java dont la classe Object.

Un package par défaut est systématiquement attribué par le compilateur aux classes qui sont définies sans déclarer explicitement une appartenance à un package. Ce package par défaut correspond au répertoire courant qui est le répertoire de travail.

4.7.3. Les importations statiques

Jusqu'à la version 1.4 de Java, pour utiliser un membre statique d'une classe, il fallait obligatoirement préfixer ce membre par le nom de la classe qui le contient.

Par exemple, pour utiliser la constante Pi définie dans la classe java.lang.Math, il est nécessaire d'utiliser Math.PI

Exemple :

```
public class TestStaticImportOld {

  public static void main(String[] args) {
    System.out.println(Math.PI);
    System.out.println(Math.sin(0));
  }
}
```

Java 1.5 propose une solution pour réduire le code à écrire concernant les membres statiques en proposant une nouvelle fonctionnalité concernant l'importation de package : l'import statique (static import).

Ce nouveau concept permet d'appliquer les mêmes règles aux membres statiques qu'aux classes et interfaces pour l'importation classique.

Cette nouvelle fonctionnalité est développée dans la JSR 201. Elle s'utilise comme une importation classique en ajoutant le mot clé static.

Exemple (java 1.5) :

```
import static java.lang.Math.*;

public class TestStaticImport {

  public static void main(String[] args) {
    System.out.println(PI);
    System.out.println(sin(0));
  }
}
```

L'utilisation de l'importation statique s'applique à tous les membres statiques : constantes et méthodes statiques de l'élément importé.

4.7.4. La collision de classes

Deux classes entrent en collision lorsqu'elles portent le même nom mais qu'elles sont définies dans des packages différents. Dans ce cas, il faut qualifier explicitement le nom de la classe avec le nom complet du package.

4.7.5. Les packages et l'environnement système

Les classes Java sont chargées par le compilateur (au moment de la compilation) et par la machine virtuelle (au moment de l'exécution). Les techniques de chargement des classes varient en fonction de l'implémentation de la machine virtuelle. Dans la plupart des cas, une variable d'environnement CLASSPATH référence tous les répertoires qui hébergent des packages susceptibles d'être importés.

Exemple sous Windows :

CLASSPATH = .;C:\MonApplication\lib\classes.zip;C:\MonApplication\classes

L'importation des packages ne fonctionne que si le chemin de recherche spécifié dans une variable particulière pointe sur les packages, sinon le nom du package devra refléter la structure du répertoire où il se trouve. Pour déterminer l'endroit où se trouvent les fichiers .class à importer, le compilateur utilise une variable d'environnement dénommée CLASSPATH. Le compilateur peut lire les fichiers .class comme des fichiers indépendants ou comme des fichiers ZIP ou JAR dans lesquels les classes sont réunies et compressées.

4.8. Les classes internes

Les classes internes (inner classes) sont une extension du langage Java introduite dans la version 1.1 de Java. Ce sont des classes qui sont définies dans une autre classe. Les difficultés dans leur utilisation concernent leur visibilité et leur accès aux membres de la classe dans laquelle elles sont définies.

Exemple très simple :

```
public class ClassePrincipale1 {
  class ClasseInterne {
  }
}
```

Les classes internes sont particulièrement utiles pour :

- permettre de définir une classe à l'endroit où une seule autre en a besoin
- définir des classes de type adapter (essentiellement à partir du JDK 1.1 pour traiter des événements émis par les interfaces graphiques)
- définir des méthodes de type callback d'une façon générale

Pour permettre de garder une compatibilité avec la version précédente de la JVM, seul le compilateur a été modifié. Le compilateur interprète la syntaxe des classes internes pour modifier le code source et générer du bytecode compatible avec la première JVM.

Il est possible d'imbriquer plusieurs classes internes. Java ne possède pas de restrictions sur le nombre de classes qu'il est ainsi possible d'imbriquer. En revanche une limitation peut intervenir au niveau du système d'exploitation en ce qui concerne la longueur du nom du fichier .class généré pour les différentes classes internes.

Si plusieurs classes internes sont imbriquées, il n'est pas possible d'utiliser un nom pour la classe qui soit déjà attribué à une de ses classes englobantes. Le compilateur génèrera une erreur à la compilation.

Exemple :

```
public class ClassePrincipale6 {
  class ClasseInterne1 {
    class ClasseInterne2 {
      class ClasseInterne3 {
      }
    }
  }
}
```

Le nom de la classe interne utilise la notation qualifiée avec le point préfixé par le nom de la classe principale. Ainsi, pour utiliser ou accéder à une classe interne dans le code, il faut la préfixer par le nom de la classe principale suivi d'un point.

Cependant cette notation ne représente pas physiquement le nom du fichier qui contient le bytecode. Le nom du fichier qui contient le bytecode de la classe interne est modifié par le compilateur pour éviter des conflits avec d'autres noms d'entités : à partir de la classe principale, le point de séparation entre chaque classe interne est remplacé par un caractère $ (dollar).

Par exemple, la compilation du code de l'exemple précédent génère quatre fichiers contenant le bytecode :
ClassePrincipale6$ClasseInterne1$ClasseInterne2$ClasseInterne3.class
ClassePrincipale6$ClasseInterne1$ClasseInterne2.class
ClassePrincipale6$ClasseInterne1.class
ClassePrincipale6.class

L'utilisation du signe $ entre la classe principale et la classe interne permet d'éviter des confusions de nom entre le nom d'une classe appartenant à un package et le nom d'une classe interne.

L'avantage de cette notation est de créer un nouvel espace de nommage qui dépend de la classe et pas d'un package. Ceci renforce le lien entre la classe interne et sa classe englobante.

C'est le nom du fichier qu'il faut préciser lorsque l'on tente de charger la classe avec la méthode forName() de la classe Class. C'est aussi sous cette forme qu'est restitué le résultat d'un appel aux méthodes getClass().getName() sur un objet qui est une classe interne.

Exemple :

```
public class ClassePrincipale8 {
  public class ClasseInterne {
  }

  public static void main(String[] args) {
    ClassePrincipale8 cp = new ClassePrincipale8();
    ClassePrincipale8.ClasseInterne ci = cp. new ClasseInterne() ;
    System.out.println(ci.getClass().getName());
  }
}
```

Résultat :

```
java ClassePrincipale8
ClassePrincipale8$ClasseInterne
```

L'accessibilité à la classe interne respecte les règles de visibilité du langage. Il est même possible de définir une classe interne private pour limiter son accès à sa seule classe principale.

Exemple :

```
public class ClassePrincipale7 {
  private class ClasseInterne {
  }
}
```

Il n'est pas possible de déclarer des membres statiques dans une classe interne :

Exemple :

```
public class ClassePrincipale10 {
  public class ClasseInterne {
    static int var = 3;
  }
}
```

Résultat :

```
javac ClassePrincipale10.java
ClassePrincipale10.java:3: Variable var can't be static in inner class ClassePri
ncipale10. ClasseInterne.  Only members of interfaces and top-level classes can
be static.
                static int var = 3;
                               ^
1 error
```

Pour pouvoir utiliser une variable de classe dans une classe interne, il faut la déclarer dans sa classe englobante.

Il existe quatre types de classes internes :

- les classes internes non statiques : elles sont membres à part entière de la classe qui les englobe et peuvent accéder à tous les membres de cette dernière
- les classes internes locales : elles sont définies dans un bloc de code. Elles peuvent être static ou non.
- les classes internes anonymes : elles sont définies et instanciées à la volée sans posséder de nom
- les classe internes statiques : elles sont membres à part entière de la classe qui les englobe et peuvent accéder uniquement aux membres statiques de cette dernière

4.8.1. Les classes internes non statiques

Les classes internes non statiques (member inner-classes) sont définies dans une classe dite « principale » (top-level class) en tant que membres de cette classe. Leur avantage est de pouvoir accéder aux autres membres de la classe principale même ceux déclarés avec le modificateur private.

Exemple :

```
public class ClassePrincipale20 {
  private int valeur = 1;

  class ClasseInterne {
    public void afficherValeur() {
      System.out.println("valeur = "+valeur);
    }
  }

  public static void main(String[] args) {
    ClassePrincipale20 cp = new ClassePrincipale20();
    ClasseInterne ci = cp. new ClasseInterne();
    ci.afficherValeur();
        }
}
```

Résultat :

```
C:\testinterne>javac ClassePrincipale20.java
```

```
C:\testinterne>java ClassePrincipale20
valeur = 1
```

Le mot clé this fait toujours référence à l'instance en cours. Ainsi this.var fait référence à la variable var de l'instance courante. L'utilisation du mot clé this dans une classe interne fait donc référence à l'instance courante de cette classe interne.

Exemple :

```
public class ClassePrincipale16 {
  class ClasseInterne {
    int var = 3;

    public void affiche() {
      System.out.println("var      = "+var);
      System.out.println("this.var = "+this.var);
    }
  }

  ClasseInterne ci = this. new ClasseInterne();

  public static void main(String[] args) {
    ClassePrincipale16 cp = new ClassePrincipale16();
    ClasseInterne ci = cp. new ClasseInterne();
    ci.affiche();
        }
}
```

Résultat :

```
C:\>java ClassePrincipale16
var      = 3
this.var = 3
```

Une classe interne a accès à tous les membres de sa classe principale. Dans le code, pour pouvoir faire référence à un membre de la classe principale, il suffit simplement d'utiliser son nom de variable.

Exemple :

```
public class ClassePrincipale17 {
  int valeur = 5;

  class ClasseInterne {
    int var = 3;

    public void affiche() {
      System.out.println("var      = "+var);
      System.out.println("this.var = "+this.var);
      System.out.println("valeur   = "+valeur);
    }
  }

  ClasseInterne ci = this. new ClasseInterne();

  public static void main(String[] args) {
    ClassePrincipale17 cp = new ClassePrincipale17();
    ClasseInterne ci = cp. new ClasseInterne();
    ci.affiche();
        }
}
```

Résultat :

```
C:\testinterne>java ClassePrincipale17
var      = 3
this.var = 3
```

```
valeur   = 5
```

La situation se complique un peu plus si la classe principale et la classe interne possèdent toutes les deux un membre de même nom. Dans ce cas, il faut utiliser la version qualifiée du mot clé this pour accéder au membre de la classe principale. La qualification se fait avec le nom de la classe principale ou plus généralement avec le nom qualifié d'une des classes englobantes.

Exemple :

```
public class ClassePrincipale18 {
  int var = 5;

  class ClasseInterne {
    int var = 3;

    public void affiche() {
      System.out.println("var                          = "+var);
      System.out.println("this.var                     = "+this.var);
      System.out.println("ClassePrincipale18.this.var  = "
        +ClassePrincipale18.this.var);
    }
  }

  ClasseInterne ci = this. new ClasseInterne();

  public static void main(String[] args) {
    ClassePrincipale18 cp = new ClassePrincipale18();
    ClasseInterne ci = cp. new ClasseInterne();
    ci.affiche();
        }
}
```

Résultat :

```
C:\>java ClassePrincipale18
var                          = 3
this.var                     = 3
ClassePrincipale18.this.var  = 5
```

Comme une classe interne ne peut être nommée du même nom que l'une de ses classes englobantes, ce nom qualifié est unique et il ne risque pas d'y avoir de confusion.

Le nom qualifié d'une classe interne est nom_classe_principale.nom_classe_interne. C'est donc le même principe que celui utilisé pour qualifier une classe contenue dans un package. La notation avec le point est donc légèrement étendue.

L'accès aux membres de la classe principale est possible car le compilateur modifie le code de la classe principale et celui de la classe interne pour fournir à la classe interne une référence sur la classe principale.

Le code de la classe interne est modifié pour :

- ajouter une variable privée finale du type de la classe principale nommée this$0
- ajouter un paramètre supplémentaire dans le constructeur qui sera la classe principale et qui va initialiser la variable this$0
- utiliser cette variable pour préfixer les attributs de la classe principale utilisés dans la classe interne.

La code de la classe principale est modifié pour :

- ajouter une méthode static pour chaque champ de la classe principale qui attend en paramètre un objet de la classe principale. Cette méthode renvoie simplement la valeur du champ. Le nom de cette méthode est de la forme access$0
- modifier le code d'instanciation de la classe interne pour appeler le constructeur modifié

Dans le bytecode généré, une variable privée finale contient une référence vers la classe principale. Cette variable est nommée this$0. Comme elle est générée par le compilateur, cette variable n'est pas utilisable dans le code source. C'est à partir de cette référence que le compilateur peut modifier le code pour accéder aux membres de la classe principale.

Pour pouvoir avoir accès aux membres de la classe principale, le compilateur génère dans la classe principale des accesseurs sur ses membres. Ainsi, dans la classe interne, pour accéder à un membre de la classe principale, le compilateur appelle un de ses accesseurs en utilisant la référence stockée. Ces méthodes ont un nom de la forme access$numero_unique et sont bien sûr inutilisables dans le code source puisqu'elles sont générées par le compilateur.

En tant que membre de la classe principale, une classe interne peut être déclarée avec le modificateur private ou protected.

Grâce au mot clé this, une classe peut faire référence dans le code source à son unique instance lors de l'exécution. Une classe interne possède au moins deux références :

- l'instance de la classe interne elle-même
- l'instance de sa classe principale
- éventuellement les instances des classes internes imbriquées

Dans la classe interne, il est possible pour accéder à une de ces instances d'utiliser le mot clé this préfixé par le nom de la classe suivi d'un point :
nom_classe_principale.this
nom_classe_interne.this

Le mot this seul désigne toujours l'instance de la classe courante dans son code source, donc this seul dans une classe interne désigne l'instance de cette classe interne.

Une classe interne non statique doit toujours être instanciée relativement à un objet implicite ou explicite du type de la classe principale. A la compilation, le compilateur ajoute dans la classe interne une référence vers la classe principale contenue dans une variable privée nommée this$0. Cette référence est initialisée avec un paramètre fourni au constructeur de la classe interne. Ce mécanisme permet de lier les deux instances.

La création d'une classe interne nécessite donc obligatoirement une instance de sa classe principale. Si cette instance n'est pas accessible, il faut en créer une et utiliser une notation particulière de l'opérateur new pour pouvoir instancier la classe interne. Par défaut, lors de l'instanciation d'une classe interne, si aucune instance de la classe principale n'est utilisée, c'est l'instance courante qui est utilisée (mot clé this).

Exemple :

```
public class ClassePrincipale14 {
  class ClasseInterne {
  }

  ClasseInterne ci = this. new ClasseInterne();
}
```

Pour créer une instance d'une classe interne dans une méthode statique de la classe principale, (la méthode main() par exemple), il faut obligatoirement instancier un objet de la classe principale avant et utiliser cet objet lors de la création de l'instance de la classe interne. Pour créer l'instance de la classe interne, il faut alors utiliser une syntaxe particulière de l'opérateur new.

Exemple :

```
public class ClassePrincipale15 {
  class ClasseInterne {
  }

  ClasseInterne ci = this. new ClasseInterne();

  static void maMethode() {
    ClassePrincipale15 cp = new ClassePrincipale15();
    ClasseInterne ci = cp. new ClasseInterne();
        }
}
```

Il est possible d'utiliser une syntaxe condensée pour créer les deux instances en une seule et même ligne de code.

Exemple :

```
public class ClassePrincipale19 {
  class ClasseInterne {
  }

  static void maMethode() {
    ClasseInterne ci = new ClassePrincipale19(). new ClasseInterne();
        }
}
```

Une classe peut hériter d'une classe interne. Dans ce cas, il faut obligatoirement fournir aux constructeurs de la classe une référence sur la classe principale de la classe mère et appeler explicitement dans le constructeur le constructeur de cette classe principale avec une notation particulière du mot clé super

Exemple :

```
public class ClassePrincipale9 {
  public class ClasseInterne {
  }

  class ClasseFille extends ClassePrincipale9.ClasseInterne {
    ClasseFille(ClassePrincipale9 cp) {
      cp. super();
    }
  }
}
```

Une classe interne peut être déclarée avec les modificateurs final et abstract. Avec le modificateur final, la classe interne ne pourra être utilisée comme classe mère. Avec le modificateur abstract, la classe interne devra être étendue pour pouvoir être instanciée.

4.8.2. Les classes internes locales

Ces classes internes locales (local inner-classes) sont définies à l'intérieur d'une méthode ou d'un bloc de code. Ces classes ne sont utilisables que dans le bloc de code où elles sont définies. Les classes internes locales ont toujours accès aux membres de la classe englobante.

Exemple :

```
public class ClassePrincipale21 {
   int varInstance = 1;

   public static void main(String args[]) {
     ClassePrincipale21 cp = new ClassePrincipale21();
     cp.maMethode();
   }

   public void maMethode() {

     class ClasseInterne {
       public void affiche() {
         System.out.println("varInstance = " + varInstance);
       }
     }

     ClasseInterne ci = new ClasseInterne();
     ci.affiche();
   }
 }
```

Résultat :

```
C:\testinterne>javac ClassePrincipale21.java

C:\testinterne>java ClassePrincipale21
varInstance = 1
```

Leur particularité, en plus d'avoir un accès aux membres de la classe principale, est d'avoir aussi un accès à certaines variables locales du bloc où est définie la classe interne.

Ces variables définies dans la méthode (variables ou paramètres de la méthode) sont celles qui le sont avec le mot clé final. Ces variables doivent être initialisées avant leur utilisation par la classe interne. Elles sont utilisables n'importe où dans le code de la classe interne.

Le modificateur final désigne une variable dont la valeur ne peut être changée une fois qu'elle a été initialisée.

Exemple :

```
public class ClassePrincipale12 {

  public static void main(String args[]) {
    ClassePrincipale12 cp = new ClassePrincipale12();
    cp.maMethode();
  }

  public void maMethode() {
    int varLocale = 3;

    class ClasseInterne {
      public void affiche() {
        System.out.println("varLocale = " + varLocale);
      }
    }

    ClasseInterne ci = new ClasseInterne();
    ci.affiche();
  }
}
```

Résultat :

```
javac ClassePrincipale12.java
ClassePrincipale12.java:14: Attempt to use a non-final variable varLocale from a
 different method. From enclosing blocks, only final local variables are availab
le.
         System.out.println("varLocale = " + varLocale);
                                             ^
1 error
```

Cette restriction est imposée par la gestion du cycle de vie d'une variable locale. Une telle variable n'existe que durant l'exécution de cette méthode. Une variable finale est une variable dont la valeur ne peut être modifiée après son initialisation. Ainsi, il est possible sans risque pour le compilateur d'ajouter un membre dans la classe interne et de copier le contenu de la variable finale dedans.

Exemple :

```
public class ClassePrincipale13 {

  public static void main(String args[]) {
    ClassePrincipale13 cp = new ClassePrincipale13();
    cp.maMethode();
  }

  public void maMethode() {
    final int varLocale = 3;

    class ClasseInterne {
      public void affiche(final int varParam) {
        System.out.println("varLocale = " + varLocale);
        System.out.println("varParam  = " + varParam);
      }
    }

    ClasseInterne ci = new ClasseInterne();
    ci.affiche(5);
   }
}
```

Résultat :

```
C:\>javac ClassePrincipale13.java

C:\>java ClassePrincipale13
varLocale = 3
varParam  = 5
```

Pour permettre à une classe interne locale d'accéder à une variable locale utilisée dans le bloc de code où est définie la classe interne, la variable doit être stockée dans un endroit accessible par la classe interne. Pour que cela fonctionne, le compilateur ajoute les variables nécessaires dans le constructeur de la classe interne.

Les variables accédées sont dupliquées dans la classe interne par le compilateur. Il ajoute pour chaque variable un membre privé dans la classe interne dont le nom est de la forme val$nom_variable. Comme la variable accédée est déclarée finale, cette copie peut être faite sans risque. La valeur de chacune de ces variables est fournie en paramètre du constructeur qui a été modifié par le compilateur.

Une classe qui est définie dans un bloc de code n'est pas un membre de la classe englobante : elle n'est donc pas accessible en dehors du bloc de code où elle est définie. Ses restrictions sont équivalentes à la déclaration d'une variable dans un bloc de code.

Les variables ajoutées par le compilateur sont préfixées par this$ et val$. Ces variables et le constructeur modifié par le compilateur ne sont pas utilisables dans le code source.

Etant visible uniquement dans le bloc de code qui la définit, une classe interne locale ne peut pas utiliser les modificateurs public, private, protected et static dans sa définition. Leur utilisation provoque une erreur à la compilation.

Exemple :

```
public class ClassePrincipale11 {
  public void maMethode() {
    public class ClasseInterne {
    }
  }
}
```

Résultat :

```
javac ClassePrincipale11.java
ClassePrincipale11.java:2: '}' expected.
        public void maMethode() {
                                 ^
ClassePrincipale11.java:3: Statement expected.
                public class ClasseInterne {
                ^
ClassePrincipale11.java:7: Class or interface declaration expected.
}
^
3 errors
```

4.8.3. Les classes internes anonymes

Les classes internes anonymes (anonymous inner-classes) sont des classes internes qui ne possèdent pas de nom. Elles ne peuvent donc être instanciées qu'à l'endroit où elles sont définies.

Ce type de classe est très pratique lorsqu'une classe doit être utilisée une seule fois : c'est par exemple le cas d'une classe qui doit être utilisée comme un callback.

Une syntaxe particulière de l'opérateur new permet de déclarer et instancier une classe interne :
new classe_ou_interface () {
// définition des attributs et des méthodes de la classe interne
}

Cette syntaxe particulière utilise le mot clé new suivi d'un nom de classe ou d'interface que la classe interne va respectivement étendre ou implémenter. La définition de la classe suit entre deux accolades. Une classe interne anonyme peut soit hériter d'une classe soit implémenter une interface mais elle ne peut pas explicitement faire les deux.

Si la classe interne étend une classe, il est possible de fournir des paramètres entre les parenthèses qui suivent le nom de la classe. Ces arguments éventuels fournis au moment de l'utilisation de l'opérateur new sont passés au constructeur de la super-classe. En effet, comme la classe ne possède pas de nom, elle ne possède pas non plus de constructeur.

Les classes internes anonymes qui implémentent une interface héritent obligatoirement de la classe Object. Comme cette classe ne possède qu'un constructeur sans paramètre, il n'est pas possible lors de l'instanciation de la classe interne de lui fournir des paramètres.

Une classe interne anonyme ne peut pas avoir de constructeur puisqu'elle ne possède pas de nom mais elle peut avoir des initialisateurs.

Exemple :

```
public void init() {
  boutonQuitter.addActionListener(
    new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        System.exit(0);
      }
    }
  );
}
```

Les classes anonymes sont un moyen pratique de déclarer un objet sans avoir à lui trouver un nom. La contrepartie est que cette classe ne pourra être instanciée dans le code qu'à l'endroit où elle est définie : elle est déclarée et instanciée en un seul et unique endroit.

Le compilateur génère un fichier ayant pour nom la forme suivante : nom_classe_principale$numéro_unique. En fait, le compilateur attribut un numéro unique à chaque classe interne anonyme et c'est ce numéro qui est donné au nom du fichier préfixé par le nom de la classe englobante et d'un signe '$'.

4.8.4. Les classes internes statiques

Les classes internes statiques (static member inner-classes) sont des classes internes qui ne possèdent pas de référence vers leur classe principale. Elles ne peuvent donc pas accéder aux membres d'instance de leur classe englobante. Elles peuvent toutefois avoir accès aux variables statiques de la classe englobante.

Pour les déclarer, il suffit d'utiliser en plus le modificateur static dans la déclaration de la classe interne.

Leur utilisation est obligatoire si la classe est utilisée dans une méthode statique qui par définition peut être appelée sans avoir d'instance de la classe et que l'on ne peut pas avoir une instance de la classe englobante. Dans le cas contraire, le compilateur indiquera une erreur :

Exemple :

```
public class ClassePrincipale4 {

  class ClasseInterne {
    public void afficher() {
      System.out.println("bonjour");
    }
  }

  public static void main(String[] args) {
    new ClasseInterne().afficher();
  }
}
```

Résultat :

```
javac ClassePrincipale4.java
ClassePrincipale4.java:10: No enclosing instance of class ClassePrincipale4 is i
n scope; an explicit one must be provided when creating inner class ClassePrinci
pale4. ClasseInterne, as in "outer. new Inner()" or "outer. super()".
                new ClasseInterne().afficher();
                ^
1 error
```

En déclarant la classe interne static, le code se compile et peut être exécuté.

Exemple :

```
public class ClassePrincipale4 {

  static class ClasseInterne {
    public void afficher() {
      System.out.println("bonjour");
    }
  }

  public static void main(String[] args) {
    new ClasseInterne().afficher();
  }
}
```

Résultat :

```
javac ClassePrincipale4.java

java ClassePrincipale4
bonjour
```

Comme elle ne possède pas de référence sur sa classe englobante, une classe interne statique est traduite par le compilateur comme une classe principale. En fait, il est difficile de les mettre dans une catégorie (classe principale ou classe interne) car dans le code source c'est une classe interne (classe définie dans une autre) et dans le bytecode généré c'est une classe principale. Ce type de classe n'est pas très employé.

4.8.5. Les membres static dans une classe interne

Historiquement, une erreur est émise par le compilateur si une classe interne déclare un membre static qui n'est pas une constante.

Exemple (code Java 15) :

```
public class TestStatic {

  public void traiter() {
    class MaClasse {
      public final static int valeur = 0;

      public static void afficher() {
        System.out.println(valeur);
      }
    };
  }
}
```

Résultat :

```
C:\java>javac TestStatic.java
TestStatic.java:7: error: Illegal static declaration in inner class MaClasse
      public static void afficher() {
                         ^
  modifier 'static' is only allowed in constant variable declarations
1 error
```

Cela implique qu'une classe interne ne peut déclarer un membre soit la définition record puisque les records imbriqués sont implicitement static.

Exemple (code Java 15) :

```
public class TestInnerRecord {

  class MaClasse {
    record MonRecord(String nom) {};
  };
}
```

Résultat :

```
C:\java>javac -version
javac 15

C:\java>javac TestInnerRecord.java
TestInnerRecord.java:4: warning: 'record' may become a restricted type name in a future release
and may be unusable for type declarations or as the element type of an array
    record MonRecord(String nom) {};
    ^
TestInnerRecord.java:4: error: cannot find symbol
    record MonRecord(String nom) {};
    ^
  symbol:   class record
  location: class TestInnerRecord.MaClasse
1 error
1 warning
```

La JEP 395 introduite dans Java 16, permet à une classe interne de déclarer des membres qui soient implicitement ou explicitement static.

Exemple (code Java 16) :

```
public class TestStatic {

  public void traiter() {
    class MaClasse {
      public static int valeur = 0;

      public static void afficher() {
        System.out.println(valeur);
      }
    };
  }
}
```

Résultat :

```
C:\java>javac TestStatic.java

C:\java>
```

Cela permet notamment à une classe interne de définir des membres qui soient la définition d'un record.

Exemple (code Java 16) :

```
public class TestInnerRecord {

  class MaClasse {
    record MonRecord(String nom) {};
  };
}
```

Résultat :

```
C:\java>javac -version
javac 16.0.1

C:\java>javac TestInnerRecord.java

C:\java>
```

4.9. Les types scellés

En Java, l'héritage permet de créer une hiérarchie de classe. L'héritage facilite la réutilisation du code en permettant d'étendre une classe pour enrichir ou modifier son comportement. Par défaut, il n'y pas de contraintes sur le nombre de classes filles et la profondeur de la hiérarchie.

Cependant, parfois l'objectif d'une hiérarchie de classes n'est pas de réutiliser de code. Parfois, le but est de modéliser les différentes possibilités qui existent dans un domaine, comme par exemple un ensemble fini d'objets. Ainsi le besoin de restreindre l'ensemble des sous-classes est fréquent dans la conception d'une API. Lorsque la hiérarchie de classes est utilisée de cette manière, la restriction de l'ensemble des sous-classes peut rationaliser la modélisation.

Avant Java 17, les développeurs ne pouvaient utiliser que les modificateurs final ou de visibilité pour contrôler l'héritage.

Les classes scellées sont une fonctionnalité standard en Java 17 qui apportent une flexibilité supplémentaire aux développeurs Java lors de la définition de hiérarchies limitées de classes.

Les classes et interfaces scellées (sealed classes and interfaces) permettent de restreindre les classes ou interfaces qui peuvent les étendre ou les implémenter. Cela permet de définir et limiter la hiérarchie de classes qu'il sera possible d'implémenter d'où la dénomination de scellées (sealed) et offre ainsi une possibilité de contrôler le code responsable de leur implémentation. Ainsi une classe ou une interface scellée ne peut être étendue ou implémentée que par les classes et interfaces autorisées à le faire. Cela permet donc de limiter de manière déclarative l'utilisation d'une super-classe.

Les objectifs de cette fonctionnalité sont notamment :

- de permettre à une classe ou une interface de contrôler quelles autres classes ou interfaces peuvent les étendre ou les implémenter
- de fournir un moyen plus déclaratif que les modificateurs d'accès pour restreindre l'héritage d'une super-classe
- de permettre de vérifier l'exhaustivité des classes filles utilisées avec le pattern matching dans une instruction switch

D'autres langages, incluant certains de la JVM comme Kotlin ou Scala, possèdent déjà le concept de classes scellées.

4.9.1. L'historique de la fonctionnalité

Certains mécanismes essentiels au niveau de la plate-forme pour les types scellés ont été introduits dans Java 11 avec la fonctionnalité appelée nestmates.

Les classes scellées ont fait l'objet de deux previews. Les classes scellées ont été proposées en première preview en Java 15 via la JEP 360.

Une seconde preview avec des améliorations est proposée en Java 16 via la JEP 397 avec quelques évolutions :

- spécifier la notion de mot-clé contextuel, remplaçant les notions antérieures d'identifiant restreint et de mot-clé restreint dans la JLS. Introduire les séquences de caractères sealed, non-sealed et permits comme mots-clés contextuels,
- comme pour les classes anonymes et les expressions lambda, les classes locales ne peuvent pas être des sous-classes de classes scellées lors de la détermination des sous-classes autorisées implicitement déclarées d'une classe scellée ou d'une interface scellée,
- l'amélioration de la conversion de référence restrictive pour effectuer une vérification plus stricte des conversions de type cast par rapport aux hiérarchies de types scellées

La JEP 409, introduite en Java 17, ajoute les classes scellées en standard sans changement par rapport à la seconde preview.

4.9.2. Les possibilités pour limiter les classes d'une hiérarchie avant les types scellés

Historiquement en Java, une classe peut être :

- abstraite : pour servir de base à la spécialisation de classes filles
- finale : pour interdire de la sous-typer et donc elle ne peut pas avoir de classes filles

De plus, une nécessité lors de la création de nouvelles classes et interfaces consiste à décider du modificateur de visibilité à utiliser. C'est toujours du cas par cas et, jusqu'à Java 17, les options fournies par le langage n'étaient pas forcément assez granulaires :

- par défaut : la classe n'est visible qu'aux autres types dans le même package (package private)
- public : la classe est visible par tous les autres types

Avant l'ajout des classes scellées, le système de type de Java partait du principe que la réutilisation du code était toujours un objectif. Chaque classe pouvait être étendue par un nombre quelconque de classes filles.

Ainsi par défaut, une classe peut avoir autant de que filles que nécessaire ou ne pas avoir de classe fille du tout. Mais parfois, une hiérarchie de classes est utilisée pour modéliser un ensemble fini et identifié de classes d'un domaine.

Historiquement, il fallait utiliser des astuces plus ou moins élégantes pour restreindre les classes filles d'une classe :

- définir une classe de base avec un constructeur package private qui ne peut être accédé que par d'autres classes du même package et définir les classes filles dans le même package
- utiliser une énumération qui par définition définit un nombre limité d'instance. Mais ce n'est pas le rôle d'une énumération et elle est peu maniable

Dans les versions antérieures à 17 de Java, le langage offrait des options limitées pour le contrôle de l'héritage et ainsi restreindre l'ensemble des sous-classes :

- rendre une classe finale pour l'empêcher d'avoir de classes filles
- rendre une classe ou son constructeur package private (sans modificateur de visibilité) pour qu'elle ne puisse avoir que des sous-classes dans le même package. Cela limite les possibilités d'extensions mais n'empêche pas d'ajouter des classes filles tant qu'elles sont dans le même package que la classe même.

L'exemple ci-dessous définit une classe mère avec la visibilité package-private pour limiter les classes filles.

Exemple :

```
abstract class ClasseMere {...}
```

Exemple :

```
public final class ClasseFilleA extends ClasseMere {...}
```

Exemple :

```
public final class ClasseFilleB extends ClasseMere {...}
```

Avec le mot-clé final sur les classes filles, on empêche la création de descendance.

Le JDK utilise parfois cette technique notamment avec la classe mère AbstractStringBuilder dont la visibilité est package-private et ses classes filles StringBuffer et StringBuilder qui sont final. Cette approche est utile lorsque l'objectif est la réutilisation du code comme dans l'exemple ci-dessus du JDK.

Cependant, cette approche est inutile lorsque l'objectif est de modéliser des alternatives, puisque le code utilisateur ne peut pas accéder à la super-classe. Il n'est pas possible de permettre aux utilisateurs d'accéder à la super-classe sans leur permettre également de l'étendre.

4.9.3. Le rôle des types scellés

Comme le nom le suggère, les classes scellées permettent de restreindre la hiérarchie des classes filles à certains types seulement. Le scellement permet à une classe ou une interface parente de contrôler leurs sous-classes directes, ce qui permet d'avoir un contrôle précis sur la hiérarchie des sous-types autorisés.

Une classe scellée ou une interface peut spécifier une liste de classes ou d'interfaces, qui peuvent les étendre ou les implémenter. Ainsi, aucune autre classe ou interface, à l'exception de celles autorisées par une classe scellée ou une interface scellée, ne peut être leur sous-type.

Cette fonctionnalité a pour but de permettre un contrôle plus fin de l'héritage en Java. Un type scellé est une classe ou une interface qui restreint de manière déclarative les autres classes ou interfaces qui peuvent l'étendre ou l'implémenter. Cela permet d'utiliser une déclaration explicite plutôt que d'utiliser le modificateur final ou la visibilité package-private.

Les classes scellées présentent plusieurs bénéfices et avantages :

- le principal avantage est d'avoir un contrôle fin sur les sous-classes : seules les sous-classes autorisées peuvent étendre la super-classe scellée
- une classe scellée peut être accessible et avoir un contrôle sur son extensibilité
- l'utilisation d'un modificateur dédié permet de facilement identifier l'intention du scellement
- peut permettre une meilleure modélisation en fermant la hiérarchie
- permet au compilateur de faire des vérifications notamment car il peut déterminer l'exhaustivité des sous-classes. Les classes scellées sont ainsi intéressantes en combinaison avec l'utilisation du pattern matching dans le langage Java : le compilateur peut vérifier que tous les cas sont gérés et d'avoir une erreur de compilation si la hiérarchie est modifiée et que le nouveau type n'est pas pris en compte

Les classes scellées permettent de compléter les possibilités actuelles de modélisation d'une hiérarchie de classes. Il est parfois souhaitable d'avoir une hiérarchie fermée (closed hierarchy) : les types scellés expriment l'intention d'avoir une hiérarchie fermée. Les types scellés sont notamment très utiles dans la modélisation d'un domaine car ils permettent d'avoir un contrôle sur une hiérarchie de types.

Les types scellés peuvent aussi avoir une utilité technique (réduction de la complexité, écriture de code plus simple, détection d'erreurs par le compilateur, ...).

Les classes scellées permettent de créer des hiérarchies en dissociant l'accessibilité de l'extensibilité ou pour exposer des interfaces tout en contrôlant toutes les implémentations. Les classes scellées permettent à une super-classe d'être accessible et de limiter de manière déclarative leur extensibilité. Cela permet à une super-classe d'être accessible mais pas arbitrairement extensible puisque ses classes filles directes sont restreintes par définition dans sa déclaration. Donc si une super-classe ou une interface doit être accessible mais pas arbitrairement extensible, il faut la sceller.

Les classes scellées permettent au compilateur et à la JVM de connaître l'exhaustivité des classes filles de premier niveau. Cela permet par exemple de proposer une interface et de connaître précisément toutes les implémentations. Le compilateur peut ainsi connaitre de manière exhaustive les sous-types et ainsi faire des vérifications lors de l'utilisation avec une instruction switch, instanceof ou un cast.

4.9.4. La définition d'une classe scellée

Les types scellés proposent un moyen de restreindre l'implémentation d'une interface ou l'héritage d'une classe, créant ainsi une classe ou une interface scellée. De cette façon, il est possible de fournir des restrictions plus explicites et déclaratives qu'en utilisant les modificateurs final ou de visibilité package-private.

Une fois scellée, une classe ne peut pas avoir d'autres classes filles directes que celles explicitement ou implicitement définies.

La mise en oeuvre des types scellés utilise plusieurs nouveaux mots-clés contextuels pour permettre la déclaration syntaxique : sealed, non-sealed et permits.

Une classe est scellée lorsque le modificateur sealed est utilisé dans sa déclaration. Le mot-clé contextuel sealed est utilisé comme modificateur dans la définition d'une classe, abstraite ou concrète, ou d'une interface pour préciser qu'elle est marquée comme étant scellée.

Pour une déclaration explicite, après d'éventuelle clause implements ou extends, il faut obligatoirement une clause permits suivi de la liste des classes filles permises pour une classe scellée ou suivi de la liste des classes autorisées à implémenter ou des interfaces autorisées à hériter d'une interface scellée, chacune séparée par une virgule.

Exemple (code Java 17) :

```
package fr.jmdoudoux.dej;

public sealed class FormeGeometrique permits Cercle, Triangle {
}
```

Dans l'exemple ci-dessus, la classe FormeGeometrique ne pourra avoir que deux classes filles directes : les classes Cercle et Triangle dont elles pourront hériter. Toute autre classe qui héritera de la classe FormeGeometrique provoquera une erreur de compilation.

Toute autre classe ou interface qui n'est pas dans la liste des types suivant permit et qui tente d'étendre ou implémenter le type scellé provoquera une erreur de compilation ou une erreur d'exécution si le byte code est altéré ou si une classe est générée dynamiquement.

Le mot clé restreint permits, permet de préciser la liste des sous-classes autorisées d'un type scellé. Si aucune liste de sous-types autorisés n'est fournie, le compilateur tente de déduire des sous-types de la même unité de compilation. Si aucun n'est trouvé alors une erreur de compilation est émise.

4.9.5. Les contraintes sur les classes filles

Une classe scellée doit avoir explicitement (avec une clause permits) ou implicitement (en définissant des classes dans le fichier source) des classes filles.

Une classe scellée impose plusieurs contraintes à ses classes filles autorisées :

- elles doivent explicitement et directement hériter de leur classe scellée
- chaque sous-classe autorisée doit utiliser un modificateur pour décrire comment elle propage le scellement initié par sa super-classe en utilisant un des modificateurs final, sealed ou non-sealed

- la classe scellée et ses sous-classes autorisées doivent appartenir au même module, et si elle est déclarée dans un unnamed module au même paquetage. Elles doivent être dans le même module nommé que leur classe mère, éventuellement dans un autre package. Elles doivent être dans le même package que leur classe mère si elles sont dans un unnamed module.

Les classes filles d'une classe scellée peuvent être abstraites ou concrètes.

Comme les clauses extends et permits utilisent des noms de classes, une sous-classe permise et sa super-classe scellée doivent être accessibles l'une à l'autre. Cependant, les sous-classes autorisées ne doivent pas nécessairement avoir la même accessibilité l'une par rapport à l'autre, ou par rapport à la classe scellée. En particulier, une sous-classe peut être moins accessible que la classe scellée.

4.9.5.1. Les modificateurs final, sealed ou non-sealed

La super-classe ne peut pas restreindre la hiérarchie de ses classes petites-filles. Cette restriction doit être gérée au niveau de chaque classe fille.

Exemple (code Java 17) :

```
public abstract sealed class Vehicule permits Voiture, Camion, Moto {
}
```

Toutes les sous-classes directes d'une classe scellée doivent exister et doivent obligatoirement préciser à l'aide d'un modificateur comment le scellement initié par leur classe mère se propage. Les classes qui héritent d'une classe scellée doivent obligatoirement avoir implicitement ou explicitement un des trois modificateurs :

- final : la classe ne peut pas être étendue. Une sous-classe d'une classe scellée peut être déclarée final pour empêcher que sa hiérarchie des classes ne soit étendue. Le modificateur final est une forme de scellement strict qui empêche de créer des classes filles

  Exemple (code Java 17) :

  ```
  public final class Moto extends Vehicule {
  }
  ```

- non-sealed : la hiérarchie des classes filles est libre. La classe permet d'être étendue sans restriction et donc avoir autant de classes filles que nécessaire comme pour une classe classique. Une sous-classe d'une classe scellée peut être déclarée non-sealed afin que sa hiérarchie directe redevienne ouverte à l'extension par des sous-classes. Une classe scellée ne peut pas empêcher ses sous-classes autorisées de faire cela. Le modificateur non-sealed est le premier mot-clé composé avec un trait d'union proposé pour Java

  Exemple (code Java 17) :

  ```
  public non-sealed class Voiture extends Vehicule {
  }
  ```

  Exemple (code Java 17) :

  ```
  public class Berline extends Voiture {
  }
  ```

  Exemple (code Java 17) :

  ```
  public class Monospace extends Voiture {
  }
  ```

- sealed : la classe ne peut être étendue que par les classes fournies dans sa clause permits : la hiérarchie de ces classes filles est limitée à celle autorisée. Une sous-classe d'une classe scellée peut être déclarée sealed pour permettre à sa hiérarchie directe d'être étendue mais d'une manière restreinte à la liste des classes filles autorisées

Exemple (code Java 17) :

```
public sealed class Camion extends Vehicule permits CamionCiterne, CamionBenne {
}
```

Exemple (code Java 17) :

```
public final class CamionCiterne extends Camion {
}
```

Exemple (code Java 17) :

```
public final class CamionBenne extends Camion {
}
```

Un seul de ces modificateurs doit être utilisé dans chaque classe fille d'une classe scellée : ils ne peuvent pas se combiner. Il n'est pas possible qu'une classe fille soit à la fois :

- sealed (impliquant des sous-classes) et final (impliquant aucune sous-classe)
- ou non-sealed (impliquant des sous-classes) et final (impliquant aucune sous-classe)
- ou sealed (impliquant des sous-classes restreintes) et non-sealed (impliquant des sous-classes non restreintes)

Une classe scellée peut avoir des classes filles abstraites du moment qu'elles soient sealed ou non-sealed. Dans ce cas, elles ne peuvent évidemment pas être final.

4.9.5.2. La déclaration implicite des classes filles

Si les classes filles sont peu nombreuses, il peut être pratique de les déclarer dans le même fichier source que la classe scellée :

- soit sous la forme de classes internes imbriquées dans la classe mère
- soit sous la forme de classes auxiliaires

Lorsqu'elles sont déclarées de cette manière, la classe scellée peut omettre la directive permits, et le compilateur déterminera les sous-classes autorisées à partir des déclarations dans le fichier source.

Les sous-classes peuvent être des classes auxiliaires.

Exemple (code Java 17) :

```
package fr.jmdoudoux.dej;

public sealed class Forme {
}

final class Triangle extends Forme {
}

final class Cercle extends Forme {
}
```

Les sous-classes peuvent aussi être des classes imbriquées.

Exemple (code Java 17) :

```
package fr.jmdoudoux.dej;

public sealed class Forme {

  final class Triangle extends Forme {
  }
  final class Cercle extends Forme {
  }
}
```

Dans les deux exemples ci-dessus, le compilateur détermine que la classe scellée Forme ne peut avoir que deux classes filles : Triangle et Cercle.

Si les classes filles sont déclarées dans le même fichier source, la clause permits est donc facultative.

Il est aussi possible de mixer la définition des classes filles sous la forme de classes internes et de classes auxiliaires.

Exemple (code Java 17) :

```
public sealed class Forme {

  final class Triangle extends Forme {
  }
}

final class Cercle extends Forme {
}
```

La mise en oeuvre de cette possibilité devrait être un cas rare.

Il n'est pas possible de mixer déclaration implicite et explicite donc de déclarer une ou plusieurs classe filles dans une clause permits et d'autres classes filles sous la forme de classes internes ou de classes auxiliaires.

Exemple (code Java 17) :

```
public sealed class Forme permits Cercle {

  final class Triangle extends Forme {
  }
}

final class Cercle extends Forme {
}
```

Résultat :

```
C:\java>javac Forme.java
Forme.java:3: error: class is not allowed to extend sealed class: Forme (as it is not listed
in its permits clause)
  final class Triangle extends Forme {
        ^
1 error
```

Il est cependant possible que les classes définies dans la clause permits soient des classes internes.

Exemple (code Java 17) :

```
public sealed class Forme permits Cercle, Forme.Triangle {

  final class Triangle extends Forme {
  }
}

final class Cercle extends Forme {
}
```

4.9.5.3. La localisation des classes filles

Les types scellés et leurs implémentations sont généralement un ensemble de classes développées ensemble, puisque l'idée est que le développeur de la classe scellée est capable de contrôler l'ensemble de ses sous-classes. Cela entraîne des restrictions quant à l'endroit où les classes scellées peuvent être définies.

Comme il existe un couplage fort entre une classe scellée et ses classes filles, le langage Java impose des contraintes sur la localisation de la hiérarchie de classes afin de garantir leur co-maintenance et le fait qu'elles soient accessibles à la compilation. Les classes scellées et leurs classes filles doivent être obligatoirement dans le même module.

Les classes filles précisées dans la clause permits d'une classe scellée ont des restrictions concernant leur localisation :

- dans un module nommé : elles doivent être dans le même module que leur classe mère scellée, éventuellement dans des packages différents
- dans l'unnamed module : elles doivent être dans le même package que leur classe mère scellée

4.9.6. Les interfaces scellées

Avant l'introduction des classes scellées, une classe publique pouvait définir un constructeur privé ou package-private pour restreindre son extensibilité, ce qui n'était pas le cas des interfaces qui ne peuvent pas définir de constructeurs.

Les interfaces scellées sont des interfaces qui définissent explicitement les interfaces filles et les classes ou les records qui peuvent l'implémenter.

Une interface peut être scellée en utilisant le modificateur sealed dans sa définition. Après une éventuelle clause extends, il faut obligatoirement une clause permits qui précise les classes d'implémentations et les interfaces filles possibles.

Exemple (code Java 17) :

```
public sealed interface Paiement permits Espece, Cheque, CarteBancaire  {
}

final class Espece implements Paiement {}

final class Cheque implements Paiement {}

final class CarteBancaire implements Paiement {}
```

La directive permits dans la définition permet de définir les classes qui sont autorisées à implémenter l'interface scellée. Les classes autorisées à implémenter une interface scellée doivent avoir un des modificateurs final, sealed ou non-sealed

Exemple (code Java 17) :

```
public sealed interface Paiement permits Espece, Cheque {}

final class Espece implements Paiement {}

sealed abstract class Cheque implements Paiement permits ChequeNonBarre, ChequeDeBanque {}

final class ChequeNonBarre extends Cheque {}

final class ChequeDeBanque extends Cheque {}
```

La clause permits dans la définition permet aussi de définir les interfaces qui sont autorisées à hériter de l'interface scellée.

Comme il n'est pas possible de déclarer une interface avec le modificateur final, car les interfaces sont destinées à être implémentées, la définition des interfaces autorisées doit avoir soit le modificateur sealed soit le modificateur non-sealed.

Exemple (code Java 17) :

```
public sealed interface Paiement permits Espece, Cheque {}

non-sealed interface Espece extends Paiement {}

sealed interface Cheque extends Paiement permits ChequeNonBarre, ChequeDeBanque {}

final class ChequeNonBarre implements Cheque {}

final class ChequeDeBanque implements Cheque {}
```

La clause permits d'une interface scellée peut contenir des classes et des interfaces.

Exemple (code Java 17) :

```
sealed interface Vehicule permits Voiture, Camion, Bus {}

non-sealed interface Voiture extends Vehicule {}

non-sealed interface Camion extends Vehicule {}

final class Bus implements Vehicule {}
```

4.9.7. Les types scellés et les records

Comme les records héritent déjà implicitement de la classe java.lang.Record, il n'est pas possible d'utiliser des records comme classes filles d'une classe scellée.

Mais un record peut implémenter une interface et cette interface peut être scellée, d'autant qu'un record est implicitement final.

Comme les records sont implicitement final, il n'est pas possible d'utiliser les mots clés sealed et non-sealed et le mot clé final est facultatif.

Il n'est pas nécessaire de redéfinir explicitement des méthodes générées par le compilateur qui soient définies dans l'interface sauf pour des besoins différents de ce ceux du code généré.

Exemple (code Java 17) :

```
package fr.jmdoudoux.dej.typescelles;

public sealed interface Personne permits Employe {

  String nom();
  String prenom();
  String getNomPrenom();
}
```

Exemple (code Java 17) :

```
package fr.jmdoudoux.dej.typescelles;

public record Employe(String nom, String prenom) implements Personne {

  @Override
  public String getNomPrenom() {
    return nom + " " + prenom;
  }
}
```

4.9.8. Les vérifications à l'exécution

L'ajout des classes scellées impose des changements dans la JVM et les fichiers .class.

Le fichier .class d'une classe scellée possède un attribut PermittedSubclasses qui contient les sous-classes autorisées. La liste des sous-classes autorisées est obligatoire dans le fichier .class d'une classe scellée même lorsque les sous-classes autorisées sont déduites par le compilateur, ces sous-classes déduites sont explicitement incluses dans l'attribut PermittedSubclasses.

Le fichier .class d'une sous-classe autorisée ne contient pas de nouveaux attributs.

En plus des vérifications faites par le compilateur sur les types scellés, il y a aussi une vérification à l'exécution par la JVM. La JVM vérifie les classes et les interfaces scellées à l'exécution pour s'assurer que des classes ou interfaces filles non autorisées ne sont pas définies dynamiquement ou par du code qui n'a pas été recompilé.

Une exception de type IncompatibleClassChangeError est levée par la JVM lors de la définition d'une classe dont la super-classe ou une super-interface est scellée et qu'elle n'est pas définie dans leur attribut PermittedSubclasses.

4.9.9. Les situations illicites

Plusieurs situations illicites seront détectées par le compilateur javac et provoqueront l'émission d'une erreur de compilation.

Il n'est pas possible d'hériter d'une classe scellée qui ne l'autorise pas.

Exemple (code Java 17) :

```
public sealed class Forme permits Cercle {
}

final class Cercle extends Forme {
}

final class Triangle extends Forme {
}
```

Résultat :

```
C:\java>javac Forme.java
Forme.java:7: error: class is not allowed to extend sealed class: Forme (as it is not listed
in its permits clause)
final class Triangle extends Forme {
      ^
1 error
```

Une classe scellée doit avoir au moins une classe fille autorisée.

Exemple (code Java 17) :

```
public sealed class Operation { }
```

Résultat :

```
C:\java>javac Operation.java
Operation.java:1: error: sealed class must have subclasses
public sealed class Operation { }
              ^
1 error
```

Une interface scellée doit avoir au moins une classe qui l'implémente ou une interface fille.

Exemple (code Java 17) :

```
public sealed interface Operation { }
```

Résultat :

```
C:\java>javac Operation.java
Operation.java:1: error: sealed class must have subclasses
public sealed interface Operation { }
              ^
1 error
```

Les classes spécifiées dans une clause permits doivent avoir un nom canonique, sinon une erreur est émise par le compilateur. Cela signifie que les classes anonymes et les classes locales ne peuvent pas être des sous-types autorisés d'une classe scellée.

Il n'est donc pas possible de créer une classe anonyme qui soit l'implémentation d'un type scellé.

Exemple (code Java 17) :

```
public class Main {

  public static void main(String[] args) {
    Forme forme = new Forme() {};
  }
}
```

Résultat :

```
C:\java>javac Main.java
Main.java:4: error: local classes must not extend sealed classes
    Forme forme = new Forme() {};
                              ^
1 error
```

Si une interface scellée respecte les contraintes pour être une interface fonctionnelle alors il n'est pas possible d'en fournir une implémentation sous la forme d'une expression Lambda. Une interface fonctionnelle ne peut en aucun cas être scellée : cela a du sens car le but d'une interface fonctionnelle est de pouvoir proposer un nombre indéfini d'implémentations.

Exemple (code Java 17) :

```
public sealed interface Operation permits Addition {

  long appliquer(long a, long b);

  public static void main(String[] args) {
    Operation Soustraction = (a,b) -> a-b;
  }
}

final class Addition implements Operation {
  @Override
  public long appliquer(long a, long b) {
    return a+b;
  }
}
```

Résultat :

```
C:\java>javac Operation.java
Operation.java:6: error: incompatible types: Operation is not a functional interface
        Operation Soustraction = (a,b) -> a-b;
                                 ^
1 error
```

Une classe finale, qu'elle soit explicitement déclarée avec le modificateur final ou implicitement finale comme les classes enum ou record, ne peut pas être scellée.

Exemple (code Java 17) :

```
public final sealed class Operation { }
```

Résultat :

```
C:\java>javac Operation.java
Operation.java:1: error: illegal combination of modifiers: final and sealed
public final sealed class Operation { }
                    ^
1 error
```

Une classe ne peut pas avoir le modificateur non-sealed si elle n'hérite pas d'une classe scellée marquée avec le modificateur sealed.

Exemple (code Java 17) :

```
public non-sealed class Operation { }
```

Résultat :

```
C:\java>javac Operation.java
Operation.java:1: error: non-sealed modifier not allowed here
public non-sealed class Operation { }
                  ^
  (class Operation does not have any sealed supertypes)
1 error
```

4.9.10. L'opérateur instanceof et les classes scellées

L'opérateur instanceof évalue la possibilité qu'une instance soit d'un type spécifique.

Exemple :

```
public class MaClasse implements MonInterface {

  public void traiter(MaClasse mc) {
    if (mc instanceof MonInterface) {
      MonInterface mi = (MonInterface) mc;
      System.out.println(mi);
    }
  }
}

interface MonInterface {}
```

Les règles de conversion des types reposent sur une notion d'extensibilité ouverte : le système de type Java ne suppose pas un monde fermé. Par défaut, les classes et les interfaces peuvent être étendues à l'avenir. Ainsi, Java est très permissif quant aux types qui sont autorisés dans ces types d'expressions.

Exemple :

```
public class MaClasse {

  public void traiter(MaClasse mc) {
    if (mc instanceof MonInterface) {
      MonInterface mi = (MonInterface) mc;
      System.out.println(mi);
    }
  }
}
```

```
interface MonInterface {}
```

Ce code se compile sans erreur. Le compilateur javac ne peut pas exclure la possibilité qu'une instance de MaClasse soit de type Moninterface, car il est possible pour une sous-classe de MaClasse d'implémenter l'interface MonInterface.

Cependant, le compilateur peut invalider cette possibilité dans certains cas et émettre une erreur.

Si le type de la variable n'est pas compatible et ne peut pas être étendue car elle est déclarée avec le modificateur final, alors le compilateur émet une erreur.

Exemple :

```
public final class MaClasse {

  public static void traiter(MaClasse mc) {
    if (mc instanceof MonInterface) {
      MonInterface mi = (MonInterface) mc;
      System.out.println(mi);
    }
  }
}

interface MonInterface {}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:4: error: incompatible types: MaClasse cannot be converted to MonInterface
    if (mc instanceof MonInterface) {
        ^
MaClasse.java:5: error: incompatible types: MaClasse cannot be converted to MonInterface
      MonInterface mi = (MonInterface) mc;
                                       ^
2 errors
```

Dans l'exemple ci-dessus, le compilateur sait qu'il ne peut pas y avoir de sous-classe de MaClasse et comme MaClasse n'implémente pas l'interface MonInterface, il n'est pas possible qu'une instance de type MaClasse soit compatible avec MonInterface.

Le support des classes scellées entraîne une modification de la définition de la conversion de référence restrictive pour que le compilateur navigue dans les hiérarchies scellées et de déterminer, au moment de la compilation, les conversions qui ne sont pas possibles.

Si la classe est scellée et que le compilateur peut déterminer toutes les classes filles de manière exhaustive, alors il émet une erreur si aucune des classes n'implémente l'interface et ne permet à au moins une classe fille de le faire.

Exemple (code Java 17) :

```
public sealed class MaClasse permits MaClasseFille {

  public static void traiter(MaClasse mc) {
    if (mc instanceof MonInterface) {
      MonInterface mi = (MonInterface) mc;
      System.out.println(mi);
    }
  }
}

final class MaClasseFille extends MaClasse {}

interface MonInterface {}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:4: error: incompatible types: MaClasse cannot be converted to MonInterface
    if (mc instanceof MonInterface) {
        ^
MaClasse.java:5: error: incompatible types: MaClasse cannot be converted to MonInterface
      MonInterface mi = (MonInterface) mc;
                                       ^
2 errors
```

Dans l'exemple ci-dessus, la classe scellée et son unique classe fille finale n'implémentent pas l'interface, ce qui rend impossible la possibilité de caster vers l'interface sans erreur à l'exécution. Le compilateur anticipe cette situation certaine en émettant une erreur.

Une des classes filles de la hiérarchie peut rouvrir l'héritage avec le modificateur non-sealed. Dans ce cas, il est possible qu'une des classes filles de la hiérarchie non-sealed implémente l'interface. Le code se compile sans erreur.

Exemple (code Java 17) :

```
public sealed class MaClasse permits MaClasseFille {

  public static void traiter(MaClasse mc) {
    if (mc instanceof MonInterface) {
      MonInterface mi = (MonInterface) mc;
      System.out.println(mi);
    }
  }
}

non-sealed class MaClasseFille extends MaClasse {}

interface MonInterface {}
```

Résultat :

```
C:\java>javac MaClasse.java

C:\java>
```

4.10. La gestion dynamique des objets

Tout objet appartient à une classe et Java sait la reconnaitre dynamiquement.

Java fournit dans son API un ensemble de classes qui permettent d'agir dynamiquement sur des classes. Cette technique est appelée introspection et permet :

- de décrire une classe ou une interface : obtenir son nom, sa classe mère, la liste de ses méthodes, de ses variables de classe, de ses constructeurs et de ses variables d'instances
- d'agir sur une classe en envoyant à un objet Class des messages comme à tout autre objet. Par exemple, créer dynamiquement à partir d'un objet Class une nouvelle instance de la classe représentée

Voir le chapitre «La gestion dynamique des objets et l'introspection» pour obtenir plus d'informations.

5. Les génériques (generics)

Chapitre 5

Niveau : 🟩🟩⬜⬜⬜⬜ Elémentaire

Les génériques en Java (generics) sont un ensemble de caractéristiques du langage liées à la définition et à l'utilisation de types et de méthodes génériques. En Java, les types ou méthodes génériques diffèrent des types et méthodes ordinaires dans le fait qu'ils possèdent des paramètres de type.

Les génériques ont été introduits dans le but d'ajouter une couche supplémentaire d'abstraction sur les types et de renforcer la sécurité des types. Les génériques permettent d'accroître la lisibilité du code et surtout d'en renforcer la sécurité grâce à un typage plus exigeant. Ils permettent de préciser explicitement le type d'un objet et rendent le cast vers ce type implicite. Cette fonctionnalité est spécifiée dans la JSR 14 et intégrée dans Java 1.5.

Les génériques permettent à un type ou à une méthode d'opérer sur des objets de différents types tout en assurant la sécurité des types au moment de la compilation. Les génériques permettent de définir certains types et des méthodes pour lesquelles un ou plusieurs types utilisés sont précisés lors de leur utilisation en tant que paramètre.

Ils permettent par exemple de spécifier quel type d'objets une collection peut contenir et ainsi éviter l'utilisation d'un cast pour obtenir un élément de la collection.

L'inconvénient majeur du cast est que celui-ci ne peut être vérifié qu'à l'exécution et qu'il peut échouer en levant une exception de type ClassCastException. Avec l'utilisation des génériques, le compilateur pourra réaliser cette vérification lors de la phase de compilation : la sécurité du code est ainsi renforcée.

Exemple (code Java 1.4) :

```java
import java.util.*;

public class SansGenerique {

  public static void main(String[] args) {

    List liste = new ArrayList();
    String valeur = null;
    for(int i = 0; i < 10; i++) {
      valeur = ""+i;
      liste.add(valeur);
    }

    for (Iterator iter = liste.iterator(); iter.hasNext(); ) {
      valeur = (String) iter.next();
      System.out.println(valeur.toUpperCase());
    }
  }
}
```

L'utilisation des génériques permet au compilateur de faire la vérification au moment de la compilation et ainsi de s'assurer d'une exécution correcte. Ce mécanisme permet de s'assurer que les objets contenus dans la collection seront homogènes.

La syntaxe pour mettre en oeuvre les génériques utilise les symboles < et > pour préciser le ou les types des objets à utiliser. Seuls des objets peuvent être utilisés avec les génériques : si un type primitif est utilisé dans les génériques, une erreur de type « unexpected type » est générée lors de la compilation.

Exemple (code Java 5.0) :

```
import java.util.*;

public class AvecGenerique {

  public static void main(String[] args) {

    List<String> liste = new ArrayList();
    String valeur = null;
    for(int i = 0; i < 10; i++) {
      valeur = ""+i;
      liste.add(valeur);
    }

    for (Iterator<String> iter = liste.iterator(); iter.hasNext(); ) {
      System.out.println(iter.next().toUpperCase());
    }
  }
}
```

Si un objet de type différent de celui déclaré dans le générique est utilisé dans le code, le compilateur émet une erreur lors de la compilation.

Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.Date;
import java.util.Iterator;
import java.util.List;

public class AvecGenerique {

  public static void main(String[] args) {

    List<String> liste = new ArrayList();
    String valeur = null;
    for (int i = 0; i < 10; i++) {
      liste.add(valeur);
      liste.add(new Date());
    }

    for (Iterator<String> iter = liste.iterator(); iter.hasNext();) {
      System.out.println(iter.next().toUpperCase());
    }
  }
}
```

Résultat :

```
C:\java>javac AvecGenerique.java
AvecGenerique.java:14: error: no suitable method found for add(Date)
      liste.add(new Date());
           ^
    method List.add(int,String) is not applicable
      (actual and formal argument lists differ in length)
    method List.add(String) is not applicable
      (actual argument Date cannot be converted to String by method invocation c
onversion)
Note: AvecGenerique.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.


1 error
```

Un concept important dans la mise en oeuvre des génériques en Java est l'effacement de type. Cela signifie que les informations relatives aux génériques ne sont pas incluses dans le bytecode généré par le compilateur. Le bytecode ne contient aucune information relative aux génériques pour des raisons de compatibilité.

Les génériques en Java sont un sucre syntaxique dans le code pour la sécurité de type car toutes ces informations de type ne sont pas incluses dans le bytecode à cause de la mise en oeuvre par le compilateur de l'effacement de type.

Donc un point important concernant les génériques : les génériques Java n'existent plus à l'exécution, c'est un concept utilisable uniquement dans le code source et exploité par le compilateur. Les génériques en Java ont été ajoutés pour fournir une vérification de type au moment de la compilation. Pour maintenir la compatibilité, ils n'ont aucune utilité au moment de l'exécution.

Depuis leur introduction dans le langage Java, les types du JDK utilisent de manière importante les génériques. Le JDK propose de nombreux types génériques, notamment dans l'API Collection mais aussi dans de nombreuses autres API. Il est aussi possible de définir ses propres types génériques.

Les génériques sont l'une des fonctionnalités les plus controversées et surement l'une des plus complexe du langage Java. Généralement les génériques sont considérés comme facile par les développeurs Java : c'est vrai lorsque l'on se place du point de vue de l'utilisation de classes génériques comme les classes de l'API Collections. Mais la mise en oeuvre des génériques pour définir des classes génériques et beaucoup plus compliqué et peut même devenir très complexes dans certains cas.

Ce chapitre contient plusieurs sections :

- Le besoin des génériques
- La définition des concepts
- L'utilisation de types génériques
- La définition de types génériques
- La pollution du heap (Heap Pollution)
- La mise en oeuvre des génériques
- Les méthodes et les constructeurs génériques
- Les paramètres de type bornés (bounded type parameters)
- Les paramètres de type avec wildcard
- Les bornes multiples (Multiple Bounds) avec l'intersection de types
- L'effacement de type (type erasure)
- La différence entre l'utilisation d'un argument de type et un wildcard
- Les conséquences et les effets de bord de l'effacement de type
- Les restrictions dans l'utilisation des génériques

5.1. Le besoin des génériques

De nombreuses classes de l'API Collection possèdent des paramètres de type Object et renvoient les valeurs des méthodes sous forme d'Object. Sous cette forme, elles peuvent prendre n'importe quel type comme argument et retourner le même. Elles permettent donc de contenir des valeurs hétérogènes, c'est-à-dire qu'elles ne sont pas d'un type similaire particulier. Pour garantir que seul un type d'objet est ajouté dans la collection, il fallait tester le type avant d'invoquer les méthodes d'ajouts.

Exemple :

```
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;

public class MaClasse {

  public static void main(String[] args) {
    List liste = new ArrayList();
    liste.add("test");

    Iterator it = liste.iterator();
    while(it.hasNext()) {
      String valeur = it.next();
      System.out.println(valeur);
    }
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:13: error: incompatible types: Object cannot be converted to String
      String valeur = it.next();
                             ^
Note: MaClasse.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
1 error
```

Pour supporter tous les types d'objet, le type Object est utilisé pour stocker les valeurs dans la collection. Il faut donc obligatoirement utiliser un cast vers le type d'objet pour pouvoir le manipuler.

Exemple :

```
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;

public class MaClasse {

  public static void main(String[] args) {
    List liste = new ArrayList();
    liste.add("test");

    Iterator it = liste.iterator();
    while(it.hasNext()) {
      String valeur = (String) it.next();
      System.out.println(valeur);
    }
  }
}
```

Cela n'empêche cependant pas d'ajouter des objets d'un autre type et donc d'avoir une exception de type CastClassException à l'exécution.

Exemple :

```
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;

public class MaClasse {

  public static void main(String[] args) {
    List liste = new ArrayList();
    liste.add("test");
    liste.add(new Integer(1));

    Iterator it = liste.iterator();
    while(it.hasNext()) {
      String valeur = (String) it.next();
      System.out.println(valeur);
    }
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java
Note: MaClasse.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

```
C:\java>java MaClasse
test
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be
 cast to java.lang.String
        at MaClasse.main(MaClasse.java:14)
```

5.1.1. L'apport des génériques

Il serait beaucoup plus pratique et sûr de pouvoir exprimer l'intention d'utiliser des types spécifiques et que le compilateur puisse garantir l'exactitude de ces types. C'est ce qu'il est possible de faire avec les génériques.

Les génériques sont des fonctionnalités du langage qui permettent de définir et d'utiliser des types et des méthodes génériques. Un type ou une méthode générique définit un paramètre de type qui sera précisé lors de la création d'une instance ou de l'invocation de la méthode.

Les génériques permettent au compilateur de faire des vérifications sur l'utilisation de types, dénommées sécurité de type (type safety) et améliore la robustesse du code en évitant l'utilisation de cast qui pouvaient lever des exceptions de type ClassCastException.

Le compilateur vérifie les affectations de types avec une conversion si elle est possible. Si les types sont incompatibles, il est nécessaire d'utiliser un cast pour éviter une erreur de compilation. Cela indique au compilateur que l'affectation est valide mais elle ne garantit pas qu'elle va irrémédiablement réussir à l'exécution. Si ce n'est pas le cas, une exception de type ClassCastException est levée à l'exécution.

Les génériques permettent de définir des classes, des interfaces, des records et des méthodes qui utilisent un type précisé à la création de l'instance ou à l'invocation de la méthode.

La définition et l'utilisation d'un type générique se fait en utilisant l'opérateur diamant <> dans lequel on précise le type à utiliser.

L'utilisation des génériques améliore significativement la robustesse du code et le rend plus lisible.

5.1.2. Les génériques et l'API Collection

Avant Java 5, il est possible d'ajouter des instances de différents types dans une collection car les méthodes acceptent et retournent des instances de type Object. Pour utiliser ses instances, il est alors nécessaire de faire un cast qui peut échouer à l'exécution si le type ne correspond pas.

Exemple :

```
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;

public class MaClasse {

  public static void main(String[] args) {
    List liste = new ArrayList();
    liste.add("test");
    liste.add(new Integer(1));

    Iterator it = liste.iterator();
    while(it.hasNext()) {
      String valeur = (String) it.next();
      System.out.println(valeur);
    }
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java
Note: MaClasse.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

C:\java>java MaClasse
test
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be
 cast to java.lang.String
        at MaClasse.main(MaClasse.java:14)
```

En Java 5, l'API Collection a été revue pour utiliser les génériques : les types de l'API Collections sont génériques ce qui permet de préciser le type des objets pouvant être stockés et récupérés d'une collection. L'API Collections est probablement celle qui contient les classes et interfaces génériques les plus utilisées. C'est aussi un cas d'usage très simple et pratique, puisque ce sont des conteneurs d'objets généralement de même type.

Le compilateur va vérifier que seules des instances du type précisé seront ajoutées dans la collection, renforçant ainsi la fiabilité et la robustesse du code. Grâce à cette vérification, il n'est plus nécessaire de faire un cast lorsque l'on récupère une valeur de la collection.

Exemple :

```
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;

public class MaClasse {

  public static void main(String[] args) {
    List<String> liste = new ArrayList<String>();
    liste.add("test");

    Iterator<String> it = liste.iterator();
    while(it.hasNext()) {
      String valeur = it.next();
      System.out.println(valeur);
    }
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java

C:\java>java MaClasse
test
```

Les génériques ne sont pas utiles que dans les types de la classe Collection qui encapsulent des objets.

5.2. La définition des concepts

L'utilisation des génériques met en oeuvre plusieurs concepts :

- un type générique (generic type) est une classe, une interface ou un record qui est paramétrée avec au moins un paramètre de type, par exemple, ArrayList<T>
- un paramètre de type (type parameter) ou variable de type est une variable définie dans une section entre <>, par exemple T dans ArrayList<T>
- un type paramétré (parameterized type) est une classe où le paramètre de type est défini avec un type comme argument, par exemple, ArrayList<Long>
- un argument de type (type argument) est le type précisé dans une section entre <> dans un type paramétré, par exemple Long dans ArrayList<Long>
- un type brut (raw type) est un type générique qui n'est paramétré avec rien, comme new ArrayList()

5.3. L'utilisation de types génériques

L'utilisation des génériques permet de rendre le code plus lisible et plus sûr notamment car il n'est plus nécessaire d'utiliser un cast et de définir une variable intermédiaire.

Les génériques peuvent être utilisés avec :

- des types (classes, interfaces, records)
- des méthodes et des constructeurs

Les types génériques sont instanciés pour former des types paramétrés en fournissant des arguments de type réels qui remplacent les paramètres de type formels utilisés dans leur définition. Une classe comme ArrayList<E> est un type générique, qui possède un paramètre de type E. Les instanciations, telles que ArrayList<Integer> ou ArrayList<String>, sont appelées types paramétrés, et Integer et String sont les arguments de type réels respectifs.

Exemple (code Java 5.0) :

```
public class MaClasseGenerique<T1, T2> {
  private T1 param1;
  private T2 param2;

  public MaClasseGenerique(T1 param1, T2 param2) {
    this.param1 = param1;
    this.param2 = param2;
  }

  public T1 getParam1() {
    return this.param1;
  }

  public T2 getParam2() {
    return this.param2;
  }
}
```

Lors de l'utilisation de la classe, il faut définir les types des paramètres de types à utiliser.

Exemple (code Java 5.0) :

```
import java.util.*;

public class TestClasseGenerique {

  public static void main(String[] args) {
    MaClasseGenerique<Integer, String> maClasse =
          new MaClasseGenerique<Integer, String>(1, "valeur 1");
    Integer param1 = maClasse.getParam1();
    String  param2 = maClasse.getParam2();
  }
}
```

Le principe est identique avec les interfaces.

La syntaxe utilisant les caractères < et > se situe toujours après l'entité qu'elle concerne.

Exemple (code Java 5.0) :

```
    MaClasseGenerique<Integer, String> maClasse =
        new MaClasseGenerique<Integer, String>(1, "valeur 1");
    MaClasseGenerique<Integer, String>[] maClasses;
```

Le cast peut être utilisé avec les types génériques en utilisant le type paramétré dans le cast.

5.3.1. L'opérateur diamant

Avant Java 7, il était obligatoire, lors de l'instanciation d'une classe utilisant les génériques, de préciser l'argument de type dans la déclaration de la variable et dans l'invocation du constructeur.

Exemple (code Java 5.0) :

```
    Map<Integer, String> maMap = new HashMap<Integer, String>();
```

Avec Java 7, il est possible de remplacer les types génériques utilisés lors de l'invocation du constructeur pour créer une instance par le simple opérateur <>, dit opérateur diamant (diamond operator), qui permet donc de demander une inférence de type par le compilateur.

Ceci est possible tant que le compilateur peut déterminer les arguments utilisés dans la déclaration du type paramétré à créer.

Exemple (code Java 7) :

```
    Map<Integer, String> maMap = new HashMap<>();
```

L'utilisation de l'opérateur diamant n'est pas obligatoire. Si l'opérateur diamant est omis, le compilateur génère un warning de type unchecked conversion.

Exemple (code Java 7) :

```
      Map<Integer, String> maMap = new HashMap();
      // unchecked conversion warning
```

La déclaration et l'instanciation d'un type qui utilise les génériques peuvent être verbeux. L'opérateur diamant est très pratique lorsque les types génériques utilisés sont complexes : le code est moins verbeux et donc plus simple à lire

Exemple (code Java 5.0) :

```
    Map<Integer, Map<String, List<String>>> maCollection = new HashMap<Integer,
      Map<String, List<String>>>();
```

L'inconvénient dans le code Java 5 ci-dessus est que le type générique utilisé doit être utilisé dans la déclaration et dans la création de l'instance : cette utilisation est redondante. Avec Java 7 et l'utilisation de l'opérateur diamant, le compilateur va automatiquement reporter le type utilisé dans la déclaration.

Exemple (code Java 7) :

```
   Map<Integer,Map<String, List<String>>> maCollection = new HashMap<>();
```

Cette inférence de type réalisée avec l'opérateur diamant n'est utilisable qu'avec un constructeur.

L'utilisation de l'opérateur est conditionnée par le fait que le compilateur puisse déterminer le type. Dans le cas contraire, une erreur de compilation est émise.

Exemple (code Java 7) :

```
package fr.jmdoudoux.dej;

import java.util.ArrayList;
import java.util.List;

public class TestOperateurDiamant {
  public static void main(String[] args) {
    List<String> liste = new ArrayList<>();
    liste.add("element1");
    liste.addAll(new ArrayList<>());
  }
}
```

Résultat :

```
C:\java>javac com\jmdoudoux\test\TestOperateurDiamant.java
com\jmdoudoux\test\TestOperateurDiamant.java:11: error: no suitable method found
 for addAll(ArrayList<Object>)
    liste.addAll(new ArrayList<>());
        ^
    method List.addAll(int,Collection<? extends String>) is not applicable
      (actual and formal argument lists differ in length)
    method List.addAll(Collection<? extends String>) is not applicable
      (actual argument ArrayList<Object> cannot be converted to Collection<? extends
String> by method invocation conversion)


1 error
```

La compilation de l'exemple ci-dessus échoue puisque la méthode addAll() attend en paramètre un objet de type Collection<String>.

L'exemple suivant compile car le compilateur peut explicitement déterminer le type à utiliser avec l'opérateur diamant.

Exemple (code Java 7) :

```
package fr.jmdoudoux.dej;

import java.util.ArrayList;
import java.util.List;

public class TestOperateurDiamant {
  public static void main(String[] args) {
    List<String> liste = new ArrayList<>();
    liste.add("element1");

    List<? extends String> liste2 = new ArrayList<>();
    liste2.add("element2");
    liste.addAll(liste2);
  }
}
```

L'opérateur diamant peut aussi être utilisé lors de la création d'une nouvelle instance dans une instruction return : le compilateur peut déterminer le type à utiliser par rapport à la valeur de retour de la méthode.

Exemple (code Java 7) :

```
  public Map<String, List<String>> getParametres(String contenu) {
    if (contenu == null) {
      return new HashMap<>();
    }
    // ...
  }
```

Avant Java 9, il n'est pas possible d'utiliser l'opérateur diamant lors de la définition d'une classe anonyme interne.

Exemple (code Java 5.0) :

```
public interface Operation<T> {

  T ajouter(T a, T b);
}
```

Exemple (code Java 9) :

```
public class MainOperation {

  public static void main(String[] args) {
    Operation<Integer> op = new Operation<>() {
      public Integer ajouter(Integer a, Integer b) {
        return a+b;
      }
    };
  }
}
```

Résultat :

```
C:\java>javac -version
javac 1.8.0_252

C:\java>javac MainOperation.java
MainOperation.java:4: error: cannot infer type arguments for Operation<T>
    Operation<Integer> op = new Operation<>() {
                                         ^
  reason: cannot use '<>' with anonymous inner classes
  where T is a type-variable:
    T extends Object declared in interface Operation
1 error
```

A partir de Java 9, il est possible d'utiliser l'opérateur diamant dans la déclaration d'un classe anonyme interne.

Résultat :

```
C:\java>javac -version
javac 9.0.1

C:\java>javac MainOperation.java

C:\java>
```

5.3.2. Le type brut (raw type)

Un type brut (raw type) est le nom donné à une classe ou une interface générique sans aucun argument de type précisé.

Pour des raisons de compatibilité ascendante, l'affectation d'un type paramétré à son type brut est autorisée. Lors de l'utilisation des types bruts, le comportement est similaire à celui pré-générique.

Si le type générique n'est pas fourni au moment de la création d'une instance, le compilateur émet un avertissement et le type générique devient Object. Le compilateur ne peut plus faire de vérification sur le type et il sera nécessaire de faire un cast.

Le rôle des types brutes est de conserver la rétrocompatibilité avec le code antérieure à Java 5 : il n'est donc pas recommandé de les utiliser dans un autre contexte.

L'affectation d'une instance d'une classe générique à une variable dont le type est brut (raw type) provoque l'émission d'un avertissement de type rawtype par le compilateur.

Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.List;
```

```
public class UtilisationGeneriques {
  public static void main(String[] args) {
    List liste = new ArrayList<String>();
  }
}
```

Résultat :

```
C:\java>javac -Xlint UtilisationGeneriques.java
UtilisationGeneriques.java:6: warning: [rawtypes] found raw type: List
    List liste = new ArrayList<String>();
    ^
  missing type arguments for generic class List<E>
  where E is a type-variable:
    E extends Object declared in interface List
1 warning
```

Lorsque l'on mélange des types bruts et des types génériques, le compilateur émet des avertissements de type rawtype.

Lors de l'affectation à une variable d'un type générique d'une instance d'un type brut, le compilateur émet un avertissement de type unchecked.

Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.List;

public class MaClasse {
  public static void main(String[] args) {
   List<String> liste = new ArrayList();
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java
Note: MaClasse.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Un warning de type "unchecked" signifie que le compilateur ne dispose pas de suffisamment d'informations sur les types pour effectuer toutes les vérifications de type nécessaires afin d'assurer la sécurité des types.

L'avertissement "unchecked" est désactivé par défaut, bien que le compilateur indique qu'un warning de ce type soit présent. Pour afficher tous les avertissements "unchecked", il faut utiliser l'option -Xlint:unchecked du compilateur.

Résultat :

```
C:\java>javac -Xlint:unchecked MaClasse.java
MaClasse.java:7: warning: [unchecked] unchecked conversion
   List<String> liste = new ArrayList();
                        ^
  required: List<String>
  found:    ArrayList
1 warning
```

Un avertissement de type "unchecked » est aussi émis par le compilateur lors de l'invocation d'une méthode avec un paramètre générique sur une instance d'un type brut.

Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.List;
```

```
public class MaClasse {

 public static void main(String[] args) {
   List liste = new ArrayList();
   liste.add("test");
 }
}
```

Résultat :

```
C:\java>javac -Xlint:unchecked MaClasse.java
MaClasse.java:8: warning: [unchecked] unchecked call to add(E) as a member of the raw type List
   liste.add("test");
            ^
  where E is a type-variable:
    E extends Object declared in interface List
1 warning
```

L'exemple ci-dessous génère plusieurs avertissements par le compilateur.

Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.List;

public class Test {

  public static void main(String[] args) {
    List nombres = new ArrayList<Integer>();
    nombres.add(1);
    List chaines = new ArrayList<String>();
    chaines.add("1");

    List<String> liste;
    liste = chaines;
    String chaine = liste.get(0);
    liste = nombres;
    chaine = liste.get(0);
  }
}
```

Résultat :

```
C:\java>javac -Xlint:unchecked MaClasse.java
MaClasse.java:8: warning: [unchecked] unchecked call to add(E) as a member of the raw type List
    nombres.add(1);
               ^
  where E is a type-variable:
    E extends Object declared in interface List
MaClasse.java:10: warning: [unchecked] unchecked call to add(E) as a member of the raw type List
    chaines.add("1");
               ^
  where E is a type-variable:
    E extends Object declared in interface List
MaClasse.java:12: warning: [unchecked] unchecked conversion
    liste = chaines;
            ^
  required: List<String>
  found:    List
MaClasse.java:15: warning: [unchecked] unchecked conversion
    liste = nombres;
            ^
  required: List<String>
  found:    List
4 warnings
```

Le compilateur émet quatre avertissements de type unchecked :

- les deux premiers car le compilateur ne peut pas vérifier le type des données ajoutées puisque c'est le type brut qui est utilisé
- le troisième car le compilateur ne peut pas vérifier le type des données lors de l'affectation d'une variable de type brut à une variable de type paramétré
- le quatrième concerne la même raison que le précédent mais il engendre une pollution du heap liée à l'affection d'une List<Integer> à une List<String>. Le compilateur ne peut pas le vérifier de façon formelle puisque la List qui contient des entiers est défini avec le type brut

A l'exécution, si les données contenues dans les variables de type brut et de type paramétré sont de même type, tout se passe correctement. Si ce n'est pas le cas, une exception de type ClassCastException est levée.

Résultat :

```
C:\java>java MaClasse
Exception in thread "main" java.lang.ClassCastException:
java.lang.Integer cannot be cast to java.lang.String
        at MaClasse.main(MaClasse.java:16)
```

C'est la raison pour laquelle le compilateur n'émet que des avertissements et pas des erreurs : à l'exécution selon le type de données cela peut se passer correctement ou non.

Il n'y a aucun avantage à utiliser le type brut. Mixer des types bruts et des paramétrés doit être évité, si possible. Cela ne peut être évité lorsque du code legacy non générique est combiné avec du code générique. Dans les autres cas, mixer des types bruts et des paramétrés n'est pas une bonne pratique.

5.4. La définition de types génériques

Un type (classe, interface ou record) est générique s'il déclare au moins un paramètre de type.

Il est possible de définir ses propres types génériques permettant de définir un ou plusieurs types paramétrables. Un type générique est une classe, une interface ou un record qui est paramétré pour un ou plusieurs types.

La classe ci-dessous encapsule simplement une propriété de type Object.

Exemple :

```
public class Conteneur {
  private Object valeur;

  public void set(Object valeur) {
    this.valeur = valeur;
  }

  public Object get() {
    return valeur;
  }
}
```

A partir de Java 5, il est possible de rendre une classe générique.

La déclaration d'un type générique ressemble à une déclaration d'un type non générique, sauf que le nom de la classe est suivi d'une section de paramètres de type.

Pour définir une classe utilisant les génériques, il suffit de déclarer leur utilisation dans la signature de la classe à l'aide des caractères < et >. Ce type de déclaration est appelé type paramétré (parameterized type). Dans ce cas, les paramètres fournis dans la déclaration du générique sont des variables de type. Si la déclaration possède plusieurs variables de type alors il faut les séparer par un caractère virgule.

La section des paramètres de type d'un type générique peut comporter un ou plusieurs paramètres de type séparés par des virgules. Ces types sont appelés types paramétrés car ils acceptent un ou plusieurs types en paramètres.

Exemple (code Java 5.0) :

```
public class Conteneur<T> {
  private T valeur;

  public void set(T valeur) {
    this.valeur = valeur;
  }

  public T get() {
    return valeur;
  }
}
```

5.4.1. La portée des paramètres de type

La portée d'un type paramétré dépend de l'endroit où il est défini :

la classe ou l'interface s'il s'agit d'un paramètre de type d'un type générique

la méthode ou le constructeur, s'il s'agit d'un paramètre de type d'une méthode ou d'un constructeur générique

Pour éviter toute confusion liée à l'utilisation d'un même nom, il est recommandé de nommer différemment les paramètres de type d'une méthode ou d'un constructeur et le paramètres de type du type dans lequel ils sont déclarés.

5.4.2. Les conventions de nommage sur les noms des types

Le nom d'un type générique doit être un identifiant valide en Java. Cependant par convention, on utilise une seule lettre majuscule.

Dans le JDK, les noms des paramètres de types couramment utilisés sont :

- T : un type comme premier paramètre
- E : un élément notamment dans les types de l'API Collection
- N : un nombre
- S, U V : un type comme paramètre supplémentaire
- K : la clé d'une Map
- V : la valeur d'une map

Le respect de cette convention permet de facilement identifier l'utilisation d'un type générique dans le code source.

5.4.3. L'utilisation de plusieurs paramètres de types

Il est possible de définir plusieurs paramètres de types en les séparant chacun par une virgule.

Exemple (code Java 5.0) :

```
import java.util.List;

public class MaClasseGenerique<T, U> {

  private List<T> listeDeT;
  private List<U> listeDeU;

  public static void main(String[] args) {
    MaClasseGenerique<Integer, String> donnees = new MaClasseGenerique<>();
  }
}
```

5.4.4. Les interfaces génériques

Comme les classes, les interfaces peuvent être aussi génériques.

Exemple (code Java 5.0) :

```
public MonInterface<T> {
  void traiter(T donnees);
}
```

Elles peuvent aussi avoir plusieurs paramètres de types.

Exemple (code Java 5.0) :

```
public MonInterface<T, U> {
  void traiter(T conteneur, U donnees);
}
```

5.4.5. Les tableaux de génériques

Il est possible de définir un tableau d'un type générique.

Exemple (code Java 5.0) :

```
import java.lang.reflect.Array;

public class MaClasse<T> {

  private T[] tableau;

  public MaClasse()  {
  }
}
```

Les tableaux sont un ensemble d'éléments d'un même type. L'ajout d'un élément non compatible dans un tableau lève une exception de type ArrayStoreException car un tableau maintient son type dans le bytecode.

Exemple :

```
public class MaClasse {

  public static void main(String... args) {
    Object[] tab = new String[5];
    tab[0] = "test";
    tab[1] = 123;
  }
}
```

Résultat :

```
C:\java>java MaClasse
Exception in thread "main" java.lang.ArrayStoreException: java.lang.Integer
        at MaClasse.main(MaClasse.java:6)

C:\java>
```

Cela va à l'encontre de l'effacement de type : ainsi il n'est pas possible de créer un tableau de génériques en utilisant l'opérateur new.

Une autre raison pour laquelle il n'est pas possible de créer un tableau de génériques avec l'opérateur new est que les tableaux sont co-variants, ce qui signifie qu'un tableau d'objets d'un type hérité est un super-type d'un tableau d'objets de type fille. Autrement dit, Object[] est un supertype de String[] et on peut accéder à un tableau de chaînes de caractères par le biais d'une variable de type Object[].

Exemple :

```
public class MaClasse {

  public static void main(String[] args) {
    Object[] tab = new String[1];
    tab[0] = "test";
  }
}
```

Les paramètres de type ne sont de base pas covariants.

5.5. La pollution du heap (Heap Pollution)

Dans la documentation Java, la dénomination pollution du tas (heap pollution) désigne quelque chose dans le tas qui n'est pas du type attendu. C'est par exemple le cas d'un objet de type A alors que l'on pense avoir un objet de type B, celui précisé par un type générique. Dans ce cas, il est possible qu'une exception de type ClassCastException soit levée à l'exécution.

Une pollution du heap peut survenir dans plusieurs situations :

- lorsque l'on mixe des types bruts avec des types génériques
- lorsque l'on fait un cast
- lorsque l'on compile les classes séparément

Dans les deux premiers cas, le compilateur émet un avertissement sur le risque potentiel de pollution du heap. La pollution du heap ne se produit pas nécessairement, même si le compilateur émet un avertissement de type unckecked.

L'implémentation en Java des génériques respose sur l'effacement de type : cela assure la compatibilité binaire avec du code qui a été créée avant l'ajout des génériques. C'est la raison pour laquelle c'est le choix historique de la solution d'implémentation des génériques en Java.

L'effacement de type implique qu'un type paramétré dans une classe générique est non réifiable. Un type non réifiable est un type qui n'est pas plus disponible au moment de l'exécution.

L'effacement de type permet l'interopérabilité entre du code générique et non générique mais peut induire la possibilité d'une pollution du tas. Cela survient lorsque le type attendu à la compilation n'est pas celui utilisé à l'exécution.

Une pollution du heap peut survenir lorsque des types bruts et des types paramétrés sont utilisés conjointement et une variable d'un type brut est assignée à une variable d'un type paramétré (une variable avec un type générique fait référence à un objet qui n'est pas de ce type générique).

Cette situation est détectée par le compilateur qui émet dans ce cas un avertissement de type unchecked.

Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.List;

public class MaClasse {

  public static void main(String[] args) {
    List entiers = new ArrayList<Integer>();
    List<String> chaines = entiers;       // warning de type unchecked
    entiers.add(10);                      // warning de type unchecked
    String s = chaines.get(0);            // ClassCastException
  }
}
```

Résultat :

```
C:\java>javac -Xlint:unchecked MaClasse.java
MaClasse.java:8: warning: [unchecked] unchecked conversion
    List<String> chaines = entiers;       // unchecked warning
                           ^
  required: List<String>
  found:    List
MaClasse.java:9: warning: [unchecked] unchecked call to add(E) as a member of the raw type List
    entiers.add(10);                      // unchecked warning
               ^
  where E is a type-variable:
    E extends Object declared in interface List
2 warnings
```

A la compilation, l'effacement de type (type erasure) est appliqué par le compilateur : par exemple List<String> est transformé en List.

La variable chaines est une List paramétrée avec le type générique String. Lorsque la variable entiers est affectée à la variable chaines, le compilateur émet un warning de type unchecked. Le compilateur indique que la JVM ne sera pas en mesure de savoir le type paramétré de la variable entiers. Dans ce cas, une pollution du heap survient.

Le compilateur émet un autre avertissement de type unchecked lors de l'invocation de la méthode add(). Le compilateur ne connait pas le type paramétré de la variable entiers. Une nouvelle fois, une pollution du heap survient.

Lors de l'invocation de la méthode get(), le compilateur n'émet pas de warning car pour lui le type paramétré de la variable chaines est String donc la valeur retournée par la méthode get() devrait être un objet de type String.

Malheureusement, une exception de type ClassCastException est levée à l'exécution car il n'est pas possible de convertir un objet de type Integer en un objet de type String.

Exemple (code Java 5.0) :

```
C:\java>java MaClasse
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be
 cast to java.lang.String
        at MaClasse.main(MaClasse.java:10)
```

La pollution du heap survient lorsqu'une instance d'un type générique est affecté à une variable avec un type générique différent voire pas de type générique. Dans ce dernier cas, on parle de type brut.

Dans l'exemple précédent, cela survient lors qu'une instance de type List est affectée à une variable de type List<String>. Pour préserver la compatibilité ascendante, le compilateur accepte cette affectation. A cause de l'effacement de type à la compilation, List<Integer> et List<String> sont remplacé par le type brut List.

Une pollution du heap survient aussi lorsque la méthode add() est invoquée : à cause du type erasure le compilateur accepte l'invocation car pour lui la méthode add() attend un Object.

Dans les deux cas, le compilateur émet un avertissement de type unchecked indiquant qu'une pollution du heap peut survenir.

Celle-ci se matérialise à l'exécution si les types ne correspondent pas à une exception de type ClassCastException, ce qui est le cas lorsque l'on assigne le premier élément qui est de type Integer à une variable de type String. Ce cast est généré par le compilateur, car la List<Integer> est affecté à une List<String> : un cast vers le type String est ajouté avant l'affectation de la valeur dans le bytecode.

Une pollution du heap peut survenir lorsque du code non générique utilise des classes génériques, ou lorsque nous utilisons des casts non vérifiés ou des types bruts pour contenir une référence à une variable du mauvais type générique. Lorsque nous avons utilisé des casts non vérifiés ou des types bruts, le compilateur nous avertit qu'une pollution du heap peut survenir. Par exemple :

Exemple (code Java 5.0) :

```
public class Conteneur<T> {

  private T valeur;

  public Conteneur(T valeur) {
      this.valeur = valeur;
  }

  public T get() {
      return valeur;
  }

  public static void main(String... args) {
    Conteneur<String> cs = new Conteneur<>("test");   // ok
    Conteneur<?> cq = cs;                       // ok
    Conteneur<Integer> ci = (Conteneur<Integer>) cq; // warning du compilateur
    Integer i = ci.get();                // ClassCastException
  }
}
```

Résultat :

```
C:\java>javac -Xlint:unchecked Conteneur.java
Conteneur.java:16: warning: [unchecked] unchecked cast
    Conteneur<Integer> ci = (Conteneur<Integer>) cq; // warning du compilateur
                                                 ^
  required: Conteneur<Integer>
  found:    Conteneur<CAP#1>
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
1 warning

C:\java>java Conteneur
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be
 cast to java.lang.Integer
        at Conteneur.main(Conteneur.java:17)
```

5.6. La mise en oeuvre des génériques

Les classes génériques sont des classes qui sont typées au moment de la définition d'une variable ou de la création d'une instance.

Exemple : une classe qui possède un paramètre de type générique

Exemple (code Java 5.0) :

```
public MaClasse<T> {
}
```

Ainsi cette déclaration se lit MaClasse de T.

Il est alors possible d'utiliser T comme un type dans le corps de la classe.

Exemple (code Java 5.0) :

```
public MaClasse<T> {
  T monChamp;
}
```

Pour créer une instance de type MaClasse, il faut préciser le type réel qui sera utilisé : la classe est alors un type paramétré avec un argument de type.

Exemple (code Java 5.0) :

```
  MaClasse<String> maClasse = new MaClasse<String>();
```

Les types génériques sont instanciés pour former des types paramétrés en fournissant des arguments de type réels qui remplacent les paramètres de type formels. Une classe comme ArrayList<E> est un type générique, qui possède un paramètre de type E. Les types d'une instance, telles que ArrayList<Integer> ou ArrayList<String>, sont appelées types paramétrés, et String et Integer sont les arguments de type réels qui sont utilisés.

Un type paramétré peut prendre deux formes :

- utilisation d'un type concret, exemple : List<String>, ...
- utilisation d'un wilcard, exemple : List<?>, List<? extends Number>, ...

Les types paramétrés avec un wilcard sont essentiellement utilisés pour définir le type d'une variable ou d'un paramètre.

A la compilation d'un type paramétré avec un wilcard, le compilateur effectue une capture pour conversion (capture conversion) pour remplacer le wilcard par un type concret qui est inconnu.

Par exemple : List<? extends Number> est le supertype de toute List<X> concrète où X est un sous-type de Number.

Une classe ou une interface générique peut être considérée comme un modèle de code : il faut remplacer ses paramètres de type par des types réels pour obtenir une classe ou une interface concrète.

```
interface List<T> {
  int   size();
  T     get(int);
  void  add(T);
  ...
}
```

```
interface List<Integer> {
  int     size();
  Integer get(int);
  void    add(Integer);
  ...
}
```

Ces types concrets s'excluent mutuellement, ce qui signifie que List<A> et List<B> (avec A différent de B) sont des types différents qui ne partagent aucune relation même si une relation existe entre les types A et B. Par exemple, un objet ne peut pas être à la fois une instance de List<Number> et de List<Integer>. Chaque objet est une instance d'un type de classe concret (par exemple ArrayList<Number>), qui a des super-classes concrètes et des super-interfaces concrètes (par exemple List<Number>).

5.6.1. Les génériques et l'héritage

En application de l'héritage et du polymorphisme, il est possible d'assigner une instance d'une classe fille à une variable du type d'une de ses classes mères.

Exemple :

```
public class MaClasse {

  public static void main(String[] args) {
    Object objet = new Object();
    Integer entier = Integer.valueOf(10);
    objet = entier;
    traiter(entier);
    traiter(Double.valueOf(2.0));
  }

  public static void traiter(Number valeur) {
  }
}
```

De même, pour un type paramétré d'une collection, il est possible d'ajouter des instances du type de l'argument de type ou d'une de ses sous-classes.

Exemple (code Java 5.0) :

```
import java.util.List;
import java.util.ArrayList;

public class MaClasse {

  public static void main(String[] args) {
    List<Number> liste1 = new ArrayList<Number>();
    liste1.add(Integer.valueOf(1));
    liste1.add(Double.valueOf(2.0));

    Number valeur = liste1.get(0);
    System.out.println(valeur);
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java

C:\java>java MaClasse
1
```

Par contre, il n'y a pas de relation entre deux types paramétrés même si les arguments de types utilisés possèdent une relation. Ceci est dû au fait que les types paramétrés sont invariants.

Il existe une relation d'héritage entre les types Number et Integer : Number est la classe mère de la classe Integer.

Bien que ce lien existe entre ces deux types, il n'y a pas de relation entre List<Number> et List<Integer>.

Exemple (code Java 5.0) :

```
import java.util.List;
import java.util.ArrayList;

public class MaClasse {

  public static void main(String[] args) {
    List<Number> nombres = new ArrayList<Number>();
    List<Integer> entiers = new ArrayList<Integer>();
    nombres = entiers;
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:9: error: incompatible types
    nombres = entiers;
              ^
  required: List<Number>
  found:    List<Integer>
1 error
```

5.6.2. Les classes génériques et le sous-typage

Il est possible de sous-typer une classe ou une interface générique en l'étendant ou en l'implémentant.

Une classe qui hérite d'une classe générique peut spécifier les arguments de type, peut conserver les paramètres de type ou ajouter des paramètres de type supplémentaire.

La relation entre les paramètres de type d'une classe ou d'une interface et les paramètres de type d'une autre est déterminée par les clauses extends et implements.

Par exemple, ArrayList<E> implémente List<E>, et List<E> étend Collection<E>. Ainsi, ArrayList<String> est un sous-type de List<String>, qui est un sous-type de Collection<String>. Tant que le type générique n'est pas modifié, la relation de sous-typage est préservée entre les types.

Exemple (code Java 5.0) :

```
public interface MaList<E> extends List<E> {
}
```

Il est possible de rajouter d'autres paramètres de type tant que ceux hérités sont maintenus.

Par exemple, Il est possible de définir une interface générique qui implémente List<E> et de lui ajouter un paramètre de type supplémentaire tant que le paramètre de type de l'interface mère est conservé.

Exemple (code Java 5.0) :

```
public interface MaList<E,T> extends List<E> {
}
```

Il n'est cependant pas possible de retirer un paramètre de type hérité.

Exemple (code Java 5.0) :

```
public interface MaList extends List<E> {
}
```

Résultat :

```
C:\java>javac MaList.java
MaList.java:3: error: cannot find symbol
public interface MaList extends List<E> {
                                     ^
  symbol: class E
1 error
```

Il est possible d'hériter d'un type paramétré.

Exemple (code Java 5.0) :

```
import java.util.Map.Entry;

public class IdNomEntry implements Entry<Integer, String> {

    private Integer id;
    private String nom;

    public IdNomEntry(Integer id, String nom) {
      this.id = id;
      this.nom = nom;
    }

    @Override
    public Integer getKey() {
      return id;
    }

    @Override
    public String getValue() {
      return nom;
    }

    @Override
    public String setValue(String value) {
      this.nom = value;
      return nom;
    }
}
```

Il est possible d'hériter d'un type générique mixant paramètre de type et argument de type.

Exemple (code Java 5.0) :

```
package com.jmdoudoux.dej;

import java.util.Map.Entry;

public class IdValeurEntry<V> implements Entry<Integer, V> {

    private Integer id;
    private V valeur;

    public IdValeurEntry(Integer id, V valeur) {
      this.id = id;
      this.valeur = valeur;
    }

    @Override
    public Integer getKey() {
      return id;
    }

    @Override
    public V getValue() {
      return valeur;
    }

    @Override
    public V setValue(V value) {
      this.valeur = value;
      return valeur;
    }
}
```

5.6.3. Le transtypage des instances génériques

L'héritage permet d'affecter à une variable de type A toute instance d'une classe qui hérite de A.

Cependant ce comportement ne fonctionne pas avec les génériques.

Ainsi il n'est pas possible d'affecter une variable de type List<String> à une variable de type List<Object> car elles n'ont pas de relation.

Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.List;

public class MaClasse<T> {

  public static void main(String... args) {

    List<String> chaines = new ArrayList<String>();
    List<Object> objets = chaines;
    objets.add(new Object());
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:9: error: incompatible types: List<String> cannot be converted to List<Object>
    List<Object> objets = chaines;
```

```
                                         ^
1 error
```

Il n'est pas possible non plus de caster l'affectation d'une variable d'un type générique même si c'est vers une variable d'un même type et d'un type générique qui soit un sous-type.

Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.List;

public class MaClasse<T> {

  public static void main(String... args) {

    List<String> chaines = new ArrayList<String>();
    List<Object> objets = (List<Object>) chaines;
    objets.add(new Object());
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:9: error: incompatible types: List<String> cannot be converted to List<Object>
    List<Object> objets = (List<Object>) chaines;
                                         ^
1 error
```

Cela permet d'éviter des exceptions à l'exécution dû l'effacement de type utilisé dans la mise en oeuvre des génériques. Celui-ci impliquerait un cast en String de l'instance d'Object ajoutée dans la List.

5.7. Les méthodes et les constructeurs génériques

Parfois, la classe ne doit pas être générique mais seulement une méthode ou une méthode a besoin d'un type générique uniquement pour son usage propre.

Parfois, nous ne voulons pas que la classe entière soit paramétrée avec un générique mais uniquement une ou plusieurs méthodes. Dans ce cas, il est possible de définir une méthode générique.

Il est ainsi possible de définir des méthodes ou des constructeurs génériques.

Les méthodes génériques sont des méthodes qui sont écrites avec une seule déclaration et qui peuvent être invoquées avec des arguments de différents types. Le compilateur garantit l'exactitude du type utilisé.

Les méthodes génériques définissent un ou plusieurs types paramétrés qui leur sont propres.

Les méthodes génériques diffèrent des classes génériques dans la portée des types paramétrés définis : pour les méthodes génériques c'est uniquement dans la méthode alors que pour les classes génériques, c'est dans toute la classe.

Il est possible d'utiliser un type générique sur une méthode, que celle-ci soit dans une classe générique ou non.

La syntaxe de définition d'une méthode générique doit suivre plusieurs règles :

- toutes les déclarations de méthodes génériques comportent une section de définition de type paramétré délimitée par une paire < et > qui précède le type de retour de la méthode
- chaque section de type paramétré contient un ou plusieurs types paramétrés séparés par des virgules. Un type paramétré est un identifiant qui spécifie un nom de type générique. Un type paramétré ne peut pas être un type primitif
- les types paramétrés peuvent être utilisés pour déclarer le type de retour et le type des paramètres

Exemple (code Java 5.0) :

```
public class MaClasse {

  public static < E > void afficher( E[] donnees ) {
    for(E element : donnees) {
      System.out.print(element);
    }
    System.out.println();
  }

  public static void main(String args[]) {
      Integer[] entiers = { 1, 2, 3, 4, 5 };
      String[] chaines = { "a", "b", "c", "d", "e" };

      afficher(entiers);
      afficher(chaines);
   }
}
```

Résultat :

```
C:\java>java MaClasse
12345
abcde
```

Pour invoquer une méthode générique, ses paramètres de type doivent être remplacés par des types réels, soit par inférence soit explicitement.

En fonction des types d'arguments transmis à la méthode générique, le compilateur traite chaque appel de méthode de manière appropriée. Lors de l'invocation de la méthode, deux cas de figures peuvent se présenter :

- le cas le plus courant, le compilateur est capable d'inférer le type générique et dans ce cas il suffit simplement d'invoquer la méthode comme une méthode sans générique
- si compilateur ne peut pas faire l'inférence, il faut préciser le type générique entre un < et un > entre le point et le nom de la méthode générique à invoquer

Exemple (code Java 5.0) :

```
public class MaClasse {

  public static <T extends Comparable<T>> T max(T x, T y) {
      T max = x;
      if(y.compareTo(max) > 0) {
         max = y;
      }
      return max;
   }

  public static void main(String... args) {
    System.out.println(MaClasse.max(123, 26));
    System.out.println(MaClasse.max("abc", "xyz"));
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java

C:\java>java MaClasse
123
xyz
```

Il est aussi possible et parfois nécessaire d'indiquer explicitement le type générique notamment si le compilateur ne peut pas l'inférer.

Exemple (code Java 5.0) :

```
public class MaClasse {

  public static <T extends Comparable<T>> T max(T x, T y) {
    T max = x;
    if(y.compareTo(max) > 0) {
      max = y;
    }
    return max;
  }

  public static void main(String... args) {
    System.out.println(MaClasse.max(123, 26));
    System.out.println(MaClasse.max("abc", "xyz"));
    System.out.println(MaClasse.<Integer>max(123, 26));
    System.out.println(MaClasse.<String>max("abc", "xyz"));
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java

C:\java>java MaClasse
123
xyz
123
xyz
```

Comme les constructeurs sont une forme spéciale de méthode, il est possible de définir un constructeur générique.

Exemple (code Java 5.0) :

```
public class MaClasse {

  public <T> void MaClasse (T donnees) {
    // ...
  }
}
```

5.8. Les paramètres de type bornés (bounded type parameters)

Il est parfois nécessaire de restreindre les types qui peuvent être utilisés comme arguments de type dans un type paramétré. Par exemple, une méthode qui opère sur des nombres ne peut accepter que des instances de type Number ou de ses sous-classes ou dans une méthode qui compare deux objets et que l'on souhaite s'assurer que les objets fournis implémentent l'interface Comparable.

Pour cela, il faut utiliser des paramètres de type bornés (bounded type parameters) qui permettent de définir des restrictions sur le type qui sera autorisé à être utilisé comme type paramétré.

Les paramètres de type bornés peuvent être utilisés dans la définition des types et des méthodes génériques. Ils permettent de restreindre les types utilisables avec une classe ou une méthode générique tout en ayant la flexibilité de travailler avec les différents types définis dans le générique.

Pour déclarer un paramètre de type borné, il faut indiquer le nom du paramètre de type, suivi du mot-clé extends, suivi du type qui représente sa borne supérieure.

Dans ce contexte, le mot clé extends indique que le type étend la borne supérieure dans le cas d'une classe ou implémente une borne supérieure dans le cas d'une interface. Exemple :

```
<T extends Number>
```

T pourra être le type Number ou une de ses classes filles lors de la déclaration d'une type paramétré.

Il est possible de préciser une relation entre une variable de type et une classe ou une interface : ainsi, avec le mot-clé extends dans la variable de type, il sera possible d'utiliser une instance du type paramétré avec n'importe quel objet qui hérite ou implémente la classe ou l'interface précisée.

Exemple (code Java 5.0) :

```
import java.util.*;

public class MaClasseGenerique<T extends Collection> {
  private T param;

  public MaClasseGenerique2(T param) {
    this.param = param;
  }

  public T getParam() {
    return this.param;
  }
}
```

L'utilisation du type paramétré MaClasseGenerique peut être réalisée avec n'importe quelle classe qui hérite de l'interface java.util.Collection.

Exemple (code Java 5.0) :

```
import java.util.*;

public class TestClasseGenerique {

  public static void main(String[] args) {
    MaClasseGenerique<ArrayList> maClasseA =
          new MaClasseGenerique2<ArrayList>(new ArrayList());
    MaClasseGenerique<TreeSet> maClasseB =
          new MaClasseGenerique2<TreeSet>(new TreeSet());
  }
}
```

Ce mécanisme permet une utilisation un peu moins stricte du typage dans les génériques.

L'utilisation d'une classe qui n'hérite pas de la classe ou n'implémente pas l'interface définie dans la variable de type, provoque une erreur à la compilation.

Exemple (code Java 5.0) :

```
import java.util.*;

public class TestClasseGenerique {

  public static void main(String[] args) {
    MaClasseGenerique<ArrayList> maClasseA =
          new MaClasseGenerique2<ArrayList>(new ArrayList());
    MaClasseGenerique<TreeSet> maClasseB =
          new MaClasseGenerique2<TreeSet>(new TreeSet());
    MaClasseGenerique<String> maClasseC =
          new MaClasseGenerique2<String>("test");
  }
}
```

Résultat :

```
C:\java>javac TestClasseGenerique.java
TestClasseGenerique.java:10: error: type argument String is not within bounds of type-variable T
    MaClasseGenerique2<String> maClasseC =
                     ^
  where T is a type-variable:
    T extends Collection declared in class MaClasseGenerique
TestClasseGenerique.java:11: error: type argument String is not within bounds of type-variable T
```

```
          new MaClasseGenerique<String>("test");
                              ^
  where T is a type-variable:
    T extends Collection declared in class MaClasseGenerique
2 errors
```

Un autre exemple : restreindre le type d'objets qui peut être utilisé dans le type paramétré dans une méthode qui compare deux objets pour assurer que les objets acceptés implémentent l'interface Comparable.

Exemple (code Java 5.0) :

```
public class MaClasse<T> {

  public static <T extends Comparable<T>> int comparer(T t1, T t2){
    return t1.compareTo(t2);
  }
}
```

Si la méthode comparer() est invoquée avec un type générique qui n'implémente pas l'interface Comparable, alors le compilateur émet une erreur.

Au-delà d'une meilleure vérification du type générique utilisé, l'utilisation d'une borne supérieure permet dans le code d'invoquer les méthodes définies dans le type précisé.

5.9. Les paramètres de type avec wildcard

En raison de l'implémentation des types génériques reposant sur l'effacement des types, la JVM n'a aucun moyen de connaître à l'exécution les informations de type des paramètres de type. Par conséquent, il ne peut pas se protéger contre la pollution du tas au moment de l'exécution.

C'est pour cela que les types génériques sont invariants. Les paramètres de type doivent correspondre exactement, afin de se protéger contre la pollution du tas.

Exemple (code Java 5.0) :

```
import java.util.ArrayList;

public class UtilisationGeneriques {
  public static void main(String[] args) {
    ArrayList<Integer> listInteger = new ArrayList<>();
    ArrayList<Integer> autreListNumber = listInteger;
  }
}
```

Toute tentative d'utiliser un autre type même d'un sous-type provoque une erreur à la compilation.

Exemple (code Java 5.0) :

```
import java.util.ArrayList;

public class UtilisationGeneriques {
  public static void main(String[] args) {
    ArrayList<Integer> listInteger = new ArrayList<>();
    ArrayList<Number> listNumber = listInteger; // Erreur de compilation
  }
}
```

Résultat :

```
C:\java>javac UtilisationGeneriques.java
```

```
UtilisationGeneriques.java:6: error: incompatible types: ArrayList<Integer> cannot be converted
 to ArrayList<Number>
    ArrayList<Number> listNumber = listInteger; // Erreur de compilation
                                   ^
1 error
```

Pour permettre de contourner ces limitations et gagner en flexibilité en proposant un support de la covariance, contravariance et la bi-variance, les génériques propose les wildcards.

Un wildcard désigne un type quelconque : un wildcard n'est donc pas un type.

Un paramètre de type avec un wildcard (wilcard parameterized type) permet de représenter toute une famille de type en utilisant au moins un wildcard dans sa définition.

Les wildcards sont représentés, dans la déclaration d'un paramètre de type, par un point d'interrogation « ? » et sont utilisés pour faire référence à un type inconnu. Les wildcards sont particulièrement utiles lors de l'utilisation de génériques et peuvent être utilisés comme paramètre de type

Un wildcard peut être utilisé dans la définition d'un type de paramètre dans diverses situations : pour la définition d'un type pour un paramètre, un champ ou une variable locale ; parfois comme type de retour (même cela n'est pas recommandé car il est préférable d'être plus précis).

Un wildcard ne peut pas être utilisé dans des substitutions : cela n'a pas sens de créer une instance de type List<?> ni d'invoquer une méthode dont le type inféré serait un wildcard.

Les types paramétrés avec un wildcard imposent des restrictions par rapport aux types paramétrés avec un type concret :

- il n'est pas possible de créer une instance, par exemple : new ArrayList<?>
- il n'est pas possible d'hériter d'un type paramétré avec un wildcard, par exemple : interface MaList extends List<?>

Un paramètre de type avec un wildcard n'est pas un type concret qui pourrait apparaître avec l'utilisation d'un opérateur new. Il permet de préciser quels types sont valides dans des scénarios particuliers d'utilisation de génériques.

L'utilisation d'un wildcard dans un paramètre de type peut être non borné (unbounded wildcard) ou borné (bounded wildcard). Un type paramétré avec un wildcard peut avoir une borne inférieure ou une borne supérieure mais pas les deux.

5.9.1. Le besoin des wildcards avec les génériques

Les génériques sont invariants par nature. Même si la classe Object est le supertype de toutes les classes Java, une collection d'Object n'est pas le supertype d'une collection quelconque.

Ainsi, une List<Object> n'est pas le supertype de List<String> et l'affectation d'une variable de type List<Object> à une variable de type List<String> provoque une erreur de compilation. Ceci permet de prévenir d'éventuels conflits qui peuvent survenir si on ajoute des types hétérogènes à la même collection.

Comme les génériques sont invariants, il n'est possible d'affecter une référence d'un objet générique à une variable de la même classe mais avec un type générique différent même si ce type est un sous-type du générique. Ce comportement est dû à l'utilisation de l'effacement de type à la compilation.

Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.List;

public class UtilisationGeneriques {
  public static void main(String[] args) {
    List<Object> liste = new ArrayList<String>();
  }
}
```

Résultat :

```
C:\java>javac UtilisationGeneriques.java
UtilisationGeneriques.java:6: error: incompatible types: ArrayList<String> cannot be converted
 to List<Object>
    List<Object> liste = new ArrayList<String>();
                         ^
1 error
```

Fréquemment cette contrainte est limitative, car il est parfois utile d'utiliser une instance d'un sous-type ou d'un super-type d'une classe. Dans ces cas, il faut utiliser les concepts de covariance et de contravariance. Exemple :

Exemple (code Java 5.0) :

```
  public static void dessinerToutes(List<Forme> formes) {
    for(Forme f : forms) {
      f.dessiner();
    }
  }
```

Il n'est pas possible de passer en paramètre de cette méthode, une List<Carre> ou List<Triangle> même si Carre et Triangle sont des classes filles de Forme.

Pour permettre cela, il faut utiliser un wildcard avec une borne supérieure.

Exemple (code Java 5.0) :

```
  public static void dessinerToutes(List<? extends Forme> formes) {
    for(Forme f : forms) {
      f.dessiner();
    }
  }
```

Cela permet de passer en paramètre une List<Forme> mais aussi une List dont l'argument de type est un sous-type de Forme.

5.9.2. L'utilisation de wildcards dans les types paramétrés

Un wildcard est représenté par un point d'interrogation et permet d'indiquer un type arbitraire dans un type paramétré. Un type paramétré avec un wildcard (wildcard parameterized type) utilise au moins un wildcard dans la définition d'un type paramétré.

Par exemple :

- Set<?> : pour un ensemble qui peut contenir tout type d'objet. Cependant, à l'exécution, cet ensebmle devra être assignée à un type concret
- List<? extends Number> : pour une liste qui peut contenir des instances de type Number ou sous-type de Number
- Comparator<? super Integer> : pour une comparateur d'objets de type Integer ou d'un de ses super-types

Parfois au lieu de préciser un type, il faudrait préciser un type mais aussi ses sous-types. Cela se fait en utilisant le wildcard « ? » suivi du mot clé extends suivi du type. On dit alors que que le type générique est le wildcard avec comme borne inférieure le type précisé.

Par exemple, dans l'API Collections, la méthode addAll() de l'interface Collection permet d'ajouter tous les éléments de la Collection passée en paramètre et contenant des éléments du type de la collection ou d'un de ses sous-types. La signature de la méthode est de la forme :

```
boolean addAll(Collection<? extends E> c)
```

Le type paramétré <? extends E> indique qu'il ne sera possible que d'utiliser des instances de type E ou d'un sous-type de E.

Exemple (code Java 5.0) :

```
import java.util.List;
import java.util.ArrayList;

public class MaClasse {

  public static void main(String... args) {
    List<Number> nombres = new ArrayList<Number>();
    ArrayList<Integer> entiers = new ArrayList<Integer>();
    ArrayList<Long> entierlongs = new ArrayList<Long>();
    ArrayList<Float> flottants = new ArrayList<Float>();
    nombres.addAll(entiers);
    nombres.addAll(entierlongs);
    nombres.addAll(flottants);
  }
}
```

Parfois au lieu de préciser un type, il faudrait préciser un type mais aussi un de ses super-types. Cela se fait en utilisant le wildcard « ? » suivi du mot clé super suivi du type. On dit alors que le type générique est le wildcard avec comme borne supérieure le type précisé.

La mise en oeuvre des bornes inférieures et supérieures impose des contraintes pour garantir la sécurité de type.

5.9.2.1. Les types paramétrés avec wildcard non bornés (Unbounded wildcard parameterized type)

Les types génériques sont par nature invariants : parfois il est nécessaire d'utiliser des types qui supportent la bi-variance.

Un wilcard indique un type inconnu. Lors utilisation dans un paramètre d'une méthode permet d'exécuter des traitements sans connaître le type passé en paramètre.

Par exemple, pour définir une méthode générique qui affiche une liste d'objets quelconque, il pourrait être tentant d'écrire l'implémentation ci-dessous :

Exemple (code Java 5.0) :

```
import java.util.List;
import java.util.ArrayList;
import java.util.Arrays;

public class MaClasse {

  public static void afficher(List<Object> liste) {
    for (Object element : liste) {
      System.out.print(element + " ");
    }
    System.out.println();
  }

  public static void main(String... args) {
    List<Integer> entiers = Arrays.asList(1, 2, 3);
    List<String> chaines = Arrays.asList("A", "B", "C");
    afficher(entiers);
    afficher(chaines);
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:17: error: incompatible types: List<Integer> cannot be converted to List<Object>
    afficher(entiers);
```

```
             ^
MaClasse.java:18: error: incompatible types: List<String> cannot be converted to List<Object>
    afficher(chaines);
             ^
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full output
2 errors
```

Malheureusement un type paramétré est invariant. Cette méthode ne fonctionne que pour une List<Object> qui ne contient que des instances d'Object : elle ne fonctionne pour aucune autre List telle que List<Integer>, List<String>, ...

Un type paramétré par un Wildcard non borné représente toute version du type générique. Les wildcards non bornés sont utiles lorsque le type utilisé n'a pas d'importance.

Il existe deux scénarios dans lesquels un wildcard non borné est une approche utile :

- si seules les méthodes de la classe Object sont utiles
- lorsque le code utilise des méthodes de la classe générique qui ne dépendent pas du type paramétré. Par exemple, Class<?> est souvent utilisée parce que la plupart des méthodes de Class<T> ne dépendent pas de T.

Un type non borné est spécifié à l'aide d'un wilcard représenté par le caractère point d'interrogation « ? ». Par exemple, List<?> qui représente une List de type inconnu. Il est possible de considérer que List<?> exprime la bi-variance puisqu'il permet d'utiliser n'importe quel type.

Il faut affecter une instance d'une classe générique à une variable à un type générique composée d'un wildcard.

Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.List;

public class UtilisationGeneriques {
  public static void main(String[] args) {
    List<?> liste = new ArrayList<String>();
  }
}
```

La déclaration d'une variable dont le type générique est un wildcard non borné permet de lui affecter une instance dont le type générique peut être quelconque.

Exemple (code Java 5.0) :

```
import java.util.List;
import java.util.ArrayList;

public class MaClasse {

  public static void main(String[] args) {
    List<?> liste = new ArrayList<Integer>();
    liste = new ArrayList<String>();
  }
}
```

Ainsi pour définir une méthode qui puisse utiliser une liste de n'importe quel type, il faut utiliser un wildcard dans le type paramétré. Dans l'exemple suivant, la liste paramétrée par un wildcard permet d'accepter toute sorte de liste :

Exemple (code Java 5.0) :

```
import java.util.List;
import java.util.ArrayList;
import java.util.Arrays;

public class MaClasse {
  public static void afficher(List<?> liste) {
    for (Object element : liste) {
      System.out.print(element + " ");
    }
    System.out.println();
  }

  public static void main(String... args) {
    List<Integer> entiers = Arrays.asList(1, 2, 3);
    List<String> chaines = Arrays.asList("A", "B", "C");
    afficher(entiers);
    afficher(chaines);
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java

C:\java>java MaClasse
1 2 3
A B C
```

La déclaration d'une variable de type List<Object> est différente de List<?>.

Une collection de type Collection<Object> est paramétrée avec la classe mère directe ou indirecte de toutes les autres classes. Une telle collection peut donc contenir n'importe quel type d'objet : c'est donc un ensemble hétérogène d'objet.

Exemple (code Java 5.0) :

```
import java.util.Date;
import java.util.List;
import java.util.ArrayList;

public class MaClasse {

  public static void main(String... args) {
    List<Object> liste = new ArrayList<Object>();
    liste.add("test");
    liste.add(123);
    liste.add(new Date());
  }
}
```

Une collection de type Collection<?> devra au runtime posséder un type ou une limite de type qui précisera le type à utiliser. A l'exécution, une telle collection contiendra donc un ensemble homogène d'objets.

Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class MaClasse {

  public static void main(String... args) {
    List<Integer> entiers = Arrays.asList(1, 2, 3);
    List<String> chaines = Arrays.asList("A", "B", "C");

    List<?> listeEntiers = entiers;
    List<?> listeChaines = chaines;

    System.out.println(listeEntiers);
    System.out.println(listeChaines);
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java

C:\java>java MaClasse
[1, 2, 3]
[A, B, C]
```

Le type List<Object> est donc différent du type List<?>

List<Object> liste	List<?> liste
Il est possible d'ajouter n'importe quel objet (une instance de type Object ou n'importe quel de ses sous-types) dans la liste avec la méthode add()	Il n'est possible que d'ajouter null à la liste avec la méthode add()
Invariant : il n'est possible que d'affecter une autre de type List<Object> à la variable liste	Bivariant : il est possible d'affecter n'importe quelle référence à une List peu importe le type générique

List<?> est équivalent à List<? extends Object)

Les types paramétrés avec wildcard non bornés sont utiles, mais présentent des limitations.

Dans une collection avec un tel type paramétré, il est possible d'obtenir des valeurs mais il n'est pas possible d'en ajouter d'autres, excepté la valeur null car il n'y a pas d'information sur l'objet.

Il n'est donc pas possible dans d'ajouter un nouvel élément dans la List même si cet élément est du type du type générique précisé lors de la création de l'instance.

Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.List;

public class UtilisationGeneriques {
  public static void main(String[] args) {
    List<?> liste = new ArrayList<String>();
    liste.add("test");
  }
}
```

Résultat :

```
C:\java>javac UtilisationGeneriques.java
UtilisationGeneriques.java:7: error: incompatible types: String cannot be converted to CAP#1
    liste.add("test");
              ^
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full output
1 error

C:\java>javac -Xdiags:verbose UtilisationGeneriques.java
UtilisationGeneriques.java:7: error: no suitable method found for add(String)
    liste.add("test");
         ^
    method List.add(CAP#1) is not applicable
      (argument mismatch; String cannot be converted to CAP#1)
    method List.add(int,CAP#1) is not applicable
      (actual and formal argument lists differ in length)
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
1 error
```

Il peut paraître peu utile d'avoir une collection dans laquelle il n'est pas possible d'ajouter de valeurs mais c'est la contrainte pour pouvoir accepter n'importe quelle liste tout en garantissant la sécurité de type.

Certaines méthodes de l'API Collection attendent en paramètre une collection avec un type paramétré avec un wildcard Exemple : les méthodes containsAll() et removeAll() de l'interface java.util.List<E>

```
public boolean containsAll(Collection<?> c)

public boolean removeAll(Collection<?> c);
```

Ces deux méthodes ont uniquement besoin de lire les données passées en paramètres dans leur traitement. Ces traitements n'utilisent que des méthodes de la classe Object telles que equals() et hashCode().

Il n'est pas possible d'utiliser un wildcard dans un type générique avec l'opérateur new.

Exemple (code Java 5.0) :

```
import java.util.Set;
import java.util.HashSet;

public class MaClasse {

  public static void main(String... args) {

    Set<?> valeurs = new HashSet<?>();
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:7: error: unexpected type
    Set<?> valeurs = new Set<?>();
                            ^
  required: class or interface without bounds
  found:    ?
1 error
```

Il faut utiliser un type concret avec l'opérateur new.

Exemple (code Java 5.0) :

```
import java.util.Set;
import java.util.HashSet;

public class MaClasse {

  public static void main(String... args) {
    Set<?> valeurs = new HashSet<String>();
  }
}
```

5.9.2.2. Les types paramétrés avec wildcard borné (Bounded wildcard parameterized type)

Pour ajouter une restriction sur les types paramétrés, il est possible d'appliquer au wildcard une borne supérieure ou inférieure.

Cette restriction est appliquée à l'aide des mots-clés « extends »ou« super » qui permettent de définir des bornes inférieures et des bornes supérieures.

Les wildcards bornés imposent certaines restrictions sur les types possibles qu'il est possible d'utiliser pour créer une instance d'un type paramétré.

Il existe deux types de génériques avec wildcard bornés :

- les génériques avec bornes inférieures (lower bounded generics) : permet de préciser une classe dont l'argument de type doit être une super-classe
- les génériques avec bornes supérieures (upper bounded generics) : permet de préciser un super-type que doit hériter l'argument de type

La borne supérieure exprime la covariance, la borne inférieure exprime la contravariance.

Il est possible de définir une borne supérieure ou une borne inférieure mais pas les deux.

L'utilisation d'un wildcard avec une borne supérieure ou inférieure permet d'accroître la flexibilité d'une API.

5.9.2.3. Les wildcards avec bornes supérieures (Upper Bounded Wildcards)

Les wildcards avec borne supérieure sont utilisés pour assouplir la restriction sur le type d'un paramètre de type : ils permettent de leur assigner toute instance dont l'argument de type est le type de la borne ou un ses sous-types. Ce cas exprime la version covariante du type générique List<E>.

Par exemple, si une méthode doit pouvoir accepter en paramètre une List<Integer>, ou List<Double> ou List<Number>, il est possible d'utiliser un wildcard avec une borne supérieure dont le type de la borne sera Number.

Pour utiliser un wildcard avec une borne supérieure, il faut utiliser le caractère point d'interrogation suivi du mot clé extends et du type (classe ou interface) qui représente la borne.

Le mot clé extends peut s'utiliser aussi avec une interface plutôt qu'une classe.

Par exemple :

```
List<? extends Comparable>
```

Dans ce cas d'usage, extends signifie soit "étend" si la borne est une classe soit "implémente" si la borne est une interface.

Le type List<Number> est différent du type List<? Extends Number>

List<Number> liste	List<? extends Number> liste

Il est possible d'ajouter une instance de type Number ou n'importe quel de ses sous-types dans la liste avec la méthode add()	Il n'est possible que d'ajouter null à la liste avec la méthode add()
	List<? extends Number> liste = ArrayList<Integer>();
	liste.add(null);

Invariant : il n'est possible que d'affecter une autre instance de type List<Number> à la variable liste	Covariant : il est possible d'affecter n'importe quelle référence à une List dont le type générique est Number ou un sous-type de Number
	List<Integer> entiers = new ArrayList<Integer>();
	List<? extends Number> liste = entiers;

Il est possible d'utiliser toutes les méthodes de la classe utilisée comme borne supérieure, puisque tous les éléments de la collection héritent de cette classe.

L'exemple ci-dessous définit et utilise une méthode sommer() qui accepter une List d'objets de type de Number ou une liste d'un sous-type de Number. Elle permet donc de faire la somme de valeurs de type Integer ou Double.

Exemple (code Java 5.0) :

```
import java.util.Arrays;
import java.util.List;

public class MaClasse {

  public static void main(String... args) {
    List<Integer> entiers = Arrays.asList(1, 2, 3);
    System.out.println("Somme = " + sommer(entiers));
    List<Double> doubles = Arrays.asList(1.1, 2.2, 3.3);
    System.out.println("Somme = " + sommer(doubles));
  }

  public static double sommer(List<? extends Number> valeurs) {
    double s = 0.0;
    for (Number n : valeurs)
        s += n.doubleValue();
    return s;
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java

C:\java>java MaClasse
Somme = 6.0
Somme = 6.6
```

Cela est possible car par définition pour être valide les objets contenus dans la liste héritent de la classe Number. On a donc la garantie que toutes les instances possèdent au moins les méthodes de la classe Number.

Il est possible de lire un élément qui sera nécessairement compatible avec le type de la borne supérieure Number puisque les éléments contenus seront des sous-types de Number. Lorsque l'on accède à un élément d'une liste avec une borne supérieure, la référence obtenue peut être affecté de manière fiable à une référence de type borne supérieure. Cela est possible car il est garanti que tous les éléments soit du type de la borne supérieure ou l'une de ses classes-filles.

Exemple (code Java 5.0) :

```
import java.util.List;
import java.util.ArrayList;

public class MaClasse {

  public static void main(String... args) {
    List<Integer> entiers = new ArrayList<Integer>();
    entiers.add(1);
    List<? extends Number> liste = entiers;
    Number valeur = liste.get(0);
    System.out.println(valeur);
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java

C:\java>java MaClasse
1
```

Autre exemple, l'interface Collection contient la méthode addAll() dont la signature est la suivante :

Exemple (code Java 5.0) :

```
public interface Collection<E> {
  // ...
  public boolean addAll(Collection<? extends E> c);
  // ...
}
```

Elle permet d'ajouter tous les éléments d'une autre collection contenant des éléments compatibles à la collection.

La Collection attendue en paramètre est typée avec un wildcard avec une borne supérieure avec le paramètre de type E de la Collection. Ainsi, il est possible d'ajouter tous les éléments d'une Collection contenant des objets de type E ou d'un sous-type de E.

Exemple (code Java 5.0) :

```
import java.util.Arrays;
import java.util.ArrayList;
import java.util.List;

public class MaClasse {

  public static void main(String... args) {
    List<Number> nombres = new ArrayList<Number>();
    List<Integer> entiers = Arrays.asList(1, 2, 3);
    List<Double> doubles = Arrays.asList(1.1, 2.2, 3.3 );
    nombres.addAll(entiers);
    nombres.addAll(doubles);
    System.out.println(nombres);
  }
}
```

Résultat :

```
[1, 2, 3, 1.1, 2.2, 3.3]
```

Ce code est valide et fonctionne car la méthode addAll() attend en paramètre quelque chose de type Collection<? extends E>. Dans l'exemple ci-dessus, E correspond au type Number. Les deux List, respectivement typées avec Integer et Double passées en paramètre dans l'exemple sont compatibles avec ce type.

La mise en oeuvre d'une collection avec un type paramétré avec un wildcard avec borne supérieure ne permet pas d'ajouter des éléments dans la collection. La raison est que lorsque l'on récupère une valeur, le compilateur n'a aucune idée de son type, mais seulement qu'il hérite du type utilisé comme borne supérieure. Le compilateur ne pouvant assurer le type safety, puisqu'il est possible d'utiliser n'importe quel sous-type, l'ajout d'un élément dans une telle collection est interdit et génère une erreur par le compilateur.

Par exemple, `Collection<? extends Number>` permet d'utiliser n'importe quelle instance qui est un sous-type de Number. Il n'y a aucun moyen de savoir lequel de ses sous-types et donc de garantir la sécurité de type : il n'est donc pas possible d'ajouter l'élément dans la collection.

Exemple (code Java 5.0) :

```
import java.util.List;
import java.util.ArrayList;

public class MaClasse {

  public static void main(String... args) {
    List<Integer> entiers = new ArrayList<Integer>();
    List<? extends Number> liste = entiers;
    liste.add(1);
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:9: error: no suitable method found for add(int)
        liste.add(1);
             ^
    method Collection.add(CAP#1) is not applicable
      (argument mismatch; int cannot be converted to CAP#1)
    method List.add(CAP#1) is not applicable
      (argument mismatch; int cannot be converted to CAP#1)
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Number from capture of ? extends Number
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full output
1 error
```

Remarque : il n'est pas possible d'ajouter de nouveaux éléments dans une collection avec un type paramétré avec un wildcard avec une borne supérieure, hormis la valeur null.

Un type paramétré avec un wildcard ne peut pas être utilisé comme type générique avec l'opérateur new.

Exemple (code Java 5.0) :

```
import java.util.List;
import java.util.ArrayList;

public class MaClasse {

  public static void main(String[] args) {
    List<? extends Number> list= new ArrayList<? extends Number>();
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:7: error: unexpected type
    List<? extends Number> list= new ArrayList<? extends Number>();
                                              ^
  required: class or interface without bounds
  found:    ? extends Number
1 error
```

Il faut obligatoirement utiliser un type dont le bytecode peut être chargé dans l'argument de type avec l'opérateur new.

Exemple (code Java 5.0) :

```
import java.util.List;
import java.util.ArrayList;

public class MaClasse {

  public static void main(String[] args) {
    List<? extends Number> list= new ArrayList<Integer>();
  }
}
```

Il est possible d'utiliser de manière équivalente un paramètre de type borné à la place d'un wildcard avec borne supérieure.

Exemple (code Java 5.0) :

```
  public static double sommer(List<? extends Number> valeurs) {
    double s = 0.0;
    for (Number n : valeurs)
        s += n.doubleValue();
    return s;
  }
```

L'exemple ci-dessus est équivalent à l'exemple ci-dessous

Exemple (code Java 5.0) :

```
  public static <E extends Number> double sommer(List<E> valeurs) {
    double s = 0.0;
    for (Number n : valeurs)
        s += n.doubleValue();
    return s;
  }
```

5.9.2.4. Les wildcards avec bornes inférieures (Lower Bounded Wildcards)

Pour qu'un type générique accepte tous les types qui sont un super-type d'un type particulier, il faut utiliser un wildcard avec une borne inférieure.

Ainsi dans une List avec un wildcard avec borne inférieure, il n'est possible de lui affecter qu'une List dont l'argument de type est le type de la borne ou un de ces super-types.

Une borne inférieure se définit en faisant suivre le wildcard par le mot clé super et le type de la borne inférieure : <? super T> indique un type qui peut être T ou un super-type de T.

Exemple avec List<? super Integer> indique qu'il sera possible d'utiliser List<Integer> ou List<Number> ou List<Object>.

Exemple (code Java 5.0) :

```
import java.util.Arrays;
import java.util.ArrayList;
import java.util.List;

public class MaClasse {

  public static void main(String... args) {
    List<? super Integer> valeurs = new ArrayList<Integer>();
    valeurs = new ArrayList<Number>();
    valeurs = new ArrayList<Object>();
  }
}
```

En fait, List<? super Integer> n'est pas un type défini : c'est un modèle décrivant un ensemble de types qui sont autorisés comme argument de type. Il est important de noter que List<Integer> n'est pas un sous-type de List<? super Integer>, mais un type qui correspond à ce modèle.

Une List<Integer> est plus restrictive que List<? super Integer> car la première correspond à une liste de type Integer uniquement, tandis que la seconde correspond à une liste de type Integer de tout type qui est un super-type d'Integer.

L'utilisation d'une borne inférieure est requise pour permettre l'ajout des éléments dans la collection.

Exemple : la classe Collections possède la méthode addAll() dont la signature est :

```
public static <T> boolean addAll(Collection<? super T> c, T... elements) ;
```

Elle permet d'ajouter tous les éléments du varargs de type T dans la Collection fournis en paramètre.

Le type paramétré <? super T> signifie que la liste de destination peut avoir des éléments de tout type qui est un super-type de T. La méthode accepte en entrée toute liste de type T ou d'un super-type de T. Ce cas exprime la version contravariante du type générique List<T>.

Exemple (code Java 5.0) :

```
import java.util.Arrays;
import java.util.ArrayList;
import java.util.List;

public class MaClasse {

  public static void main(String... args) {
    List<Integer> entiers = Arrays.asList(1, 2, 3);
    List<? super Integer> valeurs = new ArrayList<Integer>();
    valeurs.addAll(entiers);
    valeurs.add(4);
    Object valeur = valeurs.get(0);
    Integer entier = (Integer) valeurs.get(0);
    System.out.println(valeurs);
  }
}
```

Résultat :

```
[1, 2, 3, 4]
```

L'intuition pourrait permettre de conclure qu'une List<? super Fille> est une List d'instance de type Fille ou d'un super-type de Fille et que donc il est possible d'ajouter des instances d'un de ses super-types. Ce raisonnement est erroné.

La déclaration List<? super Fille> garantit que la liste sera d'un type permettant d'ajouter à la liste de tout ce qui est une instance de Fille. Lorsque l'on utilise des wildcards, ceux-ci s'appliquent au type de la List passée en argument à la méthode, et non au type de l'élément lors d'ajout d'un élément à la List. Les wildcards ne s'appliquent pas aux éléments eux-mêmes : les restrictions ne s'appliquent qu'au type générique qui les utilisent.

La List se comporte comme toujours lorsque l'on a une collection d'éléments. On ne pourra ajouter que des éléments de type de la borne inférieure ou d'un de ces sous-types. Par exemple avec List<? super Fille>, il sera possible d'ajouter que des instances de Fille ou des sous-types de Fille.

Il n'est pas possible d'ajouter une instance d'un super-type mais il est possible d'ajouter une instance de type Fille, d'un de ses sous-types ou null. Comme Object n'est pas une sous-classe de classe Fille, puisqu'une de ses classes mères, le compilateur n'autorise pas l'ajout d'une instance d'une classe mère dans une telle List.

Exemple (code Java 5.0) :

```
import java.util.Arrays;
import java.util.ArrayList;
import java.util.List;

public class MaClasse {

  public static void main(String... args) {
    List<? super Integer> valeurs = new ArrayList<Integer>();
    valeurs.add(new Object());
    Object valeur = valeurs.get(0);
    System.out.println(valeurs);
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:9: error: no suitable method found for add(Object)
    valeurs.add(new Object());
           ^
    method Collection.add(CAP#1) is not applicable
      (argument mismatch; Object cannot be converted to CAP#1)
    method List.add(CAP#1) is not applicable
      (argument mismatch; Object cannot be converted to CAP#1)
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object super: Integer from capture of ? super Integer
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full output
1 error
```

Les raisons de ces contraintes sont simples : la sécurité de type. S'il était possible d'ajouter une Mere à une List<? super Fille> alors il serait aussi possible d'ajouter une AutreFille qui une classe fille de Mere. Mais une Fille n'est pas une AutreFille.

Afin d'appliquer l'effacement de type, le compilateur doit concrétiser un wildcard avec une borne en un type spécifique en utilisant l'inférence.

Dans l'exemple, le compilateur infère que c'est une List<Fille>.

Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.List;

public class MaClasse {

  public static void main(String... args) {
    List<Object> objets = new ArrayList<Object>();
    List<Fille> filles = new ArrayList<Fille>();
    ajouter(objets);
    ajouter(filles);
    System.out.println(objets);
    System.out.println(filles);
  }

  public static void ajouter(List<? super Fille> liste){
    liste.add(new Fille());
    liste.add(new PetiteFille());
  }
}

class Mere {
}

class Fille extends Mere {
}

class PetiteFille extends Fille {
}
```

Résultat :

```
[Fille@cac736f, PetiteFille@5e265ba4]
[Fille@36aa7bc2, PetiteFille@76ccd017]
```

Le compilateur vérifie que la méthode add() de List a bien en paramètre une instance de Fille ou un sous-type de Fille.

Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.List;

public class MaClasse {

  public static void main(String... args) {
    List<Object> objets = new ArrayList<Object>();
    List<Fille> filles = new ArrayList<Fille>();
    ajouter(objets);
    ajouter(filles);
    System.out.println(objets);
    System.out.println(filles);
  }

  public static void ajouter(List<? super Fille> liste){
    liste.add(new Fille());
    liste.add(new PetiteFille());
    liste.add(new Mere());
  }
}

class Mere {
}

class Fille extends Mere {
}

class PetiteFille extends Fille {
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:18: error: no suitable method found for add(Mere)
    liste.add(new Mere());
         ^
    method Collection.add(CAP#1) is not applicable
      (argument mismatch; Mere cannot be converted to CAP#1)
    method List.add(CAP#1) is not applicable
      (argument mismatch; Mere cannot be converted to CAP#1)
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object super: Fille from capture of ? super Fille
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full output
1 error
```

L'information indiquée par le compilateur dit que CAP#1 extends Object super: Fille from capture of ? super Fille. Il faut considérer 'Object super : Fille' comme un objet dont Fille est un supertype, ou plus simplement, un objet de type Fille. Une instance de Mere n'est pas une instance de Fille, et donc la compilation échoue.

Attention, lors de l'exploitation des données contenues dans la List : le type des objets est celui inféré et un cast vers le type de la borne peut échouer à l'exécution si le type inféré est un des super-types de la borne.

Exemple (code Java 5.0) :

```
import java.util.Arrays;
import java.util.List;

public class MaClasse {

  public static void main(String... args) {
    List<Object> objets = Arrays.asList(new Object());
    List<Integer> valeurs = Arrays.asList(1, 2, 3);
    afficherPremier(valeurs);
    afficherPremier(objets);
  }

  public static void afficherPremier(List<? super Integer> liste ) {
    Object valeur = liste.get(0);
    try {
      Integer entier = (Integer) liste.get(0);
    } catch(Exception e) {
      e.printStackTrace();
    }
    System.out.println(liste);
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java

C:\java>java MaClasse
[1, 2, 3]
java.lang.ClassCastException: java.lang.Object cannot be cast to java.lang.Integer
        at MaClasse.afficherPremier(MaClasse.java:16)
        at MaClasse.main(MaClasse.java:10)
[java.lang.Object@15db9742]
```

5.9.2.5. Comment choisir entre borne inférieure et supérieure

L'un des aspects qui peut être à l'origine de confusion lors de la mise en oeuvre des génériques est de déterminer quand il faut utiliser un wildcard avec borne inférieure et avec borne supérieure.

Les mêmes principes sont utilisés dans différents contextes avec des noms différents :

- in/out : utilisé dans le tutorial Java SE
- producer/consumer : utilisé par Joshua Block dans son livre de référence "Effective Java"
- get/put

Quelques soient leurs noms, ces principes doivent être utilisés comme guide pour décider de l'utilisation d'un wildcard avec ou sans borne supérieure ou inférieure.

Le principe in/producer/get représente des données à lire. Il faut utiliser un wildcard avec borne supérieure et donc utiliser le mot clé extends

Le principe out/consumer/put représente des données à écrire. Il faut utiliser un wildcard avec borne inférieure et donc utiliser le mot clé super

Pour choisir d'utiliser une borne inférieure ou supérieure, il faut tenir compte de plusieurs contraintes :

- L'utilisation d'une borne inférieure, avec le mot clé extends concernent des objets désignés par les termes Producer : elle permet uniquement d'ajouter des valeurs. Elle permet de mettre en oeuvre la covariance. Il faut utiliser <? extends T> s'il faut obtenir des objets de type T d'une collection. Par exemple, la méthode addAll(Collection<? extends E> c) de l'interface Collection permet d'ajouter tous les éléments de la collection passée en paramètre à l'instance de la collection : elle doit donc les lire et le type paramétré utilise extends.
- L'utilisation d'une borne supérieure, avec le mot clé super concernent des objets désignés par les termes Consumer : elle permet uniquement de lire des valeurs. Elle permet de mettre en oeuvre la covariance. Il faut utiliser <? super T> s'il faut ajouter des éléments de type T dans la collection. Par exemple, la méthode static <T> boolean addAll(Collection<? super T> c, T... elements) de l'interface Collection permet d'ajouter tous les éléments du vargargs à la collection passée en premier paramètre : elle doit donc modifier la collection et le type paramétré utilise super.
- Pour permettre de lire et d'ajouter d'une valeur, il faut utiliser un type explicite. Il ne faut donc pas utiliser de wildcard si des éléments doivent être lus et ajoutés dans la collection

L'utilisation d'une borne supérieure ne rend pas la collection totalement en lecture seule car il est tout de même possible d'ajouter null à la collection.

L'utilisation d'une borne inférieure ne rend pas la collection totalement en écriture seule car il est tout de même possible de lire des éléments en tant qu'Object.

Ces contraintes s'illustrent bien dans la méthode copy() de la classe Collections dont la signature est : copy(List<? super T> dest, List<? extends T> src). Elle permet de copier tous les éléments de la List src dans la List dest. Il est donc nécessaire de lire les éléments de la List src, raison pour laquelle son type paramétré utilise extends. Il est aussi nécessaire d'ajouter les éléments dans le List dest, raison pour laquelle son type paramétré utilise super.

Pour résumer :

- une borne supérieure rend une collection en lecture seule
- une borne inférieure rend une collection en écriture seule

Il est préférable d'utiliser les bornes inférieures et supérieures dans les paramètres d'entrée. Il n'est pas recommandé de les utiliser comme types de retour pour restreindre les manipulations de données potentielles.

L'utilisation d'un wildcard comme type de retour n'est pas recommandé car cela oblige les appelants qui invoquent la méthode à traiter avec les jokers.

5.9.2.6. Les wildcards et le sous-typage

Les types génériques ne proposent pas de relations entre-eux même s'il existe une relation entre les types utilisés comme paramètre dans les types génériques.

Cependant, il est possible d'utiliser un wildcard pour définir une relation entre des types génériques.

Soit une classe Mere et une classe Fille dont elle hérite. Grâce à l'héritage en Java, il est de créer une instance de type Fille et de l'affecter à une variable de type Mere car la classe Fille est une sous-classe de la classe Mere.

Exemple :

```
Fille fille = new Fille();
Mere mere = fille;
```

Cela ne s'applique pas aux types génériques

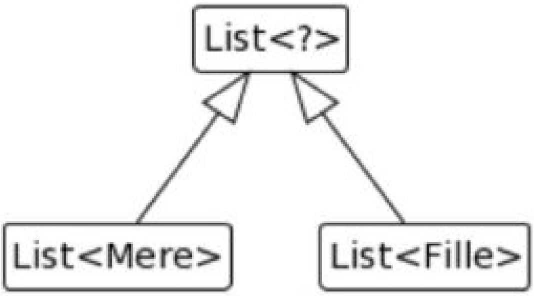
Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.List;

public class Mere {

  public static void main(String[] args) {
    Fille fille = new Fille();
    Mere mere = fille;

    List<Fille> filles = new ArrayList<Fille>();
    List<Mere> meres = filles;
  }
}

class Fille extends Mere {
}
```

Résultat :

```
C:\java>javac Mere.java
Mere.java:11: error: incompatible types: List<Fille> cannot be converted to List<Mere>
    List<Mere> meres = filles;
                       ^
1 error
```

Bien que la classe Fille est un sous-type de la classe Mere, List<Fille> n'est pas un sous-type de List<Number>. Le type commun de List<Mere> et de List<Fille> est List<?>.



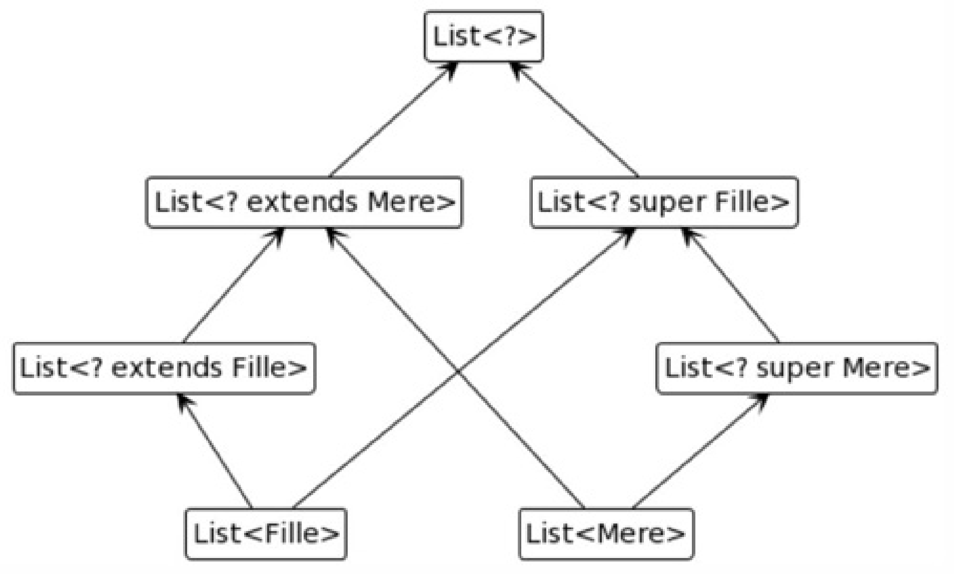
Pour qu'il y ait une relation entre ces classes, il faut utiliser un wildcard avec une borne supérieure.

Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.List;

public class Mere {

  public static void main(String[] args) {
    Fille fille = new Fille();
    Mere mere = fille;

    List<? extends Fille> filles = new ArrayList<Fille>();
    List<? extends Mere>  meres = filles;
  }
}

class Fille extends Mere {
}
```

L'exemple ci-dessus se compile sans erreur car List<? extends Fille> est un sous-type de List<? extends Mere>. La classe Fille héritant de la classe Mere, il y a une relation entre List<? extends Fille> et List<? extends Mere>.

Le diagramme ci-dessous illustre les relations entre List génériques qui utilisent des wildcards avec bornes supérieures et inférieures.



5.9.2.7. La capture des wildcards et les méthodes helper

L'utilisation d'un type paramétré avec un wildcard empêche de savoir qu'elles sont les signatures des méthodes et donc quelles méthodes des objets pourront être invoquées.

Exemple avec List<?> : quel serait le type retourné par la méthode get() et quel serait le type du paramètre de la méthode add().

Pour permettre leur invocation, il faudra que l'objet ait un type concret. Lors de la compilation, ce type concret n'est pas connu.

Le compilateur utilise un mécanisme nommé capture lorsque le type paramétré utilise un wildcard : chaque wildcard est remplacé par une variable de type. Ainsi, le compilateur ne doit traiter que les objets de type concret.

La capture des wildcards dépend des bornes des wildcards et des bornes des paramètres de types.

Chaque paramètre de type à une borne supérieure qui limite les types avec lesquels il peut être substitué. La limite supérieure par défaut est Object. Exemple :

List<T> : la borne est Object

List<T extends Number> : la borne est Number

En Java, les paramètres de type n'ont pas de bornes inférieures.

Un wildcard possède soit une borne inférieure soit une borne supérieure mais par les deux. Par défaut, un wilcard possède une borne inférieure et supérieure sur Object.

Lors de la capture d'un wildcard avec borne supérieure, celui-ci est remplacé par une nouvelle variable de type, qui prend la limite supérieure du wildcard, et la limite supérieure du paramètre de type.

Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.List;

public class ListUtils {

  public static void traiter(List<? extends Number> nombres) {
    Number nombre = nombres.get(0);
        System.out.println(nombre);
  }

  public static void main(String[] args) {
          List<Integer> liste = new ArrayList<Integer>();
          liste.add(1);
          traiter(liste);
  }
}
```

Dans l'exemple ci-dessus, le compilateur opère la capture de List<? extends Number> en List<XXX> ou XXX est un sous-type de Number. Au moment de la compilation le compilateur ne connait pas précisément le type XXX mais il sait que XXX doit être un sous-type de Number.

Comme XXX est un sous type Number, il est tout à fait légal d'assigner la valeur de retour de la méthode get() à une variable de type Number.

La méthode add(XXX) où XXX est un sous-type de Number ne peut cependant pas être utilisé avec un type Number car Number n'est pas sous-type de XXX.

Exemple (code Java 5.0) :

```
import java.util.List;

public class ListUtils {

  public void ajouter(List<? extends Number> nombres, Number nombre) {
    nombres.add(nombre);
  }
}
```

Résultat :

```
C:\java>javac ListUtils.java
ListUtils.java:6: error: no suitable method found for add(Number)
    nombres.add(nombre);
           ^
    method Collection.add(CAP#1) is not applicable
      (argument mismatch; Number cannot be converted to CAP#1)
    method List.add(CAP#1) is not applicable
      (argument mismatch; Number cannot be converted to CAP#1)
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Number from capture of ? extends Number
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full output
1 error
```

Lors de la compilation, le compilateur utilise un nom qui lui est propre, par exemple de la forme CAP#n, pour les variables de type issues de la capture.

Le compilateur applique la capture sur chaque expression qui produit une valeur dans un objet défini avec un type paramétré utilisant un wildcard.

Exemple (code Java 5.0) :

```
import java.util.List;

public class ListUtils {

  void ajouter(List<? extends Number> nombres) {
    nombres.add(nombres.get(0));
}
```

Résultat :

```
C:\java>javac ListUtils.java
ListUtils.java:6: error: no suitable method found for add(CAP#1)
    nombres.add(nombres.get(0));
           ^
    method Collection.add(CAP#2) is not applicable
      (argument mismatch; Number cannot be converted to CAP#2)
    method List.add(CAP#2) is not applicable
      (argument mismatch; Number cannot be converted to CAP#2)
  where CAP#1,CAP#2 are fresh type-variables:
    CAP#1 extends Number from capture of ? extends Number
    CAP#2 extends Number from capture of ? extends Number
```

Lors de la compilation de cet exemple, le compilateur réalise deux captures mais il ne peut pas s'assurer que les types concrets qui seront utilisés seront les mêmes.

Les variables de type introduites par les captures sont inutilisables dans le code source. Leurs noms sont donnés par le compilateur (exemple "CAP#1"). Elles n'ont pas de noms propres qu'il serait possible d'utiliser dans le code source.

Il est possible de gérer ces cas en définissant une méthode générique avec des variables de type nommées. Une telle méthode est désignée sous le nom capture helper.

Le but d'une méthode capture helper est d'attribuer un nom à la variable de type introduite par la capture conversion. Cette technique est utile pour permettre de faire référence dans le code au type d'un type paramétré avec un wildcard.

Exemple (code Java 5.0) :

```
public class Paire<T> {

  private T val1;
  private T val2;

  public Paire(T val1, T val2) {
    this.val1 = val1;
    this.val2 = val2;
  }

  public T getVal1() {
    return val1;
  }

  public void setVal1(T val1) {
    this.val1 = val1;
  }

  public T getVal2() {
    return val2;
  }

  public void setVal2(T val2) {
    this.val2 = val2;
  }

  static <V> void intervertirValeurs(Paire<V> paire) {
    V getVal1 = paire.getVal1();
    paire.setVal1(paire.getVal2());
    paire.setVal2(getVal1);
  }

  public static void main(String[] args) {
    Paire<Integer> p1 = new Paire<Integer>(10, 20);
    System.out.println("Avant : " + p1.getVal1() + ", " + p1.getVal2());

    Paire<?> p2 = p1;
    intervertirValeurs(p2);

    System.out.println("Apres : " + p2.getVal1() + ", " + p2.getVal2());
    System.out.println(p2.getVal1().getClass().getName());
  }
}
```

Résultat :

```
C:\java>javac Paire.java

C:\java>java Paire
Avant : 10, 20
Apres : 20, 10
java.lang.Integer
```

La définition d'une variable avec un type paramétré utilisant un wilcard n'a que peu d'intérêt hormis de montrer que dans ce cas le compilateur peut réaliser la capture.

Une autre possibilité serait d'utiliser un type paramétré avec un wildcard pour le paramètre de la méthode.

Exemple (code Java 5.0) :

```
  static void intervertirValeurs(Paire<?> paire) {

    Object getVal1 = paire.getVal1();
    paire.setVal1(paire.getVal2());
    paire.setVal2(getVal1);
  }

  public static void main(String[] args) {
    Paire<Integer> p1 = new Paire<Integer>(10, 20);
    System.out.println("Avant : " + p1.getVal1() + ", " + p1.getVal2());

    Paire<Integer> p2 = p1;
    intervertirValeurs(p2);

    System.out.println("Apres : " + p2.getVal1() + ", " + p2.getVal2());
    System.out.println(p2.getVal1().getClass().getName());
  }
```

Résultat :

```
C:\java>javac Paire.java
Paire.java:30: error: incompatible types: Object cannot be converted to CAP#1
    paire.setVal1(paire.getVal2());
                               ^
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
Paire.java:31: error: incompatible types: Object cannot be converted to CAP#1
    paire.setVal2(getVal1);
                  ^
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full output
2 errors
```

Le code ne compile pas car lors de l'invocation des méthodes setVal1() et setVal2() le type des paramètres passés n'est pas connu à cause de l'utilisation du wildcard.

Dans cet exemple, le compilateur traite le paramètre comme étant de type Paire d'Object. Lorsque la méthode invoque les setters, le compilateur n'est pas en mesure de confirmer le type d'objet qui est inséré dans la liste, et une erreur est produite. Lorsque ce type d'erreur se produit, cela signifie généralement que le compilateur pense que vous affectez le mauvais type à une variable qui va à l'encontre du but des génériques qui est de renforcer la sécurité des types à la compilation.

La solution pour contourner cette erreur de compilation est de définir une méthode helper dont le type paramétré utilise une variable nommée. Le compilateur peut alors faire la capture et on peut faire référence au type paramétré dans le code.

Exemple (code Java 5.0) :

```
  static <V> void intervertirValeursHelper(Paire<V> paire) {
    V getVal1 = paire.getVal1();
    paire.setVal1(paire.getVal2());
    paire.setVal2(getVal1);
  }

  static <V> void intervertirValeurs(Paire<?> paire) {
    intervertirValeursHelper(paire);
  }

  public static void main(String[] args) {
    Paire<Integer> p1 = new Paire<Integer>(10, 20);
    System.out.println("Avant : " + p1.getVal1() + ", " + p1.getVal2());

    Paire<?> p2 = p1;
    intervertirValeurs(p2);

    System.out.println("Apres : " + p2.getVal1() + ", " + p2.getVal2());
    System.out.println(p2.getVal1().getClass().getName());
  }
```

Grâce à la méthode helper, le compilateur utilise l'inférence pour déterminer que T est CAP#1, la variable de capture, dans l'invocation. Le code se compile et s'exécute correctement

Résultat :

```
C:\java>javac Paire.java

C:\java>java Paire
Avant : 10, 20
Apres : 20, 10
java.lang.Integer
```

Par convention de nom de la méthode helper est le nom de la méthode suivi de « Helper ».

La problématique est identique lorsque l'on tente d'écrire une méthode qui intervertit les deux premiers éléments d'une liste. Une première version d'une telle méthode attend en paramètre une List<?> pour permettre une utilisation sur une liste contenant n'importe quel éléments.

Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.List;

public class MaClasse {

  public static void intervertirDeuxPremiers(List<?> liste) {
    if (liste.size() > 1) {
      Object valeur = liste.get(0);
      liste.set(0, liste.get(1));
      liste.set(1, valeur);
    }
  }

  public static void main(String... args) {
    List<String> liste = new ArrayList<String>();
    liste.add("valeur1");
    liste.add("valeur2");
    intervertirDeuxPremiers(liste);
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:9: error: incompatible types: Object cannot be converted to CAP#1
      liste.set(0, liste.get(1));
                            ^
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
MaClasse.java:10: error: incompatible types: Object cannot be converted to CAP#1
      liste.set(1, valeur);
                   ^
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full output
2 errors
```

La solution est de nouveau d'utiliser une méthode helper qui va permettre d'utiliser le type générique défini dans la signature pour pouvoir l'utiliser dans le code source.

Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.List;

public class MaClasse {

  public static void intervertirDeuxPremiers(List<?> liste) {
    intervertirDeuxPremiersHelper(liste);
  }

  private static <T> void intervertirDeuxPremiersHelper(List<T> liste) {
    if (liste.size() > 1) {
      T valeur = liste.get(0);
      liste.set(0, liste.get(1));
      liste.set(1, valeur);
    }
  }

  public static void main(String... args) {
    List<String> liste = new ArrayList<String>();
    liste.add("valeur1");
    liste.add("valeur2");
    intervertirDeuxPremiers(liste);
  }
}
```

Malheureusement, une méthode capture helper ne permet de résoudre tous les cas.

Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.List;

public class MaClasse {

  public static void main(String... args) {
    List<String> l1 = new ArrayList<String>();
    List<String> l2 = new ArrayList<String>();
    l1.add("valeur1");
    l2.add("valeur2");
    intervertirPremier(l1,l2);
    System.out.println("liste1 = "+l1);
    System.out.println("liste2 = "+l2);
  }
  public static void intervertirPremier(List<?> liste1, List<?> liste2) {
    Object valeur = liste1.get(0);
    liste1.set(0, liste2.get(0));
    liste2.set(0, valeur);
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:18: error: incompatible types: Object cannot be converted to CAP#1
    liste1.set(0, liste2.get(0));
                            ^
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
MaClasse.java:19: error: incompatible types: Object cannot be converted to CAP#1
    liste2.set(0, valeur);
                  ^
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full output
2 errors
```

La mise en oeuvre d'une méthode capture helper ne permet de résoudre le problème ou plutôt en implique un autre.

Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.List;

public class MaClasse {

  public static void main(String... args) {
    List<String> l1 = new ArrayList<String>();
    List<String> l2 = new ArrayList<String>();
    l1.add("valeur1");
    l2.add("valeur2");
    intervertirPremier(l1,l2);
    System.out.println("liste1 = "+l1);
    System.out.println("liste2 = "+l2);
  }

  public static void intervertirPremier(List<?> liste1, List<?> liste2) {
    intervertirPremierHelper(liste1, liste2);
  }

  public static <T> void intervertirPremierHelper(List<T> liste1, List<T> liste2) {
    T valeur = liste1.get(0);
    liste1.set(0, liste2.get(0));
    liste2.set(0, valeur);
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:17: error: method intervertirPremierHelper in class MaClasse cannot be applied
 to given types;
    intervertirPremierHelper(liste1, liste2);
    ^
  required: List<T>,List<T>

  found: List<CAP#1>,List<CAP#2>
  reason: inferred type does not conform to equality constraint(s)
    inferred: CAP#2
    equality constraints(s): CAP#2,CAP#1
  where T is a type-variable:
    T extends Object declared in method <T>intervertirPremierHelper(List<T>,List<T>)
  where CAP#1,CAP#2 are fresh type-variables:
    CAP#1 extends Object from capture of ?
    CAP#2 extends Object from capture of ?
1 error
```

Le compilateur ne peut pas garantir qu'à l'exécution les types des deux listes soient exactement le même à cause de l'utilisation des wildcards. La solution dans ce cas est de ne pas utiliser de wilcards.

Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.List;

public class MaClasse {

  public static void main(String... args) {
    List<String> l1 = new ArrayList<String>();
    List<String> l2 = new ArrayList<String>();
    l1.add("valeur1");
    l2.add("valeur2");
    intervertirPremier(l1,l2);
    System.out.println("liste1 = "+l1);
    System.out.println("liste2 = "+l2);
  }

  public static <T> void intervertirPremier(List<T> liste1, List<T> liste2) {
    T valeur = liste1.get(0);
    liste1.set(0, liste2.get(0));
    liste2.set(0, valeur);
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java

C:\java>java MaClasse
liste1 = [valeur2]
liste2 = [valeur1]
```

5.10. Les bornes multiples (Multiple Bounds) avec l'intersection de types

L'intersection de types est une des fonctionnalités avancées des génériques en Java.

Le type paramétré peut avoir plusieurs bornes en utilisant l'intersection de types. Cela permet à une variable de type ou à un wildcard d'avoir plusieurs bornes.

Une intersection de types est une forme de type anonyme créée en combinant au moins deux types différents.

Par exemple :

<T extends A> T est un sous type de A

< T extends A & B> T est un sous type de A et B

L'intersection de types définit un type anonyme qui n'établit aucune relation hiérarchique entre les types combinés.

Exemple : deux interfaces qui proposent chacune une fonctionnalité

Exemple :

```
package com.jmdoudoux.dej;

public interface Journalisable {
  void journaliser(String message);
}
```

Exemple :

```
package com.jmdoudoux.dej;

public interface Calculable {
  String calculer();
}
```

Le besoin est de pouvoir passer en paramètre d'une méthode, un objet qui implémente ces deux interfaces.

Il est possible de définir une classe qui implémente ces deux interfaces.

Exemple :

```
package com.jmdoudoux.dej;

public class MonTraitement implements Journalisable, Calculable {

  @Override
  public void journaliser(String message) {
    System.out.println(message);
  }

  @Override
  public String calculer() {
    return "MaClasse";
  }
}
```

Il est possible de définir et d'utiliser la méthode qui attend en paramètre la classe.

Exemple :

```
package com.jmdoudoux.dej;

public class MaClasse {

  static void traiter(MonTraitement traitement) {
    String valeur = traitement.calculer();
    traitement.journaliser(valeur);
  }

  public static void main(String[] args) {
    MonTraitement mc = new MonTraitement();
    traiter(mc);
  }
}
```

Cela fonctionne mais cela introduit un couplage avec ce type particulier, ce qui manque de souplesse.

Il est possible de définir une interface qui hérite des deux interfaces.

Exemple :

```
package com.jmdoudoux.dej;

public interface JournalisableCalculable extends Journalisable, Calculable {
}
```

Il suffit alors que les classes implémentent cette interface qui devient le type attendu en paramètre de la méthode traiter().

Exemple :

```
package com.jmdoudoux.dej;

public class MaClasse {
  static void traiter(JournalisableCalculable traitement) {
    String valeur = traitement.calculer();
    traitement.journaliser(valeur);
  }

  public static void main(String[] args) {
    MonTraitement mc = new MonTraitement();
    traiter(mc);
  }
}
```

Ce mécanisme est type-safe mais requiert plus de code et manque aussi de souplesse.

Une autre possibilité est d'utiliser l'intersection de type dans un générique.

Exemple (code Java 5.0) :

```
package com.jmdoudoux.dej;

public class MaClasse {
  static <T extends Journalisable & Calculable> void traiter(T traitement) {
    String valeur = traitement.calculer();
    traitement.journaliser(valeur);
  }

  public static void main(String[] args) {
    MonTraitement mc = new MonTraitement();
    traiter(mc);
  }
}
```

Avec une telle implémentation, il est possible de passer en paramètre de la méthode toute instance qui implémente les deux interfaces.

Cette solution est type-safe ne requiert pas la création de types superflus.

Un paramètre de type paramétré peut avoir plusieurs bornes. Les bornes sont séparées par un caractère & dans la définition pour former une intersection de types. Exemple :

`<T extends Runnable & AutoCloseable>`

`<T extends Comparable & Serializable>`

`<T extends Number & Comparable<? super T>>`

Les bornes multiples peuvent rapidement devenir non triviale à lire et à comprendre. Exemple :

`List<Number & Comparable<? extends Number & Comparable<?>>>`

Le premier type peut être une classe ou une interface. Les bornes multiples suivent les mêmes contraintes que celles suivies par une classe : le type ne peut pas étendre deux classes et si une des bornes est une classe, elle doit être en premier suivie éventuellement par une ou plusieurs interfaces.

Les bornes peuvent donc toutes être des interfaces.

Par exemple, il est possible de définir une contrainte telle que le type doit être un CharSequence et qu'il implémente l'interface Comparable.

Exemple (code Java 5.0) :

```
public class MaClasse {
  public static <T extends CharSequence & Comparable<T>> int comparer(T v1, T v2) {
    return v1.compareTo(v2);
  }
}
```

Cela permet de s'assurer que seules des instances de CharSequence qui implémente l'interface Comparable pourront être passées en paramètres.

Un seul des types bornés peut être une classe et elle doit être obligatoirement en première position. Les autres types doivent être des interfaces.

Exemple (code Java 5.0) :

```
public class MaClasse {

  public static <T extends Number & Comparable<? super T>> int comparer(T v1, T v2){
    return v1.compareTo(v2);
  }
}
```

Dans le cas contraire, le compilateur émet une erreur. C'est notamment le cas, si les bornes sont des classes :

Exemple (code Java 5.0) :

```
public class MaClasse {

  public static <T extends String & Number > void afficher(T v1) {
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:3: error: interface expected here
  public static <T extends String & Number > void afficher(T v1) {
                                    ^
1 error
```

C'est aussi le cas si la classe n'est pas en première position :

Exemple (code Java 5.0) :

```
public class MaClasse {

  public static <T extends Comparable<? super T> & Number > int comparer(T v1, T v2) {
    return 0;
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:3: error: interface expected here
  public static <T extends Comparable<? super T> & Number > int comparer(T v1, T v2) {
                                                   ^
1 error
```

Il est parfois nécessaire d'utiliser l'intersection de type notamment pour garantir la rétro compatibilité du code pré générique.

Par exemple, la méthode max() de la classe Collections :

```
public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)
```

Le paramètre générique étend Object et Comparable pour des raisons de compatibilité.

La signature historique de la méthode max() est :

```
public static Object max(Collection)
```

A cause du type erasure, si la signature générique de la méthode avait été :

```
public static <T extends Comparable<? super T>> max(Collection<? extends T>)
```

Alors la signature de la méthode dans le bytecode serait équivalent à :

```
public static Comparable max(Collection)
```

Ce qui est différent de la signature historique. Pour maintenir la même signature, il faut utiliser un paramètre de type avec intersection de type :

```
public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T>)
```

5.11. L'effacement de type (type erasure)

Les génériques en Java ont été ajoutés au langage pour permettre une vérification de type au moment de la compilation mais à cause de leur implémentation, ils n'ont aucune utilité à l'exécution.

Pour s'assurer que le bytecode généré soit toujours compatible avec celui généré par les versions précédentes de Java, le compilateur applique un processus appelé effacement de type (type erasure) sur les génériques lors de la compilation.

Les génériques en Java ne sont donc une fonctionnalité qui n'est utilisable qu'au niveau du compilateur. Le compilateur Java effectue les vérifications mais il met en oeuvre l'effacement de type qui n'inclut aucune information relative aux génériques dans le bytecode. Il remplace les types génériques par Object ou le type d'une borne et ajoute des casts lorsque cela est nécessaire.

5.11.1. Le choix de l'effacement de type

L'un des défis des évolution sur un langage comme Java est qu'il doit supporter la rétrocompatibilité. Lorsque les génériques ont été ajoutés au langage, il a été décidé de ne pas les inclure dans le bytecode produit par le compilateur.

Pour maintenir la compatibilité ascendante et éviter des modifications majeures de l'environnement d'exécution de Java, les génériques sont mis en oeuvre en utilisant une technique appelée effacement de type (type erasure). Cela revient dans le bytecode à ne pas inclure les informations de type supplémentaires et à ajouter des casts lorsque cela est nécessaire.

L'effacement des types permet de pas créer de nouvelles classes pour chaque type paramétré utilisé comme c'est le cas dans d'autres langages qui n'utilisent pas l'effacement de type.

5.11.2. La mise en oeuvre de l'effacement de type

A la compilation, les génériques ne sont pas inclus dans le bytecode : toutes les références aux variables de type sont remplacées. L'effacement de type est mis en oeuvre par le compilateur, basiquement en appliquant plusieurs règles :

- les paramètres de type non bornés sont remplacés par Object
- les paramètres de type borné sont remplacés par le type de leur première borne
- des casts sont insérés lorsque cela est nécessaire
- des méthodes pont (bridge) sont générées au besoin pour préserver le polymorphisme

Ainsi, le bytecode généré par le compilateur ne contient que des classes, interfaces et méthodes normales, sans aucune information relative aux génériques.

Exemple avec une méthode générique :

Exemple (code Java 5.0) :

```
public <T> List<T> methodeGenerique(List<T> liste) {
  // ...
}
```

Le bytecode généré par le compilateur est équivalent à :

Exemple (code Java 5.0) :

```
public List<Object> methodeGenerique(List<Object> liste) {
  // ...
}
```

Mais comme le bytecode ne contient aucune information relative aux génériques, le bytecode généré est plutôt équivalent à :

Exemple (code Java 5.0) :

```
public List methodeGenerique(List liste) {
  // ...
}
```

Pour un type générique, tous les types paramétrés partagent le même type à l'exécution : le type brut (Rawtype) qui correspond au type générique sans l'argument de type.

Si le type est borné, alors le type sera remplacé par la borne au moment de la compilation :

Exemple (code Java 5.0) :

```
public <T extends MaCLasse> void methodeGenerique(T donnees) {
  // ...
}
```

Le bytecode généré par le compilateur est équivalent à :

Exemple (code Java 5.0) :

```
public void methodeGenerique(MaClasse donnees) {
  // ...
}
```

5.11.2.1. L'effacement de type pour un type paramétré inconnu

Une classe ou une interface peut être déclarée avec un ou plusieurs types paramétrés. Le ou les types paramétrés doivent être fournis lors de la création d'une instance avec l'opérateur new.

Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.List;

public class MaClasse {

  public static void main(String... args) {
    List<String> chaines = new ArrayList<String>();
    chaines.add("test");
  }
}
```

L'effacement de type (type erasure) opéré par le compilateur retire tous les types génériques dans le bytecode pour les remplacer par défaut par le type Object. A l'issue de la compilation, les informations de type sont effacées par le compilateur : ainsi le bytecode créé est similaire à celui par le compilateur du JDK 1.4.

Cela garantit une compatibilité du bytecode entre les différentes versions de Java. Ainsi, une List ou une List avec un générique sont toutes représentées par le même type dans le bytecode et donc à l'exécution : le type brut List.

Après avoir compilé la classe, il est possible d'utiliser l'outil javap du JDK pour vérifier l'effacement du type dans le bytecode créé dans le compilateur.

Résultat :

```
C:\java>javap -c MaClasse.class
Compiled from "MaClasse.java"
public class MaClasse {
  public MaClasse();
    Code:
       0: aload_0
       1: invokespecial #1            // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String...);
    Code:
       0: new           #2            // class java/util/ArrayList
       3: dup
       4: invokespecial #3            // Method java/util/ArrayList."<init>":()V
       7: astore_1
       8: aload_1
       9: ldc           #4            // String test
      11: invokeinterface #5,  2      // InterfaceMethod java/util/List.add:(Ljava/lang/Object;)Z
      16: pop
      17: return
}
```

Ceci s'applique aussi si le type paramétré est utilisé comme type d'un paramètre ou d'une valeur de retour d'une méthode. Exemple :

Exemple (code Java 5.0) :

```
public class MonConteneur<T> {

    public T valeur;

    public MonConteneur(T valeur) {
        this.valeur = valeur;
    }

    public T get() {
        return valeur;
    }
}
```

Résultat :

```
C:\java>javap -s MonConteneur.class
Compiled from "MonConteneur.java"
public class MonConteneur<T> {
  public T valeur;
    descriptor: Ljava/lang/Object;
  public MonConteneur(T);
    descriptor: (Ljava/lang/Object;)V
  public T get();
    descriptor: ()Ljava/lang/Object;
}
```

Le paramètre de type est remplacé par Object. Dans le code qui utilise cette classe générique, un cast est ajouté pour obtenir le type correspondant au type générique utilisé.

Exemple (code Java 5.0) :

```
public class MaClasse {

  public static void main() {
    MonConteneur<String> mc = new MonConteneur<>("test");
    String valeur = mc.get();
  }
}
```

L'outil javap peut de nouveau être utilisé pour visualiser le bytecode généré suite à la compilation de la classe.

Résultat :

```
C:\java>javap -c MaClasse.class
Compiled from "MaClasse.java"
public class MaClasse {
  public MaClasse();
    Code:
       0: aload_0
       1: invokespecial #1           // Method java/lang/Object."<init>":()V
       4: return

  public static void main();
    Code:
       0: new           #2          // class MonConteneur
       3: dup
       4: ldc           #3          // String test
       6: invokespecial #4          // Method MonConteneur."<init>":(Ljava/lang/Object;)V
       9: astore_0
      10: aload_0
      11: invokevirtual #5          // Method MonConteneur.get:()Ljava/lang/Object;
      14: checkcast     #6          // class java/lang/String
      17: astore_1
      18: return
}
```

La méthode get() invoquée est celle qui retourne un objet de type Object et un cast vers le type java.lang.String est effectué dans la méthode main().

5.11.2.2. L'effacement de type pour un type paramétré avec borne supérieure

Si le type paramétré utilise une borne supérieure, alors le type utilisé est celui précisé dans la borne.

Exemple (code Java 5.0) :

```
public class MonConteneur<T extends Comparable<T>> {

  public T valeur;

  public MonConteneur(T valeur) {
    this.valeur = valeur;
  }

  public T get() {
    return valeur;
  }
}
```

L'outil javap peut de nouveau être utilisé pour visualiser le bytecode généré suite à la compilation de la classe.

Résultat :

```
C:\java>javap -c MonConteneur.class
Compiled from "MonConteneur.java"
public class MonConteneur<T extends java.lang.Comparable<T>> {
  public T valeur;

  public MonConteneur(T);
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: aload_0
       5: aload_1
       6: putfield      #2                  // Field valeur:Ljava/lang/Comparable;
       9: return

  public T get();
    Code:
       0: aload_0
       1: getfield      #2                  // Field valeur:Ljava/lang/Comparable;
       4: areturn
}
```

Le type utilisé n'est plus Object mais Comparable, celui précisé dans la borne. Si le type paramétré utilise plusieurs types alors c'est le premier défini qui est utilisé comme type dans le bytecode.

Exemple (code Java 5.0) :

```
import java.io.Serializable;

public class MonConteneur<T extends Serializable & Comparable<T>> {

    public T valeur;

    public MonConteneur(T valeur) {
        this.valeur = valeur;
    }

    public T get() {
        return valeur;
    }
}
```

Résultat :

```
C:\java>javap -c MonConteneur.class
Compiled from "MonConteneur.java"
public class MonConteneur<T extends java.io.Serializable & java.lang.Comparable<T>> {
  public T valeur;

  public MonConteneur(T);
    Code:
       0: aload_0
       1: invokespecial #1               // Method java/lang/Object."<init>":()V
       4: aload_0
       5: aload_1
       6: putfield      #2               // Field valeur:Ljava/io/Serializable;
       9: return

  public T get();
    Code:
       0: aload_0
       1: getfield      #2               // Field valeur:Ljava/io/Serializable;
       4: areturn
}
```

5.11.2.3. Les méthodes pont (bridge method)

L'effacement de type peut parfois amener à une situation où deux méthodes doivent avoir la même signature avec des types de retour différents. Cela n'est pas possible dans le code source Java puisque la valeur de retour ne fait pas partie de la signature de la méthode. Dans ce cas, le compilateur va ajouter des méthodes pont (bridge method) dans le bytecode.

Exemple (code Java 5.0) :

```
public class MonConteneur<T> {

  public T valeur;

  public MonConteneur(T valeur) {
    this.valeur = valeur;
  }

  public T get() {
    return valeur;
  }

  public void set(T valeur) {
    this.valeur = valeur;
  }
}
```

Une classe fille hérite de la classe MonConteneur en la typant avec java.lang.String.

Exemple (code Java 5.0) :

```
public class MonConteneurChaine<T> extends MonConteneur<String> {

  public MonConteneurChaine(String valeur) {
    super(valeur);
  }

  public String get() {
    return valeur;
  }

  public void set(String valeur) {
    this.valeur = valeur;
  }
}
```

La décompilation du bytecode affiche deux méthodes supplémentaires dans la classe fille : deux méthodes pont.

Résultat :

```
C:\java>javap -c -s -v MonConteneurChaine
Classfile /C:/java/MonConteneurChaine.class
  Last modified 2 mai 2022; size 656 bytes
  MD5 checksum 5253b513a9a143d48352dce0bf35dd6b
  Compiled from "MonConteneurChaine.java"
public class MonConteneurChaine<T extends java.lang.Object>
  extends MonConteneur<java.lang.String>
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
...
{
  public MonConteneurChaine(java.lang.String);
    descriptor: (Ljava/lang/String;)V
    flags: ACC_PUBLIC
    Code:
      stack=2, locals=2, args_size=2
         0: aload_0
         1: aload_1
         2: invokespecial #1              // Method MonConteneur."<init>":(Ljava/lang/Object;)V
         5: return
      LineNumberTable:
        line 6: 0
        line 7: 5

  public java.lang.String get();
    descriptor: ()Ljava/lang/String;
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: getfield      #2              // Field valeur:Ljava/lang/Object;
         4: checkcast     #3              // class java/lang/String
         7: areturn
      LineNumberTable:
        line 10: 0

  public void set(java.lang.String);
    descriptor: (Ljava/lang/String;)V
    flags: ACC_PUBLIC
    Code:
      stack=2, locals=2, args_size=2
         0: aload_0
         1: aload_1
         2: putfield      #2              // Field valeur:Ljava/lang/Object;
         5: return
      LineNumberTable:
        line 14: 0
        line 15: 5

  public void set(java.lang.Object);
    descriptor: (Ljava/lang/Object;)V
    flags: ACC_PUBLIC, ACC_BRIDGE, ACC_SYNTHETIC
    Code:
      stack=2, locals=2, args_size=2
         0: aload_0
         1: aload_1
         2: checkcast     #3              // class java/lang/String
         5: invokevirtual #4              // Method set:(Ljava/lang/String;)V
         8: return
      LineNumberTable:
        line 3: 0

  public java.lang.Object get();
    descriptor: ()Ljava/lang/Object;
    flags: ACC_PUBLIC, ACC_BRIDGE, ACC_SYNTHETIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokevirtual #5              // Method get:()Ljava/lang/String;
         4: areturn
      LineNumberTable:
        line 3: 0
}
Signature: #18                      // <T:Ljava/lang/Object;>LMonConteneur<Ljava/lang/String;>;
SourceFile: "MonConteneurChaine.java"
```

A cause de l'effacement de type, il y a deux méthodes get() :

- une qui retourne une instance d'Object, qui est une méthode pont
- une qui retourne une instance de type String

La méthode set() présente aussi deux surcharges:

- une qui attend en paramètre une instance d'Object, qui est une méthode pont

- une qui attend en paramètre une instance de String

Remarque : la possibilité d'avoir deux surcharges identiques qui ne diffèrent que par leur type retour n'est pas permise dans le code source mais tout à fait possible dans le bytecode.

La classe fille ne peut pas être sûre que ses méthodes invoquées en utilisant la méthode héritée dont le type est effacé ou la méthode avec le type explicite. Les méthodes pont sont ajoutées pour assurer le bon traitement dans tous les cas notamment en effectuant un cast vers le type java.lang.String. Si ce cast échoue alors une exception de type ClassCastException est levée assurant ainsi la vérification que le type passé en paramètre est bien de type String.

Les méthodes pont sont aussi utilisées pour permettre à une classe étendant une classe générique ou implémentant une interface générique (avec un paramètre de type concret) d'être toujours utilisable comme un type brut.

Exemple (code Java 5.0) :

```
import java.util.Comparator;

public class MonComparator implements Comparator<Integer> {
  public int compare(Integer a, Integer b) {
    return b - a;
  }
}
```

Sans la méthode pont ajoutée, il ne serait pas possible d'utiliser la classe sous sa forme brute (raw type) donc sans préciser le type générique.

Résultat :

```
C:\java>javap -c -s -v MonComparator
Classfile /C:/java/MonComparator.class
  Last modified 6 mai 2022; size 567 bytes
  MD5 checksum 9f635c783594dae3d1fc497b7141d062
  Compiled from "MonComparator.java"
public class MonComparator extends java.lang.Object
  implements java.util.Comparator<java.lang.Integer>
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
...
{
  public MonComparator();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1          // Method java/lang/Object."<init>":()V
         4: return
      LineNumberTable:
        line 3: 0

  public int compare(java.lang.Integer, java.lang.Integer);
    descriptor: (Ljava/lang/Integer;Ljava/lang/Integer;)I
    flags: ACC_PUBLIC
    Code:
      stack=2, locals=3, args_size=3
         0: aload_2
         1: invokevirtual #2          // Method java/lang/Integer.intValue:()I
         4: aload_1
         5: invokevirtual #2          // Method java/lang/Integer.intValue:()I
         8: isub
         9: ireturn
      LineNumberTable:
        line 5: 0

  public int compare(java.lang.Object, java.lang.Object);
    descriptor: (Ljava/lang/Object;Ljava/lang/Object;)I
    flags: ACC_PUBLIC, ACC_BRIDGE, ACC_SYNTHETIC
    Code:
      stack=3, locals=3, args_size=3
         0: aload_0
         1: aload_1
         2: checkcast     #3      // class java/lang/Integer
         5: aload_2
         6: checkcast     #3      // class java/lang/Integer
         9: invokevirtual #4      // Method compare:(Ljava/lang/Integer;Ljava/lang/Integer;)I
        12: ireturn
      LineNumberTable:
        line 3: 0
}
Signature: #16                 // Ljava/lang/Object;Ljava/util/Comparator<Ljava/lang/Integer;>;
SourceFile: "MonComparator.java"
```

Le compilateur ajoute une méthode pont compare() qui attend en paramètre deux instances de type Object. L'implémentation de cette méthode pont effectue un cast des paramètres vers le type Integer et invoque la méthode définie dans le code source. Il est ainsi possible d'utiliser la classe MonComparator dans sa forme brute.

Exemple (code Java 5.0) :

```
  public static void main(String[] args) {
    Object val1 = 1;
    Object val2 = 3;

    Comparator comp = new MonComparator();
    int resultat = comp.compare(val1, val2);
  }
```

Les méthodes pont permettent aussi la bonne mise en oeuvre de l'effacement de type.

Exemple (code Java 5.0) :

```
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;

public class TestComparator {

  public static <T> T min(List<T> liste, Comparator<T> comp) {
    T resultat = null;
    if (liste != null && liste.size() > 0) {
      resultat = liste.get(0);
      for (T element : liste) {
        if (comp.compare(element, resultat) > 0) {
          resultat = element;
        }
      }
    }
    return resultat;
  }

  public static void main(String[] args) {
    List<Integer> liste = Arrays.asList(1, 2, 3, 4, 5);
    Integer plusPetit = min(liste, new MonComparator());
    System.out.println(plusPetit);
  }
}
```

Avec l'effacement de type, le type T dans le bytecode est remplacé par Object et des cast selon le type du contexte d'invocation de la méthode. Dans ce cas aussi, c'est la méthode pont, celle attendant en paramètre deux instances de type Object qui est invoquée.

5.12. La différence entre l'utilisation d'un argument de type et un wildcard

L'utilisation d'un paramètre de type permet d'imposer d'avoir le même type de paramètres à une méthode. Par exemple pour une méthode dont le but est de copier les éléments d'une liste source dans une liste destination

Exemple (code Java 5.0) :

```
public static <T extends Number> void copier(List<T> src, List<T> dest) {
  // ...
}
```

L'intérêt est de s'assurer que les éléments des deux listes seront les mêmes, vérifié par le compilateur et donc que la copie peut se faire sans soucis. Cela n'est pas possible en utilisant des wildcards.

Exemple (code Java 5.0) :

```
public static void copier(List<? extends Number> src, List<? extends Number> dest) {
  // ...
}
```

Lors de l'invocation de la méthode, il est possible de fournir des instances de types différents tels que List<Integer> et List<Double>

Il n'est aussi pas possible d'utiliser des bornes multiples avec un wildcard.

Exemple (code Java 5.0) :

```
import java.util.List;
import java.io.Serializable;

public class ListUtils {

  public static void afficher(List<? extends Number & Serializable> liste) {
    // ...
  }
}
```

Résultat :

```
C:\java>javac ListUtils.java
ListUtils.java:6: error: > expected
  public static void afficher(List<? extends Number & Serializable> liste) {
                                                    ^
ListUtils.java:6: error: ')' expected
  public static void afficher(List<? extends Number & Serializable> liste) {
                                                     ^
ListUtils.java:6: error: ';' expected
  public static void afficher(List<? extends Number & Serializable> liste) {
                                                                  ^
ListUtils.java:6: error: <identifier> expected
  public static void afficher(List<? extends Number & Serializable> liste) {
                                                                         ^
4 errors
```

Les wildcards peuvent avoir une borne supérieure ou inférieure.

Exemple (code Java 5.0) :

```
  public static void afficher(List<? super Integer> list) {
    // ...
  }
```

Un paramètre de type peut avoir une borne supérieure mais pas une borne inférieure. Le code ci-dessous ne compile pas.

Exemple (code Java 5.0) :

```
import java.util.List;

public class Utils {

  public static <T super Integer> void print(List>T< list) {
    // ...
  }
}
```

Résultat :

```
C:\java>javac Utils.java
Utils.java:5: error: > expected
  public static >T super Integer< void print(List<T> list) {
                  ^
Utils.java:5: error: illegal start of type
  public static <T super Integer> void print(List<T> list) {
                   ^
Utils.java:5: error: '(' expected
  public static <T super Integer> void print(List<T> list) {
                                ^
3 errors
```

5.13. Les conséquences et les effets de bord de l'effacement de type

L'effacement de type implique certains effets de bords ou contraintes dont il faut tenir compte sous réserve pour la plupart d'avoir des erreurs à la compilation.

5.13.1. Les types génériques et les casts

L'utilisation d'un cast avec un type générique différent provoque une erreur de compilation.

Exemple (code Java 5.0) :

```
import java.util.List;
import java.util.ArrayList;

public class MaClasse {

  public static void main(String... args) {
    List<Integer> li = new ArrayList<Integer>();
    List<Number>  ln = (List<Number>) li;  //  erreur du compilateur
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:9: error: incompatible types: List<Integer> cannot be converted to List<Number>
    List<Number>  ln = (List<Number>) li;  //  erreur du compilateur
                                      ^
1 error
```

Il existe une exception si le cast et la variable cible utilisent un wilcard non borné.

Exemple (code Java 5.0) :

```
import java.util.List;
import java.util.ArrayList;
```

```
public class MaClasse {

  public static void main(String... args) {
    List<Integer> li = new ArrayList<Integer>();
    List<?>  ln = (List<?>) li;  // ok
  }
}
```

Si le type générique est le même et que les classes sont compatibles alors le cast est aussi autorisé par le compilateur.

Exemple (code Java 5.0) :

```
import java.util.List;
import java.util.ArrayList;

public class MaClasse {

  public static void main(String... args) {
    List<String> ls = new ArrayList<String>();
    ArrayList<String> als = (ArrayList<String>) ls;  // OK
  }
}
```

5.13.2. Le type des instances génériques

Comme l'effacement de type empêche de conserver le type générique dans le bytecode, il n'est pas possible de distinguer le type de deux instances d'une même classe dont seul le type générique diffère dans le code source.

Cela a une conséquence, c'est que les classes de deux variables d'un même type avec des types génériques différents sont identiques et correspond au type brut.

Exemple (code Java 5.0) :

```
public class Conteneur<T> {

  private T valeur;

  public Conteneur(T valeur) {
    this.valeur = valeur;
  }

  public T get() {
    return valeur;
  }

  public static void main(String... args) {
    Conteneur<String> cs = new Conteneur<String>("test");
    Conteneur<Integer> ci = new Conteneur<Integer>(123);
    System.out.println(cs.getClass()==ci.getClass());
  }
}
```

Résultat :

```
C:\java>javac Conteneur.java

C:\java>java Conteneur
true
```

Exemple (code Java 5.0) :

```
import java.util.List;
import java.util.ArrayList;

public class MaClasse {
  public static void main(String[] args) {
    List<Integer> entiers = new ArrayList<Integer>();
    List<Double> doubles = new ArrayList<Double>();
    System.out.println(entiers.getClass() == doubles.getClass());
  }
}
```

Résultat :

```
true
```

5.13.3. Les génériques et les méthodes surchargées

Lors de l'invocation d'une méthode avec plusieurs surcharges à partir d'un type générique, le comportement peut paraitre surprenant comme dans l'exemple ci-dessous :

Exemple (code Java 5.0) :

```
public class MaClasse<T> {

  public static void main(String[] args) {
    MaClasse<String> maClasse = new MaClasse<>();
    maClasse.traiter("test");
  }

  String obtenirDonnees(Object s) {
    return "object";
  }

  String obtenirDonnees(String s) {
    return "string";
  }

  public void traiter(T t) {
    System.out.println(obtenirDonnees(t));
  }
}
```

Résultat :

```
object
```

C'est un effet de bord de l'effacement de type : comme dans le bytecode, les types génériques sont remplacés par le type Object, c'est la surcharge de type Object qui est invoquée à l'exécution.

5.13.4. Les génériques et l'opérateur instanceof

Le type utilisé avec l'opérateur instanceof doit être réifiable : cela implique que le type doit être connu à l'exécution, ce qui n'est pas le cas avec les types génériques à cause de l'effacement de type.

Il est tout à fait légal d'utiliser une instance d'un type paramétré à la gauche de l'opérateur instanceof.

Exemple (code Java 5.0) :

```
public class MaClasse<T> {

  public boolean tester(T o) {
    return o instanceof String;
  }

  public static void main(String... args) {
    MaClasse<String> mcString = new MaClasse<>();
    MaClasse<Integer> mcInteger = new MaClasse<>();

    System.out.println(mcString.tester("abc"));
    System.out.println(mcInteger.tester(123));
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java

C:\java>java MaClasse
true
false
```

Il n'est pas possible d'utiliser un type générique comme type dans un opérateur instanceof puisque l'effacement de type ne permet pas au runtime de connaître le type paramétré.

Exemple (code Java 5.0) :

```
public class MaClasse<T> {

  public boolean tester(Object o) {
    return o instanceof T;
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:4: error: illegal generic type for instanceof
    return o instanceof T;
                        ^
1 error
```

Il existe une exception qui concerne l'utilisation d'un wildcard non borné qui est considéré comme un type réifiable.

Exemple (code Java 5.0) :

```
import java.util.List;

public class MaClasse<T> {

  public boolean tester(Object o) {
    return o instanceof List<?>;
  }
}
```

Dans l'exemple ci-dessus, cela permet de vérifier que l'objet passé en paramètre est une List de quelque chose.

Il est possible de fournir en plus la classe du type générique utilisé et de vérifier que le type de l'objet et assignable à la classe du type générique.

Exemple (code Java 5.0) :

```
public class MaClasse<T> {

  private Class<T> t;

  public MaClasse(Class<T> t) {
    this.t = t;
  }

  public boolean tester(Object o) {
    return o != null && t.isAssignableFrom(o.getClass());
  }

  public static void main(String... args) {
    MaClasse<String> mcString = new MaClasse<>(String.class);
    MaClasse<Integer> mcInteger = new MaClasse<>(Integer.class);

    System.out.println(mcString.tester("abc"));
    System.out.println(mcInteger.tester(123));
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java

C:\java>java MaClasse
true
true
```

L'inconvénient de cette solution est de devoir fournir le type deux fois : dans le paramètre de type et en paramètre.

5.13.5. Les surcharges avec un type générique et un type Object

Le compilateur émet une erreur lorsque qu'une méthode possède deux surcharges, l'un avec un paramètre de type et l'autre qui attend un paramètre de type Object.

Exemple (code Java 5.0) :

```
public class MaClasseGenerique<T> {

  public void traiter(Object o) {
  }

  public void traiter(T t) {
  }
}
```

Résultat :

```
C:\java>javac MaClasseGenerique.java
MaClasseGenerique.java:6: error: name clash: traiter(T) and traiter(Object) have the same
 erasure
  public void traiter(T t) {
              ^
  where T is a type-variable:
    T extends Object declared in class MaClasse
1 error
```

5.13.6. La création d'une instance de type générique

Il n'est pas possible de créer une instance d'un type générique en utilisant l'opérateur new : à cause de l'effacement de type, le type paramétré n'est pas inclus dans le bytecode.

Le compilateur émet donc une erreur si on tente de créer une instance d'un paramètre de type avec l'opérateur new.

Exemple (code Java 5.0) :

```
public class MaClasseGenerique<T> {

  private T instance;

  public MaClasseGenerique() {
    this.instance = new T();
  }
}
```

Résultat :

```
C:\java>javac MaClasseGenerique.java
MaClasseGenerique.java:6: error: unexpected type
    this.instance = new T();
                        ^
  required: class
  found:    type parameter T
  where T is a type-variable:
    T extends Object declared in class MaClasse
1 error
```

Il n'est pas non plus possible de créer une instance en utilisant la méthode newInstance() sur la classe d'un type paramétré.

Exemple (code Java 5.0) :

```
public class MaClasseGenerique<T> {

  private T instance;

  public MaClasseGenerique() {
    this.instance = T.class.newInstance();
  }
}
```

Résultat :

```
C:\java>javac MaClasseGenerique.java
MaClasseGenerique.java:6: error: cannot select from a type variable
    this.instance = T.class.newInstance();
                     ^
1 error
```

Cette instanciation n'est pas possible car le type T n'est pas connu dans la déclaration de la classe générique.

Pour créer une instance, il faut utiliser l'API Reflection qui impose d'avoir une instance de type Class du type générique utilisé passée en paramètre.

Exemple (code Java 5.0) :

```
import java.util.Date;

public class MaClasseGenerique<T> {

  private T instance;

  public MaClasseGenerique(Class<T> classe) throws InstantiationException,
    IllegalAccessException {
    this.instance = classe.newInstance();
    System.out.println(instance);
  }

  public static void main(String[] args) throws Exception {
    MaClasseGenerique<Date> maClasse = new MaClasseGenerique<Date>(Date.class);
  }
}
```

Dans l'exemple ci-dessus, le constructeur par défaut est invoqué dynamiquement pour obtenir une instance.

5.13.7. La création d'une instance d'un tableau de type générique

Il n'est pas possible de créer une instance d'un tableau d'un type générique avec l'opérateur new. A l'exécution le type n'est pas connu et cela pourrait conduire à des erreurs.

Exemple (code Java 5.0) :

```
import java.lang.reflect.Array;

public class MaClasse<T> {

  private T[] tableau;

  public MaClasse()  {
    this.tableau = new T[10];
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:9: error: generic array creation
    this.tableau = new T[10];
                   ^
1 error
```

Comme la création d'une instance d'un type générique, la création d'un tableau de générique n'est pas possible car le type réel du générique ne sera connu qu'à l'exécution. Ainsi dans la déclaration de la classe, il n'est pas possible de créer une instance d'un type générique que le compilateur ne peut pas connaître.

Il est cependant possible de créer une instance dynamiquement en utilisant l'API Reflexion. Pour cela, il est nécessaire de passer en paramètre une instance de type Class du type générique pour créer une instance du tableau en invoquant la méthode newInstance() de la classe Array en lui passant en paramètre le type et le nombre d'éléments.

Exemple (code Java 5.0) :

```
import java.lang.reflect.Array;
import java.util.Date;

public class MaClasseGenerique<T> {

  private T[] tableau;

  public MaClasseGenerique (Class<T> classe) throws InstantiationException,
    IllegalAccessException  {
    this.tableau = (T[]) Array.newInstance(classe, 10);
    System.out.println(tableau);
  }

  public static void main(String[] args) throws Exception {
    MaClasseGenerique<Date> maClasse = new MaClasseGenerique<Date>(Date.class);
  }
}
```

Il est nécessaire de faire un cast car la méthode newInstance() renvoie un Object. Cette technique est utilisée dans la méthode toArray() des classes de l'API Collections.

5.13.8. Les varargs et les génériques

Le compilateur transforme un paramètre de type varargs en un tableau. Lorsque le varargs concerne un type paramétré, l'effacement de type remplace le type paramétré par Object. Il y a donc un risque de pollution de heap.

Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class MaClasse {

  public void traiter(List<String>... liste) {
    Object[] tableau = liste;
    tableau[0] = Arrays.asList(2, 4, 6);
    String s = liste[0].get(0);    // ClassCastException
  }

  public static void main(String[] args) {
    MaClasse instance = new MaClasse();
    instance.traiter(new ArrayList<String>());
  }
}
```

Résultat :

```
C:\java >javac -Xlint:unchecked MaClasseGenerique.java
MaClasseGenerique:7: warning: [unchecked] Possible heap pollution from parameterized
 vararg type List<String>
  public void traiter(List<String>... liste) {
                                      ^
MaClasseGenerique.java:15: warning: [unchecked] unchecked generic array creation for
 varargs parameter of type List<String>[]
    instance.traiter(new ArrayList<String>());
                    ^
2 warnings
```

Le compilateur affiche deux avertissements de type unchecked à la compilation sur un potentiel risque de pollution du heap lors de l'utilisation d'un tableau sur un type générique

Lors de la déclaration d'une méthode avec varargs qui a un ou des types paramétrés, et que l'on est sûr que le corps de la méthode ne lèvera pas de ClassCastException en raison d'une mauvaise gestion du paramètre de type varargs, il est possible de demander au compilateur de ne pas générer un avertissement que le compilateur génère pour ces types de méthodes avec varargs en utilisant l'une des options suivantes :

- pour une méthode qui ne peut pas être redéfinie (constructeurs, méthodes final, static et private (depuis Java 9)) : il faut utiliser l'annotation @SafeVarargs
- pour toute autre méthode : il faut utiliser l'annotation @SuppressWarnings({"unchecked", "varargs"}) sur la méthode

Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.List;

public class VarArgs<T>{
  public final void afficher(T... messages) {
    for (T message : messages) {
      System.out.println(message);
    }
  }

  public static void main(String[] args) {
    VarArgs<String> varargs = new VarArgs<String>();
    varargs.afficher("msg1","msg2","msg3");
  }
}
```

Résultat :

```
C:\java>javac -Xlint:unchecked VarArgs.java
VarArgs.java:6: warning: [unchecked] Possible heap pollution from parameterized vararg type T
```

```
  public final void afficher(T... messages) {
                                ^
  where T is a type-variable:
    T extends Object declared in class VarArgs
1 warning

C:\java>java VarArgs
msg1
msg2
msg3
```

Dans l'exemple ci-dessus, comme il n'y a pas de risque de pollution du heap, il est possible d'utiliser l'annotation @Safevarargs sur la méthode final pour demander au compilateur de ne pas afficher l'avertissement.

Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.List;

public class VarArgs<T>{
  @SafeVarargs
  public final void afficher(T... messages) {
    for (T message : messages) {
      System.out.println(message);
    }
  }

  public static void main(String[] args) {
    VarArgs<String> varargs = new VarArgs<String>();
    varargs.afficher("msg1","msg2","msg3");
  }
}
```

Résultat :

```
C:\java>javac VarArgs.java

C:\java>java VarArgs
msg1
msg2
msg3
```

5.13.9. Les collisions de méthodes liées à l'effacement de type

Il n'est pas possible de définir une surcharge d'une méthode qui attend en paramètre un générique alors qu'il existe déjà une autre surcharge qui corresponde à la méthode qui sera générée par le compilateur en application de l'effacement de type.

Ainsi, il n'est pas possible de définir deux surcharges d'une même méthode d'une classe générique T acceptant en paramètre pour la première un type T et pour la seconde un type Object.

Exemple (code Java 5.0) :

```
public class MaClasse<T> {

  public void traiter(Object valeur) {
  }

  public void traiter(T valeur) {
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:6: error: name clash: traiter(T) and traiter(Object) have the same erasure
```

```
  public void traiter(T valeur) {
              ^
  where T is a type-variable:
    T extends Object declared in class MaClasse
1 error
```

Ce n'est pas possible non plus si une surcharge est déjà héritée.

Exemple (code Java 5.0) :

```
public class MaClasse<T> {

  public boolean equals(T t) {
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:3:
error: name clash: equals(T) in MaClasse and equals(Object) in Object have the same erasure,
 yet neither overrides the other
  public  boolean equals(T t) {
                  ^
  where T is a type-variable:
    T extends Object declared in class MaClasse
1 error
```

Dans l'exemple ci-dessous la méthode equals(Object) est héritée de la classe Object et la même méthode est générée par le compilateur suite à l'application de l'effacement de type.

5.13.10. Le contournement lors de l'utilisation d'un type brut d'une classe générique

L'effacement de type peut parfois avoir des effets surprenants aux premiers abords. Exemple :

Exemple (code Java 5.0) :

```
import java.util.Arrays;
import java.util.List;

public class MonConteneur<T> {
  private T valeur;

  public List<Integer> getEntiers() {
    return Arrays.asList(1, 2, 3);
  }
}
```

Exemple (code Java 5.0) :

```
public class TestMonConteneur {

  public static void afficher(MonConteneur conteneur) {
    for (int entier : conteneur.getEntiers()) {
      System.out.println(entier);
    }
  }

  public static void main(String[] args) {
    MonConteneur mc = new MonConteneur();
    afficher(mc);
  }
}
```

Cette classe ne se compile pas à cause de l'effacement de type et de l'utilisation du type brut comme paramètre de la méthode.

Résultat :

```
C:\java>javac TestMonConteneur.java
TestMonConteneur.java:4: error: incompatible types: Object cannot be converted to int
    for (int entier : conteneur.getEntiers()) {
                                          ^
1 error
```

A la compilation, l'effacement de type remplace List<Integer> par son type brut List. Le type but utilisé dans le paramètre de la méthode ne fournit aucune information au compilateur sur le fait que la classe soit générique.

Une solution pour contourner ce problème est d'utiliser un paramètre de type avec wildcard comme type en paramètre de la méthode.

Exemple (code Java 5.0) :

```
public class TestMonConteneur {

  public static void afficher(MonConteneur<?> conteneur) {
    for (int entier : conteneur.getEntiers()) {
      System.out.println(entier);
    }
  }

  public static void main(String[] args) {
    MonConteneur mc = new MonConteneur();
    afficher(mc);
  }
}
```

Cette version de la classe se compile et s'exécute correctement.

Résultat :

```
C:\java\workspace_2021-09\Test\src>javac TestMonConteneur.java

C:\java\workspace_2021-09\Test\src>java TestMonConteneur
1
2
3
```

L'ajout du paramètre de type avec un wildcard, implique que le compilateur va utiliser le type de plus précis possible selon la borne utilisée et ajouter un cast vers ce type. Dans l'exemple, au lieu d'utiliser Object du type brut, un cast est ajouté vers Integer puisque dans le code c'est une List d'Integer qui est utilisée.

Résultat :

```
C:\java\workspace_2021-09\Test\src>javap -c -s TestMonConteneur.class
Compiled from "TestMonConteneur.java"
public class TestMonConteneur {
  public TestMonConteneur();
    descriptor: ()V
    Code:
       0: aload_0
       1: invokespecial #1                // Method java/lang/Object."<init>":()V
       4: return

  public static void afficher(MonConteneur<?>);
    descriptor: (LMonConteneur;)V
    Code:
       0: aload_0
       1: invokevirtual #2                // Method MonConteneur.getEntiers:()Ljava/util/List;
       4: invokeinterface #3,  1          // InterfaceMethod java/util/List.iterator:()Ljava/
util/Iterator;
       9: astore_1
      10: aload_1
      11: invokeinterface #4,  1          // InterfaceMethod java/util/Iterator.hasNext:()Z
      16: ifeq          42
      19: aload_1
      20: invokeinterface #5,  1          // InterfaceMethod java/util/Iterator.next:()Ljava/
lang/Object;
      25: checkcast     #6                // class java/lang/Integer
      28: invokevirtual #7                // Method java/lang/Integer.intValue:()I
      31: istore_2
      32: getstatic     #8                // Field java/lang/System.out:Ljava/io/PrintStream;
      35: iload_2
      36: invokevirtual #9                // Method java/io/PrintStream.println:(I)V
      39: goto          10
      42: return

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    Code:
       0: new           #10               // class MonConteneur
       3: dup
       4: invokespecial #11               // Method MonConteneur."<init>":()V
       7: astore_1
       8: aload_1
       9: invokestatic  #12               // Method afficher:(LMonConteneur;)V
      12: return
}
```

5.14. Les restrictions dans l'utilisation des génériques

Plusieurs restrictions s'appliquent lors de la mise en oeuvre des génériques et sont vérifiées par le compilateur.

5.14.1. L'implémentation plusieurs fois de la même interface générique

Il n'est pas possible d'implémenter plusieurs fois la même interface avec des types génériques différents ou identiques.

Exemple (code Java 5.0) :

```
public interface MonInterface<T> {
}
```

Exemple (code Java 5.0) :

```
public class MaClasse implements MonInterface<String>, MonInterface<String>  {
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:1: error: repeated interface
public class MaClasse implements MonInterface<String>, MonInterface<String>  {
                                                                   ^
1 error
```

5.14.2. Les génériques ne supportent pas le sous-typage

Les génériques ne proposent pas de support pour les sous-types car cela pose des problèmes pour assurer la sécurité de type. Ainsi, List<T> n'est pas considérée comme un sous-type de List<S> où S est un super-type de T.

Par exemple, même si Integer hérite de Number, il n'est pas possible d'assigner une List<Integer> à une List<Number>.

Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.List;

public class MaClasse {

  public MaClasse () {

    List<Number> valeurs = new ArrayList<Integer>();
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:8: error: incompatible types: ArrayList<Integer> cannot be converted
 to List<Number>
    List<Number> valeurs = new ArrayList<Integer>();
                           ^
1 error
```

De la même manière, il n'est pas possible d'affecter deux variables d'un même type avec des types génériques différents.

Exemple (code Java 5.0) :

```
import java.util.List;

public class MaClasse {

  public MaClasse() {
    List<Number> nombres;
    List<Integer> entiers;
    nombres = entiers;
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:8: error: incompatible types: List<Integer> cannot be converted to List<Number>
    nombres = entiers;
              ^
1 error
```

L'exemple ci-dessous permet de comprendre pourquoi ce n'est pas autorisé : si c'était autorisé cela permettrait d'ajouter des Double dans une List d'Integer.

Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.List;

public class MaClasse {

  public static void main(String[] args) {
    List<Integer> listeEntiers = new ArrayList<Integer>();
    listeEntiers.add(100);
    List<Number> listeNombres = listeEntiers; // erreur a la compilation
    listeNombres.add(3.14116);
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:9: error: incompatible types: List<Integer> cannot be converted to List<Number>
```

```
    List<Number> listeNombres = listeEntiers; // erreur a la compilation
                                ^
1 error
```

C'est pour éviter ce genre de soucis que le compilateur ne permet pas le support du sous-typage dans les génériques.

5.14.3. Les types génériques et les membres statiques

Le type générique est précisé à la création d'une instance. Cependant un membre static est lié à la classe et non à une instance. C'est la raison pour laquelle il n'est pas possible d'utiliser un type paramétré dans un membre statique.

Un champ static est géré au niveau classe car il est partagé par toutes les instances. Il n'est donc pas possible qu'il soit générique puisque le type générique est précisé à la création d'une instance et peut donc être différent.

Exemple (code Java 5.0) :

```
public class MaClasse<T> {

  private static T donnees;

  public static void traiter(T donnees) {
  }

  public static T obtenir() {
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:3: error: non-static type variable T cannot be referenced from a static context
  private static T donnees;
                 ^
MaClasse.java:5: error: non-static type variable T cannot be referenced from a static context
  public static void traiter(T donnees) {
                             ^
MaClasse.java:8: error: non-static type variable T cannot be referenced from a static context
  public static T obtenir() {
                ^
3 errors
```

5.14.4. Les génériques et les types primitifs

Les génériques ne fonctionnent qu'avec des classes à cause de l'effacement de type. Il n'est donc pas possible d'utiliser des types primitifs comme type générique. Il est impératif d'utiliser les classes wrapper correspondantes. Les valeurs peuvent être fournies ou obtenues en utilisant l'autoboxing ou l'unboxing.

Exemple (code Java 5.0) :

```
import java.util.List;

public class MaClasse {

  public MaClasse () {
    List<int> valeurs;
  }
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:6: error: unexpected type
                List<int> valeurs;
```

```
                     ^
  required: reference
  found:    int
1 error
```

Il est impératif d'utiliser les classes wrapper correspondantes. Les valeurs peuvent être fournies ou obtenues en utilisant l'autoboxing ou l'unboxing.

Exemple (code Java 5.0) :

```
import java.util.List;

public class MaClasse {

  public MaClasse () {
    List<Integer> valeurs;
  }
}
```

5.14.5. Une exception ne peut pas être générique

Le compilateur émet une erreur lors de la définition d'une exception, donc une classe qui hérite de Throwable, avec un type générique.

Exemple (code Java 5.0) :

```
public class MonException<T> extends Throwable {
}
```

Résultat :

```
C:\java>javac MonException.java
MonException.java:2: error: a generic class may not extend java.lang.Throwable
public class MonException<T> extends Throwable {
                                     ^
1 error
```

Il n'est pas possible non plus d'utiliser un type paramétré dans une clause catch : dans ce cas, le compilateur émet une erreur.

Exemple (code Java 5.0) :

```
public class MaClasseGenerique<T> {

  public void traiter() {
    try {
      // ...
   }  catch (T ex) {
   }
  }
}
```

Résultat :

```
C:\java>javac -Xlint:unchecked MaClasseGenerique.java
MaClasseGenerique.java:6: error: unexpected type
   }  catch (T ex) {
             ^
  required: class
  found:    type parameter T
  where T is a type-variable:
    T extends Object declared in class MaClasseGenerique
1 error
```

5.14.6. Une annotation ne peut pas être générique

La définition d'une interface d'annotation ne peut pas être générique.

Exemple (code Java 5.0) :

```
public @interface MonAnnotation<T> {
}
```

Résultat :

```
C:\java>javac MonAnnotation.java
MonAnnotation.java:1: error: annotation type MonAnnotation cannot be generic
public @interface MonAnnotation<T> {
                                ^
1 error
```

5.14.7. Une énumération ne peut pas être générique

La définition d'une énumération ne peut pas être générique.

Exemple (code Java 5.0) :

```
public enum MonEnumeration<T> {
}
```

Résultat :

```
C:\java>javac MonEnumeration.java
MonEnumeration.java:1: error: '{' expected
public enum MonEnumeration<T> {
                          ^
MonEnumeration.java:1: error: < expected
public enum MonEnumeration<T> {
                           ^
MonEnumeration.java:1: error: <identifier> expected
public enum MonEnumeration<T> {
                            ^
MonEnumeration.java:2: error: reached end of file while parsing
}
 ^
4 errors
```

6. Les chaînes de caractères

Chapitre 6

Niveau : 🟩⬜⬜⬜⬜⬜ Fondamental

Dans de nombreux langages, les chaînes de caractères sont représentées et manipulées sous la forme de tableau de caractères. Dans les langages plus récents comme Java, les chaînes de caractères sont encapsulées et manipulées dans des objets.

En Java, une chaîne de caractères est encapsulée de manière immuable dans une instance de type java.lang.String.

Comme la classe String est fréquemment utilisée, le langage Java propose quelques simplifications pour faciliter leur manipulation, notamment :

- la définition d'une instance sans utiliser l'opérateur new mais en utilisant directement la syntaxe littérale
- la possibilité de concaténer des chaînes en utilisant l'opérateur +

Exemple :

```
String texte = "Bonjour";
```

Même avec une syntaxe similaire à celle utilisée pour définir de variables primitives, les variables de type String sont des objets. Partout où des constantes chaînes de caractères figurent entre guillemets, le compilateur Java génère un objet de type String avec le contenu spécifié. Il est donc possible d'écrire :

Exemple :

```
   String texte = "Java Java Java".replace('a','o');
```

Il est impossible de modifier le contenu d'un objet de type String. Cependant, il est possible d'utiliser les méthodes de la classe String pour effectuer une modification qui va créer une nouvelle instance de type String encapsulant la nouvelle chaîne de caractères.

Il est donc important d'utiliser la chaîne retournée par ces méthodes. Il est par exemple possible d'affecter le résultat de ces méthodes à la variable pour qu'elle pointe sur la nouvelle chaîne de caractères contenant les modifications ou sur une autre variable pour pouvoir conserver une référence sur l'originale.

Exemple :

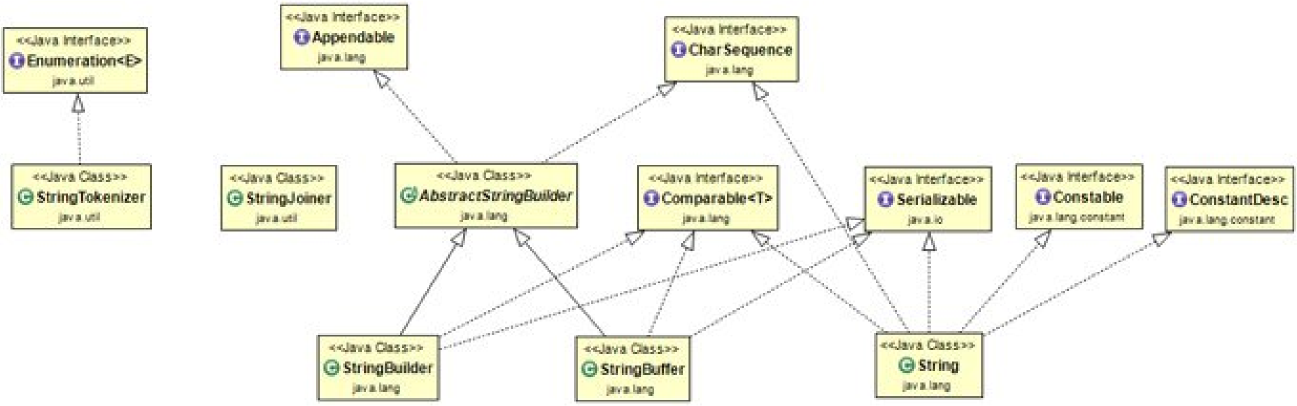
```
String texte = "Java Java Java";
texte.replace('a','o');
System.out.println(texte) ;
texte = texte.replace('a','o');
System.out.println(texte) ;
```

Résultat :

```
Java Java Java
Jovo Jovo Jovo
```

Plusieurs classes implémentant des interfaces sont proposées pour manipuler des chaînes de caractères :

- java.lang.String : encapsule une chaîne de caractères de manière immuable
- java.lang.StringBuffer : concaténer des chaînes de manière mutable (thread safe)
- java.lang.StringBuilder : concaténer des chaînes de manière mutable (non thread safe)
- java.util.StringTokenizer : séparation d'une chaîne selon un séparateur
- java.util.StringJoiner : concaténer des chaînes avec un séparateur et éventuellement un préfixe et un suffixe



Ce chapitre contient plusieurs sections :

- ♦ Les chaînes de caractères littérales
- ♦ La classe java.lang.String
- ♦ La création d'un objet de type String
- ♦ Les opérations sur les chaînes de caractères
- ♦ La conversion de et vers une chaîne de caractères
- ♦ La concaténation de chaînes de caractères
- ♦ La classe StringTokenizer
- ♦ Les chaînes de caractères et la sécurité
- ♦ Le stockage en mémoire
- ♦ Les blocs de texte (Text Blocks)

6.1. Les chaînes de caractères littérales

Historiquement, la syntaxe du langage Java propose des chaînes de caractères littérales mono-ligne.

Depuis Java 15, la syntaxe de Java propose les blocs de texte qui permettent de facilement exprimer des chaînes de caractères littérales multilignes.

6.1.1. Les chaînes de caractères littérales mono ligne

La syntaxe pour définir une chaîne de caractères littérale utilise comme séparateur des doubles quotes.

Il est possible de :

- définir une variable de type java.lang.String
- de l'initialiser en fournissant la chaîne directement après l'opérateur =
- fournir la valeur de la chaîne en paramètre d'un objet de type String

Exemple :

```
    String libelle = "";
    System.out.println("test");
```

6.1.2. Les séquences d'échappement

Dans une chaîne de caractères littérale, certains caractères doivent être échappés en utilisant une séquence d'échappement. Une séquence d'échappement débute par le caractère d'échappement qui est le caractère backslash \.

Exemple :

```
    System.out.println("\"Bonjour\"");
```

Evidemment, le caractère backslash doit lui-même être échappé pour être inclus dans une chaîne de caractères littérale.

Exemple :

```
    System.out.println("c:\\temp");
```

Dans une chaîne de caractères, plusieurs caractères particuliers doivent être utilisés avec le caractère d'échappement \. Le tableau ci-dessous recense les principaux caractères définis dans les spécifications Java :

Caractères spéciaux	Affichage	Unicode
\'	Apostrophe	
\"	Double quote	
\'	Quote	\u0027
\\	Backslash	\u005c
\t	Tabulation	\u0009
\b	Retour arrière (backspace BS)	\u0008
\r	Retour chariot (carriage return CR)	\u000d
\f	Saut de page (form feed FF)	\u000c
\n	Saut de ligne (line feed LF)	\u000a
\0ddd	Caractère ASCII ddd (octal)	
\xdd	Caractère ASCII dd (hexadécimal)	
\udddd	Caractère Unicode dddd (hexadécimal)	

Il est aussi possible d'utiliser des séquences d'échappement octale ou hexadécimale pour les caractères ASCII et Unicode.

Chaque caractère est associé à une valeur entière en fonction de l'encodage de caractère utilisé pour le représenter.

Exemple avec le caractère Euro :

Encoding	Valeur(s) hexadécimale(s)
ASCII	Impossible
ISO 8859-15	0xA4
CP 1252	0x80
UTF-8	0xE2 0x82 0xA0
UTF-16	0x20A0
UTF-32	0x000020A0

Pour les caractères ASCII, donc sur un seul octet, une séquence d'échappement permet de fournir la valeur du caractère en octal (0 à 377) ou en hexadécimal (0 à FF).

Il est possible de représenter un caractère en utilisant son code Unicode dans une séquence d'échappement dédiée. Cette séquence est composée de :

- \u
- suivi du code Unicode du caractère en hexadécimal sur 4 chiffres

Exemple :

```
    System.out.println("10 \u20AC");
```

Résultat :

```
10 €
```

Attention : les séquences d'échappement Unicode ne supportent que des valeurs comprises entre 0000 et FFFF (65635).

Il y a deux possibilités pour palier à cette limitation :

- fournir les deux valeurs hexadécimales du code en UTF-16
- depuis Java 1.5, utiliser la méthode toChars() de la classe Character qui attend en paramètre un entier correspond au code Unicode

Exemple (code Java 5.0) :

```
    int ch = 0x1F4A9; // caractère unicode emoji tas de crotte
    String s = new String(Character.toChars(ch));
    System.out.println(s);

    System.out.println("\uD83D\uDCA9");
```

6.2. La classe java.lang.String

Les chaînes de caractères sont encapsulées de manière immuable dans des instances de la classe java.lang.String.

6.2.1. Les constructeurs

La classe String possède plusieurs constructeurs :

Constructeur	Description
String()	Obtenir une nouvelle instance qui encapsule une chaîne vide
String(byte[] bytes)	Depuis Java 1.1, obtenir une nouvelle instance en décodant le tableau d'octets avec le charset par défaut du système
String(byte[] ascii, int hibyte)	Déprécié depuis Java 1.1 car elle ne convertit pas correctement les octets en caractères
String(byte[] bytes, int offset, int length)	Depuis Java 1.1, obtenir une nouvelle instance en décodant les octets du sous-tableau avec le charset par défaut de la plateforme
String(byte[] ascii, int hibyte, int offset, int count)	Déprécié depuis Java 1.1 car elle ne convertit pas correctement les octets en caractères

String(byte[] bytes, int offset, int length, String charsetName)	Depuis Java 1.1, obtenir une nouvelle instance en décodant le sous-tableau d'octets avec le charset dont le nom est précisé
String(byte[] bytes, int offset, int length, Charset charset)	Obtenir une nouvelle instance en décodant le sous-tableau d'octets avec le charset précisé
String(byte[] bytes, String charsetName)	Depuis Java 1.1, obtenir une nouvelle instance en décodant le tableau d'octets avec le charset dont le nom est précisé
String(byte[] bytes, Charset charset)	Depuis Java 1.6, obtenir une nouvelle instance en décodant le tableau d'octets avec le charset précisé

String(char[] value)	Obtenir une nouvelle instance encapsulant le tableau de caractères
	Exemple :
	`char chars[] = { 't', 'e', 's', 't' };`
	`String chaine = new String(chars);`

String(char[] value, int offset, int count)	Obtenir une nouvelle instance encapsulant le sous-tableau de caractères dont le premier caractère est indiqué avec le paramètre offset et le nombre de caractères à inclure est indiqué avec le paramètre count
	Exemple :
	`char chars[] = { 't', 'e', 's', 't' };`
	`String chaine = new String(chars,1,2);`

String(int[] codePoints, int offset, int count)	Depuis Java 1.5, obtenir une nouvelle instance en décodant le sous-tableau de codepoints Unicode avec le charset dont le nom est précisé
String(String original)	Obtenir une nouvelle instance encapsulant les caractères de la chaîne passée en paramètre (faire une copie de la chaîne) Exemple : `String chaine = new String("test");`
String(StringBuffer buffer)	Obtenir une nouvelle instance encapsulant les caractères contenus dans le StringBuffer
String(StringBuilder builder)	Depuis Java 1.5, obtenir une nouvelle instance encapsulant les caractères contenus dans le StringBuilder

6.2.2. Les méthodes

La classe String possède de nombreuses méthodes pour manipuler la chaîne de caractères qu'elle encapsule dont voici les principales :

Méthode	Rôle
char charAt(int index)	Renvoyer le caractère à la position fournie en paramètre
IntStream chars()	Depuis Java 9, renvoyer un IntStream des valeurs entières des caractères de la chaîne
int codePointAt(int index)	

	Depuis Java 1.5, renvoyer le code point Unicode du caractère à l'index précisé
int codePointBefore(int index)	Depuis Java 1.5, renvoyer le code point Unicode du caractère précédent l'index précisé
int codePointCount(int beginIndex, int endIndex)	Depuis Java 1.5, renvoyer le nombre de code point Unicode entre les index fournis en paramètre
IntStream codePoints()	Depuis Java 9, renvoyer un Stream des code points Unicode des caractères de la chaîne
int compareTo(String anotherString)	Comparer la chaîne avec celle fournie de manière lexicographique
int compareToIgnoreCase(String str)	Depuis Java 1.2, comparer la chaîne avec celle fournie de manière lexicographique sans tenir compte de la casse
String concat(String str)	Concaténer la chaîne avec celle fournie en paramètre
boolean contains(CharSequence s)	Depuis Java 1.5, renvoyer un booléen qui indique si la chaîne contient la séquence de caractères fournie en paramètre
boolean contentEquals(CharSequence cs)	Depuis Java 1.5, comparer la chaîne avec la séquence de caractères fournie en paramètre
boolean contentEquals(StringBuffer sb)	Depuis Java 1.4, comparer la chaîne avec celle contenue dans le StringBuilder fourni en paramètre
static String copyValueOf(char[] data)	Equivalent à valueOf(char[])
static String copyValueOf(char[] data, int offset, int count)	Equivalent à valueOf(char[], int, int)
boolean endsWith(String suffix)	Tester si la chaîne se termine par celle fournie en paramètre
boolean equals(Object anObject)	Comparer la chaîne avec celle fournie en paramètre en tenant compte de la casse
boolean equalsIgnoreCase(String anotherString)	Comparer la chaîne avec celle fournie en paramètre sans tenir compte de la casse
static String format(String format, Object... args)	Depuis Java 1.5, renvoyer une chaîne formatée en utilisant la chaîne de format et les arguments fournis
static String format(Locale l, String format, Object... args)	Depuis Java 1.5, renvoyer une chaîne formatée en utilisant la Locale, la chaîne de format et les arguments fournis
String formatted(Object... args)	Depuis Java 15, renvoyer une chaîne formatée en utilisant la chaîne comme format et les arguments fournis
byte[] getBytes()	Depuis Java 1.1, renvoyer un tableau des octets des caractères de la chaîne encodé avec le charset par défaut du système
void getBytes(int srcBegin, int srcEnd, byte[] dst, int dstBegin)	Dépréciée depuis Java 1.1 car elle ne convertit pas correctement les caractères en octets
byte[] getBytes(String charsetName)	Depuis Java 1.1, renvoyer un tableau des octets des caractères de la chaîne encodé avec le charset dont le nom est précisé
byte[] getBytes(Charset charset)	Depuis Java 1.6, renvoyer un tableau des octets des caractères de la chaîne encodé avec le charset précisé
void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)	Copier les caractères de la chaîne entre les positions précisées dans le tableau fourni en paramètre
int hashCode()	Renvoyer la valeur de hachage de la chaîne
String indent(int n)	Depuis Java 11, ajouter une indentation de n espaces précisés en paramètre et normaliser les caractères de terminaison
int indexOf(int ch)	Renvoyer l'index de la première occurrence du caractère fourni en paramètre dans la chaîne

int indexOf(int ch, int fromIndex)	Retourner l'index dans la chaîne de la première occurrence du caractère en commençant la recherche à partir de l'index précisé
int indexOf(String str)	Renvoyer la position de la première occurrence de la chaîne fournie en paramètre si l'instance la contient, sinon elle renvoie -1
int indexOf(String str, int fromIndex)	Retourner l'index dans la chaîne de la première occurrence de la chaîne fournie en commençant la recherche à partir de l'index précisé
String intern()	Retourner une représentation canonique de la chaîne. Elle vérifie si une instance de String est égale à une qui se trouve dans le pool. Si elle est présente, alors l'instance de String du pool est renvoyée. Sinon, l'instance de String est ajoutée au pool et la référence à cette instance est renvoyée. Par conséquent, pour deux chaînes de caractères s1 et s2, s1.intern() == s2.intern() est vrai si et seulement si s1.equals(s2) est vrai
boolean isBlank()	Depuis Java 11, renvoyer un booléen qui précise si la chaîne est vide ou ne contient que des caractères considérés comme des espaces
boolean isEmpty()	Depuis Java 1.6, renvoyer un booléen qui indique si la chaîne ne contient aucun caractère : donc true si length() renvoie 0
static String join(CharSequence delimiter, CharSequence... elements)	Depuis Java 1.8, renvoyer une chaîne composée de copies des éléments fournis concaténés avec le délimiteur
static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	Depuis Java 1.8, renvoyer une chaîne composée de copies des éléments fournis concaténés avec le délimiteur
int lastIndexOf(int ch)	Renvoyer l'index de la dernière occurrence dans la chaîne du caractère précisé
int lastIndexOf(int ch, int fromIndex)	Renvoyer l'index de la dernière occurrence dans la chaîne du caractère précisé, la recherche commençant en reculant à l'index fourni
int lastIndexOf(String str)	Renvoyer l'index de la dernière occurrence dans la chaîne de la chaîne précisée
int lastIndexOf(String str, int fromIndex)	Renvoyer l'index dans la chaîne de la dernière occurrence de la sous-chaîne spécifiée, la recherche commençant en reculant à l'index fourni
int length()	Renvoyer le nombre de caractères de la chaîne
Stream<String> lines()	Depuis Java 11, renvoyer un Stream dont les éléments sont les lignes de la chaîne
boolean matches(String regex)	Depuis Java 1.4, renvoyer un booléen qui précise si la chaîne respecte l'expression régulière fournie en paramètre
boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)	Tester si une portion de la chaîne est égale à celle d'une autre chaîne fournie en paramètre
boolean regionMatches(int toffset, String other, int ooffset, int len)	Tester si une portion de la chaîne est égale à celle d'une autre chaîne fournie en paramètre
String repeat(int count)	Depuis Java 11, renvoyer une chaîne qui est la concaténation d'elle-même le nombre de fois fournie en paramètre
String replace(char oldChar, char newChar)	Depuis Java 1.5, renvoyer une chaîne dont tous les caractères oldChar sont remplacés par newChar
String replace(CharSequence target, CharSequence replacement)	Renvoyer une chaîne dont laquelle toutes les sous-chaînes précisées dans target sont remplacées par celles précisées par replacement
String replaceAll(String regex, String replacement)	Depuis Java 1.4, renvoyer une chaîne dont toutes les portions qui respectent l'expression régulière sont remplacées par la chaîne fournie

String replaceFirst(String regex, String replacement)	Depuis Java 1.4, renvoyer une chaîne dans laquelle la première sous-chaîne qui correspond à l'expression régulière fournie est remplacée par la valeur fournie dans replacement
String[] split(String regex)	Depuis Java 1.4, découper la chaîne selon le séparateur fourni en paramètre sous la forme d'une expression régulière
String[] split(String regex, int limit)	Depuis Java 1.4, renvoyer un tableau de chaîne qui contient le découpage de la chaîne selon l'expression régulière fournie
boolean startsWith(String prefix)	Renvoyer un booléen qui indique si la chaîne commence par celle fournie en paramètre
boolean startsWith(String prefix, int toffset)	Tester si la sous-chaîne de cette chaîne commençant à l'index spécifié commence par le préfixe fourni
String strip()	Depuis Java 11, renvoyer une chaîne dont les espaces de début et de fin sont retirés
String stripIndent()	Depuis Java 15, renvoyer une chaîne dont les indentations accessoires en début et fin de lignes dans une chaîne de caractères sont retirées
String stripLeading()	Depuis Java 11, renvoyer une chaîne dont la valeur est cette chaîne, avec tous les espaces blancs en début de chaîne dont supprimés
String stripTrailing()	Depuis Java 11, renvoyer une chaîne dont la valeur est cette chaîne, avec tous les espaces blancs en fin de chaîne sont supprimés
CharSequence subSequence(int beginIndex, int endIndex)	Depuis Java 1.4, renvoyer une sous-chaîne correspondant aux index de début et de fin fournis
String substring(int beginIndex)	Renvoyer une chaîne qui contient les caractères de la chaîne à partir de la position fournie en paramètre
String substring(int beginIndex, int endIndex)	Renvoyer une chaîne qui contient les caractères de la chaîne à entre les positions de début et de fin fournies en paramètres
char[] toCharArray()	Renvoyer un tableau des caractères de la chaîne
String toLowerCase()	Renvoyer une nouvelle chaîne qui contient l'ensemble des caractères en minuscules
String toLowerCase(Locale locale)	Depuis Java 1.1, renvoyer une nouvelle chaîne qui contient l'ensemble des caractères en minuscules en tenant de la Locale fournie
String toString()	Renvoie l'instance elle-même
String toUpperCase()	Renvoyer une nouvelle chaîne qui contient l'ensemble des caractères en majuscules
String toUpperCase(Locale locale)	Depuis Java 1.1, renvoyer une nouvelle chaîne qui contient l'ensemble des caractères en majuscules en tenant de la Locale fournie
<R> R transform(Function<? super String,? extends R> f)	Depuis Java 12, renvoyer le résultat de l'application de la Function fournie en paramètre à la chaîne de caractères
String translateEscapes()	Depuis Java 15, renvoyer une chaîne dont la valeur est cette chaîne avec ses séquences d'échappement traduites comme dans une chaîne littérale
String trim()	Retourner une chaîne de caractères dont les caractères non significatifs ayant un code inférieur ou égal à "U+0020" de début et de fin sont retirés
static String valueOf(boolean b)	Retourner la représentation sous la forme d'une chaîne de caractères du booléen fourni en paramètre
static String valueOf(char c)	Retourner la représentation sous la forme d'une chaîne de caractères du caractère fourni en paramètre
static String valueOf(char[] data)	Retourner la représentation sous la forme d'une chaîne de caractères du tableau de caractères fourni en paramètre

static String valueOf(char[] data, int offset, int count)	Retourner la représentation sous la forme d'une chaîne de caractères du sous-tableau de caractères obtenu avec les paramètres fournis
static String valueOf(double d)	Renvoyer une représentation textuelle de la valeur double fournie en paramètre
static String valueOf(float f)	Renvoyer une représentation textuelle de la valeur float fournie en paramètre
static String valueOf(int i)	Renvoyer une représentation textuelle de la valeur entière fournie en paramètre
static String valueOf(long l)	Renvoyer une représentation textuelle de la valeur entière fournie en paramètre
static String valueOf(Object obj)	Renvoyer une représentation textuelle de l'objet passé en paramètre

Comme la classe String encapsule une chaîne de caractères de manière immutable, toutes les méthodes de modification retourne une nouvelle instance qui encapsule le résultat de la modification.

6.2.3. La classe String est immuable

Une chaîne de caractères en Java, encapsulée dans une instance de type java.lang.String est immutable : elle ne peut pas être modifiée. Le résultat d'une action de modification créé une nouvelle chaîne qui contient le résultat de la modification et laisse la chaîne initiale dans le même état.

Chaque traitement qui vise à transformer une instance de la classe est implémentée par une méthode qui laisse l'objet d'origine inchangé et renvoie une nouvelle instance de String contenant les modifications.

Exemple :

```
private String uneChaine;

void miseEnMajuscule(String chaine) {
  uneChaine = chaine.toUpperCase()
}
```

Il est ainsi possible d'enchaîner plusieurs méthodes :

Exemple :

```
uneChaine = chaine.toUpperCase().trim();
```

Les chaînes de caractères sont immuables pour améliorer la performance et la sécurité.

L'immutabilité des chaînes de caractères apporte plusieurs bénéfices :

- elles sont de fait thread-safe
- leur hashcode peut être mis en cache
- permet de réutiliser des chaînes identiques en utilisant un pool de Strings

6.2.4. L'encapsulation des caractères des chaînes de caractères

Les chaînes de caractères ne sont pas des tableaux que l'on peut manipuler directement car les caractères sont encapsulés dans la classe String : il faut utiliser les méthodes de la classe String sur une instance pour effectuer des manipulations.

Historiquement, Java ne fonctionne pas avec le jeu de caractères ASCII ou ANSI, mais avec Unicode (Universal Code). Ceci concerne les types char et les chaînes de caractères.

Jusqu'à Java 8, la classe String encapsule une chaîne de caractères en utilisant en interne un tableau de char, dans lequel chaque caractère est encodé en UTF-16. Cela implique que chaque caractère requiert deux octets même si ce caractère est dans la table ASCII.

A partir de Java 9, une nouvelle implémentation de la classe String est proposée sous le nom de Compact Strings. Selon les caractères contenus, les caractères sont stockés dans un tableau de char (UTF-16) ou un tableau de byte (Latin-1).

Si les caractères sont tous dans la table ASCII, ils pourront être stockés sur un seul octet. Ainsi, si tous les caractères sont dans la table ASCII, on peut réduire la taille en mémoire requise pour stocker les chaînes de caractères.

6.3. La création d'un objet de type String

La syntaxe du langage Java propose deux possibilités pour obtenir une référence vers un objet de type java.lang.String.

6.3.1. L'utilisation de la syntaxe littérale

Le plus simple pour définir une instance de type String est d'utiliser la syntaxe littérale.

Exemple :

```
String chaine = "Bonjour";
```

Cette syntaxe littérale permet de définir une instance de type String sans avoir à utiliser l'instruction new comme pour tous les autres types.

Il est aussi possible d'utiliser une expression qui n'utilise que la syntaxe littérale.

Exemple :

```
String chaine = "Bon"+"jour";
```

6.3.2. L'utilisation d'un constructeur

Il est aussi possible d'invoquer un des constructeurs de la classe String.

Exemple :

```
char[] chars = { 'B', 'o', 'n', 'j', 'o', 'u', 'r' };
String chaine = new String(chars);
```

6.3.3. Syntaxe littérale vs utilisation d'un constructeur

Si on utilise la syntaxe littérale, on obtient la référence d'une nouvelle instance si la chaîne n'est pas présente dans le pool et elle y est ajoutée, sinon on obtient la référence de chaîne présente dans le pool.

Exemple :

```
String chaine1 = "Bonjour";
String chaine2 = "Bonjour";
System.out.println(chaine1 == chaine2);
```

Résultat :

```
true
```

Lorsque que l'on créé une instance de type String en utilisant l'instructeur new, un nouvel objet est systématiquement créé et stocké dans le heap.

Exemple :

```
    String chaine1 = "Bonjour";
    String chaine2 = new String("Bonjour");
    System.out.println(chaine1 == chaine2);
```

Résultat :

```
false
```

La création d'une chaîne en utilisant un des constructeurs crée systématiquement une nouvelle instance.

L'utilisation de la syntaxe littérale pour créer une chaîne implique toujours que celle-ci soit gérée dans le pool de chaînes.

Il est donc recommandé d'utiliser la syntaxe littérale plutôt que l'opérateur new pour créer des instances de type String lorsque que l'on connaît déjà sa valeur.

6.4. Les opérations sur les chaînes de caractères

La classe String possède de nombreuses méthodes pour effectuer des traitements sur la chaîne qu'elle encapsule et/ou sur d'autres chaînes.

6.4.1. Le test de l'égalité de deux chaînes de caractères

Comme c'est le cas dans d'autres langages, il est tentant de tester l'égalité de deux chaînes avec l'opérateur ==.

Ce test est valide, il teste l'égalité sur les références des deux instances. De par le fonctionnement en interne dans la JVM et de la manière dont les chaînes sont définies, ce test peut renvoyer true ou false.

Il est préférable d'utiliser la méthode equals() pour tester l'égalité de la chaîne de caractères avec celle fournie en paramètre.

La classe String redéfinit la méthode equals() héritée de la classe Objet pour retourner true si le paramètre n'est pas null et si c'est un objet de type String qui encapsule une chaîne ayant la même séquence de caractères en tenant compte de la casse.

Exemple :

```
String texte1 = "texte 1";
String texte2 = "texte 2";
if ( texte1.equals(texte2) ) {
  // les deux chaînes sont égales
  // ...
}
```

Le test réalisé par la méthode equals() est sensible à la casse. Pour ne pas tenir compte de la casse, il faut utiliser la méthode equalsIgnoreCase().

La méthode equalsIgnoreCase() compare la chaîne avec celle fournie en paramètre en ne tenant pas compte de la casse. Elle renvoie true si les deux chaînes de caractères ont la même taille et contiennent la même séquence de caractères sans tenir compte de la casse.

Exemple :

```
String texte1 = "texte";
String texte2 = "TEXTE";
System.out.println(texte1.equals(texte2));
System.out.println(texte1.equalsIgnoreCase(texte2));
```

Résultat :

```
false
true
```

Les méthodes equals() et equalsIgnoreCase() permettent d'effectuer une vérification lexicographique de l'égalité de deux chaînes.

L'utilisation de l'opérateur == effectue une vérification sur les références. Il n'est pas recommandé d'utiliser l'opérateur == car celui-ci teste l'égalité des références des deux chaînes.

Exemple :

```
String texte1a = "texte 1";
String texte1b = "texte 1";
String texte2a = "texte 2";
String texte2b = new String("texte 2");
System.out.println(texte1a == texte1b);
System.out.println(texte1a == texte2a);
System.out.println(texte2a == texte2b);
```

Résultat :

```
true
false
false
```

A quelques rares exceptions près, la comparaison de deux chaînes de caractères doit se faire avec la méthode equals().

Il est aussi possible de comparer la chaîne de caractères avec une autre en utilisant les méthodes compareTo() et compareToIgnoreCase(). Celles-ci retournent 0 si les deux chaînes sont égales, une valeur négative si la chaîne est plus petite que celle fournie et une valeur positive si la chaîne est plus grande que celle fournie.

Exemple :

```
    String texte1 = "texte";
    String texte2 = "texte";
    String texte3 = "TEXTE";
    System.out.println(texte1.compareTo(texte2));
    System.out.println(texte1.compareToIgnoreCase(texte3));
```

Résultat :

```
0
0
```

6.4.1.1. equals() vs ==

La méthode equals() compare le contenu de la chaîne avec celle fournie en paramètre.

L'opérateur == teste l'égalité des références : si les deux variables de type String pointe sur la même référence alors l'opérateur renvoie true. Si les références sont différentes alors il renvoie false.

Exemple :

```
    String s0 = "Te";
    String s1 = "Test";                   // Chaîne littérale
    String s2 = "Te"+"st";                // Chaîne littérale
    String s3 = s1;                       // affectation de la référence
    String s4 = new String("Test");       // nouvelle instance
    String s5 = new String("Test");       // nouvelle instance
    String s6 = s0 + "st";                // Chaîne non littérale

    System.out.println(s1 == s1);         // true, même référence
    System.out.println(s1 == s2);         // true, même référence, celle du pool de String
    System.out.println(s1 == s3);         // true, même référence
    System.out.println(s1 == s4);         // false, références différentes
    System.out.println(s4 == s5);         // false, références différentes
    System.out.println(s1 == s6);         // false, références différentes

    System.out.println(s1.equals(s2));    // true, contenu identique
    System.out.println(s1.equals(s3));    // true, contenu identique
    System.out.println(s1.equals(s4));    // true, contenu identique
    System.out.println(s4.equals(s5));    // true, contenu identique
    System.out.println(s1.equals(s6));    // true, contenu identique
```

Résultat :

```
true
true
true
false
false
false
true
true
true
true
true
```

6.4.2. La comparaison de chaînes de caractère

Plusieurs méthodes de classe String peuvent être utilisées pour comparer tout ou partie d'une chaîne de caractères avec une autre.

La méthode compareTo() est une redéfinition de la méthode définie dans l'interface Comparable qui effectue une comparaison lexicographique des codepoints Unicode de chaque caractères avec ceux de la chaîne fournie en paramètre.

Elle renvoie une valeur entière qui peut être :

- un entier négatif si la chaîne précède lexicographiquement la chaîne passée en paramètre
- zéro si les deux chaînes sont égales
- un entier positif si la chaîne suit lexicographiquement la chaîne passée en paramètre

Dans l'ordre lexicographique, deux chaînes sont différentes si :

- leurs tailles sont différentes
- ou leurs caractères à au moins un index sont différents

Depuis Java 1.2, la méthode compareToIgnoreCase() effectue une comparaison lexicographique de chaque caractères avec ceux de la chaîne fournie en paramètre sans tenir compte de la casse.

Elle retourne un entier dont la valeur dépend du résultat de la comparaison de la chaîne avec celle fournie en paramètre :

- un entier négatif si la chaîne précède lexicographiquement la chaîne passée en paramètre
- zéro si les deux chaînes sont égales
- un entier positif si la chaîne suit lexicographiquement la chaîne passée en paramètre

Exemple (code Java 1.2) :

```
    String texte1 = "test";
    String texte2 = "tester";
    String texte3 = "Test";
    System.out.println(texte1.compareTo(texte2));
    System.out.println(texte1.compareTo(texte3));
    System.out.println(texte1.compareToIgnoreCase(texte3));
```

Résultat :

```
-2
32
0
```

Ces méthodes ne prennent pas en compte de Locale. La classe java.text.Collator permet de faire des comparaisons de chaînes en prenant en compte des spécificités liées à une Locale.

La classe Collator est abstraite : les sous-classes mettent en oeuvre des stratégies. La sous-classe RuleBasedCollator fournie par Java est utilisable avec de nombreux langages. D'autres sous-classes peuvent être créées pour répondre à des besoins plus spécifiques. La sous-classe RuleBasedCollator hérite de Collator et implémente l'interface Comparator.

Les surcharges de la méthode getInstance() permettent d'obtenir une instance de type Collator :

- getInstance() : sans paramètre, elle renvoie une instance pour la Locale par défaut
- getInstance(Locale) : elle renvoie une instance pour la Locale fournie en paramètre

L'exemple ci-dessous compare deux chaînes de caractères en utilisant un Collator pour la Locale par défaut.

Exemple :

```
    Collator collator = Collator.getInstance();
    System.out.println(collator.compare("test", "TEST"));
```

Résultat :

```
-1
```

L'exemple ci-dessous compare deux chaînes de caractères en utilisant un Collator pour la Locale US.

Exemple :

```
    Collator collator = Collator.getInstance(Locale.US);
    System.out.println(collator.compare("test", "TEST"));
```

Résultat :

```
-1
```

La propriété strength détermine le niveau minimum de différence considéré comme significatif lors de la comparaison. Quatre constantes sont utilisables : PRIMARY, SECONDARY, TERTIARY et IDENTICAL. Leur rôle dépend de la Locale et peut par exemple prendre en compte ou non la classe et les caractères accentués.

L'exemple ci-dessous compare deux chaînes de caractères en utilisant un Collator pour la Locale US sans tenir compte de la casse.

Exemple :

```
    Collator collator = Collator.getInstance(Locale.US);
    collator.setStrength(Collator.PRIMARY);
    System.out.println(collator.compare("test", "TEST"));
```

Résultat :

```
0
```

La méthode endsWith() de la classe String permet de tester si une chaîne se termine par le suffixe fourni en paramètre. Elle renvoie true si la chaîne se termine par le suffixe ou si le suffixe est une chaîne vide ou si le suffixe est égal à la chaîne.

Exemple :

```
    System.out.println("nom test".endsWith("test"));
    System.out.println("nom test".endsWith(""));
    System.out.println("nom test".endsWith("nom test"));
```

Résultat :

```
true
true
true
```

6.4.3. Le formatage d'une chaîne de caractères

Depuis java 1.5, la méthode statique format() de la classe String permet de formater une chaîne de caractères sur la base du format fourni en premier paramètre et des valeurs fournis dans un varargs en paramètres.

Exemple (code Java 5.0) :

```
    String nom = "JM";
    String salutation = String.format("Bonjour %s", nom);
    System.out.println(salutation);
```

Résultat :

```
Bonjour JM
```

Par défaut, la méthode format() utilise la Locale par défaut du système pour formater certaines données notamment numériques et temporelles.

Depuis Java 1.5, une surcharge de la méthode format() attend en premier paramètre la Locale à utiliser.

Exemple (code Java 5.0) :

```
    String format = "La distance est de %,f metres";
    String libelle = String.format(Locale.FRENCH, format, 12345.67);
    System.out.println(libelle);
    libelle = String.format(Locale.ENGLISH, format, 12345.67);
    System.out.println(libelle);
```

Résultat :

```
La distance est de 12345,670000 metres
La distance est de 12,345.670000 metres
```

S'il y a plus de données que de déterminants dans la chaîne de formatage alors les données supplémentaires sont ignorées.

Le comportement si une valeur est null dépend du format du déterminant concerné.

Les surcharges de la méthode format() lève une exception de type IllegalFormatException si la syntaxe de la chaîne indiquant le format n'est pas valide, si un déterminant n'est pas compatible avec la donnée correspondante ou s'il n'y a pas assez de données par rapport aux déterminants dans le format.

Elles lèvent une exception de type NullPointerException si la chaîne de formatage est null.

6.4.3.1. Le format de la chaîne de formatage

Le format est précisé sous la forme d'une chaîne de caractères qui peut contenir du texte brut et des spécificateurs de format.

Le format des spécificateurs de format pour des données générales, textuelles et numériques est de la forme :

%[argument_index$][flags][width][.precision]conversion

La partie optionnelle argument_index permet de préciser sous la forme d'une valeur entière l'index de l'argument dans le varargs fournis. Le premier argument est référencé par "1$".

La partie optionnelle flags est un ensemble de caractères pour préciser le format. Les flags utilisables sont dépendant de la conversion à réaliser.

La partie optionnelle width permet de préciser sous la forme d'une valeur entière le nombre minimum de caractères de la valeur.

La partie optionnelle precision permet de restreindre sous la forme d'une valeur entière le nombre de caractères de la valeur selon la conversion à réaliser.

La partie obligatoire conversion est un caractère qui permet de préciser comment l'argument sera formaté. Le caractère utilisé précise le type de données à convertir.

Le format des spécificateurs de format pour des données temporelles est de la forme :

%[argument_index$][flags][width]conversion

Les parties optionnelles argument_index, flags et width sont similaires à ceux du format précédent.

La partie obligatoire conversion est une séquence de 2 caractères. Le premier caractères est 't' ou 'T'. Le second caractère précise le format à utiliser.

Le format des spécificateurs de format qui ne correspondent pas aux arguments ont la syntaxe suivante :

%[flags][width]conversion

Les parties optionnelles flags et width sont similaires à ceux du format précédent.

La partie obligatoire conversion est une séquence d'un caractère qui précise le contenu à insérer.

6.4.3.2. Les conversions

Les conversions sont regroupées en plusieurs catégories :

- général : s'applique à n'importe quel type
- caractère : s'applique à des données textuelles qui représente un ou plusieurs caractères Unicode (char, Character, byte, Byte, short, Short et int ou Integer si leur valeur passée en paramètre à la méthode Character.isValidCodePoint() renvoie true)
- numérique : pour des données entières (byte, Byte, short, Short, int, Integer, long, Long et BigItneger) ou pour des données flottantes (float, Float, double, Double et BigDecimal)
- date/heure : pour des données temporelles (long, Long, Calendar, Date et TemporalAccessor)

- pourcent : le littéral '%' ('\u0025')

Sauf mention contraire, le résultat du formatage d'une valeur null est la chaîne de caractères « null ».

Les conversions sont précisées avec le tableau des caractères ci-dessous. Les caractères en majuscules permettent de demander la mise en majuscules en utilisant la Locale par défaut si aucune n'est précisée explicitement en paramètre de la méthode.

Si aucune locale explicite n'est précisée, que ce soit lors de la construction de l'instance ou en tant que paramètre lors de l'invocation de la méthode, alors la Locale par défaut est utilisée.

Conversion	Catégorie	Description
'b', 'B'	Général	Si l'argument est null, alors le résultat est false. Si l'argument est un boolean ou un Boolean alors c'est le résultat de la méthode String.valueOf(arg). Sinon le résultat est true
'h', 'H'	Général	Le résultat de l'invocation de la méthode Integer.toHexString(arg.hashCode())
's', 'S'	Général	Si l'argument implémente java.util.Formattable alors c'est le résultat de l'invocation de sa méthode formatTo() sinon le résultat est l'invocation de sa méthode toString()
'c', 'C'	Caractère	Un caractère Unicode
'd'	Entier	Un nombre entier décimal
'o'	Entier	Un nombre entier octal
'x', 'X'	Entier	Un nombre entier hexadécimal
'e', 'E'	Flottant	Un nombre décimal en notation scientifique
'f'	Flottant	Un nombre flottant décimal
'g', 'G'	Flottant	Un nombre flottant utilisant la notation scientifique ou le format décimal, en fonction de la précision et de la valeur après arrondie
'a', 'A'	Flottant	Un nombre hexadécimal à virgule flottante avec un exposant. Les arguments de type BigDecimal ne sont pas supportés
't', 'T'	Date/heure	Le préfixe pour la conversion de données temporelles
'%'	Pourcent	Le caractère '%' ('\u0025')
'n'	Séparateur de ligne	Le séparateur de ligne spécifique à la plate-forme

6.4.3.3. Les conversions de données temporelles

La conversion des données temporelles utilise le préfixe 't' and 'T' suivi d'un caractère qui précise la conversion à réaliser. Les conversions utilisables pour les heures sont précisées dans le tableau ci-dessous :

'H'	Heure du jour formatée sur 24 heures avec deux chiffres avec un zéro de tête si nécessaire : 00 - 23
'I'	Heure du jour formatée sur 12 heures avec deux chiffres avec un zéro de tête si nécessaire : 01 - 12
'k'	Heure du jour formatée sur 24 heures : 0 - 23
'l'	Heure du jour formatée sur 24 heures : 0 - 12
'M'	Minute du jour formatée sur deux chiffres avec un zéro de tête si nécessaire : 00 - 59
'S'	Seconde du jour formatée sur deux chiffres avec un zéro de tête si nécessaire : 00 - 60 (60 est une valeur spéciale requise pour supporter les secondes intercalaires)
'L'	Milliseconde du jour formatée sur trois chiffres avec des zéros de tête si nécessaire : 000 - 999
'N'	Nanoseconde du jour formatée sur neuf chiffres avec des zéros de tête si nécessaire : 000000000 - 999999999

'p'	Matin ou après-midi en minuscules spécifiques selon la Locale. Préfixer par 'T' pour être en majuscules
'z'	Offset du fuseau horaire par rapport à l'heure GMT dans le style défini par la RFC 822 qui tient compte de l'heure d'été/hiver. Pour des valeurs de types long, Long et Date, le fuseau horaire par défaut de la JVM
'Z'	Une représentation abrégée du fuseau horaire qui tient compte de l'heure d'été/hiver. Pour des valeurs de types long, Long et Date, le fuseau horaire par défaut de la JVM
's'	Secondes écoulées depuis l'Epoch (1 janvier 1970 00:00:00 UTC)
'Q'	Millisecondes écoulées depuis l'Epoch (1 janvier 1970 00:00:00 UTC)

Exemple (code Java 5.0) :

```
    Calendar heureDebut = new GregorianCalendar(1995, 0, 10, 8, 7, 6);
    Calendar heureFin = new GregorianCalendar(1995, 0, 10, 17, 7, 6);
    System.out.println(String.format("%tH", heureDebut));
    System.out.println(String.format("%tI", heureFin));
    System.out.println(String.format("%tk", heureDebut));
    System.out.println(String.format("%tl", heureFin));
    System.out.println(String.format("%tM", heureDebut));
    System.out.println(String.format("%tS", heureDebut));
    System.out.println(String.format("%tL", heureDebut));
    System.out.println(String.format("%tN", heureDebut));
    System.out.println(String.format("%tp", heureDebut));
    System.out.println(String.format("%Tp", heureDebut));
    System.out.println(String.format("%tp", heureFin));
    System.out.println(String.format("%Tp", heureFin));
    System.out.println(String.format("%tz", heureDebut));
    System.out.println(String.format("%tZ", heureDebut));
    System.out.println(String.format("%ts", heureDebut));
    System.out.println(String.format("%tQ", heureDebut));
```

Résultat :

```
08
05
8
5
07
06
000
000000000
am
AM
pm
PM
+0100
CET
789721626
789721626000
```

Les conversions utilisables pour les dates sont précisées dans le tableau ci-dessous :

'B'	Le nom long du mois selon la Locale : Exemple : "janvier", "février"
'b'	Le nom abrégé du mois selon la Locale Exemple : "janv.", "févr."
'h'	Identique à 'b'
'A'	Le nom long du jour de la semaine selon la Locale Exemple : "lundi", "mardi"
'a'	Le nom cours du mois selon la Locale

	Exemple : "lun.", "mar."
'C'	Année sur quatre chiffres divisée par 100, formatée en deux chiffres avec un zéro en tête si nécessaire Exemple : 00, 99
'Y'	Année, formatée en au moins quatre chiffres avec des zéros de tête si nécessaire, par exemple 0536 équivaut à 536 après Jésus Christ dans le calendrier grégorien
'y'	Deux derniers chiffres de l'année avec un zéro en tête si nécessaire Exemple : 00,99
'j'	Jour de l'année, formaté sur 3 chiffres avec des zéros en tête si nécessaire : 001 - 366
'm'	Mois de l'année, formaté sur 2 chiffres avec un zéro en tête si nécessaire : 01 - 12
'd'	Jour du mois, formaté sur 2 chiffres avec un zéro en tête si nécessaire : 01 - 31
'e'	Jour du mois, formaté sur 2 chiffres : 1 - 31

Exemple (code Java 5.0) :

```
    Calendar dateLimite = new GregorianCalendar(1995, 3, 7);
    System.out.println(String.format("%tB", dateLimite));
    System.out.println(String.format("%tb", dateLimite));
    System.out.println(String.format("%th", dateLimite));
    System.out.println(String.format("%tA", dateLimite));
    System.out.println(String.format("%ta", dateLimite));
    System.out.println(String.format("%tC", dateLimite));
    System.out.println(String.format("%ty", dateLimite));
    System.out.println(String.format("%tY", dateLimite));
    System.out.println(String.format("%tj", dateLimite));
    System.out.println(String.format("%tm", dateLimite));
    System.out.println(String.format("%td", dateLimite));
    System.out.println(String.format("%te", dateLimite));
```

Résultat :

```
avril
avr.
avr.
vendredi
ven.
19
95
1995
097
04
07
7
```

Les conversions utilisables pour les dates/heures sont précisées dans le tableau ci-dessous :

'R'	Heure formatée au format 24 heures "%tH:%tM"
'T'	Heure formatée au format 24 heures "%tH:%tM:%tS"
'r'	Heure formatée au format 12 heures "%tI:%tM:%tS %Tp"
'D'	Date formatée au format "%tm/%td/%ty"
'F'	Date formatée au format ISO-8601 "%tY-%tm-%td"
'c'	Date et heure formatée au format "%ta %tb %td %tT %tZ %tY" Exemple "dim. avr. 17 12:30:45 CEST 1988"

Tout caractère non explicitement défini comme suffixe de conversion pour date/heure est illégal et est réservé aux futures extensions.

Exemple (code Java 5.0) :

```
    Calendar dateHeureLimite = new GregorianCalendar(2015, 6, 11, 11, 20, 45);
    System.out.println(String.format("%tR", dateHeureLimite));
    System.out.println(String.format("%tT", dateHeureLimite));
    System.out.println(String.format("%tr", dateHeureLimite));
    System.out.println(String.format("%tD", dateHeureLimite));
    System.out.println(String.format("%tF", dateHeureLimite));
    System.out.println(String.format("%tc", dateHeureLimite));
```

Résultat :

```
11:20
11:20:45
11:20:45 AM
07/11/15
2015-07-11
sam. juil. 11 11:20:45 CEST 2015
```

6.4.3.4. Les flags

Plusieurs flags sont utilisables selon le type de la valeur fournie en argument :

Flag	Général	Caractère	Numérique	Flottant	Date/Heure	Description
'-'	Oui	Oui	Oui	Oui	Oui	Justification à gauche
'#'	Oui [1]	-	Oui [3]	Oui	-	Le résultat dépendant d'une conversion spécifique
'+'	-	-	Oui [4]	Oui	-	Toujours inclure le signe
' '	-	-	Oui [4]	Oui	-	Le résultat comprendra un espace en tête pour les valeurs positives
'0'	-	-	Oui	Oui	-	Complété avec des zéros
','	-	-	Oui [2]	Oui [5]	-	Inclure des séparateurs de groupement spécifiques à la Locale
'('	-	-	Oui [4]	Oui [5]	-	Entourés un nombre négatif par des parenthèses

[1] Selon l'implémentation de la méthode formatTo() de l'interface java.util.Formattable

[2] Uniquement pour les conversions de type 'd'

[3] Uniquement pour les conversions de type 'o', 'x' et 'X'

[4] Les conversions de type 'd', 'o', 'x' et 'X' pour BigInteger et les conversions de type 'd' pour byte, Byte, short, Short, int, Integer, long et Long

[5] Uniquement pour les conversions de type 'e', 'E', 'f', 'g', et 'G'

Exemple (code Java 5.0) :

```
    int montant = -1234;
    System.out.println(String.format("% d euros", montant));
    System.out.println(String.format("%(d euros", montant));
    System.out.println(String.format("%+d euros", montant));
    System.out.println(String.format("%+d euros", 1234));
    System.out.println(String.format(Locale.ENGLISH, "%,d euros", montant));
    System.out.println(String.format("%-10d euros", montant));
    System.out.println(String.format("%010d euros", montant));
```

Résultat :

```
-1234 euros
(1234) euros
-1234 euros
+1234 euros
-1,234 euros
```

Certains flags doivent obligatoirement être utilisés avec une taille.

Exemple (code Java 5.0) :

```
    int montant = -1234;
    System.out.println(String.format("%-10d euros", montant));
    System.out.println(String.format("%010d euros", montant));
```

Résultat :

```
-1234      euros
-000001234 euros
```

6.4.3.5. La taille (Width)

La partie width permet de préciser le nombre minimum de caractères à l'issue de la conversion.

Exemple (code Java 5.0) :

```
    int montant = -1234;
    System.out.println(String.format("Prix : %10d euros", montant));
```

Résultat :

```
Prix :      -1234 euros
```

6.4.3.6. La précision (Precision)

Pour les conversions de nombres flottants ('a', 'A', 'e', 'E' et 'f'), la précision définit le nombre de chiffres de la partie décimale. Si la conversion est 'g' ou 'G' alors la précision définit le nombre total de chiffres après l'arrondi.

La partie width permet de préciser le nombre minimum de caractères à l'issue de la conversion.

Exemple (code Java 5.0) :

```
    double prix = 1234.5678;
    System.out.println(String.format("%.2f euros", prix));
```

Résultat :

```
1234,57 euros
```

Pour les conversions de caractères, nombres entiers et dates/heures, la précision n'est pas utilisable : si c'est le cas alors une exception est levée.

Pour les autres types d'argument, la précision définit le nombre maximum de caractères à l'issu de la conversion.

6.4.3.7. Les index des arguments

L'index des arguments est un nombre entier précisant la position de l'argument dans la liste des arguments. Le premier argument est référencé par "1$", le second par "2$", ...

Il est aussi possible d'utiliser "<" ("\u003c") qui permet de faire référence à l'index du spécificateur de format précédent.

Exemple (code Java 5.0) :

```
    Calendar dateVal = new GregorianCalendar(2015, 6, 11);
    String libelle1 = String.format("Date de validité : %1$te/%1$tm/%1$tY", dateVal);
    System.out.println(libelle1);
    String libelle2 = String.format("Date de validité : %1te/%<tm/%<tY", dateVal);
    System.out.println(libelle2);
```

Résultat :

```
Date de validité : 11/07/2015
Date de validité : 11/07/2015
```

6.4.4. La conversion en minuscules ou en majuscules

Les méthodes toUpperCase() et toLowerCase() de la classe String permettent respectivement d'obtenir une nouvelle chaîne tout en majuscules ou tout en minuscules.

La méthode toLowerCase() permet de retourner une chaîne dont tous les caractères ont été transformés en minuscules.

Exemple :

```
    String message = "PRODUITS À 10 €";
    System.out.println(message.toLowerCase());
```

Résultat :

```
produits à 10 €
```

Sans paramètre, la méthode toUpperCase() utilise la Locale par défaut du système. Une surcharge de la méthode permet de préciser la Locale à utiliser.

Exemple :

```
    String message = "PRODUITS À 10 €";
    System.out.println(message.toLowerCase(Locale.ENGLISH));
```

La méthode toUpperCase() permet de retourner une chaîne dont tous les caractères ont été transformés en majuscules.

Exemple :

```
    String message = "Produits à 10 €";
    System.out.println(message.toUpperCase());
```

Résultat :

```
PRODUITS À 10 €
```

Sans paramètre, la méthode toUpperCase() utilise la Locale par défaut du système. Une surcharge de la méthode permet de préciser la Locale à utiliser.

Exemple :

```
    String message = "Produits à 10 €";
    System.out.println(message.toUpperCase(Locale.ENGLISH));
```

6.4.5. L'obtention d'une sous-chaîne

Les surcharges de la méthode substring() permettent d'obtenir une sous-chaîne.

La méthode substring(int, int) permet d'extraire la sous-chaîne composée des caractères compris entre l'index de début et celui de fin fournis en paramètres.

La surcharge substring(int) permet d'extraire la sous-chaîne composée des caractères compris entre l'index de début fourni en paramètre et celui de la fin de la chaîne.

Exemple :

```
    String message = "Produits à 10 €";
    System.out.println(message.substring(11));
    System.out.println(message.substring(0, 9));
```

Résultat :

```
10 €
Produits
```

6.4.6. L'obtention de la taille d'une chaîne

La méthode length() renvoie la taille de la chaîne de caractères.

Exemple :

```
    String message = "Produits à 10 €";
    System.out.println(message.length());
```

Résultat :

```
15
```

6.4.7. Le découpage d'une chaîne de caractères

La méthode split() de la classe String permet de couper une chaîne de caractères selon le séparateur fourni en paramètre. Elle renvoie un tableau de chaînes qui contient les sous-éléments.

Exemple :

```
    String[] noms = "nom1,nom2,nom3".split(",");
    System.out.println(noms[1]);
```

Résultat :

```
nom2
```

6.4.8. La vérification du contenu d'une chaîne

Depuis Java 1.5, la méthode contains() de la classe String renvoie un booléen qui indique si la chaîne contient la séquence fournie en paramètre.

Exemple :

```
    String chaine = "Le langage Java est orienté objet";
    boolean contientJava = chaine.contains("Java");
    System.out.println(contientJava);
```

Résultat :

```
true
```

6.4.9. L'obtention de la position d'un caractère ou d'une chaîne

Les surcharges de la méthode indexOf() permettent d'obtenir la première position de l'élément dans la chaîne de caractères. Celles qui attendent un int en second paramètre permet de préciser l'index à partir duquel la recherche doit s'effectuer. Elle retourne -1 si le caractère n'est pas trouvé.

La méthode indexOf(int) renvoie l'index de la première occurrence du caractère dans la chaîne dont le code Unicode est passé en paramètre.

Exemple :

```
    String chaine = "Le langage Java";
    int pos = chaine.indexOf(97);
    System.out.println(pos);
```

Résultat :

```
4
```

La surcharge indexOf(int, int) renvoie l'index de la première occurrence du caractère dans la chaîne à partir de la position fournie dont le code Unicode est passé en paramètre.

Exemple :

```
    String chaine = "Le langage Java";
    int pos = chaine.indexOf(97, 8);
    System.out.println(pos);
```

Résultat :

```
12
```

La surcharge indexOf(String) renvoie l'index de la première occurrence de la chaîne fournie en paramètre ou -1 si la chaîne n'est pas trouvée.

Exemple :

```
    String chaine = "Le langage Java";
    int pos = chaine.indexOf("a");
    System.out.println(pos);
```

Résultat :

```
4
```

La surcharge indexOf(String, int) renvoie l'index de la première occurrence de la chaîne à partir de l'index fournis en paramètres ou -1 si la chaîne n'est pas trouvée.

Exemple :

```
    String chaine = "Le langage Java";
    int pos = chaine.indexOf("a", 8);
    System.out.println(pos);
```

Résultat :

```
12
```

Les surcharges de la méthode lastIndexOf() permettent d'obtenir la dernière position de l'élément dans la chaîne de caractères. Celles qui attendent un int en second paramètre permet de préciser l'index à partir duquel la recherche doit s'effectuer. Elle retourne -1 si le caractère n'est pas trouvé.

La méthode lastIndexOf(int) renvoie l'index de la dernière occurrence du caractère dans la chaîne dont le code Unicode est passé en paramètre.

Exemple :

```
    String chaine = "Le langage Java";
    int pos = chaine.lastIndexOf(103); // caractère 'g'
    System.out.println(pos);
```

Résultat :

```
8
```

La surcharge lastIndexOf(int, int) renvoie l'index de la dernière occurrence du caractère dans la chaîne en reculant à partir de la position fournie dont le code Unicode est passé en paramètre.

Exemple :

```
    String chaine = "Le langage Java";
    int pos = chaine.lastIndexOf(101, 5); // caractère 'e'
    System.out.println(pos);
```

Résultat :

```
1
```

La surcharge lastIndexOf(String) renvoie l'index de la dernière occurrence de la chaîne fournie en paramètre ou -1 si la chaîne n'est pas trouvée.

Exemple :

```
    String chaine = "Le langage Java est un gage de qualité";
    int pos = chaine.lastIndexOf("gage");
    System.out.println(pos);
```

Résultat :

```
23
```

La surcharge lastIndexOf(String, int) renvoie l'index de la dernière occurrence de la chaîne à partir de l'index fourni en paramètre en reculant ou -1 si la chaîne n'est pas trouvée.

Exemple :

```
    String chaine = "Le langage Java est un gage de qualité";
    int pos = chaine.lastIndexOf("gage", 20);
    System.out.println(pos);
```

Résultat :

```
6
```

6.4.10. L'obtention d'un caractère de la chaîne

La méthode charAt() retourne le caractère se trouvant à l'index fourni en paramètre. L'index commence à la valeur 0. Si l'index fourni en paramètre n'est pas compris entre 0 et la taille de la chaîne - 1, alors une exception de type IndexOutOfBounsException est levée.

Exemple :

```
    String chaine = "Java";
    char car = chaine.charAt(2);
    System.out.println(car);
```

Résultat :

```
v
```

Depuis Java 1.5, la méthode codePointAt() retourne le codepoint Unicode du caractère se trouvant à l'index fourni en paramètre. Si l'index fourni en paramètre n'est pas compris entre 0 et la taille de la chaîne - 1, alors une exception de type IndexOutOfBounsException est levée.

Exemple (code Java 5.0) :

```
    String chaine = "10€ les 3";
    int codepoint = chaine.codePointAt(2);
    System.out.println(codepoint);
```

Résultat :

```
8364
```

Depuis Java 1.5, la méthode codePointBefore() retourne le codepoint Unicode du caractère se trouvant avant l'index fourni en paramètre. Si l'index fourni en paramètre n'est compris entre 1 et la taille de la chaîne, alors une exception de type IndexOutOfBounsException est levée.

Exemple (code Java 5.0) :

```
    String chaine = "10€ les 3";
    int codepoint = chaine.codePointBefore(3);
    System.out.println(codepoint);
```

Résultat :

```
8364
```

6.4.11. La suppression des espaces de début et de fin

La méthode trim() renvoie une chaîne de caractères dans laquelle les caractères dont le codepoint est inférieur ou égal à U+0020 en début ou en fin de chaîne sont retirés.

Si la chaîne est vide ou si les premiers et derniers caractères ne sont pas des espaces (comme défini ci-dessus), alors c'est la référence à la chaîne qui est renvoyée.

Si la chaîne ne contient que des caractères considérés comme des espaces, alors c'est une chaîne vide qui est retournée.

Exemple :

```
    String chaine = "  test  ";
    System.out.println("*" + chaine + "*");
    System.out.println("*" + chaine.trim() + "*");
    chaine = "\ttest\t";
    System.out.println("*" + chaine + "*");
    System.out.println("*" + chaine.trim() + "*");
```

Résultat :

```
*  test  *
*test*
*       test    *
*test*
```

Depuis Java 11, la méthode strip() renvoie une chaîne de caractères dans laquelle les caractères considérés comme des espaces en début ou en fin de chaîne sont retirés. Les caractères considérés comme des espaces sont différents de ceux considérés de la méthode trim() : ce sont les caractères dont le code point Unicode passé à la méthode Character.isWhitespace(int) renvoie true.

Si la chaîne est vide ou si tous les code points de la chaîne sont des caractères considérés comme des espaces, alors une chaîne vide est renvoyée. Sinon, il renvoie une sous-chaîne de la chaîne commençant par le premier code point qui n'est pas considéré comme un espace jusqu'au dernier code point qui n'est pas considéré comme un espace.

Exemple :

```
    String chaine = "  test  ";
    System.out.println("*" + chaine + "*");
    System.out.println("*" + chaine.strip() + "*");
    chaine = "\ttest\t";
    System.out.println("*" + chaine + "*");
    System.out.println("*" + chaine.strip() + "*");
    chaine = "\u00A0\u00A0test\u00A0\u00A0";
    System.out.println("*" + chaine + "*");
    System.out.println("*" + chaine.strip() + "*");
    System.out.println(Character.isWhitespace('\u00A0'));
```

Résultat :

```
*  test  *
*test*
*       test    *
*test*
*  test  *
*  test  *
false
```

Depuis Java 11, la méthode stripLeading() renvoie une chaîne de caractères dans laquelle les caractères considérés comme des espaces en début de chaîne sont retirés. Les caractères considérés comme des espaces sont les caractères dont le code point Unicode passé à la méthode Character.isWhitespace() renvoie true.

Si la chaîne est vide ou si tous les code points de la chaîne sont des caractères considérés comme des espaces, alors une chaîne vide est renvoyée. Sinon, il renvoie une sous-chaîne de la chaîne commençant par le premier code point qui n'est pas considéré comme un espace jusqu'au dernier caractère de la chaîne.

Exemple :

```
    String chaine = "  test  ";
    System.out.println("*" + chaine + "*");
    System.out.println("*" + chaine.stripLeading() + "*");
    chaine = "\ttest\t";
    System.out.println("*" + chaine + "*");
    System.out.println("*" + chaine.stripLeading() + "*");
```

Résultat :

```
*  test  *
*test  *
*       test    *
*test   *
```

Depuis Java 11, la méthode stripTrailing() renvoie une chaîne de caractères dans laquelle les caractères considérés comme des espaces en fin de chaîne sont retirés. Les caractères considérés comme des espaces sont les caractères dont le code point Unicode passé à la méthode Character.isWhitespace() renvoie true.

Si la chaîne est vide ou si tous les code points de la chaîne sont des caractères considérés comme des espaces, alors une chaîne vide est renvoyée. Sinon, il renvoie une sous-chaîne de la chaîne commençant par le caractère de la chaîne jusqu'au dernier caractère qui n'est pas considéré comme un espace.

Exemple :

```
    String chaine = "  test  ";
    System.out.println("*" + chaine + "*");
    System.out.println("*" + chaine.stripTrailing() + "*");
    chaine = "\ttest\t";
    System.out.println("*" + chaine + "*");
    System.out.println("*" + chaine.stripTrailing() + "*");
```

Résultat :

```
*  test  *
*  test*
*       test    *
*       test*
```

6.4.12. Tester si une chaîne est vide

Le plus simple est de tester l'égalité avec une chaîne vide en invoquant la méthode equals() sur une chaîne littérale vide.

Exemple :

```
    String chaine = "";
    if ("".equals(chaine)) {
      System.out.println("La chaine est vide");
    }
```

Résultat :

```
La chaine est vide
```

Le plus performant est de tester si la taille de la chaîne est égale à zéro.

Exemple :

```
    String chaine = "";
    if (chaine != null && chaine.length() == 0) {
      System.out.println("La chaine est vide");
    }
```

Résultat :

```
La chaine est vide
```

La méthode isEmpty() ajoutée dans Java SE 6 facilite le test d'une chaîne de caractères vide. Cette méthode utilise les données de l'instance de l'objet, il est donc nécessaire de vérifier que cette instance n'est pas null pour éviter la levée d'une exception de type NullPointerException.

Exemple (code Java 6) :

```
package fr.jmdoudoux.dej.java6;

public class TestEmptyString {

  public static void main(String args[]) {

    String chaine = null;
    try {
      if (chaine.isEmpty()){
        System.out.println("la chaine est vide");
      }
    } catch (NullPointerException e) {
      System.out.println("la chaine est null");
    }

    chaine = "test";
    if (chaine.isEmpty()){
      System.out.println("la chaine est vide");
    } else  {
      System.out.println("la chaine n'est pas vide");
    }

    chaine = "";
    if (chaine.isEmpty()){
      System.out.println("la chaine est vide");
    } else  {
      System.out.println("la chaine n'est pas vide");
    }
  }
}
```

Résultat :

```
la chaine est null
la chaine n'est pas vide
la chaine est vide
```

6.4.13. Tester si une chaîne est vide ou ne contient que des espaces

Depuis Java 11, la méthode isBlank() renvoie un booléen qui est true si la chaîne est vide ou ne contient que des code points Unicode qui sont considérés comme des espaces. Ce sont les caractères dont le code point Unicode passé à la méthode Character.isWhitespace() renvoie true.

Exemple (code Java 11) :

```
    String chaine = null;
    try {
      System.out.println(chaine.isBlank());
    } catch (NullPointerException e) {
      System.out.println("la chaine est null");
    }

    chaine = "test";
    System.out.println(chaine.isBlank());

    chaine = "";
    System.out.println(chaine.isBlank());

    chaine = "  ";
    System.out.println(chaine.isBlank());
```

Résultat :

```
la chaine est null
false
```

```
true
true
```

6.4.14. Le remplacement de caractères ou de sous-chaînes

Plusieurs méthodes permettent de remplacer des caractères ou des sous-chaînes dans la chaîne de caractères.

La méthode replace(char, char) remplace tous les caractères égaux à celui fourni dans le premier paramètre par celui fourni dans le second.

Exemple (code Java 5.0) :

```
    String chaine = "titi";
    String resultat = chaine.replace('i', 'o');
    System.out.println(resultat);
```

Résultat :

```
toto
```

Depuis Java 1.5, la surcharge replace(CharSequence, CharSequence) remplace toutes occurrences de la sous-chaîne fournie en paramètre par celle fournie dans le second.

Exemple (code Java 5.0) :

```
    String chaine = "titi";
    String resultat = chaine.replace("ti", "pa");
    System.out.println(resultat);
```

Résultat :

```
papa
```

La méthode replaceAll(String regex, String remplacement) remplace dans la chaîne toutes les sous-chaînes qui respectent l'expression régulière fournie par la chaîne fournie dans le second argument. Elle lève une exception de type PatternSyntaxException si la syntaxe de l'expression régulière n'est pas valide.

Exemple (code Java 1.4) :

```
    String chaine = "titi";
    String resultat = chaine.replaceAll("ti", "ta");
    System.out.println(resultat);
```

Résultat :

```
tata
```

La méthode replaceFirst(String regex, String remplacement) remplace dans la chaîne la première sous-chaîne qui respecte l'expression régulière fournie par la chaîne fournie dans le second argument. Elle lève une exception de type PatternSyntaxException si la syntaxe de l'expression régulière n'est pas valide.

Exemple (code Java 1.4) :

```
    String chaine = "titi";
    String resultat = chaine.replaceFirst("ti", "ta");
    System.out.println(resultat);
    resultat = chaine.replaceFirst("ti$", "ta");
    System.out.println(resultat);
```

Résultat :

```
tati
tita
```

6.5. La conversion de et vers une chaîne de caractères

Plusieurs méthodes de la classe String ou d'autres classes permettent de convertir de et vers une chaîne de caractères.

6.5.1. La conversion d'une chaîne en tableau d'octets

Depuis Java 1 .1, la méthode getBytes() de la classe String retourne un tableau d'octets encodés avec le jeu de caractères par défaut du système. Elle possède plusieurs surcharges.

Afin d'améliorer la portabilité, il est préférable d'utiliser une surcharge de la méthode getBytes() qui attend en paramètre le jeu de caractères à utiliser pour l'encodage. Soit depuis java 1.1 avec son nom sous la forme d'une chaîne de caractères soit depuis Java 1.6 avec une instance de la classe java.nio.charset.CharSet.

Exemple :

```
    String message = "Produits à 10 €";

    byte[] bytes = message.getBytes();
    System.out.println(Arrays.toString(bytes));

    bytes = message.getBytes(StandardCharsets.US_ASCII);
    System.out.println(Arrays.toString(bytes));

    bytes = message.getBytes("UTF-8");
    System.out.println(Arrays.toString(bytes));
```

Résultat :

```
[80, 114, 111, 100, 117, 105, 116, 115, 32, -32, 32, 49, 48, 32, -128]
[80, 114, 111, 100, 117, 105, 116, 115, 32, 63, 32, 49, 48, 32, 63]
[80, 114, 111, 100, 117, 105, 116, 115, 32, -61, -96, 32, 49, 48, 32, -30, -126, -84]
```

La surcharge getBytes(int, int, byte[], int) est dépréciée depuis Java 1.1 car elle ne convertit pas correctement les caractères en octets.

6.5.2. La conversion d'une chaîne en tableau de caractères

La méthode toCharArray() renvoie une tableau de caractères qui est une copie du tableau de caractères interne de l'instance de la chaîne.

Exemple :

```
    String chaine1 = "test";
    char[] caracteres = chaine1.toCharArray();
    System.out.println(caracteres[1]);
```

Résultat :

```
e
```

6.5.3. La conversion d'un type primitif en chaîne

Les différentes surcharges de la méthode statique valueOf() permettent de convertir la valeur fournie en paramètre en chaîne de caractères. Celles-ci attendent en paramètre une valeur primitive ou un objet.

Exemple :

```
    String chaine = String.valueOf(123);
    System.out.println(chaine);
    chaine = String.valueOf(123.45);
    System.out.println(chaine);
```

Résultat :

```
123
123.45
```

6.5.4. La conversion d'un objet en chaîne de caractères

Pour convertir un objet en une chaîne de caractères, et donc en obtenir une représentation textuelle de l'objet, il faut utiliser la méthode toString() sur l'instance concernée.

Chaque classe héritent de la méthode toString() définie dans la classe Object directement ou indirectement d'une des redéfinitions de ses classes mères. Une classe peut aussi redéfinir la méthode toString() selon ses besoins pour fournir une représentation textuelle de son état.

Exemple :

```
    Date maintenant = new Date();
    String libelle = aujourdhui.toString();
    System.out.println(libelle);
```

Résultat :

```
Sun Sep 16 15:41:01 CEST 2001
```

6.5.5. La conversion d'une chaîne en entier

La méthode parseInt() de la classe Integer permet de convertir une chaîne de caractères contenant une valeur numérique dans sa valeur entière.

Exemple :

```
    int taille = Integer.parseInt("175");
    system.out.println(taille);
```

Résultat :

```
175
```

La chaîne de caractères doit contenir des chiffres mais peut aussi commencer par un signe + (\u002B) ou - (\u002D)

Par défaut, la méthode parseInt() utilise la base 10. Une surcharge de la méthode attend en paramètre la base à utiliser.

Exemple :

```
    int taille = Integer.parseInt("20", 8);
    system.out.println(taille);
```

Résultat :

```
16
```

6.6. La concaténation de chaînes de caractères

Elle peut se faire de plusieurs manières, selon la version de Java utilisée et les besoins :

- l'opérateur +
- la méthode concat() de la classe String
- la classe StringBuffer
- la classe StringBuilder
- la classe StringJoiner
- la méthode join() de la classe String

6.6.1. La concaténation avec l'opérateur +

Le plus simple est d'utiliser un raccourci syntaxique proposé dans le langage qui repose sur l'utilisation de l'opérateur +. Java admet l'opérateur + comme opérateur pour la concaténation de chaînes de caractères.

Exemple :

```
    String chaine1 = "Hello";
    String chaine2 = "World";
    String chaine3 = chaine1 + " " + chaine2;
    System.out.println(chaine3);
```

Résultat :

```
Hello World
```

Il est possible de concaténer des chaînes avec des types primitifs en utilisant l'opérateur +.

Exemple :

```
    String libelle = "Produits à " + 10 + " euros";
    System.out.println(libelle);
```

Résultat :

```
Produits à 10 euros
```

L'opérateur + permet de concaténer plusieurs chaînes de caractères. Il est possible d'utiliser l'opérateur += pour concaténer une chaîne avec une autre et affecter la nouvelle chaîne à la variable.

Exemple :

```
    String texte = "";
    texte += "Hello";
    texte += " ";
    texte += "world";
    System.out.println(texte);
```

Résultat :

```
Hello world
```

Cet opérateur sert aussi à concaténer des chaînes avec tous les types de bases. La variable ou constante est alors convertie en chaîne et ajoutée à la précédente. La condition préalable est d'avoir au moins une chaîne dans l'expression sinon le signe '+' est évalué comme opérateur mathématique.

Exemple :

```
    System.out.println("La valeur de PI est : " + Math.PI);
    int duree = 121;
    System.out.println("Durée = " + duree);
```

Résultat :

```
La valeur de PI est : 3.141592653589793
Durée = 121
```

Attention à l'ordre d'évaluation des opérateurs + lorsque dans la même expression ils sont utilisés pour ajouter des valeurs numériques et pour concaténer des chaînes. Cet ordre peut être modifié en utilisant des parenthèses.

Exemple :

```
    String libelle = "Produits à " + 5 + 5 + " euros";
    System.out.println(libelle);

    libelle = "Produits à " + (5 + 5) + " euros";
    System.out.println(libelle);
```

Résultat :

```
Produits à 55 euros
Produits à 10 euros
```

Remarque : le compilateur va, selon la version de Java utilisée, mettre en oeuvre différentes techniques pour réaliser la concaténation :

- à partir de Java 7, la concaténation en dehors des boucles est faite avec un StringBuilder
- à partir de Java 9, la concaténation est effectuée par une fonctionnalité fournie par la JVM invoquée via InvokeDynamic

6.6.2. La concaténation avec la méthode concat()

La méthode concat() de la classe String renvoie une chaîne de caractères qui est la concaténation d'elle-même avec celle fournie en paramètre.

Exemple :

```
    String chaine1 = "Hello";
    String chaine2 = "World";
    String chaine3 = chaine1.concat(" ").concat(chaine2);
    System.out.println(chaine3);
```

Résultat :

```
Hello World
```

6.6.3. La classe StringBuffer

Les objets de cette classe permettent de manipuler un tampon de caractères pour construire dynamiquement une chaîne de caractères. La taille du tampon évolue donc dynamiquement en fonction des méthodes invoquées.

Un objet de type StringBuffer peut être utilisé pour construire ou modifier une chaîne de caractères chaque fois que l'utilisation de la classe String nécessiterait de nombreuses instanciations d'objets temporaires.

La classe StringBuffer propose plusieurs constructeurs :

Constructeur	Rôle
StringBuffer()	Retourner une instance dont le tampon est initialisé avec une taille de 16 caractères
StringBuffer(int capacity)	Retourner une instance dont la taille du tampon est fournie est paramètre
StringBuffer(CharSequence seq)	Retourner une instance dont le tampon est initialisé avec la CharSequence fournie en paramètre. Depuis Java 1.5
StringBuffer(String str)	Retourner une instance dont le tampon est initialisé avec la chaîne fournie en paramètre

La classe StringBuffer dispose de nombreuses méthodes qui permettent de modifier dynamiquement le tampon contenant les caractères :

Méthode	Rôle
StringBuffer append(boolean b)	Ajouter une représentation de la valeur booléenne fournie en paramètre
StringBuffer append(char c)	Ajouter le caractère fourni en paramètre
StringBuffer append(char[] str)	Ajouter le tableau de caractère fourni en paramètre
StringBuffer append(char[] str, int offset, int len)	Ajouter le sous-tableau fourni et précisé en paramètre
StringBuffer append(double d)	Ajouter une représentation de la valeur de type double fournie en paramètre
StringBuffer append(float f)	Ajouter une représentation de la valeur de type float fournie en paramètre
StringBuffer append(int i)	Ajouter une représentation de la valeur de type int fournie en paramètre
StringBuffer append(long lng)	Ajouter une représentation de la valeur de type long fournie en paramètre
StringBuffer append(CharSequence s)	Ajouter la CharSequence fournie en paramètre. Depuis Java 1.5
StringBuffer append(CharSequence s, int start, int end)	Ajouter la sous-séquence fournie et précisée en paramètre. Depuis Java 1.5
StringBuffer append(Object obj)	Ajouter la représentation textuelle de l'objet fourni en paramètre
StringBuffer append(String str)	Ajouter la chaîne fournie en paramètre
StringBuffer append(StringBuffer sb)	Ajouter le StringBuffer fourni en paramètre. Depuis Java 1.4
StringBuffer appendCodePoint(int codePoint)	Ajouter une représentation textuelle du codePoint Unicode fourni en paramètre. Depuis Java 1.5
int capacity()	Renvoyer la capacité courante du tampon
char charAt(int index)	Renvoyer le caractère dans le tampon à l'index fourni en paramètre
IntStream chars()	Renvoyer un IntStream dont les éléments sont les caractères sous la forme d'entier. Depuis Java 9
int codePointAt(int index)	Renvoyer le code point Unicode du caractère à l'index fourni dans le tampon. Depuis Java 1.5
int codePointBefore(int index)	Renvoyer le code point Unicode du caractère avant l'index fourni dans le tampon. Depuis Java 1.5
int codePointCount(int beginIndex, int endIndex)	Renvoyer le nombre de code points Unicode dans la place d'index. Depuis Java 1.5
IntStream codePoints()	

	Renvoyer un IntStream dont les éléments sont les code points Unicode . Depuis Java 9
int compareTo(StringBuffer another)	Comparer de manière lexicographique le tampon avec celui du StringBuffer fourni en paramètre. Depuis Java 11
StringBuffer delete(int start, int end)	Retirer la sous-chaîne dont la plage d'index est fournie en paramètre. Depuis Java 1.2
StringBuffer deleteCharAt(int index)	Retirer le caractère à l'index fourni en paramètre. Depuis Java 1.2
void ensureCapacity(int minimumCapacity)	Définir la capacité minimale du tampon
void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)	Copier les caractères de la plage précisés dans le tableau fourni en paramètre
int indexOf(String str)	Renvoyer l'index de la première occurrence de la chaîne fournie en paramètre
int indexOf(String str, int fromIndex)	Renvoyer l'index de la première occurrence de la sous-chaîne spécifiée, à partir de l'index spécifié
StringBuffer insert(int offset, boolean b)	Insérer une représentation de la valeur booléenne à la position précisée
StringBuffer insert(int offset, char c)	Insérer le caractère à la position précisée
StringBuffer insert(int offset, char[] str)	Insérer les caractères du tableau à la position précisée
StringBuffer insert(int index, char[] str, int offset, int len)	Insérer les caractères de la plage du tableau à la position précisée. Depuis Java 1.2
StringBuffer insert(int offset, double d)	Insérer une représentation de la valeur de type double à la position précisée
StringBuffer insert(int offset, float f)	Insérer une représentation de la valeur de type float à la position précisée
StringBuffer insert(int offset, int i)	Insérer une représentation de la valeur de type int à la position précisée
StringBuffer insert(int offset, long l)	Insérer une représentation de la valeur de type long à la position précisée
StringBuffer insert(int dstOffset, CharSequence s)	Insérer la CharSequence à la position précisée. Depuis Java 1.5
StringBuffer insert(int dstOffset, CharSequence s, int start, int end)	Insérer les caractères de la plage de CharSequence à la position précisée. Depuis Java 1.5
StringBuffer insert(int offset, Object obj)	Insérer la représentation textuelle de l'objet fourni à la position précisée
StringBuffer insert(int offset, String str)	Insérer la chaîne fournie à la position fournie en paramètres
int lastIndexOf(String str)	Renvoyer l'index de la dernière position de la chaîne fournie en paramètre. Depuis Java 1.4
int lastIndexOf(String str, int fromIndex)	Renvoyer l'index de la dernière occurrence de la sous-chaîne spécifiée, en recherchant à rebours à partir de l'index spécifié. Depuis Java 1.4
StringBuffer replace(int start, int end, String str)	Remplacer la séquence de caractères indiquée par la plage d'index par la chaîne fournie en paramètre. Depuis Java 1.2
StringBuffer reverse()	Inverser l'ordre des caractères du tampon. Depuis Java 1.0.2
void setCharAt(int index, char ch)	Remplacer le caractère à l'index par celui fourni en paramètres
void setLength(int newLength)	Préciser la taille de la chaîne de caractères. Si elle est plus petite que la chaîne actuelle, alors elle est tronquée. Si elle est plus grande, alors elle est complétée avec des caractères \u0000
CharSequence subSequence(int start, int end)	Renvoyer une CharSequence qui encapsule la sous-chaîne des éléments dont la plage d'index est fournie en paramètre. Depuis Java 1.4
String substring(int start)	Renvoyer une sous-chaîne qui contient les caractères à partir de l'index fourni. Depuis Java 1.2

String substring(int start, int end)	Renvoyer une sous-chaîne qui contient les caractères contenus dans la plage d'index fournis. Depuis Java 1.2
void trimToSize()	Tenter de réduire la taille du tampon pour l'aligner sur la taille de la séquence de caractères. Depuis Java 1.5

Exemple :

```
public class TestStringBuffer {

  static final String lettreMin = "abcdefghijklmnopqrstuvwxyz";
  static final String lettreMaj = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

  public static void main(java.lang.String[] args) {
    System.out.println(mettreEnMaj("chaine avec MAJ et des min"));
  }

  public static String mettreEnMaj(String s) {
    StringBuffer sb = new StringBuffer(s);

    for (int i = 0; i < sb.length(); i++) {
      int index = lettreMin.indexOf(sb.charAt(i));
      if (index >= 0)
        sb.setCharAt(i, lettreMaj.charAt(index));
    }
    return sb.toString();
  }
}
```

Résultat :

```
CHAINE AVEC MAJ ET DES MIN
```

6.6.4. La classe StringBuilder

Depuis Java 1.5, la classe java.lang.StringBuilder permet de manipuler un tampon de caractères pour construire dynamiquement une chaîne de caractères. La taille du tampon évolue donc dynamiquement en fonction des méthodes invoquées.

Un objet de type StringBuilder peut être utilisé pour construire ou modifier une chaîne de caractères chaque fois que l'utilisation de la classe String nécessiterait de nombreuses instanciations d'objets temporaires.

La classe StringBuilder propose plusieurs constructeurs :

Constructeur	Rôle
StringBuilder()	Retourner une instance dont le tampon est initialisé avec une taille de 16 caractères
StringBuilder(int capacity)	Retourner une instance dont la taille du tampon est fournie est paramètre
StringBuilder(CharSequence seq)	Retourner une instance dont le tampon est initialisé avec la CharSequence fournie en paramètre
StringBuilder(String str)	Retourner une instance dont le tampon est initialisé avec la chaîne fournie en paramètre

La classe StringBuilder dispose de nombreuses méthodes qui permettent de modifier dynamiquement le tampon contenant les caractères :

Méthode	Rôle
StringBuffer append(boolean b)	Ajouter une représentation de la valeur booléenne fournie en paramètre
StringBuffer append(char c)	Ajouter le caractère fourni en paramètre

StringBuffer append(char[] str)	Ajouter le tableau de caractère fourni en paramètre
StringBuffer append(char[] str, int offset, int len)	Ajouter le sous-tableau fourni et précisé en paramètre
StringBuffer append(double d)	Ajouter une représentation de la valeur de type double fournie en paramètre
StringBuffer append(float f)	Ajouter une représentation de la valeur de type float fournie en paramètre
StringBuffer append(int i)	Ajouter une représentation de la valeur de type int fournie en paramètre
StringBuffer append(long lng)	Ajouter une représentation de la valeur de type long fournie en paramètre
StringBuffer append(CharSequence s)	Ajouter la CharSequence fournie en paramètre
StringBuffer append(CharSequence s, int start, int end)	Ajouter la sous-séquence fournie et précisée en paramètre
StringBuffer append(Object obj)	Ajouter la représentation textuelle de l'objet fourni en paramètre
StringBuffer append(String str)	Ajouter la chaîne fournie en paramètre
StringBuffer append(StringBuffer sb)	Ajouter le StringBuffer fourni en paramètre
StringBuffer appendCodePoint(int codePoint)	Ajouter une représentation textuelle du codePoint Unicode fourni en paramètre
int capacity()	Renvoyer la capacité courante du tampon
char charAt(int index)	Renvoyer le caractère dans le tampon à l'index fourni en paramètre
IntStream chars()	Renvoyer un IntStream dont les éléments sont les caractères sous la forme d'entier. Depuis Java 9
int codePointAt(int index)	Renvoyer le code point Unicode du caractère à l'index fourni dans le tampon
int codePointBefore(int index)	Renvoyer le code point Unicode du caractère avant l'index fourni dans le tampon
int codePointCount(int beginIndex, int endIndex)	Renvoyer le nombre de code points Unicode dans la place d'index
IntStream codePoints()	Renvoyer un IntStream dont les éléments sont les code points Unicode . Depuis Java 9
int compareTo(StringBuffer another)	Comparer de manière lexicographique le tampon avec celui du StringBuffer fourni en paramètre. Depuis Java 11
StringBuffer delete(int start, int end)	Retirer la sous-chaînes dont la plage d'index est fournie en paramètre
StringBuffer deleteCharAt(int index)	Retirer le caractère à l'index fourni en paramètre
void ensureCapacity(int minimumCapacity)	Définir la capacité minimale du tampon
void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)	Copier les caractères de la plage précisés dans le tableau fourni en paramètre
int indexOf(String str)	Renvoyer l'index de la première occurrence de la chaîne fournie en paramètre
int indexOf(String str, int fromIndex)	Renvoyer l'index de la première occurrence de la sous-chaîne spécifiée, à partir de l'index spécifié
StringBuffer insert(int offset, boolean b)	Insérer une représentation de la valeur booléenne à la position précisée
StringBuffer insert(int offset, char c)	Insérer le caractère à la position précisée
StringBuffer insert(int offset, char[] str)	Insérer les caractères du tableau à la position précisée
StringBuffer insert(int index, char[] str, int offset, int len)	Insérer les caractères de la plage du tableau à la position précisée
StringBuffer insert(int offset, double d)	Insérer une représentation de la valeur de type double à la position précisée

StringBuffer insert(int offset, float f)	Insérer une représentation de la valeur de type float à la position précisée
StringBuffer insert(int offset, int i)	Insérer une représentation de la valeur de type int à la position précisée
StringBuffer insert(int offset, long l)	Insérer une représentation de la valeur de type long à la position précisée
StringBuffer insert(int dstOffset, CharSequence s)	Insérer la CharSequence à la position précisée
StringBuffer insert(int dstOffset, CharSequence s, int start, int end)	Insérer les caractères de la plage de CharSequence à la position précisée
StringBuffer insert(int offset, Object obj)	Insérer la représentation textuelle de l'objet fourni à la position précisée
StringBuffer insert(int offset, String str)	Insérer la chaîne fournie à la position fournie en paramètres
int lastIndexOf(String str)	Renvoyer l'index de la dernière position de la chaîne fournie en paramètre
int lastIndexOf(String str, int fromIndex)	Renvoyer l'index de la dernière occurrence de la sous-chaîne spécifiée, en recherchant à rebours à partir de l'index spécifié
StringBuffer replace(int start, int end, String str)	Remplacer la séquence de caractères indiquée par la plage d'index par la chaîne fournie en paramètre
StringBuffer reverse()	Inverser l'ordre des caractères du tampon
void setCharAt(int index, char ch)	Remplacer le caractère à l'index par celui fourni en paramètres
void setLength(int newLength)	Préciser la taille de la chaîne de caractères. Si elle est plus petite que la chaîne actuelle, alors elle est tronquée. Si elle est plus grande, alors elle est complétée avec des caractères \u0000
CharSequence subSequence(int start, int end)	Renvoyer une CharSequence qui encapsule la sous-chaîne des éléments dont la plage d'index est fournie en paramètre
String substring(int start)	Renvoyer une sous-chaîne qui contient les caractères à partir de l'index fourni
String substring(int start, int end)	Renvoyer une sous-chaîne qui contient les caractères contenus dans la plage d'index fournis
void trimToSize()	Tenter de réduire la taille du tampon pour l'aligner sur la taille de la séquence de caractères

6.6.5. Différence entre StringBuilder et StringBuffer

Les classes StringBuffer et StringBuilder ont des rôles et des méthodes similaires. Leur différence est dans leur contexte d'utilisation :

- StringBuffer : les méthodes sont synchronisées, ce qui permet une utilisation dans un contexte multi-thread
- StringBuilder : à n'utiliser que dans un contexte mono-thread (exemple : comme variable locale dans une méthode). Son utilisation dans un contexte multi-thread induit un risque de résultat erroné

Ainsi l'utilisation d'une instance de StringBuffer dans un contexte mono-thread n'a aucun impact sur les résultats mais uniquement sur les performances qui seront dans ce cas légèrement dégradée.

Par contre, l'utilisation d'une instance de StringBuilder dans un contexte multi-thread pourra induire des résultats erronés. Dans ce contexte, il est important d'utiliser un StringBuffer pour s'éviter des ennuis.

6.6.6. La classe StringJoiner

La classe java.util.StringJoiner, ajoutée en Java 8, permet de concaténer des chaînes de caractères avec un séparateur et éventuellement un préfixe et un suffixe.

La classe StringJoiner implémente le design pattern builder. Les différentes chaînes à ajouter sont précisées en les passant chacune en paramètre de la méthode add(). La méthode add() renvoie l'instance elle-même, ce qui permet de chaîner l'appel de plusieurs méthodes.

La méthode toString() permet d'obtenir le résultat de la concaténation.

Elle possède deux constructeurs.

Le premier attend en paramètre le séparateur à utiliser entre chaque chaîne.

Exemple (code Java 8) :

```
    StringJoiner joiner = new StringJoiner(",");
    joiner.add("1")
          .add("2")
          .add("3");
    System.out.println(joiner.toString());
```

Résultat :

```
1,2,3
```

Le second constructeur attend trois paramètres :

- le séparateur à utiliser entre les chaînes
- le préfixe à utiliser dans le résultat
- le suffixe à utiliser dans le résultat

Exemple (code Java 8) :

```
    StringJoiner joiner = new StringJoiner(",", "[", "]");
    joiner.add("1")
          .add("2")
          .add("3");
    System.out.println(joiner.toString());
```

Résultat :

```
[1,2,3]
```

Si aucune chaîne n'est ajoutée, alors la chaîne obtenue est simplement la concaténation du préfixe et du suffixe.

Exemple (code Java 8) :

```
    StringJoiner joiner = new StringJoiner(",", "[", "]");
    System.out.println(joiner.toString());
```

Résultat :

```
[]
```

6.6.7. La méthode join() de la classe String

La méthode statique join() de la classe String est ajoutée en Java 8. Elle permet de concaténer des chaînes entre-elles avec un séparateur. Elle possède deux surcharges.

La première attend en paramètre un délimiteur et un varargs de CharSequence.

Exemple (code Java 8) :

```
    String[] valeurs = { "1", "2", "3" };
    String chaine = String.join(", ", valeurs);
    System.out.println(chaine);
```

Résultat :

```
1, 2, 3
```

La seconde surcharge attend en paramètre un délimiteur et un Iterable< ? extends CharSequence>.

Exemple (code Java 8) :

```
    List<String> valeurs = List.of("1", "2", "3");
    String chaine = String.join(", ", valeurs);
    System.out.println(chaine);
```

Résultat :

```
1, 2, 3
```

6.7. La classe StringTokenizer

Cette classe permet de découper différentes portions d'une chaîne de caractères en fonction d'un ou plusieurs séparateurs.

La classe StringTokenizer possède plusieurs constructeurs permettant notamment de préciser la chaîne à décomposer et une chaîne contenant le séparateur à utiliser :

Constructeur	Rôle
StringTokenizer(String str)	Renvoyer une instance qui utilise les délimiteurs par défaut " \t\n\r\f" (espace, tabulation, nouvelle ligne, retour chariot et form feed). Les délimiteurs ne sont pas retournés dans les éléments extraits
StringTokenizer(String str, String delim)	Renvoyer une instance qui utilise le délimiteur passé en second paramètre. Les délimiteurs ne sont pas retournés dans les éléments extraits
StringTokenizer(String str, String delim, boolean returnDelims)	Renvoyer une instance qui utilise le délimiteur passé en second paramètre. Le troisième paramètre est un booléen qui précise si les délimiteurs sont retournés dans les éléments extraits ou pas

Elle possède plusieurs méthodes pour fournir un contrôle d'itération afin de parcourir les différents éléments :

Méthode	Rôle
int countTokens()	Retourner le nombre d'éléments restant, soit le nombre de fois que la méthode nextToken() peut être invoquée avant qu'elle ne lève une exception
boolean hasMoreElements() boolean hasMoreTokens()	Retourner true s'il reste encore au moins élément à obtenir sinon false
Object nextElement()	Retourner l'élément suivant sous la forme d'un Object s'il existe sinon lève une exception de type NoSuchElementException
String nextToken()	Retourner l'élément suivant sous la forme d'un String s'il existe sinon lève une exception de type NoSuchElementException
String nextToken(String delim)	Retourner l'élément suivant, en utilisant les délimiteurs fournis, s'il existe sinon lève une exception de type NoSuchElementException

Exemple (code Java 1.1) :

Exemple :

```
import java.util.*;

class TestStringTokenizer {
  public static void main(String args[]) {
    StringTokenizer st = new StringTokenizer("chaine1,chaine2,chaine3,chaine4",",");
    while (st.hasMoreTokens()) {
      System.out.println((st.nextToken()).toString());
    }
  }
}
```

Résultat :

```
chaine1
chaine2
chaine3
chaine4
```

6.8. Les chaînes de caractères et la sécurité

La classe java.lang.String est final, ce qui garantit que celle-ci ne peut pas avoir de classe fille et donc de s'assurer de son comportement. Comme les chaînes de caractères sont immuables, elles ne peuvent pas être modifiées. Ce comportement ne permet pas d'écraser ou de mettre à zéro son contenu, ce qui rend les chaînes inappropriées pour le stockage d'informations sensibles.

Avant Java 7, les chaînes de caractères sont stockées dans la permanent generation et ne sont donc jamais récupérées par le ramasse-miettes.

A partir de Java 7, les chaînes de caractères peuvent rester dans le heap jusqu'à ce que le ramasse-miettes récupère les instances inutilisées. Avant l'exécution de cette action, il est possible d'obtenir les données en obtenant et en exploitant un heap dump.

Il est dès lors préférable de stocker des données textuelles sensibles sous la forme d'un tableau de caractères plutôt que sous la forme de chaîne de caractères. Il est notamment possible de réinitialiser le tableau de caractères une fois que les données ne sont plus utiles pour les traitements.

Attention cependant, l'utilisation d'un char[] plutôt qu'une String ne garantit pas la sécurisation des données : cela offre la possibilité de réduire les possibilités pour obtenir ces données.

6.9. Le stockage en mémoire

Selon la manière dont une chaîne est créée, elle n'est pas stockée de la même manière en mémoire. Une instance de type String peut être stockée dans deux espaces :

- le pool de String : lorsqu'une chaîne est créée et si la chaîne existe déjà dans le pool, la référence de la chaîne existante sera retournée, au lieu de créer un nouvel objet et de retourner sa référence
- heap : chaque objet String dans le heap a son propre espace comme tout autre objet, même si deux objets String ont le même contenu

Comme les chaînes de caractères sont immuables, la JVM peut ne stocker qu'une seule copie d'une chaîne de caractères littérales. Ce comportement se nomme interning et repose sur l'utilisation du pool de String.

Lors de l'utilisation de la syntaxe littérale, la chaîne obtenue est stockée dans le pool de String. Dans les versions antérieures à la version 7 de Java, le pool de String est stocké dans la Permanent Generation. A partir de Java 7, le pool de String est stocké dans le heap.

Lors de la création d'une nouvelle instance en utilisant l'instruction new, une nouvelle instance est créée dans le heap même si une instance qui encapsule la même chaîne est présente dans le pool de String.

6.9.1. Le pool de String

La JVM dispose d'un pool de String, nommé String constant pool : c'est une région de la mémoire de la JVM dans laquelle elle peut stocker des instances de chaînes de caractères uniques.

Cela permet d'éviter d'avoir plusieurs instances avec la même valeur et ainsi d'améliorer les performances grâce à :

- la réduction de la quantité de mémoire utilisée
- la réduction du nombre d'allocations
- la réduction de l'activité du ramasse-miettes

Le pool de Strings agit comme un cache pour les chaînes de caractères :

- la JVM stocke une seule instance d'une même chaîne dans le pool
- lors de la création d'une nouvelle chaîne, la JVM cherche si une chaîne avec la même valeur est présente dans le pool
- si elle est trouvée, alors c'est la référence existante qui est utilisée
- si elle n'est pas trouvée, alors elle est ajoutée dans le pool et c'est sa référence qui est utilisée. Ce mécanisme est nommé interning

La mise en cache des chaînes de caractères et leur réutilisation permet de réduire la consommation mémoire et améliore les performances. Ceci est possible car la classe String est immuable.

Les chaînes de caractères littérales sont automatiquement « interned » donc mises dans le pool si elles n'y sont pas déjà.

Exemple :

```
    String chaine1 = "test";
    String chaine2 = "test";
    System.out.println(chaine1 == chaine2);
```

Résultat :

```
true
```

Attention : ce mécanisme n'est pas mis en oeuvre lors de la création d'une instance de type String en utilisant un de ces constructeurs. Dans ce cas, une nouvelle instance est créée, même si elle existe déjà dans le pool de Strings.

Lors de la création d'une chaîne de caractères en utilisant un des constructeurs de la classe String, une nouvelle instance sera créée dans le heap, en dehors du pool de String.

Exemple :

```
    String chaine1 = "test";
    String chaine2 = "test";
    String chaine3 = new String("test");
    System.out.println(chaine1 == chaine2);
    System.out.println(chaine1 == chaine3);
```

Résultat :

```
true
false
```

Il est possible de forcer une instance à être incluse dans le pool de chaînes en utilisant la méthode intern() de l'instance. La méthode intern() ajoute la chaîne de caractères au pool de chaînes si elle n'y existe pas déjà, et renvoie la référence de cette chaîne « internée » :

Exemple :

```
    String chaine1 = "test";
    String chaine2 = "test";
    String chaine3 = new String("test");
    System.out.println(chaine1 == chaine2);
    System.out.println(chaine1 == chaine3);
    chaine3 = chaine3.intern();
    System.out.println(chaine1 == chaine3);
```

Résultat :

```
true
false
true
```

Les chaînes de caractères contenues dans le pool de Strings sont récupérables par le ramasse-miettes s'il n'existe plus aucune référence vers leurs instances.

6.10. Les blocs de texte (Text Blocks)

Historiquement, travailler avec du texte littéral multilignes a toujours été fastidieux en Java : les chaînes de caractères littérales sur plusieurs lignes nécessitaient un enchevêtrement de terminaisons de ligne, de délimiteurs de fin de chaîne et d'opérateurs de concaténation.

Les lignes longues sont très difficiles à lire et la concaténation et peu pratiques, à la fois parce qu'il n'est pas possible de copier/coller uniquement le texte et parce qu'il y a beaucoup de caractères parasites (\n" +) sur chaque ligne.

Si ce texte est de l'HTML, XML, SQL ou autres, il est généralement en plus nécessaire d'échapper certains caractères, ce qui rend la lecture et la maintenance du contenu particulièrement compliquées.

Introduits en preview dans Java 13 et en standard en Java 15, les blocs de texte offrent un moyen simple de fournir de manière prévisible des chaînes de caractères littérales multilignes tout en évitant la plupart des séquences d'échappement.

Les blocs de texte permettent de facilement exprimer dans le code une chaîne de caractères multi-lignes brute sans interpolation de variables ou d'expressions.

Un bloc de texte est une nouvelle forme pour exprimer une chaîne de caractères dans le langage Java en offrant une meilleure expressivité et moins de complexité pour des chaînes multilignes.

Comme les chaînes de caractères littérales et les blocs de texte sont compilés en objet de type java.lang.String, les deux syntaxes peuvent utiliser les mêmes fonctionnalités. Seule la syntaxe diffère pour exprimer une chaîne mono-ligne ou multilignes.

6.10.1. La situation historique

Le support des chaînes de caractères littérales en Java a toujours souffert de la façon dont les chaînes de caractères sont définies. Une chaîne littérale commence par un double guillemet et se termine par un double guillemet. Malheureusement la chaîne ne peut s'étendre que sur une seule ligne. Si la chaîne doit tenir sur plusieurs lignes, il faut utiliser le caractère d'échappement de saut de ligne \n, fermer la chaîne et concaténer la ligne suivante.

Cela rend les chaînes de caractères multilignes complexes à écrire et à lire et donc à maintenir.

Historiquement la définition de chaînes de caractères littérale multilignes a toujours été fastidieuse car elle requière l'utilisation sur chaque ligne d'un caractère d'échappement \n et de l'opérateur +.

Certaines valeurs sont donc difficiles à exprimer notamment celles qui concernent des chaînes multilignes.

Exemple :

```
String html =
  "<HTML>\n\t<BODY>\n\t\t<H1>\"Java\"</H1> \n\t</BODY>\n</HTML>\n";
```

ou

Exemple :

```
String html = "<HTML>" +
"\n\t" + "<BODY>" +
"\n\t\t" + "<H1>\"Java\"</H1>" +
"\n\t" + "</BODY>" +
"\n" + "</HTML>";
```

L'équivalent sous la forme d'un bloc de texte est :

Exemple (code Java 14) :

```
String html = """
<HTML>
  <BODY>
    <H1>"Java"</H1>
  </BODY>
</HTML>""";
```

6.10.2. Les solutions proposées

Initialement prévue Java 12, la JEP 326 (Raw String Literals) propose les chaînes brutes littérales (raw-string literals), mais les retours négatifs issus de l'évaluation des versions en early access ont conduits à sa suppression avant sa release. Elle est finalement abandonnée en décembre 2018.

Une nouvelle solution est proposée sous la forme de blocs de texte. Les blocs de texte sont introduits en Java 13 en mode preview, suivi par une seconde preview en Java 14 grâce à plusieurs JEP :

- JEP 355: Text Blocks : introduit une nouvelle syntaxe pour faciliter la saisie, le formatage et la lecture de chaînes de caractères multilignes. Notamment, il n'est plus nécessaire de concaténer chaque ligne avec un retour chariot et d'échapper la plupart des caractères dont les doubles quotes
- JEP 368: Text Blocks (Second Preview) : ajoute le support de deux nouvelles séquences d'échappement \s et \<line-terminator> pour faciliter certains cas

Finalement, la JEP 378 de Java 15, définit les blocs de texte comme standard dans la syntaxe du langage Java.

6.10.3. La description des blocs de texte

Les blocs de texte facilitent l'utilisation de chaînes de caractères littérales multilignes : il propose une syntaxe littérale pour exprimer des chaînes de caractères multilignes.

Un bloc de texte est une chaîne de caractères littérale multilignes qui évite d'avoir recours à la plupart des séquences d'échappement, formate automatiquement la chaîne de manière prévisible et donne le contrôle sur l'indentation.

Les blocs de texte possèdent plusieurs buts :

- simplifier la saisie de chaînes de caractères multilignes lignes dans le code source, en évitant les séquences d'échappement dans les cas courants

- améliorer la lisibilité des chaînes dans le code source notamment si la chaîne contient du code formaté comme HTML, XML, JSon, ...
- assurer une compatibilité avec les chaînes littérales

Un bloc de texte est une nouvelle forme pour exprimer une chaîne de caractères littérale dans le langage Java en offrant une meilleure expressivité et moins de complexité pour des chaînes multilignes.

Les blocs de texte n'introduisent pas un nouveau type mais uniquement une nouvelle syntaxe qui sera traitée pour le compilateur pour obtenir un objet de type java.lang.String. Cela permet d'utiliser un bloc de texte partout où une chaîne de caractères littérale peut être utilisée.

Résultat :

```
jshell> System.out.println("""
   ...>               <HTML>
   ...>                 <BODY>
   ...>                   <H1>"Java"</H1>
   ...>                 </BODY>
   ...>               </HTML>""");
<HTML>
  <BODY>
    <H1>"Java"</H1>
  </BODY>
</HTML>
```

Le compilateur supprimera les espaces accessoires, par exemple ceux utilis&s pour aligner le contenu. Cette gestion des espaces pour l'indentation justifie le nom de bloc de texte plutôt que chaîne littérale multilignes.

Quelques exemples simples :

Résultat :

```
String message = """
   ...> Bonjour""";
message ==> "Bonjour"

jshell> String message = """
   ...>     Bonjour""";
message ==> "Bonjour"

jshell> String message = """
   ...> Bonjour
   ...> """;
message ==> "Bonjour\n"

jshell> String message = """
   ...>     Bonjour
   ...> """;
message ==> "    Bonjour\n"

jshell> String message = """
   ...> Bonjour
   ...>     """;
message ==> "Bonjour\n"

jshell> String message = """
   ...>     Bonjour
   ...>     """;
message ==> "Bonjour\n"

jshell> String message = """
   ...>     Bonjour
   ...>     Et bienvenue""";
message ==> "Bonjour\nEt bienvenue"
```

Plusieurs points sont remarquables dans ces exemples :

- les blocs de texte sont transformés en chaînes de caractères littérales
- le positionnement du délimiteur de fin à une incidence pour avoir une ligne vide ou pas mais aussi sur l'indentation

6.10.4. Les avantages

Les blocs de texte permettent de créer facilement des chaînes de caractères multilignes. Ils sont aussi beaucoup plus simples à lire car basiquement ils ne requièrent aucune séquence d'échappement.

Les blocs de texte facilitent la saisie et la lecture du contenu d'une chaîne de caractères multilignes notamment car elle n'oblige pas à ajouter un \n, une double quote et un opérateur de concaténation + comme le requière les chaînes de caractères littérales historiques.

Les blocs de texte offrent une solution différente. Dans un bloc de texte, tout ce qui se trouve entre le délimiteur de début et de fin fait partie de la chaîne, y compris les retours chariots.

Un des avantages avec les blocs de texte c'est que syntaxiquement c'est plus simple et beaucoup plus lisible : un seul délimiteur de début et de fin et le contenu n'est pas pollué car des caractères d'échappement.

6.10.5. La syntaxe

La syntaxe est proche de celle utilisée par Python.

Un bloc de texte peut contenir zéro ou plusieurs caractères entourés par un délimiteur de début et de fin. La syntaxe repose sur deux délimiteurs particuliers distincts :

- délimiteur de début : trois double quotes """ et un retour chariot qui est obligatoire
- délimiteur de fin : trois double quotes """

L'utilisation de trois doubles quotes d'affilés comme délimiteur a été choisie pour facilement l'identification d'un bloc de texte.

Les blocs de texte débutent par un délimiteur de début qui est composé de plusieurs éléments :

- 3 caractères double quotes qui se suivent : """
- zéro ou plusieurs caractères espaces
- un retour chariot

Le retour chariot ne fait pas parti du contenu du bloc de code. Le contenu du texte débute après le retour chariot du délimiteur de début.

Le délimiteur de début peut être suivi d'aucun ou plusieurs espaces et d'un retour chariot. Un bloc de texte ne peut donc pas être exprimé sur une seule ligne.

Résultat :

```
jshell> String chaine = """" """";
|  Error:
|  illegal text block open delimiter sequence, missing line terminator
|  String chaine = """" """";
|                      ^

jshell> String chaine = """"""""";
|  Error:
|  illegal text block open delimiter sequence, missing line terminator
|  String chaine = """"""""";
|                      ^

jshell> String chaine = """"test"""";
|  Error:
```

```
|  illegal text block open delimiter sequence, missing line terminator
|  String chaine = """"test"""";
|                     ^
```

La raison principale est qu'un bloc de texte est essentiellement destiné à être utilisé pour représenter des chaînes de caractères littérales multilignes. Une autre raison est que l'obligation d'avoir un retour chariot facilite la détermination des espaces accessoires.

Contrairement aux chaînes de caractères littérales, un bloc de texte peut contenir :

- des retours chariots
- des caractères double quotes qui n'ont pas besoin d'être échappés
- des caractères échappés incluant \n même si ce dernier n'est pas nécessaire ou recommandé

Un bloc de texte se termine par le délimiteur de fin qui est composé 3 caractères double quotes qui se suivent : """

Le contenu du texte se termine au caractère précédent le délimiteur de fin.

Le délimiteur de fin peut être soit dans la dernière ligne du contenu, soit sur sa propre ligne (dans ce dernier cas, la chaîne se termine par une nouvelle ligne).

Si le délimiteur de fin est sur une ligne dédiée, alors une ligne vide est ajoutée au contenu du bloc de texte.

Exemple (code Java 15) :

```
System.out.println("""
ligne 1
ligne 2
ligne 3
""");
```

Est équivalent à :

Exemple :

```
System.out.println("ligne 1\nligne 2\nligne 3\n");
```

Ou à :

Exemple :

```
System.out.println("ligne 1\n" +
"ligne 2\n" +
"ligne 3\n");
```

Si un retour chariot n'est pas nécessaire à la fin du bloc de texte alors le délimiteur de fin peut être placé à la fin de la dernière ligne du contenu.

Exemple (code Java 15) :

```
"""
ligne 1
ligne 2
ligne 3"""
```

Le bloc de texte ci-dessous est équivalent à la chaîne de caractères.

Exemple (code Java 15) :

```
"ligne 1\nligne 2\nligne 3"
```

6.10.6. La gestion de l'indentation accessoire et essentielle

Il est possible d'aligner le contenu du bloc detexte avec une indentation accessoire. Les blocs de texte seront généralement indentés en fonction du code adjacent et cette indentation n'a qu'une utilité pour faciliter la lisibilité. Cependant, le contenu lui-même peut être indenté de manière significative voire même essentiel comme par exemple pour des données au format JSON ou YAML.

Exemple (code Java 15) :

```
String personne = """
    {
        nom: "Dupond",
        prenom: "Martin",
        taille: 175
    }
    """;
```

Dans cet exemple, la première indentation qui concerne toutes les lignes du contenu n'a qu'un rôle accessoire. Par contre, l'indentation des trois propriétés permet de formater les données.

L'indentation peut utiliser des espaces et des tabulations (\t) qui sont considérées comme des espaces. Les espaces et les tabulations non échappées utilisés dans l'indentation sont traités de deux manières par le compilateur :

- accessoire : les espaces et les tabulations qui la composent sont supprimés par le compilateur car ils sont considérés insignifiants puisqu'utilisés pour faciliter la lisibilité
- essentiel : les espaces et les tabulations significatifs qui la composent sont conservés

L'indentation d'une partie des lignes est considérée comme importante et donc préservée, mais l'indentation partagée par toutes les lignes est supprimée.

Le compilateur prend en charge de déterminer les espaces accessoires : ils sont ignorés par le compilateur qui les retire. Dans les blocs de texte, le caractère qui n'est pas un espace le plus à gauche de l'une des lignes ou le délimiteur de fermeture le plus à gauche définit le début de l'espace des espaces significatifs.

Exemple (code Java 15) :

```
String html = """
~~~~~~~~~~~~~~<HTML>
~~~~~~~~~~~~~~  <BODY>
~~~~~~~~~~~~~~    <H1>"Java"</H1>
~~~~~~~~~~~~~~  </BODY>
~~~~~~~~~~~~~~</HTML>""";
```

Dans l'exemple ci-dessous, les espaces signalés avec un caractère tilde "~" sont considérés comme accessoires et sont donc retirés par le compilateur

Résultat :

```
jshell> String html = """
   ...>               <HTML>
   ...>                 <BODY>
   ...>                   <H1>"Java"</H1>
   ...>                 </BODY>
   ...>               </HTML>""";
html ==> "<HTML>\n  <BODY>\n    <H1>\"Java\"</H1>\n  </BODY>\n</HTML>"

jshell> System.out.println(html);
<HTML>
  <BODY>
    <H1>"Java"</H1>
  </BODY>
```

```
</HTML>

jshell>
```

Les triples guillemets fermants peuvent être placés sur leur propre ligne : cela permet de contrôler le nombre d'espaces accessoires à retirer mais cela ajoute également une nouvelle ligne à la fin de la chaîne générée.

Si les triples guillemets fermants sont placés directement après la fin du texte, alors il n'est pas possible de contrôler les espaces accessoires. Cela peut cependant être fait explicitement en utilisant la méthode indent() de String pour rajouter une indentation.

Sans mettre le délimiteur sur sa propre ligne, il n'est pas possible de gérer l'indentation. Dans ce cas, il y a deux possibilités :

- placer le délimiteur de fermeture sur sa propre ligne et supprimer la nouvelle ligne de la chaîne de caractères produite par le compilateur
- placer le délimiteur de fermeture sur la dernière ligne de contenu et ajouter l'indentation en modifiant la chaîne de caractères produite par le compilateur

Les développeurs peuvent choisir de ne pas supprimer tout ou partie des espaces en tête en utilisant le positionnement du délimiteur de fin. Pour indenter toutes les lignes, il faut :

- déplacer le contenu vers la droite
- utiliser le positionnement du délimiteur de fin : en le déplaçant vers la gauche sur une ligne dédiée

Pour marquer certains espaces comme essentiels afin qu'ils ne soient pas supprimés, il est possible de positionner le délimiteur de fin à la position souhaitée. La position du délimiteur de fin peut donc être utilisée de manière significative pour indiquer le début des espaces significatifs que le compilateur doit prendre en compte. Déplacer le délimiteur de fin vers la gauche ou le contenu vers la droite a le même effet : une indentation est incluse dans la chaîne de caractères créée par le compilateur.

Exemple (code Java 15) :

```
String html = """
              <HTML>
                <BODY>
                  <H1>"Java"</H1>
                </BODY>
              </HTML>
          """;
```

Ainsi la détermination des espaces accessoires déterminés par le compilateur change.

L'inconvénient de cette solution est qu'elle ajoute une ligne vide à la fin de la chaîne.

Résultat :

```
jshell> String html = """
   ...>               <HTML>
   ...>                 <BODY>
   ...>                   <H1>"Java"</H1>
   ...>                 </BODY>
   ...>               </HTML>
   ...>           """;
html ==> "    <HTML>\n      <BODY>\n        <H1>\"Java\"</ ...    </BODY>\n    </HTML>\n"

jshell> System.out.println(html);
    <HTML>
      <BODY>
        <H1>"Java"</H1>
      </BODY>
    </HTML>
```

```
jshell>
```

Pour éviter cette ligne supplémentaire, il est possible d'utiliser la séquence d'échappement \ + retour chariot (depuis Java 14).

Résultat :

```
jshell> String html = """
   ...>               <HTML>
   ...>                 <BODY>
   ...>                   <H1>"Java"</H1>
   ...>                 </BODY>
   ...>               </HTML>\
   ...>           """;
html ==> "    <HTML>\n      <BODY>\n        <H1>\"Java\"</ ...      </BODY>\n    </HTML>"
```

Des espaces peuvent également apparaître à la fin des lignes, par inadvertance ou par un copier/coller d'un autre fichier. Ces espaces de fin de ligne sont souvent involontaires et insignifiants. Il est très probable que le promoteur ne s'en soucie pas. De plus, ils ne sont généralement pas visibles dans l'éditeur de code source sauf si une fonctionnalité visuelle permet de les faire apparaître.

Par défaut dans les blocs de texte, les espaces en fin de ligne sont également considérés comme accessoires et seront supprimés. Ceci est fait de manière à ce que le contenu du bloc de texte soit toujours visuellement discernable. Si cela n'était pas fait, un éditeur de texte qui supprime automatiquement les espaces de fin de ligne pourrait modifier de manière invisible le contenu d'un bloc de texte.

Si des espaces de fin doivent être significatifs (comme par exemple deux espaces pour une balise <br /> en Markdown), il faut utiliser la séquence d'échappement octale \040 (caractère ASCII 32, espace blanc) ou \s (depuis Java 14).

Il ne faut surtout utiliser la séquence d'échappement \u0020 car elle est traduite à la lecture du fichier source par le compilateur, et donc avant l'analyse lexicale.

6.10.7. L'utilisation des séquences d'échappement

Les blocs de texte proposent un support des séquences d'échappement de la même manière que les chaînes de caractères littérales mais généralement ce n'est pas conseillé car rarement utile.

Il est donc possible d'utiliser des séquences d'échappement dans les blocs de texte comme dans les chaînes littérales. Elles sont interprétées comme elles le sont dans les chaînes de caractères littérales.

Résultat :

```
jshell> String chaine = """
   ...> hello\nworld"""
chaine ==> "hello\nworld"

jshell> println(chaine)
hello
world
```

Il n'est cependant pas nécessaire d'échapper les doubles quotes dans le contenu d'un bloc de texte.

Résultat :

```
jshell> String message = """
   ...>     "Hello" Java""";
message ==> "\"Hello\" Java"
```

Attention, une double quote ne peut pas être accolée au délimiteur de fin du bloc de texte

Résultat :

```
jshell> String message = """
   ...>     Hello "Java"""";
|  Error:
|  unclosed string literal
|      Hello "Java"""";
|                    ^
```

Il faut soit ajouter un espace qui sera retiré comme tout espace en fin de ligne

Résultat :

```
jshell> String message = """
   ...>     Hello "Java" """;
message ==> "Hello \"Java\""
```

Soit échapper la double quote

Résultat :

```
jshell> String message = """
   ...>     Hello "Java\"""";
message ==> "Hello \"Java\""
```

Comme le délimiteur utilisé pour les blocs de texte est une triple double quotes, il faut échapper cette séquence si elle se trouve dans le contenu d'un bloc de texte. Même s'il est possible d'échapper chacune des doubles quotes, le plus simple est d'échapper uniquement la première double quote.

Résultat :

```
jshell> String chaine = """
   ...>   test \""" test""";
chaine ==> "test \"\"\" test"
```

Il faut obligatoirement échapper une séquence composée d'un triple double quote.

Exemple (code Java 15) :

```
String code = """
    String message = \"""
        Bloc de texte dans un bloc de texte
        \""";
    """;
```

Pour des raisons de lisibilité, c'est la première double quote qui est échappée mais il est possible d'échapper n'importe lequel des trois doubles quotes.

Résultat :

```
jshell> String code = """
   ...>     String message = ""\"
   ...>         Bloc de texte dans un bloc de texte
   ...>         "\"";
   ...>     """;
code ==> "String message = \"\"\"\n    Bloc de texte dans  ... c de texte\n    \"\"\";\n"

jshell>
```

La traduction des séquences d'échappements est la dernière étape du traitement des bloc de texte par le compilateur, il est donc possible de contourner les étapes de normalisation des fins de lignes et de suppression des espaces en utilisant des séquences d'échappement explicites. Par exemple :

Résultat :

```
jshell> String message = """
   ...>     Lundi   \040
   ...>     Mardi   \040
   ...>     Mercredi\040
   ...>     """;
message ==> "Lundi    \nMardi    \nMercredi \n"

jshell>
```

Les séquences d'échappement d'un caractère espace sous la forme de la valeur octale 40 (20 en décimal) garantie que les trois lignes conserveront leurs espaces car les séquences ne seront interprétées qu'à la troisième étape du traitement des blocs de texte par le compilateur.

Les séquences d'échappement invalides sont interdites par le compilateur.

Résultat :

```
jshell> String chaine = """
   ...> hello \
world"""
|
Error:
|
illegal escape character
|
hello \
world"""
|         ^
|
Error:
|
reached end of file while parsing
|  hello \ world"""
|                  ^
```

6.10.7.1. Les séquences d'échappement ajoutées en Java 14

La JEP 268 dans Java 14 propose le support de deux nouvelles séquences d'échappement :

- \<fin-de-ligne> : supprime explicitement l'inclusion du caractère de fin de ligne implicite. Cela revient à concaténer la chaîne et la suivante
- \s : est remplacé par un caractère espace

Elles peuvent être utilisées pour ajuster le formatage d'un bloc de texte.

La séquence d'échappement \<fin-de-ligne> permet de supprimer le retour chariot de la ligne et ainsi de concaténer la ligne courant avec la suivante.

Résultat :

```
jshell> String texte = """
   ...>     ligne 1 \
   ...>     ligne 2 \
   ...>     ligne 3""";
texte ==> "ligne 1 ligne 2 ligne 3"
```

Cela peut permettre de faciliter la lecture d'un bloc de texte dont le contenu est très long.

Cela peut aussi être pratique pour supprimer le retour chariot dans le cas où le délimiteur de fin est utilisé sur une ligne dédiée pour définir l'indentation accessoire.

Résultat :

```
jshell> String texte = """
   ...>     ligne 1 \
   ...>     ligne 2 \
   ...>     ligne 3 \
   ...>   """;
texte ==> "  ligne 1   ligne 2   ligne 3 "
```

La séquence d'échappement \<fin-de-ligne> ne peut être utilisé que dans les blocs de texte puisque les chaîne de caractères littérales ne peuvent être que sur une ligne unique.

La séquence d'échappement \s est remplacé par un espace (\040 soit le caractère ASCII 32) qui ne sera pas supprimé. Comme les séquences d'échappement ne sont traduites qu'après la suppression des espaces accessoires, elles peuvent servir de marque pour empêcher la suppression des espaces qui précèdent.

Cela peut par exemple être utilisé pour conserver des espaces à la fin des lignes pour les aligner ou s'assurer qu'elles sont tous la même taille.

Résultat :

```
jshell> String texte = """
   ...>     ligne 1 \s
   ...>     ligne 2 \s
   ...>     ligne 3 \s""";
texte ==> "ligne 1  \nligne 2  \nligne 3  "
```

La séquence d'échappement \s est aussi utilisable dans les chaînes de caractères littérales classiques.

Exemple (code Java 14) :

```
  public static void main(String[] args) {
    String message = "**\s\sHello\s\s**";
    System.out.println(message);
  }
```

Résultat :

```
**  Hello  **
```

6.10.8. Les traitements par le compilateur

Pour éviter des traitements coûteux à l'exécution, c'est le compilateur qui retire l'indentation accessoire en tête et en fin des lignes.

Un bloc de texte est une expression constante de type String, tout comme une chaîne de caractères littérale. Cependant, à la différence d'une chaîne littérale, le contenu d'un bloc de texte est traité par le compilateur Java en trois étapes distinctes :

- les retours chariots présents dans le contenu sont tous transformés en LF (\u000A) pour garantir le portage multi-plateforme du code source
- les espaces utilisés pour suivre l'indentation du code source Java, sont supprimés par défaut
- les séquences d'échappement dans le contenu sont interprétées

Les blocs de texte sont enregistrés dans le fichier .class en tant que constantes comme les chaînes de caractères littérales. Rien ne les distinguent dans le bytecode des fichiers .class.

A l'exécution les blocs de textes sont des instances de la classe String comme les chaînes de caractères littérales : il n'est pas possible de les distinguer en dehors du code source.

6.10.8.1. L'uniformisation des retours chariots

Une des problématiques à gérer dans les chaînes de caractères multilignes est le ou les caractères utilisés comme fin de lignes dans le code source car ils varient selon le système d'exploitation utilisé. Notamment lors de la modification du code source sur différents systèmes, il est possible d'avoir plusieurs fins de lignes utilisées, ce qui peut amener à des confusions ou des incohérences.

Pour apporter une solution, la première étape de traitements des blocs de texte par le compilateur est de normaliser les fins de lignes : quelques soient les fins de lignes CR (\u000D), CRLF (\u000D\u000A sous Windows), ou LF (\u000A sous Linux) pour les remplacer par LF (\u000A).

Cela permet de garantir d'avoir toujours les mêmes fins de lignes utilisées quel que soit le système sur lequel le code est exécuté.

S'il est nécessaire d'avoir des fins de lignes spécifiques au système d'exploitation, il faut remplacer toutes les fins de lignes \n par la séquence utilisée par le système d'exploitation.

Exemple :

```
      chaine.replaceAll("\n", System.lineSeparator()) ;
```

Les séquences d'échappement \n (LF) et \r (CR) ne sont pas interprétées durant cette étape de normalisation.

Résultat :

```
jshell> String chaine = """
   ...>     ligne 1\r
   ...>     ligne 2\n
   ...>     ligne 3\r
   ...>     """
chaine ==> "ligne 1\r\nligne 2\n\nligne 3\r\n"

jshell>
```

6.10.8.2. La gestion des espaces pour l'indentation et en fin de ligne

Le compilateur traite le bloc de texte pour différencier les espaces accessoires au début et à la fin de chaque ligne, des espaces essentiels.

Le compilateur ne détermine pas les espaces de l'indentation essentielle : il supprime les espaces accessoires et considère tout le reste comme essentiel. Le compilateur supprime la même quantité d'espace à chaque ligne de contenu jusqu'à ce qu'au moins une des lignes comporte un caractère qui ne soit pas un espace dans la position la plus à gauche.

La position du délimiteur d'ouverture n'a aucun effet sur l'algorithme, mais la position du délimiteur de fermeture a un effet si celui-ci est placée sur sa propre ligne. La position des espaces accessoires est déterminée grâce à deux règles :

- le caractère qui n'est pas un espace le plus à gauche de toutes les lignes
- ou la position la position du délimiteur final s'il est placé sur une ligne dédiée est qu'il est plus à gauche que le caractère précédent

L'algorithme est composé de plusieurs étapes :

- découper le bloc de texte en lignes individuelles. Les lignes composées uniquement d'un retour chariot restent des lignes vides
- ajouter toutes les lignes non vides ou non composées entièrement d'espaces dans un ensemble des lignes déterminantes

- si la dernière ligne du bloc de texte (celle qui contient le délimiteur de fin) est vide alors elle est ajoutée à l'ensemble des lignes déterminantes car l'indentation du délimiteur de fin peut influencer l'indentation du bloc de texte
- calculer le nombre d'espace qui compose l'indentation accessoire en comptant le nombre d'espace en tête de chaque ligne et en prenant le nombre minimum.
- supprimer le nombre d'espaces calculé en tête de chaque ligne non vide
- supprimer les espaces à la fin de toutes les lignes. Ainsi les lignes qui ne contiennent que des espaces sont conservées en tant que ligne vide

Les séquences d'échappement \b (backspace) et \t (tab) ne sont pas interprétées par l'algorithme : ils seront traités lors de la dernière étape qui interprète les séquences d'échappement.

L'implémentation de cet algorithme est proposé dans la méthode stripIndent() de la classe String pour pouvoir être utilisée dans les traitements selon les besoins.

Le compilateur retire les indentations qui sont partagées sur toutes les lignes de manière accessoire et les supprime. Les autres caractères d'espacement sont considérés comme essentiels et sont conservés.

Pour un comportement correct de l'algorithme, il faut que ce soit tous les mêmes caractères qui soient utilisés dans l'indentation accessoire.

Lors de l'utilisation d'une tabulation, le compilateur la considère comme un seul espace car il ne peut pas savoir combien d'espaces représente une tabulation. Il applique une règle qui traite chaque caractère comme un seul espace.

Ainsi si l'indentation accessoire utilise des espaces et des tabulations, le résultat du retrait de l'indentation ne sera probablement pas celui attendu.

Résultat :

```
public class TestTextBlock {

  public static void main(String[] args) {
    String message = """
        ligne1
               ligne2
        ligne3""";
    System.out.println(message);
  }
}
```

Dans l'exemple ci-dessus, la première et la dernière ligne du bloc de code est indentée avec des espaces. La seconde ligne est indentée avec des tabulations. Dans certains éditeurs de texte, selon la représentation des tabulations, ces trois lignes peuvent apparaître alignées.

Résultat :

```
C:\java>java TestTextBlock
      ligne1
ligne2
      ligne3

C:\java>
```

Le compilateur javac propose l'option -Xlint:text-blocks pour détecter les problèmes liés aux espaces blancs fortuits dans les blocs de texte. Si l'option est activée alors le compilateur émet un avertissement intitulé "inconsistent white space indentation".

Résultat :

```
C:\java>javac -Xlint:text-blocks TestTextBlock.java
TestTextBlock.java:4: warning: [text-blocks] inconsistent white space indentation
    String message = """
                     ^
```

```
1 warning
```

Cette option permet également d'activer un autre avertissement, "trailing white space will be removed", qui sera émis s'il y a un espace blanc de fin sur n'importe quelle ligne dans un bloc de texte.

Résultat :

```
public class TestTextBlock {

  public static void main(String[] args) {
    String message = """
        ligne1
               ligne2
        ligne3    """;
    System.out.println(message);
  }
}
```

Résultat :

```
C:\java>javac -Xlint:text-blocks TestTextBlock.java
TestTextBlock.java:4: warning: [text-blocks] inconsistent white space indentation
    String message = """
                     ^
TestTextBlock.java:4: warning: [text-blocks] trailing white space will be removed
    String message = """
                     ^
2 warnings

C:\java>
```

6.10.8.3. Le traitement des séquences d'échappement

Une fois le contenu re-indenté, les séquences d'échappement sont interprétées. Toutes les séquences d'échappement définis dans la JLS utilisables dans les chaînes de caractères sont utilisables dans les blocs de texte.

L'interprétation des séquences d'échappements à l'étape finale permet notamment aux développeurs d'utiliser les séquences \n, \f, et \r pour le formatage vertical d'une chaîne sans que cela n'affecte la normalisation des retours chariot à l'étape 1, et d'utiliser \b et \t pour le formatage horizontal d'une chaîne sans que cela n'affecte la suppression des espaces accessoires à l'étape 2.

Résultat :

```
jshell> String html = """
   ...>               <HTML>\r
   ...>                 <BODY>\r
   ...>                   <H1>"Java"</H1>\r
   ...>                 </BODY>\r
   ...>               </HTML>""";
html ==> "<HTML>\r\n  <BODY>\r\n    <H1>\"Java\"</H1>\r\n  </BODY>\r\n</HTML>"
```

Cela permet aussi des préserver des espaces en fin de ligne en utilisant la séquence d'échappement octale \040.

Résultat :

```
jshell> String couleurs = """
   ...>     rouge \040
   ...>     orange\040
   ...>     vert  \040""";
couleurs ==> "rouge  \norange \nvert   "
```

6.10.8.4. Le résultat de la transformation

Le bytecode généré par le compilateur pour un bloc de texte est une chaîne de caractères de type java.lang.String. Ainsi les valeurs littérales des chaînes de caractères et des blocs de texte sont compilés vers le type java.lang.String.

Dans le bytecode, il n'est donc pas possible de distinguer si le code source utilise un bloc de texte ou une chaîne de caractères littérale.

Comme toutes les chaînes de caractères littérales, celles générés par le compilateur à partir de bloc de texte sont ajoutées dans le pool de String. Les mêmes chaînes de caractères exprimées en littérales et en bloc de texte seront donc égales chaine1.equals(chaine2) retourne true et grâce à la gestion interne des chaînes dans la JVM, chaine1==chaine2 retourne également true.

Résultat :

```
jshell> String chaine1 = "test";
chaine1 ==> "test"

jshell> String chaine2 = """
   ...> test""";
chaine2 ==> "test"

jshell> chaine1==chaine2
$3 ==> true
```

6.10.9. La concaténation et le formatage de blocs de texte

Les blocs de texte peuvent être concaténés avec des chaînes de caractères (objets de type String ou valeurs littérales) et vice versa, puisque dans le byte code se sont des instances de type java.lang.String.

Résultat :

```
jshell> String message = """
   ...> Bonjour""" + ",  bienvenue"
message ==> "Bonjour,  bienvenue"

jshell>
```

La concaténation avec une chaîne de caractères peut par exemple être nécessaire pour insérer une valeur.

Résultat :

```
jshell> String nom = "JM"
nom ==> "JM"

jshell> String message = """
   ...> Bonjour """ + " " + nom + """
   ...> , bienvenue"""
message ==> "Bonjour JM, bienvenue"

jshell>
```

Cependant la syntaxe à utiliser n'est dans ce cas pas forcement la plus simple ni la plus lisible. e plus, les blocs de texte ne proposent pas de support pour l'interpolation et la concaténation. Il est cependant possible d'utiliser les méthodes replace() ou format() de la classe String.

Résultat :

```
jshell> String message = """
   ...> Bonjour $nom,
   ...>  bienvenue""".replace("$nom","JM")
message ==> "Bonjour JM,\n bienvenue"
```

```
jshell>

jshell> String message = String.format("""
   ...> Bonjour %s,
   ...>  bienvenue""", "JM")
message ==> "Bonjour JM,\n bienvenue"

jshell>
```

Il est aussi possible d'utiliser la méthode d'instance formatted() de la classe String ajoutée en Java 15.

Résultat :

```
jshell> String message = """
   ...> Bonjour %s,
   ...>  bienvenue""".formatted(nom)
message ==> "Bonjour JM,\n bienvenue"

jshell>
```

6.10.10. Les méthodes dans la classe String

Trois nouvelles méthodes en relation avec les blocs de texte ont été ajoutées à la classe String.

Méthode	Rôle
String formatted(Object... args)	Formater la chaîne qui contient le format en substituant les motifs inclus par les valeurs fournies en paramètres (depuis Java 15)
String stripIndent()	Renvoyer la chaîne elle-même dont l'indentation accessoire et de fin de ligne est retirée (depuis Java 15)
String translateEscapes()	Renvoyer la chaîne elle-même dont certaines séquences d'échappement sont traduites (depuis Java 15)

En Java 13, ces 3 méthodes sont ajoutées en étant dépréciées forRemoval=true pour indiquer que ces méthodes pourraient être retirées si les blocs de texte sont retirés pendant leur évaluation en mode preview.

En Java 14, ces 3 méthodes ne sont plus dépréciées mais elles sont annotées avec jdk.internal.PreviewFeature. Les API annotées avec cette annotation sont considérées comme les fonctionnalités du langage en mode preview puisque c'est la raison de leur ajout. Leur mise en oeuvre doit donc respecter les règles d'activation pour utilisation des fonctionnalités en mode preview.

Résultat :

```
C:\java>type TestString.java
public class TestString {

  public static void main(java.lang.String[] args) {
    String tab = "\\t".translateEscapes();
  }
}
C:\java>javac TestString.java
TestString.java:4: error: translateEscapes() is an API that is part of a preview feature
    String tab = "\\t".translateEscapes();
                      ^
1 error

C:\java>javac --enable-preview --source 14 TestString.java
Note: TestString.java uses preview language features.
Note: Recompile with -Xlint:preview for details.

C:\java>java TestString
Erreur : LinkageError lors du chargement de la classe principale TestString
       java.lang.UnsupportedClassVersionError: Preview features are not enabled for TestString
```

```
(class file version 58.65535). Try running with '--enable-preview'

C:\java>java --enable-preview TestString

C:\java>
```

6.10.10.1. La méthode translateEscapes

La méthode translateEscapes() permet de traduire les séquences d'échappement contenues dans la chaîne de caractères.

Séquence d'échappement	Traduction
\b	U+0008
\t	U+0009
\n	U+000A
\f	U+000C
\r	U+000D
\s	U+0020
\"	U+0022
\'	U+0027
\\	U+005C
\0 à \377	Caractères équivalent à la valeur octale
\<fin-de-ligne>	Suppression de la fin de ligne

Cette méthode ne traduit pas les séquences d'échappement avec une valeur Unicode (\uNNNN).

Résultat :

```
jshell> String retourChariot = "\\n".translateEscapes();
retourChariot ==> "\n"

jshell>
```

Comme la méthode est initialement associée à la fonctionnalité des blocs de texte en preview :

- en Java 13, cette méthode est dépréciée forRemoval=true
- en Java 14, elle est marquée comme étant associée à une fonctionnalité en preview
- en Java 15, elle est standard

6.10.10.2. La méthode formatted

La méthode formatted(Object... args) est une méthode d'instance qui est similaire à la méthode statique format() en lui passant en paramètres la chaîne elle-même et args. L'avantage est qu'il est possible de l'invoquer sur une valeur littérale sous la forme d'une chaîne de caractères ou d'un bloc de texte et même d'être chainée avec d'autres méthodes de classe String.

Résultat :

```
jshell> String message = """
   ...>     Reference : %s
   ...>     Prix : %.2f euros
   ...>     """.formatted("56789", 123.45);
message ==> "Reference : 56789\nPrix : 123,45 euros\n"
```

```
jshell>
```

Comme la méthode est initialement associée à la fonctionnalité des blocs de texte en preview :

- en Java 13, cette méthode est dépréciée forRemoval=true
- en Java 14, elle est marquée comme étant associée à une fonctionnalité en preview
- en Java 15, elle est standard

6.10.10.3. La méthode stripIndent

La méthode stripIndent() permet de retirer l'indentation accessoires en début et fin de lignes dans une chaîne de caractères multi-lignes en utilisant le même algorithme que celui utilisé par le compilateur pour traiter les blocs de texte.

Résultat :

```
jshell> String code = "   ligne 1\n   ligne 2\n   ligne 3";
code ==> "   ligne 1\n   ligne 2\n   ligne 3"

jshell> System.out.println(code);
   ligne 1
   ligne 2
   ligne 3

jshell> System.out.println(code.stripIndent());
ligne 1
ligne 2
ligne 3

jshell>
```

Comme la méthode est initialement associée à la fonctionnalité des blocs de texte en preview :

- en Java 13, cette méthode est dépréciée forRemoval=true
- en Java 14, elle est marquée comme étant associée à une fonctionnalité en preview
- en Java 15, elle est standard

6.10.11. Les cas d'utilisation

Les blocs de texte facilitent grandement la représentation sous la forme de chaînes de caractères littérales de code source (Java, JavaScript, SQL, ...) ou de données formatées (XML, JSON, HTML, ...) dans le code source Java.

Exemple (code Java 15) :

```
String html = """
    <html>
      <body>
        <p class="titre">Hello world</p>
      </body>
    </html>""";
```

Historiquement, la saisie et la lecture d'un document Json dans une chaîne de caractères littérale sont compliquées car il faut en plus échapper les doubles quotes.

Exemple :

```
    String json = "    {\n" +
        "      \"nom\": \"Dupond\",\n" +
        "      \"prenom\": \"Pierre\",\n" +
        "      \"id\": 1234\n" +
        "    }";
```

L'utilisation des blocs de texte facilite la saisie et la lecture de documents Json.

Exemple (code Java 15) :

```
String json = """
    {
      "nom": "Dupond",
      "prenom": "Pierre",
      "id": 1234
    }""";
```

Les blocs de texte facilitent aussi l'incorporation de requêtes SQL en limitant les risques d'erreur de saisie qui pourraient engendrer des requêtes invalides.

Exemple (code Java 15) :

```
String sql = """
    SELECT id, nom, prenom
    FROM utilisateur
    WHERE id = ? """;
```

Il est possible d'utiliser les blocs de texte pour facilement inclure du code source Java dans une chaîne de caractères.

Exemple (code Java 15) :

```
String source = """
         String message = "Hello world";
         System.out.println(message);
    """;
```

Les blocs texte peuvent aussi faciliter l'insertion d'expressions régulières qui requièrent souvent d'échapper les caractères spéciaux qu'elles contiennent.

6.10.12. Les bonnes pratiques

Un bloc de texte peut être utilisé pour représenter une chaîne de caractères vide mais cela n'est pas recommandé car sa représentation nécessite deux lignes.

Exemple (code Java 15) :

```
String chaine = """
""";
```

C'est la même chose pour une chaîne qui ne contient qu'une seule ligne

Exemple (code Java 15) :

```
String chaine = """
test""";
```

Dans les deux cas, il est préférable d'utiliser des chaînes de caractères littérales plutôt que des blocs de texte.

Il est préférable de réserver l'utilisation des caractères d'échappement lorsque cela améliore la lisibilité.

Exemple (code Java 15) :

```
String csv = """
    Nom;Adresse;Ville
    Alain Dupond;"23 rue de la paille\nAppartement 12";Paris
    Sophie Durand;"9 rue de la tour\nRésidence Victor Hugo";Lyon
""";
```

Il n'est pas conseillé d'aligner le contenu sur le début du délimiteur de fin car cela implique de décaler toutes les lignes si nom de la variable change.

Exemple (code Java 15) :

```
String chaine = """
                ligne1
                ligne2
                ligne3""";
```

Il est recommandé d'utiliser sa propre indentation accessoire.

Exemple (code Java 15) :

```
String chaine = """
      ligne1
      ligne2
      ligne3""";
```

Il est recommandé d'utiliser la séquence d'échappement \<fin_de_ligne> pour éviter la ligne vide lorsque le délimiteur de fin est sur sa propre ligne.

Exemple (code Java 15) :

```
String chaine = """
      ligne1
      ligne2
      ligne3\
  """;
```

Il n'est pas recommandé de mixer l'utilisation d'espaces et de tabulations dans l'indentation d'un bloc de texte

Il n'est pas recommandé d'utiliser un bloc de texte directement dans une expression complexe comme une expression Lambda. Il est préférable de définir le bloc de texte dans une variable locale et d'utiliser cette variable dans l'expression.

7. Les packages de bases

Chapitre 7

Niveau : 🟩🟩⬜⬜⬜⬜ Elémentaire

Le JDK se compose de nombreuses classes regroupées selon leurs fonctionnalités en packages. La première version du JDK était composée de 8 packages qui constituent encore aujourd'hui les packages de bases des différentes versions du JDK.

Ce chapitre contient plusieurs sections :

- Les packages selon la version du JDK
- Le package java.lang
- La présentation rapide du package awt java
- La présentation rapide du package java.io
- Le package java.util
- La présentation rapide du package java.net
- La présentation rapide du package java.applet

7.1. Les packages selon la version du JDK

Selon sa version, le JDK contient un certain nombre de packages, chacun étant constitué par un ensemble de classes qui couvrent un même domaine et apportent de nombreuses fonctionnalités. Les différentes versions du JDK sont constamment enrichies avec de nouveaux packages :

	Java													
Packages	1.0	1.1	1.2	1.3	1.4	1.5	6.0	7.0	8.0	9	10	11	12	13
java.applet Développement des applets	X	X	X	X	X	X	X	X	X	X	X	X	X	X
java.awt Toolkit pour interfaces graphiques	X	X	X	X	X	X	X	X	X	X	X	X	X	X
java.awt.color Gérer et utiliser les couleurs			X	X	X	X	X	X	X	X	X	X	X	X
java.awt.datatransfer Echanger des données par le presse-papier		X	X	X	X	X	X	X	X	X	X	X	X	X
java.awt.desktop										X	X	X	X	X
java.awt.dnd Gérer le cliquer/glisser			X	X	X	X	X	X	X	X	X	X	X	X
java.awt.event Gérer les événements utilisateurs		X	X	X	X	X	X	X	X	X	X	X	X	X
java.awt.font Utiliser les fontes			X	X	X	X	X	X	X	X	X	X	X	X

java.awt.geom dessiner des formes géométriques			X	X	X	X	X	X	X	X	X	X	X	X
java.awt.im				X	X	X	X	X	X	X	X	X	X	X
java.awt.im.spi				X	X	X	X	X	X	X	X	X	X	X
java.awt.image Afficher des images		X	X	X	X	X	X	X	X	X	X	X	X	X
java.awt.image.renderable Modifier le rendu des images			X	X	X	X	X	X	X	X	X	X	X	X
java.awt.print Réaliser des impressions			X	X	X	X	X	X	X	X	X	X	X	X
java.beans Développer des composants réutilisables		X	X	X	X	X	X	X	X	X	X	X	X	X
java.beans.beancontext			X	X	X	X	X	X	X	X	X	X	X	X
java.io Gérer les flux	X	X	X	X	X	X	X	X	X	X	X	X	X	X
java.lang Classes de base du langage	X	X	X	X	X	X	X	X	X	X	X	X	X	X
java.lang.annotation						X	X	X	X	X	X	X	X	X
java.lang.instrument						X	X	X	X	X	X	X	X	X
java.lang.invoke								X	X	X	X	X	X	X
java.lang.management						X	X	X	X	X	X	X	X	X
java.lang.module										X	X	X	X	X
java.lang.ref			X	X	X	X	X	X	X	X	X	X	X	X
java.lang.reflect Utiliser la réflexion (introspection)		X	X	X	X	X	X	X	X	X	X	X	X	X
java.math Utiliser des opérations mathématiques		X	X	X	X	X	X	X	X	X	X	X	X	X
java.net Utiliser les fonctionnalités réseaux	X	X	X	X	X	X	X	X	X	X	X	X	X	X
java.net.http												X	X	X
java.net.spi										X	X	X	X	X
java.nio					X	X	X	X	X	X	X	X	X	X
java.nio.channels					X	X	X	X	X	X	X	X	X	X
java.nio.channels.spi					X	X	X	X	X	X	X	X	X	X
java.nio.charset					X	X	X	X	X	X	X	X	X	X
java.nio.charset.spi					X	X	X	X	X	X	X	X	X	X
java.nio.file								X	X	X	X	X	X	X
java.nio.file.attribute								X	X	X	X	X	X	X
java.nio.file.spi								X	X	X	X	X	X	X
java.rmi Développement d'objets distribués		X	X	X	X	X	X	X	X	X	X	X	X	X
java.rmi.activation			X	X	X	X	X	X	X	X	X	X	X	X
java.rmi.dgc		X	X	X	X	X	X	X	X	X	X	X	X	X

java.rmi.registry		X	X	X	X	X	X	X	X	X	X	X	X	X
java.rmi.server Gérer les objets serveurs de RMI		X	X	X	X	X	X	X	X	X	X	X	X	X
java.security Gérer les signatures et les certifications		X	X	X	X	X	X	X	X	X	X	X	X	X
java.security.acl		X	X	X	X	X	X	X	X	X	X	X	X	X
java.security.cert		X	X	X	X	X	X	X	X	X	X	X	X	X
java.security.interfaces		X	X	X	X	X	X	X	X	X	X	X	X	X
java.security.spec			X	X	X	X	X	X	X	X	X	X	X	X
java.sql API JDBC pour l'accès aux bases de données		X	X	X	X	X	X	X	X	X	X	X	X	X
java.text Formater des objets en texte		X	X	X	X	X	X	X	X	X	X	X	X	X
java.text.spi							X	X	X	X	X	X	X	X
java.time									X	X	X	X	X	X
java.time.chrono									X	X	X	X	X	X
java.time.format									X	X	X	X	X	X
java.time.temporal									X	X	X	X	X	X
java.time.zone									X	X	X	X	X	X
java.util Utilitaires divers	X	X	X	X	X	X	X	X	X	X	X	X	X	X
java.util.concurrent						X	X	X	X	X	X	X	X	X
java.util.concurrent.atomic						X	X	X	X	X	X	X	X	X
java.util.cocurrent.locks						X	X	X	X	X	X	X	X	X
java.util.function									X	X	X	X	X	X
java.util.jar Gérer les fichiers jar			X	X	X	X	X	X	X	X	X	X	X	X
java.util.logging Utiliser des logs					X	X	X	X	X	X	X	X	X	X
java.util.prefs Gérer des préférences					X	X	X	X	X	X	X	X	X	X
java.util.regex Utiliser les expressions régulières					X	X	X	X	X	X	X	X	X	X
java.util.spi							X	X	X	X	X	X	X	X
java.util.stream									X	X	X	X	X	X
java.util.zip Gérer les fichiers zip		X	X	X	X	X	X	X	X	X	X	X	X	X
javax.accessibility			X	X	X	X	X	X	X	X	X	X	X	X
javax.activation							X	X	X	X	X			
javax.activity						X	X	X	X	X	X			
javax.annotation							X	X	X	X	X	X	X	X
javax.annotation.processing							X	X	X	X	X	X	X	X
					X	X	X	X	X	X	X	X	X	X

javax.crypto Utiliser le cryptage des données														
javax.crypto.interfaces					X	X	X	X	X	X	X	X	X	X
javax.crypto.spec					X	X	X	X	X	X	X	X	X	X
javax.imageio					X	X	X	X	X	X	X	X	X	X
javax.imageio.event					X	X	X	X	X	X	X	X	X	X
javax.imageio.metadata					X	X	X	X	X	X	X	X	X	X
javax.imageio.plugins.bmp						X	X	X	X	X	X	X	X	X
javax.imageio.plugins.jpeg					X	X	X	X	X	X	X	X	X	X
javax.imageio.plugins.tiff											X	X	X	X
javax.imageio.spi					X	X	X	X	X	X	X	X	X	X
javax.imageio.stream					X	X	X	X	X	X	X	X	X	X
javax.jnlp											X			
javax.jws							X	X	X	X	X			
javax.jws.soap							X	X	X	X	X			
javax.lang.model							X	X	X	X	X	X	X	X
javax.lang.model.element							X	X	X	X	X	X	X	X
javax.lang.model.type							X	X	X	X	X	X	X	X
javax.lang.model.util							X	X	X	X	X	X	X	X
javax.management						X	X	X	X	X	X	X	X	X
javax.management.loading						X	X	X	X	X	X	X	X	X
javax.management.modelmbean						X	X	X	X	X	X	X	X	X
javax.management.monitor						X	X	X	X	X	X	X	X	X
javax.management.openmbean						X	X	X	X	X	X	X	X	X
javax.management.relation						X	X	X	X	X	X	X	X	X
javax.management.remote						X	X	X	X	X	X	X	X	X
javax.management.remote.rmi						X	X	X	X	X	X	X	X	X
javax.management.timer						X	X	X	X	X	X	X	X	X
javax.naming				X	X	X	X	X	X	X	X	X	X	X
javax.naming.directory				X	X	X	X	X	X	X	X	X	X	X
javax.naming.event				X	X	X	X	X	X	X	X	X	X	X
javax.naming.ldap				X	X	X	X	X	X	X	X	X	X	X
javax.naming.spi				X	X	X	X	X	X	X	X	X	X	X
javax.net					X	X	X	X	X	X	X	X	X	X
javax.net.ssl Utiliser une connexion réseau sécurisée avec SSL					X	X	X	X	X	X	X	X	X	X
javax.print					X	X	X	X	X	X	X	X	X	X
javax.print.attribute					X	X	X	X	X	X	X	X	X	X
javax.print.attribute.standard					X	X	X	X	X	X	X	X	X	X
javax.print.event					X	X	X	X	X	X	X	X	X	X

javax.rmi				X	X	X	X	X	X	X	X			
javax.rmi.CORBA				X	X	X	X	X	X	X	X			
javax.rmi.ssl						X	X	X	X	X	X	X	X	X
javax.script							X	X	X	X	X	X	X	X
javax.security.auth API JAAS pour l'authentification et l'autorisation					X	X	X	X	X	X	X	X	X	X
javax.security.auth.callback					X	X	X	X	X	X	X	X	X	X
javax.security.auth.kerberos					X	X	X	X	X	X	X	X	X	X
javax.security.auth.login					X	X	X	X	X	X	X	X	X	X
javax.security.auth.spi					X	X	X	X	X	X	X	X	X	X
javax.security.auth.x500					X	X	X	X	X	X	X	X	X	X
javax.security.cert					X	X	X	X	X	X	X	X	X	X
javax.security.sasl						X	X	X	X	X	X	X	X	X
javax.smartcardio										X	X	X	X	X
javax.sound.midi				X	X	X	X	X	X	X	X	X	X	X
javax.sound.midi.spi				X	X	X	X	X	X	X	X	X	X	X
javax.sound.sampled				X	X	X	X	X	X	X	X	X	X	X
javax.sound.sampled.spi				X	X	X	X	X	X	X	X	X	X	X
javax.sql					X	X	X	X	X	X	X	X	X	X
javax.sql.rowset						X	X	X	X	X	X	X	X	X
javax.sql.rowset.serial						X	X	X	X	X	X	X	X	X
javax.sql.rowset.spi						X	X	X	X	X	X	X	X	X
javax.swing Swing pour développer des interfaces graphiques			X	X	X	X	X	X	X	X	X	X	X	X
javax.swing.border Gérer les bordures des composants Swing			X	X	X	X	X	X	X	X	X	X	X	X
javax.swing.colorchooser Composant pour sélectionner une couleur			X	X	X	X	X	X	X	X	X	X	X	X
javax.swing.event Gérer des événements utilisateur des composants Swing			X	X	X	X	X	X	X	X	X	X	X	X
javax.swing.filechooser Composant pour sélectionner un fichier			X	X	X	X	X	X	X	X	X	X	X	X
javax.swing.plaf Gérer l'aspect des composants Swing			X	X	X	X	X	X	X	X	X	X	X	X
javax.swing.plaf.basic			X	X	X	X	X	X	X	X	X	X	X	X
javax.swing.plaf.metal Gérer l'aspect metal des composants Swing			X	X	X	X	X	X	X	X	X	X	X	X
javax.swing.plaf.multi			X	X	X	X	X	X	X	X	X	X	X	X
javax.swing.plaf.nimbus								X	X	X	X	X	X	X
javax.swing.plaf.synth						X	X	X	X	X	X	X	X	X
javax.swing.table			X	X	X	X	X	X	X	X	X	X	X	X

javax.swing.text			X	X	X	X	X	X	X	X	X	X	X	X
javax.swing.text.html			X	X	X	X	X	X	X	X	X	X	X	X
javax.swing.text.html.parser			X	X	X	X	X	X	X	X	X	X	X	X
javax.swing.text.rtf			X	X	X	X	X	X	X	X	X	X	X	X
javax.swing.tree Un composant de type arbre			X	X	X	X	X	X	X	X	X	X	X	X
javax.swing.undo Gérer les annulations d'opérations d'édition			X	X	X	X	X	X	X	X	X	X	X	X
javax.tools							X	X	X	X	X	X	X	X
javax.transaction				X	X	X	X	X	X	X	X			
javax.transaction.xa					X	X	X	X	X	X	X	X	X	X
javax.xml						X	X	X	X	X	X	X	X	X
javax.xml.bind							X	X	X	X	X			
javax.xml.bind.annotation							X	X	X	X	X			
javax.xml.bind.annotation.adapters							X	X	X	X	X			
javax.xml.bind.attachment							X	X	X	X	X			
javax.xml.bind.helpers							X	X	X	X	X			
javax.xml.bind.util							X	X	X	X	X			
javax.xml.catalog										X	X	X	X	X
javax.xml.crypto							X	X	X	X	X	X	X	X
javax.xml.crypto.dom							X	X	X	X	X	X	X	X
javax.xml.crypto.dsig							X	X	X	X	X	X	X	X
javax.xml.crypto.dsig.dom							X	X	X	X	X	X	X	X
javax.xml.crypto.dsig.keyinfo							X	X	X	X	X	X	X	X
javax.xml.crypto.dsig.spec							X	X	X	X	X	X	X	X
javax.xml.datatype						X	X	X	X	X	X	X	X	X
javax.xml.namespace						X	X	X	X	X	X	X	X	X
javax.xml.parsers API JAXP pour utiliser XML					X	X	X	X	X	X	X	X	X	X
javax.xml.soap							X	X	X	X	X			
javax.xml.stream							X	X	X	X	X	X	X	X
javax.xml.stream.events							X	X	X	X	X	X	X	X
javax.xml.stream.util							X	X	X	X	X	X	X	X
javax.xml.transform transformer un document XML avec XSLT					X	X	X	X	X	X	X	X	X	X
javax.xml.transform.dom					X	X	X	X	X	X	X	X	X	X
javax.xml.transform.sax					X	X	X	X	X	X	X	X	X	X
javax.xml.transform.stax							X	X	X	X	X	X	X	X
javax.xml.transform.stream					X	X	X	X	X	X	X	X	X	X
javax.xml.validation						X	X	X	X	X	X	X	X	X

javax.xml.ws							X	X	X	X	X			
javax.xml.ws.handler							X	X	X	X	X			
javax.xml.ws.handler.soap							X	X	X	X	X			
javax.xml.ws.http							X	X	X	X	X			
javax.xml.ws.soap							X	X	X	X	X			
javax.xml.ws.spi							X	X	X	X	X			
javax.xml.ws.spi.http							X	X	X	X	X			
javax.xml.ws.wsaddressing							X	X	X	X	X			
javax.xml.xpath						X	X	X	X	X	X	X	X	X
org.ietf.jgss					X	X	X	X	X	X	X	X	X	X
org.omg.CORBA			X	X	X	X	X	X	X	X	X			
org.omg.CORBA_2_3				X	X	X	X	X	X	X	X			
org.omg.CORBA_2_3.portable				X	X	X	X	X	X	X	X			
org.omg.CORBA.DynAnyPackage			X	X	X	X	X	X	X	X	X			
org.omg.CORBA.ORBPackage			X	X	X	X	X	X	X	X	X			
org.omg.CORBA.portable			X	X	X	X	X	X	X	X	X			
org.omg.CORBA.TypeCodePackage			X	X	X	X	X	X	X	X	X			
org.omg.CosNaming			X	X	X	X	X	X	X	X	X			
org.omg.CosNaming.NamingContextExtPackage			X	X	X	X	X	X	X	X	X			
org.omg.CosNaming.NamingContextPackage					X	X	X	X	X	X	X			
org.omg.Dynamic					X	X	X	X	X	X	X			
org.omg.DynamicAny					X	X	X	X	X	X	X			
org.omg.DynamicAny.DynAnyFactoryPackage					X	X	X	X	X	X	X			
org.omg.DynamicAny.DynAnyPackage					X	X	X	X	X	X	X			
org.omg.IOP					X	X	X	X	X	X	X			
org.omg.IOP.CodecFactoryPackage					X	X	X	X	X	X	X			
org.omg.IOP.CodecPackage					X	X	X	X	X	X	X			
org.omg.Messaging					X	X	X	X	X	X	X			
org.omg.PortableInterceptor					X	X	X	X	X	X	X			
org.omg.PortableInterceptor.ORBInitInfoPackage					X	X	X	X	X	X	X			
org.omg.PortableServer					X	X	X	X	X	X	X			
org.omg.PortableServer.CurrentPackage					X	X	X	X	X	X	X			
org.omg.PortableServer.POAManagerPackage					X	X	X	X	X	X	X			
org.omg.PortableServer.POAPackage					X	X	X	X	X	X	X			
org.omg.PortableServer.ServantLocatorPackage					X	X	X	X	X	X	X			
org.omg.PortableServer.portable					X	X	X	X	X	X	X			
org.omg.SendingContext				X	X	X	X	X	X	X	X			
org.omg.stub.java.rmi				X	X	X	X	X	X	X	X	X	X	X
					X	X	X	X	X	X	X	X	X	X

org.w3c.dom Utiliser DOM pour un document XML														
org.w3c.dom.bootstrap						X	X	X	X	X	X	X	X	X
org.w3c.dom.css										X	X	X	X	X
org.w3c.dom.events						X	X	X	X	X	X	X	X	X
org.w3c.dom.html										X	X	X	X	X
org.w3c.dom.ls						X	X	X	X	X	X	X	X	X
org.w3c.dom.ranges										X	X	X	X	X
org.w3c.dom.stylesheets										X	X	X	X	X
org.w3c.dom.traversal										X	X	X	X	X
org.w3c.dom.views									X	X	X	X	X	X
org.w3c.dom.xpath										X	X	X	X	X
org.xml.sax Utiliser SAX pour un document XML					X	X	X	X	X	X	X	X	X	X
org.xml.sax.ext					X	X	X	X	X	X	X	X	X	X
org.xml.sax.helpers					X	X	X	X	X	X	X	X	X	X

7.2. Le package java.lang

Ce package de base contient les classes fondamentales telles que Object, Class, Math, System, String, StringBuffer, Thread, les wrappers etc ... Certaines de ces classes sont détaillées dans les sections suivantes.

Il contient également plusieurs classes qui permettent de demander des actions au système d'exploitation sur lequel la machine virtuelle tourne, par exemple les classes ClassLoader, Runtime, SecurityManager.

Certaines classes sont détaillées dans des chapitres dédiés : la classe Math est détaillée dans le chapitre «Les fonctions mathématiques», la classe Class est détaillée dans le chapitre «La gestion dynamique des objets et l'introspection» et la classe Thread est détaillée dans le chapitre «Le multitâche».

Ce package est tellement fondamental qu'il est implicitement importé dans tous les fichiers sources par le compilateur.

7.2.1. La classe Object

C'est la super-classe de toutes les classes Java : toutes ses méthodes sont donc héritées par toutes les classes.

7.2.1.1. La méthode getClass()

La méthode getClass() renvoie un objet de la classe Class qui représente la classe de l'objet.

Le code suivant permet de connaitre le nom de la classe de l'objet

Exemple :

```
String nomClasse = monObject.getClass().getName();
```

7.2.1.2. La méthode toString()

La méthode toString() de la classe Object renvoie le nom de la classe , suivi du séparateur @, lui-même suivi par la valeur de hachage de l'objet.

7.2.1.3. La méthode equals()

La méthode equals() implémente une comparaison par défaut. Sa définition dans Object compare les références : donc obj1.equals(obj2) ne renverra true que si obj1 et obj2 désignent le même objet. Dans une sous-classe de Object, pour laquelle on a besoin de pouvoir dire que deux objets distincts peuvent être égaux, il faut redéfinir la méthode equals() héritée de Object.

7.2.1.4. La méthode finalize()

A l'inverse de nombreux langages orientés objet tels que le C++ ou Delphi, le programmeur Java n'a pas à se préoccuper de la destruction des objets qu'il instancie. Ceux-ci sont détruits et leur emplacement mémoire est récupéré par le ramasse-miettes de la machine virtuelle dès qu'il n'y a plus de référence sur l'objet.

La machine virtuelle garantit que toutes les ressources Java sont correctement libérées mais, quand un objet encapsule une ressource indépendante de Java (comme un fichier par exemple), il peut être préférable de s'assurer que la ressource sera libérée quand l'objet sera détruit. Pour cela, la classe Object définit la méthode protected finalize(), qui est appelée quand le ramasse-miettes doit récupérer l'emplacement de l'objet ou quand la machine virtuelle termine son exécution.

Exemple :

```java
import java.io.*;

public class AccesFichier {
  private FileWriter fichier;

  public AccesFichier(String s) {
    try {
      fichier = new FileWriter(s);
    }
    catch (IOException e) {
      System.out.println("Impossible d'ouvrir le fichier");
    }
  }

  protected void finalize() throws Throwable {
    super.finalize(); // obligatoire : appel finalize heritee
    System.out.println("Appel de la méthode finalize");
    termine();
  }

  public static void main(String[] args) {
    AccesFichier af = new AccesFichier("c:\test");
    System.exit(0);
  }

  public void termine() {
    if (fichier != null) {
      try {
        fichier.close();
      } catch (IOException e) {
        System.out.println("Impossible de fermer le fichier");
      }
      fichier = null;
    }
  }
}
```

Attention : selon l'implémentation du garbage collector dans la machine virtuelle, il n'est pas possible de prévoir le moment ou un objet sera traité par le garbage collector. De plus, l'appel du finaliseur n'est pas garanti : par exemple, si la machine virtuelle est brusquement arrêtée par l'utilisateur, le ramasse-miettes ne libérera pas la mémoire des objets en cours d'utilisation et les finaliseurs de ces objets ne seront pas appelés.

L'utilisation des finalizer n'est jamais été préconisée et il est même fortement déconseillée.

En Java 9, la méthode finalize() est dépréciée.

7.2.1.5. La méthode clone()

La section «Le clonage d'un objet» du chapitre «Les techniques de développement spécifiques à Java» détaille la mise en oeuvre de la méthode clone().

7.2.2. Les classes de manipulations de chaînes de caractères

Plusieurs classes sont proposées pour utiliser ou manipuler des chaînes de caractères notamment java.lang.String, java.util.StringBuffer et java.util.StringBuilder.

Le chapitre «Les chaînes de caractères» détaille la mise en oeuvre de ces classes.

7.2.3. Les wrappers

Les objet de type wrappers (enveloppes) représentent des objets qui encapsulent une donnée de type primitif et fournissent un ensemble de méthodes permettant notamment de faire des conversions.

Ces classes offrent toutes les services suivants :

- un constructeur qui permet une instanciation à partir du type primitif et un constructeur qui permet une instanciation à partir d'un objet String (dépréciés à partir de Java 9)
- des surcharges de fabrique valueOf() pour obtenir une instance
- une méthode pour fournir la valeur primitive représentée par l'objet
- une implémentation de la méthode equals() pour la comparaison.

Les méthodes de conversion opèrent sur des instances, mais il est possible d'utiliser des méthodes statiques.

Exemple :

```
int valeur = Integer.valueOf("999").intValue();
```

Jusqu'à Java 5 et l'introduction de l'autoboxing, ces classes ne sont pas interchangeables avec les types primitifs d'origine car il s'agit d'objets.

Exemple :

```
Float objetPi = new Float("3.1415");
System.out.println(5 * objetPi); // erreur à la compilation
```

Pour obtenir la valeur contenue dans l'objet, il faut utiliser la méthode xxxValue() où xxx est le nom du type standard.

Exemple :

```
Integer entier = new Integer("10");
```

```
int entier = entier.intValue();
```

Les classes Integer, Long, Float et Double définissent toutes les constantes MAX_VALUE et MIN_VALUE qui représentent leurs valeurs minimales et maximales.

Lorsque l'on effectue certaines opérations mathématiques sur des nombres à virgules flottantes (float ou double), le résultat peut prendre l'une des valeurs suivantes :

- NEGATIVE_INFINITY : infini négatif causé par la division d'un nombre négatif par 0.0
- POSITIVE_INFINITY : infini positif causé par la division d'un nombre positif par 0.0
- NaN: n'est pas un nombre (Not a Number) causé par la division de 0.0 par 0.0

Il existe des méthodes pour tester le résultat :

Float.isNaN(float); // pour les valeurs de type float
Double.isInfinite(double); // idem pour les valeurs de type double

Exemple :

```
float res = 5.0f / 0.0f;

if (Float.isInfinite(res)) { ... };
```

La constante Float.NaN n'est ni égale à un nombre dont la valeur est NaN ni à elle même. Float.NaN == Float.NaN retourne False

Lors de la division par zéro d'un nombre entier, une exception est levée.

Exemple :

```
System.out.println(10/0);

Exception in thread "main" java.lang.ArithmeticException: / by zero
at test9.main(test9.java:6)
```

7.2.4. La classe System

Cette classe possède de nombreuses fonctionnalités pour utiliser des services du système d'exploitation.

7.2.4.1. L'utilisation des flux d'entrée/sortie standard

La classe System définit trois variables statiques qui permettent d'utiliser les flux d'entrée/sortie standards du système d'exploitation.

Variable	Type	Rôle
in	InputStream	Entrée standard du système. Par défaut, c'est la console
out	PrintStream	Sortie standard du système. Par défaut, c'est la console
err	PrintStream	Sortie standard des erreurs du système. Par défaut, c'est la console

Exemple :

```
System.out.println("bonjour");
```

La classe System possède trois méthodes qui permettent de rediriger ces flux : setIn(InputStream), setOut(PrintStream) et setErr(PrintStream).

le mode de fonctionnement bien connu dans le langage C a été repris pour être ajouté dans l'API Java avec la méthode printf().

Exemple :

```
public class TestPrintf {
  public static void main(String[] args) {
    System.out.printf("%4d",32);
  }
}
```

La méthode printf() propose :

- un nombre d'arguments variable
- des formats standard pour les types primitifs, String et Date
- des justifications possibles avec certains formats
- l'utilisation de la localisation pour les données numériques et de type date

Exemple (java 1.5):

```
import java.util.*;

public class TestPrintf2 {

  public static void main(String[] args) {
    System.out.printf("%d \n"                     ,13);
    System.out.printf("%4d \n"                    ,13);
    System.out.printf("%04d \n"                   ,13);
    System.out.printf("%f \n"                     ,3.14116);
    System.out.printf("%.2f \n"                   ,3.14116);
    System.out.printf("%s \n"                     ,"Test");
    System.out.printf("%10s \n"                   ,"Test");
    System.out.printf("%-10s \n"                  ,"Test");
    System.out.printf("%tD \n"                    , new Date());
    System.out.printf("%tF \n"                    , new Date());
    System.out.printf("%1$te %1$tb %1$ty \n"       , new Date());
    System.out.printf("%1$tA %1$te %1$tB %1$tY \n", new Date());
    System.out.printf("%1$tr \n"                  , new Date());
  }
}
```

Résultat :

```
C:\tiger>java TestPrintf2
13
13
0013
3,141160
3,14
Test
Test
Test
08/23/04
2004-08-23
23 août 04
lundi 23 août 2004
03:56:25 PM
```

Une exception est levée lors de l'exécution si un des formats utilisés est inconnu.

Exemple (java 1.5):

```
C:\tiger>java TestPrintf2
13 1300133,1411603,14Test TestTest 08/23/04Exception in thread "main"
java.util.UnknownFormatConversionException: Conversion = 'tf'
at java.util.Formatter$FormatSpecifier.checkDateTime(Unknown Source)
at java.util.Formatter$FormatSpecifier.<init>(Unknown Source)
at java.util.Formatter.parse(Unknown Source)
at java.util.Formatter.format(Unknown Source)
at java.io.PrintStream.format(Unknown Source)
at java.io.PrintStream.printf(Unknown Source)
at TestPrintf2.main(TestPrintf2.java:15)
```

7.2.4.2. Les variables d'environnement et les propriétés du système

JDK 1.0 propose la méthode statique getenv() qui renvoie la valeur de la propriété système dont le nom est fourni en paramètre.

Depuis le JDK 1.1, cette méthode est deprecated car elle n'est pas très portable. Son utilisation lève une exception :

Exemple :

```
java.lang.Error: getenv no longer  supported, use properties and -D instead: windir
      at  java.lang.System.getenv(System.java:691)
      at  fr.jmdoudoux.dej.TestPropertyEnv.main(TestPropertyEnv.java:6)
Exception  in thread "main"
```

Elle est remplacée par un autre mécanisme qui n'interroge pas directement le système mais qui recherche les valeurs dans un ensemble de propriétés. Cet ensemble est constitué de propriétés standard fournies par l'environnement Java et par des propriétés ajoutées par l'utilisateur. Jusqu'au JDK 1.4, il est nécessaire d'utiliser ces propriétés de la JVM.

Voici une liste non exhaustive des propriétés fournies par l'environnement Java :

Nom de la propriété	Rôle
java.version	Version du JRE
java.vendor	Auteur du JRE
java.vendor.url	URL de l'auteur
java.home	Répertoire d'installation de java
java.vm.version	Version de l'implémentation de la JVM
java.vm.vendor	Auteur de l'implémentation de la JVM
java.vm.name	Nom de l'implémentation de la JVM
java.specification.version	Version des spécifications de la JVM
java.specification.vendor	Auteur des spécifications de la JVM
java.specification.name	Nom des spécifications de la JVM
java.ext.dirs	Chemin du ou des répertoires d'extension
os.name	Nom du système d'exploitation
os.arch	Architecture du système d'exploitation
os.version	Version du système d'exploitation
file.separator	Séparateur de fichiers (exemple : "/" sous Unix, "\" sous Windows)
path.separator	Séparateur de chemin (exemple : ":" sous Unix, ";" sous Windows)
line.separator	Séparateur de lignes
user.name	Nom de l'utilisateur courant

user.home	Répertoire d'accueil du user courant
user.dir	Répertoire courant au moment de l'initialisation de la propriété

Exemple :

```
public class TestProperty {

  public static void main(String[] args) {
    System.out.println("java.version                ="+System.getProperty("java.version"));
    System.out.println("java.vendor                 ="+System.getProperty("java.vendor"));
    System.out.println("java.vendor.url             ="+System.getProperty("java.vendor.url"));
    System.out.println("java.home                   ="+System.getProperty("java.home"));
    System.out.println("java.vm.specification.version ="
      +System.getProperty("java.vm.specification.version"));
    System.out.println("java.vm.specification.vendor  ="
      +System.getProperty("java.vm.specification.vendor"));
    System.out.println("java.vm.specification.name  ="
      +System.getProperty("java.vm.specification.name"));
    System.out.println("java.vm.version             ="+System.getProperty("java.vm.version"));
    System.out.println("java.vm.vendor              ="+System.getProperty("java.vm.vendor"));
    System.out.println("java.vm.name                ="+System.getProperty("java.vm.name"));
    System.out.println("java.specification.version  ="
      +System.getProperty("java.specification.version"));
    System.out.println("java.specification.vendor   ="
      +System.getProperty("java.specification.vendor"));
    System.out.println("java.specification.name     ="
      +System.getProperty("java.specification.name"));
    System.out.println("java.class.version          ="
      +System.getProperty("java.class.version"));
    System.out.println("java.class.path             ="
      +System.getProperty("java.class.path"));
    System.out.println("java.ext.dirs               ="+System.getProperty("java.ext.dirs"));
    System.out.println("os.name                     ="+System.getProperty("os.name"));
    System.out.println("os.arch                     ="+System.getProperty("os.arch"));
    System.out.println("os.version                  ="+System.getProperty("os.version"));
    System.out.println("file.separator              ="+System.getProperty("file.separator"));
    System.out.println("path.separator              ="+System.getProperty("path.separator"));
    System.out.println("line.separator              ="+System.getProperty("line.separator"));
    System.out.println("user.name                   ="+System.getProperty("user.name"));
    System.out.println("user.home                   ="+System.getProperty("user.home"));
    System.out.println("user.dir                    ="+System.getProperty("user.dir"));
  }
}
```

Par défaut, l'accès aux propriétés système est restreint par le SecurityManager pour les applets.

Pour définir ses propres propriétés, il faut utiliser l'option -D de l'interpréteur Java sur la ligne de commandes.

La méthode statique getProperty() permet d'obtenir la valeur de la propriété dont le nom est fourni en paramètre. Une version surchargée de cette méthode permet de préciser un second paramètre qui contiendra la valeur par défaut, si la propriété n'est pas définie.

Exemple : obtenir une variable système (java 1.1, 1.2, 1.3 et 1.4)

```
package fr.jmdoudoux.dej;

public class TestPropertyEnv {

  public static void main(String[] args) {
    System.out.println("env.windir ="+System.getProperty("env.windir"));
  }
}
```

Exemple : Execution

```
C:\java>java -Denv.windir=%windir% -cp . fr.jmdoudoux.dej.TestPropertyEnv
env.windir =C:\WINDOWS
```

Java 5 propose de nouveau une implémentation pour la méthode System.getenv() possédant deux surcharges :

- Une sans paramètre qui renvoie une collection des variables système
- Une avec un paramètre de type String qui contient le nom de la variable à obtenir

Exemple (Java 5):

```
package fr.jmdoudoux.dej;

public class TestPropertyEnv {

  public static void main(String[] args) {
    System.out.println(System.getenv("windir"));
  }
}
```

La surcharge sans argument permet d'obtenir une collection de type Map contenant les variables d'environnement système.

Exemple (Java 5) :

```
package fr.jmdoudoux.dej;

import java.util.Iterator;
import java.util.Map;
import java.util.Set;

public class TestPropertyEnv {

  public static void main(String[] args) {
    Map map = System.getenv();
    Set cles = map.keySet();
    Iterator iterator = cles.iterator();
    while (iterator.hasNext()) {
      String cle = (String) iterator.next();
      System.out.println(cle+" : "+map.get(cle));
    }
  }
}
```

7.2.5. Les classes Runtime et Process

La classe Runtime permet d'interagir avec le système dans lequel l'application s'exécute : obtenir des informations sur le système, arrêter de la machine virtuelle, exécuter un programme externe.

Cette classe ne peut pas être instanciée mais il est possible d'obtenir une instance en appelant la méthode statique getRuntime() de la classe RunTime.

Les méthodes totalMemory() et freeMemory() permettent d'obtenir respectivement la quantité totale de la mémoire et la quantité de mémoire libre.

Exemple :

```
package fr.jmdoudoux.dej;

public class TestRuntime1 {

    public static void main(String[] args) {
        Runtime runtime = Runtime.getRuntime();
        System.out.println("Mémoire totale = " + runtime.totalMemory());
        System.out.println("Memoire libre = " + runtime.freeMemory());
    }
}
```

La méthode exec() permet d'exécuter des processus sur le système d'exploitation où s'exécute la JVM. Elle lance la commande de manière asynchrone et renvoie un objet de type Process pour obtenir des informations sur le processus lancé.

Il existe plusieurs surcharges de cette méthode pouvant toutes, entre autres, lever une exception de type SecurityException, IOException, NullPointerException :

Méthode	Remarque
Process exec(String command)	
Process exec(String[] cmdarray)	
Process exec(String[] cmdarray, String[] envp)	
Process exec(String[] cmdarray, String[] envp, File dir)	(depuis Java 1.3)
Process exec(String cmd, String[] envp)	
Process exec(String command, String[] envp, File dir)	(depuis Java 1.3)

La commande à exécuter peut être fournie sous la forme d'une chaîne de caractères ou sous la forme d'un tableau dont le premier élément est la commande et les éléments suivants sont ses arguments. Deux des surcharges acceptent un objet de type File qui encapsule le répertoire dans lequel la commande va être exécutée.

Important : la commande exec() n'est pas un interpréteur de commandes. Il n'est par exemple pas possible de préciser dans la commande une redirection vers un fichier. Ainsi pour exécuter une commande de l'interpréteur DOS sous Windows, il est nécessaire de préciser l'interpréteur de commandes à utiliser (command.com sous Windows 95 ou cmd.exe sous Windows 2000 et XP).

Remarque : avec l'interpréteur de commandes cmd.exe, il est nécessaire d'utiliser l'option /c qui permet de demander de quitter l'interpréteur à la fin de l'exécution de la commande.

L'inconvénient d'utiliser cette méthode est que la commande exécutée est dépendante du système d'exploitation.

La classe abstraite Process encapsule un processus : son implémentation est fournie par la JVM puisqu'elle est dépendante du système.

Les méthodes getOutputStream(), getInputStream() et getErrorStream() permettent d'avoir un accès respectivement au flux de sortie, d'entrée et d'erreur du processus.

La méthode waitFor() permet d'attendre la fin du processus.

La méthode exitValue() permet d'obtenir le code de retour du processus. Elle lève une exception de type IllegalThreadStateException si le processus n'est pas terminé.

La méthode destroy() permet de détruire le processus.

Exemple :

```
package fr.jmdoudoux.dej;

import java.io.BufferedReader;
import java.io.InputStreamReader;

public class TestRuntime2 {
  public static void main(String[] args) {
    try {
      Process proc =
        Runtime.getRuntime().exec("cmd.exe /c set");
      BufferedReader in =
        new BufferedReader(new InputStreamReader(proc.getInputStream()));
      String str;
      while ((str = in.readLine()) != null) {
        System.out.println(str);
      }
      in.close();
      proc.waitFor();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Le code ci-dessus est fourni à titre d'exemple mais il n'est pas la solution idéale même s'il fonctionne. Il est préférable de traiter les flux dans un thread dédié.

Exemple :

```
package fr.jmdoudoux.dej;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;

public class TestRuntime3 {

  public TestRuntime3() {
    try {

      Runtime runtime = Runtime.getRuntime();
      Process proc = runtime.exec("cmd.exe /c set");

      TestRuntime3.AfficheFlux afficheFlux =
        new AfficheFlux(proc.getInputStream());

      afficheFlux.start();

      int exitVal = proc.waitFor();
      System.out.println("exitVal = " + exitVal);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public static void main(String[] args) {
    new TestRuntime3();
  }

  private class AfficheFlux extends Thread {
    InputStream is;

    AfficheFlux(InputStream is) {
      this.is = is;
    }

    public void run() {
      try {
        InputStreamReader isr = new InputStreamReader(is);
        BufferedReader br = new BufferedReader(isr);
        String line = null;
        while ((line = br.readLine()) != null)
          System.out.println(line);
      } catch (IOException ioe) {
        ioe.printStackTrace();
      }
    }
  }
}
```

Sous Windows, il est possible d'utiliser un fichier dont l'extension est associée à une application.

Exemple :

```
Process proc = Runtime.getRuntime().exec("cmd.exe /c \"c:\\test.doc\"");
```

7.3. La présentation rapide du package awt java

AWT est une collection de classes pour la réalisation d'applications graphiques ou GUI (Graphic User Interface)

Les composants qui sont utilisés par les classes définies dans ce package sont des composants dit "lourds" : ils dépendent entièrement du système d'exploitation. D'ailleurs leur nombre est limité car ils sont communs à plusieurs systèmes d'exploitation pour assurer la portabilité. Cependant, la représentation d'une interface graphique avec awt sur plusieurs systèmes peut ne pas être identique.

AWT se compose de plusieurs packages dont les principaux sont:

- java.awt : c'est le package de base de la bibliothèque AWT
- java.awt.images : ce package permet la gestion des images
- java.awt.event : ce package permet la gestion des événements utilisateurs
- java.awt.font : ce package permet d'utiliser les polices de caractères
- java.axt.dnd : ce package permet l'utilisation du cliquer/glisser

Le chapitre «La création d'interfaces graphiques avec AWT» détaille l'utilisation de ce package.

7.4. La présentation rapide du package java.io

Ce package définit un ensemble de classes pour la gestion des flux d'entrées-sorties.

Le chapitre «Les flux d'entrée/sortie» détaille l'utilisation de ce package.

7.5. Le package java.util

Ce package contient un ensemble de classes utilitaires : la gestion des dates (Date et Calendar), la génération de nombres aléatoires (Random), la gestion des collections ordonnées ou non telles que la table de hachage (HashTable), le vecteur (Vector), la pile (Stack) ..., la gestion des propriétés (Properties), des classes dédiées à l'internationalisation (ResourceBundle, PropertyResourceBundle, ListResourceBundle) etc ...

Certaines de ces classes sont présentées plus en détail dans les sections suivantes.

7.5.1. La classe Random

La classe Random permet de générer des nombres pseudo-aléatoires. Après l'appel au constructeur, il suffit d'appeler la méthode correspondant au type désiré : nextInt(), nextLong(), nextFloat() ou nextDouble()

Méthodes	valeur de retour
nextInt()	entre Integer.MIN_VALUE et Interger.MAX_VALUE
nextLong()	entre long.MIN_VALUE et long.MAX_VALUE
nextFloat() ou nextDouble()	entre 0.0 et 1.0

Exemple (code Java 1.1) :

```
import java.util.*;
class test9 {
  public static void main (String args[]) {
    Random r = new Random();
    int a = r.nextInt() %10; //entier entre -9 et 9
    System.out.println("a = "+a);
  }
}
```

7.5.2. Les classes Date et Calendar

En Java 1.0, la classe Date permet de manipuler les dates.

Exemple (code Java 1.0) :

```
import java.util.*;
...
    Date maintenant = new Date();
    if (maintenant.getDay() == 1)
       System.out.println(" lundi ");
```

Le constructeur d'un objet Date l'initialise avec la date et l'heure courante du système.

Exemple (code Java 1.0) :

```
import java.util.*;
import java.text.*;

public class TestHeure {
  public static void main(java.lang.String[] args) {
    Date date = new Date();
    System.out.println(DateFormat.getTimeInstance().format(date));
  }
}
```

Résultat :

```
22:05:21
```

La méthode getTime() permet de calculer le nombre de millisecondes écoulées entre la date qui est encapsulée dans l'objet qui reçoit le message getTime et le premier janvier 1970 à 0 heure GMT.

En Java 1.1, de nombreuses méthodes et constructeurs de la classe Date sont deprecated, notamment celles qui permettent de manipuler les éléments qui composent la date et leur formatage : il faut utiliser la classe Calendar.

Exemple (code Java 1.1) :

```
import java.util.*;

public class TestCalendar {
  public static void main(java.lang.String[] args) {

    Calendar c = Calendar.getInstance();
    if (c.get(Calendar.DAY_OF_WEEK) == Calendar.MONDAY)
      System.out.println(" nous sommes lundi ");
    if (c.get(Calendar.DAY_OF_WEEK) == Calendar.TUESDAY)
      System.out.println(" nous sommes mardi ");
    if (c.get(Calendar.DAY_OF_WEEK) == Calendar.WEDNESDAY)
      System.out.println(" nous sommes mercredi ");
    if (c.get(Calendar.DAY_OF_WEEK) == Calendar.THURSDAY)
      System.out.println(" nous sommes jeudi ");
    if (c.get(Calendar.DAY_OF_WEEK) == Calendar.FRIDAY)
      System.out.println(" nous sommes vendredi ");
    if (c.get(Calendar.DAY_OF_WEEK) == Calendar.SATURDAY)
      System.out.println(" nous sommes samedi ");
    if (c.get(Calendar.DAY_OF_WEEK) == Calendar.SUNDAY)
      System.out.println(" nous sommes dimanche ");
  }
}
```

Résultat :

```
 nous sommes lundi
```

La mise en oeuvre détaillée de ces classes est proposée dans le chapitre «L'utilisation des dates»

7.5.3. La classe SimpleDateFormat

La classe SimpleDateFormat est la seule implémentation de la classe DateFormat fournie en standard.

Elle utilise une syntaxe particulière pour spécifier le format de la date à utiliser pour le formatage ou le parsing. Ce format est fourni par le constructeur ou en invoquant la méthode applyPattern().

Exemple :

```
final SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy HH:mm:ss");
final String dateStr = sdf.format(new Date());
System.out.println(dateStr);
try {
  final Date date = sdf.parse(dateStr);
  System.out.println(date);
}
catch (final ParseException pe) {
  pe.printStackTrace();
}
```

La méthode parse() peut lever une exception de type ParseException si l'extraction de la date selon le format et la date fournis échoue.

La mise en oeuvre détaillée de cette classe est incluse dans le chapitre «L'utilisation des dates»

7.5.4. La classe Vector

Un objet de la classe Vector peut être considéré comme une tableau évolué qui peut contenir un nombre indéterminé d'objets.

Les méthodes principales sont les suivantes :

Méthode	Rôle
void addElement(Object)	ajouter un objet dans le vecteur
boolean contains(Object)	retourne true si l'objet est dans le vecteur
Object elementAt(int)	retourne l'objet à l'index indiqué
Enumeration elements()	retourne une énumeration contenant tous les éléments du vecteur
Object firstElement()	retourne le premier élément du vecteur (celui dont l'index est égal à zéro)
int indexOf(Object)	renvoie le rang de l'élément ou -1
void insertElementAt(Object, int)	insérer un objet à l'index indiqué

boolean isEmpty()	retourne un booléen si le vecteur est vide
Objet lastElement()	retourne le dernier élément du vecteur
void removeAllElements()	vider le vecteur
void removeElement(Object)	supprime l'objet du vecteur
void removeElementAt(int)	supprime l'objet à l'index indiqué
void setElementAt(object, int)	remplacer l'élément à l'index par l'objet
int size()	nombre d'objets du vecteur

On peut stocker des objets de classes différentes dans un vecteur mais les éléments stockés doivent obligatoirement être des objets (Avant Java 5, pour les types primitifs, il faut utiliser les wrappers tels que Integer ou Float respectivement pour des données de type int ou float).

Exemple (code Java 1.1) :

```
Vector v = new Vector();
v.addElement(new Integer(10));
v.addElement(new Float(3.1416));
v.insertElementAt("chaine ",1);
System.out.println(" le vecteur contient "+v.size()+ " elements ");
String retrouve = (String) v.elementAt(1);
System.out.println(" le 1er element = "+retrouve);

C:\$user\java>java test9
le vecteur contient 3 elements
le 1er element = chaine
```

Exemple (code Java 1.1) :

```
   Vector v = new Vector();
   ...

   for (int i = 0; i < v.size() ; i ++) {
      System.out.println(v.elementAt(i));
   }
```

Il est aussi possible de parcourir l'ensemble des éléments en utilisant une instance de l'interface Enumeration.

Remarque : A partir de Java 1.2, il est préférable d'utiliser une classe de «Les collections».

7.5.5. La classe Hashtable

Les informations d'une Hastable sont stockées sous la forme clé - données. Cet objet peut être considéré comme un dictionnaire.

Exemple (code Java 1.1) :

```
Hashtable dico = new Hashtable();
dico.put("livre1", " titre du livre 1 ");
dico.put("livre2", "titre du livre 2 ");
```

Il est possible d'utiliser n'importe quel objet comme clé et comme donnée.

Exemple (code Java 1.1) :

```
dico.put("jour", new Date());
dico.put(new Integer(1),"premier");
dico.put(new Integer(2),"deuxième");
```

Pour lire dans la table, on utilise get(object) en donnant la clé en paramètre.

Exemple (code Java 1.1) :

```
System.out.println(" nous sommes le " +dico.get("jour"));
```

La méthode remove(Object) permet de supprimer une entrée du dictionnaire correspondant à la clé passée en paramètre.

La méthode size() permet de connaitre le nombre d'associations du dictionnaire.

Remarque : A partir de Java 1.2, il est préférable d'utiliser une classe de «Les collections».

7.5.6. L'interface Enumeration

L'interface Enumeration est utilisée pour permettre le parcours séquentiel de collections.

Enumeration est une interface qui définit 2 méthodes :

Méthodes	Rôle
boolean hasMoreElements()	retourne true si l'énumération contient encore un ou plusieurs éléments
Object nextElement()	retourne l'objet suivant de l'énumération Elle lève une Exception NoSuchElementException si la fin de la collection est atteinte.

Exemple (code Java 1.1) : contenu d'un vecteur et liste des clés d'une Hastable

```
import java.util.*;

class test9 {

   public static void main (String args[]) {

      Hashtable h = new Hashtable();
      Vector v = new Vector();

      v.add("chaine 1");
      v.add("chaine 2");
      v.add("chaine 3");

      h.put("jour", new Date());
      h.put(new Integer(1),"premier");
      h.put(new Integer(2),"deuxième");

      System.out.println("Contenu du vector");

      for (Enumeration e = v.elements() ; e.hasMoreElements() ; ) {
         System.out.println(e.nextElement());
      }

      System.out.println("\nContenu de la hashtable");

      for (Enumeration e = h.keys() ; e.hasMoreElements() ; ) {
         System.out.println(e.nextElement());
      }
   }
}

C:\$user\java>java test9
```

```
Contenu du vector
chaine 1
chaine 2
chaine 3
Contenu de la hashtable
jour
2
1
```

7.5.7. La manipulation d'archives zip

Depuis sa version 1.1, le JDK propose des classes permettant la manipulation d'archives au format zip. Ce format de compression est utilisé par Java lui-même notamment pour les fichiers de packaging (jar, war, ear ...).

Ces classes sont regroupées dans le package java.util.zip. Elles permettent de manipuler les archives aux formats zip et Gzip et d'utiliser des sommes de contrôles selon les algorithmes Adler-32 et CRC-32.

La classe ZipFile encapsule une archive au format zip : elle permet de manipuler les entrées qui composent l'archive.

Elle possède trois constructeurs :

Constructeur	Rôle
ZipFile(File)	ouvre l'archive correspondant au fichier fourni en paramètre
ZipFile(File, int)	ouvre l'archive correspondant au fichier fourni en paramètre selon le mode précisé : OPEN_READ ou OPEN_READ | OPEN_DELETE
ZipFile(string)	ouvre l'archive dont le nom de fichier est fourni en paramètre

Exemple :

```
   try {
    ZipFile test = new ZipFile(new File("C:/test.zip"));
   }  catch (ZipException e) {
    e.printStackTrace();
   }  catch (IOException e) {
    e.printStackTrace();
   }
```

Cette classe possède plusieurs méthodes dont les principales sont :

Méthode	Rôle
Enumeration entries()	Obtenir une énumération des entrées de l'archive sous la forme d'objets de type ZipEntry
close()	Fermer l'archive
ZipEntry getEntry(String)	Renvoie l'entrée dont le nom est précisé en paramètre
InputStream getInputStream(ZipEntry)	Renvoie un flux de lecture pour l'entrée précisée

La classe ZipEntry encapsule une entrée dans l'archive zip. Une entrée correspond à un fichier avec des informations le concernant dans l'archive.

Cette classe possède plusieurs méthodes dont les principales sont :

Méthode	Rôle

getName()	Renvoie le nom de l'entrée (nom du fichier avec sa sous-arborescence dans l'archive)
getTime() / setTime()	Renvoie ou modifie la date de modification de l'entrée
getComment() / setComment()	Renvoie ou modifie le commentaire associé à l'entrée
getSize() / setSize()	Renvoie ou modifie la taille de l'entrée non compressée
getCompressedSize() / setCompressedSize()	Renvoie ou modifie la taille de l'entrée compressée
getCrc() / setCrc()	Renvoie ou modifie la somme de contrôle permettant de vérifier l'intégrité de l'entrée
getMethod() / setMethod()	Renvoie ou modifie la méthode utilisée pour la compression
isDirectory()	Renvoie un booléen précisant si l'entrée est un répertoire

Exemple : afficher le contenu d'une archive

```
  public static void listerZip(String nomFichier) {
    ZipFile zipFile;
    try {
      zipFile = new ZipFile(nomFichier);
      Enumeration entries = zipFile.entries();
      while (entries.hasMoreElements()) {
        ZipEntry entry = (ZipEntry) entries.nextElement();
        String name = entry.getName();
        System.out.println(name);
      }
      zipFile.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
```

La classe ZipOutputStream est un flux qui permet l'écriture de données dans l'archive.

Cette classe possède plusieurs méthodes dont les principales sont :

Méthode	Rôle
setMethod()	Modifier la méthode de compression utilisée par défaut. Les valeurs possibles sont STORED (aucune compression) ou DEFLATED (avec compression)
setLevel()	Modifier le taux de compression : les valeurs entières possibles vont de 0 à 9 où 9 correspond au taux de compression le plus élevé. Des constantes sont définies dans la classe Deflater : Deflater.BEST_COMPRESSION, Deflater.DEFAULT_COMPRESSION, Deflater.BEST_SPEED, Deflater.NO_COMPRESSION
putNextEntry(ZipEntry)	Permet de se positionner dans l'archive pour ajouter l'entrée fournie en paramètre
write(byte[] b, int off, int len)	Permet d'écrire un tableau d'octets dans l'entrée courante
closeEntry()	Fermer l'entrée courante et se positionne pour ajouter l'entrée suivante
close()	Fermer le flux

Exemple : compresser un fichier dans une archive

```
  public static void compresser(String nomArchive, String nomFichier) {
    try {
      ZipOutputStream zip = new ZipOutputStream(
          new FileOutputStream(nomArchive));
      zip.setMethod(ZipOutputStream.DEFLATED);
      zip.setLevel(Deflater.BEST_COMPRESSION);
      // lecture du fichier
      File fichier = new File(nomFichier);
      FileInputStream fis = new FileInputStream(fichier);
      byte[] bytes = new byte[fis.available()];
      fis.read(bytes);

      // ajout d'une nouvelle entrée dans l'archive contenant le fichier
      ZipEntry entry = new ZipEntry(nomFichier);
      entry.setTime(fichier.lastModified());
      zip.putNextEntry(entry);
      zip.write(bytes);

      // fermeture des flux
      zip.closeEntry();
      fis.close();
      zip.close();
    } catch (FileNotFoundException fnfe) {
      fnfe.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }
  }
```

La classe ZipInputStream est un flux qui permet la lecture de données dans l'archive.

Cette classe possède plusieurs méthodes dont les principales sont :

Méthode	Rôle
getNextEntry()	Permet de se positionner sur l'entrée suivante dans l'archive
read(byte[] b, int off, int len)	permet de lire un tableau d'octets dans l'entrée courante
close()	permet de fermer le flux

Exemple :

```
public static void decompresser(String nomArchive, String chemin) {
  try {

    ZipFile zipFile = new ZipFile(nomArchive);
    Enumeration entries = zipFile.entries();
    ZipEntry entry = null;
    File fichier = null;
    File sousRep = null;

    while (entries.hasMoreElements()) {
      entry = (ZipEntry) entries.nextElement();

      if (!entry.isDirectory()) {
        System.out.println("Extraction du fichier " + entry.getName());
        fichier = new File(chemin + File.separatorChar + entry.getName());
        sousRep = fichier.getParentFile();

        if (sousRep != null) {
          if (!sousRep.exists()) {
            sousRep.mkdirs();
          }
        }

        int i = 0;
        byte[] bytes = new byte[1024];
        BufferedOutputStream out = new BufferedOutputStream(
            new FileOutputStream(fichier));
        BufferedInputStream in = new BufferedInputStream(zipFile
            .getInputStream(entry));
        while ((i = in.read(bytes)) != -1)
          out.write(
              bytes,
              0,
              i);

        in.close();
        out.flush();
        out.close();
      }
    }
    zipFile.close();
  } catch (FileNotFoundException fnfe) {
    fnfe.printStackTrace();
  } catch (IOException ioe) {
    ioe.printStackTrace();
  }
}
```

7.5.8. Les expressions régulières

Le JDK 1.4 propose une API en standard pour utiliser les expressions régulières. Les expressions régulières permettent de comparer une chaîne de caractères à un motif pour vérifier s'il y a concordance.

La package java.util.regexp contient deux classes et une exception pour gérer les expressions régulières :

Classe	Rôle
Matcher	comparer une chaîne de caractères avec un motif
Pattern	encapsule une version compilée d'un motif
PatternSyntaxException	exception levée lorsque le motif contient une erreur de syntaxe

7.5.8.1. Les motifs

Les expressions régulières utilisent un motif. Ce motif est une chaîne de caractères qui contient des caractères et des méta caractères. Les méta caractères ont une signification particulière et sont interprétés.

Il est possible de déspécialiser un méta caractère (lui enlever sa signification particulière) en le faisant précéder d'un caractère backslash. Ainsi pour utiliser le caractère backslash, il faut le doubler.

Les méta caractères reconnus par l'api sont :

méta caractères	rôle
()	créer des groupes
[]	définir un ensemble de caractères
{}	définir une répétition du motif précédent
\	déspécialisation du caractère qui suit
^	début de la ligne
$	fin de la ligne
|	le motif précédent ou le motif suivant
?	motif précédent répété zéro ou une fois
*	motif précédent répété zéro ou plusieurs fois
+	motif précédent répété une ou plusieurs fois
.	un caractère quelconque

Certains caractères spéciaux ont une notation particulière :

Notation	Rôle
\t	tabulation
\n	nouvelle ligne (ligne feed)
\\	backslash

Il est possible de définir des ensembles de caractères à l'aide des caractères [et]. Il suffit d'indiquer les caractères de l'ensemble entre ces deux crochets.

Exemple : toutes les voyelles
`[aeiouy]`

Il est possible d'utiliser une plage de caractères consécutifs en séparant le caractère de début de la plage et le caractère de fin de la plage avec un caractère -.

Exemple : toutes les lettres minuscules
`[a-z]`

L'ensemble peut être l'union de plusieurs plages.

Exemple : toutes les lettres
`[a-zA-Z]`

Par défaut l'ensemble [] désigne tous les caractères. Il est possible de définir un ensemble de la forme tous sauf ceux précisés en utilisant le caractère ^ suivi des caractères à enlever de l'ensemble

Exemple : tous les caractères sauf les lettres
`[^a-zA-Z]`

Il existe plusieurs ensembles de caractères prédéfinis :

Notation	Contenu de l'ensemble
\d	un chiffre
\D	tous sauf un chiffre
\w	une lettre ou un underscore
\W	tous sauf une lettre ou un underscore
\s	un séparateur (espace, tabulation, retour chariot, ...)
\S	tous sauf un séparateur

Plusieurs méta caractères permettent de préciser un critère de répétition d'un motif

méta caractères	rôle
{n}	répétition du motif précédent n fois

{n,m}	répétition du motif précédent entre n et m fois
{n,}	répétition du motif précédent
?	motif précédent répété zéro ou une fois
*	motif précédent repété zéro ou plusieurs fois
+	motif précédent répété une ou plusieurs fois

Exemple : la chaîne AAAAA

```
A{5}
```

7.5.8.2. La classe Pattern

Cette classe encapsule une représentation compilée d'un motif d'une expression régulière.

La classe Pattern ne possède pas de constructeur public mais propose une méthode statique compile().

Exemple :

```
      private static Pattern motif = null;
      ...
      motif = Pattern.compile("liste[0-9]");
```

Une version surchargée de la méthode compile() permet de préciser certaines options dont la plus intéressante permet de rendre insensible à la casse les traitements en utilisant le flag CASE_INSENSITIVE.

Exemple :

```
      private static Pattern motif = null;
      ...
      motif = Pattern.compile("liste[0-9]",Pattern.CASE_INSENSITIVE);
```

Cette méthode compile() renvoie une instance de la classe Pattern si le motif est syntaxiquement correct sinon elle lève une exception de type PatternSyntaxException.

La méthode matches(String, String) permet de rapidement et facilement utiliser les expressions régulières avec un seul appel de méthode en fournissant le motif et la chaîne à traiter.

Exemple :

```
      if (Pattern.matches("liste[0-9]","liste2")) {
            System.out.println("liste2 ok");
      } else {
            System.out.println("liste2 ko");
      }
```

7.5.8.3. La classe Matcher

La classe Matcher est utilisée pour effectuer la comparaison entre une chaîne de caractères et un motif encapsulé dans un objet de type Pattern.

Cette classe ne possède aucun constructeur public. Pour obtenir une instance de cette classe, il faut utiliser la méthode matcher() d'une instance d'un objet Pattern en lui fournissant la chaîne à traiter en paramètre.

Exemple :

```
      motif = Pattern.compile("liste[0-9]");
      matcher = motif.matcher("liste1");
```

Exemple :

```
      motif = Pattern.compile("liste[0-9]");
      matcher = motif.matcher("liste1");
      if (matcher.matches()) {
            System.out.println("liste1 ok");
      } else {
            System.out.println("liste1 ko");
      }
      matcher = motif.matcher("liste10");
      if (matcher.matches()) {
            System.out.println("liste10 ok");
      } else {
            System.out.println("liste10 ko");
      }
```

Résultat :

```
liste1 ok
liste10 ko
```

La méthode lookingAt() tente de rechercher le motif dans la chaîne à traiter.

Exemple :

```
            motif = Pattern.compile("liste[0-9]");
            matcher = motif.matcher("liste1");
            if (matcher.lookingAt()) {
                  System.out.println("liste1 ok");
            } else {
                  System.out.println("liste1 ko");
            }
            matcher = motif.matcher("liste10");
            if (matcher.lookingAt()) {
                  System.out.println("liste10 ok");
            } else {
                  System.out.println("liste10 ko");
            }
```

Résultat :

```
 liste1 ok
 liste10 ok
```

La méthode find() permet d'obtenir des informations sur chaque occurrence où le motif est trouvé dans la chaîne à traiter.

Exemple :

```
      matcher = motif.matcher("zzliste1zz");
      if (matcher.find()) {
            System.out.println("zzliste1zz ok");
      } else {
           System.out.println("zzliste1zz ko");
      }
```

Résultat :

```
zzliste1zz ok
```

Il est possible d'appeler successivement cette méthode pour obtenir chacune des occurrences.

Exemple :

```
        int i = 0;
        motif = Pattern.compile("liste[0-9]");
        matcher = motif.matcher("liste1liste2liste3");
        while (matcher.find()) {
              i++;
        }
        System.out.println("nb occurrences = " + i);
```

Les méthodes start() et end() permettent de connaitre respectivement la position de début et de fin de la chaîne dans l'occurrence en cours de traitement.

Exemple :

```
        int i = 0;
        motif = Pattern.compile("liste[0-9]");
        matcher = motif.matcher("liste1liste2liste3");
        while (matcher.find()) {
            System.out.print("pos debut : "+matcher.start());
            System.out.println(" pos fin : "+matcher.end());
            i++;
        }
        System.out.println("nb occurrences = " + i);
```

Résultat :

```
  pos debut : 0 pos fin : 6
  pos debut : 6 pos fin : 12
  pos debut : 12 pos fin : 18
  nb occurences = 3
```

La classe Matcher propose aussi les méthodes replaceFirst() et replaceAll() pour facilement remplacer la première ou toutes les occurrences du motif trouvé par une chaîne de caractères.

Exemple : remplacement de la première occurrence

```
        motif = Pattern.compile("liste[0-9]");
        matcher = motif.matcher("zz liste1 zz liste2 zz");
        System.out.println(matcher.replaceFirst("chaine"));
```

Résultat :

```
zz chaine zz liste2 zz
```

Exemple : remplacement de toutes les occurrences

```
      motif = Pattern.compile("liste[0-9]");
      matcher = motif.matcher("zz liste1 zz liste2 zz");
      System.out.println(matcher.replaceAll("chaine"));
```

Résultat :

```
zz chaine zz chaine zz
```

7.5.9. La classe Formatter

La méthode printf() utilise la classe Formatter pour réaliser le formatage des données fournies selon leurs valeurs et le format donné en paramètre.

Cette classe peut aussi être utilisée pour formater des données pour des fichiers ou dans une servlet par exemple.

La méthode format() attend en paramètre une chaîne de caractères qui précise le format des données à formater.

Exemple (java 1.5) :

```
import java.util.*;

public class TestFormatter {

  public static void main(String[] args) {
    Formatter formatter = new Formatter();
    formatter.format("%04d \n",13);
    String resultat = formatter.toString();
    System.out.println("chaine = " + resultat);

  }

}
```

Résultat :

```
C:\tiger>java TestFormatter
chaine = 0013
```

7.5.10. La classe Scanner

Cette classe facilite la lecture dans un flux. Elle est particulièrement utile pour réaliser une lecture de données à partir du clavier dans une application de type console.

La méthode next() bloque l'exécution jusqu'à la lecture de données et les renvoie sous la forme d'une chaîne de caractères.

Exemple (java 1.5) :

```
import java.util.*;

public class TestScanner {

  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    String chaine = scanner.next();
    scanner.close();
  }
}
```

Cette classe possède plusieurs méthodes nextXXX() où XXX représente un type primitif. Ces méthodes bloquent l'exécution jusqu'à la lecture de données et tente de les convertir dans le type XXX

Exemple (java 1.5) :

```
import java.util.*;

public class TestScanner {

  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    int entier = scanner.nextInt();
    scanner.close();
  }
}
```

Une exception de type InputMismatchException est levée si les données lues dans le flux ne sont pas du type requis.

Exemple (java 1.5) :

```
C:\tiger>java TestScanner
texte
Exception in thread "main" java.util.InputMismatchException
```

```
        at java.util.Scanner.throwFor(Unknown Source)
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at TestScanner.main(TestScanner.java:8)
```

La classe Scanner peut être utilisée avec n'importe quel flux.

7.6. La présentation rapide du package java.net

Ce package contient un ensemble de classes pour permettre une interaction avec le réseau notamment de recevoir et d'envoyer des données.

Le chapitre «L'interaction avec le réseau» détaille l'utilisation de ce package.

7.7. La présentation rapide du package java.applet

Ce package contient les classes nécessaires au développement des applets. Une applet est une petite application téléchargée par le réseau et exécutée sous de fortes contraintes de sécurité dans une page Web par le navigateur.

Le développement des applets est détaillé dans le chapitre «Les applets»

8. Les fonctions mathématiques

Chapitre 8

Niveau : 🟩🟩⬜⬜⬜⬜ Elémentaire

La classe java.lang.Math contient une série de méthodes et variables mathématiques. Comme la classe Math fait partie du package java.lang, elle est automatiquement importée. De plus, il n'est pas nécessaire de déclarer un objet de type Math car les méthodes sont toutes static.

Exemple (code Java 1.1) : Calculer et afficher la racine carrée de 3

```
public class Math1 {
  public static void main(java.lang.String[] args) {
    System.out.println(" = " + Math.sqrt(3.0));
  }
}
```

Ce chapitre contient plusieurs sections :

- Les variables de classe
- Les fonctions trigonométriques
- Les fonctions de comparaisons
- Les arrondis
- La méthode IEEEremainder(double, double)
- Les Exponentielles et puissances
- La génération de nombres aléatoires
- La classe BigDecimal
- La précision des calculs en virgule flottante

8.1. Les variables de classe

PI représente pi dans le type double (3,14159265358979323846)

E représente e dans le type double (2,7182818284590452354)

Exemple (code Java 1.1) :

```
public class Math2 {
  public static void main(java.lang.String[] args) {
    System.out.println(" PI = "+Math.PI);
    System.out.println(" E = "+Math.E);
  }
}
```

8.2. Les fonctions trigonométriques

Les méthodes sin(), cos(), tan(), asin(), acos(), atan() sont déclarées : public static double fonctiontrigo(double angle)

Les angles doivent être exprimés en radians. Pour convertir des degrés en radian, il suffit de les multiplier par PI/180

8.3. Les fonctions de comparaisons

max (n1, n2)
min (n1, n2)

Ces méthodes existent pour les types int, long, float et double : elles déterminent respectivement les valeurs maximales et minimales des deux paramètres.

Exemple (code Java 1.1) :

```
public class Math1 {

  public static void main(String[] args) {
    System.out.println(" le plus grand = " + Math.max(5, 10));
    System.out.println(" le plus petit = " + Math.min(7, 14));
  }
}
```

Résultat :

```
le plus grand = 10
le plus petit = 7
```

8.4. Les arrondis

La classe Math propose plusieurs méthodes pour réaliser différents arrondis.

8.4.1. La méthode round(n)

Pour les types float et double, cette méthode ajoute 0,5 à l'argument et restitue la plus grande valeur entière (int) inférieure ou égale au résultat.

Exemple (code Java 1.1) :

```
public class Arrondis1 {
  static double[] valeur = {-5.7, -5.5, -5.2, -5.0, 5.0, 5.2, 5.5, 5.7 };

  public static void main(String[] args) {
    for (int i = 0; i < valeur.length; i++) {
      System.out.println("round("+valeur[i]+") = "+Math.round(valeur[i]));
    }
  }
}
```

Résultat :

```
round(-5.7) = -6
round(-5.5) = -5
round(-5.2) = -5
round(-5.0) = -5
round(5.0) = 5
round(5.2) = 5
round(5.5) = 6
round(5.7) = 6
```

8.4.2. La méthode rint(double)

Cette méthode effectue la même opération mais renvoie un type double.

Exemple (code Java 1.1) :

```
public class Arrondis2 {
  static double[] valeur = {-5.7, -5.5, -5.2, -5.0, 5.0, 5.2, 5.5, 5.7 };

  public static void main(String[] args) {
    for (int i = 0; i < valeur.length; i++) {
    System.out.println("rint("+valeur[i]+") = "+Math.rint(valeur[i]));
    }
  }
}
```

Résultat :

```
rint(-5.7) = -6.0
rint(-5.5) = -6.0
rint(-5.2) = -5.0
rint(-5.0) = -5.0
rint(5.0) = 5.0
rint(5.2) = 5.0
rint(5.5) = 6.0
rint(5.7) = 6.0
```

8.4.3. La méthode floor(double)

Cette méthode renvoie l'entier le plus proche inférieur ou égal à l'argument.

Exemple (code Java 1.1) :

```
public class Arrondis3 {
  static double[] valeur = {-5.7, -5.5, -5.2, -5.0, 5.0, 5.2, 5.5, 5.7 };

  public static void main(String[] args) {
    for (int i = 0; i < valeur.length; i++) {
      System.out.println("floor("+valeur[i]+") = "+Math.floor(valeur[i]));
    }
  }
}
```

Résultat :

```
floor(-5.7) = -6.0
floor(-5.5) = -6.0
floor(-5.2) = -6.0
floor(-5.0) = -5.0
floor(5.0) = 5.0
floor(5.2) = 5.0
floor(5.5) = 5.0
floor(5.7) = 5.0
```

8.4.4. La méthode ceil(double)

Cette méthode renvoie l'entier le plus proche supérieur ou égal à l'argument

Exemple (code Java 1.1) :

```
public class Arrondis4 {
  static double[] valeur = {-5.7, -5.5, -5.2, -5.0, 5.0, 5.2, 5.5, 5.7 };
  public static void main(String[] args) {
    for (int i = 0; i < valeur.length; i++) {
      System.out.println("ceil("+valeur[i]+") = "+Math.ceil(valeur[i]));
    }
  }
}
```

Résultat :

```
ceil(-5.7) = -5.0
ceil(-5.5) = -5.0
ceil(-5.2) = -5.0
ceil(-5.0) = -5.0
ceil(5.0) = 5.0
ceil(5.2) = 6.0
ceil(5.5) = 6.0
ceil(5.7) = 6.0
```

8.4.5. La méthode abs(x)

Cette méthode donne la valeur absolue de x (les nombres négatifs sont convertis en leur opposé). La méthode est définie pour les types int, long, float et double.

Exemple (code Java 1.1) :

```
public class Math1 {
  public static void main(String[] args) {
    System.out.println(" abs(-5.7) = "+Math.abs(-5.7));
  }
}
```

Résultat :

```
abs(-5.7) = 5.7
```

8.5. La méthode IEEEremainder(double, double)

Cette méthode renvoie le reste de la division du premier argument par le deuxième

Exemple (code Java 1.1) :

```
public class Math1 {
  public static void main(String[] args) {
    System.out.println(" reste de la division de 10 par 3 = "
      +Math.IEEEremainder(10.0, 3.0) );
  }
}
```

Résultat :

```
reste de la division de 10 par 3 = 1.0
```

8.6. Les Exponentielles et puissances

8.6.1. La méthode pow(double, double)

Cette méthode élève le premier argument à la puissance indiquée par le second.

Exemple (code Java 1.1) :

```
public static void main(java.lang.String[] args) {
  System.out.println(" 5 au cube  = "+Math.pow(5.0, 3.0) );
}
```

Résultat :

```
5 au cube  = 125.0
```

8.6.2. La méthode sqrt(double)

Cette méthode calcule la racine carrée de son paramètre.

Exemple (code Java 1.1) :

```
public static void main(java.lang.String[] args) {
  System.out.println(" racine carrée de 25  = "+Math.sqrt(25.0) );
}
```

Résultat :

```
racine carrée de 25 = 5.0
```

8.6.3. La méthode exp(double)

Cette méthode calcule l'exponentielle de l'argument

Exemple (code Java 1.1) :

```
public static void main(java.lang.String[] args) {
  System.out.println(" exponentiel de 5  = "+Math.exp(5.0) );
}
```

Résultat :

```
exponentiel de 5  = 148.4131591025766
```

8.6.4. La méthode log(double)

Cette méthode calcule le logarithme naturel de l'argument

Exemple (code Java 1.1) :

```
public static void main(java.lang.String[] args) {
  System.out.println(" logarithme de 5  = "+Math.log(5.0) );
}
```

Résultat :

```
logarithme de 5  = 1.6094379124341003
```

8.7. La génération de nombres aléatoires

La méthode random() renvoie un nombre aléatoire compris entre 0.0 et 1.0.

Exemple (code Java 1.1) :

```
public static void main(java.lang.String[] args) {
  System.out.println(" un nombre aléatoire  = "+Math.random() );
}
```

Résultat :

```
un nombre aléatoire  = 0.8178819778125899
```

8.8. La classe BigDecimal

La classe java.math.BigDecimal est incluse dans l'API Java depuis la version 5.0.

La classe BigDecimal qui hérite de la classe java.lang.Number permet de réaliser des calculs en virgule flottante avec une précision dans les résultats similaire à celle de l'arithmétique scolaire.

La classe BigDecimal permet ainsi une représentation exacte des valeurs ce que ne peuvent garantir les données primitives de type numérique flottant (float ou double). Les calculs en virgule flottante privilégient en effet la vitesse de calcul plutôt que la précision.

Exemple :

```
package fr.jmdoudoux.dej.bigdecimal;

public class CalculDouble {

  public static void main(String[] args) {
    double valeur = 10*0.09;
    System.out.println(valeur);
  }
}
```

Résultat :

```
0.8999999999999999
```

Cependant certains calculs, notamment ceux relatifs à des aspects financiers par exemple, requièrent une précision particulière : ces calculs utilisent généralement une précision de deux chiffres.

La classe BigDecimal permet de réaliser de tels calculs en permettant d'avoir le contrôle sur la précision (nombre de décimales significatives après la virgule) et la façon dont l'arrondi est réalisé.

Exemple :

```
package fr.jmdoudoux.dej.bigdecimal;

import java.math.BigDecimal;

public class CalculBigDecimal {

  public static void main(
      String[] args) {
    BigDecimal valeur1 = new BigDecimal("10");
    BigDecimal valeur2 = new BigDecimal("0.09");

    BigDecimal valeur = valeur1.multiply(valeur2);

    System.out.println(valeur);
  }
}
```

Résultat :

```
0.90
```

De plus, la classe BigDecimal peut gérer des valeurs possédant plus de 16 chiffres significatifs après la virgule.

La classe BigDecimal propose de nombreux constructeurs qui attendent en paramètre la valeur en différents types.

Remarque : il est préférable d'utiliser le constructeur attendant en paramètre la valeur sous forme de chaîne de caractères.

Exemple :

```
package fr.jmdoudoux.dej.bigdecimal;

import java.math.BigDecimal;

public class CalculBigDecimal3 {

  public static void main(String[] args) {
    BigDecimal valeur1 = new BigDecimal(2.8);
    BigDecimal valeur2 = new BigDecimal("2.8");

    System.out.println("valeur1="+valeur1);
    System.out.println("valeur2="+valeur2);
  }
}
```

Résultat :

```
valeur1=2.79999999999999982236431605997495353221893310546875
valeur2=2.8
```

Avec cette classe, il est parfois nécessaire de devoir créer une nouvelle instance de BigDecimal à partir de la valeur d'une autre instance de BigDecimal. Aucun constructeur de la classe BigDecimal n'attend en paramètre un objet de type BigDecimal : il est nécessaire d'utiliser le constructeur qui attend en paramètre la valeur sous la forme d'une chaîne de caractères et de lui passer en paramètre le résultat de l'appel de la méthode toString() de l'instance de BigDecimal encapsulant la valeur.

La classe BigDecimal propose de nombreuses méthodes pour réaliser des opérations arithmétiques sur la valeur qu'elle encapsule telles que add(), substract(), multiply(), divide(), min(), max(), pow(), remainder(), divideToIntegralValue(), ...

Le classe BigDecimal est immuable : la valeur qu'elle encapsule ne peut pas être modifiée. Toutes les méthodes qui effectuent une opération sur la valeur encapsulée retournent un nouvel objet de type BigDecimal qui encapsule le résultat de l'opération.

Une erreur courante est d'invoquer la méthode mais de ne pas exploiter le résultat de son exécution.

Exemple :

```
package fr.jmdoudoux.dej.bigdecimal;

import java.math.BigDecimal;

public class CalculBigDecimal7 {
  public static void main(String[] args) {
    BigDecimal valeur = new BigDecimal("10.5");
    BigDecimal bonus = new BigDecimal("4.2");

    valeur.add(bonus);
    System.out.println("valeur=" + valeur);

    valeur = valeur.add(bonus);
    System.out.println("valeur=" + valeur);
  }
}
```

Résultat :

```
valeur=10.5
valeur=14.7
```

La méthode setScale() permet de spécifier la précision de la valeur et éventuellement le mode d'arrondi à appliquer. Elle retourne un objet de type BigDecimal correspondant aux caractéristiques fournies puisque l'objet BigDecimal est immuable.

C'est une bonne pratique de toujours préciser le mode d'arrondi car si un arrondi est nécessaire et que le mode d'arrondi n'est pas précisé alors une exception de type ArithmeticException est levée.

Exemple :

```
package fr.jmdoudoux.dej.bigdecimal;

import java.math.BigDecimal;

public class CalculBigDecimal4 {

  public static void main(String[] args) {
    BigDecimal valeur1 = new BigDecimal(2.8);
    valeur1.setScale(1);
    System.out.println("valeur1="+valeur1);
  }
}
```

Résultat :

```
Exception in thread "main" java.lang.ArithmeticException: Rounding necessary
        at java.math.BigDecimal.divide(BigDecimal.java:1346)
        at java.math.BigDecimal.setScale(BigDecimal.java:2310)
        at java.math.BigDecimal.setScale(BigDecimal.java:2350)
        at fr.jmdoudoux.dej.bigdecimal.CalculBigDecimal4.main(CalculBigDecimal4.java:10)
```

La classe BigDecimal propose plusieurs modes d'arrondis : ROUND_CEILING, ROUND_DOWN, ROUND_FLOOR, ROUND_HALF_UP, ROUND_HALF_DOWN, ROUND_HALF_EVEN, ROUND_UNNECESSARY et ROUND_UP

Exemple :

```
package fr.jmdoudoux.dej.bigdecimal;

import java.math.BigDecimal;

public class CalculBigDecimal5 {

  public static void main(String[] args) {
    BigDecimal valeur = null;
    String strValeur = null;

    strValeur = "0.222";
    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_CEILING);
    System.out.println("ROUND_CEILING    "+strValeur+" :  "+valeur.toString());

    strValeur = "-0.222";
    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_CEILING);
    System.out.println("ROUND_CEILING   "+strValeur+" : "+valeur.toString());

    strValeur = "0.222";
    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_DOWN);
    System.out.println("ROUND_DOWN       "+strValeur+" :  "+valeur.toString());

    strValeur = "0.228";
    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_DOWN);
    System.out.println("ROUND_DOWN       "+strValeur+" :  "+valeur.toString());

    strValeur = "-0.228";
    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_DOWN);
    System.out.println("ROUND_DOWN      "+strValeur+" : "+valeur.toString());

    strValeur = "0.222";
    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_FLOOR);
    System.out.println("ROUND_FLOOR      "+strValeur+" :  "+valeur.toString());

    strValeur = "-0.222";
    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_FLOOR);
    System.out.println("ROUND_FLOOR     "+strValeur+" : "+valeur.toString());

    strValeur = "0.222";
    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_HALF_UP);
    System.out.println("ROUND_HALF_UP    "+strValeur+" :  "+valeur.toString());

    strValeur = "0.225";
    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_HALF_UP);
    System.out.println("ROUND_HALF_UP    "+strValeur+" :  "+valeur.toString());

    strValeur = "0.225";
    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_HALF_DOWN);
    System.out.println("ROUND_HALF_DOWN  "+strValeur+" :  "+valeur.toString());

    strValeur = "0.226";
    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_HALF_DOWN);
    System.out.println("ROUND_HALF_DOWN  "+strValeur+" :  "+valeur.toString());

    strValeur = "0.215";
    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_HALF_EVEN);
    System.out.println("ROUND_HALF_EVEN  "+strValeur+" :  "+valeur.toString());

    strValeur = "0.225";

    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_HALF_EVEN);
    System.out.println("ROUND_HALF_EVEN  "+strValeur+" :  "+valeur.toString());

    strValeur = "0.222";
    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_UP);
    System.out.println("ROUND_UP         "+strValeur+" :  "+valeur.toString());

    strValeur = "0.226";
    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_UP);
    System.out.println("ROUND_UP         "+strValeur+" :  "+valeur.toString());
  }
}
```

Résultat :

```
ROUND_CEILING    0.222 :  0.23
ROUND_CEILING   -0.222 : -0.22
ROUND_DOWN       0.222 :  0.22
ROUND_DOWN       0.228 :  0.22
ROUND_DOWN      -0.228 : -0.22
ROUND_FLOOR      0.222 :  0.22
ROUND_FLOOR     -0.222 : -0.23
ROUND_HALF_UP    0.222 :  0.22
ROUND_HALF_UP    0.225 :  0.23
ROUND_HALF_DOWN  0.225 :  0.22
ROUND_HALF_DOWN  0.226 :  0.23
ROUND_HALF_EVEN  0.215 :  0.22
ROUND_HALF_EVEN  0.225 :  0.22
ROUND_UP         0.222 :  0.23
ROUND_UP         0.226 :  0.23
```

Le mode d'arrondi doit aussi être précisé lors de l'utilisation de la méthode divide().

Exemple :

```
package fr.jmdoudoux.dej.bigdecimal;

import java.math.BigDecimal;

public class CalculBigDecimal6 {
  public static void main(String[] args) {
    BigDecimal valeur = new BigDecimal("1");
    System.out.println(valeur.divide(new BigDecimal("3")));
  }
}
```

Résultat :

```
Exception in thread "main" java.lang.ArithmeticException:
Non-terminating decimal expansion; no exact representable decimal result.
        at java.math.BigDecimal.divide(BigDecimal.java:1514)
        at fr.jmdoudoux.dej.bigdecimal.CalculBigDecimal6.main(CalculBigDecimal6.java:9)
```

Le même exemple en précisant le mode d'arrondi fonctionne parfaitement.

Exemple :

```
package fr.jmdoudoux.dej.bigdecimal;

import java.math.BigDecimal;

public class CalculBigDecimal6 {
  public static void main(String[] args) {
    BigDecimal valeur = new BigDecimal("1");
    System.out.println(valeur.divide(new BigDecimal("3"),4,BigDecimal.ROUND_HALF_DOWN));
  }
}
```

Résultat :

```
0.3333
```

La précision et le mode d'arrondi doivent être choisis avec attention parce que leur choix peut avoir de grandes conséquences sur les résultats de calculs notamment si le résultat final est constitué de multiples opérations. Dans ce cas, il est préférable de garder la plus grande précision durant les calculs et de n'effectuer l'arrondi qu'à la fin.

Il faut être vigilent lors de la comparaison entre deux objets de type BigDecimal. La méthode equals() compare les valeurs mais en tenant compte de la précision. Ainsi, il est préférable d'utiliser la méthode compareTo() qui n'effectue la comparaison que sur la valeur.

Exemple :

```
package fr.jmdoudoux.dej.bigdecimal;

import java.math.BigDecimal;

public class CalculBigDecimal8 {

  public static void main(String[] args) {
    BigDecimal valeur1 = new BigDecimal("10.00");
    BigDecimal valeur2 = new BigDecimal("10.0");

    System.out.println("valeur1.equals(valeur2) = "+valeur1.equals(valeur2));
    System.out.println("valeur1.compareTo(valeur2) = "+(valeur1.compareTo(valeur2)==0));
  }
}
```

Résultat :

```
valeur1.equals(valeur2) = false
valeur1.compareTo(valeur2) = true
```

La méthode compareTo() renvoie 0 si les deux valeurs sont égales, renvoie -1 si la valeur de l'objet fourni en paramètre est plus petite et renvoie 1 si la valeur de l'objet fourni en paramètre est plus grande.

Il est possible de passer en paramètre de la méthode format() de la classe NumberFormat un objet de type BigDecimal : attention dans ce cas, le nombre de décimales est limité à 16.

Exemple formatage d'un BigDecimal avec un format monétaire :

```
package fr.jmdoudoux.dej.bigdecimal;

import java.math.*;
import java.text.*;
import java.util.*;

public class CalculBigDecimal9 {

  public static void main(String[] args) {
    BigDecimal payment = new BigDecimal("1234.567");
    NumberFormat n = NumberFormat.getCurrencyInstance(Locale.FRANCE);
    String s = n.format(payment);
    System.out.println(s);
  }
}
```

Résultat :

```
1 234,57 €
```

La mise en oeuvre de la classe BigDecimal est plutôt fastidieuse comparée à d'autres langages qui proposent un support natif d'un type de données décimal mais elle permet d'effectuer des calculs précis.

L'utilisation de la classe BigDecimal n'est recommandée que si une précision particulière est nécessaire car sa mise en oeuvre est coûteuse.

8.9. La précision des calculs en virgule flottante

L'un des principaux aspects du langage Java est l'indépendance vis-à-vis de la plate-forme : compiler et exécuter le même code sur différentes machines et s'assurer que le résultat est le même quel que soit le hardware et le système d'exploitation utilisés.

Ce but n'est pas complétement atteint notamment à cause de la précision des calculs en virgule flottante. Certaines architectures hardware ont été conçues pour être efficaces, tandis que d'autres ont été conçues pour être précises.

Ainsi, les machines les plus précises utilisent une taille de virgule flottante de 80 bits, tandis que les machines les plus efficaces/rapides utilisent des doubles de 64 bits. Cette différence de précision peut induire des résultats de calculs différents d'une même opération exécutée dans des JVM sur différentes plateformes.

La précision standard pour les calculs en virgule flottante peut varier selon la CPU utilisée : la précision de 32 bits est différente de celle de 64 bits pour les machines x86 par exemple.

Historiquement, par défaut, les calculs en virgule flottante avec Java sont dépendants de la plate-forme. Ainsi, la précision du résultat du calcul en virgule flottante dépend du matériel utilisé et peut donc varier selon l'environnement d'exécution. Lors de l'exécution de calculs en virgules flottantes sur différentes plates-formes, les résultats peuvent donc varier en raison de la capacité de la CPU à traiter les virgules flottantes.

8.9.1. Les calculs en virgule flottante stricte

Différentes plates-formes utilisent un hardware différent qui calcule en virgule flottante avec plus de précision et une plus grande plage de valeurs que ne l'exige la spécification Java. Cela peut produire des résultats différents sur différentes plates-formes.

Le standard IEEE 754 (Standard for Binary Floating-Point Arithmetic) définit une norme standard pour les calculs en virgule flottante et le stockage des valeurs en virgule flottante dans différents formats, y compris la précision simple (32 bits, utilisée par le type float) ou double (64 bits, utilisée par le type double). Elle définit également des normes pour les calculs intermédiaires et pour les formats de précision étendue.

Certains matériels peuvent fournir une précision plus élevée et/ou une plage d'exposants plus large. Sur ces architectures, il peut être plus efficace de calculer les résultats intermédiaires en utilisant ces formats étendus. Cela permet d'éviter les erreurs d'arrondi, les débordements et les sous-débordements qui se produiraient autrement, mais les programmes peuvent produire des résultats différents sur ces architectures.

Il était coûteux d'éviter l'utilisation de la précision étendue sur les machines x86 dotées de l'architecture traditionnelle à virgule flottante x87. Bien qu'il soit facile de contrôler la précision des calculs, la limitation de la plage d'exposants pour les résultats intermédiaires a nécessité des instructions supplémentaires coûteuses.

Avant la version 1.2 de la JVM, les calculs en virgule flottante devaient être stricts : tous les résultats intermédiaires en virgule flottante devaient se comporter comme s'ils étaient représentés à l'aide des précisions simples ou doubles de l'IEEE. Il était donc coûteux, sur le matériel courant basé sur x87, de s'assurer que les débordements se produisaient là où c'était nécessaire.

Depuis la version 1.2 de la JVM, les calculs intermédiaires sont, par défaut, autorisés à dépasser les plages d'exposants standard associées aux formats IEEE 32 bits et 64 bits. Ils peuvent être représentés comme un membre de l'ensemble de valeurs "extended-exponent". Sur des plates-formes telles que x87, les débordements et les sous-débordements peuvent ne pas se produire là où ils sont attendus, produisant des résultats peut-être plus précis, mais moins reproductibles.

Ainsi historiquement par défaut de Java 1.2 à Java 16, les calculs sur des nombres flottants (float ou double) sont réalisés de manière non stricte.

A partir de Java 17, les calculs sur des nombres flottants (float ou double) sont systématiquement réalisés de manière stricte. Le mot clé strictfp n'a alors plus d'utilité.

8.9.2. Le mot clé réservé strictfp

Les calculs en virgule flottante dépendent de la plate-forme : des résultats différents peuvent être obtenus lorsque du bytecode est exécuté sur différents processeurs. Pour résoudre ce type de problème, le mot-clé strictfp a été introduit dans la version 1.2 du JDK.

Le mot-clé réservé strictfp est utilisé pour garantir que les opérations en virgule flottante donnent le même résultat quel que soit la plate-forme utilisée pour les exécuter et donc d'être indépendant de l'architecture matérielle.

Le mot clé réservé strictfp s'utilise comme un modificateur dont le rôle est de demander d'appliquer la sémantique FP-Strict (Floating Point Strict) de la norme standard IEEE 754 lors des calculs en virgule flottante afin de garantir la portabilité sur toutes les plates-formes.

En l'absence d'overflow ou d'underflow, il n'y a pas de différence entre les résultats obtenus avec ou sans strictfp. Si la répétabilité est essentielle, le modificateur strictfp peut être utilisé pour s'assurer que les d'overflows et les underflows se produisent aux mêmes endroits sur toutes les plates-formes. Sans le modificateur strictfp, les résultats intermédiaires peuvent utiliser une plage d'exposants plus importante.

Le modificateur strictfp permet d'atteindre ses objectifs en représentant toutes les valeurs intermédiaires comme des valeurs IEEE avec précision simple ou de double, comme c'était le cas dans les versions antérieures à la JVM 1.2.

L'utilisation du modificateur strictfp permet de garantir que les résultats des calculs sur des valeurs flottantes soient les mêmes sur toutes les JVM. Si strictfp n'est pas utilisé, la JVM est libre d'utiliser toute précision supplémentaire disponible sur le matériel de la plate-forme d'exécution.

Le mot clé strictfp s'utilise comme un modificateur et ne peut être appliqué que sur :

- des classes
- des interfaces
- des méthodes non abstraites

L'élément déclaré avec le modificateur strictfp est dit FP-strict.

Toutes les méthodes déclarées dans une classe ou une interface, ainsi que tous les types imbriqués déclarés dans une classe sont implicitement strictfp si la classe ou l'interface est déclarée avec le modificateur strictfp.

Dans une expression FP-strict, toutes les valeurs intermédiaires en virgule flottante doivent être conformes à la spécification IEEE 754.

Dans une expression qui n'est pas FP-strict, une certaine marge de manoeuvre est accordée à une implémentation d'une JVM pour utiliser une plage d'exposants étendue pour représenter les résultats intermédiaires.

En particulier, les processeurs x86 peuvent stocker des résultats intermédiaires avec une précision différente de la spécification IEEE 754. La situation se complique lorsque le JIT optimise un calcul particulier, notamment en changeant l'ordre des instructions, ce qui peut entraîner un arrondi légèrement différent.

Si le modificateur strictfp n'est pas utilisé, alors la JVM et le compilateur JIT sont libres d'exécuter les calculs en virgule flottante comme ils le veulent. Pour des raisons de performance, ils délégueront probablement le calcul au processeur. Si strictfp est utilisé, les calculs doivent être conformes à norme IEEE 754, ce qui, dans la pratique, signifie probablement que la JVM effectuera elle-même les calculs.

Avec ce modificateur, les résultats des calculs en virgule flottante sont prédictibles et identiques, quelle que soit la plateforme sous-jacente sur laquelle s'exécute la JVM. L'inconvénient est que si la plateforme sous-jacente est capable de supporter une plus grande précision, des calculs strictfp ne pourront pas en profiter.

Le surcoût induit par strictfp dépend beaucoup du processeur et du JIT. Si le JIT peut générer des instructions SSE pour effectuer un calcul, la surcharge est faible voire nulle.

Les expressions constantes à la compilation utilisent toujours le comportement FP-strict.

Le modificateur strictfp est utile pour garantir d'obtenir les mêmes résultats dans les calculs flottants quel que soit la plateforme d'exécution notamment pour des tests unitaires reproductibles.

8.9.2.1. L'utilisation de strictfp sur des classes

Lors de l'utilisation du modificateur strictfp sur une classe, tous les calculs à l'intérieur de la classe utilisent des calculs en virgule flottante strictes selon la norme IEEE 754.

L'utilisation de strictfp sur une classe a pour effet de rendre toutes les expressions float ou double dans la déclaration de la classe (y compris dans les initialisateurs de variables, les initialisateurs d'instances, les initialisateurs statiques et les constructeurs) explicitement FP-strict.

Cela implique que toutes les méthodes déclarées dans la classe, et toutes les classes imbriquées et interfaces, records et énumérations déclarés dans la classe, sont implicitement strictfp.

Exemple (code Java 1.2) :

```
strictfp class MaClasse {

    // toutes les methodes concrètes sont implicitement strictfp
}
```

Le modificateur strictfp peut être utilisé dans la définition d'une classe abstraite.

Exemple (code Java 1.2) :

```
strictfp abstract class MaClasse {

    // toutes les methodes concrètes sont implicitement strictfp
}
```

Si une superclasse est déclarée avec le mot-clé strictfp, une sous-classe n'héritera pas de ce comportement.

8.9.2.2. L'utilisation de strictfp sur des méthodes non abstraites

Si une classe n'est pas déclarée avec strictfp, il est possible obtenir le comportement FP-strict méthode par méthode, en déclarant chaque méthode concernée avec le modificateur strictfp.

Lorsqu'il est utilisé sur une méthode, il fait en sorte que tous les calculs dans le corps de la méthode utilisent des calculs en virgule flottante strictes selon la norme IEEE 754.

Exemple (code Java 1.2) :

```
class MaClasse {
  strictfp void calculer() {}
}
```

La présence ou l'absence du modificateur strictfp n'a absolument aucun effet sur les règles de surcharge des méthodes et d'implémentation des méthodes abstraites. Par exemple, il est permis à une méthode qui n'est pas FP-strict de surcharger une méthode FP-strict et il est permis à une méthode FP-strict de surcharger une méthode qui n'est pas FP-strict.

Le compilateur émet une erreur de compilation si une déclaration de méthode contenant le mot-clé abstract ou native contient également le mot clé strictfp.

Exemple (code Java 1.2) :

```
class MaClasse {
  strictfp MaClasse() {}
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:2: error: modifier native,strictfp not allowed here
  native strictfp MaClasse();
                  ^
1 error
```

8.9.2.3. L'utilisation de strictfp sur des interfaces

L'effet du modificateur strictfp est de faire en sorte que toutes les expressions flottantes ou doubles dans le corps d'une méthode par défaut ou statique soient explicitement FP-strict.

Cela implique que toutes les méthodes déclarées dans l'interface, et tous les types imbriqués déclarés dans l'interface, sont implicitement strictfp.

Exemple (code Java 1.2) :

```
strictfp interface MonInterface {
  // toutes les méthodes sont implicitement strictes
}
```

Le compilateur émet une erreur si une déclaration de méthode d'une interface qui contient le mot-clé abstract (implicite ou explicite) contient également le mot-clé strictfp.

8.9.2.4. Les cas d'utilisation invalides de strictfp

Plusieurs cas d'usage de strictfp sont invalides : le mot-clé strictfp ne peut pas être utilisé dans la déclaration de méthodes abstraites, de variables ou de constructeurs.

Le mot-clé strictfp n'est pas utilisable dans la déclaration d'une méthode abstraite.

Exemple (code Java 1.2) :

```
class MaClasse {

  strictfp abstract void calculer();
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:2: error: illegal combination of modifiers: abstract and strictfp
  strictfp abstract void calculer();
                         ^
MaClasse.java:1: error: MaClasse is not abstract and does not override abstract method
 calculer() in MaClasse
class MaClasse {
^
2 errors
```

Puisque les méthodes d'une interface sont implicitement abstraites, strictfp ne peut pas être utilisé sur les méthodes d'une interface qui ne sont pas static ou default.

Exemple (code Java 1.2) :

```
strictfp interface MonInterface {

    strictfp double calculer();
}
```

Résultat :

```
C:\java>javac MonInterface.java
MonInterface.java:3: error: modifier strictfp not allowed here
    strictfp double calculer();
                    ^
1 error
```

Le mot-clé trictfp n'est pas utilisable dans la déclaration d'une variableou d'un champ.

Exemple (code Java 1.2) :

```
class MaClasse {
  strictfp float valeur; // modifier strictfp not allowed here
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:2: error: modifier strictfp not allowed here
  strictfp float valeur;
                 ^
1 error
```

Le mo-clé strictfp n'est pas utilisable dans la déclaration d'un constructeur. Contrairement aux méthodes, un constructeur ne peut pas avoir de modificateur strictfp : l'impossibilité de déclarer un constructeur comme strictfp, contrairement à une méthode, est un choix intentionnel de conception du langage qui garantit qu'un constructeur est FP-strict si et seulement si sa classe est FP-strict.

Exemple (code Java 1.2) :

```
class MaClasse {
  strictfp MaClasse() {}
}
```

Résultat :

```
C:\java>javac MaClasse.java
MaClasse.java:2: error: modifier strictfp not allowed here
  strictfp MaClasse() {}
           ^
1 error
```

8.9.2.5. L'utilisation de strictfp à partir de Java 17

La JEP 306, implémentée dans le JDK 17, rend les opérations en virgule flottante systématiquement strictes, plutôt que d'avoir à la fois une sémantique stricte de virgule flottante (strictfp) et une sémantique de virgule flottante par défaut potentiellement différente.

Les extensions SSE2 (Streaming SIMD Extensions 2), fournies dans les processeurs Pentium 4 et les processeurs ultérieurs à partir de 2001, peuvent prendre en charge les opérations strictes en virgule flottante de manière directe et sans overhead excessif.

Étant donné qu'Intel et AMD supportent depuis longtemps SSE2 et les extensions ultérieures qui permettent un support intégré de la sémantique flottante stricte, la motivation technique pour avoir une sémantique flottante par défaut différente de la sémantique stricte n'est plus nécessaire.

Comme le jeu d'instruction x87 n'est plus nécessaire sur les processeurs x86 supportant le SSE2 et que sur les processeurs modernes il n'y a plus de coûts supplémentaires en termes de performances, Java 17 a de nouveau rendu toutes les opérations en virgule flottante strictes, rétablissant ainsi la sémantique d'avant la version 1.2.

En Java 17, le modificateur strictfp est donc devenu inutile et ne doit plus être utilisé. Sa présence ou son absence n'a aucun effet sur les résultats des calculs, car toutes les opérations en virgule flottante sont strictes.

Le compilateur à partir du JDK 17 émet un avertissement à chaque utilisation du modificateur strictfp.

Résultat :

```
C:\java>java -version
openjdk version "17" 2021-09-14
OpenJDK Runtime Environment (build 17+35-2724)
OpenJDK 64-Bit Server VM (build 17+35-2724, mixed mode, sharing)

C:\java>javac StrictFP.java
StrictFP.java:3: warning: [strictfp] as of release 17, all floating-point expressions
 are evaluated strictly and 'strictfp' is not required
    public static strictfp double calculerStrict(double a, double b) {
                                  ^
1 warning
```

9. La gestion des exceptions

Chapitre 9

Niveau : 🟩🟩⬜⬜⬜⬜ Elémentaire

Les exceptions représentent le mécanisme de gestion des erreurs intégré au langage Java. Il se compose d'objets représentant les erreurs et d'un ensemble de trois mots clés qui permettent de détecter et de traiter ces erreurs (try, catch et finally) mais aussi de les lever ou les propager (throw et throws).

Lors de la détection d'une erreur, un objet qui hérite de la classe Exception est créé (on dit qu'une exception est levée) et propagé à travers la pile d'exécution jusqu'à ce qu'il soit traité.

Ces mécanismes permettent de renforcer la sécurité du code Java.

Exemple : une exception levée à l'exécution non capturée

```
public class TestException {
  public static void main(java.lang.String[] args) {
    int i = 3;
    int j = 0;
    System.out.println("résultat = " + (i / j));
  }
}
```

Résultat :

```
C:>java TestException
Exception in thread "main" java.lang.ArithmeticException: /
by zero
        at tests.TestException.main(TestException.java:23)
```

Si dans un bloc de code on fait appel à une méthode qui peut potentiellement générer une exception, on doit soit essayer de la récupérer avec try/catch, soit ajouter le mot clé throws dans la déclaration du bloc. Si on ne le fait pas, il y a une erreur à la compilation. Les erreurs et exceptions du paquetage java.lang échappent à cette contrainte. Throws permet de déléguer la responsabilité des erreurs à la méthode appelante

Ce procédé présente un inconvénient : de nombreuses méthodes des packages java indiquent dans leur déclaration qu'elles peuvent lever une exception. Cependant ceci garantit que certaines exceptions critiques seront prises explicitement en compte par le programmeur.

Ce chapitre contient plusieurs sections :

- Les mots clés try, catch et finally
- La classe Throwable
- Les classes Exception, RunTimeException et Error
- Les exceptions personnalisées
- Les exceptions chaînées
- L'utilisation des exceptions
- L'instruction try-with-resources
- Des types plus précis lorsqu'une exception est relevée dans une clause catch
- Multiples exceptions dans une clause catch

9.1. Les mots clés try, catch et finally

Le bloc try rassemble les appels de méthodes susceptibles de produire des erreurs ou des exceptions. L'instruction try est suivie d'instructions entre des accolades.

Exemple (code Java 1.1) :

```
    try {
      operation_risquée1;
      opération_risquée2;
    } catch (ExceptionInteressante e) {
      traitements
    } catch (ExceptionParticulière e) {
      traitements
    } catch (Exception e) {
      traitements
    } finally {
      traitement_pour_terminer_proprement;
    }
```

Si un événement indésirable survient dans le bloc try, la partie éventuellement non exécutée de ce bloc est abandonnée et le premier bloc catch est traité. Si un bloc catch est défini pour capturer l'exception issue du bloc try alors elle est traitée en exécutant le code associé au bloc. Si le bloc catch est vide (aucune instruction entre les accolades) alors l'exception capturée est ignorée. Une telle utilisation de l'instruction try/catch n'est pas une bonne pratique : il est préférable de toujours apporter un traitement adapté lors de la capture d'une exception.

S'il y a plusieurs types d'erreurs et d'exceptions à intercepter, il faut définir autant de blocs catch que de types d'événements. Par type d'exception, il faut comprendre « qui est du type de la classe de l'exception ou d'une de ses sous-classes ». Ainsi dans l'ordre séquentiel des clauses catch, un type d'exception ne doit pas venir après un type d'une exception d'une super-classe. Il faut faire attention à l'ordre des clauses catch pour traiter en premier les exceptions les plus précises (sous-classes) avant les exceptions plus générales. Un message d'erreur est émis par le compilateur dans le cas contraire.

Exemple (code Java 1.1) : erreur à la compil car Exception est traité en premier alors que ArithmeticException est une sous-classe de Exception

```
public class TestException {
  public static void main(java.lang.String[] args) {
    // Insert code to start the application here.
    int i = 3;
    int j = 0;
    try {
      System.out.println("résultat = " + (i / j));
    } catch (Exception e) {
    } catch (ArithmeticException e) {
    }
  }
}
```

Résultat :

```
C:\tests>javac TestException.java
TestException.java:11: catch not reached.
        catch (ArithmeticException e) {
        ^
1 error
```

Si l'exception générée est une instance de la classe déclarée dans la clause catch ou d'une classe dérivée, alors on exécute le bloc associé. Si l'exception n'est pas traitée par un bloc catch, elle sera transmise au bloc de niveau supérieur. Si l'on ne se trouve pas dans un autre bloc try, on quitte la méthode en cours, qui regénère à son tour une exception dans la méthode appelante.

L'exécution totale du bloc try et d'un bloc d'une clause catch sont mutuellement exclusives : si une exception est levée, l'exécution du bloc try est arrêtée et si elle existe, la clause catch adéquate est exécutée.

La clause finally définit un bloc qui sera toujours exécuté, qu'une exception soit levée ou non. Ce bloc est facultatif. Il est aussi exécuté si dans le bloc try il y a une instruction break ou continue.

9.2. La classe Throwable

Cette classe descend directement de la classe Object : c'est la classe de base pour le traitement des erreurs.

Cette classe possède deux constructeurs :

Méthode	Rôle
Throwable()	
Throwable(String)	La chaîne en paramètre permet de définir un message qui décrit l'exception et qui pourra être consulté dans un bloc catch.

Les principales méthodes de la classe Throwable sont :

Méthodes	Rôle
String getMessage( )	lecture du message
void printStackTrace( )	affiche l'exception et l'état de la pile d'exécution au moment de son appel
void printStackTrace(PrintStream s)	Idem mais envoie le résultat dans un flux

Exemple (code Java 1.1) :

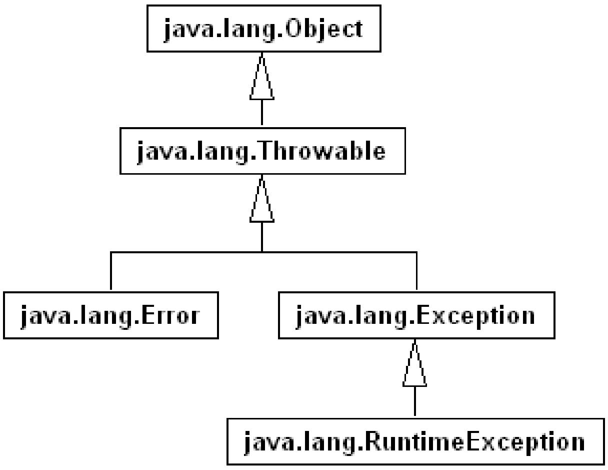
```
public class TestException {
  public static void main(java.lang.String[] args) {
    // Insert code to start the application here.
    int i = 3;
    int j = 0;
    try {
      System.out.println("résultat = " + (i / j));
    } catch (ArithmeticException e) {
      System.out.println("getmessage");
      System.out.println(e.getMessage());
      System.out.println(" ");
      System.out.println("toString");
      System.out.println(e.toString());
      System.out.println(" ");
      System.out.println("printStackTrace");
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
C:>java TestException
getmessage
/ by zero

toString
java.lang.ArithmeticException: / by zero

printStackTrace
java.lang.ArithmeticException: / by zero
        at tests.TestException.main(TestException.java:24)
```

9.3. Les classes Exception, RunTimeException et Error

Ces trois classes descendent de Throwable : en fait, toutes les exceptions dérivent de la classe Throwable.



La classe Error représente une erreur grave intervenue dans la machine virtuelle Java ou dans un sous système Java. L'application Java s'arrête instantanément dès l'apparition d'une exception de la classe Error.

La classe Exception représente des erreurs moins graves. Les exceptions héritant de la classe RuntimeException n'ont pas besoin d'être détectées impérativement par des blocs try/catch.

9.4. Les exceptions personnalisées

Pour générer une exception, il suffit d'utiliser le mot clé throw, suivi d'un objet dont la classe dérive de Throwable. Si l'on veut générer une exception dans une méthode avec throw, il faut l'indiquer dans la déclaration de la méthode, en utilisant le mot clé throws.

En cas de nécessité, on peut créer ses propres exceptions. Elles descendent des classes Exception ou RunTimeException mais pas de la classe Error. Il est préférable (par convention) d'inclure le mot « Exception » dans le nom de la nouvelle classe.

Exemple (code Java 1.1) :

```
public class SaisieErroneeException extends Exception {

  public SaisieErroneeException() {
    super();
  }

  public SaisieErroneeException(String s) {
    super(s);
  }
}

public class TestSaisieErroneeException {
  public static void controle(String chaine) throws SaisieErroneeException {
    if (chaine.equals("") == true)
      throw new SaisieErroneeException("Saisie erronee : chaine vide");
  }

  public static void main(java.lang.String[] args) {
    String chaine1 = "bonjour";
    String chaine2 = "";

    try {
      controle(chaine1);
    } catch (SaisieErroneeException e) {
      System.out.println("Chaine1 saisie erronee");
    }

    try {
      controle(chaine2);
    } catch (SaisieErroneeException e) {
      System.out.println("Chaine2 saisie erronee");
    }
  }
}
```

Les méthodes pouvant lever des exceptions doivent inclure une clause throws nom_exception dans leur en-tête. L'objectif est double : avoir une valeur documentaire et préciser au compilateur que cette méthode pourra lever cette exception et que toute méthode qui l'appelle devra prendre en compte cette exception (traitement ou propagation).

Si la méthode appelante ne traite pas l'erreur ou ne la propage pas, le compilateur génère l'exception nom_exception must be caught or it must be declared in the throws clause of this method.

Java n'oblige à déclarer les exceptions dans l'en-tête de la méthode que pour les exceptions dites contrôlées (checked). Les exceptions non contrôlées (unchecked) peuvent être capturées mais n'ont pas à être déclarées. Les exceptions et erreurs qui héritent de RunTimeException et de Error sont non contrôlées. Toutes les autres exceptions sont contrôlées.

9.5. Les exceptions chaînées

Il est fréquent durant le traitement d'une exception de lever une autre exception. Pour ne pas perdre la trace de l'exception d'origine, Java propose le chaînage d'exceptions pour conserver l'empilement des exceptions levées durant les traitements.

Il y a deux façons de chaîner deux exceptions :

- Utiliser la surcharge du constructeur de Throwable qui attend un objet Throwable en paramètre
- Utiliser la méthode initCause() d'une instance de Throwable

Exemple :

```
package fr.jmdoudoux.dej;

import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class TestExceptionChainee {

  public static void main(String[] args) {
    try {
      String donnees = lireFichier();
      System.out.println("donnees=" + donnees);
    } catch (MonException e) {
      e.printStackTrace();
    }
  }

  public static String lireFichier() throws MonException {
    File fichier = new File("c:/tmp/test.txt");
    FileReader reader = null;

    StringBuffer donnees = new StringBuffer();

    try {
      reader = new FileReader(fichier);
      char[] buffer = new char[2048];
      int len;
      while ((len = reader.read(buffer)) > 0) {
        donnees.append(buffer, 0, len);
      }
    } catch (IOException e) {
      throw new MonException("Impossible de lire le fichier", e);
    } finally {
      try {
        if (reader != null) {
          reader.close();
        }
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
    return donnees.toString();
  }
}
```

Résultat :

```
fr.jmdoudoux.dej.MonException: Impossible de lire le fichier
        at fr.jmdoudoux.dej.TestExceptionChainee.lireFichier(TestExceptionChainee.java:33)
        at fr.jmdoudoux.dej.TestExceptionChainee.main(TestExceptionChainee.java:12)
Caused by: java.io.FileNotFoundException: c:\tmp\test.txt (The system cannot
        find the path specified)
        at java.io.FileInputStream.open(Native Method)
        at java.io.FileInputStream.<init>(FileInputStream.java:106)
        at java.io.FileReader.<init>(FileReader.java:55)
        at fr.jmdoudoux.dej.TestExceptionChainee.lireFichier(TestExceptionChainee.java:26)
        ... 1 more
```

La méthode getCause() héritée de Throwable permet d'obtenir l'exception originale.

Exemple :

```
  public static void main(
      String[] args) {
    try {
      String donnees = lireFichier();
      System.out.println("donnees=" + donnees);
    } catch (MonException e) {
      // e.printStackTrace();
      System.out.println(e.getCause().getMessage());
    }
  }
```

Résultat :

```
c:\tmp\test.txt (The system cannot find the path specified)
```

9.6. L'utilisation des exceptions

Il est préférable d'utiliser les exceptions fournies par Java lorsqu'une de ces exceptions répond au besoin plutôt que de définir sa propre exception.

Il existe trois types d'exceptions :

- Error : ces exceptions concernent des problèmes liés à l'environnement. Elles héritent de la classe Error (exemple : OutOfMemoryError)
- RuntimeException : ces exceptions concernent des erreurs de programmation qui peuvent survenir à de nombreux endroits dans le code (exemple : NullPointerException). Elles héritent de la classe RuntimeException
- Checked exception : ces exceptions doivent être traitées ou propagées. Toutes les exceptions qui n'appartiennent pas aux catégories précédentes sont de ce type

Les exceptions de type Error et RuntimeException sont dites unchecked exceptions car les méthodes n'ont pas d'obligation à les traiter ou à déclarer leur propagation explicitement. Ceci se justifie par le fait que leur levée n'est pas facilement prédictible.

Il n'est pas recommandé de créer ses propres exceptions en dérivant d'une exception de type unchecked (classe de type RuntimeException). Même si cela peut sembler plus facile puisqu'il n'est pas obligatoire de déclarer leur propagation, cela peut engendrer certaines difficultés, notamment :

- oublier de traiter cette exception
- ne pas savoir que cette exception peut être levée par une méthode.

Cependant, l'utilisation d'exceptions de type unchecked se répend de plus en plus notamment depuis la diffusion de la plate-forme .Net qui ne propose que ce type d'exceptions.

9.7. L'instruction try-with-resources

Des ressources comme des fichiers, des flux, des connexions, ... doivent être fermées explicitement par le développeur pour libérer les ressources sous-jacentes qu'elles utilisent. Généralement cela est fait en utilisant un bloc try / finally pour garantir leur fermeture dans la quasi-totalité des cas.

De plus, la nécessité de fermer explicitement la ressource implique un risque potentiel d'oubli de fermeture qui entraine généralement une fuite de ressources.

Avec Java 7, l'instruction try avec ressource permet de définir une ressource qui sera automatiquement fermée à la fin de l'exécution du bloc de code de l'instruction.

Ce mécanisme est aussi désigné par l'acronyme ARM (Automatic Ressource Management).

Avant Java 7, il était nécessaire d'utiliser un bloc finally pour s'assurer que le flux sera fermé même si une exception est levée durant les traitements. Ce type de traitement possède plusieurs inconvénients :

- La ressource utilisée doit être déclarée en dehors du bloc try pour pouvoir être utilisée dans le bloc finally
- L'invocation de la méthode close() sur la ressource peut aussi lever une exception de type IOException qu'il faut gérer en propageant cette exception ou en incluant l'invocation de cette méthode dans un bloc try/catch

Exemple :

```
package fr.jmdoudoux.dej.java7;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class TestCloseRessource {

  public static void main(String[] args) throws IOException {
    System.out.println(lireContenu(new File("monfichier.txt")));
  }

  static public String lireContenu(File fichier) {
    StringBuilder contenu = new StringBuilder();
    try {
      BufferedReader input = null;
      try {
        input = new BufferedReader(new FileReader(fichier));
        String ligne = null;
        while ((ligne = input.readLine()) != null) {
          contenu.append(ligne);
          contenu.append("\n");
        }
      } finally {
        if (input != null) {
          input.close();
        }
     }
   } catch (IOException ex) {
     ex.printStackTrace();
   }
   return contenu.toString();
 }
}
```

L'inconvénient de cette solution est que l'exception propagée serait celle de la méthode close() si elle lève une exception qui pourrait alors masquer une exception levée dans le bloc try. Il est possible de capturer l'exception de la méthode close().

Exemple :

```
package fr.jmdoudoux.dej.java7;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class TestCloseRessource {

  public static void main(String[] args) throws IOException {
    System.out.println(lireContenu(new File("monfichier.txt")));
  }

  static public String lireContenu(File fichier) {
    StringBuilder contenu = new StringBuilder();
    try {
      BufferedReader input = null;
      try {
        input = new BufferedReader(new FileReader(fichier));
        String ligne = null;
        while ((ligne = input.readLine()) != null) {
          contenu.append(ligne);
          contenu.append("\n");
        }
      } finally {
        if (input != null) {

        try {
            input.close();
          } catch (IOException e) {
            e.printStackTrace();
          }
        }
      }
    } catch (IOException ex) {
      ex.printStackTrace();
    }
    return contenu.toString();
  }
}
```

L'inconvénient de cette solution est que l'exception qui peut être levée par la méthode close() n'est pas propagée. De plus la quantité de code produite devient plus importante.

Avec Java 7, le mot clé try peut être utilisé pour déclarer une ou plusieurs ressources.

Une ressource est un objet qui doit être fermé lorsque l'on a plus besoin de lui : généralement cette ressource encapsule ou utilise des ressources du système : fichiers, flux, connexions vers des serveurs, ...

Une nouvelle interface a été définie pour indiquer qu'une ressource peut être fermée automatiquement : java.lang.AutoCloseable.

Tous les objets qui implémentent l'interface java.lang.AutoCloseable peuvent être utilisés dans une instruction de type try-with-resources. L'instruction try avec des ressources garantit que chaque ressource déclarée sera fermée à la fin de l'exécution de son bloc de traitement.

L'interface java.lang.Autocloseable possède une unique méthode close() qui sera invoquée pour fermer automatiquement la ressource encapsulée par l'implémentation de l'interface.

L'interface java.io.Closable introduite par Java 5 hérite de l'interface AutoCloseable : ainsi toutes les classes qui implémentent l'interface Closable peuvent être utilisées comme ressource dans une instruction try-with-resource.

La méthode close() de l'interface Closeable lève une exception de type IOException alors que la méthode close() de l'interface AutoCloseable lève une exception de type Exception. Cela permet aux interfaces filles de AutoCloseable de redéfinir la méthode close() pour qu'elles puissent lever une exception plus spécifique ou aucune exception.

Contrairement à la méthode close() de l'interface Closeable, une implémentation de la méthode close() de l'interface AutoCloseable n'est pas supposée être idempotente : son invocation une seconde fois peut avoir des effets de bords.

Une implémentation de la méthode close() de l'interface AutoCloseable() devrait déclarer une exception plus précise que simplement Exception ou ne pas déclarer d'exception du tout si l'opération de fermeture ne peut échouer.

Il faut garder à l'esprit que l'exception levée sera masquée par l'instruction try-with-resource : l'implémentation de la méthode close() doit faire attention aux exceptions qu'elle peut lever (par exemple, comme le précise la Javadoc, elle ne doit pas lever une exception de type InterruptedException)

L'instruction try avec des ressources utilise le mot clé try avec une ou plusieurs ressources définies dans sa portée, chacune séparée par un point-virgule.

Exemple (code Java 7) :

```
try {
  try (BufferedReader bufferedReader = new BufferedReader(new
    FileReader("C:/Users/jm/AppData/Local/Temp/monfichier.txt"))) {
    String ligne=null;
    while ((ligne = bufferedReader.readLine()) != null) {
      System.out.println(ligne);
    }
  }
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

Dans l'exemple ci-dessus, la ressource de type BufferedReader sera fermée proprement à la fin normale ou anormale des traitements.

Les ressources sont implicitement final : il n'est donc pas possible de leur affecter une nouvelle instance dans le bloc de l'instruction try.

Une instruction try avec ressources peut avoir des clauses catch et finally comme une instruction try classique. Avec l'instruction try avec ressources, les clauses catch et finally sont exécutées après que la ou les ressources ont été fermées.

Exemple (code Java 7) :

```
try (BufferedReader bufferedReader = new
  BufferedReader(new FileReader("C:/Users/jm/AppData/Local/Temp/monfichier.txt"))) {
  String ligne=null;
  while ((ligne = bufferedReader.readLine()) != null) {
    System.out.println(ligne);
  }
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

Il est possible de déclarer plusieurs ressources dans une même instruction try avec ressources, chacune séparée par un caractère point-virgule. Dans ce cas, la méthode close() des ressources déclarées est invoquée dans l'ordre inverse de leur déclaration.

L'instruction try-with-resource présente un petit inconvénient : il est obligatoire de définir la variable qui encapsule la ressource entre les parenthèses qui suivent l'instruction try. Il n'est par exemple pas possible de fournir en paramètre de l'instruction try une instance déjà créé.

Exemple (code Java 7) :

```
package fr.jmdoudoux.dej.java7;

import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.Reader;

public class TestTryWithRessources {

  public static void main(String[] args) {
    FileReader fr;
    try {
      fr = new FileReader("C:/Users/jm/AppData/Local/Temp/monfichier.txt");
      afficherFichier(fr);
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }

  public static void afficherFichier(Reader flux) throws IOException {
    try (flux) {
      int donnee;
      while ((donnee = flux.read()) >= 0) {
        System.out.print((char) donnee);
      }
    }
  }
}
```

Le compilateur génère une erreur lors de la compilation de ce code.

Résultat :

```
C:\Users\jm\java\JavaApplication1\src\com\jmdoudoux\test\java7>javac
TestTryWithRessources.java
TestTryWithRessources.java:22: error:
<identifier> expected
   try (flux) {
             ^
TestTryWithRessources.java:22: error: ')' expected
   try (flux) {
              ^
TestTryWithRessources.java:22: error: '{' expected
   try (flux) {
                ^
TestTryWithRessources.java:23: error: not a statement
     int donnee;
          ^
4 errors
```

L'exemple ci-dessus génère une erreur à la compilation puisqu'aucune variable n'est définie entre les parenthèses de l'instruction try.

Pour pallier ce petit inconvénient, il est possible de définir une variable et de l'initialiser avec l'instance existante.

Exemple (code Java 7) :

```
package fr.jmdoudoux.dej.java7;

import java.io.FileReader;
import java.io.IOException;
import java.io.Reader;

public class TestTryWithRessources {

  public static void main(String[] args) {
    FileReader fr;
    try {
      fr = new FileReader("C:/Users/jm/AppData/Local/Temp/monfichier.txt");
      afficherFichier(fr);
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }

  public static void afficherFichier(Reader flux) throws IOException {
    try (Reader closeableReader = flux) {
      int donnee;
      while ((donnee = flux.read()) >= 0) {
        System.out.print((char) donnee);
      }
    }
  }
}
```

Dans l'exemple ci-dessus, comme la variable définie et celle existante pointent sur la même référence, les deux variables peuvent être utilisées indifféremment. L'instruction try-with-resource se charge de fermer automatiquement le flux.

Attention, seules les ressources déclarées dans l'instruction try seront fermées automatiquement. Si une ressource est explicitement instanciée dans le bloc try, la gestion de la fermeture et de l'exception qu'elle peut lever doit être gérée par le développeur.

Une exception peut être levée dans le bloc de l'instruction try mais aussi durant l'invocation de la méthode close() de la ou des ressources déclarées. La méthode close() pouvant lever une exception, celle-ci pourrait masquer une éventuelle exception levée dans le bloc de code de l'instruction try.

Il est obligatoire de gérer l'exception pouvant être levée par la méthode close() de la ressource soit en la capturant pour la traiter soit en propageant cette exception pour laisser le soin de son traitement à la méthode appelante.

Exemple (code Java 7) :

```
package fr.jmdoudoux.dej;

public class MaRessource implements AutoCloseable {
  @Override
  public void close() throws MonException {
    throw new MonException("Erreur durant la fermeture");
  }
}
```

Exemple (code Java 7) :

```
package fr.jmdoudoux.dej;

public class TestMaRessource {
  public static void main(String[] args) {
    try (MaRessource res = new MaRessource()) {
      // utilisation da la ressource
    }
  }
}
```

Résultat :

```
C:\eclipse helios\workspace\TestJava\src>javac
com\jmdoudoux\test\TestMaRessource.java
com\jmdoudoux\test\TestMaRessource.java:6: error: unreported
exception MonException; must be caught or declared to be thrown
    try (MaRessource
res = new MaRessource()) {
                     ^
  exception thrown from implicit call to close() on resource variable 'res'
1 error
```

Cette exemple ne se compile pas car l'exception pouvant être levée lors de l'invocation de la méthode close() n'est pas gérée.

Les exemples suivants utilisent deux exceptions personnalisées.

Exemple :

```
package fr.jmdoudoux.dej.java7;

public class MonException1 extends Exception{
    public MonException1(String message){
        super(message);
    }
}

package fr.jmdoudoux.dej.java7;

public class MonException2 extends Exception{
    public MonException2(String message){
        super(message);
    }
}
```

Une ressource générique est définie : elle possède une méthode utiliser() et une redéfinition de la méthode close() car elle implémente l'interface AutoCloseable. Durant leur exécution, ces deux méthodes lèvent une exception.

Exemple (code Java 7) :

```
package fr.jmdoudoux.dej.java7;

public class MaRessource implements AutoCloseable {
  private String nom;

  public MaRessource(String nom) {
    this.nom = nom;
  }

  public String getNom() {return nom;}

  public void utiliser() throws MonException1{
    System.out.println("Utilisation de la ressource "+nom);
    throw new MonException1("Erreur durant l'utilisation de la ressource "+nom);
  }

  @Override
  public void close() throws MonException2{
    System.out.println("Fermeture de la ressource"+nom);
    throw new MonException2("Erreur durant la fermeture de la ressource "+nom);
  }
}
```

La ressource peut être utilisée dans du code compatible avec la version 6 de Java.

Exemple (code Java 7) :

```
package fr.jmdoudoux.dej.java7;

public class TestRessourceJ6 {

  public static void  main(String[] args) {
    MaRessource res = null;
    try {
      res = new MaRessource("Ressource1");
      res.utiliser();
    } catch (Exception e) {
       e.printStackTrace();
    } finally{
      try {
        res.close();
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
  }
}
```

Résultat :

```
Utilisation de la ressource Ressource1
Fermeture de la ressourceRessource1
fr.jmdoudoux.dej.java7.MonException1:
Erreur durant l'utilisation de la ressource Ressource1
      at fr.jmdoudoux.dej.java7.MaRessource.utiliser(MaRessource.java:14)
      at fr.jmdoudoux.dej.java7.TestRessourceJ6.main(TestRessourceJ6.java:10)
fr.jmdoudoux.dej.java7.MonException2:
Erreur durant la fermeture de la ressource Ressource1
      at fr.jmdoudoux.dej.java7.MaRessource.close(MaRessource.java:20)
      at fr.jmdoudoux.dej.java7.TestRessourceJ6.main(TestRessourceJ6.java:15)
```

L'utilisation de la ressource avec l'instruction try-with-resource de Java 7 simplifie le code.

Exemple (code Java 7) :

```
package fr.jmdoudoux.dej.java7;

public class TestRessourceJ7 {
  public static void main(String[] args) {
    try(MaRessource res = new MaRessource("Ressource1")){
      res.utiliser();
    } catch(Exception e){
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
Utilisation de la ressource Ressource1
fr.jmdoudoux.dej.java7.MonException1:
Erreur durant l'utilisation de la ressource Ressource1
Fermeture de la ressourceRessource1
      at fr.jmdoudoux.dej.java7.MaRessource.utiliser(MaRessource.java:14)
      at fr.jmdoudoux.dej.java7.TestRessourceJ7.main(TestRessourceJ7.java:6)
      Suppressed:
fr.jmdoudoux.dej.java7.MonException2:
Erreur durant la fermeture de la ressource Ressource1
            at fr.jmdoudoux.dej.java7.MaRessource.close(MaRessource.java:20)
            at fr.jmdoudoux.dej.java7.TestRessourceJ7.main(TestRessourceJ7.java:7)
```

Le résultat est aussi légèrement différent : c'est l'exception levée lors de l'utilisation de la ressource qui est propagée et non l'exception levée lors de la fermeture de la ressource.

Si une exception est levée dans le bloc try et lors de la fermeture de la ressource, c'est l'exception du bloc try qui est propagée et l'exception levée lors de la fermeture est masquée.

Pour obtenir l'exception masquée, il est possible d'invoquer la méthode getSuppressed() de la classe Throwable sur l'instance de l'exception qui est propagée.

L'ARM fonctionne aussi si plusieurs ressources sont utilisées dans plusieurs instructions try-with-resources imbriquées.

Exemple (code Java 7) :

```
package fr.jmdoudoux.dej.java7;

public class TestRessourcesJ7 {
  public static void main(String[] args) {
    try (MaRessource res1 = new MaRessource("Ressource1");
         MaRessource res2 = new MaRessource("Ressource2")) {
      try (MaRessource res3 = new MaRessource("Ressource3")) {
        res3.utiliser();
      } catch (Exception e) {
        e.printStackTrace();
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
Utilisation de la ressource Ressource3
Fermeture de la ressource Ressource3
Fermeture de la ressource Ressource2
Fermeture de la ressource Ressource1
fr.jmdoudoux.dej.java7.MonException1:
Erreur durant l'utilisation de la ressource Ressource3
      at fr.jmdoudoux.dej.java7.MaRessource.utiliser(MaRessource.java:14)
      at fr.jmdoudoux.dej.java7.TestRessourcesJ7.main(TestRessourcesJ7.java:8)
      Suppressed:
fr.jmdoudoux.dej.java7.MonException2: Erreur durant la fermeture de la
ressource Ressource3
            at fr.jmdoudoux.dej.java7.MaRessource.close(MaRessource.java:20)
            at fr.jmdoudoux.dej.java7.TestRessourcesJ7.main(TestRessourcesJ7.java:9)
fr.jmdoudoux.dej.java7.MonException2:
Erreur durant la fermeture de la ressource Ressource2
      at fr.jmdoudoux.dej.java7.MaRessource.close(MaRessource.java:20)
      at fr.jmdoudoux.dej.java7.TestRessourcesJ7.main(TestRessourcesJ7.java:12)
      Suppressed: fr.jmdoudoux.dej.java7.MonException2:
Erreur durant la fermeture de la ressource Ressource1
            ... 2 more
```

Toutes les exceptions levées lors de la fermeture des ressources sont inhibées et peuvent être obtenues en invoquant la méthode getSuppressed().

Exemple (code Java 7) :

```
package fr.jmdoudoux.dej.java7;

public class TestRessourcesJ7 {
  public static void main(String[] args) {
    try (MaRessource res1 = new MaRessource("Ressource1");
         MaRessource res2 = new MaRessource("Ressource2")) {
      try (MaRessource res3 = new MaRessource("Ressource3")) {
        res3.utiliser();
      }
    } catch (Exception e) {
      System.out.println("Exception : " +
        e.getClass().getSimpleName() + " : " + e.getMessage());
      if (e.getSuppressed() != null) {
        for (Throwable t : e.getSuppressed()) {
          System.out.println(t.getClass().getSimpleName() +
            " : " + t.getMessage());
        }
      }
    }
  }
}
```

Résultat :

```
Utilisation de la ressource Ressource3
Fermeture de la ressource Ressource3
Fermeture de la ressource Ressource2
Fermeture de la ressource Ressource1
Exception : MonException1 : Erreur durant l'utilisation de la ressource
Ressource3
MonException2 : Erreur durant la fermeture de la ressource Ressource3
MonException2 : Erreur durant la fermeture de la ressource Ressource2
MonException2 : Erreur durant la fermeture de la ressource Ressource1
```

La méthode getSuppressed() renvoie un tableau d'instances de Throwable qui contient les exceptions capturées lors de la fermeture des ressources et non propagées.

La classe Throwable est aussi enrichie d'un nouveau constructeur qui permet de prendre en compte ou non des exceptions supprimées. Si le booléen enableSuppression est à false, alors la méthode getSuppressed() renvoie un tableau vide et l'invocation de la méthode addSuppressed() n'aura aucun effet.

9.8. Des types plus précis lorsqu'une exception est relevée dans une clause catch

Il est possible de repropager une exception qui a été gérée par une instruction catch en utilisant le mot clé throw.

Avant Java 7, il n'était pas possible de relever une exception qui soit un super-type de l'exception capturée dans une clause catch : dans ce cas, le compilateur émettait une erreur "unreported exception Exception; must be caught or declared to be thrown".

Dans l'exemple ci-dessous, l'exception MonExceptionFille hérite de l'exception MonExceptionMere.

Exemple :

```
public void maMethode() throws MonExceptionMere {

  try {
    // traitement
    throw new MonExceptionFille();
  } catch(MonExceptionMere e) {
    throw e;
  }
}
```

Java 7 propose une analyse plus fine de la situation et permet de déclarer la levée d'une exception de type MonExceptionFille même si l'exception gérée et relevée est de type MonExceptionMere.

Exemple (code Java 7) :

```
  public void maMethode() throws MonExceptionFille {
    try {
      // traitement
      throw new MonExceptionFille();
    } catch (MonExceptionMere e) {
      throw e;
    }
  }
```

Avant Java 7, cette portion de code aurait provoqué une erreur de compilation « unreported exception MonExceptionMere ». Ceci s'applique aussi pour plusieurs exceptions.

Exemple :

```
public class MaClasse {
  public void maMethode(boolean valeur) throws MonExceptionA,
      MonExceptionB {
    try {
      if (valeur) {
        throw new MonExceptionA();
      } else {
        throw new MonExceptionB();
      }
    } catch (Exception e) {
      throw e;
    }
  }

  static class MonExceptionA extends Exception { }
  static class MonExceptionB extends Exception { }
}
```

Résultat :

```
C:\eclipse helios\workspace\TestJava\src>javac MaClasse.java
MaClasse.java:11:
unreported exception java.lang.Exception; must be caught or declared to be thrown
      throw e;
      ^
1 error
```

Le compilateur vérifie si le type d'une exception levée dans un bloc catch correspond à un des types d'exceptions déclaré dans la clause throws de la méthode. Si le type de l'exception capturée par la clause catch est Exception alors la clause throws ne peut pas être d'un de ses sous-types.

Exemple :

```
public class MaClasse {
  public void maMethode(boolean valeur) throws Exception {
    try {
      if (valeur) {
        throw new MonExceptionA();
      } else {
        throw new MonExceptionB();
      }
    } catch (Exception e) {
      throw e;
    }
  }

  static class MonExceptionA extends Exception { }
  static class MonExceptionB extends Exception { }
}
```

Pour déclarer dans la clause throws les exceptions précises, il faut les capturer individuellement dans des clauses catch dédiées.

Exemple :

```
public class MaClasse {
  public void maMethode(boolean valeur) throws MonExceptionA,
      MonExceptionB {
    try {
      if (valeur) {
        throw new MonExceptionA();
      } else {
        throw new MonExceptionB();
      }
    } catch (MonExceptionA e) {
      throw e;
    } catch (MonExceptionB e) {
      throw e;
    }
  }

  static class MonExceptionA extends Exception { }
  static class MonExceptionB extends Exception { }
}
```

Le compilateur de Java 7 effectue une analyse plus fine qui lui permet de connaitre précisément les exceptions qui peuvent être relevées indépendamment du type déclaré dans la clause catch qui va les capturer. Il est ainsi possible de capturer un super-type des exceptions qui seront relevées et déclarer le type précis des exceptions dans la clause throws.

Lorsqu'une clause catch déclare plusieurs types d'exceptions et relève l'exception dans son bloc de code, le compilateur vérifie :

- Que le code du bloc try peut lever les exceptions déclarées
- Qu'aucune autre clause catch ne déclare prendre en charge un des types d'exceptions
- Que l'exception relevée est du type ou un sous-type d'un des types d'exceptions déclaré dans la clause catch

Exemple (code Java 7) :

```
public class MaClasse {

  public void maMethode(boolean valeur) throws MonExceptionA,
      MonExceptionB {
    try {
      if (valeur) {
        throw new MonExceptionA();
      } else {
        throw new MonExceptionB();
      }
    } catch (Exception e) {
      throw e;
    }
  }

  static class MonExceptionA extends Exception { }
  static class MonExceptionB extends Exception { }
}
```

Attention cependant, il y a un cas ou la compatibilité du code antérieur n'est pas assurée avec Java 7 : ce cas concerne l'imbrication de deux try/catch quand le second bloc apparaît dans la clause catch du premier try. Le code du bloc try imbriqué lève une exception.

L'exemple ci-dessous se compile sans problème avec Java 6 :

Exemple :

```
  public void maMethode() throws MonExceptionMere {
    try {
    // traitement
    throw new MonExceptionFille();
    } catch (MonExceptionMere e) {
      try {
        // traitement
        throw e;
      } catch (MonExceptionFille2 mem) {
      }
    }
  }
```

Ce même code ne se compile plus avec Java 7 car l'exception de type MonExceptionMere ne sera jamais traitée par la seconde clause catch.

Pour être compilé en Java 7, le code devra être modifié.

9.9. Multiples exceptions dans une clause catch

Java SE 7 propose une amélioration de la gestion des exceptions en permettant le traitement de plusieurs exceptions dans une même clause catch.

Il n'est pas rare d'avoir à dupliquer les mêmes lignes de code dans le bloc de code de plusieurs clauses catch().

Exemple :

```
try {
  // traitements pouvant lever les exceptions
} catch(ExceptionType1 e1) {
  // Traitement de l'exception
} catch(ExceptionType2 e2) {
  // Traitement de l'exception
} catch(ExceptionType3 e3) {
  // Traitement de l'exception
}
```

Avant Java 7, il était difficile d'éviter la duplication de code car chaque exception est de type différent.

Une solution utilisée pour éviter cette duplication est de catcher un super-type d'exception, généralement le type Exception. Cependant cette solution a plusieurs effets de bord, notamment le fait que le traitement s'appliquera à toutes les exceptions filles et englobera peut-être des exceptions qui auraient nécessité un traitement particulier. De plus, il ne sera pas possible de propager un autre type d'exception que celui capturé.

A partir de Java 7, la même portion de code est simplifiée : il suffit de déclarer les exceptions dans une même clause catch en les séparant par le caractère "|".

Exemple (code Java 7) :

```
try {
  // traitements pouvant lever les exceptions
} catch(ExceptionType1|ExceptionType2|ExceptionType3 ex) {
  // Traitement de l'exception
}
```

Il n'est plus nécessaire de définir un bloc catch pour chaque exception et de dupliquer le code du bloc si c'est le même pour tous.

La clause catch peut contenir plusieurs types d'exceptions qui provoqueront l'exécution du bloc de code associé, chaque type d'exception est séparé d'un autre en utilisant le caractère barre verticale.

Il est possible d'utiliser plusieurs blocs catch notamment si les traitements des exceptions sont différents selon leur type.

Exemple (code Java 7) :

```
try {
  // traitements pouvant lever les exceptions
} catch(ExceptionType1|ExceptionType2|ExceptionType3 ex) {
  // Traitement de l'exception
} catch(ExceptionType4|ExceptionType5 ex) {
  // Traitement de l'exception
}
```

Si plusieurs types d'exceptions sont déclarés dans une clause catch alors la variable qui permettra un accès à l'exception concernée est implicitement déclarée final.

Le paramètre de la clause catch étant implicitement final, il n'est pas possible de réaffecter sa valeur dans le bloc de code dans lequel il est défini.

Exemple (code Java 7) :

```
public class MaClasse {

  public void rethrowException(boolean valeur) throws MonExceptionA,
      MonExceptionB {
    try {
      if (valeur) {
        throw new MonExceptionA();
      } else {
        throw new MonExceptionB();
      }
    } catch (MonExceptionA|MonExceptionB e) {
      e = new MonExceptionB();
      throw e;
    }
  }

  static class MonExceptionA extends Exception { }
  static class MonExceptionB extends Exception { }
}
```

Résultat :

```
C:\eclipse helios\workspace\TestJava\src>javac MaClasse.java
MaClasse.java:11:
error: multi-catch parameter e may not be assigned
      e =
new MonExceptionB();
      ^
1 error
```

C'est le compilateur qui prend en charge la génération du code correspondant au support multi exceptions de la clause catch sans duplication de code.

L'avantage de cette gestion de plusieurs exceptions dans une clause catch n'est pas seulement syntaxique car il réduit la quantité de code produite. Le bytecode généré par le compilateur est meilleur comparé à celui produit pour plusieurs clauses catch équivalentes.

10. Les énumérations (type enum)

Chapitre 10

Niveau : 🟩🟩⬜⬜⬜⬜ Elémentaire

Souvent lors de l'écriture de code, il est utile de pouvoir définir un ensemble fini de valeurs d'une donnée ; par exemple, pour définir les valeurs possibles qui vont caractériser l'état de cette donnée.

Pour cela, le type énumération permet de définir un ensemble de constantes : une énumération est un ensemble fini d'éléments constants. Cette fonctionnalité existe déjà dans les langages C et Delphi, entre autres.

Jusqu'à la version 1.4 incluse, la façon la plus pratique pour palier le manque du type enum était de créer des constantes dans une classe.

Exemple :

```
public class FeuTricolore {
  public static final int VERT = 0;
  public static final int ORANGE = 1;
  public static final int ROUGE = 2;
}
```

Cette approche fonctionne : les constantes peuvent être sérialisées et utilisées dans une instruction switch mais leur mise en oeuvre n'est pas type safe. Rien n'empêche d'affecter une autre valeur à la donnée de type int qui va stocker une des valeurs constantes.

A défaut, cette solution permet de répondre au besoin mais elle présente cependant quelques inconvénients :

- le principal inconvénient de cette technique est qu'il n'y a pas de contrôle sur la valeur affectée à une donnée surtout si les constantes ne sont pas utilisées : il est possible d'utiliser n'importe quelle valeur permise par le type de la variable en plus des constantes définies. Le compilateur ne peut faire aucun contrôle sur les valeurs utilisées
- il n'est pas possible de faire une itération sur chacune des valeurs
- il n'est pas possible d'associer des traitements à une constante
- les modifications faites dans ces constantes notamment les changements de valeurs ne sont pas automatiquement reportées dans les autres classes qui doivent être explicitement recompilées

Java 5 apporte un nouveau type nommé enum qui permet de définir un ensemble de champs constants. Cette nouvelle fonctionnalité est spécifiée dans la JSR 201.

Un exemple classique est l'énumération des jours de la semaine.

Exemple :

```
public enum Jour {
  LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE
}
```

Les énumérations permettent de définir un ensemble fini de constantes, chacune d'entre-elles est séparée des autres par une virgule. Comme ces champs sont constants, leur nom est en majuscule par convention.

Ce chapitre contient plusieurs sections :

- Les énumérations locales

10.0.1. La définition d'une énumération

La définition d'une énumération ressemble à celle d'une classe avec quelques différences :

- utilisation du mot clé enum introduit spécifiquement dans ce but à la place du mot clé class
- un ensemble de valeurs constantes définies au début du corps de la définition, chaque valeur étant séparée par une virgule
- par convention le nom des constantes est en majuscule

Une énumération peut prendre plusieurs formes et être enrichie de fonctionnalités puisqu'une énumération est une classe Java.

Dans sa forme la plus simple, la déclaration d'une énumération se résume à définir l'ensemble des constantes.

Exemple :

```
public enum FeuTricolore {
  VERT, ORANGE, ROUGE
};
```

Les énumérations peuvent être déclarées à plusieurs niveaux. Le mot clé enum est au même niveau que le mot clé class ou interface : une énumération peut donc être déclarée au même endroit qu'une classe ou une interface, que cela soit dans un fichier dédié ou dans le fichier d'une autre classe.

Exemple :

```
public class TestEnum2 {
  public enum MonStyle {
    STYLE_1, STYLE_2, STYLE_3, STYLE_4, STYLE_5
  };

  public static void main(String[] args) {
    afficher(MonStyle.STYLE_2);
  }

  public static void afficher(MonStyle style) {
    System.out.println(style);
  }
}
```

Lors de la compilation de cet exemple, une classe interne est créée pour encapsuler l'énumération.

Résultat :

```
C:\java\workspace\TestEnum\bin>dir
 Le volume dans le lecteur C s'appelle Disque_C
 Le numéro de série du volume est 043F-2ED6

 Répertoire de C:\java\workspace\TestEnum\bin

15/07/2010  16:33    <REP>          .
15/07/2010  16:33    <REP>          ..
15/07/2010  16:39             1 160 TestEnum2$MonStyle.class
15/07/2010  16:39               743 TestEnum2.class
               2 fichier(s)           1 903 octets
               2 Rép(s)  23 175 589 888 octets libres
```

Les modificateurs d'accès s'appliquent à une énumération.

L'outil Javadoc recense les énumérations dans le fichier package-summary.html.

10.0.2. L'utilisation d'une énumération

Le nom utilisé dans la déclaration de l'énumération peut être utilisé comme n'importe quelle classe dans la déclaration d'un type.

Une fois définie, il est possible d'utiliser l'énumération simplement en définissant une variable du type de l'énumération.

Exemple :

```
public class TestEnum {
  Jour jour;

  public TestEnum(Jour jour) {
    this.jour = jour;
  }

  public void afficherJour() {
    switch (jour) {
    case LUNDI:
      System.out.println("Lundi");
      break;
    case MARDI:
      System.out.println("Mardi");
      break;
    case MERCREDI:
      System.out.println("Mercredi");
      break;
    case JEUDI:
      System.out.println("Jeudi");
      break;
    case VENDREDI:
      System.out.println("Vendredi");
      break;
    case SAMEDI:
      System.out.println("Samedi");
      break;
    case DIMANCHE:
      System.out.println("Dimanche");
      break;
    }
  }

  public static void main(String[] args) {
    TestEnum testEnum = new TestEnum(Jour.SAMEDI);
    testEnum.afficherJour();
  }
}
```

Lors de l'utilisation d'une constante, son nom doit être préfixé par le nom de l'énumération sauf dans le cas de l'utilisation dans une instruction switch.

Les énumérations étant transformées en une classe par le compilateur, ce dernier effectue une vérification de type lors de l'utilisation de l'énumération..

L'instruction switch a été modifiée pour permettre de l'utiliser avec une énumération puisque bien qu'étant physiquement une classe, celle-ci possède une liste finie de valeurs associées.

L'utilisation d'une énumération dans l'instruction switch impose de n'utiliser que le nom de la valeur sans la préfixer par le nom de l'énumération sinon une erreur est émise par le compilateur.

Exemple :

```
    switch(feu) {
    case (FeuTricolore.VERT) :
      System.out.println("passer");
      break;
    default :
      System.out.println("arreter");
      break;
    }
```

Résultat :

```
Feu.java:24: an enum switch case label must be the unqualified name of an enumeration constant
```

Chaque élément d'une énumération est associé à une valeur par défaut, qui débute à zéro et qui est incrémentée de un en un. La méthode ordinal() permet d'obtenir cette valeur.

Exemple :

```
    FeuTricolore feu = FeuTricolore.VERT;
    System.out.println(feu.ordinal());
```

Il y a plusieurs avantages à utiliser les enums à la place des constantes notamment le typage fort et le préfixe de l'élément par le nom de l'énumération.

10.0.3. L'enrichissement de l'énumération

Une énumération peut mettre en oeuvre la plupart des fonctionnalités et des comportements d'une classe :

- implémenter une ou plusieurs interfaces
- avoir plusieurs constructeurs
- avoir des champs et des méthodes
- avoir un bloc d'initialisation statique
- avoir des classes internes (inner classes)

Un type enum hérite implicitement de la classe java.lang.Enum : il ne peut donc pas hériter d'une autre classe.

Exemple :

```
public enum MonEnum extends Object {
  UN, DEUX, TROIS;
}
```

Résultat :

```
MyType.java:3: '{' expected
public enum MonEnum extends Object {
MonEnum.java:6: expected
2 errors
```

Chacun des éléments de l'énumération est instancié par le constructeur sous la forme d'un champ public static.

Si les éléments de l'énumération sont définis sans argument alors un constructeur sans argument doit être proposé dans la définition de l'énumération (celui-ci peut être le constructeur par défaut si aucun autre constructeur n'est défini).

Le fait qu'une énumération soit une classe permet de définir un espace de nommage pour ses éléments ce qui évite les collisions, par exemple Puissance.ELEVEE et Duree.ELEVEE.

A partir du code source de l'énumération, le compilateur va générer une classe enrichie avec certaines fonctionnalités.

Exemple :

```
C:\Users\Jean Michel\workspace\TestEnum\bin>javap FeuTricolore
Compiled from "FeuTricolore.java"
public final class FeuTricolore extends java.lang.Enum{
    public static final FeuTricolore VERT;
    public static final FeuTricolore ORANGE;
    public static final FeuTricolore ROUGE;
    static {};
    public static FeuTricolore[] values();
    public static FeuTricolore valueOf(java.lang.String);
}
```

La classe compilée a été enrichie automatiquement par le compilateur qui a identifié l'entité comme une énumération grâce au mot clé enum :

- la classe compilée hérite de la classe java.lang.Enum.
- un champ public static final du type de l'énumération est ajouté pour chaque élément
- un bloc d'initialisation static permet de créer les différentes instances statiques des éléments
- les méthodes valueOf() et values() sont ajoutées

Le compilateur ajoute automatiquement certaines méthodes à une classe de type enum lors de la compilation, notamment les méthodes statiques :

- values() qui renvoie un tableau des valeurs de l'énumération
- valueOf() qui renvoie l'élément de l'énumération dont le nom est fourni en paramètre

Le nom fourni en paramètre de la méthode valueOf() doit correspondre exactement à l'identifiant utilisé dans la déclaration de l'énumération. Il n'est pas possible de redéfinir la méthode valueOf().

Une énumération propose une implémentation par défaut de la méthode toString() : par défaut, elle renvoie le nom de la constante. Il est possible de la redéfinir au besoin.

Il est possible de préciser une valeur pour chaque élément de l'énumération lors de sa définition : celle-ci sera alors stockée et pourra être utilisée dans les traitements.

Exemple :

```
public enum Coefficient {
  UN(1), DEUX(2), QUATRE(4);

  private final int valeur;

  private Coefficient(int valeur) {
    this.valeur = valeur;
  }

  public int getValeur() {
    return this.valeur;
  }
}
```

Dans ce cas, l'énumération doit implicitement définir :

- un attribut qui contient la valeur associée à l'élément
- un constructeur qui attend en paramètre la valeur
- une méthode de type getter pour obtenir la valeur

Attention : toutes les données manipulées dans un élément d'une énumération doivent être immuables. Par exemple, il ne faut pas encapsuler dans un élément d'une énumération une donnée dont la valeur peut fluctuer dans le temps puisque l'élément est instancié une seule et unique fois.

Il faut obligatoirement définir les constantes en premier, avant toute définition de champs ou de méthodes. Si l'enum contient des champs et/ou des méthodes, il est impératif de terminer la définition des contantes par un point virgule.

Il est aussi possible de fournir plusieurs valeurs à un élément de l'énumération : comme une énumération est une classe, il est possible d'associer plusieurs valeurs à un élément de l'énumération. Ces valeurs seront stockées sous la forme de propriétés et l'énumération doit fournir un constructeur qui doit accepter en paramètre les valeurs de chaque propriété.

Exemple :

```
import java.math.BigDecimal;

public enum Remise {

  COURANTE(new BigDecimal("0.05"), "Remise de 5%"),
  FIDELITE(new BigDecimal("0.07"), "Remise de 7%"),
  EXCEPTIONNELLE(new BigDecimal("0.10"), "Remise de 10%");

  private final BigDecimal taux;
  private final String libelle;

  private Remise(BigDecimal taux, String libelle) {
    this.taux = taux;
    this.libelle = libelle;
  }

  public BigDecimal getTaux() {
    return this.taux;
  }

  public String getLibelle() {
    return this.libelle;
  }

  public BigDecimal calculer(BigDecimal valeur) {
    return valeur.multiply(taux).setScale(2, BigDecimal.ROUND_FLOOR);
  }

  public static void main(String[] args) {
    BigDecimal montant = new BigDecimal("153.99");

    for (Remise remise : Remise.values()) {
      System.out.println(remise.getLibelle() + " \t"
        + remise.calculer(montant));
    }
  }
}
```

Résultat :

```
Remise de 5%   7.69
Remise de 7%   10.77
Remise de 10%  15.39
```

Dans l'exemple précédent, chaque constante est définie avec les deux paramètres qui la compose : le taux et le libellé. Ces valeurs sont passées au constructeur par le bloc d'initialisation static qui est créé par le compilateur.

Le constructeur d'une classe de type enum ne peut pas être public car il ne doit être invoqué que par la classe elle-même pour créer les constantes définies au début de la déclaration de l'énumération.

Un élément d'une énumération ne peut avoir que la valeur avec laquelle il est défini dans sa déclaration. Ceci justifie que le constructeur ne soit pas public et qu'une énumération ne puisse pas avoir de classes filles.

Tous les éléments de l'énumération sont encapsulés dans une instance finale du type de l'énumération : ce sont donc des singletons. De plus, les valeurs peuvent être testées avec l'opérateur == puisqu'elles sont déclarées avec le modificateur final.

Plusieurs fonctionnalités permettent de s'assurer qu'il n'y aura pas d'autres instances que celles définies par le compilateur à partir du code source :

- il n'y a pas de constructeur public qui permette de l'invoquer pour créer une nouvelle instance
- il n'est pas possible d'hériter d'une autre classe que la classe Enum
- il n'est pas possible de créer une classe fille de l'énumération puisqu'elle est déclarée finale
- l'invocation de la méthode clone() de l'énumération lève une exception de type CloneNotSupportedException

Une énumération peut implémenter une ou plusieurs interfaces. Comme une énumération est une classe, elle peut aussi contenir une méthode main().

10.0.4. La personnalisation de chaque élément

Le type Enum de Java est plus qu'une simple liste de constantes car une énumeration est définie dans une classe. Une classe de type Enum peut donc contenir des champs et des méthodes dédiées.

Pour encore plus de souplesse, il est possible de définir chaque élément sous la forme d'une classe interne dans laquelle on fournit une implémentation particulière pour chaque élément.

Il est ainsi possible de définir explicitement, pour chaque valeur, le corps de la classe qui va l'encapsuler. Une telle définition est similaire à la déclaration d'une classe anonyme. Cette classe est implicitement une extension de la classe englobante. Il est ainsi possible de redéfinir une méthode de l'énumération.

Exemple :

```
import java.math.BigDecimal;

public enum Remise {

  COURANTE(new BigDecimal("0.05"), "Remise de 5%") {
    @Override
    public String toString() {
      return "Remise 5%";
    }
  },
  FIDELITE(new BigDecimal("0.07"), "Remise de 7%") {
    @Override
    public String toString() {
      return "Remise fidelite 7%";
    }
  },
  EXCEPTIONNELLE(new BigDecimal("0.10"), "Remise de 10%") {
    @Override
    public String toString() {
      return "Remise exceptionnelle 10%";
    }

    @Override
    public String getLibelle() {
      return "Remise à titre exceptionnel de 10%";
    }
  };

  private final BigDecimal taux;
  private final String libelle;

  private Remise(BigDecimal taux, String libelle) {
    this.taux = taux;
    this.libelle = libelle;
  }

  public BigDecimal getTaux() {
    return this.taux;
  }

  public String getLibelle() {
    return this.libelle;
  }

  public BigDecimal calculer(BigDecimal valeur) {
    return valeur.multiply(taux).setScale(2, BigDecimal.ROUND_FLOOR);
  }

  public static void main(String[] args) {
    BigDecimal montant = new BigDecimal("153.99");

    for (Remise remise : Remise.values()) {
      System.out.println(remise + " \t" + remise.calculer(montant));
    }
  }
}
```

Résultat :

```
Remise 5%      7.69
Remise fidelite 7%     10.77
Remise exceptionnelle 10%      15.39
```

Il est aussi possible de définir une méthode abstract dans l'énumération pour forcer la définition de la méthode dans chaque élément.

Exemple :

```
import java.math.BigDecimal;

public enum Remise {

  COURANTE(new BigDecimal("0.05"), "Remise de 5%") {

    @Override
    public String getLibelle() {
      return this.libelle;
    }

  },
  FIDELITE(new BigDecimal("0.07"), "Remise de 7%") {

    @Override
    public String getLibelle() {
      return "Remise fidélité de 7%";
    }
  },
  EXCEPTIONNELLE(new BigDecimal("0.10"), "Remise de 10%") {

    @Override
    public String getLibelle() {
      return "Remise à titre exceptionnel de 10%";
    }
  };

  private final BigDecimal taux;
  protected final String libelle;

  private Remise(BigDecimal taux, String libelle) {
    this.taux = taux;
    this.libelle = libelle;
  }

  public BigDecimal getTaux() {
    return this.taux;
  }

  public abstract String getLibelle();

  public BigDecimal calculer(BigDecimal valeur) {
    return valeur.multiply(taux).setScale(2, BigDecimal.ROUND_FLOOR);
  }

  public static void main(String[] args) {
    BigDecimal montant = new BigDecimal("153.99");

    for (Remise remise : Remise.values()) {
      System.out.println(remise.getLibelle() + " \t" + remise.calculer(montant));
    }
  }
}
```

Résultat :

```
Remise de 5%   7.69
Remise fidélité de 7%  10.77
Remise à titre exceptionnelle de 10%  15.39
```

Il faut cependant utiliser cette possibilité avec parcimonie car le code est moins lisible.

10.0.5. Les limitations dans la mise en oeuvre des énumérations

La mise en oeuvre des énumérations présente plusieurs limitations.

L'ordre de définition du contenu de l'énumération est important : les éléments de l'énumération doivent être définis en premier.

Un élément d'une énumération ne doit pas être null.

Un type Enum hérite implicitement de la classe java.lang.Enum : il ne peut pas hériter d'une autre classe mère.

Pour garantir qu'il n'y ait qu'une seule instance d'un élément d'une énumération, le constructeur n'est pas accessible et l'énumération ne peut pas avoir de classe fille.

Une énumération ne peut pas être définie localement dans une méthode.

La méthode values() renvoie un tableau des éléments de l'énumération dans l'ordre dans lequel ils sont déclarés mais il ne faut surtout pas utiliser l'ordre des éléments d'une énumération dans les traitements : il ne faut par exemple pas tester la valeur retournée par la méthode ordinal() dans les traitements. Des problèmes apparaitront à l'exécution si l'ordre des éléments est modifié car le compilateur ne peut pas détecter ce type de changement.

Il n'est pas possible de personnaliser la sérialisation d'une énumération en redéfinissant les méthodes writeObject() et writeReplace() qui seront ignorées lors de la sérialisation. De plus, la déclaration d'un serialVersionUlD est ignorée car sa valeur est toujours 0L.

10.1. Les énumérations locales

Jusqu'à Java 15, il n'est pas possible de définir des énumérations locales.

Exemple (code Java 15) :

```
public class TestEnumLocale {

  public void traiter() {
    enum Statut { VALIDE, INVALIDE };
  }
}
```

Résultat :

```
C:\java>javac -version
javac 15

C:\java>javac TestEnumLocale.java
TestEnumLocale.java:4: error: enum types must not be local
    enum Statut { VALIDE, INVALIDE };
    ^
1 error

C:\java>
```

Java 16 permet de définir des énumérations locales, qui ne pourront donc être utilisées que dans la méthode où elles sont définies.

Résultat :

```
C:\java>javac -version
javac 16.0.1

C:\java>javac TestEnumLocale.java

C:\java>
```

Les énumérations locales ne peuvent pas capturer les variables non static du contexte englobant comme les paramètres de la méthode par exemple.

Exemple (code Java 16) :

```
public class TestEnumLocale {

  public void traiter(int valeur) {
    enum Statut {
      VALIDE(valeur), INVALIDE(valeur+1);

      private final int v;

      Statut(int v) {
        this.v = v;
      }
    };
  }
}
```

Résultat :

```
C:\java>javac TestEnumLocale.java
TestEnumLocale.java:5: error: non-static variable valeur cannot be referenced from a static
 context
      VALIDE(valeur), INVALIDE(valeur+1);
             ^
TestEnumLocale.java:5: error: non-static variable valeur cannot be referenced from a static
 context
      VALIDE(valeur), INVALIDE(valeur+1);
                               ^
2 errors

C:\java>
```

Les énumérations locales peuvent capturer les variables static du contexte englobant.

Exemple (code Java 16) :

```
public class TestEnumLocale {

  static int valeur = 10;

  public void traiter() {

    enum Statut {
      VALIDE(valeur), INVALIDE(valeur+1);
      private final int v;

      Statut(int v) {
        this.v = v;
      }
    };
  }
}
```

11. Les annotations

Chapitre 11

Niveau : 🟩🟩🟨⬜⬜⬜ Intermédiaire

Java SE 5 a introduit les annotations qui sont des métadonnées incluses dans le code source. Les annotations ont été spécifiées dans la JSR 175 : leur but est d'intégrer au langage Java des métadonnées.

Des métadonnées étaient déjà historiquement mises en oeuvre avec Java notamment avec Javadoc ou exploitées par des outils tiers notamment XDoclet : l'outil open source XDoclet propose depuis longtemps des fonctionnalités similaires aux annotations. Avant Java 5, seul l'outil Javadoc utilisait des métadonnées en standard pour générer une documentation automatique du code source.

Javadoc propose l'annotation @deprecated qui bien qu'utilisée dans les commentaires permet de marquer une méthode comme obsolète et de faire afficher un avertissement par le compilateur.

Le défaut de Javadoc est d'être trop spécifique à l'activité de génération de documentation même si le tag deprecated est aussi utilisé par le compilateur.

Depuis leur introduction dans Java 5, les annotations sont de plus en plus utilisées dans le développement d'applications avec les plates-formes Java SE et Java EE.

Ce chapitre contient plusieurs sections :

- La présentation des annotations
- La mise en oeuvre des annotations
- L'utilisation des annotations
- Les annotations standard
- Les annotations communes (Common Annotations)
- Les annotations personnalisées
- L'exploitation des annotations
- L'API Pluggable Annotation Processing
- Les ressources relatives aux annotations

11.1. La présentation des annotations

Les annotations de Java 5 apportent une standardisation des métadonnées dans un but généraliste. Ces métadonnées associés aux entités Java peuvent être exploitées à la compilation ou à l'exécution.

Java a été modifié pour permettre la mise en oeuvre des annotations :

- une syntaxe dédiée a été ajoutée dans le langage Java pour permettre la définition et l'utilisation d'annotations.
- le bytecode est enrichi pour permettre le stockage des annotations.

Les annotations peuvent être utilisées avec quasiment tous les types d'entités et de membres de Java : packages, classes, interfaces, constructeurs, méthodes, champs, paramètres, variables ou annotations elles-mêmes.

Java propose plusieurs annotations standard et permet la création de ses propres annotations.

Une annotation précède l'entité qu'elle concerne. Elle est désignée par un nom précédé du caractère @.

Il existe plusieurs catégories d'annotations :

- les marqueurs (markers) : ces annotations ne possèdent pas d'attribut (exemple : @Deprecated, @Override, ...)
- les annotations paramétrées (single value annotations) : ces annotations ne possèdent qu'un seul attribut(exemple : @MonAnnotation("test"))
- les annotations multi paramétrées (full annotations) : ces annotations possèdent plusieurs attributs (exemple : @MonAnnotation(arg1="test 3", arg2="test 2", arg3="test3"))

Les arguments fournis en paramètres d'une annotation peuvent être de plusieurs types : les chaînes de caractères, les types primitifs, les énumérations, les annotations, le type Class.

Les annotations sont définies dans un type d'annotation. Une annotation est une instance d'un type d'annotation. Les paramètres d'une annotation peuvent avoir des valeurs par défaut.

La disponibilité d'une annotation est définie grâce à une retention policy.

Les usages des annotations sont nombreux : génération de documentations, de code, de fichiers, ORM (Object Relational Mapping), ...

Les annotations ne sont guère utiles sans un mécanisme permettant leur exploitation.

Une API est proposée pour assurer ces traitements : elle est regroupée dans les packages com.sun.mirror.apt, com.sun.mirror.declaration, com.sun.mirror.type et com.sun.mirror.util.

L'outil apt (annotation processing tool) permet un traitement des annotations personnalisées durant la phase de compilation (compile time). L'outil apt permet la génération de nouveaux fichiers mais ne permet pas de modifier le code existant.

Il est important de se souvenir que lors du traitement des annotations le code source est parcouru mais il n'est pas possible de modifier ce code.

L'API reflexion est enrichie pour permettre de traiter les annotations lors de la phase d'exécution (runtime).

Java 6 intègre deux JSR concernant les annotations :

- Pluggable Annotation Processing API (JSR 269)
- Common Annotations (JSR 250)

L'Api Pluggable Annotation Processing permet d'intégrer le traitement des annotations dans le processus de compilation du compilateur Java ce qui évite d'avoir à utiliser apt.

Les annotations vont évoluer dans la plate-forme Java notamment au travers de plusieurs JSR qui sont en cours de définition :

- JSR 305 Annotations for Software Defect Detection
- JSR 308 Annotations on Java Types : doit permettre de mettre en oeuvre les annotations sur tous les types notamment les generics et sur les variables locales à l'exécution.

11.2. La mise en oeuvre des annotations

Les annotations fournissent des informations sur des entités : elles n'ont pas d'effets directs sur les entités qu'elles concernent.

Les annotations utilisent leur propre syntaxe. Une annotation s'utilise avec le caractère @ suivi du nom de l'annotation : elle doit obligatoirement précéder l'entité qu'elle annote. Par convention, les annotations s'utilisent sur une ligne dédiée.

Les annotations peuvent s'utiliser sur les packages, les classes, les interfaces, les méthodes, les constructeurs et les paramètres de méthodes.

Exemple :

```
@Override
public void maMethode() {
}
```

Une annotation peut avoir un ou plusieurs attributs : ceux-ci sont précisés entre parenthèses, séparés par une virgule. Un attribut est de la forme clé=valeur.

Exemple :

```
@SuppressWarnings(value = "unchecked")
void maMethode() { }
```

Lorsque l'annotation ne possède qu'un seul attribut, il est possible d'omettre son nom.

Exemple :

```
@SuppressWarnings("unchecked")
void maMethode() { }
```

Un attribut peut être de type tableau : dans ce cas, les différentes valeurs sont fournies entre accolades, chaque valeur placée entre guillemets et séparée de la suivante par une virgule.

Exemple :

```
@SuppressWarnings(value={"unchecked", "deprecation"})
```

Le tableau peut contenir des annotations.

Exemple :

```
@TODOItems({
  @Todo(importance = Importance.MAJEUR,
      description = "Ajouter le traitement des erreurs",
      assigneA = "JMD",
      dateAssignation = "07-11-2007"),
  @Todo(importance = Importance.MINEURE,
      description = "Changer la couleur de fond",
      assigneA = "JMD",
      dateAssignation = "13-12-2007")
})
```

11.3. L'utilisation des annotations

Les annotations prennent une place de plus en plus importante dans la plate-forme Java et dans de nombreuses API open source.

Les utilisations des annotations concernent plusieurs fonctionnalités :

- Utilisation par le compilateur pour détecter des erreurs ou ignorer des avertissements
- Documentation
- Génération de code
- Génération de fichiers

11.3.1. La documentation

Les annotations peuvent être mises en oeuvre pour permettre la génération de documentations indépendantes de JavaDoc : listes de choses à faire, de services ou de composants, ...

Il peut par exemple être pratique de rassembler certaines informations mises sous la forme de commentaires dans des annotations pour permettre leur traitement.

Par exemple, il est possible de définir une annotation qui va contenir les métadonnées relatives aux informations sur une classe. Traditionnellement, une classe débute par un commentaire d'en-tête qui contient des informations sur l'auteur, la date de création, les modifications, ... L'idée est de fournir ces informations dans une annotation dédiée. L'avantage est de facilement extraire et manipuler ces données qui ne seraient qu'informatives sous leur forme de commentaires.

11.3.2. L'utilisation par le compilateur

Les trois annotations fournies en standard avec la plate-forme entrent dans cette catégorie qui consiste à faire réaliser par le compilateur quelques contrôles basiques.

11.3.3. La génération de code

Les annotations sont particulièrement adaptées à la génération de code source afin de faciliter le travail des développeurs notamment sur des tâches répétitives.

Attention, le traitement des annotations ne peut pas modifier le code existant mais simplement créer de nouveaux fichiers sources.

11.3.4. La génération de fichiers

Les API standard ou les frameworks open source nécessitent fréquemment l'utilisation de fichiers de configuration ou de déploiement généralement au format XML.

Les annotations peuvent proposer une solution pour maintenir le contenu de ces fichiers par rapport aux entités incluses dans le code de l'application.

La version 5 de Java EE fait un important usage des annotations dans le but de simplifier les développements de certains composants notamment les EJB, les entités et les services web. Pour cela, l'utilisation de descripteurs est remplacée par l'utilisation d'annotations ce qui rend le code plus facile à développer et plus clair.

11.3.5. Les API qui utilisent les annotations

De nombreuses API standard utilisent les annotations depuis leur intégration dans Java notamment :

- JAXB 2.0 : JSR 222 (Java Architecture for XML Binding 2.0)
- Les services web de Java 6 (JAX-WS) : JSR 181 (Web Services Metadata for the Java Platform) et JSR 224 (Java APIs for XML Web Services 2.0 API)
- Les EJB 3.0 et JPA : JSR 220 (Enterprise JavaBeans 3.0)
- Servlets 3.0, CDI
- ...

De nombreuses API open source utilisent aussi les annotations notamment JUnit, TestNG, Hibernate, ...

11.4. Les annotations standard

Java 5 propose plusieurs annotations standard.

11.4.1. L'annotation @Deprecated

Cette annotation a un rôle similaire au tag de même nom de Javadoc.

C'est un marqueur qui précise que l'entité concernée est obsolète et qu'il ne faudrait plus l'utiliser. Elle peut être utilisée avec une classe, une interface ou un membre (méthode ou champ)

Exemple :

```
public class TestDeprecated {

  public static void main(String[] args) {
    MaSousClasse td = new MaSousClasse();
    td.maMethode();
  }
}

@Deprecated
class MaSousClasse {

  /**
   * Afficher un message de test
   * @deprecated methode non compatible
   */
  @Deprecated
  public void maMethode() {
    System.out.println("test");
  }
}
```

Les entités marquées avec l'annotation @Deprecated devraient être documentées avec le tag @deprecated de Javadoc en lui fournissant la raison de l'obsolescence et éventuellement l'entité de substitution.

Il est important de tenir compte de la casse : @Deprecated pour l'annotation et @deprecated pour Javadoc.

Lors de la compilation, le compilateur donne une information si une entité obsolète est utilisée.

Exemple :

```
C:\Documents and Settings\jmd\workspace\Tests>javac TestDeprecated.java
Note: TestDeprecated.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

L'option -Xlint :deprecation permet d'afficher le détail sur les utilisations obsolètes.

Exemple :

```
C:\Documents and Settings\jmd\workspace\Tests>javac -Xlint:deprecation TestDepre
cated.java
TestDeprecated.java:7: warning: [deprecation] MaSousClasse in unnamed package ha
s been deprecated
    MaSousClasse td = new MaSousClasse();
    ^
TestDeprecated.java:7: warning: [deprecation] MaSousClasse in unnamed package ha
s been deprecated
    MaSousClasse td = new MaSousClasse();
                          ^
TestDeprecated.java:8: warning: [deprecation] maMethode() in MaSousClasse has be
en deprecated
```

```
    td.maMethode();
      ^
3 warnings
```

Il est aussi possible d'utiliser l'option -deprecation de l'outil javac.

11.4.2. L'annotation @Override

Cette annotation est un marqueur utilisé par le compilateur pour vérifier la réécriture de méthodes héritées.

@Override s'utilise pour annoter une méthode qui est une réécriture d'une méthode héritée. Le compilateur lève une erreur si aucune méthode héritée ne correspond.

Exemple :

```
@Override
public void maMethode() {
}
```

Son utilisation n'est pas obligatoire mais recommandée car elle permet de détecter certains problèmes.

Exemple :

```
public class MaClasseMere {
}

class MaClasse extends MaClasseMere {

  @Override
  public void maMethode() {
  }
}
```

Ceci est particulièrement utile pour éviter des erreurs de saisie dans le nom des méthodes à redéfinir.

Exemple :

```
public class TestOverride {

  private String nom;
  private long id;

  public int hasCode() {
    final int PRIME = 31;
    int result = 1;
    result = PRIME * result + (int) (id ^ (id >>> 32));
    result = PRIME * result + ((nom == null) ? 0 : nom.hashCode());
    return result;
  }
}
```

Dans l'exemple ci-dessous, le développeur souhaitait redéfinir la méthode hashCode() mais suite à une faute frappe a simplement défini une nouvelle méthode nommée hasCode(). Cette classe se compile parfaitement mais elle comporte une erreur qui est signalée en utilisant l'annotation @Override.

Exemple :

```
public class TestOverride {

  private String nom;
  private long id;

  @Override
  public int hasCode() {
    final int PRIME = 31;
    int result = 1;
    result = PRIME * result + (int) (id ^ (id >>> 32));
    result = PRIME * result + ((nom == null) ? 0 : nom.hashCode());
    return result;
  }
}
```

Résultat :

```
C:\Documents and Settings\jmd\workspace\Tests>javac TestOverride.java
TestOverride.java:6: method does not override or implement a method from a super
type
  @Override
  ^
1 error
```

11.4.3. L'annotation @SuppressWarning

Les compilateurs peuvent détecter des cas qui sont potentiellement suspicieux et qui devraient nécessiter d'être regardés attentivement par les développeurs mais n'empêchent pas la compilation. Les compilateurs les reportent sous la forme d'avertissements (warning).

Les warnings indiqués par un compilateur permettent de préciser un risque potentiel dans le code : cela ne représente qu'un risque et pas une erreur. Cependant, il est risqué d'ignorer un warning sans s'interroger au préalable sur sa pertinence. Il est même préférable de corriger l'origine du warning si c'est possible plutôt que de l'ignorer.

Il arrive parfois que ces avertissements, qui sont des risques potentiels, ne soient pas fondés après vérification, mais soient des faux positifs ou qu'il n'y ait pas de moyen de les résoudre sans avoir un impact non souhaité sur les fonctionnalités. Si vous êtes sûr que le warning peut être ignoré sans risque alors il est possible d'utiliser l'annotation @SuppressWarnings pour demander au compilateur de l'ignorer.

Elle permet de demander au compilateur d'ignorer un ou plusieurs avertissements pouvant être émis lors de la compilation de l'élément annoté ou un de ses éléments enfants. L'ensemble des avertissements d'un élément se cumule avec ceux définis sur un élément enfant pour ce dernier. Exemple : si un avertissement est ignoré sur une classe et un autre sur une méthode, les deux avertissements seront ignorés pour la méthode.

L'annotation @SuppressWarnings est une annotation standard fournie dans le JDK depuis Java 5.

11.4.3.1. L'utilisation de l'annotation @SuppressWarnings

L'annotation @SuppressWarnings ne peut être utilisée que sur une déclaration. Elle peut donc s'utiliser sur différents éléments :

- Sur une classe : dans ce cas l'annotation s'applique sur l'ensemble des éléments de la classe

Exemple (code Java 5.0) :

```
@SuppressWarnings("unchecked")
public class MaClasse{
  // ...
}
```

- Sur une méthode : dans ce cas l'annotation s'applique uniquement sur la méthode

Exemple (code Java 5.0) :

```
public class MaClasse {

  @SuppressWarnings("unchecked")
  public void maMathode() {
    // ...
  }
}
```

- Sur un champ : dans ce cas l'annotation s'applique sur ce champ

Exemple (code Java 5.0) :

```
public class MaClasse {

  @SuppressWarnings({ "rawtypes", "unchecked" })
  private List liste = (List<String>) new ArrayList();
}
```

- Sur une variable locale ou un paramètre

Attention : ignorer un avertissement peut cacher un bug potentiel. Il est donc préférable de tenter de le corriger plutôt que de l'ignorer. L'annotation @SuppressWarnings ne devrait être utilisée que lorsque cela est nécessaire et il faut toujours avoir une bonne connaissance des conséquences que son utilisation pourrait engendrer.

11.4.3.2. Les attributs de l'annotation @SuppressWarnings

L'attribut de l'annotation @SuppressWarnings permet d'indiquer un ou plusieurs avertissements qui doivent être ignorés par le compilateur pour l'élément annoté. Il est possible de mettre des doublons mais dans ce cas seul le premier sera pris en compte.

L'annotation @SuppressWarnings ne possède qu'un seul attribut qui est un tableau de String.

Il est possible de préciser une seule valeur :

Exemple (code Java 5.0) :

```
@SuppressWarnings("unchecked")
```

Il est aussi possible de préciser plusieurs valeurs en les séparant avec une virgule :

Exemple (code Java 5.0) :

```
@SuppressWarnings({"unchecked", "deprecation"})
```

L'annotation est standard, par contre les valeurs à lui passer en paramètre sont dépendantes de chaque compilateur. Ainsi les warnings précisés comme attribut de l'annotation sont spécifiques à chaque compilateur car chacun n'a pas les mêmes capacités de détections des avertissements.

Seules trois valeurs sont définies dans la JLS (Java Langage Specification) :

- unchecked
- deprecation
- removal (à partir de Java 9)

Toutes les autres valeurs sont spécifiques à chaque compilateur ou IDE utilisé.

Les avertissements inconnus du compilateur sont ignorés mais il peut émettre un avertissement relatif au fait qu'il ne connait pas l'avertissement indiqué.

11.4.3.3. Les options du compilateur pour afficher les warnings

Par défaut, le compilateur javac n'affiche pas les warnings : certains d'entre eux affichent une note pour indiquer leur présence.

Résultat :

```
C:\java>javac MaClasse.java
Note: MaClasse.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

L'option -Xlint permet de gérer les warnings qui doivent être affichés par le compilateur.

Comme indiqué, pour avoir le détail, il faut relancer javac avec l'option -Xlint suivi du caractère deux points et d'un type de warning. Les valeurs possibles pour le type de warning dépendent pour la plupart du compilateur utilisé.

Résultat :

```
C:\java>javac -Xlint:unchecked MaClasse.java
```

Il est possible de préciser plusieurs types en les séparant par une virgule.

Résultat :

```
C:\java>javac -Xlint:rawtype,unchecked MaClasse.java
```

Il est aussi possible de préfixer un type de warning avec un caractère moins pour le désactiver.

Résultat :

```
C:\java>javac -Xlint:all,-serial MaClasse.java
```

Enfin, il est possible d'activer tous les warnings en utilisant simplement l'option -Xlint ou -Xlint:all

Résultat :

```
C:\java>javac -Xlint:all MaClasse.java
C:\java>javac -Xlint MaClasse.java
```

Pour obtenir la liste des attributs utilisable dans @SuppressWarnings par un compilateur javac du JDK utilisé, il suffit d'utiliser son option -X

Résultat :

```
C:\java>javac -X
```

Parmi les options affichées, il y a celles relatives aux avertissements précisés dans l'option -Xlint.

A partir de Java 17, il faut utiliser l'option --help-lint de javac pour obtenir la liste des avertissements supporté par -Xlint.

Résultat :

```
C:\java>javac --help-lint
```

Pour désactiver tous les avertissements du compilateur, il est possible d'utiliser l'option -Xlint:none ou l'option -nowarn.

Il est aussi possible de désactiver un avertissement particulier en le préfixant avec un moins dans l'option -Xlint. Exemple : -Xlint:serial

Jusqu'à la version 8 des spécifications du langage Java, seules les avertissements unchecked et deprecation devraient être implémentées par tous les compilateurs.

D'autres IDE, compilateurs ou des outils d'analyse de code peuvent supporter d'autres valeurs possibles pour @SuppressWarnings : ces valeurs sont alors spécifiques à ces outils.

11.4.3.4. Les options du compilateur d'OpenJDK pour afficher les warnings

Le compilateur javac d'OpenJDK propose plusieurs avertissements :

Valeur	Rôle	Version de Java
all	Activer tous les avertissements	5
auxiliaryclass	Avertissement lors de l'utilisation d'une classe auxiliaire utilisée dans une classe définie dans un autre fichier	8
cast	Avertissement lors de l'utilisation de cast inutile	6
classfile		7
deprecation	Avertissement lors de l'utilisation d'éléments deprecated. Similaire à l'option historique -deprecation du compilateur	5
dep-ann	Avertissement concernant des éléments marqués comme deprecated dans la Javadoc sans qu'elle soit annotée avec @Deprecated	7
divzero	Avertissement concernant une division avec la constante entière 0	6
empty	Avertissement concernant une instruction if sans traitement	6
exports		9
fallthrough	Avertissement lors de l'absence possible d'une instruction break dans une clause case d'un switch	5
finally	Avertissement concernant l'utilisation d'une instruction return dans un bloc finally	5
lossy-conversions	Avertissement concernant l'utilisation d'assignations composées avec pertes possibles lors de conversions	20
missing-explicit-ctor	Avertissement sur les constructeurs explicites manquants dans les classes publiques et protégées dans les paquets exportés	16
module	Avertissement concernant les noms de module qui ne devraient pas se terminer par des chiffres	9
opens		9
options	Avertissement concernant l'utilisation de certaines combinaisons d'options du compilateur	7
overloads	Avertissement concernant la définition de surcharges d'une méthode qui pourraient engendrer des ambiguï¿½tés pour déterminer celle à invoquer	9
overrides	Avertissement concernant la redéfinition d'une méthode avec un tableau en un varargs ou vice versa	6
path	Avertissement concernant un élément du classpath inexistant	5
preview	Avertissement concernant l'utilisation de fonctionnalités en mode preview	11

processing	Avertissement concernant les annotations qui ne sont pas traitées par un processeur d'annotations	7
rawtypes	Avertissement concernant l'utilisation d'une classe générique sans préciser le type générique	7
removal	Avertissement concernant l'utilisation d'un élément de l'API est marqué comme étant dépréciée avec l'attribut forRemoval=true	9
requires-automatic	Avertissement concernant l'utilisation d'un automatic module avec une clause requires	9
requires-transitive-automatic	Avertissement concernant l'utilisation d'un automatic module avec une clause requires transitive	9
serial	Avertissement concernant l'absence d'un champ serialVersionUID dans une classe Serializable	5
static	Avertissement concernant l'accès à un membre static à partir d'une instance	7
strictfp	Avertissement sur l'utilisation inutile du modificateur strictfp	17
synchronization		16
text-blocks	Avertissement sur les caractères d'espacement incohérents dans l'indentation des blocs de texte	13
try	Avertissement concernant une utilisation d'une instruction try-with-resources	7
unchecked	Avertissement concernant le fait que le compilateur ne peut pas garantir la vérification de type (type safety)	5
varargs	Avertissement concernant une possible utilisation d'un varargs en paramètre d'une méthode	7
none	Désactiver tous les avertissements	6

11.4.3.4.1. auxiliaryclass

Le compilateur émet un avertissement lorsqu'une classe auxiliaire est utilisée dans une classe qui n'est pas définie dans le même fichier source. Une classe auxiliaire est une classe définie dans un fichier dont le nom n'est pas celui de la classe elle-même.

Exemple :

```
public class MaClasse {
}

class MaClasseAuxiliaire {
}
```

Exemple :

```
public class MonAutreClasse {

  MaClasseAuxiliaire mac = null;
}
```

C'est légal en Java mais il n'est pas recommandé d'utiliser une classe auxiliaire en dehors du fichier dans lequel elle est définie.

Résultat :

```
C:\java>javac -Xlint MaClasse.java MonAutreClasse.java
MonAutreClasse.java:3: warning: auxiliary class MaClasseAuxiliaire in
MaClasse.java should not be accessed from outside its own source file
  MaClasseAuxiliaire mac = null;
  ^
1 warning
```

Une solution pour éviter cette situation est de définir la classe non pas comme une classe auxiliaire mais comme une classe imbriquée.

Exemple :

```
public class MaClasse {

  static class MaClasseAuxiliaire {
  }
}
```

Exemple :

```
public class MonAutreClasse {

  MaClasse.MaClasseAuxiliaire mca = null;
}
```

Résultat :

```
C:\java>javac -Xlint MaClasse.java MonAutreClasse.java
C:\java>
```

11.4.3.4.2. cast

L'utilisation inutile d'un cast provoque un avertissement de la part du compilateur.

Exemple :

```
public class MaClasse {

  public static void main(String... args) {
    String message = (String) "texte";
  }
}
```

Résultat :

```
C:\java>javac -Xlint:all MaClasse.java
MaClasse.java:4: warning: [cast] redundant cast to String
    String message = (String) "texte";
                     ^
1 warning
```

Il est possible de désactiver l'avertissement du compilateur relatif à l'utilisation inutile d'un cast.

Exemple :

```
public class MaClasse {

  @SuppressWarnings("cast")
  public static void main(String... args) {
    String message = (String) "texte";
  }
}
```

Résultat :

```
C:\java>javac -Xlint:cast MaClasse.java
```

```
C:\java>
```

Plutôt que d'ignorer cet avertissement, il est de préférable de le corriger simplement en supprimant le cast concerné.

11.4.3.4.3. deprecation

L'utilisation d'un élément deprecated provoque un avertissement de la part du compilateur.

Exemple :

```
import java.util.Date;

public class MaClasse {

  public static void main(String... args) {
    Date date =  new Date(10,10,10);
  }
}
```

Résultat :

```
C:\java> javac MaClasse.java
Note: MaClasse.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

C:\java> javac -Xlint:deprecation MaClasse.java
MaClasse.java:6: warning: [deprecation] Date(int,int,int) in Date has been deprecated
          Date date =  new Date(10,10,10);
                       ^
1 warning
```

Il est possible de désactiver l'avertissement du compilateur relatif à l'utilisation d'un élément deprecated.

Exemple :

```
import java.util.Date;

public class MaClasse {

  public static void main(String... args) {
    @SuppressWarnings("deprecation")
    Date date =  new Date(10,10,10);
  }
}
```

Résultat :

```
C:\java> javac MaClasse.java

C:\java>
```

Jusqu'à Java 8, pour demander au compilateur d'OpenJDK d'ignorer les avertissements relatifs à l'utilisation d'élément deprecated, il fallait utiliser l'annotation @SuppressWarnings("deprecation").

A partir de Java 9, la valeur « deprecation » ne concerne que les éléments deprecated forRemoval=false. Les éléments deprecated forRemoval=true génère un avertissement de la part du compilateur. Ceci permet de maintenir l'avertissement pour ces éléments et ainsi éviter de ne pas être informé de leur futur suppression. C'est nécessaire car de nombreux développeurs supposent que les éléments deprecated ne seront jamais retirés pour des raisons de compatibilité ascendante.

Pour demander au compilateur d'OpenJDK d'ignorer les avertissements pour ces éléments, il faut utiliser l'annotation @SuppressWarnings("removal"). Pour demander d'ignorer l'utilisation de tous les types d'éléments deprecated, il faut utiliser l'annotation @SuppressWarnings({"deprecation", "removal"}).

11.4.3.4.4. dep-ann

La définition d'une méthode dont la Javadoc précise qu'elle est deprecated sans qu'elle soit annotée avec @Deprecated provoque un avertissement de la part du compilateur.

Exemple :

```
public class MaClasse {

  public static void main(String... args) {
    methodeDeprecated();
  }

  /**
   * @deprecated
   */
  public static void methodeDeprecated() {
  }
}
```

Résultat :

```
C:\java> javac -Xlint:all MaClasse.java
MaClasse.java:12: warning: [dep-ann] deprecated item is not annotated with
@Deprecated
  public static void methodeDeprecated() {
                     ^
1 warning
```

Il est possible de désactiver cet avertissement.

Exemple :

```
public class MaClasse {

  public static void main(String... args) {
    methodeDeprecated();
  }

  /**
   * @deprecated
   */
  @SuppressWarnings("dep-ann")
  public static void methodeDeprecated() {
  }
}
```

Résultat :

```
C:\java> javac -Xlint:dep-ann MaClasse.java

C:\java>
```

11.4.3.4.5. divzero

Une division avec la valeur littérale zéro provoque un avertissement de la part du compilateur.

Exemple :

```
public class MaClasse {

  public static void main(String... args) {
    long valeur = 123 / 0;
  }
}
```

Résultat :

```
C:\java>javac -Xlint:divzero MaClasse.java
MaClasse.java:6: warning: [divzero] division by zero
    long valeur = 123 / 0;
                        ^
1 warning
```

Il est possible de désactiver l'avertissement du compilateur relatif à division par zéro.

Exemple :

```
public class MaClasse {

  public static void main(String... args) {
    @SuppressWarnings("divzero")
    long valeur = 123 / 0;
  }
}
```

Résultat :

```
C:\java> javac -Xlint:divzero MaClasse.java

C:\java>
```

Attention : cet avertissement ne fonctionne que lors de l'utilisation de la valeur littérale zéro ou d'une constante dont la valeur est zéro.

Exemple :

```
public class MaClasse {

  private static final int ZERO = 0;

  public static void main(String... args) {
    int zero = 0;
    long valeur = 123 / zero;

    valeur = 123 / ZERO;
  }
}
```

Résultat :

```
C:\java>javac -Xlint:divzero MaClasse.java
MaClasse.java:11: warning: [divzero] division by zero
        valeur = 123 / ZERO;
                       ^
1 warning
```

Ignorer cet avertissement ne va pas empêcher la levée d'une exception de type java.lang.ArithmeticException à l'exécution du code.

Résultat :

```
C:\java>java MaClasse
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at MaClasse.main(MaClasse.java:9)
```

11.4.3.4.6. empty

L'utilisation d'un bloc de code vide dans une instruction if provoque un avertissement de la part du compilateur.

Exemple :

```
public class MaClasse {

  public static void main(String... args) {
    if (args.length > 0);
    System.out.println("Presence de parametres");
  }
}
```

Résultat :

```
C:\java>javac -Xlint:empty MaClasse.java
MaClasse.java:5: warning: [empty] empty statement after if
    if (args.length > 0);
                        ^
1 warning
```

Il est possible de désactiver l'avertissement du compilateur relatif à l'utilisation d'un bloc de code vide dans une instruction if.

Exemple :

```
public class MaClasse {

  @SuppressWarnings("empty")
  public static void main(String... args) {
    if (args.length > 0);
    System.out.println("Presence de parametres");
  }
}
```

Résultat :

```
C:\java>javac -Xlint:empty MaClasse.java

C:\java>
```

Attention : cet avertissement ne concerne que les blocs vides pour les instructions if sans clause else.

Exemple :

```
public class MaClasse {

  public static void main(String... args) {
    if (args.length > 0);
    else;

    for(String arg : args);

    while (args.length == 0);
    do;

    while (args.length == 0);
  }
}
```

Résultat :

```
C:\java>javac -Xlint:all MaClasse.java

C:\java>
```

Dans l'exemple ci-dessus, aucun avertissement n'est affiché par le compilateur.

11.4.3.4.7. fallthrough

Par défaut sans instruction break à la fin d'une instruction case, le code d'une instruction case exécute le code des instructions case suivantes jusqu'à la rencontre d'une instruction break.

Parfois bien pratique notamment si le même code concerne plusieurs valeurs, cela peut aussi parfois amener à des bugs subtils lors de l'oubli d'instructions breaks. C'est la raison pour laquelle si une instruction case possède au moins une instruction et pas d'instruction break alors le compilateur génère un avertissement.

Exemple :

```
public class MaClasse {

  public static void main(String... args) {
    String valeur = "A";
    switch(valeur) {
    case "A":
      System.out.println("A");
    case "B":
      System.out.println("B");
    }
  }
}
```

Résultat :

```
C:\java> javac -Xlint:all MaClasse.java
MaClasse.java:9: warning: [fallthrough] possible fall-through into case
    case "B":

^
1 warning
```

Il est possible de désactiver cet avertissement du compilateur.

Exemple :

```
public class MaClasse {

  @SuppressWarnings("fallthrough")
  public static void main(String... args) {
    String valeur = "A";
    switch(valeur) {
    case "A":
      System.out.println("A");
    case "B":
      System.out.println("B");
    }
  }
}
```

Résultat :

```
C:\java>javac -Xlint:fallthrough MaClasse.java

C:\java>
```

11.4.3.4.8. finally

L'utilisation d'une instruction return dans un bloc finally n'est pas une bonne pratique et sa bonne exécution n'est pas garantie : elle provoque donc un avertissement de la part du compilateur.

Exemple :

```
public class MaClasse {

  public static int convertir(String valeur) {
    int resultat = -1;
    try {
      resultat = Integer.parseInt(valeur);
    } catch (ArithmeticException ae) {
    } finally {
      return resultat;
    }
  }
}
```

Résultat :

```
C:\java>javac -Xlint:finally MaClasse.java
MaClasse.java:18: warning: [finally] finally clause cannot complete normally
    }

^
1 warning
```

Il est possible de désactiver l'avertissement du compilateur relatif à l'utilisation d'une instruction return dans un bloc finally.

Exemple :

```
public class MaClasse {

  @SuppressWarnings("finally")
  public static int convertir(String valeur) {
    int resultat = -1;
    try {
      resultat = Integer.parseInt(valeur);
    } catch (ArithmeticException ae) {
    } finally {
      return resultat;
    }
  }
}
```

Résultat :

```
C:\java> javac -Xlint:finally MaClasse.java

C:\java>
```

D'autres situations utilisant un return peuvent générer cet avertissement.

Exemple :

```
public class MaClasse {

  public static void main(String... args) {
    System.out.println(saluer("Pierre"));
  }

  public static String saluer(String nom) {
    String message = "Bonjour ";

    try {
      message+=nom;
    } finally {
      return message.toUpperCase();
    }
  }
}
```

Exemple :

```
public class MaClasse {

  public static void main(String... args) {
    System.out.println(convertir("ABC"));
  }

  public static int convertir(String valeur) {
    int resultat = 0;
    try {
      resultat = Integer.parseInt(valeur);
    } catch (ArithmeticException ae) {
      return -1;
    } finally {
      return resultat;
    }
  }
}
```

Tous les exemples ci-dessus sont syntaxiquement valides en Java mais ne sont pas de bonnes pratiques : il est donc préférable de ne pas ignorer cet avertissement.

11.4.3.4.9. lossy-conversions

Lors de la manipulation de valeurs primitives de différents types, il est parfois nécessaire d'adapter une valeur plus grande que celle du type utilisé dans la déclaration de la variable. Cela peut entraîner une perte d'informations puisque certains octets devront être supprimés. Dans ce cas, il faut utiliser un cast pour indiquer que nous sommes conscients de la situation et que nous la validons ou changer le type de la variable cible.

Exemple :

```
package fr.jmdoudoux.dej;

public class MaClasse {

  public static void main(String[] args) {
    byte a = 60;
    int b = 120;
    a = a + b;
    System.out.println(a);
  }
}
```

Résultat :

```
C:\java>javac src\fr\jmdoudoux\dej\MaClasse.java
src\fr\jmdoudoux\dej\MaClasse.java:8: error: incompatible types: possible lossy
 conversion from
 int to byte
    a = a + b;
          ^
1 error
```

Le compilateur émet une erreur, car le résultat de l'addition donne une valeur de type int (4 octets) qui est affectée avec une variable de type byte (1 octet). Il y a donc un risque de perte de données.

Pour corriger l'erreur, il faut modifier le type de la variable a en int ou ajouter un cast explicite vers le type short. L'ajout d'un cast permet de compiler le code, mais le résultat obtenu peut être erroné à cause de la perte de données liée à l'affectation vers un type dont la plage des valeurs possibles est inférieure.

Exemple :

```
package fr.jmdoudoux.dej;

public class MaClasse {

  public static void main(String[] args) {
    byte a = 60;
    int b = 120;
    a = (byte) (a + b);
    System.out.println(a);
  }
}
```

Résultat :

```
C:\java>javac src\fr\jmdoudoux\dej\MaClasse.java

C:\java>java -cp src fr.jmdoudoux.dej.MaClasse
-76
```

Le comportement du compilateur est différent lors de l'utilisation d'un opérateur d'affectation composé.

Exemple :

```
package fr.jmdoudoux.dej;

public class MaClasse {

  public static void main(String[] args) {
    byte a = 60;
    int b = 120;
    a += b;
    System.out.println(a);
  }
}
```

Résultat :

```
C:\java>javac src\fr\jmdoudoux\dej\MaClasse.java

C:\java>java -cp src fr.jmdoudoux.dej.MaClasse
-76
```

Avec l'utilisation d'un opérateur d'affection composé, le compilateur ajoute un cast si le type de l'opérande de droite d'une affectation composée n'est pas compatible avec le type de la variable, une conversion de type est implicitement effectuée avec une perte potentielle qui peut survenir.

Résultat :

```
C:\java>javap -c -cp src fr.jmdoudoux.dej.MaClasse
Compiled from "MaClasse.java"
public class fr.jmdoudoux.dej.MaClasse {
  public fr.jmdoudoux.dej.MaClasse();
    Code:
       0: aload_0
       1: invokespecial #1           // Method java/lang/Object."<init>":()V
       4: return
  public static void main(java.lang.String[]);
    Code:
       0: bipush        60
       2: istore_1
       3: bipush        120
       5: istore_2
       6: iload_1
       7: iload_2
       8: iadd
       9: i2b
      10: istore_1
      11: getstatic     #7           // Field java/lang/System.out:Ljava/io/PrintStream;
      14: iload_1
      15: invokevirtual #13          // Method java/io/PrintStream.println:(I)V
      18: getstatic     #7           // Field java/lang/System.out:Ljava/io/PrintStream;
      21: iload_1
      22: invokevirtual #13          // Method java/io/PrintStream.println:(I)V
      25: return
}
```

Dans l'exemple ci-dessus, le compilateur a ajouté dans le bytecode une instruction i2b qui réalise un cast vers le type byte. Ainsi le code se compile mais le résultat est potentiellement faux, comme c'est le cas dans l'exemple ci-dessus.

Comme cela peut engendrer des bugs silencieux, dans le JDK 20, un nouveau lint "lossy-conversions" a été ajouté au compilateur javac pour avertir lors de l'utilisation d'assignations composées avec pertes possibles lors des conversions.

L'avertissement n'est pas activé par défaut par le compilateur javac : il faut demander son activation au compilateur en utilisant l'option -Xlint, -Xlint:all, ou -Xlint:lossy-conversions.

Résultat :

```
C:\java>javac -Xlint src\fr\jmdoudoux\dej\MaClasse.java
src\fr\jmdoudoux\dej\MaClasse.java:8: warning: [lossy-conversions] implicit cast from
 int to byte in compound assignment is possibly lossy
    a += b;
         ^
1 warning
```

11.4.3.4.10. missing-explicit-ctor

Le compilateur javac génère un constructeur par défaut si une classe ne déclare explicitement aucun constructeur. Bien que cette politique puisse être pratique, cela peut s'apparenter à une mauvaise pratique de programmation, ne serait-ce que parce que le constructeur par défaut ne sera pas inclus dans la documentation javadoc.

A partir du JDK 16, le compilateur javac émet un avertissement lors de la compilation d'une classe qui ne possède pas de constructeur explicite et qui impliquera l'ajout d'un constructeur par défaut.

Pour que l'avertissement soit affiché, il faut utiliser l'option -Xlint

L'avertissement n'est cependant pas émis pour toutes les classes pour lesquelles un constructeur par défaut est ajouté par le compilateur. Deux conditions doivent être respectées pour que le compilateur émette l'avertissement :

- la classe doit être publique
- être dans un package exposé d'un module

Dans l'exemple ci-dessous, un module expose un package qui contient une classe sans constructeur explicite, de sorte qu'un constructeur par défaut soit ajouté dans le byte-code par le compilateur.

Résultat :

```
C:\java>tree /f /a src
C:\JAVA\SRC
|   module-info.class
|   module-info.java
|
\---fr
    \---jmdoudoux
        \---dej
               MaClasse.class
               MaClasse.java
```

Exemple (code Java 9) :

```
module fr.jmdoudoux.dej {
  exports fr.jmdoudoux.dej;
}
```

Exemple (code Java 9) :

```
package fr.jmdoudoux.dej;

public class MaClasse {

  public static void main(String[] args) {
  }
}
```

Résultat :

```
C:\java>javac --source-path=src src\module-info.java

C:\java>javac --source-path=src src\fr\jmdoudoux\dej\MaClasse.java

C:\java>
```

L'avertissement n'est pas affiché par défaut par le compilateur javac : il faut demander son affichage en utilisant l'option -Xlint, -Xlint:all, ou -Xlint:missing-explicit-ctor.

Résultat :

```
C:\java>javac -Xlint --source-path=src src\fr\jmdoudoux\dej\MaClasse.java
src\fr\jmdoudoux\dej\MaClasse.java:3: warning: [missing-explicit-ctor] class MaClasse
 in exported package fr.jmdoudoux.dej declares no explicit constructors, thereby
 exposing a default constructor to clients of module fr.jmdoudoux.dej
public class MaClasse {
       ^
1 warning
```

Le message d'avertissement émit par javac décrit le problème et indique précisément le package exporté avec la classe incriminée et le nom du module qui exporte ce package.

Si la classe n'est pas publique, alors l'avertissement n'est pas émis.

Exemple (code Java 9) :

```
package fr.jmdoudoux.dej;

class MaClasse {

  public static void main(String[] args) {
  }
}
```

Résultat :

```
C:\java>javac -Xlint --source-path=src src\fr\jmdoudoux\dej\MaClasse.java

C:\java>
```

Si le package n'est pas exposé alors l'avertissement n'est pas émis par le compilateur, même si la classe est publique.

Exemple (code Java 9) :

```
module fr.jmdoudoux.dej {
}
```

Exemple (code Java 9) :

```
package fr.jmdoudoux.dej;

public class MaClasse {

  public static void main(String[] args) {
  }
}
```

Résultat :

```
C:\java>javac --source-path=src src\module-info.java

C:\java>javac -Xlint --source-path=src src\fr\jmdoudoux\dej\MaClasse.java

C:\java>
```

La solution pour éviter cet avertissement est d'ajouter explicitement le constructeur par défaut dans les classes de packages exportés d'un module qui ne contiennent pas de constructeurs.

Exemple (code Java 9) :

```
package fr.jmdoudoux.dej;

public class MaClasse {

   public MaClasse() {
   }

  public static void main(String[] args) {
  }
}
```

Résultat :

```
C:\java>javac -Xlint --source-path=src src\fr\jmdoudoux\dej\MaClasse.java

C:\java>
```

11.4.3.4.11. module

Le compilateur émet un avertissement lorsqu'un des éléments qui compose le nom d'un module se termine par un nombre car cela devrait être évité.

Exemple (code Java 9) :

```
module fr.jmdoudoux.dej2.module123 {
}
```

Résultat :

```
C:\java>javac module-info.java
module-info.java:1: warning: [module] module name component module123
should avoid terminal digits
module fr.jmdoudoux.dej2.module123 {
                         ^
module-info.java:1: warning: [module] module name component dej2 should
```

```
avoid terminal digits
module fr.jmdoudoux.dej2.module123 {
                    ^
2 warnings
```

11.4.3.4.12. options

Le compilateur émet un avertissement lorsque certaines combinaisons d'options sont utilisées lors de l'invocation du compilateur.

Résultat :

```
C:\java>javac -source 1.7 MaClasse.java
warning: [options] bootstrap class path not set in conjunction with -source 1.7
```

11.4.3.4.13. overloads

Le compilateur émet un avertissement lorsque des surcharges d'une méthode peuvent préter à confusion.

Exemple (code Java 8) :

```
import java.util.function.Consumer;

public class MaClasse {

  static void traiter(int  i, Consumer<Integer> action) { }

  static void traiter(long l, Consumer<Long>    action) { }
}
```

Résultat :

```
C:\java>javac -Xlint:all MaClasse.java
MaClasse.java:5: warning: [overloads]
traiter(int,Consumer<Integer>) in MaClasse is potentially ambiguous with
traiter(long,Consumer<Long>) in MaClasse
    static void traiter(int  i, Consumer<Integer> action) { }
                ^
1 warning
```

Dans cet exemple, le compilateur nous avertit sur un potentiel problème qui pourrait empêcher la détermination claire de la surcharge à invoquer.

Exemple (code Java 8) :

```
import java.util.function.Consumer;

public class MaClasse {

  static void traiter(int  i, Consumer<Integer> action) { }

  static void traiter(long l, Consumer<Long>    action) { }

  public static void main(String[] args) {
    traiter(1, (i) -> { });
  }
}
```

Résultat :

```
C:\java>javac -Xlint:all MaClasse.java
MaClasse.java:5: warning: [overloads] traiter(int,Consumer<Integer>)
in MaClasse is potentially ambiguous with traiter(long,Consumer<Long>) in MaClasse
    static void traiter(int  i, Consumer<Integer> action) { }
                ^
```

```
MaClasse.java:9: error: reference to traiter is ambiguous
      traiter(1, (i) -> { });
      ^
  both method
traiter(int,Consumer<Integer>) in MaClasse and method
traiter(long,Consumer<Long>) in MaClasse match
1 error
1 warning
```

Pour résoudre cette erreur, il faut indiquer le type explicitement pour permettre de déterminer de manière univoque la surcharge à invoquer. Dans cet exemple, il est par exemple possible de préciser explicitement le type générique du paramètre de type Consumer.

Exemple (code Java 8) :

```
import java.util.function.Consumer;

public class MaClasse {

  static void traiter(int  i, Consumer<Integer> action) { }

  static void traiter(long l, Consumer<Long>    action) { }

  public static void main(String[] args) {
    traiter(1, (Long i) -> { });
  }
}
```

Résultat :

```
C:\java>javac -Xlint:all MaClasse.java
MaClasse.java:5: warning: [overloads] traiter(int,Consumer<Integer>)
in MaClasse is potentially ambiguous with traiter(long,Consumer<Long>) in MaClasse
    static void traiter(int  i, Consumer<Integer> action) { }
                ^
1 warning
```

Il est possible d'utiliser l'annotation @SuppressWarning("overloads") pour demander au compilateur d'ignorer l'avertissement sur l'élément marqué.

Exemple (code Java 8) :

```
import java.util.function.Consumer;

public class MaClasse {

  @SuppressWarnings("overloads")
  static void traiter(int  i, Consumer<Integer> action) { }

  @SuppressWarnings("overloads")
  static void traiter(long l, Consumer<Long>    action) { }

  public static void main(String[] args) {
    traiter(1, (Long i) -> { });
  }
}
```

Résultat :

```
C:\java>javac -Xlint:all MaClasse.java
C:\java>
```

11.4.3.4.14. overrides

Le compilateur transforme un varargs un tableau dans le byte code.

L'avertissement overrides émis par le compilateur lors de la redéfinition d'une méthode dont le dernier paramètre est un tableau dans la classe mère et un varargs dans la classe fille.

Exemple :

```
import java.util.ArrayList;
import java.util.List;

public class MaClasse {

  protected List<String> elements = new ArrayList<>();

  public void ajouter(final String[] elmts) {
    for (final String element : elmts) {
      elements.add(element);
    }
  }
}

class MaClasseFille extends MaClasse {

  @Override
  public void ajouter(final String... elmts) {
    for (final String element : elmts) {
      elements.add(element);
    }
  }
}
```

Résultat :

```
C:\java> javac -Xlint:overrides MaClasse.java
MaClasse.java:19: warning: ajouter(String...) in MaClasseFille overrides
ajouter(String[]) in MaClasse; overridden method has no '...'
  public void ajouter(final String... elmts) {
              ^
1 warning
```

Cet avertissement survient aussi lors de la redéfinition d'une méthode dont le dernier paramètre est un varargs dans la classe mère et un tableau dans la classe fille.

Exemple :

```
import java.util.ArrayList;
import java.util.List;

public class MaClasse {

  protected List<String> elements = new ArrayList<>();

  public void ajouter(final String... elmts) {
    for (final String element : elmts) {
      elements.add(element);
    }
  }
}

class MaClasseFille extends MaClasse {

  @Override
  public void ajouter(final String[] elmts) {
    for (final String element : elmts) {
      elements.add(element);
    }
  }
}
```

Il est possible de désactiver cet avertissement du compilateur.

Exemple :

```
class MaClasseFille extends MaClasse {

  @Override
  @SuppressWarnings("overrides")
  public void ajouter(final String... elmts) {
    for (final String element : elmts) {
      elements.add(element);
    }
  }
}
```

Résultat :

```
C:\java> javac -Xlint:overrides MaClasse.java

C:\java>
```

L'option -Xlint:overrides ne remplace pas l'annotation @Override : d'ailleurs la première est un avertissement la seconde est une erreur de compilation.

L'option -Xlint:overrides concerne des situations de redéfinition de méthodes utilisant un tableau est un varargs.

11.4.3.4.15. path

Les classes sont chargées à partir du classpath qui peut être composé de répertoires ou de fichiers .jar. Il est très facile de faire une erreur typographique lors de la saisie des éléments du classpath notamment lorsque ce dernier contient de nombreux éléments.

Par défaut, le compilateur ne fournit aucun avertissement relatif aux éléments du classpath qui n'existent pas.

L'option -Xlint:path permet de demander au compilateur d'afficher sous la forme de warnings les éléments du classpath qui ne sont pas trouvés. Les warnings émis ne concernent donc pas l'analyse du code source mais elle est particulièrement utile pour identifier certains problèmes de chargement de classes.

Résultat :

```
C:\java> javac -cp .;./repinexistant;./inexistant.jar -Xlint:path MaClasse.java
warning: [path] bad path element ".\repinexistant": no such file or directory
warning: [path] bad path element ".\inexistant.jar": no such file or directory
2 warnings
```

11.4.3.4.16. preview

Le compilateur émet un avertissement lors de l'utilisation de fonctionnalités proposées en mode preview.

Exemple (code Java 13) :

```
public class MaClasse {
  public static void main(String... args) {

    String message = """
                          Bonjour""";
 }
}
```

Résultat :

```
C:\java>javac --enable-preview -source 13 -Xlint:preview MaClasse.java
MaClasse.java:5: warning: [preview] text blocks are a preview feature and
may be removed in a future release.
    String message = """
                     ^
1 warning
```

Il n'est pas possible de demander d'ignorer cet avertissement ni avec l'annotation @SuppressWarnings ni avec le paramètre -Xlint. Le compilateur ne va pas tenir compte de cette demande.

Exemple (code Java 13) :

```
public class MaClasse {

  public static void main(String... args) {

    @SuppressWarnings("preview")
    String message = """
                          Bonjour""";
  }
}
```

Résultat :

```
C:\java>javac --enable-preview -source 13 MaClasse.java
Note: MaClasse.java uses preview language features.
Note: Recompile with -Xlint:preview
for details.
C:\java>javac --enable-preview -source 13 -Xlint:-preview MaClasse.java
Note: MaClasse.java uses preview language features.
Note: Recompile with -Xlint:preview
for details.
```

11.4.3.4.17. processing

Le compilateur peut émettre un avertissement pour les annotations qui ne sont pas traitées par un processeur d'annotations.

Par exemple, les deux annotations suivantes :

Exemple :

```
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Retention(RUNTIME)
@Target(TYPE)
public @interface MonAnnotation {
}
```

Exemple :

```
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Retention(RUNTIME)
@Target(TYPE)
public @interface MonAutreAnnotation {
}
```

Et une classe qui est processeur pour traiter l'annotation @MonAnnotation

Exemple :

```
import java.util.Set;
import javax.annotation.processing.AbstractProcessor;
import javax.annotation.processing.RoundEnvironment;
import javax.annotation.processing.SupportedAnnotationTypes;
import javax.lang.model.SourceVersion;
import javax.lang.model.element.Element;
import javax.lang.model.element.TypeElement;
import javax.tools.Diagnostic.Kind;

@SupportedAnnotationTypes("MonAnnotation")
public class MonProcessor extends AbstractProcessor {

  @Override
  public boolean process(Set<? extends TypeElement> elems, RoundEnvironment renv) {
    Set<? extends Element> elements = renv.getElementsAnnotatedWith(MonAnnotation.class);
    for (Element e : elements) {
      processingEnv.getMessager().printMessage(Kind.NOTE,
          "Utilisation de l'annotation @MonAnnotation sur la classe " + e.getSimpleName());
    }
    return true;
  }

  @Override
  public SourceVersion getSupportedSourceVersion() {
    return SourceVersion.RELEASE_8;
  }
}
```

Résultat :

```
C:\java>javac MonProcessor.java
```

Il est possible d'utiliser le processeur avec le compilateur pour compiler une classe qui utilise l'annotation @MonAnntotation.

Exemple :

```
import java.util.ArrayList;
import java.util.List;

@MonAnnotation
public class MaClasse {
}
```

Résultat :

```
C:\java>javac -cp . -processor MonProcessor -proc:only MaClasse.java
Note: Utilisation de l'annotation
@MonAnnotation sur la classe MaClasse
```

Le compilateur peut lever un avertissement pour les annotations qui ne sont pas traités par un processeur en utilisant l'option -Xlint:processing.

Exemple :

```
import java.util.ArrayList;
import java.util.List;

@MonAutreAnnotation
public class MaClasse {
}
```

Résultat :

```
C:\java>javac -cp . -Xlint:processing -processor MonProcessor -proc:only MaClasse.java
warning: No processor claimed any of these annotations: MonAutreAnnotation
1 warning
```

11.4.3.4.18. rawtypes

L'utilisation d'une classe typée avec generic sans préciser le type souhaité provoque un avertissement de la part du compilateur.

Exemple :

```
import java.util.List;
import java.util.ArrayList;

public class MaClasse {
  public static void main(String... args) {
    List liste = new ArrayList();
  }
}
```

Résultat :

```
C:\java>javac -Xlint:rawtypes MaClasse.java
MaClasse.java:8: warning: [rawtypes] found raw type: List
    List liste = new ArrayList();
    ^
  missing type arguments for generic class List<E>
  where E is a type-variable:
    E extends Object declared in interface List
MaClasse.java:8: warning: [rawtypes] found raw type: ArrayList
    List liste = new ArrayList();
                     ^
  missing type arguments for generic class ArrayList<E>
  where E is a type-variable:
    E extends Object declared in class ArrayList
2 warnings
```

Il est possible de désactiver l'avertissement du compilateur relatif à l'absence du type generic.

Exemple :

```
import java.util.List;
import java.util.ArrayList;

public class MaClasse {

  public static void main(String... args) {
    @SuppressWarnings("rawtypes")
    List liste = new ArrayList();
  }
}
```

Résultat :

```
C:\java>javac -Xlint:rawtypes MaClasse.java

C:\java>
```

A défaut de mieux, il est possible de préciser le type generic <?> qui permet d'indiquer n'importe quelle type. L'idéal est d'indiquer un type précis dans le generic.

Exemple :

```
import java.util.ArrayList;
import java.util.List;

public class MaClasse {

  public static void main(String... args) {
    List<?> liste = new ArrayList<>();
  }
}
```

Résultat :

```
C:\java>javac -Xlint:all MaClasse.java

C:\java>
```

11.4.3.4.19. serial

La définition d'une classe implémentant l'interface Serialisable sans définir un attribut serialVersionUID provoque un avertissement de la part du compilateur.

Exemple :

```
import java.io.Serializable;

public class MaClasse implements Serializable {
}
```

Résultat :

```
C:\java>javac -Xlint:serial MaClasse.java
MaClasse.java:3: warning: [serial] serializable class MaClasse has no
definition of serialVersionUID
public class MaClasse implements Serializable {
       ^
1 warning
```

Il est possible de désactiver l'avertissement du compilateur relatif à l'absence d'un attribut serialVersionUUID d'une classe Serializable.

Exemple :

```
import java.io.Serializable;

@SuppressWarnings("serial")
public class MaClasse implements Serializable {
}
```

Résultat :

```
C:\java> javac -Xlint:serial MaClasse.java

C:\java>
```

Il est fortement recommandé de définir explicitement un champ serialVersionUID plutôt que de la valeur générée. Cette génération dépend des implémentations et peut donc engendrer la levée d'exceptions de type InvalidClassException pendant les opérations de désérialisation. Ignorer les avertissements de type serial n'est donc pas recommandé.

11.4.3.4.20. static

L'invocation d'une méthode statique sur une instance provoque un avertissement de la part du compilateur.

Exemple :

```
public class MaClasse {

  public void methode() {
    this.methodeStatique();
  }

  public static void methodeStatique() {
  }
}
```

Résultat :

```
C:\java>javac -Xlint:all MaClasse.java
MaClasse.java:4: warning: [static] static method should be qualified by
type name, MaClasse, instead of by an expression
    this.methodeStatique();
        ^
1 warning
```

Il est possible de désactiver l'avertissement du compilateur relatif à l'invocation d'une mï¿½thode statique sur une instance.

Exemple :

```
public class MaClasse {

  @SuppressWarnings("static")
  public void methode() {
    this.methodeStatique();
  }

  public static void methodeStatique() {
  }
}
```

Résultat :

```
C:\java>javac -Xlint:static MaClasse.java

C:\java>
```

Plutôt que d'ignorer cet avertissement, il est de préférable de le corriger simplement en remplaçant l'instance par la classe.

11.4.3.4.21. strictfp

Depuis Java 1.2, le modificateur strictfp est utilisé pour garantir que les résultats des calculs en virgule flottante seront les mêmes quelle que soit la plate-forme sur laquelle la JVM s'exécute. Sans le modificateur strictfp, la précision des résultats peut varier d'une plate-forme à l'autre en raison de la capacité du matériel à les traiter.

A partir de Java 17 (JEP 306), les opérations en virgule flottante sont de nouveau systématiquement strictes, plutôt que d'avoir à la fois une sémantique de virgule flottante "stricte" (strictfp) et une sémantique de virgule flottante "par défaut" potenciellement différente selon le processeur. Cela rétablit la sémantique flottante originale du langage et de la VM, correspondant à la sémantique avant l'introduction des modes flottants "strict" et "par défaut" dans Java SE 1.2.

Le modificateur strictfp est ainsi obsolète en Java 17 : sa présence n'a plus d'effet car il n'y a plus de différence de sémantique.

Un avertissement concernant le lint de type strictfp est émis par le compilateur javac lors de l'utilisation du modificateur strictfp. Ce lint est activé par défaut.

Exemple (code Java 1.2) :

```
package fr.jmdoudoux.dej;

public class MaClasse {

  public strictfp double multiplier(double d1, double d2) {
    return d1 * d2;
  }
}
```

Résultat :

```
C:\java>javac -version
javac 17

C:\java>javac src\fr\jmdoudoux\dej\MaClasse.java
src\fr\jmdoudoux\dej\MaClasse.java:5: warning: [strictfp] as of release 17, all
 floating-point expressions are evaluated strictly and 'strictfp' is not required
    public strictfp double multiplier(double d1, double d2) {
                           ^
1 warning
```

Comme à partir de Java 17, le mot clé strictfp est ignoré, la solution est de le retirer dans le code source.

Exemple (code Java 17) :

```
package fr.jmdoudoux.dej;

public class MaClasse {

  public double multiplier(double d1, double d2) {
    return d1 * d2;
  }
}
```

Résultat :

```
C:\java>javac src\fr\jmdoudoux\dej\MaClasse.java

C:\java>
```

11.4.3.4.22. synchronization

Plusieurs restrictions d'utilisation sur des instances de type value based doivent être respectées par les développeurs pour éviter de futurs problèmes : le comportement lié à l'identité des classes value based peut changer dans une version ultérieure de Java. Par exemple, la synchronisation pourrait ne pas fonctionner comme attendu.

Une de ces restrictions est qu'il ne faut pas utiliser une instance d'une classe valued based comme moniteur de synchronisation.

A partir Java 16, les classes value based sont annotées avec @ValueBased ce qui permet de les identifier par le compilateur et la JVM.

Via la JEP 390, le compilateur émet un avertissement de type synchronization lorsqu'une instance d'un objet identifié comme value based est utilisée comme moniteur de synchronisation.

L'avertissement de type synchronization est activée par défaut.

Exemple :

```
package fr.jmdoudoux.dej;

public class MaClasse {

  public static void main(String[] args) {
    Integer i = 0;
    synchronized (i) {
      // traitements
    }
  }
}
```

Résultat :

```
C:\java>javac src\fr\jmdoudoux\dej\MaClasse.java
src\fr\jmdoudoux\dej\MaClasse.java:7: warning: [synchronization] attempt to synchronize on an
 instance of a value-based class
    synchronized (i) {
    ^
1 warning
```

Certains cas ne sont pas détectés par le compilateur notamment si l'instance de type value based est assignée à une variable d'un type non value based utilisée comme moniteur de synchronisation. Ces cas peuvent cependant être détectés par la JVM en utilisant l'option -XX:DiagnoseSyncOnValueBasedClasses.

Exemple :

```
package fr.jmdoudoux.dej;

public class MaClasse {

  public static void main(String[] args) {
    Integer i =0;
    Object o = i;
    synchronized (o) {
      // traitements
    }
  }
}
```

Résultat :

```
C:\java>javac src\fr\jmdoudoux\dej\MaClasse.java

C:\java>java -cp src fr.jmdoudoux.dej.MaClasse

C:\java>java -XX:+UnlockDiagnosticVMOptions -XX:DiagnoseSyncOnValueBasedClasses=2 -cp src
 fr.jmdoudoux.dej.MaClasse
[0.078s][info][valuebasedclasses] Synchronizing on object 0x000000008972a038 of klass java
.lang.Integer
[0.079s][info][valuebasedclasses]       at fr.jmdoudoux.dej.MaClasse.main(MaClasse.java:8)
[0.079s][info][valuebasedclasses]       - locked <0x000000008972a038> (a java.lang.Integer)
```

La solution pour éviter cet avertissement est d'utiliser une instance d'un type qui ne soit pas value based ni de type String. Cela peut être simplement une instance de type Object.

Exemple :

```
package fr.jmdoudoux.dej;

public class MaClasse {

  public static void main(String[] args) {
    Object i = new Object();
    synchronized (i) {
      // traitements
    }
  }
}
```

Résultat :

```
C:\java>javac src\fr\jmdoudoux\dej\MaClasse.java

C:\java>
```

11.4.3.4.23. text-blocks

Dans un bloc de texte, la gestion de l'indentation accessoire se fait en utilisant des caractères d'espacements. Plusieurs caractères d'espacement peuvent être utilisés : espace, tabulation, ...

Dans l'exemple ci-dessous, l'indentation de ligne1 et ligne3 utilise des espaces et l'indentation de ligne2 utilise des tabulations.

Exemple (code Java 13) :

```
package fr.jmdoudoux.dej;

public class MaClasse {

  public static void main(String[] args) {
    String colors = """
        ligne1
        ligne2
        ligne3""";
  }
}
```

Le compilateur javac ne peut pas savoir comment les caractères de tabulation sont affichés dans les différents éditeurs et environnements. Par conséquent, la règle veut que chaque caractère d'espacement soit traité de la même manière. Un espace unique est traité de la même manière qu'une tabulation unique, même si cette dernière peut entraîner un espacement équivalent à plusieurs espaces lorsqu'elle est affichée. Ainsi, le mélange de caractères d'espacement dans l'indentation accessoire peut avoir des effets incohérents et involontaires.

Résultat :

```
C:\java>javac src\fr\jmdoudoux\dej\MaClasse.java

C:\java>java -cp src fr.jmdoudoux.dej.MaClasse
       ligne1
ligne2
       ligne3
```

Comme il n'est pas recommandé d'utiliser différents caractères d'espacement dans l'indentation accessoire d'un même bloc de texte, le compilateur javac peut émettre un avertissement de type text-blocks à partir de Java 13.

L'avertissement n'est pas activé par défaut par le compilateur javac : il faut demander son activation en utilisant l'option -Xlint, -Xlint:all, ou -Xlint:text-blocks.

Résultat :

```
C:\java>javac -Xlint src\fr\jmdoudoux\dej\MaClasse.java
src\fr\jmdoudoux\dej\MaClasse.java:6: warning: [text-blocks] inconsistent white space indentation
    String colors = """
                    ^
1 warning
```

La solution pour éviter cet avertissement est d'utiliser le même caractère d'espacement dans l'indentation d'un bloc de texte.

Si le lint text-blocks est activé, le compilateur émet aussi un avertissement si des caractères d'espacement se trouve à la fin d'une ou plusieurs lignes d'un bloc de texte.

Dans l'exemple ci-dessous, la ligne contenant ligne2 se termine par quatre espaces.

Exemple (code Java 13) :

```
package fr.jmdoudoux.dej;

public class MaClasse {

  public static void main(String[] args) {
    String colors = """
        ligne1
        ligne2    
        ligne3""";
  }
}
```

Résultat :

```
C:\java>javac -Xlint src\fr\jmdoudoux\dej\MaClasse.java
src\fr\jmdoudoux\dej\MaClasse.java:6: warning: [text-blocks] trailing white space will be removed
    String lignes = """
                    ^
1 warning
```

11.4.3.4.24. try

Une mauvaise utilisation d'une instruction try provoque un avertissement de la part du compilateur. C'est par exemple le cas, si la ressource n'est pas utilisée dans le bloc de code de l'instruction try.

Exemple (Java 7) :

```
import java.io.FileReader;
import java.io.IOException;

public class MaClasse {
  public static void main(String... args) {
    try (FileReader reader = new FileReader("monfichier.txt")) {
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }
  }
}
```

Résultat :

```
C:\java> javac -Xlint:all MaClasse.java
MaClasse.java:7: warning: [try] auto-closeable resource reader is never
referenced in body of corresponding try statement
    try (FileReader reader = new FileReader("monfichier.txt")) {
                    ^
1 warning
```

Il est possible de désactiver l'avertissement du compilateur relatif à une mauvaise utilisation d'une instruction try.

Exemple :

```
import java.io.FileReader;
import java.io.IOException;

public class MaClasse {

  @SuppressWarnings("try")
  public static void main(String... args) {
    try (FileReader reader = new FileReader("monfichier.txt")) {
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }
  }
}
```

Résultat :

```
C:\java>javac -Xlint:try MaClasse.java

C:\java>
```

11.4.3.4.25. unchecked

Les avertissements de type unchecked indique que le compilateur ne peut pas garantir la vérification de type lors de l'utilisation d'un type possédant un type generic sans préciser le type à utiliser. La sécurité de type (type-safety) signifie qu'un programme est considéré comme sûr s'il compile sans erreurs ni avertissements et ne génère pas d'exception inattendue de type ClassCastException lors de son exécution.

Lors de l'utilisation de génériques, il est fréquent d'avoir des avertissements de type unchecked de différentes origines :

- unchecked cast
- unchecked method invocation
- unchecked parameterized vararg type
- unchecked conversion

Une bonne connaissance de l'utilisation des génériques permet de réduire les avertissements.

Exemple :

```
import java.util.List;
import java.util.ArrayList;

public class MaClasse {
  public static void main(String... args) {
    List liste = new ArrayList();
    liste.add("element1");
  }
}
```

Résultat :

```
C:\java> javac -Xlint:unchecked MaClasse.java
MaClasse.java:9: warning: [unchecked] unchecked call to add(E) as a member
of the raw type List

liste.add("element1");
                 ^
  where E is a type-variable:
    E extends Object declared in interface List
1 warning
```

Les avertissements de type unchecked sont fréquents dans un code legacy écrit avant Java 5 mixé avec du code utilisant des generics.

Exemple :

```
import java.util.List;
import java.util.ArrayList;

public class MaClasse {

  public static void main(String... args) {
    List liste = new ArrayList<String>();
    List<Integer> entiers = liste;
  }
}
```

Résultat :

```
C:\java>javac -Xlint:unchecked MaClasse.java
MaClasse.java:9: warning: [unchecked] unchecked conversion
        List<Integer> entiers = liste;
                                ^
  required: List<Integer>
  found:    List
1 warning
```

Ce genre d'affectation est autorisée par le compilateur à cause du type erasure qui permet d'assurer la compatibilité avec du code avant les génériques. Mais un avertissement prévient le développeur d'une possible erreur de type ClassCastException à l'exécution.

Le compilateur émet aussi un avertissement de type unchecked pour prévenir d'une possible pollution du tas (heap pollution). A cause du type erasure, le compilateur ne peut pas vérifier le type d'objets utiliser dans les collections de l'exemple ci-dessous :

Exemple :

```
import java.util.List;
import java.util.ArrayList;

public class MaClasse {

  public static void main(String... args) {
    List liste = new ArrayList<String>();
    liste.add("1");
    List<Integer> entiers = liste;
    Integer i = entiers.get(0);
  }
}
```

Résultat :

```
C:\java>javac -Xlint:unchecked MaClasse.java
MaClasse.java:8: warning: [unchecked] unchecked call to add(E) as a member
of the raw type List
    liste.add("1");
             ^
  where E is a type-variable:
    E extends Object declared in interface List
MaClasse.java:9: warning: [unchecked] unchecked conversion
    List<Integer> entiers = liste;
                            ^
  required: List<Integer>
  found:    List
2 warnings
C:\java>java MaClasse
Exception in thread "main" java.lang.ClassCastException:
java.lang.String cannot be cast to java.lang.Integer
        at MaClasse.main(MaClasse.java:10)
```

L'utilisation de certains frameworks peut aussi générer des avertissements de type unchecked lorsque leurs API peuvent manipuler plusieurs types n'ayant aucun type en commun. C'est notamment le cas avec l'API Java EE JPA qui permet de manipuler potentiellement d'importe qu'elle type d'entité.

Exemple :

```
@SuppressWarnings("unchecked")
public List<Personne> obtenirPersonnesToutes(){
    Query query = entitymanager.createQuery("select p from Personne p");
    return (List<Personne>)query.getResultList();
}
```

Il est possible de désactiver ce type avertissement.

Exemple :

```
import java.util.List;
import java.util.ArrayList;

public class MaClasse {

  @SuppressWarnings("unchecked")
  public static void main(String... args) {
    List liste = new ArrayList();
    liste.add("element1");
  }
}
```

Résultat :

```
C:\java> javac -Xlint:unchecked MaClasse.java

C:\java>
```

La suppression d'un avertissement de type unchecked ne devrait être utilisée que si l'on est sûr que le code est type-safe.

Exemple (code Java 5.0) :

```
import java.util.ArrayList;
import java.util.Collection;

public class MaClasse {

  public static void main(String[] args) {
    traiter(new ArrayList<Object>());
  }

  public static void traiter(Object object) {
    if (object instanceof Collection) {
      Collection<?> coll = (Collection<?>) object;
      calculer(coll);
    }
  }

  public static void calculer(Collection<Object> object) {
    System.out.println("Collection<Object>");
  }

  public static void calculer(Object object) {
    System.out.println("Object");
  }
}
```

Résultat :

```
C:\java>javac -Xlint:unchecked MaClasse.java
C:\java>java MaClasse
Object
```

Le code ci-dessus se compile sans avertissement mais c'est la surcharge qui attend un paramètre de type Object qui est invoquée. Pour que cela soit la surcharge qui attend un paramètre de type Collection, il faut préciser le type générique Object et donc effectuer un cast.

Exemple (code Java 5.0) :

```
  public static void traiter(Object object) {
    if (object instanceof Collection) {
      Collection<Object> coll = (Collection<Object>) object;
      calculer(coll);
    }
  }
```

Résultat :

```
C:\java>javac -Xlint:unchecked MaClasse.java
MaClasse.java:12: warning: [unchecked] unchecked cast

Collection<Object> coll = (Collection<Object>) object;
                                                    ^

required: Collection<Object>

found:    Object
1 warning
C:\java>java MaClasse
Collection<Object>
```

Le compilateur émet un avertissement suite au cast. Cependant ce cast est sûr puisque ce dernier est conditionné par le fait que le paramètre soit de type Collection. Dans ce cas, il est possible de demander la suppression de l'avertissement.

Exemple (code Java 5.0) :

```
  public static void traiter(Object object) {
    if (object instanceof Collection) {
      @SuppressWarnings("unchecked")
      Collection<Object> coll = (Collection<Object>) object;
      calculer(coll);
    }
  }
```

Résultat :

```
C:\java>javac -Xlint:unchecked MaClasse.java
C:\java>java MaClasse
Collection<Object>
```

Les avertissements de type unchecked sont importants et ne devraient pas être ignorés. Un tel avertissement implique la possibilité d'avoir une ClassCastException à l'exécution.

Il ne faut utiliser l'annotation @SuppressWarnings("unchecked") que si l'avertissement ne peut être résolu et que l'on soit sûr que le code est type-safe.

Ce n'est pas une bonne pratique de délibérément utiliser un type générique sans préciser le générique et d'utiliser l'annotation @SuppressWarnings

11.4.3.5. Les bonnes pratiques d'utilisation

La suppression des avertissements doit se faire avec d'extrêmes précautions.

Un avertissement permet au compilateur de signaler qu'il a trouvé quelque chose qui semble louche. Cela ne veut pas dire que c'est louche, mais ça y ressemble pour le compilateur.

Parfois, il est possible de modifier le code pour éliminer l'avertissement.

Parfois, le code est bon mais il n'est pas possible d'éviter un avertissement. Dans ce cas, très rare, il est possible de demander au compilateur d'ignorer l'avertissement. C'est un dernier recours qui ne doit être utilisé que si c'est justifié.

Cependant, il est fréquent de trouver des annotations @SuppressWarning utilisées simplement pour désactiver les avertissements afin d'améliorer artificiellement la qualité du code lié à la réduction du nombre d'avertissements. Il est préférable dans ce cas, de passer du temps pour résoudre l'avertissement lorsque c'est possible plutôt que d'ignorer systématiquement les avertissements.

Il ne faut donc pas ignorer un warning émis par le compilateur car il nous signale un problème potentiel même si celui-ci n'empêche pas la compilation. Par exemple, un avertissement de type unchecked est un risque potentiel de voir une exception de type ClassCastException au runtime.

Il est important de s'assurer qu'il n'est pas possible de corriger le code pour éviter l'émission d'un warning : si cela n'est pas possible ou le warning est vraiment injustifié alors il est possible de demander au compilateur de l'ignorer en utilisant l'annotation @SuppressWarnings.

L'annotation @SuppressWarning peut être appliquée sur une classe, un champ, une méthode, un constructeur, un paramètre ou une variable en locale. Il n'est pas recommandé d'utiliser l'annotation @SuppressWarnings au niveau d'une classe : il est préférable de l'utiliser sur l'élément le plus bas qui génère un avertissement. Ceci évite de masquer d'autres avertissements du même type pouvant être émis par des éléments sous-jacent.

Il est donc important de n'utiliser l'annotation @SuppressWarning qu'au niveau le plus bas, le plus prêt de l'origine du warning. Il est par exemple préférable de mettre l'annotation @SuppressWarnings sur une variable locale plutôt que sur la méthode qui la contienne.

Il est fortement recommandé de mettre un commentaire précisant les raisons de l'utilisation de @SuppressWarnings.

Il est fortement déconseillé d'utiliser @SuppressWarnings("all") car elle demande d'ignorer tous les warnings et ainsi masquer de futurs avertissements.

11.5. Les annotations communes (Common Annotations)

Les annotations communes sont définies par la JSR 250 et sont intégrées dans Java 6. Leur but est de définir des annotations couramment utilisées et ainsi d'éviter leur redéfinition pour chaque outil qui en aurait besoin.

Les annotations définies concernent :

- la plate-forme standard dans le package javax.annotation (@Generated, @PostConstruct, @PreDestroy, @Resource, @Resources)
- la plate-forme entreprise dans le package javax.annotation.security (@DeclareRoles, @DenyAll, @PermitAll, @RolesAllowed, @RunAs).

11.5.1. L'annotation @Generated

De plus en plus d'outils ou de frameworks génèrent du code source pour faciliter la tâche des développeurs notamment pour des portions de code répétitives ayant peu de valeur ajoutée.

Le code ainsi généré peut être marqué avec l'annotation @Generated.

Exemple :

```
@Generated(
    value = "entite.qui.a.genere.le.code",
    comments = "commentaires",
    date = "12 April  2008"
)
public void toolGeneratedCode(){
}
```

L'attribut obligatoire value permet de préciser l'outil à l'origine de la génération

Les attributs facultatifs comments et date permettent respectivement de fournir un commentaire et la date de génération.

Cette annotation peut être utilisée sur toutes les déclarations d'entités.

11.5.2. Les annotations @Resource et @Resources

L'annotation @Resource définit une ressource requise par une classe. Typiquement, une ressource est par exemple un composant Java EE de type EJB ou JMS.

L'annotation @Resource possède plusieurs attributs :

Attribut	Description
authentificationType	Type d'authentification pour utiliser la ressource (Resource.AuthenticationType.CONTAINER ou Resource.AuthenticationType.APPLICATION)
description	Description de la ressource
mappedName	Nom de la ressource spécifique au serveur utilisé (non portable)
name	Nom JNDI de la ressource
shareable	Booléen qui précise si la ressource est partagée
type	Le type pleinement qualifié de la ressource

Cette annotation peut être utilisée sur une classe, un champ ou une méthode.

Lorsque l'annotation est utilisée sur une classe, elle correspond simplement à une déclaration des ressources qui seront requises à l'exécution.

Lorsque l'annotation est utilisée sur un champ ou une méthode, le serveur d'applications va injecter une référence sur la ressource correspondante. Pour cela, lors du chargement d'une application par le serveur d'applications, celui-ci recherche les annotations @Resource afin d'assigner une instance de la ressource correspondante.

Exemple :

```
@Resource(name="MaQueue",
    type = "javax.jms.Queue",
    shareable=false,
    authenticationType=Resource.AuthenticationType.CONTAINER,
    description="Queue de test"
)
private javax.jms.Queue maQueue;
```

L'annotation @Resources est simplement une collection d'annotation de type @Resource.

Exemple :

```
@Resources({
    @Resource(name = "maQueue" type = javax.jms.Queue),
    @Resource(name = "monTopic" type = javax.jms.Topic),
})
```

11.5.3. Les annotations @PostConstruct et @PreDestroy

Les annotations @PostConstruct et @PreDestroy permettent respectivement de désigner des méthodes qui seront exécutées après l'instanciation d'un objet et avant la destruction d'une instance.

Ces deux annotations ne peuvent être utilisées que sur des méthodes.

Ces annotations sont par exemple utiles dans Java EE car généralement un composant géré par le conteneur est instancié en utilisant le constructeur sans paramètre. Une méthode marquée avec l'annotation @PostConstruct peut alors être exécutée juste après l'appel au constructeur.

Une telle méthode doit respecter certaines règles :

- ne pas avoir de paramètres sauf dans des cas précis (exemple avec les intercepteurs des EJB)
- ne pas avoir de valeur de retour (elle doit renvoyer void)
- ne doit pas lever d'exceptions vérifiées
- ne doit pas être statique

L'annotation @PostConstruct est utilisée en général sur une méthode qui initialise des ressources en fonction du contexte.

Dans une même classe, chacune de ces annotations n'est utilisable que par une seule méthode.

11.6. Les annotations personnalisées

Java propose la possibilité de définir ses propres annotations. Pour cela, le langage possède un type dédié : le type d'annotation (annotation type).

Un type d'annotation est similaire à une classe et une annotation est similaire à une instance de classe.

11.6.1. La définition d'une annotation

Sur la plate-forme Java, une annotation est une interface lors de sa déclaration et est une instance d'une classe qui implémente cette interface lors de son utilisation.

La définition d'une annotation nécessite une syntaxe particulière utilisant le mot clé @interface. Une annotation se déclare donc de façon similaire à une interface.

Exemple : le fichier MonAnnotation.java

```
package fr.jmdoudoux.dej.annotations;

public @interface MonAnnotation {

}
```

Une fois compilée, cette annotation peut être utilisée dans le code. Pour utiliser une annotation, il faut importer l'annotation et l'appeler dans le code en la faisant précéder du caractère @.

Exemple :

```
package fr.jmdoudoux.dej.annotations;

@MonAnnotation
public class MaCLasse {

}
```

Si l'annotation est définie dans un autre package, il faut utiliser la syntaxe pleinement qualifiée du nom de l'annotation ou ajouter une clause import pour le package.

Il est possible d'ajouter des membres à l'annotation simplement en définissant une méthode dont le nom correspond au nom de l'attribut en paramètre de l'annotation.

Exemple :

```
package fr.jmdoudoux.dej.annotations;

public @interface MonAnnotation {
  String arg1();
  String arg2();
}

package fr.jmdoudoux.dej.annotations;

@MonAnnotation(arg1="valeur1", arg2="valeur2")
public class MaCLasse {

}
```

Les types utilisables sont les chaînes de caractères, les types primitifs, les énumérations, les annotations, les chaînes de caractères, le type Class.

Il est possible de définir un membre comme étant un tableau à une seule dimension d'un des types utilisables.

Exemple :

```
package fr.jmdoudoux.dej.annotations

public @interface MonAnnotation {
    String arg1();
    String[] arg2();
    String arg3();
}
```

Il est possible de définir une valeur par défaut, ce qui rend l'indication du membre optionnelle. Cette valeur est précisée en la faisant précéder du mot clé default.

Exemple :

```
package fr.jmdoudoux.dej.annotations;

public @interface MonAnnotation {
  String arg1() default "";
  String[] arg2();
  String arg3();
}
```

La valeur par défaut d'un tableau utilise une syntaxe raccourcie.

Exemple :

```
package fr.jmdoudoux.dej.annotations

public @interface MonAnnotation {
    String arg1();
    String[] arg2() default {"chaine1", "chaine2" };
    String arg3();
}
```

Il est possible de définir une énumération comme type pour un attribut

Exemple :

```
package fr.jmdoudoux.dej.annotations;

public @interface MonAnnotation {
  public enum Niveau {DEBUTANT, CONFIRME, EXPERT} ;
  String arg1() default "";
  String[] arg2();
  String arg3();
  Niveau niveau() default Niveau.DEBUTANT;

}
```

11.6.2. Les annotations pour les annotations

La version 5 de Java propose quatre annotations dédiées aux types d'annotations qui permettent de fournir des informations sur l'utilisation.

Ces annotations sont définies dans le package java.lang.annotation

11.6.2.1. L'annotation @Target

L'annotation @Target permet de préciser les entités sur lesquelles l'annotation sera utilisable. Cette annotation attend comme valeur un tableau de valeurs issues de l'énumération ElementType.

Valeur de l'énumération	Rôle
ANNOTATION_TYPE	Types d'annotation
CONSTRUCTOR	Constructeurs
LOCAL_VARIABLE	Variables locales
FIELD	Champs
METHOD	Méthodes hors constructeurs
PACKAGE	Packages
PARAMETER	Paramètres d'une méthode ou d'un constructeur
TYPE	Classes, interfaces, énumérations, types d'annotation

Si une annotation est utilisée sur une entité non précisée par l'annotation, alors une erreur est émise lors de la compilation.

Exemple :

```
package fr.jmdoudoux.dej.annotations;

import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

@Target({ElementType.METHOD, ElementType.CONSTRUCTOR })
public @interface MonAnnotation {
  String arg1() default "";
  String arg2();
}

package fr.jmdoudoux.dej.annotations;

@MonAnnotation(arg1="valeur1", arg2="valeur2")
public class MaCLasse {

}
```

Résultat de la compilation :

```
C:\Documents and Settings\jmd\workspace\Tests>javac com/jmdoudoux/test/annotatio
ns/MaClasse.java
com\jmdoudoux\test\annotations\MaClasse.java:3: annotation type not applicable t
```

```
o this kind of declaration
@MonAnnotation(arg1="valeur1", arg2="valeur2")
^
1 error
```

11.6.2.2. L'annotation @Retention

Cette annotation permet de préciser à quel niveau les informations concernant l'annotation seront conservées. Cette annotation attend comme valeur un élément de l'énumération RetentionPolicy.

Enumération	Rôle
RetentionPolicy.SOURCE	informations conservées dans le code source uniquement (fichier .java) : le compilateur les ignore
RetentionPolicy.CLASS	informations conservées dans le code source et le bytecode (fichiers .java et .class)
RetentionPolicy.RUNTIME	informations conservées dans le code source et le bytecode : elles sont disponibles à l'exécution par introspection

Cette annotation permet de déterminer de quelle façon l'annotation pourra être exploitée.

Exemple :

```
@Retention(RetentionPolicy.RUNTIME)
```

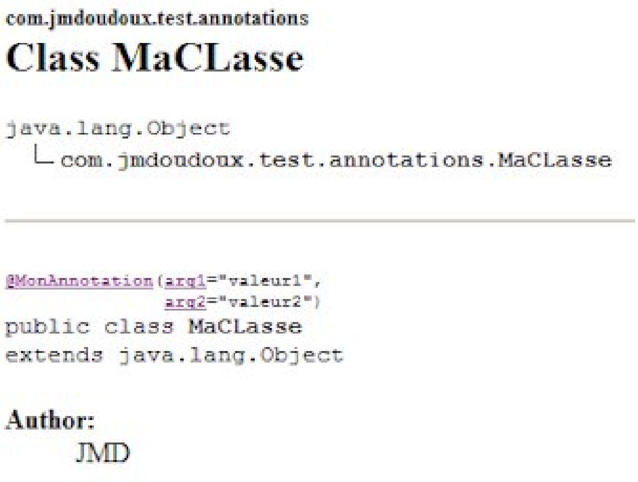
11.6.2.3. L'annotation @Documented

L'annotation @Documented permet de demander l'intégration de l'annotation dans la documentation générée par Javadoc.

Par défaut, les annotations ne sont pas intégrées dans la documentation des classes annotées.

Exemple :

```
package fr.jmdoudoux.dej.annotations;

import java.lang.annotation.Documented;

@Documented
public @interface MonAnnotation {
  String arg1() default "";
  String arg2();
}
```



com.jmdoudoux.test.annotations

Class MaCLasse

```
java.lang.Object
  └ com.jmdoudoux.test.annotations.MaCLasse
```

```
@MonAnnotation(arg1="valeur1",
               arg2="valeur2")
public class MaCLasse
extends java.lang.Object
```

Author:

JMD

11.6.2.4. L'annotation @Inherited

L'annotation @Inherited permet de demander l'héritage d'une annotation aux classes filles de la classe mère sur laquelle elle s'applique.

Si une classe mère est annotée avec une annotation elle-même annotée avec @Inherited alors toutes les classes filles sont automatiquement annotées avec cette annotation.

11.7. L'exploitation des annotations

Pour être profitables, les annotations ajoutées dans le code source doivent être exploitées par un ou plusieurs outils.

La déclaration et l'utilisation d'annotations sont relativement simples par contre leur exploitation pour permettre la production de fichiers est moins triviale.

Cette exploitation peut se faire de plusieurs manières

- en définissant un doclet qui exploite le code source
- en utilisant apt au moment de la compilation
- en utilisant l'introspection lors de l'exécution
- en utilisant le compilateur java à partir de Java 6.0

11.7.1. L'exploitation des annotations dans un Doclet

Pour des traitements simples, il est possible de définir un Doclet et de le traiter avec l'outil Javadoc pour utiliser les annotations.

L'API Doclet est définit dans le package com.sun.javadoc. Ce package est dans le fichier tools.jar fourni avec le JDK.

L'API Doclet définit des interfaces pour chaque entité pouvant être utilisée dans le code source.

La méthode annotation() de l'interface ProgramElementDoc permet d'obtenir un tableau de type AnnotationDesc.

L'interface AnnotationDesc représente une annotation. Elle définit deux méthodes :

Méthode	Rôle
AnnotationTypeDoc annotationType()	Renvoyer le type d'annotation
AnnotationDesc.ElementValuePair[] elementValues()	Renvoyer les éléments de l'annotation

L'interface AnnotationTypeDoc représente un type d'annotation. Elle ne définit qu'une seule méthode :

Méthode	Rôle
AnnotationTypeElementDoc[] elements()	Renvoyer les éléments d'un type d'annotation

L'interface AnnotationTypeElementDoc représente un élément d'un type d'annotation. Elle ne définit qu'une seule méthode :

Méthode	Rôle
AnnotationValue defaultValue()	Renvoyer la valeur par défaut de l'élément d'un type d'annotation

L'interface AnnotationValue représente la valeur d'un élément d'un type d'annotation. Elle définit deux méthodes :

Méthode	Rôle
String toString()	Renvoyer la valeur sous forme d'une chaîne de caractères
Object value()	Renvoyer la valeur

Pour créer un Doclet, il faut définir une classe qui contienne une méthode ayant pour signature `public static boolean start (RootDoc rootDoc).`

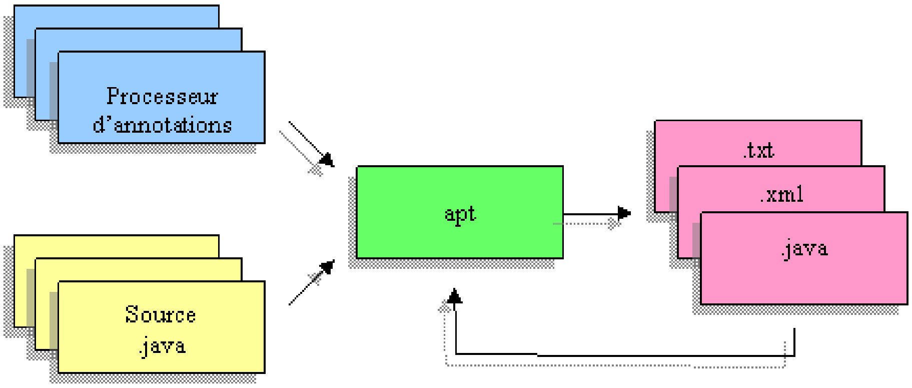
Pour utiliser le Doclet, il faut compiler la classe qui l'encapsule et utiliser l'outil javadoc avec l'option -doclet suivi du nom de la classe.

11.7.2. L'exploitation des annotations avec l'outil Apt

La version 5 du JDK fournit l'outil apt pour le traitement des annotations.

L'outil apt qui signifie annotation processing tool est l'outil le plus polyvalent en Java 5 pour exploiter les annotations.

Apt assure la compilation des classes et permet en simultané le traitement des annotations par des processeurs d'annotations créés par le développeur.



Les processeurs d'annotations peuvent générer de nouveaux fichiers sources, pouvant eux-mêmes contenir des annotations. Apt traite alors récursivement les fichiers générés jusqu'à ce qu'il n'y ait plus d'annotation à traiter et de classe à compiler.

Cette section va créer un processeur pour l'annotation personnalisée Todo

Exemple : l'annotation personnalisée Todo

```
package fr.jmdoudoux.dej.annotations;

import java.lang.annotation.Documented;

@Documented
public @interface Todo {

  public enum Importance {
    MINEURE, IMPORTANT, MAJEUR, CRITIQUE
  };

  Importance importance() default Importance.MINEURE;

  String[] description();

  String assigneA();

  String dateAssignation();
}
```

Apt et l'API à utiliser de concert ne sont disponibles qu'avec le JDK : ils ne sont pas fournis avec le JRE.

Les packages de l'API sont dans le fichier lib/tools.jar du JDK : cette bibliothèque doit donc être ajoutée au classpath lors de la mise en oeuvre d'apt.

L'API est composée de deux grandes parties :

- Modélisation du langage
- Interaction avec l'outil de traitement des annotations

L'API est contenue dans plusieurs sous-packages de com.sun.mirror notamment :

- com.sun.mirror.apt : contient les interfaces pour la mise en oeuvre d'apt
- com.sun.mirror.declaration : encapsule la déclaration des entités dans les sources qui peuvent être annotées (packages, classes, méthodes, ...) sous le forme d'interfaces qui héritent de l'interface Declaration
- com.sun.mirror.type : encapsule les types d'entités dans les sources sous la forme d'interfaces qui héritent de l'interface TypeMirror
- com.sun.mirror.util : propose des utilitaires

Un processeur d'annotations est une classe qui implémente l'interface com.sun.mirror.apt.AnnotationProcessor. Cette interface ne définit qu'une seule méthode process() qui va contenir les traitements à réaliser pour une annotation.

Il faut fournir un constructeur qui attend en paramètre un objet de type com.sun.mirror.apt.AnnotationProcessorEnvironment : ce constructeur sera appelé par une fabrique pour en créer une instance.

L'interface AnnotationProcessorEnvironment fournit des méthodes pour obtenir des informations sur l'environnement d'exécution des traitements des annotations et créer de nouveaux fichiers pendant les traitements.

L'interface Declaration permet d'obtenir des informations sur une entité :

Méthode	Rôle
<A extends Annotation> getAnnotation(Class<A> annotationType)	Renvoie une annotation d'un certain type associée à l'entité
Collection<AnnotationMirror> getAnnotationMirrors()	Renvoie les annotations associées à l'entité
String getDocComment()	Renvoie le texte des commentaires de documentations Javadoc associés à l'entité
Collection<Modifier> getModifiers()	Renvoie les modificateurs de l'entité
SourcePosition getPosition()	Renvoie la position de la déclaration dans le code source
String getSimpleName()	Renvoie le nom de la déclaration

De nombreuses interfaces héritent de l'interface Declaration : AnnotationTypeDeclaration, AnnotationTypeElementDeclaration, ClassDeclaration, ConstructorDeclaration, EnumConstantDeclaration, EnumDeclaration, ExecutableDeclaration, FieldDeclaration, InterfaceDeclaration, MemberDeclaration, MethodDeclaration, PackageDeclaration, ParameterDeclaration, TypeDeclaration, TypeParameterDeclaration

Chacune de ces interfaces propose des méthodes pour obtenir des informations sur la déclaration et sur le type concernés.

L'interface TypeMirror permet d'obtenir des informations sur un type utilisé dans une déclaration.

De nombreuses interfaces héritent de l'interface TypeMirror : AnnotationType, ArrayType, ClassType, DeclaredType, EnumType, InterfaceType, PrimitiveType, ReferenceType, TypeVariable, VoidType, WildcardType.

La classe com.sun.mirror.util.DeclarationFilter permet de définir un filtre des entités annotées avec les annotations concernées par les traitements du processeur. Il suffit de créer une instance de cette classe en ayant redéfini sa méthode match(). Cette méthode renvoie un booléen qui précise si l'entité fournie en paramètre sous la forme d'un objet de type Declaration est annotée avec une des annotations concernées par le processeur.

Exemple :

```
package fr.jmdoudoux.dej.annotations.outils;

import java.util.Collection;

import fr.jmdoudoux.dej.annotations.Todo;
import com.sun.mirror.apt.AnnotationProcessor;
import com.sun.mirror.apt.AnnotationProcessorEnvironment;
import com.sun.mirror.declaration.Declaration;
import com.sun.mirror.declaration.TypeDeclaration;
import com.sun.mirror.util.DeclarationFilter;

public class TodoAnnotationProcessor implements AnnotationProcessor {
  private final AnnotationProcessorEnvironment env;

  public TodoAnnotationProcessor(AnnotationProcessorEnvironment env) {
    this.env = env;
  }

  public void process() {
    // Création d'un filtre pour ne retenir que les déclarations annotées avec Todo
    DeclarationFilter annFilter = new DeclarationFilter() {
      public boolean matches(
          Declaration d) {
        return d.getAnnotation(Todo.class) != null;
      }
    };

    // Recherche des entités annotées avec Todo
    Collection<TypeDeclaration> types = annFilter.filter(env.getSpecifiedTypeDeclarations());
    for (TypeDeclaration typeDecl : types) {
      System.out.println("class name: " + typeDecl.getSimpleName());

      Todo todo = typeDecl.getAnnotation(Todo.class);

      System.out.println("description : ");
      for (String desc : todo.description()) {
        System.out.println(desc);
      }
      System.out.println("");
    }
  }
}
```

Il faut créer une fabrique de processeurs d'annotations : cette fabrique est en charge d'instancier des processeurs d'annotations pour un ou plusieurs types d'annotations. La fabrique doit implémenter l'interface com.sun.mirror.apt.AnnotationProcessorFactory.

L'interface AnnotationProcessorFactory déclare trois méthodes :

Méthode	Rôle
AnnotationProcessor getProcessorFor(Set<AnnotationTypeDeclaration> atds, AnnotationProcessorEnvironment env)	Renvoyer un processeur d'annotations pour l'ensemble de types d'annotations fournis en paramètres.
Collection<String> supportedAnnotationTypes()	Renvoyer une collection des types d'annotations dont un processeur peut être instancié par la fabrique

Collection<String> supportedOptions()	Renvoyer une collection des options supportées par la fabrique ou par les processeurs d'annotations créés par la fabrique

Exemple :

```
package fr.jmdoudoux.dej.annotations.outils;

import com.sun.mirror.apt.*;
import com.sun.mirror.declaration.*;

import java.util.Collection;
import java.util.Set;
import java.util.Collections;
import java.util.Arrays;

public class TodoAnnotationProcessorFactory implements AnnotationProcessorFactory {
  private static final Collection<String> supportedAnnotations =
    Collections.unmodifiableCollection(Arrays
      .asList("fr.jmdoudoux.dej.annotations.Todo"));

  private static final Collection<String> supportedOptions     = Collections.emptySet();

  public Collection<String> supportedOptions() {
    return supportedOptions;
  }

  public Collection<String> supportedAnnotationTypes() {
    return supportedAnnotations;
  }

  public AnnotationProcessor getProcessorFor(
      Set<AnnotationTypeDeclaration> atds,
      AnnotationProcessorEnvironment env) {
    return new TodoAnnotationProcessor(env);
  }
}
```

Dans l'exemple ci-dessus, aucune option n'est supportée et la fabrique ne prend en charge que l'annotation personnalisée Todo.

Pour mettre en oeuvre les traitements des annotations, il faut que le code source utilise ces annotations.

Exemple : une classe annotée avec l'annotation Todo

```
package fr.jmdoudoux.dej;

import fr.jmdoudoux.dej.annotations.Todo;
import fr.jmdoudoux.dej.annotations.Todo.Importance;

@Todo(importance = Importance.CRITIQUE,
      description = "Corriger le bug dans le calcul",
      assigneA = "JMD",
      dateAssignation = "11-11-2007")
public class MaClasse {

}
```

Exemple : une autre classe annotée avec l'annotation Todo

```
package fr.jmdoudoux.dej;

import fr.jmdoudoux.dej.annotations.Todo;
import fr.jmdoudoux.dej.annotations.Todo.Importance;

@Todo(importance = Importance.MAJEUR,
      description = "Ajouter le traitement des erreurs",
      assigneA = "JMD",
      dateAssignation = "07-11-2007")
public class MaClasse1 {

}
```

Pour utiliser apt, il faut que le classpath contienne la bibliothèque tools.jar fournie avec le JDK et les classes de traitements des annotations (fabrique et processeur d'annotations).

L'option -factory permet de préciser la fabrique à utiliser.

Résultat de l'exécution d'apt

```
C:\Documents and Settings\jmd\workspace\Tests>apt -cp ".;./bin;C:/Program Files/
Java/jdk1.5.0_07/lib/tools.jar" -factory fr.jmdoudoux.dej.annotations.outils.T
odoAnnotationProcessorFactory com/jmdoudoux/test/*.java
class name: MaClasse
description :
Corriger le bug dans le calcul

class name: MaClasse1
description :
Ajouter le traitement des erreurs
```

A partir de l'objet de type AnnotationProcessorEnvironment, il est possible d'obtenir un objet de type com.sun.miror.apt.Filer qui encapsule un nouveau fichier créé par le processeur d'annotations.

L'interface Filer propose quatre méthodes pour créer différents types de fichiers :

Méthode	Rôle
OutputStream createBinaryFile(Filer.Location loc, String pkg, File relPath)	Créer un nouveau fichier binaire et renvoyer un objet de type Stream pour écrire son contenu
OutputStream createClassFile(String name)	Créer un nouveau fichier .class et renvoyer un objet de type Stream pour écrire son contenu
PrintWriter createSourceFile(String name)	Créer un nouveau fichier texte contenant du code source et renvoyer un objet de type Writer pour écrire son contenu
PrintWriter createTextFile(Filer.Location loc, String pkg, File relPath, String charsetName)	Créer un nouveau fichier texte et renvoyer un objet de type Writer pour écrire son contenu

L'énumération Filter.Location permet de préciser si le nouveau fichier est créé dans la branche source (SOURCE_TREE) ou dans la branche compilée (CLASS_TREE).

Exemple :

```
package fr.jmdoudoux.dej.annotations.outils;

import java.io.File;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Collection;

import fr.jmdoudoux.dej.annotations.Todo;
import com.sun.mirror.apt.AnnotationProcessor;
import com.sun.mirror.apt.AnnotationProcessorEnvironment;
import com.sun.mirror.apt.Filer;
import com.sun.mirror.declaration.Declaration;
import com.sun.mirror.declaration.TypeDeclaration;
import com.sun.mirror.util.DeclarationFilter;

public class TodoAnnotationProcessor implements AnnotationProcessor {
  private final AnnotationProcessorEnvironment env;
  public TodoAnnotationProcessor(AnnotationProcessorEnvironment env) {
    this.env = env;
  }

  public void process() {
    // Création d'un filtre pour ne retenir que les déclarations annotées avec
    // Todo
    DeclarationFilter annFilter = new DeclarationFilter() {
      public boolean matches(
          Declaration d) {
        return d.getAnnotation(Todo.class) != null;
      }
    };

    Filer f = this.env.getFiler();
    PrintWriter out;
    try {
      out = f.createTextFile(Filer.Location.SOURCE_TREE, "", new File("todo.txt"), null);

      // Recherche des entités annotées avec Todo
      Collection<TypeDeclaration> types = annFilter.filter(env.getSpecifiedTypeDeclarations());
      for (TypeDeclaration typeDecl : types) {
        out.println("class name: " + typeDecl.getSimpleName());

        Todo todo = typeDecl.getAnnotation(Todo.class);

        out.println("description : ");
        for (String desc : todo.description()) {
          out.println(desc);
        }
        out.println("");
      }

      out.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat de l'exécution

```
C:\Documents and Settings\jm\workspace\Tests>apt -cp ".;./bin;C:/Program Files/
Java/jdk1.5.0_07/lib/tools.jar" -factory fr.jmdoudoux.dej.annotations.outils.T
odoAnnotationProcessorFactory com/jmdoudoux/test/*.java

C:\Documents and Settings\jm\workspace\Tests>dir
 Volume in drive C has no label.
 Volume Serial Number is 1D31-4F67

 Directory of C:\Documents and Settings\jmd\workspace\Tests

19/11/2007  08:39    <DIR>          .
19/11/2007  08:39    <DIR>          ..
16/11/2007  08:15               433 .classpath
31/10/2006  14:06               381 .project
14/09/2007  12:45    <DIR>          .settings
16/11/2007  08:15    <DIR>          bin
02/10/2007  15:22               854 build.xml
29/06/2007  07:12    <DIR>          com
15/11/2007  13:01    <DIR>          doc
19/11/2007  08:39               148 todo.txt
              8 File(s)          1 812 bytes
               6 Dir(s)  66 885 595 136 bytes free

C:\Documents and Settings\jm\workspace\Tests>type todo.txt
class name: MaClasse
description :
Corriger le bug dans le calcul

class name: MaClasse1
description :
```

```
Ajouter le traitement des erreurs
```

Concernant les entités à traiter, l'API Mirror fournit de nombreuses autres fonctionnalités qui permettent de rendre très riche le traitement des annotations. Parmi ces fonctionnalités, il y a le parcours des sources par des classes mettant en oeuvre le motif de conception visiteur.

11.7.3. L'exploitation des annotations par introspection

Pour qu'une annotation soit exploitée à l'exécution, il est nécessaire qu'elle soit annotée avec une RetentionPolicy à la valeur RUNTIME.

Exemple :

```
package fr.jmdoudoux.dej.annotations;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Documented
@Retention(RetentionPolicy.RUNTIME)
public @interface Todo {

  public enum Importance {
    MINEURE, IMPORTANT, MAJEUR, CRITIQUE
  };

  Importance importance() default Importance.MINEURE;

  String[] description();

  String assigneA();

  String dateAssignation();
}
```

L'interface java.lang.reflect.AnnotatedElement définit les méthodes pour le traitement des annotations par introspection :

Méthode	Rôle
<T extends Annotation> getAnnotation(Class<T>)	Renvoyer l'annotation si le type fourni en paramètre est utilisé sur l'entité, sinon null
Annotation[] getAnnotations()	Renvoyer un tableau de toutes les annotations de l'entité. Renvoie un tableau vide si aucune annotation n'est concernée
Annotation[] getDeclaredAnnotations()	Renvoyer un tableau des annotations directement associées à l'entité (en ignorant donc les annotations héritées). Renvoie un tableau vide si aucune annotation n'est concernée
boolean isAnnotationPresent(Class< ? extends Annotation>)	Renvoyer true si l'annotation dont le type est fourni en paramètre est utilisé sur l'entité. Cette méthode est particulièrement utile dans le traitement des annotations de type marqueur.

Plusieurs classes du package java.lang implémentent l'interface AnnotatedElement : AccessibleObject, Class, Constructor, Field, Method et Package

Exemple :

```
package fr.jmdoudoux.dej;

import java.lang.reflect.Method;
import fr.jmdoudoux.dej.annotations.Todo;
import fr.jmdoudoux.dej.annotations.Todo.Importance;

@Todo(importance = Importance.CRITIQUE,
      description = "Corriger le bug dans le calcul",
      assigneA = "JMD",
      dateAssignation = "11-11-2007")
public class TestInstrospectionAnnotation {

  public static void main(
      String[] args) {
    Todo todo = null;

    // traitement annotation sur la classe
    Class classe = TestInstrospectionAnnotation.class;
    todo = (Todo) classe.getAnnotation(Todo.class);
    if (todo != null) {
      System.out.println("classe "+classe.getName());
      System.out.println("  ["+todo.importance()+"]"+" ("+todo.assigneA()
            +" le "+todo.dateAssignation()+")");
      for(String desc : todo.description()) {
        System.out.println("     _ "+desc);
      }
    }

    // traitement annotation sur les méthodes de la classe
    for(Method m : TestInstrospectionAnnotation.class.getMethods()) {
      todo = (Todo) m.getAnnotation(Todo.class);
      if (todo != null) {
        System.out.println("méthode "+m.getName());
        System.out.println("  ["+todo.importance()+"]"+" ("+todo.assigneA()
                  +" le "+todo.dateAssignation()+")");
        for(String desc : todo.description()) {
          System.out.println("     _ "+desc);
        }
      }
    }
  }

  @Todo(importance = Importance.MAJEUR,
        description = "Implémenter la methode",
        assigneA = "JMD",
        dateAssignation = "11-11-2007")
  public void methode1() {

  }

  @Todo(importance = Importance.MINEURE,
        description = {"Compléter la methode", "Améliorer les logs"},
        assigneA = "JMD",
        dateAssignation = "12-11-2007")
  public void methode2() {

  }

}
```

Résultat d'exécution :

```
classe fr.jmdoudoux.dej.TestInstrospectionAnnotation
  [CRITIQUE] (JMD le 11-11-2007)
     _ Corriger le bug dans le calcul
méthode methode1
  [MAJEUR] (JMD le 11-11-2007)
     _ Implémenter la methode
méthode methode2
  [MINEURE] (JMD le 12-11-2007)
     _ Compléter la methode
     _ Améliorer les logs
```

Pour obtenir les annotations sur les paramètres d'un constructeur ou d'une méthode, il faut utiliser la méthode getParameterAnnotations() des classes Constructor ou Method qui renvoie un objet de type Annotation[][]. La première dimension du tableau concerne les paramètres dans leur ordre de déclaration. La seconde dimension contient les annotations de chaque paramètre.

11.7.4. L'exploitation des annotations par le compilateur Java

Dans la version 6 de Java SE, la prise en compte des annotations est intégrée dans le compilateur : ceci permet un traitement à la compilation des annotations sans avoir recours à un outil tiers comme apt.

Une nouvelle API a été définie par la JSR 269 (Pluggable annotations processing API) et ajoutée dans la package javax.annotation.processing.

Cette API est détaillée dans la section suivante.

11.8. L'API Pluggable Annotation Processing

La version 6 de Java apporte plusieurs améliorations dans le traitement des annotations notamment l'intégration de ces traitements directement dans le compilateur javac grâce à une nouvelle API dédiée.

L'API Pluggable Annotation Processing est définie dans la JSR 269. Elle permet un traitement des annotations directement par le compilateur en proposant une API aux développeurs pour traiter les annotations incluses dans le code source.

Apt et son API proposaient déjà une solution à ces traitements mais cette API standardise le traitement des annotations au moment de la compilation. Il n'est donc plus nécessaire d'utiliser un outil tiers post compilation pour traiter les annotations à la compilation.

Dans les exemples de cette section, les classes suivantes seront utilisées :

Exemple : MaClasse.java

```
package fr.jmdoudoux.dej;

import fr.jmdoudoux.dej.annotations.Todo;
import fr.jmdoudoux.dej.annotations.Todo.Importance;

@Todo(importance = Importance.CRITIQUE,
      description = "Corriger le bug dans le calcul",
      assigneA = "JMD",
      dateAssignation = "11-11-2007")
public class MaClasse {

}
```

Exemple : MaClasse1.java

```
package fr.jmdoudoux.dej;

import fr.jmdoudoux.dej.annotations.Todo;
import fr.jmdoudoux.dej.annotations.Todo.Importance;

@Todo(importance = Importance.MAJEUR,
      description = "Ajouter le traitement des erreurs",
      assigneA = "JMD",
      dateAssignation = "07-11-2007")
public class MaClasse1 {

}
```

Exemple : MaClasse3.java

```
package fr.jmdoudoux.dej;

@Deprecated
public class MaClasse3 {

}
```

Un exemple de mise en oeuvre de l'API est aussi fourni avec le JDK dans le sous-répertoire sample/javac/processing.

11.8.1. Les processeurs d'annotations

La mise en oeuvre de cette API nécessite l'utilisation des packages javax.annotation.processing, javax.lang.model et javax.tools.

Un processeur d'annotations doit implémenter l'interface Processor. Le traitement des annotations se fait en plusieurs passes (round). A chaque passe le processeur est appelé pour traiter des classes qui peuvent avoir été générées lors de la précédente passe. Lors de la première passe, ce sont les classes fournies initialement qui sont traitées.

L'interface javax.annotation.processing.Processor définit les méthodes d'un processeur d'annotations. Pour définir un processeur, il est possible de créer une classe qui implémente l'interface Processor mais le plus simple est d'hériter de la classe abstraite javax.annotation.processing.AbstractProcessor.

La classe AbstractProcessor contient une variable nommée processingEnv de type ProcessingEnvironment. La classe ProcessingEnvironment permet d'obtenir des instances de classes qui permettent des interactions avec l'extérieur du processeur ou fournissent des utilitaires :

- Filer : classe qui permet la création de fichiers
- Messager : classe qui permet d'envoyer des messages affichés par le compilateur
- Elements : classe qui fournit des utilitaires pour les éléments
- Types : classe qui fournit des utilitaires pour les types

La méthode getRootElements() renvoie les classes Java qui seront traitées par le processeur dans cette passe.

La méthode la plus importante est la méthode process() : c'est elle qui va contenir les traitements exécutés par le processeur. Elle possède deux paramètres :

- Un ensemble des annotations qui seront traitées par le processeur
- Un objet qui encapsule l'étape courante des traitements

Deux annotations sont dédiées aux processeurs d'annotations et doivent être utilisées sur la classe du processeur :

- @SupportedAnnotationTypes : cette annotation permet de préciser les types d'annotations traitées par le processeur. La valeur « * » permet d'indiquer que tous seront traités.
- @SupportedSourceVersion : cette annotation permet de préciser la version du code source traité par le processeur

Exemple :

```
package fr.jmdoudoux.dej.annotations.outils;

import java.util.Set;

import javax.annotation.processing.AbstractProcessor;
import javax.annotation.processing.Messager;
import javax.annotation.processing.RoundEnvironment;
import javax.annotation.processing.SupportedAnnotationTypes;
import javax.annotation.processing.SupportedSourceVersion;
import javax.lang.model.SourceVersion;
import javax.lang.model.element.Element;
import javax.lang.model.element.TypeElement;
import javax.tools.Diagnostic.Kind;

import fr.jmdoudoux.dej.annotations.Todo;

@SupportedAnnotationTypes(value = { "*" })
@SupportedSourceVersion(SourceVersion.RELEASE_6)
public class TodoProcessor extends AbstractProcessor {

  @Override
  public boolean process(
      Set<? extends TypeElement> annotations,
      RoundEnvironment roundEnv) {

    Messager messager = processingEnv.getMessager();

    for (TypeElement te : annotations) {
      messager.printMessage(Kind.NOTE, "Traitement annotation "
            + te.getQualifiedName());

      for (Element element : roundEnv.getElementsAnnotatedWith(te)) {
        messager.printMessage(Kind.NOTE, "  Traitement élément "
                  + element.getSimpleName());
        Todo todo = element.getAnnotation(Todo.class);

        if (todo != null) {
          messager.printMessage(Kind.NOTE, "  affecté le " + todo.dateAssignation()
                    + " a " + todo.assigneA());
        }
      }
    }

    return true;
  }
}
```

11.8.2. L'utilisation des processeurs par le compilateur

Le compilateur javac est enrichi avec plusieurs options concernant le traitement des annotations :

Option	Rôle
-processor	permet de préciser le nom pleinement qualifié du processeur à utiliser
-proc	vérifie si le traitement des annotations et/ou la compilation sont effectués
-processorpath	classpath des processeurs d'annotations
-A	permet de passer des options aux processeurs d'annotations sous la forme de paires cle=valeur
-XprintRounds	option non standard qui permet d'afficher des informations sur le traitement des annotations par les processeurs
-XprintProcessorInfo	option non standard qui affiche la liste des annotations qui seront traitées par les processeurs d'annotations

Le compilateur fait appel à la méthode process() du processeur en lui passant en paramètre l'ensemble des annotations trouvées par le compilateur dans le code source.

Résultat :

```
C:\Documents and Settings\jm\workspace\TestAnnotations>javac -cp ".;./bin;C:/Pr
ogram Files/Java/jdk1.6.0/lib/tools.jar" -processor fr.jmdoudoux.dej.annotati
ons.outils.TodoProcessor com/jmdoudoux/tests/*.java
Note: Traitement annotation fr.jmdoudoux.dej.annotations.Todo
Note:   Traitement élément MaClasse
Note:   affecte le 11-11-2007 a JMD
```

```
Note:   Traitement élément MaClasse1
Note:   affecte le 07-11-2007 a JMD
Note: Traitement annotation java.lang.Deprecated
Note:   Traitement élément MaClasse3
```

11.8.3. La création de nouveaux fichiers

La classe Filer permet de créer des fichiers lors du traitement des annotations.

Exemple :

```
package fr.jmdoudoux.dej.annotations.outils;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Set;

import javax.annotation.processing.AbstractProcessor;
import javax.annotation.processing.Filer;
import javax.annotation.processing.Messager;
import javax.annotation.processing.RoundEnvironment;
import javax.annotation.processing.SupportedAnnotationTypes;
import javax.annotation.processing.SupportedSourceVersion;
import javax.lang.model.SourceVersion;
import javax.lang.model.element.Element;
import javax.lang.model.element.TypeElement;
import javax.lang.model.util.Elements;
import javax.tools.StandardLocation;
import javax.tools.Diagnostic.Kind;

import fr.jmdoudoux.dej.annotations.Todo;

@SupportedAnnotationTypes(value = { "*" })
@SupportedSourceVersion(SourceVersion.RELEASE_6)
public class TodoProcessor2 extends AbstractProcessor {

  @Override
  public boolean process(
      Set<? extends TypeElement> annotations,
      RoundEnvironment roundEnv) {

    Filer filer = processingEnv.getFiler();
    Messager messager = processingEnv.getMessager();
    Elements eltUtils = processingEnv.getElementUtils();
    if (!roundEnv.processingOver()) {
      TypeElement elementTodo =
            eltUtils.getTypeElement("fr.jmdoudoux.dej.annotations.Todo");
      Set<? extends Element> elements = roundEnv.getElementsAnnotatedWith(elementTodo);
      if (!elements.isEmpty())
        try {
          messager.printMessage(Kind.NOTE, "Création du fichier Todo");
          PrintWriter pw = new PrintWriter(filer.createResource(
                      StandardLocation.SOURCE_OUTPUT, "", "Todo.txt")
              .openOutputStream());
          // .createSourceFile("Todo").openOutputStream());
          pw.println("Liste des todos\n");

          for (Element element : elements) {
            pw.println("\nélément:" + element.getSimpleName());
            Todo todo = (Todo) element.getAnnotation(Todo.class);
            pw.println("  affecté le " + todo.dateAssignation()
                          + " a " + todo.assigneA());
            pw.println("  description : ");
            for (String desc : todo.description()) {
              pw.println("    " + desc);
            }
          }

          pw.close();
        } catch (IOException ioe) {
          messager.printMessage(Kind.ERROR, ioe.getMessage());
        }
      else
        messager.printMessage(Kind.NOTE, "Rien à faire");
    } else
      messager.printMessage(Kind.NOTE, "Fin des traitements");

    return true;
  }
}
```

Résultat :

```
C:\Documents and Settings\jmd\workspace\TestAnnotations>javac -cp ".;./bin;C:/Pr
ogram Files/Java/jdk1.6.0/lib/tools.jar" -processor fr.jmdoudoux.dej.annotati
ons.outils.TodoProcessor2 com/jmdoudoux/tests/*.java
Note: Création du fichier Todo
Note: Fin des traitements

C:\Documents and Settings\jmd\workspace\TestAnnotations>type Todo.txt
Liste des todos


élément:MaClasse
  affecté le 11-11-2007 a JMD
  description :
    Corriger le bug dans le calcul

element:MaClasse1
  affecté le 07-11-2007 a JMD
  description :
    Ajouter le traitement des erreurs
```

11.9. Les ressources relatives aux annotations

La JSR 175 A Metadata Facility for the JavaTM Programming Language

La JSR 269 Pluggable Annotation Processing API

La JSR 250 Common Annotations

La page des annotations dans le tutorial Java

Le projet open source XDoclet qui propose la génération de code à partir d'attributs dans le code

12. Les expressions lambda

Chapitre 12

Java est un langage orienté objet : à l'exception des instructions et des données primitives, tout le reste est objets, même les tableaux et les chaînes de caractères.

Java ne propose pas la possibilité de définir une fonction/méthode en dehors d'une classe ni de passer une telle fonction en paramètre d'une méthode. Depuis Java 1.1, la solution pour passer des traitements en paramètres d'une méthode est d'utiliser les classes anonymes internes.

Pour faciliter, entre autres, cette mise à oeuvre, Java 8 propose les expressions lambda. Les expressions lambda sont aussi nommées closures ou fonctions anonymes : leur but principal est de permettre de passer en paramètre un ensemble de traitements.

De plus, la programmation fonctionnelle est devenue prédominante dans les langages récents. Dans ce mode de programmation, le résultat de traitements est décrit mais pas la façon dont ils sont réalisés. Ceci permet de réduire la quantité de code à écrire pour obtenir le même résultat.

Par exemple, pour afficher les éléments d'une liste

Exemple :

```
for (int i = 0; i < list.size(); i++) {
  System.out.println(list.get(i));
}
```

Ce code est composé de deux traitements : une boucle pour itérer sur chaque élément et l'affichage d'un élément dans l'itération. Le code de la boucle pourrait être factorisé dans une méthode qui attendrait en paramètre une fonction qui contenant le traitement à réaliser sur chaque élément.

Exemple (code Java 8) :

```
list.forEach(System.out::println);
```

La mise en oeuvre de cette fonctionnalité requiert deux fonctionnalités de Java 8 :

- les méthodes par défaut pour définir la méthode foreach() dans l'interface Iterable qui attend en paramètre une interface fonctionnelle de type Consumer
- les expressions lambda pour permettre de passer en paramètre une fonction, sous la forme d'une référence de méthode dans l'exemple

Les expressions lambda sont donc une des plus importantes nouveautés de Java 8 voire même la plus importante évolution apportée au langage Java depuis sa création. C'est donc logiquement la fonctionnalité la plus médiatique de Java 8.

Les spécifications des expressions lambda sont définies dans la JSR 335 (Project Lambda).

Ce chapitre contient plusieurs sections :

♦ L'historique des lambdas pour Java

- ♦ Les expressions lambda
- ♦ Les références de méthodes
- ♦ Les interfaces fonctionnelles

12.1. L'historique des lambdas pour Java

L'ajout des expressions lambda dans le langage Java a été un processus long qui a nécessité plus de huit années de travail.

Les premières propositions datent de 2006. En 2006-2007, plusieurs propositions de spécifications s'opposent :

- BGGA : proposée par Gilad Bracha, Neal Gafter, James Gosling et Peter von der Ahé. Elle utilise une syntaxe avec l'opérateur =>. Elle définit les functions type, l'annotation @Shared pour pouvoir modifier une variable du contexte, les methods references avec l'opérateur #
- CICE : proposée par Bob Lee, Doug Lea et Josh Bloch. Elle utilise une écriture simplifiée des classes anonymes internes
- FCM : proposée par Stephen Colebourne et Stefan Schulz. Elle est plus simple que BGGA et plus complète que CICE. La définition de method types se fait avec l'opérateur #. Elle permet la modification des variables du contexte englobant

Aucune de ces propositions ne sera intégralement retenue mais les meilleures idées sont utilisées pour spécifier les expressions lambda :

- ne pas ajouter un nouveau type fonction au langage pour éviter les écueils des generics
- s'appuyer sur les interfaces qui existent déjà et permettent donc une transition en douceur
- les expressions lambda ne sont pas transformées en classes par le compilateur : elles n'utilisent donc pas les classes anonymes internes

Lors de la conception des expressions lambda, le choix a été fait de ne pas ajouter un type spécial dans le langage, ce qui limite les fonctionnalités d'une expression lambda par rapport à leur équivalent dans d'autres langages mais réduit les impacts dans le langage Java. Par exemple, il n'est pas possible d'assigner une expression lambda à une variable de type Object parce que le type Object n'est pas une interface fonctionnelle.

Ce n'est qu'en 2014, avec Java 8, que les expressions lambda qui permettent la mise en oeuvre d'une forme de closures sont intégrées dans le langage Java. Avant Java 8, la seule solution était d'utiliser une classe anonyme interne.

Exemple :

```
monBouton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent event) {
    System.out.println("clic");
  }
});
```

Dans ce cas, une nouvelle instance du type de l'interface est passée en paramètre. Avec Java 8, il est possible d'utiliser une expression lambda.

Exemple (code Java 8) :

```
monBouton.addActionListener(event -> System.out.println("clic"));
```

Dans ce cas, c'est une expression lambda qui est passée en paramètre. Elle permet de définir une implémentation d'une interface fonctionnelle sous la forme d'une expression composée d'une liste de paramètres et d'un corps qui peut être une simple expression ou un bloc de code.

Une expression lambda est utilisée pour représenter une interface fonctionnelle sous la forme d'une expression de la forme :

(arguments) -> corps

L'opérateur -> sépare le ou les paramètres du bloc de code qui va les utiliser. Le type du paramètre n'est pas obligatoire : le compilateur va tenter de réaliser une inférence du type pour le déterminer selon le contexte. Dans l'exemple ci-dessus, le compilateur va déterminer que le paramètre de l'interface ActionListener est de type ActionEvent.

Une expression lambda est typée de manière statique. Ce type doit être une interface fonctionnelle.

Les exemples ci-dessous illustrent leur utilisation pour définir un thread. Avant Java 8, pour définir un thread, il était possible de créer une classe anonyme de type Runnable et de passer son instance en paramètre du constructeur d'un thread.

Exemple :

```
Thread monThread = new Thread(new Runnable() {
  @Override
  public void run(){
    System.out.println("Mon traitement ");
  }
});
monThread.start();
```

A partir de Java 8, il est possible d'utiliser une expression lambda en remplacement de la classe anonyme pour obtenir le même résultat avec une syntaxe plus concise.

Exemple :

```
Thread monThread = new Thread(() -> { System.out.println("Mon traitement"); });
monThread.start();
```

L'utilisation d'une expression lambda évite d'avoir à écrire le code nécessaire à la déclaration de la classe anonyme et de la méthode.

Associées à d'autres fonctionnalités du langage (méthode par défaut, ...) et de l'API (Stream), les lambdas modifient profondément la façon dont certaines fonctionnalités sont codées en Java. Cet impact est plus important que certaines fonctionnalités des versions précédentes de Java comme les generics ou les annotations.

12.2. Les expressions lambda

Les expressions lambda permettent d'écrire du code plus concis, donc plus rapide à écrire, à relire et à maintenir. C'est aussi un élément important dans l'introduction de la programmation fonctionnelle dans le langage Java qui était jusqu'à la version 8 uniquement orienté objet.

Une expression lambda est une fonction anonyme : sa définition se fait sans déclaration explicite du type de retour, ni de modificateurs d'accès ni de nom. C'est un raccourci syntaxique qui permet de définir une méthode directement à l'endroit où elle est utilisée.

Une expression lambda est donc un raccourci syntaxique qui simplifie l'écriture de traitements passés en paramètre. Elle est particulièrement utile notamment lorsque le traitement n'est utile qu'une seule fois : elle évite d'avoir à écrire une méthode dans une classe.

Une expression lambda permet d'encapsuler un traitement pour être passé à d'autres traitements. C'est un raccourci syntaxique aux classes anonymes internes pour une interface qui ne possède qu'une seule méthode abstraite. Ce type d'interface est nommé interface fonctionnelle.

Lorsque l'expression lambda est évaluée par le compilateur, celui-ci infère le type vers l'interface fonctionnelle. Cela lui permet d'obtenir des informations sur les paramètres utilisés, le type de la valeur de retour, les exceptions qui peuvent être levées.

Elles permettent d'écrire du code plus compact et plus lisible. Elles ne réduisent pas l'aspect orienté objet du langage qui a toujours été une force mais au contraire, rendent celui-ci plus riche et plus élégant pour certaines fonctionnalités.

12.2.1. La syntaxe d'une expression lambda

Un des avantages des expressions lambda est d'avoir une syntaxe très simple.

La syntaxe d'une expression lambda est composée de trois parties :

- un ensemble de paramètres, d'aucun à plusieurs
- l'opérateur ->
- le corps de la fonction

Elle peut prendre deux formes principales :

(paramètres) -> expression;

(paramètres) -> { traitements; }

L'écriture d'une expression lambda doit respecter plusieurs règles générales :

- zéro, un ou plusieurs paramètres dont le type peut être déclaré explicitement ou inféré par le compilateur selon le contexte
- les paramètres sont entourés par des parenthèses et séparés par des virgules. Des parenthèses vides indiquent qu'il n'y a pas de paramètre
- lorsqu'il n'y a qu'un seul paramètre et que son type est inféré alors les parenthèses ne sont pas obligatoires
- le corps de l'expression peut contenir zéro, une ou plusieurs instructions. Si le corps ne contient d'une seule instruction, les accolades ne sont pas obligatoires et le type de retour correspond à celui de l'instruction. Lorsqu'il y a plusieurs instructions alors elles doivent être entourées avec des accolades

12.2.1.1. Les paramètres d'une expression lambda

Dans la définition d'une expression lambda, l'opérateur -> permet de séparer les paramètres des traitements qui les utiliseront. Les paramètres de l'expression doivent donc être déclarés à gauche de l'opérateur ->.

Les paramètres d'une expression lambda doivent correspondre à ceux définis dans l'interface fonctionnelle.

Les paramètres de l'expression doivent respecter certaines règles :

- une expression peut n'avoir aucun, un seul ou plusieurs paramètres
- le type des paramètres peuvent être explicitement déclaré ou être inféré par le compilateur selon le contexte dans lequel l'expression est utilisée
- les paramètres sont entourés de parenthèses, chacun étant séparé par une virgule
- des parenthèses vides indiquent qu'il n'y a pas de paramètre
- s'il n'y a qu'un seul paramètre dont le type n'est pas explicitement précisé, alors l'utilisation des parenthèses n'est pas obligatoire

Une expression lambda peut ne pas avoir de paramètre.

Exemple (code Java 8) :

```java
Runnable monTraitement = () -> System.out.println("traitement");
```

Pour préciser qu'il n'y a aucun paramètre, il faut utiliser une paire de parenthèses vide. Dans ce cas, la méthode de l'interface fonctionnelle ne doit pas avoir de paramètre.

Si l'expression lambda ne possède qu'un seul paramètre alors il y a deux syntaxes possibles. La plus standard est d'indiquer le paramètre entre deux parenthèses.

Exemple (code Java 8) :

```
Consumer<String> afficher = (param) -> System.out.println(param);
```

Il est aussi possible d'omettre les parenthèses uniquement si le type peut être inféré.

Exemple (code Java 8) :

```
Consumer<String> afficher = param -> System.out.println(param);
```

Il n'est pas possible d'omettre les parenthèses si le type est précisé explicitement.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.Consumer;

public class TestLambda {

  public static void main(String[] args) {
    Consumer<String> afficher = String param -> System.out.println(param);
  }
}
```

Résultat :

```
C:\java\TestJava8\src>javac -cp . com/jmdoudoux/dej/java8/lambda/TestLambda.java
com\jmdoudoux\dej\java8\lambda\TestLambda.java:10: error: ';' expected
    Consumer<String> afficher = String param -> System.out.println(param);
                                      ^
com\jmdoudoux\dej\java8\lambda\TestLambda.java:10: error: not a statement
    Consumer<String> afficher = String param -> System.out.println(param);
                                       ^
2 errors
```

Si l'expression lambda possède plusieurs paramètres alors ils doivent être entourés obligatoirement par des parenthèses et séparés les uns des autres par une virgule.

Exemple (code Java 8) :

```
    BiFunction<Integer, Integer, Long> additionner = (val1, val2) -> (long) val1 + val2;
```

Il est possible de déclarer explicitement le type du ou des paramètres de l'expression lambda.

Exemple (code Java 8) :

```
Consumer<String> afficher = (String param) -> System.out.println(param);
```

Généralement, le type d'un paramètre n'est pas obligatoire si le compilateur est capable de l'inférer : dans la plupart des cas, le compilateur est capable de déterminer le type du paramètre à partir de celui correspondant à l'interface fonctionnelle. Si ce n'est pas le cas, le compilateur génère une erreur et il est alors nécessaire de préciser ce type explicitement.

Dans l'exemple ci-dessus, le fait que l'interface fonctionnelle Consumer soit typée avec un generic String permet au compilateur de savoir que le type du paramètre est String. L'exemple ci-dessus peut alors être écrit sans préciser le type du paramètre.

Exemple (code Java 8) :

```
Consumer<String> afficher = (param) -> System.out.println(param);
```

Le type précisé pour un paramètre doit correspondre à celui défini dans le type l'interface fonctionnelle.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.Consumer;

public class TestLambda {

  public static void main(String[] args) {

    Consumer afficher = (String param) -> System.out.println(param);
  }
}
```

L'exemple ci-dessus génère une erreur à la compilation.

Résultat :

```
C:\java\TestJava8\src>javac -cp . com/jmdoudoux/dej/java8/lambda/TestLambda.java
com\jmdoudoux\dej\java8\lambda\TestLambda.java:16: error: incompatible types: in
compatible parameter types in lambda expression
    Consumer afficher = (String param) -> System.out.println(param);
                        ^
1 error
```

Il n'est cependant pas possible de mixer dans la même expression des paramètres déclarés explicitement et inférés.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.Comparator;

public class TestLambda {

  public static void main(String[] args) {

    Comparator<String> comparator =
      (chaine1, String chaine2) -> Integer.compare(chaine1.length(),chaine2.length());
  }
}
```

Résultat :

```
C:\java\TestJava8\src>javac -cp . com/jmdoudoux/dej/java8/lambda/TestLambda.java
com\jmdoudoux\dej\java8\lambda\TestLambda.java:10: error: <identifier> expected
      (chaine1, String chaine2) -> Integer.compare(chaine1.length(),chaine2.leng
th());
              ^
1 error
```

Il est possible d'utiliser le modificateur final sur les paramètres si leur type est précisé explicitement.

Exemple (code Java 8) :

```
 Comparator<String> comparator = (final String chaine1, final String chaine2)
   -> Integer.compare(chaine1.length(),chaine2.length());
```

Ce n'est pas possible si le type est inféré par le compilateur sinon il génère une erreur.

Exemple (code Java 8) :

```
 Comparator<String> comparator = (final chaine1, final chaine2)
      -> Integer.compare(chaine1.length(),chaine2.length());
```

Résultat :

```
C:\java\TestJava8\src>javac -cp . com/jmdoudoux/dej/java8/lambda/TestLambda.java
com\jmdoudoux\dej\java8\lambda\TestLambda.java:12: error: <identifier> expected
    Comparator<String> comparator = (final chaine1, final chaine2) -> Integer.co
mpare(chaine1.length(),chaine2.length());
                                                  ^
com\jmdoudoux\dej\java8\lambda\TestLambda.java:12: error: <identifier> expected
    Comparator<String> comparator = (final chaine1, final chaine2) -> Integer.co
mpare(chaine1.length(),chaine2.length());
                                                                 ^
2 errors
```

Il est aussi possible d'annoter les paramètres d'une expression lambda uniquement si leur type est déclaré explicitement.

Exemple (code Java 8) :

```
    Comparator<String> comparator = (@NotNull String chaine1, @NotNull String chaine2)
      -> Integer.compare(chaine1.length(),chaine2.length());
```

12.2.1.2. Le corps d'une expression lambda

Le corps d'une expression lambda est défini à droite de l'opérateur ->. Il peut être :

- une expression unique
- un bloc de code composé d'une ou plusieurs instructions entourées par des accolades

Le corps de l'expression doit respecter certaines règles :

- il peut n'avoir aucune, une seule ou plusieurs instructions
- lorsqu'il ne contient qu'une seule instruction, les accolades ne sont pas obligatoires et la valeur de retour est celle de l'instruction si elle en possède une
- lorsqu'il y a plusieurs instructions, elles doivent obligatoirement être entourées d'accolades
- la valeur de retour est celle de la dernière expression ou void si rien n'est retourné

Si le corps est simplement une expression, celle-ci est évaluée et le résultat de cette évaluation est renvoyé s'il y en a un.

Exemple (code Java 8) :

```
    BiFunction<Integer, Integer, Long> additionner = (val1, val2) -> (long) val1 + val2;
```

Il n'est jamais nécessaire de préciser explicitement le type de retour : le compilateur doit être en mesure de le déterminer selon le contexte. Si ce n'est pas le cas, le compilateur émet une erreur.

Si l'expression ne produit pas de résultat, celle-ci est simplement exécutée. Il n'est pas nécessaire d'entourer d'accolades si le corps de l'expression invoque une méthode dont la valeur de retour est void.

Exemple (code Java 8) :

```
  Consumer<String> afficher = (String param) -> System.out.println(param);
```

Les traitements d'une expression lambda peuvent contenir plusieurs opérations qui doivent être regroupées dans un bloc de code entouré d'accolades comme pour le corps d'une méthode.

Exemple (code Java 8) :

```
Runnable monTraitement = () -> {
  System.out.println("traitement 1");
  System.out.println("traitement 2");};
```

Le bloc de code est évalué comme si c'était celui d'une méthode : il est possible de terminer son exécution en utilisant l'instruction return ou en levant une exception.

Si le bloc de code doit retourner une valeur, il faut utiliser le mot clé return.

Exemple (code Java 8) :

```
    Function<Integer, Boolean> isPositif = valeur -> {
            return valeur >= 0;
    };
```

Important : toutes les branches du code du corps de l'expression doivent obligatoirement retourner une valeur ou lever une exception si au moins une branche retourne une valeur. Si ce n'est pas le cas, le compilateur émet une erreur.

Exemple (code Java 8) :

```
    Function<Integer, Boolean> isPositif = valeur -> {
          if (valeur >= 0) {
              return true;
          }
    };
```

L'exemple ci-dessus ne se compile pas car lorsque la valeur est négative, il n'y a pas de valeur de retour définie.

Résultat :

```
C:\java\TestJava8\src>javac -cp . com/jmdoudoux/dej/java8/lambda/TestLambda.java
com\jmdoudoux\dej\java8\lambda\TestLambda.java:9: error: incompatible types: bad
 return type in lambda expression
    Function<Integer, Boolean> isPositif = valeur -> {
                                         ^
    missing return value
1 error
```

Les mots clés break et continue ne peuvent pas être utilisés comme instructions de premier niveau dans le bloc de code mais ils peuvent être utilisés dans des boucles.

12.2.2. Des exemples d'expressions lambda

Cette section fournit quelques exemples d'expressions lambda.

Exemple	Description
() -> 123 () -> { return 123 };	N'accepter aucun paramètre et renvoyer la valeur 123
x -> x * 2	Accepter un nombre et renvoyer son double
(x, y) -> x + y	Accepter deux nombres et renvoyer leur somme
(int x, int y) -> x + y	Accepter deux entiers et renvoyer leur somme
(String s) -> System.out.print(s)	

	Accepter une chaîne de caractères et l'afficher sur la sortie standard sans rien renvoyer
c -> { int s = c.size(); c.clear(); return s; }	Renvoyer la taille d'une collection après avoir effacé tous ses éléments. Cela fonctionne aussi avec un objet qui possède une méthode size() renvoyant un entier et une méthode clear()
n -> n % 2 != 0;	Renvoyer un booléen qui précise si la valeur numérique est impaire
(char c) -> c == 'z';	Renvoyer un booléen qui précise si le caractère est 'z'
() -> { System.out.println("Hello World"); };	Afficher "Hello World" sur la sortie standard

(val1, val2) -> { return val1 >= val2; }
(val1, val2) -> val1 >= val2;
	Renvoyer un booléen qui précise si la première valeur est supérieure ou égale à la seconde

() -> { for (int i = 0; i < 10; i++) traiter(); }	Exécuter la méthode traiter() dix fois
p -> p.getSexe() == Sexe.HOMME && p.getAge() >= 7 && p.getAge() <= 77	Renvoyer un booléen pour un objet possédant une méthode getSexe() et getAge() qui vaut true si le sexe est masculin et l'age compris entre 7 et 77 ans

12.2.3. La portée des variables

Vis à vis de la portée et de la visibilité des variables, une expression lambda se comporte syntaxiquement comme un bloc de code imbriqué.

Comme pour les classes anonymes internes, une expression lambda peut avoir accès à certaines variables définies dans le contexte englobant.

Dans le corps d'une expression lambda, il est donc possible d'utiliser :

- les variables passées en paramètre de l'expression
- les variables définies dans le corps de l'expression
- les variables final définies dans le contexte englobant
- les variables effectivement final définies dans le contexte englobant : ces variables ne sont pas déclarées final mais une valeur leur est assignée et celle-ci n'est jamais modifiée. Il serait donc possible de les déclarer final sans que cela n'engendre de problème de compilation. Le concept de variables effectivement final a été introduit dans Java 8

Une expression lambda peut avoir accès aux variables définies dans son contexte englobant.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

public class TestPorteeLambda {

  public static void main(String[] args) {
    afficher("Bonjour",5);
  }

  public static void afficher(String message, int repetition) {
    Runnable r = () -> {
      for (int i = 0; i < repetition; i++) {
        System.out.println(message);
      }
    };
    new Thread(r).start();
  }
}
```

Dans l'exemple ci-dessus, les variables message et repetition ne sont pas définies dans l'expression lambda mais sont des paramètres de la méthode englobante qui sont accédées dans l'expression lambda. Ces variables ne sont pas déclarées final mais elles sont effectivement final donc elles sont utilisables dans l'expression lambda.

L'accès aux variables du contexte englobant est limité par une contrainte forte : seules les variables dont la valeur ne change pas peuvent être accédées.

Exemple (code Java 8) :

```java
package fr.jmdoudoux.dej.java8.lambda;

import javax.swing.JButton;

public class TestPorteeLambda {

  public static void main(String[] args) {
    // ...
    JButton bouton = new JButton("Incrementer");

    int compteur = 0;
    bouton.addActionListener(event -> compteur++);
    // ...
  }
}
```

Résultat :

```
C:\java\TestJava8\src>javac com/jmdoudoux/dej/java8/lambda/TestPorteeLambda.java
com\jmdoudoux\dej\java8\lambda\TestPorteeLambda.java:12: error: local variables
referenced from a lambda expression must be final or effectively final
    bouton.addActionListener(event -> compteur++);
                                      ^
1 error
```

La même erreur de compilation est obtenue si la modification est faite dans le contexte englobant de l'expression lambda.

Exemple (code Java 8) :

```java
package fr.jmdoudoux.dej.java8.lambda;

import javax.swing.JButton;

public class TestPorteeLambda {

  public static void main(String[] args) {
    JButton bouton = new JButton("Incrementer");

    int compteur = 0;
    compteur++;
    bouton.addActionListener(event -> System.out.println(compteur));
  }
}
```

Il est possible de passer en paramètre un objet mutable et de modifier l'état de cet objet. Le compilateur vérifie simplement que la référence que contient la variable ne change pas. Comme tout en Java, il est de la responsabilité du développeur de gérer les éventuels accès concurrents lors de ces modifications.

Comme avec les classes anonymes internes, il est aussi possible de définir un tableau d'un seul élément de type int et d'incrémenter la valeur de cet élément.

Exemple (code Java 8) :

```java
package fr.jmdoudoux.dej.java8.lambda;

import javax.swing.JButton;
```

```
public class TestPorteeLambda {

  public static void main(String[] args) {
    JButton bouton = new JButton("Incrementer");

    int[] compteur = new int[1];
    bouton.addActionListener(event -> compteur[0]++);
  }
}
```

L'utilisation de cette solution de contournement n'est pas recommandée d'autant qu'elle n'est pas thread-safe. Il est préférable d'utiliser une variable de type AtomicXXX pour le compteur.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.concurrent.atomic.AtomicInteger;
import javax.swing.JButton;

public class TestPorteeLambda {

  public static void main(String[] args) {
    JButton bouton = new JButton("Incrementer");

    AtomicInteger compteur = new AtomicInteger(0);
    bouton.addActionListener(event -> compteur.incrementAndGet());
  }
}
```

Le corps de l'expression lambda est soumis aux mêmes règles de gestion de la portée des variables qu'un bloc de code ordinaire.

Les variables locales et les paramètres déclarés dans une expression lambda doivent l'être comme s'ils l'étaient dans le contexte englobant : une expression lambda ne définit pas de nouvelle portée. Ainsi, il n'est pas possible de définir deux variables avec le même nom dans un même bloc de code, donc il n'est pas possible de définir une variable dans l'expression lambda si celle-ci est déjà définie dans le contexte englobant.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class TestComparator {

  public static void main(String[] args) {
    Personne p1 = new Personne("nom3","prenom3");
    Personne p2 = new Personne("nom1","prenom1");
    Personne p3 = new Personne("nom2","prenom2");
    List<Personne> personnes = new ArrayList(3);
    personnes.add(p1);
    personnes.add(p2);
    personnes.add(p3);

    Comparator<Personne> triParNom = (Personne p1,  Personne p2) -> {
      return p2.getNom().compareTo(p1.getNom());
    };
  }
}
```

Résultat :

```
C:\java\TestJava8\src>javac com/jmdoudoux/dej/java8/lambda/TestComparator.java
com\jmdoudoux\dej\java8\lambda\TestComparator.java:20: error: variable p1 is alr
eady defined in method main(String[])
    Comparator<Personne> triParNom = (Personne p1,  Personne p2) -> {
                                               ^
com\jmdoudoux\dej\java8\lambda\TestComparator.java:20: error: variable p2 is alr
eady defined in method main(String[])
    Comparator<Personne> triParNom = (Personne p1,  Personne p2) -> {
                                                             ^
2 errors
```

Le mot clé this fait référence à l'instance courante dans le bloc de code englobant et dans l'expression lambda.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

public class TestPorteeLambda {

  public static void main(String[] args) {
    TestPorteeLambda instance = new TestPorteeLambda();
    System.out.println(instance.toString());
    instance.executer();
  }

  public void executer() {
    Runnable runnable = () -> { System.out.println(this.toString());};
    Thread thread = new Thread(runnable);
    thread.start();
  }
}
```

Résultat :

```
fr.jmdoudoux.dej.java8.lambda.TestPorteeLambda@94b612
fr.jmdoudoux.dej.java8.lambda.TestPorteeLambda@94b612
```

Le mot clé super fait référence à l'instance de la classe mère de this.

12.2.4. L'utilisation d'une expression lambda

Une expression lambda ne peut être utilisée que dans un contexte où le compilateur peut identifier l'utilisation de son type cible (target type) qui doit être une interface fonctionnelle :

- déclaration d'une variable
- assignation d'une variable
- valeur de retour avec l'instruction return
- initialisation d'un tableau
- paramètre d'une méthode ou d'un constructeur
- corps d'une expression lambda
- opérateur ternaire ?:
- cast

Par exemple, la déclaration d'une expression lambda peut être utilisée directement en paramètre d'une méthode.

Exemple (code Java 8) :

```
monBouton.addActionListener(event -> System.out.println("clic"));
```

Il est aussi possible de définir une variable ayant pour type une interface fonctionnelle qui sera initialisée avec l'expression lambda.

Exemple (code Java 8) :

```
ActionListener monAction = event -> System.out.println("clic");

monBouton.addActionListener(monAction);
```

Une expression lambda est définie grâce à une interface fonctionnelle : une instance d'une expression lambda qui implémente cette interface est un objet. Cela permet à une expression lambda :

- de s'intégrer naturellement dans le système de type de Java
- d'hériter des méthodes de la classe Object

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

public class TestLambda {

  public static void main(String[] args) {
    Runnable monTraitement = () -> {
      System.out.println("mon traitement");
    };
    Object obj = monTraitement;
  }
}
```

Attention cependant, une expression lambda ne possède pas forcément d'identité unique : la sémantique de la méthode equals() n'est donc pas garantie.

Le type dans la déclaration, désigné par target type, est le type de l'interface fonctionnelle dans le contexte auquel l'expression lambda sera utilisée.

Une expression lambda sera transformée par le compilateur en une instance du type de son interface fonctionnelle selon le contexte dans lequel elle est définie :

- soit celle du type à laquelle l'expression est assignée
- soit celle du type du paramètre à laquelle l'expression est passée

L'expression lambda ne contient pas elle-même d'information sur l'interface fonctionnelle qu'elle implémente. Cette interface sera déduite par le compilateur en fonction de son contexte d'utilisation.

Le type de l'interface fonctionnelle est déterminé par le compilateur en fonction du contexte de son utilisation. Deux expressions lambda syntaxiquement identiques peuvent donc être compatibles avec plusieurs interfaces fonctionnelles et peuvent donc être compilées en deux objets de type différents. Une même expression lambda peut donc être assignée à plusieurs interfaces fonctionnelles tant qu'elle respecte leur contrat.

Exemple (code Java 8) :

```
    LongFunction longFunction = x -> x * 2;
    IntFunction  intFunction  = x -> x * 2;
    Callable<String> monCallable = () -> "Mon traitement";
    Supplier<String> monSupplier = () -> "Mon traitement";
```

Le compilateur associe une expression lambda à une interface fonctionnelle et comme une interface fonctionnelle ne peut avoir qu'une seule méthode abstraite :

- les types des paramètres doivent correspondre à ceux des paramètres de la méthode
- le type de retour du corps de l'expression doit correspondre à celui de la méthode
- toutes les exceptions levées dans le corps de l'expression doivent être compatibles avec les exceptions déclarées dans la clause throws de la méthode

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

public class Calculatrice {

  @FunctionalInterface
  interface OperationEntiere {
    long effectuer(int a, int b);
  }

  public long calculer(int a, int b, OperationEntiere operation) {
    return operation.effectuer(a, b);
  }

  public static void main(String[] args) {
    Calculatrice calc = new Calculatrice();
    OperationEntiere addition = (a, b) -> a + b;
    OperationEntiere soustraction = (a, b) -> a - b;

    System.out.println(calc.calculer(10, 5, addition));
    System.out.println(calc.calculer(10, 5, soustraction));
  }
}
```

Résultat :

```
15
5
```

Il est possible d'imbriquer des expressions lambda mais dans ce cas le code devient moins lisible.

12.3. Les références de méthodes

Les références de méthodes permettent d'offrir une syntaxe simplifiée pour invoquer une méthode comme une expression lambda : elles offrent un raccourci syntaxique pour créer une expression lambda dont le but est d'invoquer une méthode ou un constructeur.

Une expression lambda correspond à une méthode anonyme dont le type est défini par une interface fonctionnelle. Les références de méthodes ont un rôle similaire mais au lieu de fournir une implémentation, une référence de méthode permet d'invoquer une méthode statique ou non ou un constructeur.

Les références de méthodes ou de constructeurs utilisent le nouvel opérateur ::.

Il existe quatre types de références de méthodes :

Type	Syntaxe	Exemple
Référence à une méthode statique	nomClasse::nomMethodeStatique	String::valueOf
Référence à une méthode sur une instance	objet::nomMethode	personne::toString
Référence à une méthode d'un objet arbitraire d'un type donné	nomClasse::nomMethode	Object::toString
Référence à un constructeur	nomClasse::new	Personne::new

Il y a deux possibilités pour invoquer une méthode d'une instance :

- préciser directement l'instance concernée dans la référence de méthodes
- préciser le type concernée dans la référence de méthodes : dans ce cas, l'instance sera passée en paramètre pour désigner celle qui sera invoquée

La syntaxe d'une référence de méthode est composée de trois éléments :

- un qualificateur qui précise le nom d'une classe ou d'une instance sur lequel la méthode sera invoquée
- l'opérateur ::
- un identifiant qui précise le nom de la méthode ou l'opérateur new pour désigner un constructeur

Le qualificateur peut être :

- un type pour les méthodes statiques et les constructeurs
- un type ou une expression pour les méthodes non statiques. L'expression doit préciser l'objet sur lequel la méthode est invoquée

Une référence de constructeur comprend :

- un qualificateur qui est un type dont il est possible de créer une instance : cela exclut les interfaces et les classes abstraites
- l'opérateur ::
- le mot clé new

Il n'est pas rare que le corps d'une expression lambda se résume à invoquer une méthode qui contient les traitements à exécuter. Cette technique est d'ailleurs utile pour déboguer le contenu de l'expression lambda car il est plus facile de mettre un point d'arrêt dans le corps d'une méthode que dans une expression lambda.

Exemple :

```
monButton.setOnAction(event -> System.out.println(event));
```

Comme le corps de l'expression lambda ne contient que l'invocation d'une méthode, il est possible de remplacer l'expression lambda par une référence de méthode.

Exemple :

```
monButton.setOnAction(System.out::println);
```

Le tableau ci-dessous donne quelques exemples de références de méthodes et leurs expressions lambda équivalentes :

Type	Référence de méthode	Expression lambda
Référence à une méthode statique	System.out::println Math::pow	x -> System.out.println(x) (x, y) -> Math.pow(x, y)
Référence à une méthode sur une instance	monObject::maMethode	x -> monObject.maMethode(x)
Référence à une méthode d'un objet arbitraire d'un type donné	String::compareToIgnoreCase	(x, y) -> x.compareToIgnoreCase(y)
Référence à un constructeur	MaClasse::new	() -> new MaClasse();

Dans le cas d'une référence à une méthode statique ou d'une référence à une méthode non statique sur une instance, les paramètres définis dans l'interface fonctionnelle sont simplement passés en paramètres de la méthode invoquée.

Dans le cas d'une référence à une méthode non statique sur une classe, le premier paramètre est l'instance sur laquelle la méthode sera invoquée avec les autres paramètres qui lui sont passés.

La méthode ou le constructeur qui sera invoqué est déterminé par le compilateur selon le contexte d'utilisation. Une méthode ou un constructeur peut avoir plusieurs surcharges. Le compilateur s'appuie sur l'interface fonctionnelle utilisée dans le contexte : la surcharge invoquée sera celle dont les paramètres correspondent à ceux définis dans l'unique méthode abstraite de l'interface fonctionnelle.

12.3.1. La référence à une méthode statique

La référence à une méthode statique permet d'invoquer une méthode statique d'une classe.

La syntaxe d'utilisation est :

nom_de_la_classe::nom_de_la_methode_statique

L'exemple ci-dessous invoque la méthode statique de trois manières : la version historique en utilisant une classe anonyme interne et les deux possibilités offertes par Java 8 c'est à dire une expression lambda et une référence de méthode. Les trois invocations sont rigoureusement identiques, seule la quantité de code nécessaire pour les réaliser change.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.methode_reference;

public class TestMethodeReference {

  public static void main(String[] args) {

    // classe anonyme interne
    new Thread(new Runnable() {
      @Override
      public void run() {
        executer();
      }
    }).start();

    // expression lambda
    new Thread(() -> executer()).start();

    // référence de méthode statique
    new Thread(TestMethodeReference::executer).start();
  }

  static void executer() {
    System.out.println("execution de mon traitement par "+Thread.currentThread().getName());
  }
}
```

Résultat :

```
execution de mon traitement par Thread-0
execution de mon traitement par Thread-1
execution de mon traitement par Thread-2
```

L'exemple ci-dessous définit une interface fonctionnelle qui est utilisée en paramètre d'une méthode.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.Arrays;
import java.util.List;

public class TestMethodeReference {

  public static void main(String[] args) {
    List<String> fruits = Arrays.asList("melon", "abricot", "fraise", "cerise");
    afficher(fruits, (fmt, arg) -> String.format(fmt, arg));
  }

  public static void afficher(List<String> liste, MonFormateur formateur) {
    int i = 0;
    for (String element : liste) {
      i++;
      System.out.print(formateur.formater("%3d %s%n", i, element));
    }
  }
}

@FunctionalInterface
interface MonFormateur {
  String formater(String format, Object... arguments);
}
```

Résultat :

```
  1 melon
  2 abricot
  3 fraise
  4 cerise
```

A la place de l'expression lambda, il est possible d'utiliser directement une référence de méthodes static sur format() de la classe String.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.Arrays;
import java.util.List;

public class TestMethodeReference {

  public static void main(String[] args) {
    List<String> fruits = Arrays.asList("melon", "abricot", "fraise", "cerise");
    afficher(fruits, String::format);
  }

  public static void afficher(List<String> liste, MonFormateur formateur) {
    int i = 0;
    for (String element : liste) {
      i++;
      System.out.print(formateur.formater("%3d %s%n", i, element));
    }
  }
}

@FunctionalInterface
interface MonFormateur {

  String formater(String format, Object... arguments);
}
```

L'exemple ci-dessous utilise une référence à la méthode statique compare() de la classe Integer pour trier un tableau en invoquant la méthode sort() de la classe Arrays. Cette méthode attend en paramètre le tableau à trier et une interface fonctionnelle de type Comparator.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.methode_reference;

import java.util.Arrays;

public class TestReferenceMethodeStatique {

  public static void main(String[] args) {
    Integer[] valeurs = {10, 4, 2, 7, 5, 8, 1, 9, 3, 6};
    Arrays.sort(valeurs, Integer::compare);
    System.out.println(Arrays.deepToString(valeurs));
  }
}
```

Résultat :

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Attention, il n'est pas possible d'appliquer cette fonctionnalité si le tableau est un tableau d'entiers de type primitif.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.methode_reference;

import java.util.Arrays;

public class TestReferenceMethodeStatique {

  public static void main(String[] args) {
    int[] valeurs = {10, 4, 2, 7, 5, 8, 1, 9, 3, 6};
    Arrays.sort(valeurs, Integer::compare);
    System.out.println(Arrays.deepToString(valeurs));
  }
}
```

Résultat :

```
C:\java\TestJava8\src>jav ac -cp . com/jmdoudoux/dej/java8/methode_reference/Tes
tReferenceMethodeStatique.java
com\jmdoudoux\dej\java8\methode_reference\TestReferenceMethodeStatique.java:9: e
rror: no suitable method found for sort(int[],Integer::compare)
    Arrays.sort(valeurs, Integer::compare);
          ^
    method Arrays.<T#1>sort(T#1[],Comparator<? super T#1>) is not applicable
      (inference variable T#1 has incompatible bounds
        equality constraints: int
        upper bounds: Integer,Object)
    method Arrays.<T#2>sort(T#2[],int,int,Comparator<? super T#2>) is not applic
able
      (cannot infer type-variable(s) T#2
        (actual and formal argument lists differ in length))
  where T#1,T#2 are type-variables:
    T#1 extends Object declared in method <T#1>sort(T#1[],Comparator<? super T#1
>)
    T#2 extends Object declared in method <T#2>sort(T#2[],int,int,Comparator<? s
uper T#2>)
com\jmdoudoux\dej\java8\methode_reference\TestReferenceMethodeStatique.java:10:
error: incompatible types: int[] cannot be converted to Object[]
    System.out.println(Arrays.deepToString(valeurs));
                                           ^
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get
full output
2 errors
```

La signature de la méthode Compare() de la classe Integer est compatible avec l'interface fonctionnelle Comparator. Il est donc possible d'utiliser une référence statique sur cette méthode en paramètre de la méthode sort() de la classe Arrays.

De la même façon, il est possible d'utiliser une référence de méthode sur une méthode statique définie dans une classe de l'application tant que sa signature respecte celle de l'interface fonctionnelle.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.methode_reference;

import java.util.Arrays;

public class TestReferenceMethodeStatique {

  public static void main(String[] args) {
    Integer[] valeurs = {10, 4, 2, 7, 5, 8, 1, 9, 3, 6};
    Arrays.sort(valeurs, TestReferenceMethodeStatique::comparerEntierAscendant);
    System.out.println(Arrays.deepToString(valeurs));
  }

  public static int comparerEntierAscendant(int a, int b) {
    return a - b;
  }
}
```

12.3.2. La référence à une méthode d'une instance

La référence d'une méthode d'instance permet d'offrir un raccourci syntaxique pour invoquer une méthode d'un objet.

La syntaxe est de la forme :

instance::nom_de_la_methode

Où :

- instance est l'objet sur lequel la méthode est invoquée
- nom_de_la_methode est le nom de la méthode à invoquer

Dans l'exemple ci-dessous, la classe ComparaisonPersonne propose deux méthodes pour comparer deux objets de type Personne selon deux critères.

Exemple :

```
package fr.jmdoudoux.dej.java8.methode_reference;

import fr.jmdoudoux.dej.java8.lambda.Personne;

public class ComparaisonPersonne {

  public int comparerParNom(Personne p1, Personne p2) {
    return p1.getNom().compareTo(p2.getNom());
  }

  public int comparerParPrenom(Personne p1, Personne p2) {
    return p1.getPrenom().compareTo(p2.getPrenom());
  }
}
```

La classe ci-dessous utilise une référence à une méthode d'une instance de type ComparaisonPersonne en paramètre de la méthode sort() de la classe Arrays pour trier un tableau de type Personne.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.methode_reference;

import fr.jmdoudoux.dej.java8.lambda.Personne;
import java.util.Arrays;

public class TestReferenceMethodeInstance {

  public static void main(String[] args) {
    Personne[] personnes = {
      new Personne("nom3", "Julien"),
      new Personne("nom1", "Thierry"),
      new Personne("nom2", "Alain")
    };
    ComparaisonPersonne comparaisonPersonne = new ComparaisonPersonne();

    Arrays.sort(personnes, comparaisonPersonne::comparerParNom);
    System.out.println(Arrays.deepToString(personnes));

    Arrays.sort(personnes, comparaisonPersonne::comparerParPrenom);
    System.out.println(Arrays.deepToString(personnes));
  }
}
```

Résultat :

```
[Personne{nom=nom1, prenom=Thierry}, Personne{nom=nom2, prenom=Alain}, Personne{nom=nom3,
prenom=Julien}]
[Personne{nom=nom2, prenom=Alain}, Personne{nom=nom3, prenom=Julien}, Personne{nom=nom1,
prenom=Thierry}]
```

La classe de l'instance peut utiliser des generic tant que les types restent compatibles avec le contexte d'utilisation.

Exemple :

```
package fr.jmdoudoux.dej.java8.methode_reference;

import fr.jmdoudoux.dej.java8.lambda.Personne;

public class ComparaisonPersonne<T extends Personne> {

  public int comparerParNom(T p1, T p2) {
    return p1.getNom().compareTo(p2.getNom());
  }

  public int comparerParPrenom(T p1, T p2) {
    return p1.getPrenom().compareTo(p2.getPrenom());
  }
}
```

Le résultat est le même.

Le qualificateur de l'instance peut être le mot clé this pour désigner l'instance courante ou super pour désigner une référence sur la classe mère.

Ainsi this::equals est équivalent à l'expression lambda x -> this.equals(x).

12.3.3. La référence à une méthode d'une instance arbitraire d'un type

Une référence à une méthode d'instance sur un objet arbitraire d'un type permet d'invoquer une méthode d'une instance qui est précisée dans le premier paramètre fourni.

La syntaxe est de la forme :

nom_de_la_classe::nom_de_la_methode_d_instance

Où :

- nom_de_la_classe est le type de l'instance
- nom_de_la_methode_d_instance est le nom de la méthode à invoquer

Ce type de référence de méthodes ne précise pas explicitement le récepteur sur lequel la méthode sera invoquée. C'est le premier paramètre de la méthode de l'interface fonctionnelle qui correspond à ce récepteur.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.methode_reference;

import java.util.Arrays;

public class TestReferenceMethodeUnbound {

  public static void main(String[] args) {
    String[] fruits = {"Melon", "abricot", "fraise", "cerise", "mytille"};
    Arrays.sort(fruits, String::compareToIgnoreCase);
    System.out.println(Arrays.deepToString(fruits));
  }
}
```

Résultat :

```
[abricot, cerise, fraise, Melon, mytille]
```

L'exemple ci-dessous est le même exemple dans lequel l'utilisation d'une référence de méthode est remplacée par une expression lambda.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.methode_reference;

import java.util.Arrays;

public class TestReferenceMethodeUnbound {

  public static void main(String[] args) {
    String[] fruits = {"Melon", "abricot", "fraise", "cerise", "mytillee"};
    Arrays.sort(fruits, (s1, s2) -> s1.compareToIgnoreCase(s2) );
    System.out.println(Arrays.deepToString(fruits));
  }
}
```

Le résultat est le même.

L'expression lambda correspondante à la référence de méthode de l'exemple ci-dessus attend deux paramètres de type String. Le premier sera l'instance sur laquelle la méthode est invoquée. Le second est passé en paramètre de la méthode.

Le corps de l'expression invoque la méthode compareToIgnoreCase() sur l'instance du premier paramètre en lui passant en paramètre le second.

L'exemple ci-dessous est le même exemple dans lequel l'utilisation d'une référence de méthode est remplacée par une classe anonyme interne.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.methode_reference;

import java.util.Arrays;
import java.util.Comparator;

public class TestReferenceMethodeUnbound {

  public static void main(String[] args) {
    String[] fruits = {"Melon", "abricot", "fraise", "cerise", "mytille"};
    Arrays.sort(fruits, new Comparator<String>() {
      @Override
      public int compare(String s1, String s2) {
        return s1.compareToIgnoreCase(s2);
      }
    });
    System.out.println(Arrays.deepToString(fruits));
  }
}
```

Le résultat est le même.

12.3.4. La référence à un constructeur

Il est possible de faire référence à un constructeur.

La syntaxe d'une référence à un constructeur est de la forme :

nom_classe::new

Il est inutile de préciser la surcharge du constructeur qui sera invoquée : le compilateur la détermine selon le contexte. La surcharge utilisée sera celle dont les paramètres correspondent le mieux à ceux de la méthode de l'interface fonctionnelle.

Une référence à un constructeur peut être passée en paramètre ou assignée à une variable d'un type d'une interface fonctionnelle.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.Supplier;

public class TestConstructeurReference {

  public static void main(String[] args) {
    Supplier<Personne> supplier = Personne::new;
    System.out.println(supplier.get());
  }
}
```

Cet exemple est équivalent à celui ci-dessous qui utilise une expression lambda.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.Supplier;

public class TestConstructeurReference {

  public static void main(String[] args) {
     Supplier<Personne> supplier = () -> new Personne();
     System.out.println(supplier.get());
  }
}
```

Le tableau ci-dessous contient quelques exemples de références à un constructeur et leur équivalent sous la forme d'une expression lambda.

Référence à un constructeur	Expression lambda
Integer::new	(int valeur) -> new Integer(valeur) ou (String s) -> new Integer(s)
ArrayList<Personne>::new	() -> new ArrayList<Personne>()
String[]::new	(int size) -> new String[size]

Il est possible d'invoquer un constructeur possédant des paramètres : il faut pour cela que la méthode de l'interface fonctionnelle possède les paramètres qui correspondent à ceux du constructeur invoqué.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

@FunctionalInterface
public interface PersonneSupplier {
  Personne creerInstance(String nom, String prenom);
}
```

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.Supplier;

public class TestConstructeurReference {

  public static void main(String[] args) {
    PersonneSupplier supplier = Personne::new;
    Personne personne = supplier.creerInstance("nom1", "prenom1");
    System.out.println(personne);
  }
}
```

Résultat :

```
Personne{nom=nom1, prenom=prenom1}
```

Le constructeur qui sera invoqué dépend du contexte d'exécution : le compilateur va inférer les types des paramètres définis dans la méthode de l'interface fonctionnelle pour rechercher le constructeur possédant les mêmes types de paramètres.

Cet exemple est équivalent à celui ci-dessous qui utilise une expression lambda.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.Supplier;

public class TestConstructeurReference {

  public static void main(String[] args) {
    PersonneSupplier supplier = (nom, prenom) -> new Personne(nom, prenom);
    Personne personne = supplier.creerInstance("nom1", "prenom1");
    System.out.println(personne);
  }
}
```

Il est possible de préciser le ou les types si la classe est typée avec un generic.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.ArrayList;
import java.util.List;

public class TestConstructeurReference {

  public static void main(String[] args) {
      MaFabrique<List<String>> fabrique = ArrayList<String>::new;
   }
}

interface MaFabrique<T> {
  T creerInstance();
}
```

Il est possible d'utiliser une référence de constructeur pour un tableau.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

public class TestConstructeurReference {

  public static void main(String[] args) {
    MaFabrique<Integer[]> fabrique = Integer[]::new;
    Integer[] entiers = fabrique.creerInstance(10);
    System.out.println("taille = "+entiers.length);
  }
}

interface MaFabrique<T> {
  T creerInstance(int taille);
}
```

Résultat :

```
taille = 10
```

12.4. Les interfaces fonctionnelles

Une interface fonctionnelle (functional interface) est basiquement une interface dans laquelle une seule méthode abstraite est définie. Elle doit respecter certaines contraintes :

- elle ne doit avoir qu'une seule méthode déclarée abstraite
- les méthodes définies dans la classe Object ne sont pas prises en compte comme étant des méthodes abstraites
- toutes les méthodes doivent être public
- elle peut avoir des méthodes par défaut et static

Avant Java 8 un certain nombre d'interfaces ne définissaient qu'une seule méthode : ces interfaces sont désignées par le terme Single Abstract Method (SAM).

Avant Java 8, le JDK contenait déjà de nombreuses interfaces qui respectaient ces règles et sont donc des interfaces fonctionnelles, par exemple :

- Comparator<T> qui définit la méthode int compare(T o1, T o2)
- Callable<V> qui définit la méthode V call() throws exception
- Runnable qui définit la méthode void run()
- ActionListener qui définit la méthode void actionPerformed(ActionEvent)
- ...

Elles se prêtent bien à l'utilisation d'une classe anonyme interne. Si elles ne sont pas modifiées alors elles sont utilisables comme interfaces fonctionnelles même si elles ne sont pas toutes annotées avec @FunctionalInterface puisque ce type de méthode respecte le contrat des interfaces fonctionnelles. Il est donc possible d'utiliser une expression lambda à la place d'une classe anonyme.

Exemple (code Java 8) :

```
Consumer<String> afficher = (message) -> { System.out.println(message) };
BiConsumer<Integer, Integer> additionner = (x, y) -> x + y;
BiFunction<Integer, Integer, Long> additionner = (x, y) -> (long) x + y;
```

Une interface fonctionnelle ne définit qu'une seule méthode abstraite. Par exemple, l'interface Runnable qui ne définit que la méthode void run() est une interface fonctionnelle.

A partir de Java 8, il est possible d'utiliser une expression lambda à la place d'une classe anonyme. Une expression lambda peut être assignée explicitement ou implicitement à une interface fonctionnelle :

Exemple (code Java 8) :

```
Runnable monTraitement = () -> System.out.println("bonjour");
```

Si le type n'est pas explicitement précisé dans le code, c'est le compilateur qui va déterminer implicitement l'interface fonctionnelle correspondante.

Exemple (code Java 8) :

```
new Thread(() -> System.out.println("bonjour")).start();
```

Le compilateur va déduire l'interface fonctionnelle, dans ce cas l'interface Runnable.

Java 8 propose aussi en standard dans le JDK plusieurs interfaces fonctionnelles pour des besoins communs notamment dans le package java.util.function.

12.4.1. L'annotation @FunctionalInterface

Les interfaces fonctionnelles peuvent être annotées avec @FunctionalInterface : cette annotation permet de préciser l'intention que l'interface soit fonctionnelle.

L'utilisation de cette annotation est optionnelle mais elle apporte deux avantages :

- elle indique au compilateur que l'interface est fonctionnelle : celui-ci va pouvoir vérifier que toutes les règles soient respectées pour qu'elle soit effectivement fonctionnelle
- l'outil javadoc va utiliser l'annotation lors de la génération de la documentation

L'annotation @FunctionalInterface permet d'indiquer qu'une interface soit une interface fonctionnelle telle que définie dans la JLS. Elle ne peut être utilisée que sur une interface.

Exemple (code Java 8) :

```
@FunctionalInterface
public interface MonInterface {
  public void traiter();
}
```

L'annotation @FunctionalInterface permet d'indiquer explicitement au compilateur l'intention de conception que l'interface est une interface fonctionnelle : celui-ci pourra alors vérifier que les contraintes sont respectées. Son utilisation est facultative car le compilateur peut déterminer automatiquement si une interface correspond aux contraintes des interfaces fonctionnelles.

Si l'annotation est utilisée sur une classe, une énumération, une annotation ou sur une interface qui ne respecte pas les contraintes d'une interface fonctionnelle alors le compilateur émettra une erreur.

Certaines interfaces de type SAM existant avant Java 8 n'ont pas été annotées avec @FunctionalInterface car elles ne sont pas considérées comme étant des fonctions même si leur caractéristique fera qu'elles seront considérées comme des interfaces fonctionnelles par le compilateur. C'est par exemple le cas des interfaces Comparable et Closeable.

12.4.2. La définition d'une interface fonctionnelle

Une interface peut contenir la définition de différents types de méthodes :

- méthode abstraite
- redéclarer une méthode de la classe Object pour par exemple utiliser un commentaire Javadoc particulier
- méthode par défaut à partir de Java 8

Pour être une interface fonctionnelle, une interface ne doit avoir qu'une seule méthode abstraite déclarée. Elle peut avoir indifféremment aucune, une ou plusieurs redéfinitions de méthodes de la classe Object ou des méthodes par défaut.

Une interface fonctionnelle ne peut pas avoir plus d'une méthode abstraite. Si plusieurs méthodes abstraites sont définies dans l'interface, celle-ci ne sera pas une interface fonctionnelle.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

@FunctionalInterface
public interface MonInterfaceFonctionnelle {
  String executer();
  String effacer();
}
```

Résultat :

```
C:\java\TestJava8\src>javac -cp . com/jmdoudoux/dej/java8/lambda/MonInterfaceFonctionnelle.java
com\jmdoudoux\dej\java8\lambda\MonInterfaceFonctionnelle.java:3: error: Unexpect
ed @FunctionalInterface annotation
@FunctionalInterface
^
  MonInterfaceFonctionnelle is not a functional interface
    multiple non-overriding abstract methods found in interface MonInterfaceFonc
tionnelle
1 error
```

Les méthodes publiques de la classe Object peuvent cependant être redéfinies dans une interface fonctionnelle.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

@FunctionalInterface
public interface MonInterfaceFonctionnelle {
  String executer();
  boolean equals(Object obj);
}
```

Ci-dessous, bien que la méthode clone() soit déclarée dans la classe Object, l'interface fonctionnelle ne compile pas car la méthode clone() de la classe Object est déclarée protected et non public.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

@FunctionalInterface
public interface MonInterfaceFonctionnelle {
  String executer();
  Object clone();
}
```

Résultat :

```
C:\java\TestJava8\src>javac -cp . com/jmdoudoux/dej/java8/lambda/MonInterfaceFonctionnelle.java
com\jmdoudoux\dej\java8\lambda\MonInterfaceFonctionnelle.java:3: error: Unexpect
ed @FunctionalInterface annotation
@FunctionalInterface
^
  MonInterfaceFonctionnelle is not a functional interface
    multiple non-overriding abstract methods found in interface MonInterfaceFonc
```

```
tionnelle
1 error
```

Une interface fonctionnelle peut aussi contenir des méthodes statiques ou des méthodes par défaut.

12.4.3. L'utilisation d'une interface fonctionnelle

Les interfaces fonctionnelles définissent des types qui pourront être implémentés sous la forme d'expressions lambda ou de références de méthodes.

Chaque interface fonctionnelle ne possède qu'une seule méthode abstraite nommée méthode fonctionnelle (functional method) pour laquelle les paramètres et la valeur de retour doivent correspondre ou pouvoir être adaptés.

Une interface fonctionnelle définit une méthode qui pourra être utilisée pour passer en paramètre :

- une référence sur une méthode d'une instance
- une référence sur une méthode statique
- une référence sur un constructeur
- une expression lambda
- une classe anonyme interne

Une interface fonctionnelle définit un type qui peut être utilisé dans plusieurs situations :

- assignation

Exemple (code Java 8) :

```
    Predicate<String> estVide = String::isEmpty;
```

- directement en paramètre d'une méthode

Exemple (code Java 8) :

```
monStream.filter(e -> e.getTaille() > 100)
```

- cast

Exemple (code Java 8) :

```
monStream.map((ToIntFunction) e -> e.getTaille())
```

Une interface fonctionnelle est avant tout une interface : elle peut donc être utilisée comme telle dans le code.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

@FunctionalInterface
public interface MonInterfaceFonctionnelle {
  void executer();
}
```

Exemple :

```
package fr.jmdoudoux.dej.java8.lambda;

public class TestMonInterface {
  public static void executer(MonInterfaceFonctionnelle monInterface) {
    monInterface.executer();
  }

  public static void main(String[] args) {
    executer(new MonInterfaceFonctionnelle() {

      @Override
      public void executer() {
        System.out.println("test");
      }
    });
  }
}
```

Il est aussi possible de profiter de la simplicité de la syntaxe d'une expression lambda puisqu'une expression lambda peut être utilisée partout où un objet de type interface fonctionnelle est attendu.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

public class TestMonInterface {

  public static void executer(MonInterfaceFonctionnelle monInterface) {
    monInterface.executer();
  }

  public static void main(String[] args) {
    executer(() -> System.out.println("test"));
  }
}
```

L'exemple ci-dessous tri un tableau de chaînes de caractères en utilisant la méthode sort() de la classe Arrays. Elle attend en paramètre le tableau à trier et une instance de type Comparator. Comme l'interface Comparator est une interface fonctionnelle, il est possible d'utiliser une expression lambda.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.Arrays;

public class TestLambda {

  public static void main(String[] args) {

    String[] elements = new String[] { "aaa","zzz","fff","mmm"};
    Arrays.sort(elements, (o1, o2) -> o1.compareTo(o2));
    System.out.println(Arrays.toString(elements));
  }
}
```

Résultat :

```
[aaa, fff, mmm, zzz]
```

C'est le compilateur qui se charge d'effectuer toutes les opérations requises.

Il n'est pas possible d'assigner une expression lambda à un objet de type Object.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;
```

```
import java.util.Arrays;

public class TestLambda {

  public static void main(String[] args) {

    Object tri = (o1, o2) -> o1.compareTo(o2);
  }
}
```

Résultat :

```
C:\java\TestJava8\src>javac -cp . com/jmdoudoux/dej/java8/lambda/TestLambda.java
com\jmdoudoux\dej\java8\lambda\TestLambda.java:9: error: incompatible types: Obj
ect is not a functional interface
    Object tri = (o1, o2) -> o1.compareTo(o2);
                 ^
1 error
```

Cette assignation n'est pas légale car le compilateur ne peut pas déterminer la méthode qui devra être invoquée puisque Object n'est pas une interface fonctionnelle.

Par contre, il est possible d'assigner à un objet d'un type d'une interface fonctionnelle une expression lambda qui respecte la déclaration de la méthode abstraite.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.Arrays;
import java.util.Comparator;

public class TestLambda {

  public static void main(String[] args) {

    Comparator<String> tri = (o1, o2) -> o1.compareTo(o2);
    String[] elements = new String[] { "aaa","zzz","fff","mmm"};
    Arrays.sort(elements, tri);
    System.out.println(Arrays.toString(elements));
}
```

Il est aussi impératif de préciser le type generic de l'interface fonctionnelle pour permettre au compilateur d'inférer le type des paramètres de l'expression lambda et de vérifier que ce type possède bien une méthode compareTo().

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.Comparator;

public class TestLambda {

  public static void main(String[] args) {
    Comparator tri = (o1, o2) -> o1.compareTo(o2);
  }
}
```

Résultat :

```
C:\java\TestJava8\src>javac -cp . com/jmdoudoux/dej/java8/lambda/TestLambda.java
com\jmdoudoux\dej\java8\lambda\TestLambda.java:10: error: cannot find symbol
    Comparator tri = (o1, o2) -> o1.compareTo(o2);
                                   ^
  symbol:   method compareTo(Object)
  location: variable o1 of type Object
```

```
Note: com\jmdoudoux\dej\java8\lambda\TestLambda.java uses unchecked or unsafe op
erations.
Note: Recompile with -Xlint:unchecked for details.
1 error
```

Si une exception de type checked est levée dans le corps de l'expression lambda sans être gérée alors il est nécessaire que la méthode de l'interface fonctionnelle correspondante déclare la levée de cette exception sinon une erreur est émise à la compilation.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

public class TestLambda {

  public static void main(String[] args) {
    Runnable monTraitement = () -> {
      System.out.println("debut");
      Thread.sleep(1000);
      System.out.println("fin");
    };
  }
}
```

Résultat :

```
C:\java\TestJava8\src>javac -cp . com/jmdoudoux/dej/java8/lambda/TestLambda.java
com\jmdoudoux\dej\java8\lambda\TestLambda.java:14: error: unreported exception I
nterruptedException; must be caught or declared to be thrown
      Thread.sleep(1000);
                  ^
1 error
```

La méthode run() de l'interface Runnable ne déclare pas pouvoir lever une exception.

Pour résoudre ce problème, il y a plusieurs solutions :

- ne pas lever d'exception
- gérer l'exception dans le corps de l'expression lambda en utilisant un bloc try/catch
- déclarer que la méthode de l'interface fonctionnelle peut lever une exception (impossible dans le cas de l'interface Runnable)
- utiliser une autre interface fonctionnelle : dans le cas ci-dessous, Callable.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.concurrent.Callable;

public class TestLambda {

  public static void main(String[] args) {
    Callable monTraitement = () -> {
      System.out.println("debut");
      Thread.sleep(1000);
      System.out.println("fin");
      return null;
    };
  }
}
```

12.4.4. Les interfaces fonctionnelles du package java.util.function

Le package java.util.function propose en standard des interfaces fonctionnelles d'usage courant. Toutes les interfaces de ce package sont annotées avec @FunctionalInterface.

Le nom des interfaces du package java.util.function respecte une convention de nommage selon leur rôle :

- Function : une fonction unaire qui permet de réaliser une transformation. Elle attend un ou plusieurs paramètres et renvoie une valeur. La méthode se nomme apply()
- Consumer : une fonction qui permet de réaliser une action. Elle ne renvoie pas de valeur et attend un ou plusieurs paramètres. La méthode se nomme accept()
- Predicate : une fonction qui attend un ou plusieurs paramètres et renvoie un booléen. La méthode se nomme test()
- Supplier : une fonction qui renvoie une instance. Elle n'attend pas de paramètre et renvoie une valeur. La méthode se nomme get(). Elle peut être utilisé comme une fabrique

Le nom des interfaces fonctionnelles qui attendent en paramètre une ou plusieurs valeurs primitives sont préfixées par le type primitif.

Le nom des interfaces fonctionnelles qui renvoient une valeur primitive sont suffixées par toXXX.

Interface fonctionnelle	Description
BiConsumer<T,U>	Représente une opération qui requiert deux objets et ne renvoie aucun résultat
BiFunction<T,U,R>	Représente une opération qui requiert deux objets de type T et U et renvoie un résultat de type R
BinaryOperator<T>	Représente une opération qui attend deux paramètres de type T et renvoie une instance de type T
BiPredicate<T,U>	Représente un prédicat qui attend deux paramètres et renvoie un booléen
BooleanSupplier	Représente un fournisseur d'une valeur booléenne qui n'attend aucun paramètre
Consumer<T>	Représente un consommateur d'un unique paramètre qui ne renvoie aucune valeur
DoubleBinaryOperator	Représente une opération qui attend en paramètre deux valeurs de type double et renvoie une valeur de type double
DoubleConsumer	Représente un consommateur d'une valeur de type double
DoubleFunction<R>	Représente une fonction qui attend en paramètre une valeur de type double et renvoie un résultat de type R
DoublePredicate	Représente un prédicat qui attend en paramètre un argument de type double et renvoie un booléen
DoubleSupplier	Représente un fournisseur d'une valeur de type double
DoubleToIntFunction	Représente une opération qui attend en paramètre un double et renvoie un int comme résultat
DoubleToLongFunction	Représente une opération qui attend en paramètre un double et renvoie un long comme résultat
DoubleUnaryOperator	Représente une opération qui attend en paramètre un double et renvoie un double comme résultat
Function<T,R>	Représente une fonction qui attend un paramètre de type T et renvoie un résultat de type R
IntBinaryOperator	Représente une opération qui attend en paramètres deux valeurs de type int et renvoie une valeur de type int
IntConsumer	Représente un consommateur d'une valeur de type int
IntFunction<R>	Représente une fonction qui attend en paramètre une valeur de type int et renvoie un résultat de type R

IntPredicate	Représente un prédicat qui attend en paramètre un argument de type int et renvoie un booléen
IntSupplier	Représente un fournisseur de valeur qui n'attend aucun paramètre et renvoie un entier de type int
IntToDoubleFunction	Représente une fonction qui attend en paramètre une valeur de type int et renvoie un double
IntToLongFunction	Représente une fonction qui attend en paramètre une valeur de type int et renvoie un long
IntUnaryOperator	Représente une opération qui attend en paramètre un int et renvoie un int comme résultat
LongBinaryOperator	Représente une opération qui attend en paramètres deux valeurs de type long et renvoie une valeur de type long
LongConsumer	Représente un consommateur d'une valeur de type long
LongFunction<R>	Représente une fonction qui attend en paramètre une valeur de type long et renvoie un résultat de type R
LongPredicate	Représente un prédicat qui attend en paramètre un argument de type long et renvoie un booléen
LongSupplier	Représente un fournisseur de valeur qui n'attend aucun paramètre et renvoie un entier de type long
LongToDoubleFunction	Représente une fonction qui attend en paramètre une valeur de type long et renvoie un double
LongToIntFunction	Représente une fonction qui attend en paramètre une valeur de type long et renvoie un int
LongUnaryOperator	Représente une opération qui attend en paramètre un objet de type long et renvoie une valeur de type long
ObjDoubleConsumer<T>	Représente un consommateur qui attend en paramètres un objet de type T et un double
ObjIntConsumer<T>	Représente un consommateur qui attend en paramètres un objet de type T et un int
ObjLongConsumer<T>	Représente un consommateur qui attend en paramètres un objet de type T et un long
Predicate<T>	Représente un prédicat qui attend en paramètre un argument de type T et renvoie un booléen
Supplier<T>	Représente un fournisseur de valeur qui n'attend aucun paramètre et renvoie une instance de type T
ToDoubleBiFunction<T,U>	Représente une fonction qui attend deux paramètres de type T et U et renvoie un double
ToDoubleFunction<T>	Représente une fonction qui attend un paramètre de type T et renvoie un double
ToIntBiFunction<T,U>	Représente une fonction qui attend deux paramètres de type T et U et renvoie un int
ToIntFunction<T>	Représente une fonction qui attend un paramètre de type T et renvoie un int
ToLongBiFunction<T,U>	Représente une fonction qui attend un paramètre de type T et renvoie un long
ToLongFunction<T>	Représente une fonction qui attend un paramètre de type T et renvoie un long
UnaryOperator<T>	Représente une opération qui attend en paramètre un objet de type T et renvoie une instance de type T. Elle hérite de Function<T, T>

Ces interfaces fonctionnelles couvrent de nombreux besoins courants mais il est aussi possible de définir ses propres interfaces fonctionnelles.

12.4.4.1. Les interfaces fonctionnelles de types Consumer

Les interfaces fonctionnelles de type Consumer (Consumer, BiConsumer, DoubleConsumer, IntConsumer, LongConsumer, ObjDoubleConsumer, ObjIntConsumer, ObjLongConsumer) définissent des fonctions qui attendent différents types de paramètres et ne renvoient aucune valeur. Ne renvoyant aucune valeur, elles peuvent induire des effets de bords lors de l'exécution de leur traitement.

L'interface fonctionnelle Consumer<T> définit une fonction qui effectue une opération sur un objet et ne renvoie aucune valeur. Son exécution engendre généralement des effets de bord.

Elle définit la méthode fonctionnelle accept(T t) qui ne renvoie aucune valeur.

Elle définit aussi une méthode par défaut :

Méthode	Rôle
default Consumer andThen(Consumer< ? super T>	Renvoyer un Consumer qui exécute en séquence l'instance courante et celle fournie en paramètre

Exemple (code Java 8) :

```java
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.Consumer;

public class TestConsumer {

  public static void main(String[] args) {
    Consumer<String> c = System.out::print;
    c.andThen(c).accept("bonjour ");
  }
}
```

Résultat :

```
bonjour bonjour
```

Lors de l'invocation de la méthode andThen(), si une exception est levée par un des deux Consumer, celle-ci est propagée dans la méthode englobante.

Exemple (code Java 8) :

```java
package fr.jmdoudoux.dej.java8.lambda;

import java.util.concurrent.atomic.AtomicInteger;
import java.util.function.Consumer;

public class TestConsumer {

  public static void main(String[] args) {
    AtomicInteger i = new AtomicInteger(0);
    Consumer<String> c = (x) -> {
      i.addAndGet(1);
      System.out.println(x);
      if (i.get() == 2) {
        throw new RuntimeException();
      }
    };
    c.andThen(c).accept("bonjour");
  }
}
```

Résultat :

```
bonjour
bonjour
```

```
Exception in thread "main" java.lang.RuntimeException
       at fr.jmdoudoux.dej.java8.lambda.TestConsumer.lambda$main$0(TestConsumer.java:14)
       at fr.jmdoudoux.dej.java8.lambda.TestConsumer$$Lambda$1/2536472.accept(Unknown Source)
       at java.util.function.Consumer.lambda$andThen$14(Consumer.java:65)
       at java.util.function.Consumer$$Lambda$2/32404285.accept(Unknown Source)
       at fr.jmdoudoux.dej.java8.lambda.TestConsumer.main(TestConsumer.java:17)
Java Result: 1
```

Si une exception est levée lors de l'exécution du Consumer courant, le second Consumer n'est pas exécuté.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.Consumer;

public class TestConsumer {

  public static void main(String[] args) {
    Consumer<String> c = (x) -> {
      System.out.println(x);
      throw new RuntimeException();
    };
    c.andThen(c).accept("bonjour");
  }
}
```

Résultat :

```
bonjour
Exception in thread "main" java.lang.RuntimeException
       at fr.jmdoudoux.dej.java8.lambda.TestConsumer.lambda$main$0(TestConsumer.java:10)
       at fr.jmdoudoux.dej.java8.lambda.TestConsumer$$Lambda$1/2536472.accept(Unknown Source)
       at java.util.function.Consumer.lambda$andThen$14(Consumer.java:65)
       at java.util.function.Consumer$$Lambda$2/13604864.accept(Unknown Source)
       at fr.jmdoudoux.dej.java8.lambda.TestConsumer.main(TestConsumer.java:12)
```

Elle lève une exception de type NullPointerException si le paramètre de la méthode andThen() est null.

L'interface fonctionnelle BiConsumer définit une opération qui attend deux paramètres et ne renvoie aucun résultat. C'est une spécialisation de l'interface fonctionnelle Consumer qui attend deux paramètres.

L'interface BiConsumer<T,U> est typé avec deux generics qui précisent respectivement le type du premier et du second argument de l'opération de la fonction.

Elle définit la méthode fonctionnelle accept(T t, U u) qui ne renvoie aucune valeur.

Elle définit aussi une méthode par défaut :

Méthode	Rôle
default BiConsumer andThen(BiConsumer< ? super T>	Renvoyer un BiConsumer qui exécute en séquence l'instance courante et celle fournie en paramètre

L'interface fonctionnelle DoubleConsumer définit une opération qui attend en paramètre une valeur de type double et ne renvoie aucun résultat. C'est une spécialisation de l'interface fonctionnelle Consumer qui attend une valeur flottante.

Elle définit la méthode fonctionnelle accept(double valeur) qui ne renvoie aucune valeur.

Elle définit aussi une méthode par défaut :

Méthode	Rôle

default DoubleConsumer andThen(DoubleConsumer)	Renvoyer un DoubleConsumer qui exécute en séquence l'instance courante et celle fournie en paramètre

L'interface fonctionnelle IntConsumer définit une opération qui attend en paramètre une valeur de type int et ne renvoie aucun résultat. C'est une spécialisation de l'interface fonctionnelle Consumer qui attend une valeur entière.

Elle définit la méthode fonctionnelle accept(int valeur) qui ne renvoie aucune valeur.

Elle définit aussi une méthode par défaut :

Méthode	Rôle
default IntConsumer andThen(IntConsumer)	Renvoyer un IntConsumer qui exécute en séquence l'instance courante et celle fournie en paramètre

L'interface fonctionnelle LongConsumer définit une opération qui attend en paramètre une valeur de type long et ne renvoie aucun résultat. C'est une spécialisation de l'interface fonctionnelle Consumer qui attend une valeur entière longue.

Elle définit la méthode fonctionnelle accept(long valeur) qui ne renvoie aucune valeur.

Elle définit aussi une méthode par défaut :

Méthode	Rôle
default LongConsumer andThen(LongConsumer)	Renvoyer un LongConsumer qui exécute en séquence l'instance courante et celle fournie en paramètre

L'interface fonctionnelle ObjDoubleConsumer<T> est une spécialisation de l'interface BiConsumer pour un objet et une valeur de type double. Elle définit la méthode fonctionnelle accept(T t, double value) qui ne renvoie rien.

L'interface fonctionnelle ObjIntConsumer<T> est une spécialisation de l'interface BiConsumer pour un objet et une valeur de type int. Elle définit la méthode fonctionnelle accept(T t, int value) qui ne renvoie rien.

L'interface fonctionnelle ObjLongConsumer<T> est une spécialisation de l'interface BiConsumer pour un objet et une valeur de type long. Elle définit la méthode fonctionnelle accept(T t, long value) qui ne renvoie rien.

12.4.4.2. Les interfaces fonctionnelles de type Function

Les interfaces fonctionnelles de type Function (Function, BiFunction, DoubleFunction, DoubleToIntFunction, DoubleToLongFunction, IntFunction, IntToDoubleFunction, IntToLongFunction, LongFunction, LongToDoubleFunction, LongToIntFunction, ToDoubleBiFunction, ToDoubleFunction, ToIntBiFunction, ToIntFunction, ToLongBiFunction, ToLongFunction, BinaryOperator, DoubleBinaryOperator, DoubleUnaryOperator, IntBinaryOperator, IntUnaryOperator, LongBinaryOperator, LongUnaryOperator, UnaryOperator) définissent des fonctions qui attendent différents types de paramètres et renvoient une valeur.

L'interface fonctionnelle Function<T, R> définit une fonction qui effectue une opération sur un objet de type T et renvoie une valeur de type R.

Elle définit la méthode fonctionnelle apply(T t) qui renvoie une valeur de type R.

Elle définit aussi plusieurs méthodes par défaut et static :

Méthode	Rôle

default <V> Function<T, V> andThen(Function< ? super R, ? extends V >	Renvoyer une Function qui exécute en séquence l'instance courante et celle fournie en paramètre
default <V> Function <V, R> compose(Function< ? super V, ? extends T>	Renvoyer une Function qui exécute l'instance fournie en paramètre et applique l'instance courante sur le résultat
static <T> Function<T, T> identity()	Renvoyer une Function qui renvoie toujours la valeur fournie en paramètre

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.Function;

public class TestFunction {

  public static void main(String[] args) {
    Function<Integer,Long> doubler = (i) -> (long) i * 2;
    System.out.println(doubler.apply(2));
  }
}
```

Résultat :

```
4
```

Les méthodes andThen() et compose() permettent de mixer deux Function.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.Function;

public class TestFunction {

  public static void main(String[] args) {
    Function<Long, Long> doubler = (i) -> {
      long resultat = (long) i * 2;
      System.out.println("doubler=" + resultat);
      return resultat;
    };

    Function<Long, Long> laMoitie = (i) -> {
      long resultat = i / 2;
      System.out.println("laMoitie=" + resultat);
      return resultat;
    };

    System.out.println(doubler.andThen(laMoitie).apply(3L));
    System.out.println(doubler.compose(laMoitie).apply(3L));
  }
}
```

Résultat :

```
doubler=6
laMoitie=3
3
laMoitie=1
doubler=2
2
```

La méthode identity() renvoie une instance de Function dont le but est simplement de renvoyer la valeur fournie en paramètre.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.Function;

public class TestFunction {

  public static void main(String[] args) {
    Function<Long, Long> identite = Function.identity();
    System.out.println(identite.apply(3L));
  }
}
```

Résultat :

```
3
```

L'interface fonctionnelle BiFunction définit une opération qui attend deux paramètres et renvoie une valeur. C'est une spécialisation de l'interface fonctionnelle Function qui attend deux paramètres.

L'interface BiFunction<T,U,R> est typée avec trois generics qui précisent respectivement les types des deux arguments de l'opération de la fonction et le type de la valeur de retour.

Elle définit la méthode fonctionnelle accept(T t, U u) et renvoie une valeur de type R.

Elle définit aussi une méthode par défaut :

Méthode	Rôle
default <V> BiFunction<T, U, V> andThen(Function< ? super R, ? extends V>	Renvoyer une BiFunction qui exécute en séquence l'instance courante et celle fournie en paramètre

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.BiFunction;

public class TestBiFunction {

  public static void main(String[] args) {
    BiFunction<String, String, String> concatener = (x, y) -> x + y;
    System.out.println(concatener.apply("Bonjour", " Java"));
  }
}
```

Résultat :

```
Bonjour Java
```

L'interface fonctionnelle DoubleFunction<R> définit une opération qui attend en paramètre une valeur de type double et renvoie un résultat. C'est une spécialisation de l'interface fonctionnelle Function qui attend une valeur flottante.

L'interface DoubleFunction<R> est typée avec un generic qui précise le type de la valeur de retour.

Elle définit la méthode fonctionnelle apply(double valeur) qui renvoie une valeur de type R.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.DoubleFunction;
```

```
public class TestDoubleFunction {
  public static void main(String[] args) {
    DoubleFunction<String> formater = (x) -> String.format("%.2f", x);
    System.out.println(formater.apply(3.14116D));
  }
}
```

Résultat :

```
3,14
```

L'interface fonctionnelle DoubleToIntFunction définit une opération qui attend en paramètre une valeur de type double et renvoie un entier. C'est une spécialisation de l'interface fonctionnelle Function qui attend une valeur flottante et retourne un entier.

Elle définit la méthode fonctionnelle applyAsInt(double valeur) qui renvoie une valeur de type int.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.DoubleToIntFunction;

public class TestDoubleToIntFunction {
  public static void main(String[] args) {
    DoubleToIntFunction dtif = (x) -> {return (int) x;};
    System.out.println(dtif.applyAsInt(3.14));
  }
}
```

Résultat :

```
3
```

L'interface fonctionnelle DoubleToLongFunction définit une opération qui attend en paramètre une valeur de type double et renvoie un entier long. C'est une spécialisation de l'interface fonctionnelle Function qui attend une valeur flottante et retourne un entier long.

Elle définit la méthode fonctionnelle applyAsLong(double valeur) qui renvoie une valeur de type long.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.DoubleToLongFunction;

public class TestDoubleToLongFunction {

  public static void main(String[] args) {
    DoubleToLongFunction dtlf = (x) -> {return (long) x;};
    System.out.println(dtlf.applyAsLong(123456789012345.123D));
  }
}
```

Résultat :

```
123456789012345
```

L'interface fonctionnelle IntFunction définit une opération qui attend en paramètre une valeur de type int et renvoie un résultat. C'est une spécialisation de l'interface fonctionnelle Function qui attend une valeur entière.

L'interface IntFunction<R> est typée avec un generic qui précise le type de la valeur de retour.

Elle définit la méthode fonctionnelle apply(int valeur) qui renvoie une valeur de type R.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.IntFunction;

public class TestIntFunction {

  public static void main(String[] args) {
    IntFunction<String> formater = (x) -> String.format("%d m", x);
    System.out.println(formater.apply(3));
  }
}
```

Résultat :

```
3 m
```

L'interface fonctionnelle IntToDoubleFunction définit une opération qui attend en paramètre une valeur entière et renvoie une valeur flottante. C'est une spécialisation de l'interface fonctionnelle Function qui attend un entier et retourne une valeur flottante.

Elle définit la méthode fonctionnelle applyAsDouble(int valeur) qui renvoie une valeur de type double.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.IntToDoubleFunction;

public class TestIntToDoubleFunction {

  public static void main(String[] args) {
    IntToDoubleFunction cos = (x) -> Math.cos(x);
    System.out.println(cos.applyAsDouble(5));
  }
}
```

Résultat :

```
0.28366218546322625
```

L'interface fonctionnelle IntToLongFunction définit une opération qui attend en paramètre une valeur entière et renvoie une valeur entière longue. C'est une spécialisation de l'interface fonctionnelle Function qui attend une valeur entière et retourne un entier long.

Elle définit la méthode fonctionnelle applyAsLong(int valeur) qui renvoie une valeur de type long.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.IntToLongFunction;

public class TestIntToLongFunction {

  public static void main(String[] args) {
    IntToLongFunction doubler = (x) -> (long) x * 2;
    System.out.println(doubler.applyAsLong(Integer.MAX_VALUE));
  }
}
```

Résultat :

```
4294967294
```

L'interface fonctionnelle LongFunction définit une opération qui attend en paramètre une valeur de type long et renvoie un résultat. C'est une spécialisation de l'interface fonctionnelle Function qui attend une valeur entière longue.

L'interface LongFunction<R> est typée avec un generic qui précise le type de la valeur de retour.

Elle définit la méthode fonctionnelle apply(long valeur) qui renvoie une valeur de type R.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.LongFunction;

public class TestLongFunction {
  public static void main(String[] args) {
    LongFunction<String> formater = (x) -> String.format("%d m", x);
    System.out.println(formater.apply(123456789012345L));
  }
}
```

Résultat :

```
123456789012345 m
```

L'interface fonctionnelle LongToDoubleFunction définit une opération qui attend en paramètre une valeur entière longue et renvoie une valeur flottante. C'est une spécialisation de l'interface fonctionnelle Function qui attend une valeur entière longue et retourne une valeur flottante.

Elle définit la méthode fonctionnelle applyAsDouble(long valeur) qui renvoie une valeur de type double.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.LongToDoubleFunction;

public class TestLongToDoubleFunction {

  public static void main(String[] args) {
    LongToDoubleFunction sin = (x) -> Math.sin(x);
    System.out.println(sin.applyAsDouble(123456789012345L));
  }
}
```

Résultat :

```
-0.5986572942477425
```

L'interface fonctionnelle LongToIntFunction définit une opération qui attend en paramètre une valeur entière longue et renvoie une valeur entière. C'est une spécialisation de l'interface fonctionnelle Function qui attend une valeur entière longue et retourne une valeur entière.

Elle définit la méthode fonctionnelle applyAsInt(long valeur) qui renvoie une valeur de type int.

L'interface fonctionnelle ToDoubleBiFunction définit une opération qui attend en paramètre deux valeurs et renvoie un résultat. C'est une spécialisation de l'interface fonctionnelle BiFunction qui renvoie une valeur de type double.

L'interface ToDoubleBiFunction<T, U> est typée avec deux generic qui précisent le type de chacun des deux paramètres.

Elle définit la méthode fonctionnelle applyAsDouble(T t, U u) qui renvoie une valeur de type double.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.ToDoubleBiFunction;

public class TestToDoubleBiFunction {
  public static void main(String[] args) {
    ToDoubleBiFunction<Integer, Integer> trigo =
      (x, y) -> Math.sin(x) + Math.cos(y);
    System.out.println(trigo.applyAsDouble(3, 5));
  }
}
```

Résultat :

```
0.42478219352309343
```

L'interface fonctionnelle ToDoubleFunction définit une opération qui attend en paramètre une valeur et renvoie une valeur flottante. C'est une spécialisation de l'interface fonctionnelle Function qui renvoie une valeur flottante.

L'interface ToDoubleFunction<T> est typée avec un generic qui précise le type du paramètre.

Elle définit la méthode fonctionnelle applyAsDouble(T t) qui renvoie une valeur de type double.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.ToDoubleFunction;

public class TestToDoubleFunction {

  public static void main(String[] args) {
    ToDoubleFunction<Integer> calculCosinus = (x) -> Math.cos(x);
    System.out.println(calculCosinus.applyAsDouble(10));
  }
}
```

Résultat :

```
-0.8390715290764524
```

L'interface fonctionnelle ToIntBiFunction définit une opération qui attend en paramètre deux valeurs et renvoie un résultat. C'est une spécialisation de l'interface fonctionnelle BiFunction qui renvoie un entier.

L'interface ToIntBiFunction<T, U> est typée avec deux generic qui précisent le type de chacun des deux paramètres.

Elle définit la méthode fonctionnelle applyAsInt(T t, U u) qui renvoie une valeur de type int.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.ToIntBiFunction;

public class TestToIntBiFunction {

  public static void main(String[] args) {
    ToIntBiFunction<String, String> somme = (x, y) ->
      Integer.parseInt(x) + Integer.parseInt(y);
    System.out.println(somme.applyAsInt("123", "456"));
  }
}
```

Résultat :

```
579
```

L'interface fonctionnelle ToIntFunction définit une opération qui attend en paramètre une valeur et renvoie une valeur entière. C'est une spécialisation de l'interface fonctionnelle Function qui renvoie un entier.

L'interface ToIntFunction<T> est typée avec un generic qui précise le type du paramètre.

Elle définit la méthode fonctionnelle applyAsInt(T t) qui renvoie une valeur de type int.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.ToIntFunction;

public class TestToIntFunction {

  public static void main(String[] args) {
    ToIntFunction<String> convertEnInt = (x) -> Integer.parseInt(x);
    System.out.println(convertEnInt.applyAsInt("123"));
  }
}
```

Résultat :

```
123
```

L'interface fonctionnelle ToLongBiFunction définit une opération qui attend en paramètre deux valeurs et renvoie un résultat. C'est une spécialisation de l'interface fonctionnelle BiFunction qui renvoie un entier long.

L'interface ToLongBiFunction<T, U> est typée avec deux generic qui précisent le type de chacun des deux paramètres.

Elle définit la méthode fonctionnelle applyAsLong(T t, U u) qui renvoie une valeur de type long.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.ToLongBiFunction;

public class TestToLongBiFunction {

  public static void main(String[] args) {
    ToLongBiFunction<String, String> somme =
      (x, y) -> Long.parseLong(x) + Long.parseLong(y);
    System.out.println(somme.applyAsLong("123", "456"));
  }
}
```

Résultat :

```
579
```

L'interface fonctionnelle ToLongFunction définit une opération qui attend en paramètre une valeur et renvoie une valeur entière longue. C'est une spécialisation de l'interface fonctionnelle Function qui renvoie un entier long.

L'interface ToLongFunction<T> est typée avec un generic qui précise le type du paramètre.

Elle définit la méthode fonctionnelle applyAsLong(T t) qui renvoie une valeur de type long.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.ToLongFunction;

public class TestToLongFunction {

  public static void main(String[] args) {
    ToLongFunction<String> convertEnLong = (x) -> Long.parseLong(x);
    System.out.println(convertEnLong.applyAsLong("1234567890123456"));
  }
}
```

Résultat :

```
1234567890123456
```

L'interface fonctionnelle BinaryOperator définit une opération qui attend deux paramètres et renvoie une valeur, ces éléments étant tous du même type.

L'interface BinaryOperator<T> est typée avec un generic qui précise le type des paramètres et de la valeur de retour. Elle hérite de l'interface fonctionnelle BiFunction<T, T, T>.

Elle définit la méthode fonctionnelle apply(T t, T u) qui renvoie une valeur de type T.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.BinaryOperator;

public class TestBinaryOperator {

  public static void main(String[] args) {
    BinaryOperator<Integer> ajout = (a, b) -> a + b;
    System.out.println(ajout.apply(10, 20));
  }
}
```

Résultat :

```
30
```

Elle définit aussi plusieurs méthodes par défaut et static :

Méthode	Rôle
static <T> BinaryOperator<T> maxBy(Comparator<? super T> comparator)	Renvoyer une BinaryOperator qui renverra le plus grand des deux objets selon le Comparator fourni en paramètre
static <T> BinaryOperator<T> minBy(Comparator<? super T> comparator)	Renvoyer une BinaryOperator qui renverra le plus petit des deux objets selon le Comparator fourni en paramètre

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.Comparator;
import java.util.function.BinaryOperator;

public class TestBinaryOperator {

  public static void main(String[] args) {
    Comparator<Integer> comparateur = (a, b) -> b - a;
    BinaryOperator<Integer> biMin = BinaryOperator.minBy(comparateur);
    System.out.println(biMin.apply(2, 3));

    BinaryOperator<Integer> biMax = BinaryOperator.maxBy(comparateur);
    System.out.println(biMax.apply(2, 3));
  }
}
```

Résultat :

```
3
2
```

L'interface fonctionnelle DoubleBinaryOperator définit une opération qui attend deux paramètres et renvoie une valeur, tous ces éléments étant de type double. C'est une spécialisation de l'interface fonctionnelle BinaryOperator pour le type double.

Elle définit la méthode fonctionnelle applyAsDouble(double left, double rigth) qui renvoie une valeur de type double.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.DoubleBinaryOperator;

public class TestDoubleBinaryOperator {

  public static void main(String[] args) {
    DoubleBinaryOperator surfaceRectangle =
      (longueur, largeur) -> longueur * largeur;
    System.out.println(surfaceRectangle.applyAsDouble(10.5, 20.2));
  }
}
```

Résultat :

```
212.1
```

L'interface fonctionnelle DoubleUnaryOperator définit une opération qui attend un paramètre et renvoie une valeur, ces deux éléments étant de type double. C'est une spécialisation de l'interface fonctionnelle UnaryOperator pour le type double.

Elle définit la méthode fonctionnelle applyAsDouble(double operand) qui renvoie une valeur de type double.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.DoubleUnaryOperator;

public class TestDoubleUnaryOperator {

  public static void main(String[] args) {

    DoubleUnaryOperator surfaceCarre = (cote) -> cote * cote;
    System.out.println(surfaceCarre.applyAsDouble(10.5));
  }
}
```

Résultat :

```
110.25
```

Elle définit aussi plusieurs méthodes par défaut et static :

Méthode	Rôle
default DoubleUnaryOperator andThen(DoubleUnaryOperator)	Renvoyer une DoubleUnaryOperator qui exécute en séquence l'instance courante et celle fournie en paramètre
default DoubleUnaryOperator compose(DoubleUnaryOperator)	Renvoyer une DoubleUnaryOperator qui exécute l'instance fournie en paramètre et applique l'instance courante sur le résultat
static DoubleUnaryOperator identity()	Renvoyer un DoubleUnaryOperator qui renvoie toujours la valeur fournie en paramètre

L'interface fonctionnelle IntBinaryOperator définit une opération qui attend deux paramètres et renvoie une valeur, ces deux éléments étant de type int. C'est une spécialisation de l'interface fonctionnelle BinaryOperator pour le type int.

Elle définit la méthode fonctionnelle applyAsInt(int left, int rigth) qui renvoie une valeur de type int.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.IntBinaryOperator;

public class TestIntBinaryOperator {
  public static void main(String[] args) {
    IntBinaryOperator surfaceRectangle = (longueur, largeur) -> longueur * largeur;
    System.out.println(surfaceRectangle.applyAsInt(10, 20));
  }
}
```

Résultat :

```
200
```

L'interface fonctionnelle IntUnaryOperator définit une opération qui attend un paramètre et renvoie une valeur, ces deux éléments étant de type int. C'est une spécialisation de l'interface fonctionnelle UnaryOperator pour le type int.

Elle définit la méthode fonctionnelle applyAsInt(int operand) qui renvoie une valeur de type int.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.IntUnaryOperator;

public class TestIntUnaryOperator {

  public static void main(String[] args) {
    IntUnaryOperator surfaceCarre = (cote) -> cote * cote;
    System.out.println(surfaceCarre.applyAsInt(10));
  }
}
```

Résultat :

```
100
```

Elle définit aussi plusieurs méthodes par défaut et static :

Méthode	Rôle
default IntUnaryOperator andThen(IntUnaryOperator)	Renvoyer une IntUnaryOperator qui exécute en séquence l'instance courante et celle fournie en paramètre
default IntUnaryOperator compose(IntUnaryOperator)	Renvoyer une IntUnaryOperator qui exécute l'instance fournie en paramètre et applique l'instance courante sur le résultat
static IntUnaryOperator identity()	Renvoyer une IntUnaryOperator qui renvoie toujours la valeur fournie en paramètre

L'interface fonctionnelle LongBinaryOperator définit une opération qui attend deux paramètres et renvoie une valeur, tous ces éléments étant de type long. C'est une spécialisation de l'interface fonctionnelle BinaryOperator pour le type long.

Elle définit la méthode fonctionnelle applyAsLong(long left, long right) qui renvoie une valeur de type entière longue.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.LongBinaryOperator;

public class TestLongBinaryOperator {

  public static void main(String[] args) {
    LongBinaryOperator surfaceRectangle = (longueur, largeur) -> longueur * largeur;
    System.out.println(surfaceRectangle.applyAsLong(10L, 20L));
  }
}
```

Résultat :

```
200
```

L'interface fonctionnelle LongUnaryOperator définit une opération qui attend un paramètre et renvoie une valeur, tous ces éléments étant de type long. C'est une spécialisation de l'interface fonctionnelle UnaryOperator pour le type long.

Elle définit la méthode fonctionnelle applyAsLong(long operand) qui renvoie une valeur de type entière longue.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.LongUnaryOperator;

public class TestLongUnaryOperator {

  public static void main(String[] args) {
    LongUnaryOperator surfaceCarre = (cote) -> cote * cote;
    System.out.println(surfaceCarre.applyAsLong(10L));
  }
}
```

Résultat :

```
100
```

Elle définit aussi plusieurs méthodes par défaut et static :

Méthode	Rôle

default LongUnaryOperator andThen(LongUnaryOperator)	Renvoyer une LongUnaryOperator qui exécute en séquence l'instance courante et celle fournie en paramètre
default LongUnaryOperator compose(LongUnaryOperator)	Renvoyer une LongUnaryOperator qui exécute l'instance fournie en paramètre et applique l'instance courante sur le résultat
static LongUnaryOperator identity()	Renvoyer une LongUnaryOperator qui renvoie toujours la valeur fournie en paramètre

L'interface fonctionnelle UnaryOperator définit une opération qui attend un paramètre et renvoie une valeur, tous ces éléments étant de même type.

L'interface UnaryOperator<T> est typée avec un generic qui précise le type du paramètre et de la valeur de retour. Elle hérite de l'interface fonctionnelle Function<T, T>.

Elle définit la méthode fonctionnelle apply(T t, T u) qui renvoie une valeur de type T.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.UnaryOperator;

public class TestUnaryOperator {
  public static void main(String[] args) {
    UnaryOperator<String> minuscule  = (c)-> c.toLowerCase();
    System.out.println(minuscule.apply("TEST"));
  }
}
```

Résultat :

```
test
```

Elle définit aussi une méthode static :

Méthode	Rôle
static <T> UnaryOperator<T> identity()	Renvoyer une UnaryOperator qui renvoie toujours la valeur fournie en paramètre

12.4.4.3. Les interfaces fonctionnelles de type Predicate

Les interfaces fonctionnelles de type Predicate (Predicate, BiPredicate, DoublePredicate, IntPredicate, LongPredicate) définissent des fonctions qui attendent différents types de paramètres et renvoient une valeur booléenne.

L'interface fonctionnelle Predicate<T> définit une fonction qui effectue une opération sur un objet et renvoie une valeur booléenne.

Elle définit la méthode fonctionnelle test(T t) qui renvoie un booléen.

Elle définit aussi plusieurs méthodes par défaut ou static :

Méthode	Rôle
default Predicate<T> and(Predicate< ? super T>)	Renvoyer un Predicate qui exécute en séquence l'instance courante et celle fournie en paramètre en effectuant un ET logique sur leurs résultats. Si le prédicat courant est false alors celui fourni en paramètre n'est pas évalué. Une exception levée par l'un des deux est propagée à l'appelant. Si une exception est levée lors de l'exécution du premier prédicat, le second n'est pas évalué.

static <T> Predicate<T> isEqual(Object)	Renvoyer un Predicate qui teste l'égalité de l'objet fourni en paramètre et de celui passé en paramètre de la méthode test() en utilisant la méthode Objects.equals()
default Predicate<t> negate()	Renvoyer un Predicate qui exécute l'instance courante en effectuant un NOT logique sur son résultat
default Predicate<T> or(Predicate< ? super T>)	Renvoyer un Precidate qui exécute en séquence l'instance courante et celle fournie en paramètre en effectuant un OU logique sur leurs résultats. Si le prédicat courant est true alors celui fourni en paramètre n'est pas évalué. Une exception levée par l'un des deux est propagée à l'appelant. Si une exception est levée lors de l'exécution du premier prédicat, le second n'est pas évalué.

Exemple (code Java 8) :

```java
package fr.jmdoudoux.dej.java8.lambda;

import java.util.Objects;
import java.util.function.Predicate;

public class TestPredicate {

  public static void main(String[] args) {
    Predicate<String> possedeTailleTrois = s -> s.length() == 3;
    Predicate<String> contientX = s -> s.contains("X");
    Predicate<String> estNonNull = Objects::nonNull;
    Predicate<String> contientXOuTaille3 = contientX.or(possedeTailleTrois);
    Predicate<String> estSMS = Predicate.isEqual("SMS");

    System.out.println("1 "+contientX.negate().test("WXYZ"));
    System.out.println("2 "+contientX.or(possedeTailleTrois).test("WWW"));
    System.out.println("2 "+contientX.or(possedeTailleTrois).test("WX"));
    System.out.println("3 "+contientX.and(possedeTailleTrois).test("WXY"));
    System.out.println("3 "+contientX.and(possedeTailleTrois).test("WWW"));
    System.out.println("4 "+estNonNull.test(null));
    System.out.println("5 "+estNonNull.and(contientX).and(possedeTailleTrois)
    .test("WWW"));
    System.out.println("5 "+estNonNull.and(contientX).and(possedeTailleTrois)
    .test("XX"));
    System.out.println("5 "+estNonNull.and(contientX).and(possedeTailleTrois)
    .test(null));
    System.out.println("6 "+estNonNull.and(contientXOuTaille3).test("WWW"));
    System.out.println("6 "+estNonNull.and(contientXOuTaille3).test("XX"));
    System.out.println("6 "+estNonNull.and(contientXOuTaille3).test(null));
    System.out.println("7 "+estNonNull.and(contientX.or(possedeTailleTrois))
    .test("WWW"));
    System.out.println("7 "+estNonNull.and(contientX.or(possedeTailleTrois))
    .test("XX"));
    System.out.println("7 "+estNonNull.and(contientX.or(possedeTailleTrois))
    .test(null));
    System.out.println("8 "+estSMS.test("SMS"));
    System.out.println("8 "+estSMS.test("ABC"));
    System.out.println("8 "+estSMS.test(null));
  }
}
```

Résultat :

```
1 false
2 true
2 true
3 true
3 false
4 false
5 false
5 false
5 false
6 true
6 true
6 false
7 true
```

```
7 true
7 false
8 true
8 false
8 false
```

L'interface fonctionnelle BiPredicate définit une opération qui attend deux paramètres et renvoie une valeur booléenne. C'est une spécialisation de l'interface fonctionnelle Predicate qui attend deux paramètres.

L'interface BiPredicate<T,U> est typé avec deux generic qui précisent respectivement le type du premier et du second argument de l'opération de la fonction.

Elle définit la méthode fonctionnelle test(T t, U u) qui renvoie un booléen.

Elle définit aussi plusieurs méthodes par défaut :

Méthode	Rôle
default BiPredicate<T, U> and(BiPredicate< ? super T, ? super U>)	Renvoyer un BiPredicate qui exécute en séquence l'instance courante et celle fournie en paramètre en effectuant un ET logique sur leurs résultats. Si le prédicat courant est false alors celui fourni en paramètre n'est pas évalué. Une exception levée par l'un des deux est propagée à l'appelant. Si une exception est levée lors de l'exécution du premier prédicat, le second n'est pas évalué.
default BiPredicate<T, U> negate()	Renvoyer un BiPredicate qui exécute l'instance courante en effectuant un NOT logique sur son résultat
default BiPredicate<T, U> or(BiPredicate< ? super T, ? super U>)	Renvoyer un BiPrecidate qui exécute en séquence l'instance courante et celle fournie en paramètre en effectuant un OU logique sur leurs résultats. Si le prédicat courant est true alors celui fourni en paramètre n'est pas évalué. Une exception levée par l'un des deux est propagée à l'appelant. Si une exception est levée lors de l'exécution du premier prédicat, le second n'est pas évalué.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.BiPredicate;

public class TestBiPredicate {

  public static void main(String[] args) {
    BiPredicate<Integer, Integer> estSupOuEgal = (x, y) -> x >= y;
    BiPredicate<Integer, Integer> estLaMoitie = (x, y) -> x == y * 2;

    System.out.println("1 " + estSupOuEgal.test(2, 3));
    System.out.println("1 " + estSupOuEgal.test(3, 2));

    System.out.println("2 " + estSupOuEgal.and(estLaMoitie).test(4, 2));
    System.out.println("2 " + estSupOuEgal.and(estLaMoitie).test(3, 2));

    System.out.println("3 " + estSupOuEgal.negate().test(3, 2));

    System.out.println("4 " + estSupOuEgal.or(estLaMoitie).test(1, 1));
    System.out.println("4 " + estSupOuEgal.or(estLaMoitie).test(4, 2));
    System.out.println("4 " + estSupOuEgal.or(estLaMoitie).test(2, 4));
  }
}
```

Résultat :

```
1 false
1 true
2 true
2 false
3 false
4 true
```

```
4 true
4 false
```

L'interface fonctionnelle DoublePredicate définit une opération qui attend un nombre flottant et renvoie une valeur booléenne. C'est une spécialisation de l'interface fonctionnelle Predicate pour un nombre flottant.

Elle définit la méthode fonctionnelle test(double) qui renvoie un booléen.

Elle définit aussi des méthodes par défaut :

Méthode	Rôle
default DoublePredicate and(DoublePredicate)	Renvoyer un DoublePredicate qui exécute en séquence l'instance courante et celle fournie en paramètre en effectuant un ET logique sur leurs résultats. Si le prédicat courant est false alors celui fourni en paramètre n'est pas évalué. Une exception levée par l'un des deux est propagée à l'appelant. Si une exception est levée lors de l'exécution du premier prédicat, le second n'est pas évalué.
default DoublePredicate negate()	Renvoyer un DoublePredicate qui exécute l'instance courante en effectuant un NOT logique sur son résultat
default DoublePredicate or(DoublePredicate)	Renvoyer un DoublePredicate qui exécute en séquence l'instance courante et celle fournie en paramètre en effectuant un OU logique sur leurs résultats. Si le prédicat courant est true alors celui fourni en paramètre n'est pas évalué. Une exception levée par l'un des deux est propagée à l'appelant. Si une exception est levée lors de l'exécution du premier prédicat, le second n'est pas évalué.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.DoublePredicate;

public class TestDoublePredicate {

  private static final double DIX_MILLE = 10_000;

  public static void main(String[] args) {
    DoublePredicate estPositif = valeur -> valeur >= 0;
    DoublePredicate vautDixMille = valeur -> valeur == DIX_MILLE;

    System.out.println(estPositif.test(DIX_MILLE));
    System.out.println(estPositif.and(vautDixMille).test(DIX_MILLE));
    System.out.println(estPositif.negate().test(DIX_MILLE));
    System.out.println(estPositif.or(vautDixMille).test(100L));
  }
}
```

Résultat :

```
true
true
false
true
```

L'interface fonctionnelle LongPredicate définit une opération qui attend un entier et renvoie une valeur booléenne. C'est une spécialisation de l'interface fonctionnelle Predicate pour un entier long.

Elle définit la méthode fonctionnelle test(long) qui renvoie un booléen.

Elle définit aussi des méthodes par défaut :

Méthode	Rôle

default LongPredicate and(LongPredicate)	Renvoyer un LongPredicate qui exécute en séquence l'instance courante et celle fournie en paramètre en effectuant un ET logique sur leurs résultats. Si le prédicat courant est false alors celui fourni en paramètre n'est pas évalué. Une exception levée par l'un des deux est propagée à l'appelant. Si une exception est levée lors de l'exécution du premier prédicat, le second n'est pas évalué.
default LongPredicate negate()	Renvoyer un LongPredicate qui exécute l'instance courante en effectuant un NOT logique sur son résultat
default LongPredicate or(DoublePredicate)	Renvoyer un LongPredicate qui exécute en séquence l'instance courante et celle fournie en paramètre en effectuant un OU logique sur leurs résultats. Si le prédicat courant est true alors celui fourni en paramètre n'est pas évalué. Une exception levée par l'un des deux est propagée à l'appelant. Si une exception est levée lors de l'exécution du premier prédicat, le second n'est pas évalué.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.LongPredicate;

public class TestLongPredicate {

  private static final Long DIX_MILLE = 10_000L;

  public static void main(String[] args) {
    LongPredicate estPositif = valeur -> valeur >= 0;
    LongPredicate vautDixMille = valeur -> valeur == DIX_MILLE;

    System.out.println(estPositif.test(DIX_MILLE));
    System.out.println(estPositif.and(vautDixMille).test(DIX_MILLE));
    System.out.println(estPositif.negate().test(DIX_MILLE));
    System.out.println(estPositif.or(vautDixMille).test(100L));
  }
}
```

Résultat :

```
true
true
false
true
```

12.4.4.4. Les interfaces fonctionnelles de type Supplier

L'interface fonctionnelle Supplier définit une fonction qui renvoie une valeur dont le type correspond au type générique.

Elle définit la méthode fonctionnelle get() qui renvoie un objet de type T.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.Supplier;

public class TestSupplier {

  public static void main(String[] args) {
    Supplier<String> message = () -> "Bienvenue";
    System.out.println(message.get());
  }
}
```

Résultat :

```
Bienvenue
```

L'interface Supplier ne permet de renvoyer que des objets. Plusieurs interfaces fonctionnelles la spécialisent pour retourner des valeurs primitives : BooleanSupplier, DoubleSupplier, IntSupplier et LongSupplier.

L'interface fonctionnelle BooleanSupplier est une spécialisation de l'interface Supplier pour une valeur primitive de type booléenne.

Elle définit la méthode fonctionnelle getAsBoolean() qui renvoie une valeur de type boolean.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.BooleanSupplier;

public class TestBooleanSupplier {

  public static void main(String[] args) {
    int a = 10;
    int b = 12;
    BooleanSupplier aInferieurAB = () -> a <= b;
    System.out.println(aInferieurAB.getAsBoolean());
  }
}
```

Résultat :

```
true
```

L'interface fonctionnelle DoubleSupplier est une spécialisation de l'interface Supplier pour une valeur de type double.

Elle définit la méthode fonctionnelle getAsDouble() qui renvoie une valeur flottante de type double.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.DoubleSupplier;

public class TestDoubleSupplier {

  public static void main(String[] args) {
    DoubleSupplier pi = () -> 3.14116;
    System.out.println(pi.getAsDouble());
  }
}
```

Résultat :

```
3.14116
```

L'interface fonctionnelle IntSupplier est une spécialisation de l'interface Supplier pour une valeur de type int.

Elle définit la méthode fonctionnelle getAsInt() qui renvoie un entier de type int

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.IntSupplier;

public class TestIntSupplier {

  public static void main(String[] args) {
    int a = 10;
    int b = 12;
    IntSupplier aPlusB = () -> a + b;
    System.out.println(aPlusB.getAsInt());
  }
}
```

Résultat :

```
22
```

L'interface fonctionnelle LongSupplier est une spécialisation de l'interface Supplier pour une valeur de type long.

Elle définit la méthode fonctionnelle getAsLong() qui renvoie un entier de type long.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.lambda;

import java.util.function.LongSupplier;

public class TestLongSupplier {

  public static void main(String[] args) {
    int a = 100000000;
    LongSupplier aAuCarre = () -> (long) a * a;
    System.out.println(aAuCarre.getAsLong());
  }
}
```

Résultat :

```
10000000000000000
```

13. Les records

Chapitre 13

Niveau : 🟩🟩□□□□ Elémentaire

La définition d'une classe qui encapsule des données immuables est très verbeuse : constructeur, getters, redéfinition des méthodes equals(), hashCode(), toString(), ... Pour contourner cela, les IDE peuvent générer une bonne partie de ce code ou il est possible d'utiliser une solution comme Lombok qui repose sur le traitement d'annotations pour enrichir le bytecode.

Les records sont un nouveau type de classe dans le langage Java (record class), introduit en standard en Java 16, qui proposent une syntaxe compacte pour la déclaration de classes aux fonctionnalités restreintes qui agrègent des valeurs de manière immuable.

D'autres langages de programmation orientée objet proposent une syntaxe pour ce type de fonctionnalité : case classes en Scala, data classes en Kotlin, record classes en C#, ...

Les records ont plusieurs buts :

- créer facilement des objets qui expriment une simple agrégation de valeurs,
- offrir une syntaxe concise pour aider les développeurs à se concentrer sur la modélisation de données immuables plutôt que sur un comportement extensible,
- implémenter automatiquement des méthodes orientées données telles que les accesseurs et les méthodes equals(), hashCode() et toString()

La syntaxe des records est très concise mais ils ne sont pas aussi souples que les classes. Un record possède un nom, une description des éléments qu'il encapsule et un corps qui peut être vide, exprimé alors avec une paire d'accolades vides.

Exemple (code Java 14) :

```
public record Personne(String nom, String prenom) {}
```

Un record est une classe qui possède des caractéristiques particulières :

- c'est une classe final qui ne peut donc pas être enrichie par héritage d'un autre record ou d'une autre classe
- chaque élément de la description est encapsulé dans un champ private et final pour garantir l'immutabilité
- un getter public est proposé pour chaque élément
- un constructeur public qui possède la même signature que celle de la description qui initialise chaque élément avec la valeur correspondante fournie en paramètre
- une redéfinition des méthodes equals() et hashCode() qui garantit que deux instances sont égales si elles sont du même type et qu'elles contiennent les mêmes éléments
- une redéfinition de la méthode equals() qui contient le nom et la valeur de chaque élément encapsulé

Comme une enum, un record est une classe générée par le compilateur qui possède des contraintes. Les records possèdent des restrictions, notamment :

- ils ne peuvent pas hériter d'une autre classe
- il n'est pas possible d'ajouter d'autres champs non static que ceux définis dans la description
- ils ne peuvent pas être abstract

Il est possible d'utiliser des types génériques, d'implémenter des interfaces et d'utiliser des annotations.

Le corps du record peut permettre de définir

- des champs static
- des blocs static d'initialisation
- des constructeurs
- des méthodes statiques
- des méthodes d'instances
- des types imbriqués qui sont implicitement static

Ce chapitre contient plusieurs sections :

♦ L'introduction aux records
♦ La définition d'un record
♦ La mise en oeuvre des records

13.1. L'introduction aux records

Un record est basiquement une classe de données dont le but est d'encapsuler des données de manière immuable. Les records permettent de mettre en oeuvre des classes de données, sans avoir à écrire de code verbeux. Les classes de données simples sont réduites de nombreuses lignes de code à potentiellement une seule pour les cas les plus simples.

Les records proposent une nouvelle syntaxe dans le langage Java pour faciliter la déclaration d'une classe qui encapsule des données immuables. Le grand avantage est de réduire la quantité de code boilerplate (constructeur, getters, redéfinition des méthodes héritées d'Object (equals(), hashCode() et toString()).

Les records sont conçus pour être simples : un record a un nom et une description d'état, qui définit les composants du record. Un record permet ainsi d'exprimer clairement l'intention du type de servir de conteneur de données.

Les records sont un nouveau type qui est une forme restreinte de classe de la même manière qu'une énumération.

Le compilateur Java va générer, à partir de la déclaration d'une record, un constructeur, des champs privé finaux, des accesseurs en lecture et les méthodes hashCode(), equals() et toString().

Toutes ces fonctionnalités permettent de garantir que l'état d'un record soit immuable : tous les champs sont final et aucun setter n'est proposé.

13.1.1. L'historique

Via la JEP 350, Java 14 introduit en mode preview un nouveau type dans le langage : les records. Son but est de proposer une syntaxe compacte pour la déclaration de classes qui encapsulent des données immuables.

La JEP 384 incluse dans Java 15, une seconde preview est proposée. Elle intègre des améliorations au langage issues du feedback de la première preview :

- dans la première preview, les constructeurs canoniques devaient obligatoirement être public. Dans la seconde preview, la visibilité implicite du constructeur canonique est par défaut identique à la visibilité du record. Si le constructeur canonique est explicitement défini dans le code alors sa visibilité ne peut peut pas être inférieure à celle du record
- l'annotation @Override peut être utilisée sur un accesseur explicitement déclarée pour un composant du record
- l'affectation d'une valeur à l'un des champs d'instance dans le corps d'un constructeur compact provoque une erreur de compilation
- l'accès par réflexion à un champ d'un composant d'un record lève une exception de type IllegalAccessException
- un record ne peut pas avoir de méthode native

Elle introduit dans le langage la possibilité d'utiliser des interfaces et des énumérations locales ainsi que des records locaux.

Les records deviennent une fonctionnalité standard dans Java 16 via la JEP 395. Une seule modification est apportée aux records par rapport à la seconde preview du JDK 15 : assouplir la restriction historique selon laquelle une classe interne ne peut pas déclarer un membre qui est explicitement ou implicitement statique. Il est donc maintenant possible de permettre à une classe interne de déclarer un membre qui soit un record.

13.1.2. Le problème de la verbosité

Il est fréquent de déclarer des classes en Java qui ne contiennent que des données. C'est tellement courant que des patterns sont définis pour ce besoin tels que Value Object (VO) ou Data Transfert Object (DTO).

Java est un langage orienté objet : pour encapsuler des données, il faut écrire une classe et le code nécessaire pour permettre l'accès en lecture et/ou en écriture des données.

C'est une grande force mais pour une simple classe qui encapsule des données, la quantité de code à écrire est importante. La verbosité de Java est d'ailleurs fréquemment mise en avant.

Notamment en Java pour créer une classe qui encapsule des données, il est nécessaire d'écrire beaucoup de code à faible valeur ajoutée, répétitif et sujet aux erreurs : constructeurs, accesseurs et la redéfinition de certaines méthodes héritées de la classe Object (equals(), hashCode() et toString()).

Les IDE proposent des fonctionnalités pour générer une bonne partie de ce code mais ce code doit tout de même être lu pour comprendre son rôle.

Par exemple, une classe immuable Employe qui encapsule deux propriétés nom et prénom.

Exemple :

```
package fr.jmdoudoux.dej.records;

public final class Employe {

  private final String nom;
  private final String prenom;

  public Employe(String nom, String prenom) {
    super();
    this.nom = nom;
    this.prenom = prenom;
  }

  public String getNom() {
    return nom;
  }

  public String getPrenom() {
    return prenom;
  }

  @Override
  public int hashCode() {
    final int prime  = 31;
    int       result = 1;
    result = prime * result + ((nom == null) ? 0 : nom.hashCode());
    result = prime * result + ((prenom == null) ? 0 : prenom.hashCode());
    return result;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Employe other = (Employe) obj;
    if (nom == null) {
      if (other.nom != null)
        return false;
    } else if (!nom.equals(other.nom))
      return false;
    if (prenom == null) {
      if (other.prenom != null)
        return false;
    } else if (!prenom.equals(other.prenom))
      return false;
    return true;
  }

  @Override
  public String toString() {
    return "Employe [nom=" + nom + ", prenom=" + prenom + "]";
  }
}
```

La classe requière beaucoup de lignes de code (57) pour cette simple classe qui encapsule deux champs de manière immuables :

- la classe et les champs sont final
- un constructeur permet de fournir les valeurs
- des getters permettent de lire les valeurs
- les méthodes equals(), hashCode() et toString(), héritées de la classe Object sont redéfinies

Hormis la ligne de déclaration de la classe et de ces champs, toutes les autres lignes sont générées par l'IDE. Mais cela fait beaucoup de code à lire, même partiellement, pour comprendre le rôle de la classe.

La maintenance de ce code peut aussi introduire des problèmes : par exemple l'oubli de tenir compte dans les méthodes equals() et hashCode() lors de l'ajout d'un champs dont elles devraient tenir compte.

Il existe aussi des bibliothèques qui propose d'enrichir le bytecode avec tout ou partie des fonctionnalités requises. La plus connue et utilisée est Lombok.

13.1.3. Présentation des records

Un record est un nouveau type du langage Java qui permet au travers d'une syntaxe très simplifiée de définir une classe qui encapsule des données de manière immuable.

A première vue les records pourraient être considérés comme une solution pour réduire la quantité de code nécessaire pour créer des classes qui encapsulent des données. Comme le précise la JEP 359, les records ont aussi un objectif sémantique pour modéliser les données en tant que données. Les records offrent en réalité une solution plus sémantique : encapsuler des données de manière facile et concise en déclarant des classes qui encapsulent des données de manière immuable et fournissent des implémentations des méthodes relatives aux données.

L'utilisation des records facilite aussi la compréhension rapide du rôle du type défini. Même si une large partie du code d'une classe de données en Java peut être généré par un IDE, le développeur doit lire, même rapidement une bonne partie de ce code pour déterminer son rôle à la première lecture.

Les records permettent aussi d'ajouter une sémantique qui indique clairement que la classe encapsule des données de manière immuable. Ce type de classe est connue sous le nom de value object en Domain-Driven Design.

La sémantique des records se retrouvent dans certains autres langages comme les data classes de Kotlin ou les records de C# dont la sémantique est très proche de celle des records de Java.

Plusieurs cas d'usage des records peuvent être trouvés, par exemple :

- renvoyer plusieurs valeurs d'une méthode
- remplacer certaines structures de données utilisées par des frameworks : tuples, paires, ...
- utiliser comme DTO

- ...

Un record est une classe qui encapsule des données de manière immuable. Le but des records est de proposer un moyen syntaxiquement simple de créer une telle classe.

Basiquement la définition d'un record tient en une seule ligne.

L'exemple ci-dessous définit un record qui encapsule deux données : un nom et un prénom de manière immuable.

Exemple (code Java 14) :

```
package fr.jmdoudoux.dej.records;

public record Employe(String nom, String prenom) {
}
```

Il faut compiler le code avec le compilateur Java sans oublier de préciser les options nécessaires à l'utilisation de fonctionnalité en mode preview en Java 14 et 15.

Exemple (code Java 16) :

```
C:\java\src\fr\jmdoudoux\dej\records>javac Employe.java
```

Le compilateur va générer une classe immuable sur la base des informations fournies dans la définition du record contenant :

- des champs finaux
- un constructeur pour initialiser les valeurs des champs
- des getters
- la redéfinition des méthodes equals(), hashCode() et toString()

Résultat :

```
C:\java\src\fr\jmdoudoux\dej\records>javap Employe.class
Compiled from "Employe.java"
public final class fr.jmdoudoux.dej.records.Employe extends java.lang.Record {
  public fr.jmdoudoux.dej.records.Employe(java.lang.String, java.lang.String);
  public java.lang.String toString();
  public final int hashCode();
  public final boolean equals(java.lang.Object);
  public java.lang.String nom();
  public java.lang.String prenom();
}
```

Un record hérite de la classe java.lang.Record.

Les accesseurs en lecture seule sont des méthodes dont le nom correspond au nom du membre du record. Le nom des getters ne respecte pas la convention JavaBean qui recommande de préfixer ces méthodes par get ou is (pour les valeurs booléennes). Le nom des getters correspond au nom du champ.

La redéfinition de la méthode equals() considère deux records comme égaux s'ils sont du même type et si tous les champs ont la même valeur.

La redéfinition de la méthode toString() retourne une chaîne de caractères qui contient le nom du record suivi d'une paire de crochet qui contient les paires nom du champs = valeur séparées par une virgule.

Les méthodes sont par défaut générées par le compilateur : il est possible de fournir une redéfinition personnalisée au besoin.

Il est possible d'ajouter des membres (champs, méthodes et constructeurs) sous certaines conditions et restrictions.

L'implémentation des records dans le langage est similaire à celui des enums. Une enum est aussi une classe est une sémantique spécifique et une syntaxe plus concise. Les records sont comme les enums des formes limitées de classes.

Comme les records sont des classes, la plupart des fonctionnalités des classes sont conservées

La possibilité de redéfinir les méthodes générées ou d'ajouter certains membres offre un bon compromis entre simplicité et flexibilité.

13.2. La définition d'un record

Les records proposent une syntaxe concise qui ne permet que de déclarer que les informations importantes : un nom et les éléments qu'il encapsule.

La définition d'un record comporte plusieurs parties :

- le mot clé contextuel record
- le nom du record
- la liste des composants (éléments qui composent l'état du record) entre parenthèses
- le corps du record : par défaut il peut être vide sous la forme d'une paire d'accolades

13.2.1. Le record minimaliste

La déclaration minimale est composée de l'identifiant restreint record suivi du nom du record, d'une paire de parenthèses et d'une paire d'accolades.

Exemple (code Java 14) :

```
record Employe() {
}
```

Un record est implicitement final : il est cependant possible d'utiliser explicitement le modificateur final.

Un record hérite de la classe java.lang.Record : il n'est donc pas possible d'utiliser la clause extends dans la définition d'un record.

Il ne peut donc pas être abstrait : il ne peut pas avoir le modificateur abstract.

Ce record minimaliste peut déjà être instancié. Ces méthodes toString(), equals() et hashCode() sont redéfinies par le compilateur.

Exemple (code Java 14) :

```
  public static void main(String[] args) {
    var emp = new Employe();
    System.out.println(emp);
    System.out.println(emp.hashCode());
  }
```

Résultat :

```
Employe[]
0
```

Il est possible de tester l'égalité sur deux instances.

Exemple (code Java 14) :

```
  public static void main(String[] args) {
    var emp1 = new Employe();
    var emp2 = new Employe();
    System.out.println(emp1==emp2);
    System.out.println(emp1.equals(emp2));
  }
```

Résultat :

```
false
true
```

Les deux instances obtenues sont différentes. Par contre, elles sont égales car les données encapsulées sont les mêmes (aucune pour le moment).

13.2.2. L'identifiant restreint record

« record » est un identifiant restreint (restricted identifier) comme var mais n'est pas un mot clé réservé.

« record » a un rôle particulier lors de la définition d'un type.

Il est cependant tout à fait licite de définir une variable ou une méthode dont le nom est record.

Exemple :

```
  public int record() {
    int record = 0;
    return record;
  }
```

13.2.3. L'ajout des composants

Les composants d'un record sont précisés dans l'en-tête de la déclaration d'un record définie entre la paire de parenthèses. Chaque composant d'un record est constitué d'un type (éventuellement précédé d'une ou plusieurs annotations) et d'un identifiant qui spécifie le nom du composant. Un composant d'un record générera par le compilateur deux membres de la classe du record : un champ privé déclaré implicitement, et une méthode d'accès publique déclarée explicitement ou implicitement dont le nom est celui du composant.

Exemple (code Java 14) :

```
record Employe(String nom) {
}
```

Il est bien sûr possible d'ajouter plusieurs composants, chacun séparé par une virgule.

Exemple (code Java 14) :

```
record Employe(String nom, String prenom) {
}
```

Il n'est pas possible de déclarer deux composants avec le même nom.

Exemple (code Java 16) :

```
public record Employe(String nom, String nom) {
}
```

Résultat :

```
C:\java>javac Employe.java
Employe.java:1: error: record component nom is already defined in record Employe
public record Employe(String nom, String nom) {
                                         ^
Employe.java:1: error: method nom() is already defined in record Employe
public record Employe(String nom, String nom) {
                      ^
2 errors
```

Il n'est pas possible d'utiliser les noms clone, finalize, getClass, hashCode, notify, notifyAll, toString et wait dans les composants d'un record. Ce sont les noms des méthodes publiques et protégées sans argument dans Object. Le fait de les interdire en tant que noms de composants d'un record évite plusieurs confusions :

- chaque classe de record fournit des implémentations de hashCode() et toString() qui renvoient des représentations d'un record : elles ne peuvent pas servir de méthodes d'accès pour des composants de record appelés hashCode ou toString
- certaines classes de record peuvent fournir des implémentations de clone et de finalize, de sorte qu'un composant de record appelé clone ou finalize ne pourrait pas être accessible via une méthode de type accesseur
- les méthodes getClass, notify, notifyAll et wait d'Object sont finales, de sorte que les composants de record portant les mêmes noms ne peuvent pas avoir de méthodes de type accesseur. Les méthodes de type accesseur auraient les mêmes signatures que les méthodes finales, et tenteraient donc de les surcharger, ce qui est impossible

Exemple (code Java 16) :

```
public record Employe(String clone, String notify) {
}
```

Résultat :

```
C:\java>javac Employe.java
Employe.java:1: error: illegal record component name clone
public record Employe(String clone, String notify) {
                             ^
Employe.java:1: error: illegal record component name notify
public record Employe(String clone, String notify) {
                                           ^
2 errors
```

Un seul composant d'un record peut être un varargs et dans ce cas doit être le dernier composant.

Si un record ne possède pas de composants, sa définition doit contenir une paire de parenthèses vides.

Les méthodes toString(), equals() et hashCode() sont redéfinies par le compilateur en prenant en compte les composants du record.

Exemple (code Java 14) :

```
        public static void main(String[] args) {
               var emp = new Employe("nom1", "prenom1");
               System.out.println(emp);
               System.out.println(emp.hashCode());
               var emp2 = new Employe("nom1", "prenom1");
               System.out.println(emp.equals(emp2));
        }
```

Résultat :

```
Employe[nom=nom1, prenom=prenom1]
-213416509

true
```

Le compilateur ajoute :

- un champ privé final pour chaque composant
- un constructeur qui attend en paramètre les valeurs de chacun des composants
- un accesseur pour chaque composant dont le nom est le nom du composant
- une redéfinition des méthodes toString(), equals() et hashCode() qui prennent en compte les composants

13.2.4. L'implémentation d'interfaces

Un record peut implémenter une ou plusieurs interfaces.

Exemple :

```
public interface EtatCivil {
  String getNomPrenom();
}
```

Exemple (code Java 14) :

```
public record Employe(String nom, String prenom) implements EtatCivil {

  @Override
  public String getNomPrenom() {
    return nom + " " + prenom;
  }
}
```

13.2.5. L'utilisation d'annotations avec les records

Il est possible d'utiliser des annotations sur un record sous réserve que leur target soit correctement défini pour cela :

- ElementType.TYPE : pour qu'une annotation puisse être utilisée sur un record
- ElementType.RECORD_COMPONENT : pour qu'une annotation puisse être utilisée sur un composant d'un record

Exemple (code Java 14) :

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.RECORD_COMPONENT, ElementType.TYPE})
public @interface MonAnnotation {

}
```

Il est alors possible d'utiliser l'annotation.

Exemple (code Java 14) :

```
@MonAnnotation
public record Employe(@MonAnnotation String nom, String prenom) {
}
```

Les annotations sur une déclaration de composant de record sont disponibles par réflexion si leurs interfaces d'annotation sont applicables sur un composant de record. Indépendamment, les annotations sur une déclaration de composant de record sont propagées aux déclarations des membres et des constructeurs de la classe du record si leurs interfaces d'annotation sont applicables sur ces membres.

Les annotations sont utilisables sur les composants d'un record si elles sont applicables aux composants de records, aux paramètres, aux champs ou aux méthodes. Les annotations qui sont applicables à l'une de ces cibles sont propagées aux membres générés (champs, paramètre du constructeur, méthodes).

Les composants d'un record ont plusieurs utilités dans la déclaration d'un record. Chaque composant sera utilisé par le compilateur pour générer un champ du même nom et du même type, à un accesseur du même nom et du même type de retour, et à un paramètre du constructeur du même nom et du même type.

Les annotations sur les composants sont propagées aux éléments générés par le compilateur en fonction des éléments précisés dans la méta-annotation @Target. Lorsqu'une annotation est utilisée sur un composant d'un record, celle est appliquée sur chacun de ses éléments selon la cible d'utilisation de l'annotation définit avec l'annotation @Target (le type sur le composant du record, le type sur le champ correspondant, le type de retour de l'accesseur correspondant, le type du paramètre correspondant du constructeur canonique).

Les règles de propagation de l'annotation utilisée sur un composant d'un record sont :

- si une annotation sur un composant de record est applicable à une déclaration de champ, alors l'annotation apparaît sur le champ privé correspondant
- si une annotation sur un composant de record est applicable à une déclaration de méthode, alors l'annotation apparaît sur la méthode accesseur correspondante
- si une annotation sur un composant de record s'applique à un paramètre, l'annotation apparaît sur le paramètre correspondant du constructeur canonique s'il n'est pas déclaré explicitement, ou sur le paramètre correspondant du constructeur compact s'il est déclaré explicitement
- si une annotation sur un composant de record est applicable à un type, les règles de propagation sont les mêmes que pour les annotations de déclaration, sauf que l'annotation apparaît sur l'utilisation du type correspondant plutôt que sur la déclaration

Cela permet aux classes qui utilisent des annotations sur leurs champs, paramètres de constructeur ou méthodes d'accesseur d'être migrées vers des records sans avoir à déclarer ces membres de manière redondante.

Si un accesseur public ou un constructeur canonique (non compact) est déclaré explicitement, il ne possède que les annotations qui sont directement définies sur lui : rien n'est propagé du composant du record correspondant à ces membres.

13.2.6. Le support des génériques

Un record peut être typé avec des génériques.

Exemple (code Java 14) :

```
package fr.jmdoudoux.dej.records;

import java.util.List;

public record Employe<T>(String nom, String prenom, List<T> competences) {

}
```

Exemple (code Java 14) :

```
package fr.jmdoudoux.dej.records;

import java.util.List;

public class Main {

  public static void main(String[] args) throws Exception {

    var emp = new Employe<String>("nom1", "prenom1", List.of("Analyse","Développement"));
    System.out.println(emp);
  }
}
```

Résultat :

```
Employe[nom=nom1, prenom=prenom1, competences=[Analyse, Développement]]
```

13.2.7. L'immutabilité

Les records sont immuables par défaut, grâce à plusieurs caractéristiques :

- tous les champs déclarés sont implicitement final
- initialisées uniquement par le constructeur (aucun setter n'est proposé)
- les valeurs sont accessibles uniquement grâce à des méthodes en lecture seule
- la classe du record est final

Si les champs sont des objets, seuls les références sont immuables.

Exemple (code Java 14) :

```
public record Employe(String nom, String prenom, List<String> competences) {
}
```

Exemple (code Java 14) :

```
  public static void main(String[] args) {
    List<String> competences = new ArrayList<>();
    competences.add("Java");
    var emp = new Employe("nom1", "prenom1", competences);
    System.out.println(emp);
    System.out.println(emp.hashCode());

    competences.add("HTML");
    System.out.println(emp);
    System.out.println(emp.hashCode());
  }
```

Résultat :

```
Employe[nom=nom1, prenom=prenom1, competences=[Java]]
1976324350
Employe[nom=nom1, prenom=prenom1, competences=[Java, HTML]]
2047598599
```

Ainsi pour garantir l'immutabilité totale d'un record, il faut que toutes les instances qu'il encapsule soient elles-mêmes immuables.

Exemple (code Java 14) :

```
  public static void main(String[] args) {
    var emp = new Employe("nom1", "prenom1", List.of("Java"));
    emp.competences().add("HTML");
  }
```

Résultat :

```
Exception in thread "main" java.lang.UnsupportedOperationException
        at java.base/java.util.ImmutableCollections.uoe(ImmutableCollections.java:73)
        at java.base/java.util.ImmutableCollections$AbstractImmutableCollection.add(
ImmutableCollections.java:77)
        at fr.jmdoudoux.dej.records/fr.jmdoudoux.dej.records.Main.main(Main.java:9)
```

Dans les cas où l'on utilise des types de propriétés qui sont intrinsèquement mutables, par exemple un tableau, il faut redéfinir explicitement la méthode d'accès à la propriété pour qu'il retourne une copie des objets est un moyen d'assurer l'immuabilité.

Par exemple, en créant et en retournant une copie défensive d'un tableau.

Exemple (code Java 14) :

```
package fr.jmdoudoux.dej.records;

import java.util.Arrays;

public record Employe(String nom, String prenom, String[] competences) {

  @Override
  public String[] competences() {
    return this.competences.clone();
  }

  @Override
  public String toString() {
    return "Employe[nom=" + nom + ", prenom=" + prenom + ", competences="
        + Arrays.deepToString(competences) + "]";
  }
}
```

Exemple (code Java 14) :

```
package fr.jmdoudoux.dej.records;

public class TestEmploye {
  public static void main(String[] args) {
    String[] competences = { "Java" };
    var emp = new Employe("nom1", "prenom1", competences);
    System.out.println(emp);
    emp.competences()[0] = "UML";
    System.out.println(emp);
  }
}
```

Résultat :

```
Employe[nom=nom1, prenom=prenom1, competences=[Java]]
Employe[nom=nom1, prenom=prenom1, competences=[Java]]
```

Par exemple, en enveloppant une List avec le résultat de l'invocation de la méthode Collections.unmodifiableList().

Exemple (code Java 14) :

```
package fr.jmdoudoux.dej.records;

import java.util.Collections;
import java.util.List;

public record Employe(String nom, String prenom, List<String> competences) {

  @Override
  public List<String> competences() {
    return Collections.unmodifiableList(this.competences);
  }
}
```

Exemple (code Java 14) :

```
package fr.jmdoudoux.dej.records;

import java.util.ArrayList;
import java.util.List;

public class TestEmploye {
  public static void main(String[] args) {
    List<String> competences = new ArrayList<>();
    competences.add("Java");
    var emp = new Employe("nom1", "prenom1", competences);
    System.out.println(emp);
    emp.competences().add("UML");
  }
}
```

Résultat :

```
Employe[nom=nom1, prenom=prenom1, competences=[Java]]
Exception in thread "main" java.lang.UnsupportedOperationException
    at java.base/java.util.Collections$UnmodifiableCollection.add(Collections.java:1062)
    at fr.jmdoudoux.dej.records/fr.jmdoudoux.dej.records.TestEmploye.main(TestEmploye.java:13)
```

Comme tout objet immuable, si celui-ci doit être modifié, il faut en créer une nouvelle instance avec les valeurs modifiées.

Exemple (code Java 14) :

```
package fr.jmdoudoux.dej.records;

public record Employe(String nom, String prenom) {
}
```

Exemple (code Java 14) :

```
package fr.jmdoudoux.dej.records;

public class TestEmploye {
  public static void main(String[] args) {
    var emp = new Employe("nom1", "prenom1");
    System.out.println(emp);
    emp = new Employe(emp.nom(), emp.prenom() + " modifie");
    System.out.println(emp);
  }
}
```

Résultat :

```
Employe[nom=nom1, prenom=prenom1]
Employe[nom=nom1, prenom=prenom1 modifie]
```

13.2.8. La redéfinition de membres

La définition d'un record offre certaines flexibilités : il est possible de redéfinir les méthodes implémentées par le compilateur : constructeur, accesseurs, equals(), hashCode() et toString().

13.2.8.1. Le constructeur généré et sa redéfinition

Contrairement aux classes qui possède un constructeur par défaut si aucun constructeur n'est défini explicitement, les records ne possèdent par défaut qu'un constructeur dit canonique généré par le compilateur qui attend en paramètre les valeurs des différents composants du record.

Cependant, il est parfois nécessaire de personnaliser le constructeur notamment pour permettre de valider une ou plusieurs valeurs fournies pour initialiser l'état du record.

Mais fréquemment, un constructeur doit, en plus d'assigner des valeurs, faire des contrôles sur ces valeurs ou exécuter d'autres opérations d'initialisation.

Il est possible de redéfinir le constructeur en précisant tous les champs en paramètre dénommé le constructeur canonique (canonical constructor).

Exemple (code Java 14) :

```
package fr.jmdoudoux.dej.records;

public record Employe(String nom, String prenom) {

  public Employe(String nom, String prenom) {
    if (nom == null || nom.trim().isEmpty()) {
      throw new IllegalArgumentException("Le nom est obligatoire.");
    }
    this.nom = nom;
    this.prenom = prenom;
  }
}
```

Remarque : en Java 14, le constructeur canonique doit être explicitement public. En Java 15, sa visibilité ne peut pas être inférieure à celle du record.

Exemple (code Java 14) :

```
package fr.jmdoudoux.dej.records;

public class Main {

  public static void main(String[] args) throws Exception {

    var emp = new Employe("", "prenom1");
  }
}
```

Résultat :

```
Exception in thread "main" java.lang.IllegalArgumentException: Le nom est obligatoire.
        at fr.jmdoudoux.dej.records/fr.jmdoudoux.dej.records.Employe.<init>(Employe.java:7)
        at fr.jmdoudoux.dej.records/fr.jmdoudoux.dej.records.Main.main(Main.java:7)
```

L'ordre des paramètres du constructeur canonique doit rester l'ordre de définition des composants dans l'entête du record. Si ce n'est pas le cas, le compilateur émet une erreur.

Exemple (code Java 14) :

```
public record Employe(String nom, String prenom) {

  public Employe(String prenom, String nom) {
    if (nom == null || nom.trim()
        .isEmpty()) {
      throw new IllegalArgumentException("Le nom est obligatoire.");
    }
    this.nom = nom;
    this.prenom = prenom;
  }
}
```

Résultat :

```
C:\java>javac Employe.java
Employe.java:3: error: invalid canonical constructor in record Employe
  public Employe(String prenom, String nom) {
         ^
  (invalid parameter names in canonical constructor)
1 error

C:\java>
```

Il est possible d'utiliser une syntaxe raccourcie pour définir des traitements du constructeur canonique nommée constructeur compact (compact constructor). Ces traitements peuvent par exemple permettre la validation des valeurs fournies.

Cette syntaxe évite d'avoir à fournir explicitement les paramètres. Il ne faut même pas utiliser une paire de parenthèses vide puisque dans ce cas cela définit le constructeur par défaut.

Un constructeur compact n'est pas un constructeur en tant que tel : il définit du code qui sera exécuté en début du constructeur canonique.

Exemple (code Java 14) :

```
package fr.jmdoudoux.dej.records;

public record Employe(String nom, String prenom) {

  public Employe {
    if (nom == null || nom.trim().isEmpty()) {
      throw new IllegalArgumentException("Le nom est obligatoire.");
    }
  }
}
```

Remarque : en Java 14, le constructeur compact doit aussi être explicitement public.

En Java 15, la visibilité du constructeur compact ne peut pas être inférieure à celle du record.

13.2.8.2. La redéfinition d'un accesseur

Il est possible de redéfinir un accesseur pour modifier son comportement par défaut ou pour ajouter une annotation.

Exemple (code Java 14) :

```
public record Employe(String nom, String prenom) {

  public String nom() {
    return nom.toUpperCase();
  }
}
```

Exemple (code Java 14) :

```
  public static void main(String[] args) {
    var emp = new Employe("nom1", "prenom1");
    System.out.println(emp);
    System.out.println(emp.nom());
  }
```

Résultat :

```
Employe[nom=nom1, prenom=prenom1]
NOM1
```

13.2.9. L'ajout de membres à un record

Il n'est pas possible d'ajouter des champs d'instance à un record, ce qui est normal puisque de tels champs ne seraient pas correctement gérés pour que le record soit immuable.

Mais il est possible d'ajouter certains membres à un record :

- des champs static
- des constructeurs
- des méthodes

Bien que l'on puisse ajouter des membres supplémentaires dans un record, il n'est pas recommandé d'en abuser. Les records sont conçus pour être de simples classes qui encapsulent des données et l'ajout d'un nombre important de membres est probablement le signe qu'il est préférable d'utiliser une classe standard.

L'objectif des records est de permettre aux développeurs de regrouper des données en un seul élément immuable sans avoir à écrire un code verbeux. Cela signifie que si vous êtes tenté d'ajouter d'autres champs/méthodes à un record alors il est probable qu'une classe standard aurait surement plus de sens pour répondre au besoin.

13.2.9.1. L'implémentation de constructeurs

Par défaut, le compilateur ne génère qu'un seul constructeur dont les paramètres permettent de fournir les valeurs de tous les composants. Il ne génère donc pas de constructeur par défaut puisque cela irait à l'encontre de l'immutabilité d'un record.

Les records permettent de déclarer plusieurs constructeurs avec ou sans paramètres. Il est ainsi possible de définir ses propres constructeurs pour par exemple fournir des valeurs par défaut à certains composants.

Exemple (code Java 14) :

```
public record Employe(String nom, String prenom) {

  public Employe() {
    this("Inconnu","Inconnu");
  }
}
```

Le record possède alors deux constructeurs : celui généré par le compilateur qui attend en paramètre le nom et le prénom et celui sans paramètre qui est définit explicitement

Exemple (code Java 14) :

```
  public static void main(String[] args) {
    var emp = new Employe("nom1", "prenom1");
    var emp2 = new Employe();
    System.out.println(emp);
    System.out.println(emp2);
  }
```

Résultat :

```
Employe[nom=nom1, prenom=prenom1]
Employe[nom=Inconnu, prenom=Inconnu]
```

Un constructeur peut permettre de fournir les valeurs avec un nombre différents de paramètres de celui de la définition des composants du record.

Exemple (code Java 14) :

```
package fr.jmdoudoux.dej.records;

public record Employe(String nom, String prenom) {

  public Employe(String nomPrenom) {
    this(nomPrenom.split(" ")[0], nomPrenom.split(" ")[1]);
  }
}
```

Dans les constructeurs qui ne sont pas le constructeur canonique, il faut obligatoirement invoquer un autre constructeur comme première instruction. Si ce n'est pas le cas, le compilateur émet une erreur.

Exemple (code Java 16) :

```
public record Employe(String nom, String prenom) {

  public Employe() {
    this.nom = "Inconnu";
    this.prenom = "Inconnu";
  }
}
```

Résultat :

```
C:\java>javac Employe.java
Employe.java:3: error: constructor is not canonical, so its first statement must invoke
 another constructor of class Employe
  public Employe() {
         ^
1 error
```

Le constructeur invoqué n'est pas obligatoirement le constructeur canonique : cela peut être n'importe quel constructeur défini dans le record.

Exemple (code Java 16) :

```
public record Employe(String nom, String prenom) {

  public Employe(String nom) {
    this(nom,"Inconnu");
  }

  public Employe() {
    this("Inconnu");
  }
}
```

Dans les constructeurs qui ne sont pas le constructeur canonique, il n'est pas possible d'assigner une valeur à un des champs puisque ceux-ci ont déjà été initialisés par le constructeur invoqué. Si c'est le cas, le compilateur émet une erreur :

Exemple (code Java 16) :

```
public record Employe(String nom, String prenom) {

  public Employe() {
    this("Inconnu","Inconnu");
    this.nom = "Test";
    this.prenom = "Test";
  }
}
```

Résultat :

```
C:\java>javac Employe.java
Employe.java:5: error: variable nom might already have been assigned
    this.nom = "Test";
        ^
Employe.java:6: error: variable prenom might already have been assigned
    this.prenom = "Test";
        ^
2 errors
```

13.2.9.2. L'ajout de membres statiques

Il est possible d'ajouter des champs statiques.

Un record peut aussi avoir des méthodes statiques. Cela peut par exemple permettre de proposer des fabriques.

Exemple (code Java 14) :

```
  public static Employe getEmploye(String nomPrenom) {
    String[] split = nomPrenom.split(" ");
    return new Employe(split[0], split[1]);
  }
```

Exemple (code Java 14) :

```
  public static void main(String[] args) {
    var emp = Employe.getEmploye("nom1 prenom1");
    System.out.println(emp);
  }
```

Exemple (code Java 14) :

```
Employe[nom=nom1, prenom=prenom1]
```

13.2.9.3. Les méthodes d'instances

Bien que le but premier d'un record soit d'encapsuler des données, il est possible d'y ajouter des méthodes d'instances.

Exemple (code Java 14) :

```
package fr.jmdoudoux.dej.records;

public record Employe(String nom, String prenom) {

  public String getNomPrenom() {
    return nom+" "+prenom;
  }
}
```

Comme les champs sont final, ces méthodes ne peuvent pas changer l'état d'une instance d'un Record.

13.3. La mise en oeuvre des records

Un record peut être défini en tant que :

- classe de premier niveau
- membre d'une classe (record imbriqué)
- record local

Une nouvelle instance est créée en utilisant l'instruction new.

13.3.1. Les records comme membre imbriqué

Un record peut déclarer des types imbriqués, y compris des records.

Un record imbriqué est implicitement static : cela évite que l'instance englobante ajoute silencieusement de l'état au record.

Exemple (code Java 14) :

```
public record MonRecord() {

  public record MonRecordImbrique() {
  }
}

class TestRecordImbrique {
  public static void main(String[] args) {
    MonRecord mr = new MonRecord();

    MonRecord.MonRecordImbrique mri = new MonRecord.MonRecordImbrique();
  }
}
```

Il est permis de spécifier de manière redondante le modificateur static dans la déclaration d'un record qui est un membre d'une classe, mais cela n'est pas permis dans la déclaration d'un record local.

Exemple (code Java 14) :

```
public record MonRecord() {

  public static record MonRecordImbrique() {
  }
}
```

13.3.2. Les records locaux

Les records permettre de regrouper un ensemble de valeurs. Il est pratique de déclarer des records pour modéliser ces valeurs. Une option consiste à déclarer des records imbriqués, comme cela se faisait historiquement avec des classes helper.

La JEP 384 ajoutée dans Java 15, introduit dans le langage Java la possibilité de définir des records locaux, donc des records qui sont définis dans une méthode. Cela peut par exemple permettre de définir un record qui va stocker des valeurs intermédiaires au plus près de là elles seront utilisées. Ceci est notamment utile pour simplifier l'utilisation de certaines opérations des Streams. Il est parfois nécessaire qu'un Stream passe plusieurs valeurs pour chaque élément. Un record local peut être défini à la place de la définition d'un type dédié.

Dans l'exemple ci-dessous, un record est défini pour encapsuler un étudiant et sa moyenne calculée via l'invocation d'une méthode.

Exemple (code Java 16) :

```
import java.util.List;
import java.util.concurrent.ThreadLocalRandom;
import java.util.stream.Collectors;

public class TestRecordLocal {

  public static void main(String[] args) {
    TestRecordLocal trl = new TestRecordLocal();
    List<Etudiant> meilleurs = trl.getMeilleursEtudiants(
        List.of(new Etudiant("Nom1"), new Etudiant("Nom3"), new Etudiant("Nom2"),
            new Etudiant("Nom4")));
    System.out.println(meilleurs);
  }

  public List<Etudiant> getMeilleursEtudiants(List<Etudiant> etudiants) {

    record EtudiantMoyenne(Etudiant etudiant, double moyenne){};

    return etudiants.stream()
             .map(e->new EtudiantMoyenne(e,calculerMoyenne(e)))
             .peek(System.out::println)
             .sorted((e1,e2)->Double.compare(e2.moyenne(),e1.moyenne()))
             .map(EtudiantMoyenne::etudiant)
             .collect(Collectors.toList());
  }

  private double calculerMoyenne(Etudiant e) {
    return ThreadLocalRandom.current().nextDouble(0, 20);
  }
}
```

Les records locaux sont un cas particulier de records imbriqués. Comme tous les records imbriqués, les records locaux sont implicitement static.

Cela implique que les méthodes d'un record local ne peuvent pas accéder aux variables de la méthode englobante, ce qui permet d'éviter de capturer une instance immédiatement englobante qui ajouterait silencieusement de l'état au record.

Le fait que les records locaux soient implicitement static contraste avec les classes locales, qui ne sont pas implicitement static. Les classes locales ne sont jamais static, implicitement ou explicitement, et peuvent toujours accéder aux variables de la méthode englobante.

Il n'est cependant pas possible d'utiliser le modificateur static dans la déclaration d'un record local : le compilateur émet une erreur dans ce cas.

13.3.3. Les limitations et les incompatibilités

En raison de la finalité des records, plusieurs contraintes doivent être respectées et sont vérifiées par le compilateur :

- comme un record est immuable, ses composants sont implicitement final
- il n'est pas possible d'ajouter des propriétés d'instance différentes de celles déclarés : pour assurer l'immutabilité via les membres générés par le compilateur
- un record est implicitement final : ils ne peuvent servir de classe mère
- un record ne peut pas être abstract puisqu'il est final
- un record ne peut pas hériter explicitement d'une classe puisque la classe générée hérite déjà de la classe java.lang.Record, pas même la classe java.lang.Record elle-même
- un record ne peut pas déclarer de méthodes natives pour empêcher que le comportement du record dépende d'un état externe
- un record ne peut pas avoir de blocs d'initialisation d'instances

Il est possible de transformer des classes équivalentes en fonctionnalités en record à partir de Java 16. Comme un record ne peut pas avoir de méthode native, il n'est pas possible de migrer une classe qui possède au moins une méthode native en record.

La classe java.lang.Record est la super-classe de tous les records. Comme cette classe mère est le package java.lang, elle est implicitement importée comme toutes les autres classes du package java.lang.

Il existe donc aussi un cas potentiel d'incompatibilité avec du code existant : celui ou toutes les classes d'un package sont importées en utilisant le joker * avec une classe nommée Record existant dans le package.

Exemple :

```
package fr.jmdoudoux.dej.records;

public class Record {
}
```

Cette classe est utilisée dans une classe d'un autre package. Elle est importée en utilisant le joker *.

Exemple (code Java 14) :

```
package fr.jmdoudoux.dej.java14;

import fr.jmdoudoux.dej.records.*;
```

```
public class TestRecord {

  public static void main(String[] args) {
    Record re;
  }
}
```

Dans ce cas, une erreur de compilation est émise par le compilateur car les classes Record du package fr.jmdoudoux.dej.records et du package java.lang sont importées avec des caractères génériques. Par conséquent, aucune des deux classes n'est prioritaire, et le compilateur génère un message d'erreur lorsqu'il rencontre l'utilisation du simple nom de type Record.

Résultat :

```
C:\java\TestJava14\src>javac fr/jmdoudoux/dej/java14/TestRecord.java
fr\jmdoudoux\dej\java14\TestRecord.java:9: error: reference to Record is ambiguous
    Record re;
    ^
  both class fr.jmdoudoux.dej.records.Record in fr.jmdoudoux.dej.records and class java.lang.
Record in java.lang match
1 error
```

Pour permettre à cet exemple de compiler, l'instruction d'importation peut être modifiée de manière à importer le nom pleinement qualifié de Record : il faut donc préciser explicitement la classe sans utiliser le joker *.

Exemple (code Java 14) :

```
package fr.jmdoudoux.dej.java14;

import fr.jmdoudoux.dej.records.Record;

public class TestRecord {

  public static void main(String[] args) {
    Record re;
  }
}
```

13.3.4. La sérialisation des records

Les records peuvent être sérialisés sous réserve, comme pour tous types, d'implémenter l'interface Serializable.

Exemple (code Java 14) :

```
import java.io.Serializable;

public record Employe(String nom, String prenom) implements Serializable {

  public Employe {
    System.out.println("Invocation constructeur canonique");
  }
}
```

Il est alors possible de sérialiser un record vers un flux.

Exemple (code Java 14) :

```
  public static void main(String[] args) throws IOException {

    var emp = new Employe("nom1", "prenom1");

    try (var oos = new ObjectOutputStream(new FileOutputStream("employe.bin"))) {
      oos.writeObject(emp);
    }
  }
```

Ou d'obtenir une instance d'un record à partir d'un flux.

Exemple (code Java 14) :

```
  public static void main(String[] args) throws IOException, ClassNotFoundException {

    try (var ois = new ObjectInputStream(new FileInputStream("employe.bin"))) {
      var emp = (Employe) ois.readObject();
      System.out.println(emp);
    }
  }
```

La valeur de l'attribut serialVersionUID d'un record est toujours égale à zéro quel que soit les composants contenus dans le record.

Exemple (code Java 14) :

```
  public static void main(String[] args) {

    var emp = new Employe("nom1", "prenom1");

    long serialID = ObjectStreamClass.lookup(emp.getClass()).getSerialVersionUID();
    System.out.println(serialID);
```

Résultat :

```
0
```

La valeur de l'attribut serialVersionUID n'est pas vérifiée lors de désérialisation d'un record.

Le mécanisme de sérialisation traite les instances de type record de manière différente des autres classes et est volontairement très simple dans ce cas :

- la sérialisation d'un record repose uniquement sur l'état des composants
- la désérialisation d'un objet record n'utilise que le constructeur canonique

Cela rend la sérialisation des records plus fiable, notamment si des contrôles sont effectuées sur les valeurs fournies en paramètres du constructeur.

Une première conséquence est qu'il n'est pas possible de personnaliser la sérialisation d'un record. Le contenu de la sérialisation ne comprend que l'état des composants du record. De ce fait, il est pas possible de personnaliser le processus de sérialisation/désérialisation d'un record en utilisant les méthodes writeObject(), readObject(), readObjectNoData(), writeExternal(), ou readExternal().

La de-sérialisation d'une instance de classe lit les données, créé une nouvelle instance en utilisant le constructeur par défaut et utilise l'API Reflexion pour affecter les valeurs lues à l'instance. Ce processus n'est pas sécurisé car il n'y a possibilité de valider les données affectées. Il pourrait par exemple être possible d'obtenir une instance qu'il ne serait pas possible de créer en utilisant les membres de la classe.

La de-sérialisation d'un record utilise un mécanisme différent : l'instance obtenue est créée en utilisant le constructeur canonique en lui passant en paramètre les valeurs lues. Il est alors possible de valider ces données dans le constructeur. Les instances obtenues sont donc dans ce cas nécessairement valides.

Exemple (code Java 14) :

```
public class SerializeRecord {
  public static void main(String[] args) throws IOException, ClassNotFoundException {
    Employe emp = new Employe("nom1", "prenom1");
    System.out.println("Serialise : " + emp);

    try (var oos = new ObjectOutputStream(new FileOutputStream("employe.bin"))) {
      oos.writeObject(emp);
    }

    try (var ois = new ObjectInputStream(new FileInputStream("employe.bin"))) {
      emp = (Employe) ois.readObject();
      System.out.println("Deserialise : " + emp);
    }
  }
}
```

Résultat :

```
Invocation constructeur canonique
Serialise : Employe[nom=nom1, prenom=prenom1]
Invocation constructeur canonique
Deserialise : Employe[nom=nom1, prenom=prenom1]
```

Le constructeur est invoqué une première fois pour créer l'instance sérialisée et une seconde fois pour créer l'instance désérialisée.

Si un composant est ajouté au record et qu'une instance est obtenue par la sérialisation précédente alors la valeur du composant ajoutée passée en paramètre du constructeur canonique est sa valeur par défaut.

Exemple (code Java 14) :

```
public record Employe(String nom, String prenom, String role) implements Serializable {

  public Employe {
    System.out.println("Invocation constructeur canonique");
  }
}
```

Exemple (code Java 14) :

```
public class SerializeRecord {

  public static void main(String[] args) throws IOException, ClassNotFoundException {

    try (var ois = new ObjectInputStream(new FileInputStream("employe.bin"))) {
      Employe emp = (Employe) ois.readObject();
      System.out.println("Deserialise : " + emp);
    }
  }
}
```

Résultat :

```
Invocation constructeur canonique
Deserialise : Employe[nom=nom1, prenom=prenom1, role=null]
```

Si un composant est retiré au record et qu'une instance est obtenue par la sérialisation précédente alors la valeur du composant retiré est ignorée.

Exemple (code Java 14) :

```
public record Employe(String nom) implements Serializable {

  public Employe {
    System.out.println("Invocation constructeur canonique");
  }
}
```

Exemple (code Java 14) :

```
public class SerializeRecord {

  public static void main(String[] args) throws IOException, ClassNotFoundException {

    try (var ois = new ObjectInputStream(new FileInputStream("employe.bin"))) {
      Employe emp = (Employe) ois.readObject();
      System.out.println("Deserialise : " + emp);
    }
  }
}
```

Résultat :

```
Invocation constructeur canonique
Deserialise : Employe[nom=nom1]
```

Avec ces mécanismes pour gérer l'ajout ou le retrait d'un composant, la valeur du champ serialVersionUID n'est pas utile lors de la désérialisation d'un record.

13.3.5. L'introspection sur les records

Deux nouvelles méthodes ont été ajoutées à la classe java.lang.Class. en relation avec les records :

- la méthode isRecord() retourne un booléen qui indique si la classe est un record
- la méthode getRecordComponents() renvoie un tableau de type java.lang.reflect.RecordComponent. La classe RecordComponent permet d'obtenir des informations sur un des composants d'un record notamment : le nom, le type, les annotations, l'accesseur, ...

Exemple (code Java 14) :

```
  public static void main(String[] args) {
    var emp = new Employe("nom1", "prenom1");
    Class clazz = emp.getClass();
    System.out.println(clazz.isRecord());
    RecordComponent[] components = clazz.getRecordComponents();
    for (RecordComponent rc : components) {
      System.out.println(rc.getType().getName()+ " "+rc.getName());
    }
  }
```

Résultat :

```
true
java.lang.String nom
java.lang.String prenom
```

Les champs correspondant aux composants d'un record sont final et ne peuvent pas être modifiés par réflexion : toute tentative lève une exception de type IllegalAccessException.

Exemple (code Java 15) :

```
package fr.jmdoudoux.dej.records;

import java.lang.reflect.Field;

public class TestRecord {
  public static void main(String[] args) throws Exception {
    MonRecord mr = new MonRecord("nom1");

    Field nomField = mr.getClass().getDeclaredField("nom");
    nomField.setAccessible(true);
    nomField.set(mr, "nom2");
  }
}
```

Résultat :

```
Exception in thread "main" java.lang.IllegalAcces
sException: Can not set
final java.lang.String field fr.jmdoudoux.dej.records.MonRecord.nom to java.lang.String
        at java.base/jdk.internal.reflect.UnsafeFieldAccessorImpl.throwFinalFieldIllegalAccessE
xception(UnsafeFieldAccessorImpl.java:76)
        at java.base/jdk.internal.reflect.UnsafeFieldAccessorImpl.throwFinalFieldIllegalAccessE
xception(UnsafeFieldAccessorImpl.java:80)
        at java.base/jdk.internal.reflect.UnsafeQualifiedObjectFieldAccessorImpl.set(UnsafeQual
ifiedObjectFieldAccessorImpl.java:79)
        at java.base/java.lang.reflect.Field.set(Field.java:793)
        at TestJava15/fr.jmdoudoux.dej.records.TestRecord.main(TestRecord.java:11)
```

Partie 2 : Les API de base

Le JDK fournit un certain nombres d'API de base. Cette partie contient les chapitres suivants :

- Les collections : propose une revue des classes fournies par le JDK pour gérer des ensembles d'objets
- Les flux d'entrée/sortie : explore les classes utiles à la mise en oeuvre d'un des mécanismes de base pour échanger des données
- NIO 2 : détaille l'API FileSystem qui facilite l'utilisation de systèmes de fichiers
- La sérialisation : ce procédé offre un mécanisme standard pour transformer l'état d'un objet en un flux de données qui peut être rendu persistant ou échangé sur le réseau pour permettre de recréer un objet possédant le même état.
- L'interaction avec le réseau : propose un aperçu des API fournies par Java pour utiliser les fonctionnalités du réseau dans les applications
- L'internationalisation : traite d'une façon pratique de la possibilité d'internationaliser une application
- Les composants Java beans : examine comment développer et utiliser des composants réutilisables
- Le logging : indique comment mettre en oeuvre deux API pour la gestion des logs : Log4J du projet open source Jakarta et l'API Logging du JDK 1.4
- L'API Stream : L'API Stream permet au travers d'un approche fonctionnelle de manipuler des données d'une source dans le but de produire un résultat. Les traitements sont exprimés de manière déclarative et peuvent être exécutés au besoin en parallèle.
- Les expressions régulières : Ce chapitre détaille l'utilisation et la mise en oeuvre des expressions régulières avec les API du JDK

14. Les collections

Chapitre 14

Niveau : 🟩🟩⬜⬜⬜⬜ Elémentaire

Les collections sont des objets qui permettent de gérer des ensembles d'objets. Ces ensembles de données peuvent être définis avec plusieurs caractéristiques : la possibilité de gérer des doublons, de gérer un ordre de tri, etc. ...

Une collection est un regroupement d'objets qui sont désignés sous le nom d'éléments.

L'API Collections propose un ensemble d'interfaces et de classes dont le but est de stocker de multiples objets. Elle propose quatre grandes familles de collections, chacune définie par une interface de base :

- List : collection d'éléments ordonnés qui accepte les doublons
- Set : collection d'éléments non ordonnés par défaut qui n'accepte pas les doublons
- Map : collection sous la forme d'une association de paires clé/valeur
- Queue et Deque : collections qui stockent des éléments dans un certain ordre avant qu'ils ne soient extraits pour traitement

Ce chapitre contient plusieurs sections :

- Présentation du framework collection
- Les interfaces des collections
- Les collections de type List : les listes
- Les collections de type Set : les ensembles
- Les collections de type Map : les associations de type clé/valeur
- Les collections de type Queue : les files
- Le tri des collections
- Les algorithmes
- Les exceptions du framework

14.1. Présentation du framework collection

Les tableaux ne peuvent pas répondre à tous les besoins de stockage d'un ensemble d'objets et surtout ils manquent de fonctionnalités. La large diversité d'implémentations proposées par l'API Collections de Java permet de répondre à la plupart des besoins.

Avant Java 1.2 qui a introduit l'API Collections, seules quelques classes du package java.util permettaient de stocker et de gérer des éléments : Array, Vector, Stack, Hashtable, Properties et BitSet. L'interface Enumeration permet de parcourir le contenu de ces objets.

L'API Collections propose de structurer et de définir un ensemble d'interfaces et de classes de type collection. Les collections sont des conteneurs qui permettent de regrouper des objets en une seule entité.

Java propose l'API Collections qui offre un socle riche et des implémentations d'objets de type collection enrichies au fur et à mesure des versions de Java.

L'API Collections possède deux grandes familles chacune définies par une interface :

- java.util.Collection : pour gérer un groupe d'objets
- java.util.Map : pour gérer des éléments de type paires de clé/valeur

Une collection permet de stocker un groupe d'éléments en respectant certaines fonctionnalités selon l'implémentation : de base, elle permet d'ajouter, de supprimer, d'obtenir et de parcours ses éléments.

Les interfaces et les classes de l'API Collections qui ne proposent pas de gestion des accès concurrents sont dans le package java.util. Java 5 propose plusieurs collections dans le package java.util.concurrent telles que CopyOnWriteArrayList, ConcurrentHashMap ou CopyOnWriteArraySet qui permettent des modifications lors de leur parcours.

Les fonctionnalités des collections sont définies dans cinq interfaces de base : Collection, List, Set, Map, Queue.

Plusieurs interfaces spécialisent certaines fonctionnalités particulières :

- SortedSet
- NavigableSet
- SortedMap
- NavigableMap
- ConcurrentMap
- ConcurrentNavigableMap
- BlockingQueue
- Deque
- BlockingDeque

Elle définit plusieurs classes abstraites qui sont les classes mères de plusieurs implémentations : AbstractCollection, AbstractSet, AbstractList, AbstractSequentialList, AbstractQueue, AbstractMap.

Elle propose plusieurs implémentations à usage généraliste : HashSet, TreeSet, LinkedHashSet, ArrayList, ArrayDeque, LinkedList, PriorityQueue, HashMap, TreeMap, LinkedHashMap.

Elle propose également plusieurs implémentations pour un usage spécifique : WeakHashMap, IdentityHashMap, CopyOnWriteArrayList, CopyOnWriteArraySet, EnumSet, EnumMap.

A partir de Java 5, plusieurs implémentations permettent l'utilisation de collections de manière concurrente dans un environnement multithread : ConcurrentLinkedQueue, LinkedBlockingQueue, ArrayBlockingQueue, PriorityBlockingQueue, DelayQueue, SynchronousQueue, LinkedBlockingDeque, ConcurrentHashMap, ConcurrentSkipListSet, ConcurrentSkipListMap.

	Utilisation générale	Utilisation spécifique	Gestion des accès concurrents
List	ArrayList LinkedList	CopyOnWriteArrayList	Vector Stack CopyOnWriteArrayList
Set	HashSet TreeSet LinkedHashSet	CopyOnWriteArraySet EnumSet	CopyOnWriteArraySet ConcurrentSkipListSet
Map	HashMap TreeMap LinkedHashMap	WeakHashMap IdentityHashMap EnumMap	Hashtable ConcurrentHashMap ConcurrentSkipListMap
Queue	LinkedList ArrayDeque		ConcurrentLinkedQueue LinkedBlockingQueue

	PriorityQueue		ArrayBlockingQueue PriorityBlockingQueue DelayQueue SynchronousQueue LinkedBlockingDeque

Elle définit enfin :

- deux interfaces pour le parcours de certaines collections : Iterator et ListIterator.
- une interface et une classe pour permettre le tri de certaines collections : Comparable et Comparator
- des classes utilitaires : Arrays, Collections

Le framework Collections propose plusieurs implémentations possédant chacune un comportement et des fonctionnalités particulières.

Collection	Ordonné	Accès direct	Clé / valeur	Doublons	Null	Thread Safe
ArrayList	Oui	Oui	Non	Oui	Oui	Non
LinkedList	Oui	Non	Non	Oui	Oui	Non
HashSet	Non	Non	Non	Non	Oui	Non
TreeSet	Oui	Non	Non	Non	Non	Non
HashMap	Non	Oui	Oui	Non	Oui	Non
TreeMap	Oui	Oui	Oui	Non	Non	Non
Vector	Oui	Oui	Non	Oui	Oui	Oui
Hashtable	Non	Oui	Oui	Non	Non	Oui
Properties	Non	Oui	Oui	Non	Non	Oui
Stack	Oui	Non	Non	Oui	Oui	Oui
CopyOnWriteArrayList	Oui	Oui	Non	Oui	Oui	Oui
ConcurrentHashMap	Non	Oui	Oui	Non	Non	Oui
CopyOnWriteArraySet	Non	Non	Non	Non	Oui	Oui

Pour combler le manque d'objets adaptés, la version 2 du J.D.K. apporte un framework complet pour gérer les collections. Cette bibliothèque contient un ensemble de classes et interfaces. Elle fournit également un certain nombre de classes abstraites qui implémentent partiellement certaines interfaces.

Les interfaces à utiliser par des objets qui gèrent des collections sont :

- Collection : interface qui est implémentée par la plupart des objets qui gèrent des collections
- Map : interface qui définit des méthodes pour des objets qui gèrent des collections sous la forme clé/valeur
- Set : interface pour des objets qui n'autorisent pas de doublons dans l'ensemble
- List : interface pour des objets qui autorisent des doublons et un accès direct à un élément
- SortedSet : interface qui étend l'interface Set et permet d'ordonner l'ensemble
- SortedMap : interface qui étend l'interface Map et permet d'ordonner l'ensemble

Certaines méthodes définies dans ces interfaces sont dites optionnelles : leur définition est donc obligatoire mais si l'opération n'est pas supportée alors la méthode doit lever une exception particulière. Ceci permet de réduire le nombre d'interfaces et de répondre au maximum de cas.

Le framework propose plusieurs objets qui implémentent ces interfaces et qui peuvent être directement utilisés :

- HashSet : Hashtable qui implémente l'interface Set
- TreeSet : arbre qui implémente l'interface SortedSet
- ArrayList : tableau dynamique qui implémente l'interface List
- LinkedList : liste doublement chaînée (parcours de la liste dans les deux sens) qui implémente l'interface List
- HashMap : Hashtable qui implémente l'interface Map
- TreeMap : arbre qui implémente l'interface SortedMap

Le framework définit aussi des interfaces pour faciliter le parcours des collections et leur tri :

- Iterator : interface pour le parcours des collections
- ListIterator : interface pour le parcours des listes dans les deux sens et pour modifier les éléments lors de ce parcours
- Comparable : interface pour définir un ordre de tri naturel pour un objet
- Comparator : interface pour définir un ordre de tri quelconque

Deux classes existantes dans les précédentes versions du JDK ont été modifiées pour implémenter certaines interfaces du framework :

- Vector : tableau à taille variable qui implémente maintenant l'interface List
- Hashtable : table de hachage qui implémente maintenant l'interface Map

Le framework propose la classe Collections qui contient de nombreuses méthodes statiques pour réaliser certaines opérations sur une collection. Plusieurs méthodes unmodifiableXXX() (où XXX représente une interface d'une collection) permettent de rendre une collection non modifiable. Plusieurs méthodes synchronizedXXX() permettent d'obtenir une version synchronisée d'une collection pouvant ainsi être manipulée de façon sûre par plusieurs threads. Enfin plusieurs méthodes permettent de réaliser des traitements sur la collection : tri et duplication d'une liste, recherche du plus petit et du plus grand élément, etc. ...

Le framework fournit plusieurs classes abstraites qui proposent une implémentation partielle d'une interface pour faciliter la création d'une collection personnalisée : AbstractCollection, AbstractList, AbstractMap, AbstractSequentialList et AbstractSet.

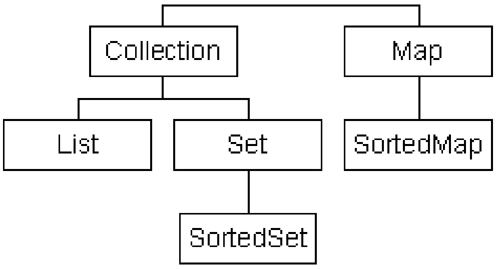
Les objets du framework stockent toujours des références sur les objets contenus dans la collection et non les objets eux-mêmes. Ce sont obligatoirement des objets qui doivent être ajoutés dans une collection. Il n'est pas possible de stocker directement des types primitifs : il faut impérativement encapsuler ces données dans des wrappers.

Toutes les classes de gestion de collections du framework ne sont pas synchronisées : elles ne prennent pas en charge les traitements multithreads. Le framework propose des méthodes pour obtenir des objets de gestion de collections qui prennent en charge cette fonctionnalité. Les classes Vector et Hashtable étaient synchronisées mais l'utilisation d'une collection ne se fait généralement pas dans ce contexte. Pour réduire les temps de traitement dans la plupart des cas, elles ne sont pas synchronisées par défaut.

Lors de l'utilisation de ces classes, il est préférable de stocker la référence de ces objets sous la forme d'une interface qu'ils implémentent plutôt que sous leur forme objet. Ceci rend le code plus facile à modifier si le type de l'objet qui gère la collection doit être changé.

14.2. Les interfaces des collections

Le framework de Java 2 définit 6 interfaces en relation directe avec les collections qui sont regroupées dans deux arborescences :



Le JDK ne fournit pas de classe qui implémente directement l'interface Collection.

Le tableau ci-dessous présente les différentes classes qui implémentent les interfaces de bases Set, List et Map :

	Set collection d'éléments uniques	List collection avec doublons	Map collection sous la forme clé/valeur
Tableau redimensionnable		ArrayList, Vector (JDK 1.1)	
Arbre	TreeSet		TreeMap
Liste chaînée		LinkedList	
Collection utilisant une table de hachage	HashSet		HashMap, Hashtable (JDK 1.1)
Classes du JDK 1.1		Stack	

Pour gérer toutes les situations de façon simple, certaines méthodes peuvent être définies dans une interface comme «optionnelles ». Pour celles-ci, les classes qui implémentent une telle interface, ne sont pas obligées d'implémenter du code qui réalise un traitement mais simplement lève une exception si cette fonctionnalité n'est pas supportée. Le nombre d'interfaces est ainsi grandement réduit.

Cette exception est du type UnsupportedOperationException. Pour éviter de protéger tous les appels de méthodes d'un objet gérant les collections dans un bloc try-catch, cette exception hérite de la classe RuntimeException.

Toutes les classes fournies par le J.D.K. qui implémentent une des interfaces héritant de Collection implémentent toutes les opérations optionnelles.

14.2.1. L'interface Collection

L'interface Collection, ajoutée à Java 1.2, définit des méthodes pour des objets qui gèrent des éléments d'une façon assez générale. Elle est la super interface de plusieurs interfaces du framework.

Plusieurs classes qui gèrent une collection implémentent une interface qui hérite de l'interface Collection. Cette interface est une des deux racines de l'arborescence des collections.

Cette interface représente un minimum commun pour les objets qui gèrent des collections : ajout d'éléments, suppression d'éléments, vérification de la présence d'un objet dans la collection, parcours de la collection et quelques opérations diverses sur la totalité de la collection.

Ce tronc commun permet entre autres de définir pour chaque objet gérant une collection, un constructeur pour cet objet demandant un objet de type Collection en paramètre. La collection est ainsi initialisée avec les éléments contenus dans la collection fournie en paramètre.

Il existe de nombreuses implémentations qui proposent différentes fonctionnalités : support des doublons ou non, tri des éléments ou non, support des null.

L'API Collections ne propose pas d'implémentation directe de cette interface : elle propose des implémentations pour des interfaces filles qui définissent les fonctionnalités des grandes familles de collections : List, Set, Map, Queue.

Elle hérite de l'interface Iterable depuis Java 5.

Chaque implémentation de l'interface Collection devrait fournir au moins deux constructeurs :

- un constructeur par défaut (sans argument)
- un constructeur qui attend en paramètre un objet de type collection qui va créer une collection contenant les éléments de la collection fournie en paramètre

L'interface Collection définit plusieurs méthodes :

Méthode	Rôle
boolean add(E e)	Ajouter un élément à la collection (optionnelle)
boolean addAll(Collection<? extends E> c)	Ajouter tous les éléments de la collection fournie en paramètre dans la collection (optionnelle)
void clear()	Supprimer tous les éléments de la collection (optionnelle)
boolean contains(Object o)	Retourner un booléen qui précise si l'élément est présent dans la collection
boolean containsAll(Collection<?> c)	Retourner un booléen qui précise si tous les éléments fournis en paramètres sont présents dans la collection
boolean equals(Object o)	Vérifier l'égalité avec la collection fournie en paramètre
int hashCode()	Retourner la valeur de hachage de la collection
boolean isEmpty()	Retourner un booléen qui précise si la collection est vide
Iterator<E> iterator()	Retourner un Iterator qui permet le parcours des éléments de la collection
boolean remove(Object o)	Supprimer un élément de la collection s'il est présent (optionnelle)
boolean removeAll(Collection<?> c)	Supprimer tous les éléments fournis en paramètres de la collection s'ils sont présents (optionnelle)
boolean retainAll(Collection<?> c)	Ne laisser dans la collection que les éléments fournis en paramètres : les autres éléments sont supprimés (optionnelle). Elle renvoie un booléen qui précise si le contenu de la collection a été modifié
int size()	Retourner le nombre d'éléments contenus dans la collection
Object[] toArray()	Retourner un tableau contenant tous les éléments de la collection
<T> T[] toArray(T[] a)	Retourner un tableau typé de tous les éléments de la collection

Attention : il ne faut pas ajouter dans une collection une référence à la collection elle-même.

Certaines méthodes de cette interface peuvent lever une exception de type UnsupportedOperationException car leur implémentation est optionnelle : add(), addAll(), remove(), removeAll, retainAll() et clear(). Cette exception peut aussi être levée si l'opération n'a aucune influence sur l'état de la collection.

Chaque implémentation est libre de :

- gérer ou non les accès concurrents
- utiliser l'algorithme de son choix pour tester l'égalité d'un élément (equals(), hashCode()/equals(), ...)
- utiliser à son avantage les fonctionnalités proposées par les éléments (implémentation de Comparable, ...)

14.2.2. L'interface Iterator

Cette interface définit des méthodes pour des objets capables de parcourir les données d'une collection.

La définition de cette nouvelle interface par rapport à l'interface Enumeration a été justifiée par l'ajout de la fonctionnalité de suppression et la réduction des noms de méthodes.

Méthode	Rôle
boolean hasNext()	Indiquer s'il reste au moins un élément à parcourir dans la collection
Object next()	Renvoyer le prochain élément dans la collection
void remove()	Supprimer le dernier élément parcouru

La méthode hasNext() est équivalente à la méthode hasMoreElements() de l'interface Enumeration.

La méthode next() est équivalente à la méthode nextElement() de l'interface Enumeration.

La méthode next() lève une exception de type NoSuchElementException si elle est appelée alors que la fin du parcours des éléments est atteinte. Pour éviter la levée de cette exception, il suffit d'appeler la méthode hasNext() et selon le résultat de conditionner l'appel à la méthode next().

Exemple (code Java 1.2) :

```
Iterator iterator = collection.iterator();
while (iterator.hasNext()) {
  System.out.println("objet = "+iterator.next());
}
```

La méthode remove() permet de supprimer l'élément renvoyé par le dernier appel à la méthode next(). Il est ainsi impossible d'appeler la méthode remove() sans un appel correspondant à next() : on ne peut pas appeler deux fois de suite la méthode remove().

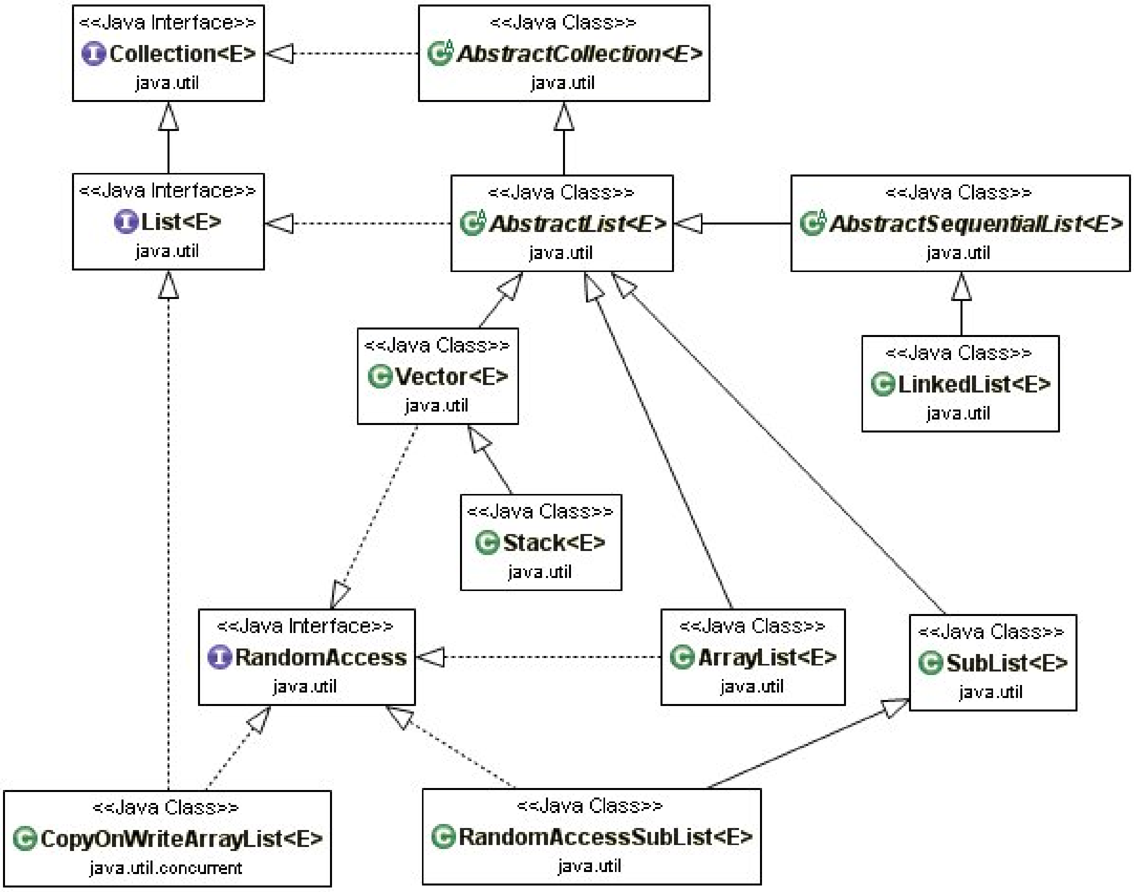
Exemple (code Java 1.2) : suppression du premier élément

```
      Iterator iterator = collection.iterator();
      if (iterator.hasNext()) {
            iterator.next();
            itérator.remove();
      }
```

Si aucun appel à la méthode next() ne correspond à celui de la méthode remove(), une exception de type IllegalStateException est levée

14.3. Les collections de type List : les listes

Une collection de type List est une collection simple et ordonnée d'éléments qui autorise les doublons. La liste étant ordonnée, un élément peut être accédé à partir de son index.



Implémentation	Rôle
java.util.Vector<E>	Une implémentation thread-safe fournie depuis Java 1.0
java.util.Stack<E>	Une implémentation d'une pile : elle hérite de la classe Vector et fournit des opérations pour un comportement de type LIFO (Last In First Out)
java.util.ArrayList<E>	Une implémentation qui n'est pas synchronized, donc à n'utiliser que dans un contexte monothread
java.util.LinkedList<E>	Une implémentation qui n'est pas synchronized d'une liste doublement chaînée. Les insertions de nouveaux éléments sont très rapides
java.util.concurrent.CopyOnWriteArrayList<E>	Une variante thread-safe de la classe ArrayList dans laquelle toutes les opérations de modification du contenu de la liste recréent une nouvelle copie du tableau utilisé pour stocker les éléments de la collection

Plusieurs classes de l'API JMX implémentent l'interface List : javax.management.AttributeList, javax.management.relation.RoleList et javax.management.relation.RoleUnresolvedList.

14.3.1. L'interface List

Cette interface, ajoutée à Java 1.2, étend l'interface Collection.

Une collection de type List permet :

- de contenir des doublons
- d'interagir avec un élément de la collection en utilisant sa position
- d'insérer des éléments null

Pour les listes, une interface particulière est définie pour permettre le parcours dans les deux sens de la liste et réaliser des mises à jour : l'interface ListIterator

L'interface List définit plusieurs méthodes qui permettent un accès aux éléments de la liste à partir d'un index, de gérer les éléments, de rechercher la position d'un élément, d'obtenir une liste partielle (sublist) et d'obtenir des Iterator :

Méthode	Rôle
void add(int index, E e)	Ajouter un élément à la position fournie en paramètre
boolean addAll(int index, Collection<? extends E> c)	Ajouter des éléments à la position fournie en paramètre
E get(int index)	Retourner l'élément à la position fournie en paramètre
int indexOf(Object o)	Retourner la première position dans la liste du premier élément fourni en paramètre. Elle renvoie -1 si l'élément n'est pas trouvé
int lastIndexOf(Object o)	Retourner la dernière position dans la liste du premier élément fourni en paramètre. Elle renvoie -1 si l'élément n'est pas trouvé
ListIterator<E> listIterator()	Renvoyer un Iterator positionné sur le premier élément de la liste
ListIterator<E> listIterator(int indx)	Renvoyer un Iterator positionné sur l'élément dont l'index est fourni en paramètre
E remove(int index)	Supprimer l'élément à la position fournie en paramètre
E set(int index, E e)	Remplacer l'élément à la position fournie en paramètre
List<E> subList(int fromIndex, int toIndex)	Obtenir une liste partielle de la collection contenant les éléments compris entre les index fromIndex inclus et toIndex exclus fournis en paramètres

La collection de type List obtenue en invoquant la méthode subList() est liée à la collection qui a permis sa création. Une modification faite dans la sous liste est reportée dans la liste originelle. Par contre, si un élément est ajouté ou supprimé dans la liste originelle alors une exception de type ConcurrentModificationException est levée lors d'une utilisation de la sous liste

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.ArrayList;
import java.util.List;

public class TestSubList {
  public static void afficherListe(final String nom,
    final List<String> sousListe) {
    int i = 0;
    for (String element : sousListe) {
      System.out.format("%s  %2d : %s\n", nom, i, element);
      i++;
    }
  }

  public static void main(final String[] args) {
    List<String> liste = new ArrayList<String>();
    liste.add("1");
    liste.add("2");
    liste.add("3");
    liste.add("4");
    liste.add("5");

    List<String> sousListe = liste.subList(1, 4);
    afficherListe("sous liste", sousListe);
    System.out.println("");

    sousListe.remove(1);
    afficherListe("liste",liste);
    System.out.println("");
    afficherListe("sous liste", sousListe);

    System.out.println("");
    liste.remove(1);
    afficherListe("liste", liste);
    System.out.println("");

    afficherListe("sous liste", sousListe);
    System.out.println("");
  }
}
```

Résultat :

```
sous liste   0 : 2
sous liste   1 : 3
sous liste   2 : 4

liste   0 : 1
liste   1 : 2
liste   2 : 4
liste   3 : 5

sous liste   0 : 2
sous liste   1 : 4

liste   0 : 1
liste   1 : 4
liste   2 : 5

Exception in thread "main" java.util.ConcurrentModificationException
    at java.util.SubList.checkForComodification(Unknown Source)
    at java.util.SubList.listIterator(Unknown Source)
    at java.util.AbstractList.listIterator(Unknown Source)
    at java.util.SubList.iterator(Unknown Source)
    at fr.jmdoudoux.dej.collections.TestSubList.afficherListe(TestSubList.java:11)
    at fr.jmdoudoux.dej.collections.TestSubList.main(TestSubList.java:39)
```

Remarque : il est préférable d'utiliser un Iterator pour parcourir les éléments d'une collection de type List plutôt que de faire une boucle sur son nombre d'éléments et d'obtenir chaque élément en utilisant son indice.

Le framework propose des classes qui implémentent l'interface List : Vector, ArrayList, LinkedList et CopyOnWriteArrayList.

14.3.2. La classe Vector

La classe Vector, présente depuis Java 1.0, est un tableau dont la taille peut varier selon le nombre d'éléments qu'il contient.

Lors de la création d'une instance de type Vector, il est possible de lui préciser une capacité initiale et une taille d'incrémentation en utilisant la surcharge correspondante du constructeur.

Toutes les méthodes de la classe Vector sont synchronized : elle est donc moins performante que la classe ArrayList car elle est thread-safe.

La classe Vector est antérieure à l'API Collections : elle a été mise à jour ultérieurement pour implémenter l'interface Liste. Il y a de ce fait plusieurs méthodes redondantes comme par exemple les méthodes add() et addElement().

Avant l'API Collections la classe Vector était fréquemment utilisée : il est préférable d'utiliser une des implémentations de l'API Collections.

Les éléments sont stockés dans l'ordre dans lequel ils sont ajoutés dans la collection. Un élément peut être ajouté ou supprimé à n'importe qu'elle position dans la collection.

14.3.3. La classe ArrayList

Les tableaux font partis du langage Java et sont faciles à utiliser mais leur taille ne peut pas varier. La classe ArrayList, ajoutée à Java 1.2, est un tableau d'objets dont la taille est dynamique : elle utilise un tableau dont la taille s'adapte automatiquement au nombre d'éléments de la collection. Cette adaptation a cependant un coût car elle nécessite l'instanciation d'un nouveau tableau et la copie des éléments dans ce nouveau tableau.

Elle hérite de la classe AbstractList donc elle implémente l'interface List.

Le fonctionnement de cette classe est similaire à celui de la classe Vector. La différence avec la classe Vector est que cette dernière est multithread (toutes ses méthodes sont synchronisées). Pour une utilisation dans un thread unique, la synchronisation des méthodes est inutile et coûteuse. Il est alors préférable d'utiliser un objet de la classe ArrayList.

La classe ArrayList est l'implémentation la plus simple de l'interface List. Elle présente plusieurs caractéristiques :

- elle n'est pas thread-safe
- elle utilise un tableau pour stocker ses éléments : le premier élément de la collection possède l'index 0
- l'accès à un élément se fait grâce à son index
- elle implémente toutes les méthodes de l'interface List
- elle autorise l'ajout d'éléments null

La classe ArrayList dispose de plusieurs constructeurs :

Constructeur	Rôle
ArrayList()	Créer une instance vide de la collection avec une capacité initiale de 10
ArrayList(Collection<? extends E> c)	Créer une instance contenant les éléments de la collection fournie en paramètre dans l'ordre obtenu en utilisant son iterator
ArrayList(int initialCapacity)	Créer une instance vide de la collection avec la capacité initiale fournie en paramètre

Elle définit plusieurs méthodes dont les principales sont :

Méthode	Rôle
boolean add(Object)	Ajouter un élément à la fin du tableau
boolean addAll(Collection)	Ajouter tous les éléments de la collection fournie en paramètre à la fin du tableau
boolean addAll(int, Collection)	Ajouter tous les éléments de la collection fournie en paramètre dans la collection à partir de la position précisée
void clear()	Supprimer tous les éléments du tableau
void ensureCapacity(int)	Augmenter la capacité du tableau pour s'assurer qu'il puisse contenir le nombre d'éléments passé en paramètre
Object get(index)	Renvoyer l'élément du tableau dont la position est précisée
int indexOf(Object)	Renvoyer la position de la première occurrence de l'élément fourni en paramètre
boolean isEmpty()	Indiquer si le tableau est vide
int lastIndexOf(Object)	Renvoyer la position de la dernière occurrence de l'élément fourni en paramètre
Object remove(int)	Supprimer dans le tableau l'élément fourni en paramètre
void removeRange(int, int)	Supprimer tous les éléments du tableau de la première position fournie incluse jusqu'à la dernière position fournie exclue

Object set(int, Object)	Remplacer l'élément à la position indiquée par celui fourni en paramètre
int size()	Renvoyer le nombre d'éléments du tableau
void trimToSize()	Ajuster la capacité du tableau sur sa taille actuelle

Chaque instance de type ArrayList possède une capacité qui correspond à la taille du tableau de stockage des éléments : c'est donc le nombre total d'éléments qu'il est possible d'insérer avant d'agrandir le tableau. Cette capacité a donc une relation avec le nombre d'éléments contenus dans la collection : elle est toujours au moins supérieure ou égale à la taille de la collection. La capacité de la collection est automatiquement ajustée selon les besoins lors de l'ajout d'un élément. Cette capacité et le nombre d'éléments de la collection déterminent si le tableau doit être agrandi.

Si un nombre important d'éléments doit être ajouté, il est possible de forcer l'agrandissement de cette capacité avec la méthode ensureCapacity() : elle permet de demander que le tableau puisse au moins accepter le nombre d'éléments fourni en paramètre. Cela peut améliorer les performances en changeant la taille une seule fois si de nombreux éléments doivent être ajoutés plutôt que de changer la taille plusieurs fois selon les besoins. Son usage évite une perte de temps liée au recalcul de la taille de la collection. Un constructeur permet de préciser la capacité initiale.

Lors de l'ajout d'un élément dans la collection, si le tableau de stockage est trop petit alors un nouveau, plus grand, est créé pour contenir les éléments courants plus le nouvel élément. Le temps d'ajout d'un élément n'est donc pas constant. Les temps d'exécution de l'insertion ou de suppression d'un élément à une position quelconque est aussi variable puisque cela peut nécessiter l'adaptation de la position d'autres éléments.

Lors de l'ajout ou du retrait d'un élément, la collection doit réindexer ses éléments. Si la taille de la collection est importante et qu'il y a de nombreux ajouts et suppressions d'éléments alors il est préférable d'utiliser une collection de type LinkedList.

Par défaut, les méthodes de la classe ArrayList ne sont pas synchronized : si plusieurs threads doivent modifier le contenu de la collection, il faut utiliser une instance retournée par la méthode synchronizedList() de la classe Collections.

```
List liste = Collections.synchronizedList(new ArrayList());
```

Les Iterator et les ListIterator de la classe ArrayList sont de type fail-fast : ils peuvent lever une exception de type ConcurrentModificationException si une modification du nombre d'éléments intervient durant le parcours sans utiliser leurs méthodes add() ou remove().

L'API Collections propose deux solutions pour convertir un tableau en ArrayList :

- la méthode asList() de la classe Arrays
- la méthode addAll() de la classe Collections : cette méthode fait une copie des éléments. Les deux objets peuvent alors être modifiés de manière indépendante

La méthode Arrays.asList() est facile à utiliser mais les éléments du tableau et de la liste sont liés. Les modifications faites aux éléments du tableau sont propagées dans la liste. Toute tentative de modification de la liste lève une exception de type UnsupportedOperationException.

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class ArrayToArrayList {
  public static void main(final String[] args) {
    String[] tableau = { "A", "B", "C", "D" };
    List<String> liste = new ArrayList<String>();

    System.out.println("Contenu du tableau");
    for (String str : tableau) {
      System.out.print(" " + str);
    }
    liste = Arrays.asList(tableau);
    System.out.println("\nContenu de la liste");
    for (String str : liste) {
      System.out.print(" " + str);
    }
    System.out.println("\n");

    tableau[0] = "AA";

    System.out.println("\nContenu de la liste");
    for (String str : liste) {
      System.out.print(" " + str);
    }
    liste.add("E");

    System.out.println("\nContenu du tableau");
    for (String str : tableau) {
      System.out.print(" " + str);
    }
  }
}
```

Résultat :

```
Contenu du tableau
 A B C D
Contenu de la liste
 A B C D
Contenu de la liste
 AA B C D

Exception in thread "main" java.lang.UnsupportedOperationException
     at java.util.AbstractList.add(AbstractList.java:148)
     at java.util.AbstractList.add(AbstractList.java:108)
     at fr.jmdoudoux.dej.collections.ArrayToArrayList.main(ArrayToArrayList.java:35)
```

La méthode Collections.addAll() copie les éléments. Les deux objets peuvent alors être modifiés de manière indépendante.

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class ArrayToArrayList {
  public static void main(final String[] args) {
    String[] tableau = { "A", "B", "C", "D" };
    List<String> liste = new ArrayList<String>();

    Collections.addAll(liste, tableau);

    System.out.println("Contenu du tableau");
    for (String str : tableau) {
      System.out.print(" " + str);
    }

    System.out.println("\nContenu de la liste");
    for (String str : liste) {
      System.out.print(" " + str);
    }

    tableau[0] = "AA";
    liste.add("E");

    System.out.println("\nContenu du tableau");
    for (String str : tableau) {
      System.out.print(" " + str);
    }
    System.out.println("\nContenu de la liste");
    for (String str : liste) {
      System.out.print(" " + str);
    }
  }
}
```

Résultat :

```
Contenu du tableau
 A B C D
Contenu de la liste
 A B C D
Contenu du tableau
 AA B C D
Contenu de la liste
 A B C D E
```

14.3.4. Les listes chaînées : la classe LinkedList

La classe LinkedList, ajoutée à Java 1.2, est une implémentation d'une liste doublement chaînée dans laquelle les éléments de la collection sont reliés par des pointeurs. La suppression ou l'ajout d'un élément se fait simplement en modifiant des pointeurs.

Elle hérite de la classe AbstractSequentialList et implémente toutes les méthodes, même celles optionnelles, de l'interface List. Elle implémente l'interface Deque à partir de Java 6.

Elle présente plusieurs caractéristiques :

- elle n'a pas besoin d'être redimensionnée quelque soit le nombre d'éléments qu'elle contient
- elle permet l'ajout d'un élément null.

La classe LinkedList possède plusieurs constructeurs :

Constructeur	Rôle
LinkedList()	Créer une nouvelle instance vide
LinkedList(Collection<? extends E> c)	Créer une nouvelle instance contenant les éléments de la collection fournie en paramètre triés dans l'ordre obtenu par son Iterator

Exemple (code Java 1.2) :

```
      LinkedList listeChainee = new LinkedList();
      listeChainee.add("element 1");
      listeChainee.add("element 2");
      listeChainee.add("element 3");
      Iterator iterator = listeChainee.iterator();
      while (iterator.hasNext()) {
            System.out.println("objet = "+iterator.next());
      }
```

La méthode toString() renvoie une chaîne qui contient tous les éléments de la liste.

Plusieurs méthodes pour ajouter, supprimer ou obtenir le premier ou le dernier élément de la liste permettent d'utiliser cette classe pour gérer une pile ou une file :

Méthode	Rôle
void addFirst(Object)	Insèrer l'objet au début de la liste
void addLast(Object)	Insèrer l'objet à la fin de la liste

Object getFirst()	Renvoyer le premier élément de la liste
Object getLast()	Renvoyer le dernier élément de la liste
Object removeFirst()	Supprimer le premier élément de la liste et renvoie l'élément qui est devenu le premier
Object removeLast()	Supprimer le dernier élément de la liste et renvoie l'élément qui est devenu le dernier

Une liste chaînée gère une collection de façon ordonnée : l'ajout d'un élément peut se faire au début ou à la fin de la collection. L'utilisation d'une LinkedList est plus avantageuse par rapport à une ArrayList lorsque des éléments doivent être ajoutés ou supprimés de la collection en dehors de son début ou de sa fin. Dans ce cas, le temps d'exécution des opérations se fait toujours de manière constant puisqu'elles consistent simplement en la manipulation de pointeurs.

De par les caractéristiques d'une liste chaînée, il n'existe pas de moyen d'obtenir un élément de la liste directement. Pourtant, la méthode contains() permet de savoir si un élément est contenu dans la liste et la méthode get() permet d'obtenir l'élément à la position fournie en paramètre. Il ne faut toutefois pas oublier que ces méthodes parcourent la liste jusqu'à obtention du résultat, ce qui peut être particulièrement gourmand en terme de temps de réponse surtout si la méthode get() est appelée dans une boucle. Pour cette raison, il ne faut surtout pas utiliser la méthode get() pour parcourir la liste.

Une collection de type ArrayList permet un accès direct à un élément dans un temps constant. L'accès direct à un élément d'une collection de type LinkedList est beaucoup moins performant car elle doit parcourir tous les éléments successivement depuis le premier élément jusqu'à l'élément désiré.

Les méthodes de la classe LinkedList ne sont pas synchronized. Si plusieurs threads doivent accéder à la collection avec au moins un d'entre-eux qui modifie la structure de la liste (ajout ou suppression d'un élément) alors il faut créer une instance de type List en invoquant la méthode synchronizedList() de la classe Collections en lui passant l'instance de type List.

List liste = Collections.synchronizedList(new LinkedList());

Les Iterator obtenus en invoquant les méthodes iterator() ou listIterator() sont de type fail-fast : une exception de type ConcurrentModificationException est généralement levée lors du parcours des Iterator si la structure de la collection est modifiée.

L'ajout d'un élément après n'importe quel élément est lié à la position courante lors d'un parcours : pour répondre à ce besoin, l'interface qui permet le parcours de la collection est une sous-classe de l'interface Iterator : l'interface ListIterator.

Comme les Iterator sont utilisés pour faire des mises à jour dans la liste, une exception de type ConcurrentModificationException est levée si un iterator parcourt la liste alors qu'une mise à mises à jour est faite (ajout ou suppression d'un élément dans la liste). Pour gérer facilement cette situation, il est préférable si l'on sait qu'il y a des mises à jour à faire de n'avoir qu'un seul iterator qui soit utilisé.

Exemple (code Java 1.2) :

```
      LinkedList listeChainee = new LinkedList();
      Iterator iterator = listeChainee.iterator();
      listeChainee.add("element 1");
      listeChainee.add("element 2");
      listeChainee.add("element 3");
      while (iterator.hasNext()) {
            System.out.println("objet = "+iterator.next());
      }
```

Résultat :

```
Exception in thread "main" java.util.ConcurrentModificationException
        at java.util.LinkedList$ListItr.checkForComodification(LinkedList.java:761)
        at java.util.LinkedList$ListItr.next(LinkedList.java:696)
        at snippet.Snippet.main(Snippet.java:14)
```

Il existe plusieurs différences entre une ArrayList et une LinkedList :

- une ArrayList stocke ses éléments en interne dans un tableau à taille fixe alors qu'une LinkedList stocke ses éléments dans une liste doublement chaînée
- une ArrayList permet un accès direct à un élément alors qu'une LinkedList doit parcourir ses éléments pour obtenir celui désiré, ce qui est particulièrement contre performant
- le coût de variation de la capacité d'une collection de type ArrayList est important car il implique une copie du tableau de stockage interne de ses éléments
- l'ajout d'un élément en début ou en fin d'une collection de type LinkedList est particulièrement performant et son temps d'exécution est constant dans le temps (LinkedList implémente aussi l'interface Deque)

14.3.5. L'interface ListIterator

L'interface ListIterator définit des fonctionnalités d'un Iterator permettant aussi le parcours en sens inverse de la collection, l'ajout d'un élément ou la modification du courant.

En plus des méthodes définies dans l'interface Iterator dont elle hérite, l'interface ListIterator définit plusieurs méthodes :

Méthode	Rôle
void add(E e)	Ajouter un élément dans la collection
boolean hasPrevious()	Retourner true si l'élément courant possède un élément précédent
int nextIndex()	Retourner l'index de l'élément qui serait retourné en invoquant la méthode next()
E previous()	Retourner l'élément précédent dans la liste
int previousIndex()	Retourner l'index de l'élément qui serait retourné en invoquant la méthode previous()
void set(E e)	Remplacer l'élément courant par celui fourni en paramètre

La méthode add() de cette interface ne retourne pas un booléen indiquant que l'ajout a réussi.

Pour ajouter un élément en début de liste, il suffit d'appeler la méthode add() sans avoir appelé une seule fois la méthode next(). Pour ajouter un élément en fin de la liste, il suffit d'appeler la méthode next() autant de fois que nécessaire pour atteindre la fin de la liste et d'appeler la méthode add(). Plusieurs appels à la méthode add() successifs, ajoutent les éléments à la position courante dans l'ordre d'appel de la méthode add().

Les méthodes set() et remove() agissent sur l'élément courant qui correspond à l'élément obtenu par la dernière invocation de la méthode next() ou previous(). Elles lèvent une exception de type IllegalStateException s'il n'y a pas d'élément courant.

14.3.6. La classe CopyOnWriteArrayList

L'utilisation d'un wrapper synchronized d'une ArrayList n'est pas toujours indiquée lorsqu'il y a beaucoup de lectures car celles-ci sont aussi synchronized dans ce cas, ce qui peut introduire de la contention si plusieurs threads effectuent des lectures concurrentes.

La classe CopyOnWriteArrayList, ajoutée à Java 1.5, est une variante thread-safe de la classe ArrayList dans laquelle toutes les opérations de modification du contenu de la liste recréent une nouvelle copie du tableau utilisé pour stocker les éléments de la collection.

En interne, les opérations de modification du contenu de la collection s'effectuent sur une nouvelle copie du tableau des éléments, ce qui permet à d'autres threads de lire le contenu de la collection sans surcoût de synchronisation.

Elle implémente les interfaces List et RandomAccess.

Elle présente plusieurs caractéristiques :

- Son mode de fonctionnement est généralement coûteux car il crée de nombreux objets lors de la mise à jour du contenu de la collection
- Il est possible d'ajouter des éléments null

Elle possède trois constructeurs :

Constructeur	Rôle
CopyOnWriteArrayList()	Créer une collection vide
CopyOnWriteArrayList(Collection<? extends E> c)	Créer une collection initialisée avec les éléments de la collection fournie en paramètre insérés dans l'ordre de l'itérateur de cette collection
CopyOnWriteArrayList(E[] toCopyIn)	Créer une collection initialisée avec les éléments du tableau fourni en paramètre

La méthode addIfAbsent() permet d'ajouter de manière atomique un élément qui n'appartient pas à la collection.

La méthode addAllAbsent(Collection< ? extends E>) permet d'ajouter de manière atomique les éléments de la collection en paramètre qui n'appartiennent pas déjà à la collection.

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.Iterator;
import java.util.List;
import java.util.Random;
import java.util.concurrent.CopyOnWriteArrayList;

public class TestCopyOnWriteArrayList {
  static Thread modifThread;
  static Thread parcoursThread;
  private static void lancerModifThread(final CopyOnWriteArrayList<String> list) {
    modifThread = new Thread(new Runnable() {
      long compteur = 0;
      @Override
      public void run() {
        while (!Thread.interrupted()) {
          int taille = list.size();
          Random random = new Random();
          if (random.nextBoolean()) {
            if (taille > 1) {
              list.remove(random.nextInt(taille - 1));
            }
          } else {
            if (taille < 10) {
              list.addIfAbsent("Element " + compteur);
            }
          }
          compteur++;
        }
        System.out.println("Arret du thread modif");
      }
    });
    modifThread.start();
  }

  private static void lancerParcoursThread(final List<String> list) {
    parcoursThread = new Thread(new Runnable() {
      @Override
      public void run() {
        while (!Thread.interrupted()) {
          Iterator<String> iter = list.iterator();
          while (iter.hasNext()) {
            String element = iter.next();
            System.out.println(element);
          }
          System.out.println("");
        }
        System.out.println("Arret du thread parcours");
      }
    });
    parcoursThread.start();
  }

  public static void main(final String[] args) {
    CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<String>();

    lancerParcoursThread(list);
    lancerModifThread(list);

    try {
      Thread.sleep(10000);
    } catch (InterruptedException ie) {
      ie.printStackTrace();
    }

    modifThread.interrupt();
    parcoursThread.interrupt();
  }
}
```

Les Iterator créés par cette collection parcourent une copie du tableau au moment de la création de leurs instances. Le contenu de ce tableau ne peut pas être modifié : les méthodes de l'Iterator qui permettent de modifier le contenu de la collection comme la méthode remove() lèvent une exception de type UnsupportedOperationException. Un Iterator ne peut donc pas lever d'exception de type ConcurrentModificationException.

L'utilisation de la classe CopyOnWriteArrayList, s'il y a de nombreuses mises à jour de ses éléments, implique un surcoût de mémoire et de temps d'exécution. Son usage est limité ; un bon exemple d'utilisation de la classe CopyOnWriteArrayList est pour stocker les listeners d'un JavaBean pour lequel il y a beaucoup de lectures et peu d'écritures normalement.

14.3.7. Le choix d'une implémentation de type List

S'il n'y a aucun accès concurrent sur la collection, le choix doit se faire entre les classes ArrayList et LinkedList. Ce choix dépendant de l'utilisation qui sera faite de la collection :

- Si l'ajout ou la suppression d'éléments se font essentiellement à la fin de la collection, alors il faut utiliser la classe ArrayList
- Si les ajouts ou la suppression d'éléments se font à une position aléatoire dans la collection, alors il faut utiliser la classe LinkedList

Un élément peut être accédé directement par son index dans une ArrayList, ce qui n'est pas possible avec une LinkedList sauf pour le premier et le dernier élément.

Si les accès concurrents doivent être gérés alors il y a deux cas de figure :

- Si la collection est peu fréquemment mise à jour alors il faut utiliser la classe copyOnWriteArrayList qui permet des lectures non bloquantes mais les mises à jour impliquent une duplication de la collection
- Sinon il faut utiliser l'instance retournée par la méthode synchronizedList() de la classe Collections en lui passant en paramètre une instance de type ArrayList ou LinkedList. Les accès à la collection sont alors thread safe mais pas concurrents.

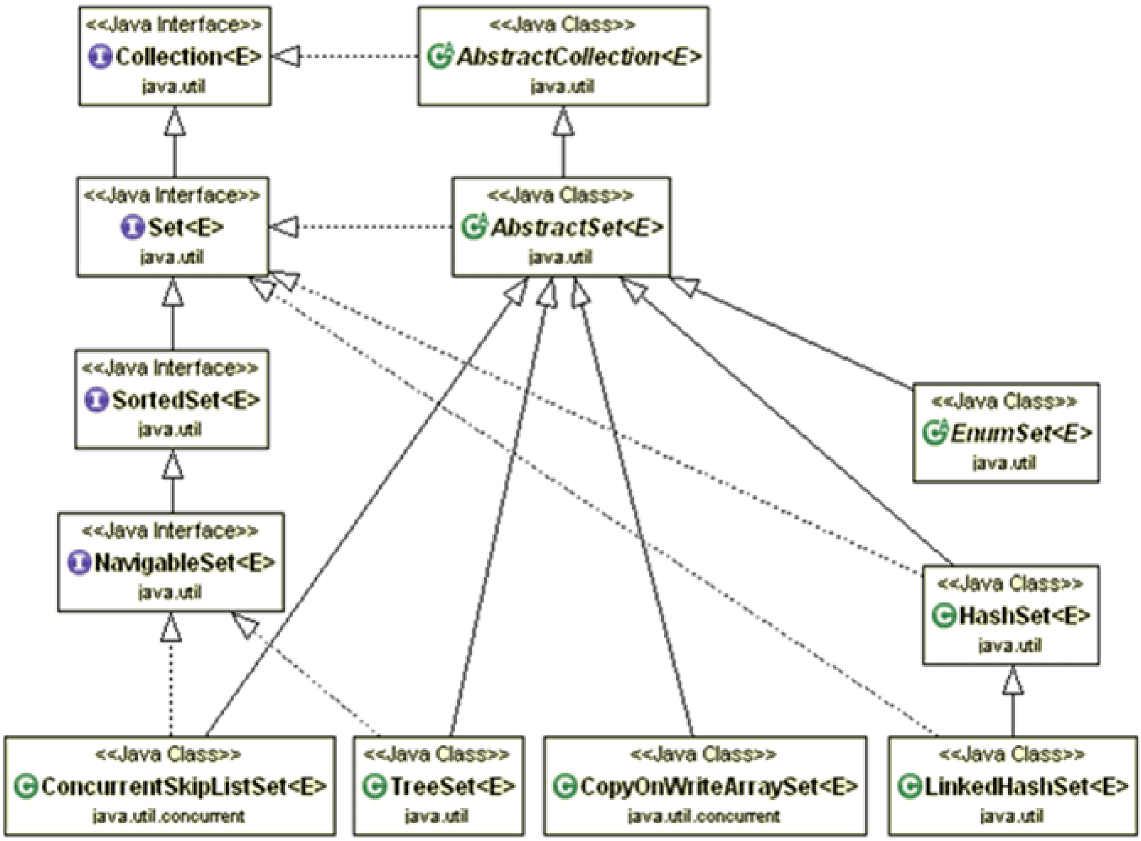
Le tableau ci-dessous compare les performances de certaines fonctionnalités de base de différentes implémentations de type List.

	get	add	contains	next	remove(0)	iterator.remove
ArrayList	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)
LinkedList	O(n)	O(1)	O(n)	O(1)	O(1)	O(1)

	get	add	contains	next	remove(0)	iterator.remove
CopyOnWriteArrayList	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)

14.4. Les collections de type Set : les ensembles

Une collection de type Set ne permet pas l'ajout de doublons ni l'accès direct à un élément de la collection. Les fonctionnalités de base de ce type de collection sont définies dans l'interface java.util.Set.



L'interface Set possède deux interfaces filles : SortedSet et NavigableSet.

L'API Collections propose plusieurs implémentations de l'interface Set:

Implémentation	Rôle
java.util.HashSet<E>	La collection n'est pas thread-safe, il est possible d'ajouter un élément null
java.util.TreeSet<E>	Les éléments sont triés, la collection n'est pas thread-safe, il est impossible d'ajouter un élément null
java.util.concurrent.CopyOnWriteArraySet<E>	La collection est thread-safe : elle créé une nouvelle copie lors de l'invocation de méthode qui modifie le contenu de la collection, se qui rend ces opérations coûteuses.
java.util.EnumSet<E extends Enum<E>>	Tous les éléments de la collection doivent appartenir à la même énumération
java.util.LinkedHashSet<E>	Similaire à la collection HashSet en définissant l'ordre de parcours qui est celui dans lequel les éléments ont été ajoutés dans la collection
java.util.concurrent.ConcurrentSkipListSet<E>	Un ensemble ordonné d'éléments capable de gérer une forte concurrence d'accès

14.4.1. L'interface Set

L'interface Set définit les fonctionnalités d'une collection qui ne peut pas contenir de doublons dans ses éléments.

Les éléments ajoutés dans une collection de type Set doivent réimplémenter leurs méthodes equals() et hashCode(). Ces méthodes sont utilisées lors de l'ajout d'un élément pour déterminer s'il est déjà présent dans la collection. La valeur retournée par hashCode() est recherchée dans la collection :

- si aucun objet de la collection n'a la même valeur de hachage alors l'objet n'est pas encore dans la collection et peut être ajouté
- si un ou plusieurs objets de la collection ont la même valeur de hachage alors la méthode equals() de l'objet à ajouter est invoquée sur chacun des objets pour déterminer si l'objet est déjà présent ou non dans la collection

Le comportement d'une collection de type Set n'est pas spécifié si des objets mutables lui sont ajoutés notamment si des modifications changent le résultat des méthodes equals() et hashCode().

Une collection de type Set peut contenir un objet null mais cela dépend des implémentations. Certaines d'entre-elles ne permettent pas l'ajout de null.

L'interface définit plusieurs méthodes :

Méthode	Rôle
boolean add(E e)	Ajouter l'élément fourni en paramètre à la collection si celle-ci ne le contient pas déjà et renvoyer un booléen qui précise si la collection a été modifiée (l'implémentation de cette opération est optionnelle)
boolean addAll(Collection<? extends E> c)	Ajouter tous les éléments de la collection fournie en paramètre à la collection si celle-ci ne les contient pas déjà et renvoyer un booléen qui précise si la collection a été modifiée (l'implémentation de cette opération est optionnelle)
void clear()	Retirer tous les éléments de la collection (l'implémentation de cette opération est optionnelle)
boolean contains(Object o)	Renvoyer un booléen qui précise si la collection contient l'élément fourni en paramètre
boolean containsAll(Collection<?> c)	Renvoyer un booléen qui précise si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
boolean equals(Object o)	Comparer l'égalité de la collection avec l'objet fourni en paramètre. L'égalité est vérifiée si l'objet est de type Set, que les deux collections ont le même nombre d'éléments et que chaque élément d'une collection est contenu dans l'autre
int hashCode()	Retourner la valeur de hachage de la collection
boolean isEmpty()	Renvoyer un booléen qui précise si la collection est vide
Iterator<E> iterator()	Renvoyer un Iterator sur les éléments de la collection
boolean remove(Object o)	Retirer l'élément fourni en paramètre de la collection si celle-ci le contient et renvoyer un booléen qui précise si la collection a été modifiée (l'implémentation de cette opération est optionnelle)
boolean removeAll(Collection<?> c)	Retirer les éléments fournis en paramètres de la collection si celle-ci les contient et renvoyer un booléen qui précise si la collection a été modifiée. (l'implémentation de cette opération est optionnelle)
boolean retainAll(Collection<?> c)	Retirer tous les éléments de la collection qui ne sont pas dans la collection fournie en paramètre (l'implémentation de cette opération est optionnelle)
int size()	Renvoyer le nombre d'éléments de la collection. Si ce nombre dépasse Integer.MAX_VALUE alors la valeur retournée est MAX_VALUE

Object[] toArray()	Renvoyer un tableau des éléments de la collection
<T> T[] toArray(T[] a)	Renvoyer un tableau des éléments de la collection dont le type est celui fourni en paramètre

Il est possible d'utiliser un Iterator pour parcourir les éléments de la collection.

L'invocation de la méthode add() avec en paramètre un élément déjà présent dans la collection n'aucun effet.

L'interface Set possède deux interfaces filles : SortedSet et NavigableSet.

L'API Collections propose plusieurs implémentations de l'interface Set : ConcurrentSkipListSet, CopyOnWriteArraySet, EnumSet, HashSet, LinkedHashSet et TreeSet.

Il faut choisir judicieusement l'implémentation à utiliser selon ses besoins de performance des méthodes add(), contains(), de l'itération sur la collection et l'ordre dans lequel les éléments sont retournés.

14.4.2. L'interface SortedSet

L'interface SortedSet, ajoutée à Java 1.2, définit les fonctionnalités pour une collection de type Set qui garantit l'ordre ascendant du parcours de ses éléments.

L'interface SortedSet hérite de l'interface Set et propose plusieurs méthodes :

Méthode	Rôle
E first()	Retourner le premier élément de la collection
E last()	Retourner le dernier élément de la collection
SortedSet headSet(E toElement)	Retourner un sous-ensemble des premiers éléments de la collection jusqu'à l'élément fourni en paramètre exclus
SortedSet tailSet(E fromElement)	Retourner un sous-ensemble contenant les derniers éléments de la collection à partir de celui fourni en paramètre inclus
SortedSet subSet(E fromElement, E toElement)	Retourner un sous-ensemble des éléments dont les bornes sont ceux fournis en paramètres. fromElement est inclus et toElement est exclus. Si les deux éléments fournis en paramètres sont les mêmes, la méthode renvoie une collection vide
Comparator< ? super E> comparator()	Renvoyer l'instance de type Comparator associée à la collection ou null s'il n'y en a pas

L'ordre des éléments peut être défini de deux manières :

- l'ordre naturel des éléments qui doivent alors implémenter l'interface Comparable
- une instance de type Comparator qui sera invoquée pour définir l'ordre de tri

L'interface SortedSet ne précise pas comment la collection va utiliser l'une ou l'autre de ces options. Généralement, les implémentations définissent un constructeur particulier qui attend en paramètre une instance de type Comparator. Si une telle instance n'est pas fournie alors c'est l'ordre naturel des objets contenus dans la collection qui est utilisé.

Un collection de type Set utilise la méthode equals() pour vérifier si un élément est déjà présent ou non dans la collection. Une collection de type SortedSet utilise la méthode compareTo() lors de l'utilisation de l'ordre naturel de ses éléments. Il est donc important que l'implémentation des méthodes equals() et compareTo() soient cohérentes.

Java 6 propose deux implémentations de l'interface SortedSet : java.util.TreeSet et java.util.concurrent.ConcurrentSkipListSet.

14.4.3. L'interface NavigableSet

L'interface NavigableSet qui hérite de l'interface SortedSet définit des fonctionnalités qui permettent le parcours de la collection dans l'ordre ascendant ou descendant et d'obtenir des éléments proches d'un autre élément.

Méthode	Rôle
E ceiling(E e)	Retourner le plus petit élément qui soit plus grand ou égal à celui fourni en paramètre. Renvoie null si aucun élément n'est trouvé
Iterator<E> descendingIterator()	Retourner un Iterator qui permet le parcours dans un ordre descendant des éléments de la collection
NavigableSet<E> descendingSet()	Retourner un ensemble parcourable dans le sens inverse de l'ordre de la collection actuelle
E floor(E e)	Retourner le plus grand élément qui soit plus petit ou égal à celui fourni en paramètre. Renvoie null si aucun élément n'est trouvé
SortedSet<E> headSet(E toElement)	Retourner un ensemble qui contient les éléments de la collection qui sont strictement plus petits que celui fourni en paramètre
NavigableSet<E> headSet(E toElement, boolean inclusive)	Retourner un ensemble parcourable qui contient les éléments de la collection qui sont strictement plus petits (ou plus petits ou égaux si le paramètre inclusive vaut true) que celui fourni en paramètre
E higher(E e)	Retourner le plus petit élément qui soit strictement plus grand que celui fourni en paramètre. Renvoie null si aucun élément n'est trouvé
Iterator<E> iterator()	Retourner un Iterator qui permet le parcours des éléments dans l'ordre ascendant
E lower(E e)	Retourner le plus grand élément qui soit strictement plus petit que celui fourni en paramètre. Renvoie null si aucun élément n'est trouvé
E pollFirst()	Retourner le premier élément et le retirer de la collection. Renvoie null si la collection est vide
E pollLast()	Retourner le dernier élément et le retirer de la collection. Renvoie null si la collection est vide
NavigableSet<E> subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)	Retourner un sous-ensemble parcourable qui contient les éléments compris entre les deux éléments fournis en paramètres. Un booléen permet de préciser si chaque borne doit être incluse ou non.
SortedSet<E> subSet(E fromElement, E toElement)	Retourner un sous-ensemble qui contient les éléments compris entre le premier fourni en paramètre inclus et le second exclu
SortedSet<E> tailSet(E fromElement)	Retourner un sous-ensemble des éléments qui sont plus grands ou égaux à celui fourni en paramètre
NavigableSet<E> tailSet(E fromElement, boolean inclusive)	Retourner un ensemble parcourable qui contient les éléments de la collection qui sont strictement plus grands (ou plus grands ou égaux si le paramètre inclusive vaut true) que celui fourni en paramètre

Il est recommandé aux implémentations de cette interface de ne pas permettre d'accepter la valeur null dans la collection pour éviter l'ambiguité lorsque certaines méthodes renvoient null.

L'API Collections propose deux implémentations de cette interface : TreeSet et ConcurrenSkipListSet.

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.Iterator;
import java.util.NavigableSet;
import java.util.Set;
import java.util.TreeSet;
```

```
public class TestNavigableSet {

  public static void afficherSet(final Set<String> set) {
    Iterator<String> iterator = set.iterator();
    while (iterator.hasNext()) {
      String element = iterator.next();
      System.out.print(element);
      if (iterator.hasNext()) {
        System.out.print(", ");
      } else {
        System.out.println("");
      }
    }
  }

  public static void main(final String[] args) {
    NavigableSet<String> set = new TreeSet<String>();
    for (int i = 1; i < 10; i++) {
      set.add("" + i);
    }

    System.out.println(set);
    System.out.println("ceiling(5)=" + set.ceiling("5"));
    System.out.println("floor(5)" + set.floor("5"));
    System.out.println("higher(5)=" + set.higher("5"));
    System.out.println("lower(5)=" + set.lower("5"));
    System.out.print("Ordre descendant=");
    afficherSet(set.descendingSet());

    System.out.print("headSet(5)=");
    afficherSet(set.headSet("5"));

    System.out.print("headSet(5,true)=");
    afficherSet(set.headSet("5", true));

    System.out.print("subSet(3,5)=");
    afficherSet(set.subSet("3", "5"));

    System.out.print("subSet(3,true,5,true)=");
    afficherSet(set.subSet("3", true, "5", true));

    System.out.print("tailSet(5)=");
    afficherSet(set.tailSet("5"));

    System.out.print("tailSet(5,true)=");
    afficherSet(set.tailSet("5", true));

    System.out.println("pollFirst()=" + set.pollFirst());
    System.out.println("pollLast()=" + set.pollLast());
    System.out.println(set);
  }
}
```

Résultat :

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
ceiling(5)=5
floor(5)5
higher(5)=6
lower(5)=4
Ordre descendant=9, 8, 7, 6, 5, 4, 3, 2, 1
headSet(5)=1, 2, 3, 4
headSet(5,true)=1, 2, 3, 4, 5
subSet(3,5)=3, 4
subSet(3,true,5,true)=3, 4, 5
tailSet(5)=5, 6, 7, 8, 9
tailSet(5,true)=5, 6, 7, 8, 9
pollFirst()=1
pollLast()=9
[2, 3, 4, 5, 6, 7, 8]
```

14.4.4. La classe HashSet

La classe HashSet, ajoutée à Java 1.2, est une implémentation simple de l'interface Set qui utilise une HashMap. La clé de la HashMap est la valeur de hachage de l'élément.

La classe HashSet présente plusieurs caractéristiques :

- elle ne propose aucune garantie sur l'ordre de parcours lors de l'itération sur les éléments qu'elle contient
- elle ne permet pas d'ajouter des doublons mais elle permet l'ajout d'un élément null

La classe HashSet utilise en interne une HashMap dont la clé est l'élément et dont la valeur est une instance d'Object identique pour tous les éléments.

La classe HashSet possède plusieurs constructeurs :

Constructeur	Rôle
HashSet()	Créer une nouvelle instance vide dont la HashMap interne utilisera une capacité initiale et un facteur de charge par défaut
HashSet(Collection<? extends E> c)	Créer une nouvelle instance contenant les éléments de la collection fournie en paramètre
HashSet(int initialCapacity)	Créer une nouvelle instance vide dont la HashMap interne utilisera la capacité initiale fournie en paramètre et un facteur de charge par défaut
HashSet(int initialCapacity, float loadFactor)	Créer une nouvelle instance vide dont la HashMap interne utilisera la capacité initiale et un facteur de charge par défaut

Il est possible de préciser, dans la surcharge de certains constructeurs, la capacité initiale de la collection et le facteur de charge (par défaut, la taille est 16 et le facteur de charge est 0,75). Le facteur de charge est une valeur qui précise le pourcentage de remplissage de la collection à atteindre avant d'augmenter sa taille.

Les classes des éléments qui sont insérés dans la collection doivent impérativement définir les méthodes equals() et hashCode() pour respecter la cohérence entre ces deux méthodes qui est imposée par contrat par Java.

La méthode add() permet d'ajouter un élément dans la collection : elle renvoie un booléen qui précise si l'opération a réussi ou non. Elle échoue par exemple si l'élément est déjà présent dans la collection.

Exemple (code Java 1.2) :

```
import java.util.*;
public class TestHashSet {
  public static void main(String args[]) {
    Set set = new HashSet();
    set.add("CCCCC");
    set.add("BBBBB");
    set.add("DDDDD");
    set.add("BBBBB");
    set.add("AAAAA");

    Iterator iterator = set.iterator();
    while (iterator.hasNext()) {
      System.out.println(iterator.next());
    }
  }
}
```

Résultat :

```
AAAAA
DDDDD
BBBBB
CCCCC
```

14.4.5. La classe TreeSet

La classe TreeSet, ajoutée à Java 1.2, stocke ses éléments de manière ordonnée en les comparant entre-eux. Cette classe permet d'insérer des éléments dans n'importe quel ordre et de restituer ces éléments dans un ordre précis lors de son parcours.

Une collection de type TreeSet ne peut pas contenir de doublons.

Elle implémente l'interface NavigableSet depuis Java 6

L'ordre des éléments de la collection peut être défini par deux moyens :

- l'ordre naturel des éléments s'ils implémentent l'interface Comparable
- l'ordre obtenu par l'utilisation d'une instance de type Comparator fournie en paramètre du constructeur de la collection

Le mécanisme utilisé pour la comparaison lors de la définition de l'ordre (Comparable ou Comparator) doit être cohérent avec l'implémentation de la méthode equals() : si element1.compareTo(element2) == 0 alors obligatoirement element1.equals(element2) == true.

Cela implique que l'algorithme de comparaison des éléments soit suffisamment discriminant pour éviter les égalités qui seraient alors interprétés comme des doublons qui n'en sont pas en réalité.

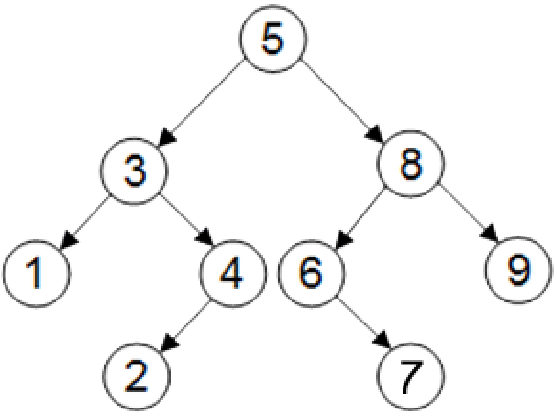
Par exemple : pour des personnes, il n'est pas possible de comparer uniquement le nom et le prénom car dans ce cas, il ne pourrait pas y avoir d'homonymes dans la collection. Il faut en plus comparer un élément discriminant ou faire la comparaison sur une valeur unique comme un numéro de sécurité sociale ou un identifiant.

En interne, la classe TreeSet utilise un arbre binaire pour stocker ses éléments. Chaque élément est encapsulé dans un noeud (node). Chaque noeud peut faire référence à aucun, un ou deux autres noeuds.

Si un noeud fait référence à un ou deux autres noeuds alors il est le noeud parent de ses noeuds fils : ceci permet de construire l'arborescence des éléments de l'arbre. Si un noeud n'a pas de fils alors c'est une feuille de l'arbre.

L'ajout d'un noeud fils suit toujours les mêmes règles :

- un des deux noeuds contient toujours un élément dont la valeur est plus petite
- l'autre noeud contient toujours un élément dont la valeur est supérieure
- c'est toujours le même noeud qui doit contenir la valeur la plus petite et l'autre la valeur la plus grande



Un seul noeud dans l'arbre ne possède pas de parent : le noeud racine.

La recherche d'un élément dans un arbre binaire est rapide : elle nécessite généralement un temps proportionnel à log(n) où n est le nombre d'éléments dans la collection.

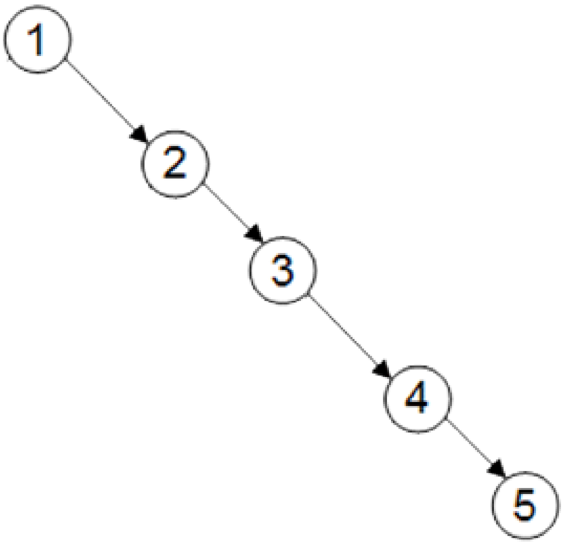
Le parcours commence par le noeud racine qui est comparé à l'élément recherché.

S'il est égal, l'élément est trouvé sinon la branche de l'arbre selon que la valeur est plus petite ou plus grande est parcourue.

A chaque noeud la valeur est testée par rapport à l'élément recherché

Si le dernier parcouru ne correspond pas à la valeur recherchée, alors l'élément n'est pas dans la collection.

Ce mode de fonctionnement est efficace si l'ordre d'insertion des éléments est aléatoire : si tous les éléments sont ajoutés dans leur ordre alors le parcours reviendra à parcourir tous les éléments un par un jusqu'à trouver la bonne valeur ou une valeur supérieure auquel cas la valeur n'est pas trouvée.



Pour pallier à cette problématique, la classe TreeSet met en oeuvre un algorithme complexe qui va permettre d'équilibrer l'arbre. Un arbre est équilibré lorsque les feuilles de l'arbre sont à peu près à la même distance de la racine de l'arbre. La distance est le nombre de noeuds parent entre la feuille et le noeud racine.

La mise en oeuvre de ce type d'algorithme peut imposer de réorganiser la structure de l'arbre à chaque ajout ou suppression d'un élément. Le maintient d'un arbre parfaitement équilibré peut être très coûteux. Certains algorithmes maintiennent un arbre partiellement équilibré jusqu'à une certaine limite clairement définie par l'algorithme.

Il est intéressant que l'arbre soit parfaitement équilibré si les recherches sont beaucoup plus nombreuses que les opérations d'ajouts ou de suppressions d'éléments. La classe TreeSet ayant pour vocation un usage généraliste, elle met en oeuvre un algorithme nommé Red-Black Tree qui équilibre partiellement l'arbre en garantissant que la distance entre la racine et la feuille la plus éloignée n'est pas plus importante que deux fois la distance entre la racine et la feuille la plus proche. Cet algorithme met en oeuvre plusieurs règles pour réorganiser les éléments de l'arbre pour le maintenir équilibré :

- chaque noeud est noir ou rouge
- le noeud racine est généralement noir
- si un noeud est rouge alors ses noeuds enfants sont noirs
- pour chaque noeud de l'arbre, les chemins vers les noeuds de type feuille doivent contenir le même nombre de noeuds noirs

La classe TreeSet stocke en interne ses éléments dans une collection de type TreeMap.

La classe TreeSet possède plusieurs constructeurs :

Constructeur	Rôle
TreeSet()	Créer une instance vide dont l'ordre naturel de tri de ses éléments est utilisé
TreeSet(Collection<? extends E> c)	Créer une instance contenant les éléments de la collection fournie en paramètre dont l'ordre naturel de tri de ses éléments est utilisé
TreeSet(Comparator<? super E> comparator)	Créer une instance vide dont l'ordre utilisé est celui définit par l'instance de type Comparator fournie en paramètre
TreeSet(SortedSet<E> s)	Créer une instance contenant les éléments de la collection fournie en paramètre dont l'ordre est celui utilisé par la collection

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.Iterator;
import java.util.TreeSet;

public class TestTreeSet {
  public static void main(final String[] args) {
    TreeSet<String> set = new TreeSet<String>();
    set.add("CCCCC");
    set.add("BBBBB");
    set.add("DDDDD");
    set.add("BBBBB");
    set.add("AAAAA");

    Iterator<String> iterator = set.iterator();
    while (iterator.hasNext()) {
      System.out.print(iterator.next() + " ");
    }
  }
}
```

Résultat :

```
AAAAA BBBBB CCCCC DDDDD
```

La classe TreeSet n'est pas thread-safe : comme aucune de ses méthodes n'est synchronized, un seul thread doit pouvoir modifier le contenu de la collection. Si plusieurs threads doivent pouvoir modifier la collection, il faut invoquer la méthode synchronizedSortedSet() de la classe Collections qui va créer un wrapper dont les méthodes sont synchronized.

SortedSet set = Collections.synchronizedSortedSet(new TreeSet());

Avec cette solution, plusieurs threads peuvent modifier la collection mais un seul à la fois, ce qui peut engendrer des dégradations de performances en cas de forte concurrence d'accès. Dans ce cas, il est préférable d'utiliser une autre implémentation de type Set comme la classe ConcurrentSkipListSet.

14.4.6. La classe ConcurrentSkipListSet

La classe ConcurrentSkipListSet, ajoutée à Java 1.6, permet de mettre en oeuvre un ensemble ordonné d'éléments capable de gérer une forte concurrence d'accès.

Elle implémente l'interface NavigableSet et utilise en interne une instance de type ConcurrentSkipListMap.

Elle présente plusieurs caractéristiques :

- les éléments sont ordonnés dans leur ordre naturel s'ils implémentent l'interface Comparable ou selon un ordre défini par l'instance de type Comparator fournie au constructeur de l'instance de la collection.
- le parcours ascendant des éléments est plus rapide que le parcours descendant.
- elle ne permet pas de contenir un objet null.
- les opérations de la classe ConcurrentSkipListSet sont de types CAS (Compare And Swap) : elles ne posent aucun verrou qui pourrait introduire de la contention.
- elle implémente toutes les méthodes optionnelles de l'interface Set

La classe ConcurrentSkipListSet utilise une structure de données de type skip list. Contrairement à un arbre binaire, dans une structure de type skip list, l'organisation n'a pas besoin d'être réajustée lors de l'ajout ou la suppression d'un élément.

La classe ConcurrentSkipListSet possède plusieurs constructeurs :

Constructeur	Rôle
ConcurrentSkipListSet()	

	Créer une nouvelle instance vide dont les éléments sont triés avec leur ordre naturel
ConcurrentSkipListSet(Collection<? extends E> c)	Créer une nouvelle instance contenant les éléments de la collection fournie en paramètre triés avec leur ordre naturel
ConcurrentSkipListSet(Comparator<? super E> comparator)	Créer une nouvelle instance vide dont les éléments sont triés en utilisant l'instance de type Comparator fournie en paramètre
ConcurrentSkipListSet(SortedSet<E> s)	Créer une nouvelle instance contenant les éléments de la collection fournie en paramètre triés selon l'ordre de cette collection

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.Iterator;
import java.util.concurrent.ConcurrentSkipListSet;

public class TestConcurrentSkipListSet {
  public static void main(final String[] args) {
    final ConcurrentSkipListSet<MaTache> set = new ConcurrentSkipListSet<MaTache>();
    System.out.println("debut");
    final Thread modificateur = new Thread(new Runnable() {
      @Override
      public void run() {
        MaTache[] MesTaches = new MaTache[5];
        for (int i = 1; i <= 5; i++) {
          MesTaches[i - 1] = new MaTache(6 - i, "Tache " + i);
        }
        for (int j = 1; j <= 100; j++) {
          if (j % 2 == 0) {
            for (int i = 1; i <= 5; i++) {
              MaTache element = MesTaches[i - 1];
              System.out.println("insertion element " + element);
              set.add(element);
            }
            System.out.println("taille de la queue=" + set.size());
          } else {
            for (int i = 1; i <= 5; i++) {
              MaTache element = MesTaches[i - 1];
              System.out.println("retirer element " + element);
              set.remove(element);
            }
          }
        }
      }
    }, "Modificateur");

    modificateur.start();

    Thread iterateur = new Thread(new Runnable() {
      @Override
      public void run() {
        int i = 0;
        while (modificateur.isAlive()) {
          Iterator<MaTache> iterator = set.iterator();
          StringBuilder contenu = new StringBuilder("[");
          while (iterator.hasNext()) {
            contenu.append(iterator.next().getDescription());
            if (iterator.hasNext()) {
              contenu.append(", ");
            }
          }

          contenu.append("]");
          System.out.println("Contenu=" + contenu);
          i++;
        }
      }
    }, "iterateur");

    iterateur.start();
    try {
      modificateur.join();
      iterateur.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    System.out.println("fin");
  }
}
```

La classe ConcurrentSkipListSet est particulièrement utile pour gérer un ensemble ordonné d'éléments qui peut être accédé et modifié par plusieurs threads. L'ajout, la suppression et l'obtention d'un élément de la collection se font de manière concurrente par plusieurs threads.

Les méthodes qui effectuent des opérations sur plusieurs éléments comme addAll(), removeAll() et containsAll() n'offrent aucune garantie de s'exécuter de manière atomique.

Les performances des opérations de base de la classe ConcurrentSkipListSet sont moins bonnes que celles de la classe TreeSet pour une utilisation mono-thread : dans ce cas, il est préférable d'utiliser une instance de type TreeSet.

Le temps d'exécution de la méthode size() n'est pas constant : il est proportionnel au nombre d'éléments de la collection car celle-ci doit reparcourir tous les éléments pour calculer le nombre d'éléments qu'elle contient. Elle ne pose aucun verrou pour maintenir fraîche la valeur du nombre d'éléments dans la collection.

Les Iterator ne lèvent jamais d'exception de type ConcurrentModificationException : il est possible de les utiliser alors que d'autres threads modifient la collection car l'itération se fait sur l'état de la collection au moment de la création de l'instance de l'Iterator.

Les itérations sur les éléments dans l'ordre ascendant sont plus rapides que les itérations dans l'ordre descendant.

14.4.7. La classe CopyOnWriteArraySet

La classe CopyOnWriteArraySet, ajoutée à Java 1.5, est une implémentation de type Set qui est thread safe et offre de bonnes performances en lecture.

Elle implémente les interfaces Collection, Set et Iterable.

Elle utilise en interne une instance de type CopyOnWriteArrayList pour stocker les éléments de la collection. Elle présente plusieurs caractéristiques :

- elle est thread safe
- les opérations de mises à jour de la collection sont coûteuses en ressources car elles impliquent une copie intégrale des éléments du tableau : la collection doit donc de préférence être de petite taille et ne pas être trop fréquemment modifiée
- le parcours grâce à un Iterator ne peut être influencé par une opération d'un autre thread : l'itération se fait sur une copie dédiée des éléments du tableau
- les Iterator ne proposent pas de support pour la méthode remove()

La classe CopyOnWriteArraySet possède plusieurs constructeurs :

Constructeur	Rôle
CopyOnWriteArraySet ()	Créer une nouvelle instance vide
CopyOnWriteArraySet (Collection<? extends E> c)	Créer une nouvelle instance contenant les éléments de la collection fournie en paramètre

14.4.8. Le choix d'une implémentation de type Set

Certaines implémentations sont spécialisées pour être utilisées dans des situations particulières.

C'est notamment le cas de la classe EnumSet qui ne doit être utilisée que pour gérer un ensemble d'énumérations.

La classe CopyOnWriteArraySet ne doit être utilisée que pour des collections thread-safe de petites tailles, où les opérations réalisées sont essentiellement des lectures et où les Iterator ne peuvent pas modifier le contenu de la collection.

Le JDK contient plusieurs implémentations généralistes de l'interface Set qui peuvent selon les besoins :

- maintenir un ordre des clés
- gérer des accès concurrents

Ordre des clés	Pas d'accès concurrent	Gestion des accès concurrents
Aucun	HashSet	
Trié	TreeSet	ConcurrentSkipListMap
Fixe	LinkedHashSet	CopyOnWriteArraySet

Si la collection n'est pas utilisée par plusieurs threads, il est possible d'utiliser les classes HashSet, LinkedHashSet et TreeSet. Si les données doivent être triées, il faut utiliser la classe TreeSet. Si les données de la collection doivent être fréquemment parcourues, il est préférable d'utiliser la classe LinkedHashSet.

Si la collection doit être utilisée par plusieurs threads, il faut utiliser la classe ConcurrentSkipListSet ou CopyOnWriteArraySet uniquement si les accès sont essentiellement des lectures. Il est aussi possible d'utiliser une version synchronized d'une implémentation de type Set en utilisant la méthode synchronizedSet() de la classe Collections.

Si les éléments de la collection doivent être triés, il faut utiliser les classes TreeSet ou ConcurrentSkipListSet. Si en plus les accès concurrents doivent être gérés, seule la classe ConcurrentSkipListSet doit être utilisée. Sinon il est préférable d'utiliser la classe TreeSet pour des collections de grandes tailles ou si de nombreuses opérations de suppressions d'éléments doivent être réalisées.

Indépendamment des fonctionnalités, les performances peuvent être un critère important dans le choix d'une implémentation de type Set.

Classe	add()	contains()	next()	thread-safe
HashSet	O(1)	O(1)	O(h/n)	Non
LinkedHashSet	O(1)	O(1)	O(1)	Non
CopyOnWriteArraySet	O(n)	O(n)	O(1)	Non
EnumSet	O(1)	O(1)	O(1)	Oui
TreeSet	O(log n)	O(log n)	O(log n)	Oui
ConcurrentSkipListSet	O(log n)	O(log n)	O(1)	Oui

Remarque : dans le tableau ci-dessus, h est la capacité de la collection

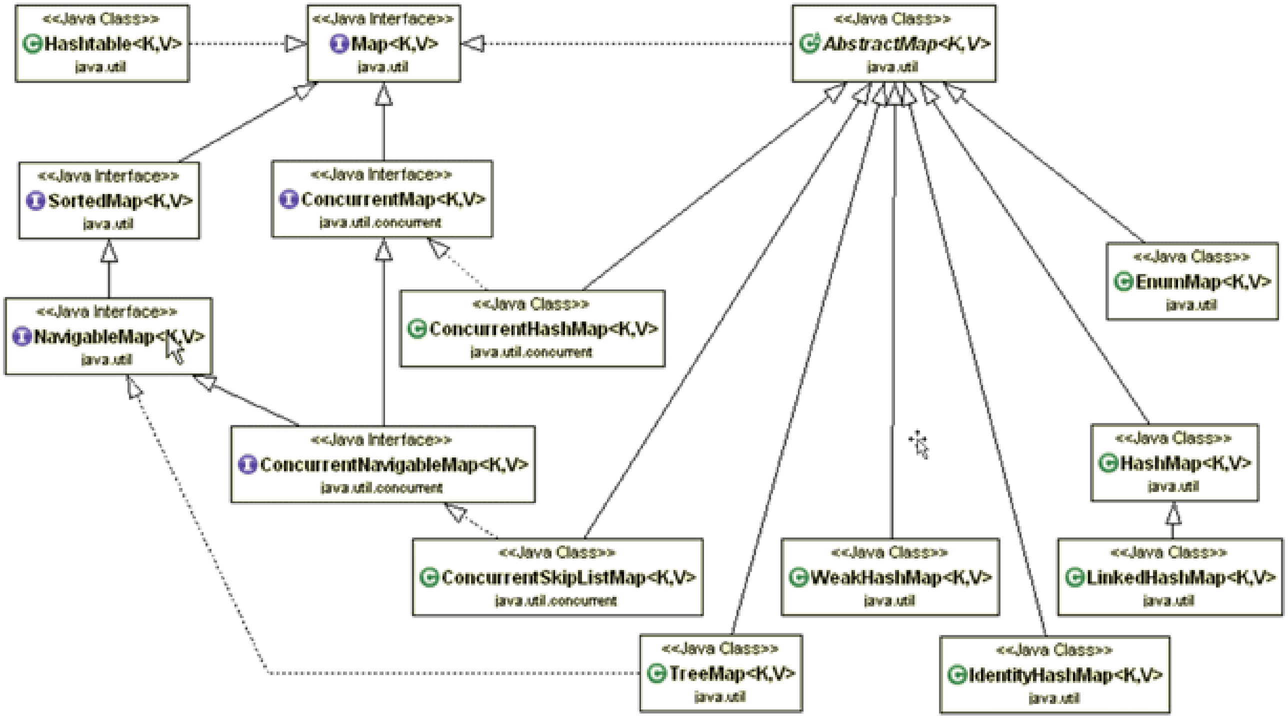
14.5. Les collections de type Map : les associations de type clé/valeur

Les collections de type Map sont définies et implémentées comme des dictionnaires sous la forme d'associations de paires de type clés/valeurs. La clé doit être unique. En revanche, la même valeur peut être associée à plusieurs clés différentes.

Avant l'apparition du framework Collections, la classe dédiée à cette gestion était la classe Hashtable.

Un objet de type Map permet de lier un objet avec une clé qui peut être un type primitif ou un autre objet. Il est ainsi possible d'obtenir un objet à partir de sa clé.

Au fur et à mesure des versions de Java, des classes et interfaces de type Map ont été ajoutées :



L'interface Map possède plusieurs interfaces filles : SortedMap, NavigableMap, ConcurrentMap et ConcurrentNavigableMap.

L'API Collections propose plusieurs implémentations de l'interface Map :

Classe	Rôle
java.util.TreeMap<K,V>	Map non thread safe dont l'ordre de parcours des clés est garanti
java.util.Hashtable<K,V>	Map thread-safe, null ne peut pas être utilisé comme clé
java.util.HashMap<K,V>	Similaire à Hashtable mais elle n'est pas thread-safe et null peut être utilisé comme clé
java.util.concurrent.ConcurrentHashMap<K,V>	Similaire à Hashtable, avec une gestion des accès concurrents et de meilleures performances
java.util.WeakHashMap<K,V>	Map qui va retirer automatiquement les éléments dont les clés ne peuvent plus être utilisées. S'il n'existe plus aucune référence forte dans le tas de la JVM sur un objet utilisé comme clé, alors l'élément correspondant dans la collection sera retiré
java.util.LinkedHashMap<E>	Map non thread safe qui conserve les clés dans leur ordre d'insertion
java.util.IdentityHashMap<K,V>	Map non thread-safe qui utilise un test d'égalité sur les références (habituellement les implémentations de l'interface Map utilisent l'égalité des objets). Deux clés cle1 et cle2 sont donc égales si cle1==cle2.
java.util.EnumMap<K,V>	Map non thread-safe dont les valeurs doivent appartenir à la même énumération
java.util.IdentityHashMap<K,V>	

	Map appropriée lorsque la comparaison des éléments doit se faire sur l'identité des objets et non sur leur égalité (elle n'utilise pas les méthodes equals() et hashCode() pour comparer les clés)

14.5.1. L'interface Map

L'interface java.util.Map<K,V> , ajoutée à Java 1.2, définit les fonctionnalités pour une collection qui associe des clés à des valeurs. Chaque clé ne peut être associée qu'à une seule valeur. Chaque clé d'une Map doit être unique.

L'interface Map de l'API Collections remplace la classe abstraite Dictionary de Java 1.0.

Elle définit plusieurs méthodes pour agir sur la collection :

Méthode	Rôle
void clear()	Supprimer tous les éléments de la collection
boolean containsKey(Object)	Indiquer si la clé est contenue dans la collection
boolean containsValue(Object)	Indiquer si la valeur est contenue dans la collection
Set entrySet()	Renvoyer un ensemble contenant les paires clé/valeur de la collection
Object get(Object)	Renvoyer la valeur associée à la clé fournie en paramètre
boolean isEmpty()	Indiquer si la collection est vide
Set keySet()	Renvoyer un ensemble contenant les clés de la collection
Object put(Object, Object)	Insèrer la clé et sa valeur associée fournies en paramètres
void putAll(Map)	Insèrer toutes les clés/valeurs de l'objet fourni en paramètre
Collection values()	Renvoyer une collection qui contient toutes les valeurs des éléments
Object remove(Object)	Supprimer l'élément dont la clé est fournie en paramètre
int size()	Renvoyer le nombre d'éléments de la collection

La méthode keySet() permet d'obtenir un ensemble contenant toutes les clés.

La méthode values() permet d'obtenir une collection contenant toutes les valeurs. La valeur de retour est une Collection et non un ensemble car il peut y avoir des doublons (plusieurs clés peuvent être associées à la même valeur).

Elle définit une interface interne Map.Entry<K,V> qui définit les fonctionnalités pour un objet qui encapsule une paire clé/valeur.

Il est recommandé d'utiliser des objets immuables comme clés.

Une collection de type Map ne propose pas directement d'Iterator sur ses éléments : la collection peut être parcourue de trois manières :

- parcours de l'ensemble des clés
- parcours des valeurs
- parcours d'un ensemble de paires clé/valeur

L'API Collections propose plusieurs implémentations de l'interface Map notamment HashMap, Hashtable, TreeMap, LinkedHashMap, ConcurrentHashMap, ConcurrentSkipListMap, EnumMap et WeakHashMap.

14.5.2. L'interface SortedMap

L'interface java.util.SortedMap<K,V>, ajoutée à Java 1.2, définit les fonctionnalités d'une Map dont les clés sont triées. Elle hérite de l'interface Map.

L'ordre dans les clés est assuré en utilisant l'ordre naturel des éléments (en implémentant l'interface Comparable) ou en fournissant un Comparator à la création de l'instance de la collection. Tous les éléments insérés dans la collection doivent donc implémenter l'interface Comparable ou pouvoir être utilisés par le Comparator associé à la Map selon la solution utilisée. Ils doivent aussi avoir une implémentation de la méthode equals() qui soit en accord avec cette solution car elle est invoquée pour déterminer si la clé est déjà dans la collection.

L'ordre des éléments est respecté lors de l'invocation des méthodes entrySet(), keySet() et values().

Les implémentations de l'interface SortedMap doivent garantir que les Iterator parcourent la collection dans l'ordre des clés.

L'interface SortedMap définit plusieurs méthodes :

Méthode	Rôle
Comparator< ? super K) comparator()	Retourner l'instance de type Comparator associée à la collection ou null si c'est l'ordre naturel qui doit être utilisé
Set<Map.Entry<K,V>> entrySet()	Retourner un ensemble des paires clé/valeur de la collection
K firstKey()	Retourner la première clé de la collection. Lève une exception de type NoSuchElementException si la collection est vide
SortedMap<K,V> headMap(K toKey)	Retourner un sous-ensemble de la collection contenant les éléments dont les clés sont strictement inférieures à celle fournie en paramètre
Set<K> keySet()	Retourner un ensemble des clés de la collection
K lastKey()	Retourner la dernière clé de la collection. Lève une exception de type NoSuchElementException si la collection est vide
sortedMap<K, V> subMap(K fromKey, K toKey)	Retourner un sous-ensemble de la collection contenant les éléments dont les clés sont strictement inférieures à celle fournie en premier paramètre et supérieures ou égales à celle fournie en second paramètre
SortedMap<K,V> tailMap(K fromKey)	Retourner un sous-ensemble de la collection contenant les éléments dont les clés sont supérieures ou égales à celle fournie en paramètre
Collection(V) values()	Retourner une collection de toutes les valeurs de la Map

Chaque implémentation de l'interface SortedMap devrait fournir au moins quatre constructeurs (ceci ne peut être qu'une recommandation puisque les constructeurs ne peuvent pas être définis dans une interface) :

- un constructeur par défaut (sans argument) qui créé une Map vide qui utilisera l'ordre naturel des objets
- un constructeur qui attend un objet de type Comparator qui créé une Map vide qui utilisera l'ordre obtenu grâce au paramètre
- un constructeur qui attend en paramètre un objet de type Map qui créé une Map contenant les éléments fournis en paramètres, triés selon leur ordre naturel
- un constructeur qui attend en paramètre un objet de type SortedMap qui créé une Map contenant les éléments fournis en paramètres, triés dans le même ordre

Elle possède deux interfaces filles depuis Java 6 : NavigableMap et ConcurrentNavigableMap.

L'API Collections propose deux implémentations de l'interface SortedMap : TreeMap et ConcurrentSkipListMap.

14.5.3. La classe Hashtable

La classe Hashtable, présente depuis Java 1.0, permet d'associer dans une collection des éléments sous la forme de paires clé/valeur.

La classe Hashtable hérite de la classe Dictionary qui n'appartient pas à l'API Collections et a été modifiée, à partir de Java 1.2, pour implémenter l'interface Map et ainsi devenir une classe de l'API Collections.

La classe Hashtable présente plusieurs caractéristiques :

- contrairement aux autres classes de l'API Collections, elle est thread-safe car toutes les méthodes sont synchronized.
- il n'est pas possible d'utiliser la valeur null comme clé ou valeur

Tous les objets qui sont utilisés comme clés doivent obligatoirement redéfinir les méthodes equals() et hashCode() en respectant le contrat portant sur l'implémentation de ces deux méthodes.

La classe Hashtable est composée de buckets : en fonction de la valeur de hachage de la clé, l'élément est inséré dans un bucket particulier. Plusieurs objets ayant la même valeur de hachage seront dans le même bucket.

La classe Hashtable possède deux propriétés qui affectent ses performances :

- la capacité initiale (initial capacity) : le nombre d'éléments que peut contenir la collection à sa création
- le facteur de charge (load factor) : ce facteur précise le pourcentage de remplissage de la collection avant son agrandissement pour être capable de contenir plus d'éléments. La valeur par défaut est 0.75.

Il est important de ne pas utiliser une capacité initiale trop importante ou un facteur de charge trop petit pour ne pas dégrader les performances lors du parcours de la collection.

Lorsque le nombre d'éléments de la collection est supérieur à la taille de la collection multiplié par le facteur de charge alors la collection est agrandie. Cette opération est coûteuse car elle impose un rehash de la collection (reconstruction de sa structure de données liée à un accroissement du nombre de buckets). L'invocation de la méthode rehash() est spécifique à l'implémentation.

La capacité estimée et le facteur de charge de la collection doivent être pris en compte pour définir la capacité initiale afin d'éviter au maximum le nombre d'opérations de type rehash effectué lors de l'agrandissement de la taille de la collection.

Si la collection doit contenir de nombreux éléments, il est donc intéressant de créer son instance avec une capacité initiale suffisament élevée pour contenir les éléments plus une marge correspondant à la capacité multipliée par le facteur de charge. Les performances seront améliorées car cela évitera une ou plusieurs opérations d'agrandissement de la collection.

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.Hashtable;

public class TestHashtable {

  public static void main(final String[] args) {
    Hashtable<Integer, String> numbers = new Hashtable<Integer, String>();
    numbers.put(1, "Element1");
    numbers.put(2, "Element2");
    numbers.put(3, "Element3");
    String n = numbers.get(2);
    if (n != null) {
      System.out.println("2 = " + n);
    }
  }
}
```

Résultat :

```
2 = Element2
```

Les Iterator de cette classe sont de type fail-fast : les instances de type Iterator retournées par la méthode iterator() lèvent généralement une exception de type ConcurrentModificationException si une modification est effectuée dans la collection durant le parcours (exception faite des modifications réalisées avec la méthode remove() de l'Iterator).

14.5.4. La classe HashMap

La classe HashMap, ajoutée à Java 1.2, est une implémentation de l'interface Map qui utilise une Hashtable. La classe HashMap est similaire à la classe Hashtable sauf qu'elle n'est pas synchronized et qu'elle autorise l'utilisation de la valeur null.

La classe HashMap utilise un tableau de listes chaînées pour le stockage de ses éléments. L'index d'une clé dans le tableau est déterminé grâce à un algorithme utilisant la valeur de hachage de l'objet.

La valeur de hachage n'est pas utilisée directement : l'algorithme fait appel à la méthode hash() de la classe HashMap qui utilise la valeur de hachage pour en déterminer une autre, ceci afin de réduire les risques de collision.

Si deux objets possèdent la même valeur de hachage, il y a une collision car les deux objets doivent être insérés dans le même bucket. Pour gérer les problèmes, le bucket contient une liste chaînée : chaque élément (sa clé et sa valeur) est encapsulé dans une instance de type Entry.

Toutes les méthodes optionnelles de l'interface Map sont implémentées.

La classe HashMap présente plusieurs caractéristiques :

- Elle permet l'utilisation de la valeur null comme clé et comme valeur.
- Elle n'est pas thread-safe.
- Elle ne garantit aucun ordre lors du parcours des éléments de la collection.

La classe HashMap possède deux propriétés :

- la capacité initiale
- le facteur de charge (load factor)

Il est important que la valeur de la capacité initiale soit une puissance de 2 : si ce n'est pas le cas la capacité initiale sera la valeur 2^n supérieure. Si la capacité c fournie est $2^{n-1} > c < 2^n$ alors la capacité sera 2^n. La valeur de la capacité doit être une puissance de 2 pour permettre à l'algorithme qui détermine l'index dans le tableau à partir de la valeur de hachage de fonctionner.

Le facteur de charge définit le pourcentage de remplissage maximum de la collection avant que celle-ci ne soit agrandie. Si le pourcentage du nombre d'éléments contenus dans la collection par rapport à la capacité de la collection est supérieur au facteur de charge, alors la capacité maximale de la collection est doublée.

La classe HashMap propose plusieurs constructeurs dont certains peuvent préciser la capacité initiale et le facteur de charge.

Pour pouvoir correctement mettre en oeuvre la classe HashMap, il est nécessaire de connaître son mode de fonctionnement.

En interne, une HashMap stocke les paires clé/valeur dans des buckets qui peuvent être vus comme un index de premier niveau. Lors de l'ajout d'un élément ou la recherche d'un élément dans la collection, la valeur de hachage de la clé est utilisée pour déterminer un index dans le tableau des buckets.

Chaque bucket possède une liste chaînée qui stocke les éléments dont la clé possède la même valeur de hachage. Comme deux objets égaux doivent avoir la même valeur de hachage mais que deux objets ayant la même valeur de hachage ne sont pas forcément égaux, une liste chaînée est utilisée pour stocker les éléments ayant la même valeur de hachage.

La classe HashMap possède la classe interne Entry qui implémente l'interface Map.Entry et encapsule une paire clé/valeur. La liste chaînée du bucket contient donc des objets de type Entry qui encapsulent la clé et la valeur. La méthode next() permet d'obtenir l'élément suivant dans la liste.

Lors de l'invocation de la méthode put(), plusieurs opérations sont réalisées :

- invocation de la méthode hashCode() sur l'objet qui encapsule la clé
- la méthode hash() de la classe HashMap utilise son propre algorithme pour déterminer un index à partir de la valeur de hachage de la clé
- un objet de type Entry qui encapsule la clé et la valeur est instancié
- si l'index du bucket est vide alors l'instance de type Entry est ajoutée
- si l'index n'est pas vide, la liste chainée est parcourue en commençant par l'élément stocké à l'index puis en invoquant successivement sa méthode next(). La clé de l'élément est comparée à la clé de l'élément Entry courant en utilisation la méthode equals()
- si la même clé est trouvée alors la valeur est remplacée par celle de l'élément à ajouter sinon le nouvel élément de type Entry est ajouté à la fin de la liste chaînée

La classe HashMap accepte la valeur null comme clé : dans ce cas, la clé est placée à l'index 0 dans le tableau.

Plusieurs éléments pouvant être stockés à un même index, il est nécessaire de parcourir ses éléments qui sont dans une liste chaînée, encapsulés dans des objets de type Entry. Le parcours se fait en invoquant la méthode next(). La clé de l'élément courant est comparée à la clé recherchée en utilisant la méthode equals() si les instances sont différentes.

Pour obtenir un élément de la collection grâce à la méthode get(), le mode de fonctionnement est similaire à l'ajout. Lors de l'invocation de la méthode get() :

- l'index du bucket dans le tableau est déterminé en utilisant la valeur de hachage de la clé : elle est utilisée par la méthode hash() pour calculer une valeur de hachage, elle-même utilisée pour déterminer l'index dans le tableau
- si l'index du bucket est vide alors l'élément n'est pas trouvé
- si le bucket ne contient qu'un seul élément, c'est la valeur de cet élément qui est retournée
- si la liste chaînée du bucket contient plusieurs éléments alors chaque élément est parcouru en commençant par l'élément stocké à l'index puis invoquant successivement sa méthode next() pour en trouver un dont la clé soit égale à la clé recherchée en invoquant la méthode equals()
- si un élément est trouvé alors c'est la valeur de cet élément qui est retournée
- si aucun élément n'est trouvé alors la valeur retournée est null

L'implémentation des méthodes equals() et hashCode() d'objets utilisés comme clés doit respecter les spécifications qui sont imposées par Java. La valeur de hachage de la clé obtenue en invoquant sa méthode hashCode() est utilisée pour calculer l'index du bucket. La méthode equals() des clés est utilisée pour assurer l'unicité des clés dans la collection et pour retrouver le bon élément.

Comme une HashMap repose sur l'utilisation de la valeur de hachage des clés, l'idéal est d'utiliser des instances immuables d'objets dont les méthodes equals() et hashCode() sont correctement implémentées. L'utilisation d'objets immuables comme clés assure, si l'implémentation de la méthode hashCode() est bien faite, que la valeur de hachage de l'objet ne change pas. Ainsi des objets de type String ou des classes de type wrapper de primitives sont de bons candidats pour les clés de la collection.

Il est très important que la valeur de hachage d'un objet utilisé comme clé ne change pas car cette valeur est utilisée pour déterminer le bucket de la collection qui contient l'élément. Si un élément est ajouté dans la collection avec une clé ayant une certaine valeur de hachage et que cette valeur de hachage est différente lors de la recherche de clé dans la collection, l'élément ne sera probablement pas retrouvé alors qu'il est bel et bien dans la collection. Cette recherche échoue car la valeur de hachage à l'ajout place l'élément dans un bucket et la recherche avec une autre valeur de hachage détermine un autre bucket qui ne contient pas l'élément.

Si le facteur de charge est atteint, alors la taille de la collection est agrandie : la taille du tableau contenant les buckets est doublée en recréant une nouvelle instance du tableau. Cette opération implique un recalcul de tous les buckets : cette fonctionnalité est appelée rehash car elle redéfinit toutes les valeurs des index.

Lorsque la collection est redimensionnée, chaque élément est déplacé du tableau initial dans le nouveau tableau dont la taille est doublée. Lors de ce transfert, l'index de l'élément dans le bucket est recalculé, ce qui fait qu'il peut rester le même ou être modifié.

Le temps d'exécution de cette opération est proportionnel à la taille de la collection : pour limiter ses occurrences, il est possible de préciser la capacité initiale de la collection si l'on possède une idée plus ou moins précise du nombre d'éléments que contiendra la collection.

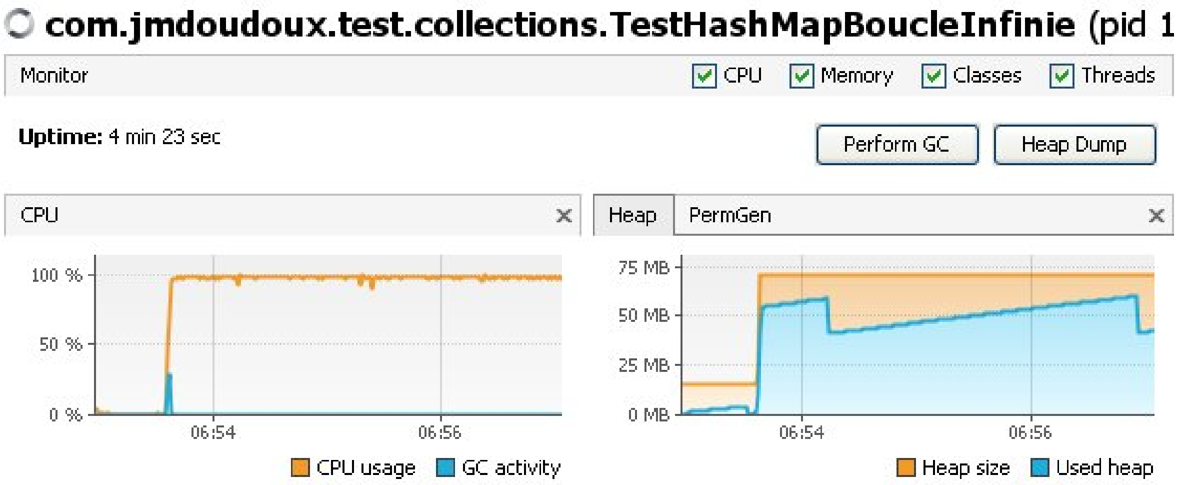
La classe HashMap n'est pas synchronized, elle n'est donc pas thread-safe. Si plusieurs threads doivent ajouter ou supprimer des éléments dans la collection, il faut gérer manuellement la concurrence d'accès. Il est par exemple possible d'utiliser une instance de la collection retournée par la méthode synchronizedMap() de la classe Collections.

```
Map m = Collections.synchronizedMap(new HashMap());
```

Une collection de type HashMap n'est pas prévue pour être utilisée par plusieurs threads. Par exemple, une invocation de la méthode get() peut déclencher une boucle infinie si une opération de rehash() est réalisée en même temps que l'insertion d'un élément. Durant l'agrandissement de la collection, les éléments de la liste chaînée d'un bucket sont déplacés dans un nouveau bucket dans leur sens inverse de la liste courante. Si un accès est fait à ce moment là, une boucle infinie peut être engendrée. Ceci peut aussi arriver si deux threads effectuent en même temps une opération de type rehash().

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.HashMap;
import java.util.Map;

class MonElement {
  private final int valeur;
  public MonElement(final int valeur) {
    this.valeur = valeur;
  }

  @Override
  public boolean equals(final Object obj) {
    if (this == obj) {
      return true;
    }
    if (obj == null) {
      return false;
    }
    if (getClass() != obj.getClass()) {
      return false;
    }
    MonElement other = (MonElement) obj;
    if (valeur != other.valeur) {
      return false;
    }
    return true;
  }

  @Override
  public int hashCode() {
    return valeur / 100;
  }
}

public class TestHashMapBoucleInfinie {
  /**
   * Attention : cette application engendre généralement une boucle infinie
   * qui consomme toute la CPU de la machine
   */
  public static void main(final String[] args) {
    final Map<MonElement, String> map = new HashMap<MonElement, String>(2, 0.2f);

    System.out.println("debut");
    for (int j = 0; j < 10; j++) {
      Thread t1 = new Thread(new Runnable() {
        @Override
        public void run() {
          for (int i = 1; i <= 100000; i++) {
            map.put(new MonElement(i), "element " + i);
          }
        }
      }, "Thread 0" + j);
      t1.start();
    }

    for (int j = 0; j < 10; j++) {
      Thread t2 = new Thread(new Runnable() {
        @Override
        public void run() {
          for (int i = 1; i <= 10000; i++) {
            map.get(i);
          }
        }
      }, "Thread 1" + j);
      t2.start();
    }
  }
}
```



Uptime: 4 min 23 sec

Les Iterator de cette classe sont de type fail-fast : ils lèvent généralement une exception de type ConcurrentModificationException si une modification de la structure de la collection est réalisée (ajout ou suppression d'un élément) durant le parcours sauf si cette modification est faite grâce à l'Iterator.

Comme l'index est calculé à partir de la valeur de hachage de l'objet, l'ordre de parcours des clés n'est pas garanti. Il peut même changer au cours du temps si de nouveaux éléments sont ajoutés dans la collection.

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.HashMap;

public class TestHashMapIterator {
  public static void main(final String[] args) {
    HashMap<String, String> map = new HashMap<String, String>(34);
    map.put("1", "un");
    map.put("2", "deux");
    map.put("3", "trois");
    map.put("4", "quatre");
    map.put("5", "cinq");
    for (String s : map.keySet()) {
      System.out.print(s + " ");
    }
  }
}
```

Résultat :

```
3 2 1 5 4
```

Les performances des opérations get() et put() sont constantes sous réserve que la répartition des éléments dans les différents buckets grâce à leur valeur de hachage soit équitable.

La valeur de hachage doit être la plus discriminante possible pour permettre notamment de réduire les collisions (objets différents ayant la même valeur de hachage). Si tous les éléments de la collection possèdent la même valeur de hachage, alors ils sont tous dans le même bucket et les performances pour retrouver un élément sont de type O(n). Si la répartition des valeurs de hachage est équilibrée alors ces performances peuvent être de type O(log n).

Il existe plusieurs différences entre les classes Hashtable et HashMap bien qu'elles implémentent toutes les deux l'interface Map et ont un mode de fonctionnement similaire :

- La classe HashMap n'est pas synchronized contrairement à la classe Hashtable : dans un contexte monothread la classe HashMap est plus performante
- la classe HashMap autorise l'utilisation de null comme clé et comme valeur contrairement à la classe Hashtable
- Java 5 a introduit la classe ConcurrentHashMap comme une alternative plus performante à la classe Hashtable
- les Iterator de la classe HashMap sont de type fail-fast : une exception ConcurrentModificationException est levée si une modification de la structure de la collection est effectuée par un autre thread sans utiliser l'Iterator

14.5.5. La classe LinkedHashMap

La classe LinkedHashMap, ajoutée à Java 1.4, est une implémentation de type Map qui utilise une liste doublement chaînée pour maintenir par défaut ses éléments dans leur ordre d'insertion.

Cette collection permet de garantir l'ordre de ses éléments sans avoir un subir un surcoût comme par exemple lors de l'utilisation d'une classe TreeSet.

La classe LinkedHashMap possède plusieurs constructeurs :

Constructeur	Rôle
LinkedHashMap()	Créer une instance vide avec les propriétés par défaut : capacité initiale à 16 et facteur de charge à 0.75.
LinkedHashMap(int initialCapacity)	Créer une instance vide avec les propriétés : capacité initiale fournie et facteur de charge par défaut.
LinkedHashMap(int initialCapacity, float loadFactor)	Créer une instance vide avec les propriétés : capacité initiale et facteur de charge fournis
LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)	Créer une instance vide avec les propriétés : capacité initiale, facteur de charge et l'ordre d'accès fournis
LinkedHashMap(Map<? extends K,? extends V> m)	Créer une instance remplie avec les éléments de la collection fournie en paramètres

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.LinkedHashMap;
import java.util.Map;

public class TestLinkedHashMap {
  private static void afficherMap(final LinkedHashMap<String, String> map) {
    for (String s : map.keySet()) {
      System.out.print(s + " ");
    }

    System.out.println("");
    for (Map.Entry<String, String> s : map.entrySet()) {
      System.out.print(s.getKey() + " ");
    }
  }
  public static void main(final String[] args) {
    LinkedHashMap<String, String> map = new LinkedHashMap<String, String>();
    map.put(null, "");
    for (int i = 10; i < 19; i++) {
      map.put("" + i, "");
    }
    afficherMap(map);
  }
}
```

Résultat :

```
null 10 11 12 13 14 15 16 17 18
null 10 11 12 13 14 15 16 17 18
```

La surcharge du constructeur qui attend en paramètre le booléen accessOrder permet de préciser l'ordre des éléments :

- true : les éléments sont triés du moins accédés au plus accédés
- false : les éléments sont triés dans leur ordre d'insertion

Cette fonctionnalité peut être intéressante pour créer un cache de type LRU.

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.LinkedHashMap;
import java.util.Map;

public class TestLinkedHashMap {
  private static void afficherMap(final LinkedHashMap<String, String> map) {
    for (String s : map.keySet()) {
      System.out.print(s + " ");
    }

    System.out.println("");
    for (Map.Entry<String, String> s : map.entrySet()) {
      System.out.print(s.getKey() + " ");
    }
  }

  public static void main(final String[] args) {
    LinkedHashMap<String, String> map = new LinkedHashMap<String, String>(16, 0.75f, true);
    map.put(null, "");
    for (int i = 10; i < 19; i++) {
      map.put("" + i, "");
    }
    map.get("11");
    map.get("14");
    map.get("14");
    map.get("11");
    map.get("16");
    map.get("11");
    afficherMap(map);
  }
}
```

Résultat :

```
null 10 12 13 15 17 18 14 16 11
null 10
12 13 15 17 18 14 16 11
```

La classe LinkedHashMap peut être utilisée pour créer une copy d'une autre collection de type Map qui permettra son parcours toujours de la même façon.

Exemple :
`Map copie = new LinkedHashMap(m);`

L'ordre des éléments n'est pas modifié si un élément est de nouveau ajouté dans la collection alors qu'il est déjà présent dans cette dernière.

La méthode removeEldestEntry(Map.Entry<K,V>) peut être surchargée pour renvoyer un booléen qui précise si les éléments les plus anciens doivent être retirés de la collection.

La méthode containsValue(Object value) renvoie un booléen qui précise si une ou plusieurs de ses clés sont associées à la valeur fournie en paramètre.

Elle présente plusieurs caractéristiques :

- Elle implémente toutes les opérations optionnelles du type Map
- Elle permet l'ajout d'éléments null

La classe LinkedHashMap n'est pas synchronized, elle n'est donc pas thread-safe. Si plusieurs threads doivent ajouter ou supprimer des éléments dans la collection, il faut gérer manuellement la concurrence d'accès. Il est par exemple possible d'utiliser une instance de la collection retournée par la méthode synchronizedMap() de la classe Collections.

```
Map m = Collections.synchronizedMap(new LinkedHashMap());
```

Les Iterator de la classe LinkedHashMap sont de type fail-fast : les instances de type Iterator retournées par la méthode iterator() lèvent généralement une exception de type ConcurrentModificationException si une modification est effectuée dans la collection durant le parcours (exception faite des modifications réalisées avec la méthode remove() de l'Iterator).

Une LinkedHashMap, comme la classe HashMap, possède deux paramètres qui peuvent affecter ses performances lors de son utilisation : capacité initiale et facteur de charge. Cependant, contrairement à une HashMap, l'utilisation d' une capacité initiale largement surestimée est moins importante car le temps d'itération de cette collection n'est pas proportionnel à sa capacité.

Si la répartition dans les différents buckets réalisée grâce à la valeur de hachage de ses éléments est équitable, les méthodes add(), contains() et remove() sont exécutées avec des performances constantes. Le temps requis pour parcourir la collection est proportionnel à sa taille.

14.5.6. La classe TreeMap

La classe TreeMap, ajoutée à Java 1.2, est une Map qui stocke des éléments de manière triée dans un arbre de type rouge-noir (Red-black tree).

Les éléments de la collection sont triés selon l'ordre naturel de leur clé (s'is implémentent l'interface Comparable) ou en utilisant une instance de type Comparator fournie au constructeur de la collection.

Elle implémente les interfaces Map et SortedMap. Elle implémente aussi l'interface NavigableMap depuis Java 6.

La classe TreeMap propose plusieurs constructeurs dont un qui permet de préciser l'objet Comparable pour définir l'ordre dans la collection :

Constructeur	Rôle
TreeMap()	Constructeur par défaut qui crée une collection vide utilisant l'ordre naturel des clés des éléments
TreeMap(Comparator<? super K> comparator)	Créer une instance vide qui utilisera le Comparator fourni en paramètre pour déterminer l'ordre des éléments
TreeMap(Map<? extends K,? extends V> m)	Créer une instance contenant les éléments fournis en paramètres qui utilisera l'ordre naturel des clés des éléments

TreeMap(SortedMap<K,? extends V> m)	Créer une instance contenant les éléments fournis en paramètres

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.Map;
import java.util.TreeMap;

public class TestTreeMap {
  public static void main(final String[] args) {
    TreeMap<String, String> map = new TreeMap<String, String>();

    map.put("cle3", "valeur3");
    map.put("cle2", "valeur2");
    map.put("cle1", "valeur1");

    for (Map.Entry<String, String> element : map.entrySet()) {
      System.out.println(element.getKey() + " : " + element.getValue());
    }
  }
}
```

La classe TreeMap n'est pas synchronized. Pour obtenir une instance synchronized, il faut invoquer la méthode synchronizedSortedMap() de la classe Collections.

```
SortedMap m = Collections.synchronizedSortedMap(new TreeMap(...));
```

Toutes les instances de type Map.Entry retournées par les méthodes d'une TreeMap représente une vue des éléments de l'instance au moment où elles sont créées. La méthode setValue() de la classe Map.Entry lève une exception de type UnsupportedOperationException si elle est invoquée.

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.Map;
import java.util.TreeMap;

public class TestTreeMap {
  public static void main(final String[] args) {
    TreeMap<String, String> map = new TreeMap<String, String>();

    map.put("cle1", "valeur1");
    map.put("cle2", "valeur2");
    map.put("cle3", "valeur3");

    Map.Entry<String, String> dernier = map.lastEntry();
    dernier.setValue("valeur3 modifie");

    for (Map.Entry<String, String> element : map.entrySet()) {
      System.out.println(element.getKey() + " : " + element.getValue());
    }
  }
}
```

Résultat :

```
Exception in thread "main" java.lang.UnsupportedOperationException
       at java.util.AbstractMap$SimpleImmutableEntry.setValue(Unknown Source)
       at fr.jmdoudoux.dej.collections.TestTreeMap.main(TestTreeMap.java:15)
```

Pour changer la valeur associer à une clé, il faut utiliser la méthode put().

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.Map;
import java.util.TreeMap;

public class TestTreeMap {
  public static void main(final String[] args) {
    TreeMap<String, String> map = new TreeMap<String, String>();

    map.put("cle1", "valeur1");
    map.put("cle2", "valeur2");
    map.put("cle3", "valeur3");
    map.put("cle3", "valeur3 modifie");

    for (Map.Entry<String, String> element : map.entrySet()) {
      System.out.println(element.getKey() + " : " + element.getValue());
    }
  }
}
```

Résultat :

```
cle1 : valeur1
cle2 : valeur2
cle3 : valeur3 modifie
```

Les opérations basiques get(), put() et remove() s'exécutent dans un délai de type O(log n).

Les Iterator sont de type fail-fast.

14.5.7. La classe WeakHashMap

La classe WeakHashMap, ajoutée dans Java 1.2, est une implémentation d'une collection de type Map dont les clés sont stockées avec des références faibles (WeakReference). La classe WeakHashMap retire automatiquement les éléments dont la clé a été récupérée par le ramasse-miettes.

Elle hérite de la classe AbstractMap et implémente l'interface Map<K,V>.

Cette collection présente plusieurs caractéristiques :

- elle permet d'utiliser des valeurs null comme clé et valeur
- des éléments peuvent être supprimés sans avertissement par la collection

Une clé n'est pas stockée directement dans une WeakHashMap : c'est une référence faible sur l'instance qui est stockée. Un objet qui encapsule une valeur est stocké dans la WeakHashMap avec une référence forte. Il ne doit pas faire référence à sa clé sinon celle-ci ne sera pas récupérée par le ramasse-miettes en cas de besoin.

Si l'objet utilisé comme clé n'est plus référencé par d'autres objets, le ramasse-miettes va le récupérer et l'élément sera retiré de la collection.

Il faut être sûr de n'utiliser comme clés que des objets qui puissent être récupérés par le ramasse-miettes : par exemple, il ne faut surtout pas utiliser des chaînes de caractères dont la valeur est codée en dur.

Seules les clés sont stockées avec des références faibles : la valeur associée à l'élément est stockée sous la forme d'une référence forte.

La classe WeakHashMap possède quatre constructeurs :

Constructeur	Rôle
WeakHashMap()	Créer une instance vide de la collection dont la capacité initiale est de 16 et le facteur de charge est 0.75

WeakHashMap(int initialCapacity)	Créer une instance vide de la collection dont la capacité initiale est fournie en paramètre et le facteur de charge est 0.75
WeakHashMap(int initialCapacity, float loadFactor)	Créer une instance vide de la collection dont la capacité initiale et le facteur de charge sont fournies en paramètres
WeakHashMap(Map map)	Créer une instance remplie avec les éléments de la collection fournie en paramètre

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.Map;
import java.util.WeakHashMap;

public class TestWeakHashMap {
  public static void main(final String args[]) {
    String maCle = new String("1");

    Map<String, String> map = new WeakHashMap<String, String>();
    map.put(maCle, "Element1");
    map.put(new String("2"), "Element2");
    map.put(new String("3"), "Element3");

    System.out.println("map.size()=" + map.size());
    System.out.println("Demande d'execution du ramasse-miettes");
    System.gc();
    try {
      Thread.sleep(1000);
    } catch (InterruptedException ignored) {
    }

    System.out.println("map.size()=" + map.size());
    maCle = null;

    System.out.println("Demande d'execution du ramasse-miettes");
    System.gc();
    try {
      Thread.sleep(1000);
    } catch (InterruptedException ignored) {
    }

    System.out.println("map.size()=" + map.size());
  }
}
```

Résultat :

```
map.size()=3
Demande d'execution du ramasse-miettes
map.size()=1
Demande d'execution du ramasse-miettes
map.size()=0
```

Le fonctionnement d'un objet de type WeakHashMap repose en partie sur l'activité du ramasse-miettes qui peut détruire des objets utilisés comme clés : ces objets seront alors supprimés de la collection. Ainsi le contenu de la collection peut être modifié alors qu'aucune opération n'est explicitement invoquée : ce comportement est spécifique pour une collection de type WeakHashMap.

Il est préférable que l'implémentation de la méthode equals() des objets utilisés comme clés utilise un test d'égalité sur l'identité des objets avec l'opérateur ==. Ceci garantit qu'un objet ne puisse pas être recréé et être égal à l'instance détruite par le ramasse-miettes.

La classe WeakHashMap n'est pas synchronized. Pour obtenir une instance synchronized, il faut invoquer la méthode synchronizedMap() de la classe Collections.

```
WeakHashMap m = Collections.synchronizedMap(new WeakHashMap(...));
```

Les performances de cette collection sont similaires à celles de la classe HashMap.

Les Iterator de la classe WeakHashMap sont de type fail-fast.

Il est tentant d'utiliser la classe WeakHashMap comme implémentation d'un cache.

Il est préférable d'utiliser la classe WeakHashMap pour associer des données à un objet : dans ce cas, la clé est l'objet et la valeur contient les informations associées. La classe WeakHashMap va garantir qu'il n'y aura pas de fuites de mémoire liées à l'oubli de suppression de l'élément dans la collection lorsque l'objet n'est plus utilisé.

Ceci peut par exemple être pratique si l'objet est accédé par plusieurs threads. Une WeakHashMap est bien adaptée lorsqu'il n'est pas facile de connaitre le moment où les threads n'auront plus besoin des informations et pourront retirer l'objet les contenant de la collection.

14.5.8. La classe EnumMap

La classe java.util.EnumMap<K extends Enum<K>, V>, ajoutée à Java 5, est une implémentation de l'interface Map qui ne peut utiliser comme clés que les éléments d'une énumérations. Les traitements de la classe sont optimisés en fonction de cette particularité.

Elle hérite de la classe AbstractMap(K, V)

Elle présente plusieurs caractéristiques :

- toutes les valeurs utilisables comme clés doivent appartenir à la même énumération
- il n'est pas possible d'utiliser la valeur null comme clé sinon une exception de type NullPointerException est levée lors de l'ajout de l'élément
- la valeur associée à la clé peut être null.

En interne, le stockage des éléments de la collection se fait dans un tableau.

Les Iterator obtenus par les méthodes keySet(), entrySet() et values() sont ordonnés dans l'ordre de définition des éléments de l'énumération. Ces itérateurs ne lèvent jamais d'exception de type ConcurrentModificationException.

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.EnumMap;
import java.util.Iterator;

public class TestEnumMap {
  public enum ETAT {
    NOUVEAU, EN_COURS, EN_PAUSE, INDETERMINE, TERMINE;
  }

  public static void main(final String args[]) {
    EnumMap<ETAT, String> etatLibelle = new EnumMap<ETAT, String>(ETAT.class);

    etatLibelle.put(ETAT.NOUVEAU, "Nouvelle tache");
    etatLibelle.put(ETAT.EN_COURS, "Tache en cours d'exécution");
    etatLibelle.put(ETAT.EN_PAUSE, "Execution de la tache en pause");
    etatLibelle.put(ETAT.TERMINE, "Tache terminée");

    System.out.println("etatLibelle : " + etatLibelle);
    System.out.println("EnumMap cle : " + ETAT.NOUVEAU + ", valeur : "
        + etatLibelle.get(ETAT.NOUVEAU));
    Iterator<ETAT> enumKeySet = etatLibelle.keySet().iterator();
    while (enumKeySet.hasNext()) {
      ETAT currentState = enumKeySet.next();
      System.out.println("cle : " + currentState + ", valeur : "
          + etatLibelle.get(currentState));
    }

    if (etatLibelle.containsKey(ETAT.NOUVEAU)) {
      System.out.println("etatLibelle contient la cle : " + ETAT.NOUVEAU);
    }
    if (!etatLibelle.containsKey(ETAT.INDETERMINE)) {
      System.out.println("etatLibelle ne contient pas la cle : "
          + ETAT.INDETERMINE);
    }
  }
}
```

Résultat :

```
etatLibelle : {NOUVEAU=Nouvelle tache, EN_COURS=Tache en cours
d'exécution, EN_PAUSE=Execution de la tache en pause, TERMINE=Tache terminée}
EnumMap cle : NOUVEAU, valeur : Nouvelle tache
cle : NOUVEAU, valeur : Nouvelle tache
cle : EN_COURS, valeur : Tache en cours d'exécution
cle : EN_PAUSE, valeur : Execution de la tache en pause
cle : TERMINE, valeur : Tache terminée
etatLibelle contient la cle : NOUVEAU
etatLibelle ne contient pas la cle : INDETERMINE
```

La classe EnumMap n'est pas synchronized. Si plusieurs threads doivent accéder à la collection et qu'au moins l'un d'entre-eux modifie la collection, il faut utiliser une instance de la collection retournée par la méthode synchronizedMap() de la classe Collections.

```
Map<MonEnum, String> map = Collections.synchronizedMap(new EnumMap<MonEnum, String>());
```

Les opérations de base s'exécutent en temps constant. Les performances des opérations d'une EnumMap sont généralement meilleures que leur équivalent de la classe HashMap avec les mêmes éléments.

Il existe plusieurs différences entre les classes EnumMap et HashMap :

- avec la classe EnumMap seuls les éléments d'une énumération sont utilisables comme clés alors que la class HashMap accepte n'importe quels types d'objets
- dans une instance de type EnumMap, toutes les clés sont différentes : elle n'utilise pas la méthode hashCode() car il ne peut pas y avoir de collision sur les clés
- la classe EnumMap est plus performante qu'une classe HashMap qui possède les mêmes éléments

14.5.9. La classe IdentityHashMap

La classe java.util.IdentityHashMap<K extends Enum<K>, V>, ajoutée à Java 1.4, est une implémentation de l'interface Map qui utilise un test d'égalité sur les références (habituellement les implémentations de l'interface Map utilise l'égalité des objets). Deux clés cle1 et cle2 sont donc égales si cle1==cle2.

De ce fait, la classe IdentityHashMap est une implémentation particulière de l'interface Map dont elle ne respecte pas le contrat qui précise que le test d'égalité sur des objets doit se faire en utilisant la méthode equals(). Ce n'est donc pas une implémentation à usage générale mais son utilisation est limitée à quelques cas bien particuliers comme par exemple conserver une trace des objets utilisés lors d'opérations de sérialisation ou de clonage.

Elle hérite de la classe AbstractMap(K, V) et implémente toutes les méthodes optionnelles de l'interface Map.

Elle possède plusieurs caractéristiques :

- en interne, le stockage des éléments de la collection se fait dans une Hashtable
- elle permet d'utiliser null comme clé et comme valeur
- elle ne garantit pas l'ordre dans lequel les éléments sont parcourus
- la valeur de hachage retournée par la méthode identityHashCode() de la classe System est utilisée pour calculer l'index du bucket dans la Hashtable interne

La classe IdentityHashMap a une propriété qui peut agir sur ses performances : le nombre maximum d'éléments prévus dans la collection. Cette propriété est utilisée pour définir le nombre de buckets. Si la taille de la collection devient insuffisante pour contenir les éléments, le stockage interne de la collection est agrandi et la méthode rehash() est invoquée. Cet agrandissement est coûteux.

La classe IdentityHashMap possède trois constructeurs :

Constructeur	Rôle
IdentityHashMap()	Créer une instance vide de la collection avec une taille maximale prévue par défaut de 21
IdentityHashMap(int expectedMaxSize)	Créer une instance vide de la collection avec une taille maximale prévue fournie en paramètre
IdentityHashMap(Map map)	Créer une instance qui contient les éléments fournis en paramètres

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.IdentityHashMap;

public class TestIdentityHashMap {
  public static void main(final String[] args) {
    IdentityHashMap<String, String> identityMap = new IdentityHashMap<String, String>();
    identityMap.put("1", "Element 1");
    identityMap.put(new String("1"), "Element 1_2");

    System.out.println("identityHashMap.size() = " + identityMap.size());
    System.out.println("identityHashMap = " + identityMap);

    identityMap.put("1", "Element 1 modifie");
    System.out.println("identityHashMap.size() = " + identityMap.size());
    System.out.println("identityHashMap = " + identityMap);
  }
}
```

Résultat :

```
identityHashMap.size() = 2
identityHashMap = {1=Element 1_2, 1=Element 1}
identityHashMap.size() = 2
identityHashMap = {1=Element 1_2, 1=Element 1 modifie}
```

La classe IdentityHashMap n'est pas synchronized. Si plusieurs threads doivent accéder à la collection et qu'au moins un d'entre-eux modifie la collection, il faut utiliser une instance de la collection retournée par la méthode synchronizedMap() de la classe Collections.

```
Map map = Collections.synchronizedMap(new IdentityHashMap());
```

Les Iterator de la classe IdentityHashMap sont de type fail-fast : les instances de type Iterator retournées par la méthode iterator() lèvent généralement une exception de type ConcurrentModificationException si une modification est effectuée dans la collection durant le parcours (exception faite des modifications réalisées avec la méthode remove() de l'Iterator).

Il existe plusieurs différences entre les classes IdentityHashMap et HashMap :

- la classe HashMap est une implémentation généraliste, la classe IdentityHashMap est une implémentation particulière
- la classe IdentityHashMap utilise l'opérateur == pour tester l'égalité des clés alors que la classe HashMap utilise la méthode equals()
- la classe HashMap utilise la méthode hashCode() des clés pour déterminer leur bucket alors que la classe IdentityHashMap utilise la méthode identityHashCode() de la classe System
- si les clés utilisées ne redéfinissent pas les méthodes equals() et hashCode(), le fonctionnement des classes IdentityHashMap et HashMap est similaire
- comme la classe IdentityHashMap n'utilise pas les méthodes equals() et hashCode(), ses performances sont meilleures que celles de la classe HashMap

- les clés utilisées dans une instance de type IdentityHashMap n'ont pas besoin d'être immuables puisqu'elles n'utilisent pas la méthode equals()

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.HashMap;
import java.util.IdentityHashMap;
import java.util.Map;

public class TestIdentityMapHashMap {
  public static void main(final String[] args) {
    Map<String, Integer> identityMap = new IdentityHashMap<String, Integer>();
    Map<String, Integer> hashMap = new HashMap<String, Integer>();

    identityMap.put("1", 1);
    identityMap.put(new String("1"), 2);
    identityMap.put("1", 3);

    hashMap.put("1", 1);
    hashMap.put(new String("1"), 2);
    hashMap.put("1", 3);

    System.out.println("identityMap.keySet().size() = "
        + identityMap.keySet().size());
    System.out.println("hashMap.keySet().size()     = "
        + hashMap.keySet().size());
  }
}
```

Résultat :

```
identityMap.keySet().size() = 2
hashMap.keySet().size()     = 1
```

14.5.10. L'interface NavigableMap

L'interface NavigableMap, ajoutée dans Java 6, hérite de l'interface SortedMap. Elle définit des fonctionnalités qui permettent le parcours de la collection dans l'ordre ascendant ou descendant grâce à plusieurs méthodes.

L'interface définit plusieurs méthodes qui peuvent être regroupées selon leur rôle :

- obtenir le premier ou le dernier élément : firstEntry(), lastEntry(), pollFirstEntry(), pollLastEntry()
- obtenir un sous-ensemble de la collection : subMap(), headMap(), tailMap()
- obtenir l'élément dont la clé est la plus approchante : ceilingEntry(), ceilingKey(), floorEntry(), floorKey(), higherEntry(), higherKey(), lowerEntry(), lowerKey()
- parcours de la collection en sens inverse : descendingMap(), descendingKeySet(), navigableKeySet()

Méthode	Rôle
Map.Entry<K,V> ceilingEntry(K key)	Renvoyer la paire clé/valeur correspondant à la plus petite clé supérieure ou égale à celle fournie en paramètre. Elle renvoie null si une telle clé n'est pas trouvée
K ceilingKey(K key)	Renvoyer la plus petite clé supérieure ou égale à celle fournie en paramètre. Elle renvoie null si une telle clé n'est pas trouvée
NavigableSet<K> descendingKeySet()	Renvoyer une collection qui permette le parcours de la collection dans le sens inverse de l'ordre des clés
NavigableMap<K,V> descendingMap()	Renvoyer une collection de paires clé/valeur permettant le parcours de la collection en sens inverse
Map.Entry<K,V> firstEntry()	Renvoyer la paire clé/valeur correspondant à la plus petite clé de la collection. Elle renvoie null si la collection est vide

Map.Entry<K,V> floorEntry(K key)	Renvoyer la paire clé/valeur correspondant à la plus grande clé inférieure ou égale à celle fournie en paramètre. Elle renvoie null si une telle clé n'est pas trouvée
K floorKey(K key)	Renvoyer la plus grande clé inférieure ou égale à celle fournie en paramètre. Elle renvoie null si une telle clé n'est pas trouvée
SortedMap<K,V> headMap(K toKey)	Renvoyer une collection qui est un sous-ensemble composé des paires clé/valeur dont la clé est strictement inférieure à celle fournie en paramètre
NavigableMap<K,V> headMap(K toKey, boolean inclusive)	Renvoyer une collection qui est un sous-ensemble composé des paires clé/valeur dont la clé est inférieure (ou égale si le paramètre inclusive vaut true) à celle fournie en paramètre
Map.Entry<K,V> higherEntry(K key)	Renvoyer la paire clé/valeur correspondant à la plus petite clé supérieure ou égale à celle fournie en paramètre. Elle renvoie null si une telle clé n'est pas trouvée
K higherKey(K key)	Renvoyer la plus petite clé supérieure ou égale à celle fournie en paramètre. Elle renvoie null si une telle clé n'est pas trouvée
Map.Entry<K,V> lastEntry()	Renvoyer la paire clé/valeur correspondant à la plus grande clé de la collection. Elle renvoie null si la collection est vide
Map.Entry<K,V> lowerEntry(K key)	Renvoyer la paire clé/valeur correspondant à la plus grande clé strictement inférieure à celle fournie en paramètre. Elle renvoie null si une telle clé n'est pas trouvée
K lowerKey(K key)	Renvoyer la plus petite clé de la collection strictement inférieure à celle fournie en paramètre. Renvoie null si aucune clé n'est trouvée
NavigableSet<K> navigableKeySet()	Renvoyer une collection de type NavigableSet contenant les clés de la collection
Map.Entry<K,V> pollFirstEntry()	Retirer de la collection et renvoyer la paire clé/valeur dont la clé est la plus petite. Elle renvoie null si la collection est vide
Map.Entry<K,V> pollLastEntry()	Retirer de la collection et renvoyer la paire clé/valeur dont la clé est la plus grande. Elle renvoie null si la collection est vide
NavigableMap<K,V> subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)	Renvoyer une collection, qui soit un sous-ensemble de la collection, contenant les paires clé/valeur dont la valeur des clés est comprise entre les valeurs de fromKey et toKey. Les paramètres fromInclusive et toInclusive précisent si ces valeurs doivent être incluses
SortedMap<K,V> subMap(K fromKey, K toKey)	Renvoyer un sous-ensemble de la collection qui va contenir les éléments dont les clés sont comprises entre fromKey incluse et toKey exclue
SortedMap<K,V> tailMap(K fromKey)	Renvoyer un sous-ensemble de la collection qui va contenir les éléments dont les clés sont plus grandes ou égales à fromKey
NavigableMap<K,V> tailMap(K fromKey, boolean inclusive)	Renvoyer un sous-ensemble de la collection qui va contenir les éléments dont les clés sont plus grandes que fromKey. L'élément dont la clé est égale est aussi inclus si le paramètre inclusive vaut true

L'API Collections propose plusieurs implémentations de l'interface NavigableMap : ConcurrentSkipListMap et TreeMap

14.5.11. L'interface ConcurrentNavigableMap

L'interface java.util.concurrent.ConcurrentNavigableMap, ajoutée à Java 1.6, définit les fonctionnalités d'une NavigableMap avec une gestion des accès concurrents pour elle-même et pour ses sous Map.

L'interface NavigableMap hérite des interfaces ConcurrentMap et NavigableMap.

L'API Collections ne propose qu'une seule implémentation de cette interface : la classe ConcurrentSkipListMap.

14.5.12. La classe ConcurrentSkipListMap

La classe ConcurrentSkipListMap, ajoutée à Java 6, est une implémentation de l'interface ConcurrentNavigableMap en utilisant un algorithme de type SkipList. Les fonctionnalités offertes par la classe ConcurrentSkipListMap peuvent être vues comme une fusion des fonctionnalités des classes ConcurrentHashMap et TreeMap.

Les éléments de cette collection sont triés selon leur ordre naturel en implémentant l'interface Comparable ou en utilisant une instance de type Compator fournie en paramètre du constructeur de la collection.

La classe ConcurrentSkipListMap présente plusieurs caractéristiques :

- elle permet la modification concurrente de la collection sans la bloquer dans son intégralité
- elle est optimisée pour les opérations de lectures qui sont non bloquantes
- par défaut les éléments sont triés selon leur ordre naturel ou selon l'ordre défini par l'instance de Comparator associée à la collection
- l'utilisation de null n'est pas possible ni pour la clé ni pour la valeur d'un élément

La classe java.util.ConcurrentSkipMap hérite de la classe abstraite AbstractMap et implémente l'interface ConcurrentNavigableMap. Elle implémente toutes les méthodes optionnelles des interfaces Map et Iterator.

La classe propose plusieurs constructeurs :

Constructeur	Rôle
ConcurrentSkipListMap()	Créer une instance vide dont les éléments seront triés selon leur ordre naturel
ConcurrentSkipListMap(Comparator<? super K> comparator)	Créer une instance vide dont les éléments seront triés selon leur l'ordre déterminé par l'instance fournie en paramètre
ConcurrentSkipListMap(Map<? extends K,? extends V> m)	Créer une instance qui va contenir les éléments de la collection fournie en paramètre triés selon leur ordre naturel
ConcurrentSkipListMap(SortedMap<K,? extends V> m)	Créer une instance qui va contenir les éléments de la collection fournie en paramètre triés selon leur ordre dans la collection

Les opérations de la classe ConcurrentSkipListMap sont thread-safe.

Les Iterator obtenus de la collection reflètent la liste des éléments à un instant donné, généralement le moment de la création de l'instance de type Iterator. Le parcours ne lève jamais d'exception de type ConcurrentModificationException. Le parcours ascendant est plus rapide que le parcours descendant.

Les opérations groupées proposées par les méthodes clear(), equals() et putAll() ne garantissent pas que leurs exécutions seront atomiques.

Toutes les instances de type Map.Entry retournées par les méthodes de classe ConcurrentSkipListMap sont une représentation des données au moment où l'instance est créée. Il n'est pas possible d'utiliser leur méthode setValue(). Pour modifier une valeur, il faut invoquer une des méthodes put(), putIfAbsent() ou replace().

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.Iterator;
import java.util.concurrent.ConcurrentSkipListMap;

public class TestConcurrentSkipListMap {
  public static void main(final String[] args) {
    final ConcurrentSkipListMap<MaTache, String> map =
      new ConcurrentSkipListMap<MaTache, String>();

    System.out.println("debut");
    final Thread modificateur = new Thread(new Runnable() {
      @Override
      public void run() {
        MaTache[] MesTaches = new MaTache[5];
        for (int i = 1; i <= 5; i++) {
          MesTaches[i - 1] = new MaTache(6 - i, "Tache " + i);
        }

        for (int j = 1; j <= 100; j++) {
          if (j % 2 == 0) {
            for (int i = 1; i <= 5; i++) {
              MaTache element = MesTaches[i - 1];
              System.out.println("insertion element " + element);
              map.putIfAbsent(element, "description " + i);
            }
            System.out.println("taille de la queue=" + map.size());
          } else {
            for (int i = 1; i <= 5; i++) {
              MaTache element = MesTaches[i - 1];
              System.out.println("retirer element " + element);
              map.remove(element);
            }
          }
        }
      }
    }, "Modificateur");

    modificateur.start();

    Thread iterateur = new Thread(new Runnable() {
      @Override
      public void run() {
        int i = 0;
        while (modificateur.isAlive()) {
          Iterator<MaTache> iterator = null;

          if (i % 2 == 0) {
            iterator = map.navigableKeySet().iterator();
          } else {
            iterator = map.descendingKeySet().iterator();
          }

          StringBuilder contenu = new StringBuilder("[");

          while (iterator.hasNext()) {
            contenu.append(iterator.next().getDescription());
            if (iterator.hasNext()) {
              contenu.append(", ");
            }
          }
          contenu.append("]");

          System.out.println("Contenu=" + contenu);
          i++;
        }
      }
    }, "iterateur");

    iterateur.start();
    try {
      modificateur.join();
      iterateur.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.println("fin");
  }
}
```

Les performances des opérations basiques se situent en moyenne dans un temps dont l'expression est en log(n).

Le temps d'exécution de la méthode size() est variable car le nombre d'éléments de la collection doit être recalculé en la reparcourant.

La classe ConcurrentSkipListMap offre une meilleure scalabilité face aux accès concurrents que la classe ConcurrentHashMap mais en contrepartie les temps d'accès aux éléments ne sont pas constants. Cette contrepartie est liée au fait que la collection soit une SortedMap.

14.5.13. L'interface ConcurrentMap

L'interface ConcurrentMap, ajoutée à Java 1.5, définit les méthodes d'une collection qui est capable de gérer les accès concurrents lors des opérations de modifications de ces éléments.

Une collection de type Map est fréquemment utilisée dans un contexte multithread comme par exemple pour implémenter un cache d'objets simple.

Utiliser une instance synchronized de type Map retournée par la méthode synchronizedMap() de la classe Collections ne permet pas d'obtenir de bonnes performances en cas de nombreux accès concurrents car c'est l'instance de la collection elle-même sur laquelle se fait le verrou lors de l'invocation d'une méthode.

De plus, une Map synchronized ne garantit pas non plus l'atomicité de ses opérations qui nécessitent l'invocation de plusieurs méthodes : par exemple, pour ajouter un élément dans la collection, il est nécessaire de vérifier au préalable qu'elle ne contient pas déjà la clé en utilisant la méthode containsKey() ou get(). Comme le verrou est posé durant l'invocation de chaque méthode, il est possible que deux threads qui tentent d'ajouter un élément entraînent une race condition.

Par exemple, le premier thread obtient le verrou en invoquant la méthode containsKey() qui renvoie false, le second obtient à son tour le verrou en invoquant la méthode containsKey() qui renvoie false, le premier thread obtient le verrou en invoquant la méthode put() pour ajouter l'élément, le premier thread obtient le verrou en invoquant la méthode put() pour écraser l'élément ajouté par le premier thread.

Ce type de problématique est difficile à détecter et à reproduire : l'interface ConcurrentMap, ajoutée dans Java 5, hérite de l'interface Map. Elle définit quatre méthodes dont le comportement est atomique

Méthode	Rôle
V putIfAbsent(K key, V value)	Ajouter un élément dans la collection de manière atomique uniquement si la clé n'est pas déjà présente dans la collection. Renvoie null si l'élément est ajouté sinon renvoie la valeur associée à la clé (qui peut être null)
boolean remove(Object key, Object value)	Retirer un élément de la collection de manière atomique si la clé est présente dans la collection et est associée à la valeur fournie en paramètre. Renvoie un booléen qui indique le succès de l'opération
V replace(K key, V value)	Modifier la valeur associée à la clé de manière atomique uniquement si la clé est présente dans la collection. Renvoie null si l'élément est modifié sinon renvoie la valeur associée à la clé (qui peut être null)
boolean replace(K key, V oldValue, V newValue)	Modifier un élément de la collection de manière atomique si la clé est présente dans la collection et est associée à la valeur fournie au paramètre oldValue. Renvoie un booléen qui indique le succès de l'opération

L'API Collections propose deux implémentations de l'interface ConcurrentMap : ConcurrentHashMap et ConcurrentSkipListMap.

14.5.14. La classe ConcurrentHashMap

La classe ConcurrentHashMap, ajoutée à Java 1.5, implémente l'interface ConcurrentMap qui utilise une HashMap en garantissant les accès concurrents et les performances. Son but est de remplacer la classe Hashtable car elle est similaire à celle-ci avec une meilleure gestion des accès concurrents.

La classe ConcurrentHashMap implémente toutes les méthodes optionnelles des interfaces Map et Iterator.

Elle présente plusieurs caractéristiques :

- le mode de fonctionnement de cette collection est similaire à celui d'une Hashtable
- elle gère la concurrence d'accès et offre de très bonnes performances notamment dans le cas de forte concurrence
- les accès pour obtenir un élément ne sont pas bloquants
- la collection ne permet pas l'utilisation de null comme clé ou valeur.

Les accès pour retrouver un élément ne sont pas bloquants. Les mises à jour de la collection ne bloquent pas l'intégralité de la HashMap utilisée en interne car elle utilise des segments. Ce paramètre influe sur les performances lors des mises à jour concurrentes : il correspond au nombre de segments dans lesquels la collection va être découpée. Chaque thread qui effectue une mise à jour le fait dans son propre segment pour éviter la contention.

Idéalement, la valeur du paramètre concurrencyLevel doit être au moins égale au nombre de threads qui peuvent mettre à jour la collection de manière concurrente. Si la valeur est trop petite, il y a un risque d'avoir de la contention. Si la valeur est vraiment trop grande, il y a une consommation excessive de la mémoire requise. Par exemple, la valeur 1 est utilisable si un seul thread peut mettre à jour la collection et que les autres threads accèdent à la collection en lecture seulement.

Un des constructeurs attend un paramètre nommé concurrencyLevel qui correspond à ce nombre de segments. La valeur par défaut du paramètre concurrencyLevel est 16.

Deux autres paramètres influent sur les performances de la collection :

- la capacité initiale (initialCapacity) dont la valeur par défaut est 16
- le facteur de charge (loadFactor) dont la valeur par défaut est 0,75

Plusieurs surcharges du constructeur permettent de préciser ces paramètres afin d'optimiser les performances.

Constructeur	Rôle
ConcurrentHashMap()	Créer une collection vide avec les paramètres par défaut
ConcurrentHashMap(int initialCapacity)	Créer une collection vide avec la capacité initiale fournie en paramètre
ConcurrentHashMap(int initialCapacity, float loadFactor)	Créer une collection vide avec la capacité initiale et le facteur de charge fournis en paramètres
ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel)	Créer une collection vide avec la capacité initiale, le facteur de charge et le niveau de concurrence fournis en paramètres
ConcurrentHashMap(Map<? extends K,? extends V> m)	Créer une collection initialisée avec la collection de type Map fournie en paramètre

Généralement les opérations qui permettent d'obtenir un élément de la collection ne sont pas bloquantes et peuvent être réalisées en concurrence avec des opérations de mises à jour : la valeur retournée est alors le résultat de la dernière opération unitaire de mise à jour entièrement terminée. Si les opérations de mises à jour concernent plusieurs éléments (clear() ou putAll() par exemple), alors la ou les valeurs retournées peuvent ne contenir que tout ou partie des mises à jour en cours de réalisation.

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.Map;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentMap;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.atomic.AtomicInteger;

public class TestConcurrentHashMap {
  private final ConcurrentMap<String, AtomicInteger> cache      =
    new ConcurrentHashMap<String, AtomicInteger>(3, 0.75f, NB_THREADS);

  private final CyclicBarrier                        barriere   =
    new CyclicBarrier(NB_THREADS);
  private static final int                           NB_THREADS = 50;

  public static void main(final String[] args) {
    System.out.println("debut");
    TestConcurrentHashMap tchm = new TestConcurrentHashMap();
    tchm.exec();
    System.out.println("fin");
  }

  private void ajouter(final String cle) {
    AtomicInteger value;
    value = cache.putIfAbsent(cle, new AtomicInteger(1));
    if (value != null) {
      value.incrementAndGet();
    }
  }

  private void exec() {
    final ExecutorService executor = Executors.newFixedThreadPool(NB_THREADS);
    try {
      for (int i = 0; i < NB_THREADS; i++) {
        executor.submit(new Runnable() {
          @Override
          public void run() {
            try {
              barriere.await();
              for (int i = 0; i < 1000000; i++) {
                ajouter("chaine1");
                ajouter("chaine2");
                ajouter("chaine3");
              }
            } catch (InterruptedException e) {
              e.printStackTrace();
            } catch (BrokenBarrierException e) {
              e.printStackTrace();
            }
          }
        });
      }
    } finally {
      executor.shutdown();
    }
    while (!executor.isTerminated()) {
      try {
        Thread.sleep(100);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    for (Map.Entry<String, AtomicInteger> entry : cache.entrySet()) {
      System.out.println("[" + entry.getKey() + ", " +
        entry.getValue() + ']');
    }
  }
}
```

Résultat :

```
debut
[chaine1, 50000000]
[chaine3, 50000000]
[chaine2, 50000000]
fin
```

La classe ConcurrentHashMap est optimisée pour les opérations courantes mais certaines opérations peuvent être coûteuses : c'est notamment le cas de la méthode size() qui pose un verrou sur tous les segments pour calculer le nombre d'éléments que chacun contient.

Les parcours de la collection avec un Iterator reflètent l'état de la collection à un instant donné ou au moment de la création de l'Iterator. Les Iterator créés par cette collection ne lève jamais d'exception de type ConcurrentModificationException si une modification de structure est réalisée dans la collection durant son parcours. Ces Iterator ne sont toutefois pas prévus pour être utilisés par plusieurs threads.

14.5.15. Le choix d'une implémentation de type Map

Le JDK contient plusieurs implémentations de l'interface Map pour un usage spécifique :

- la classe EnumMap ne doit être utilisée que si les clés sont des éléments d'une énumérations
- la classe WeakHashMap stocke les clés avec des références faibles
- la classe IdentityHashMap utilise un test d'égalité sur les références de ses clés

Le JDK contient plusieurs implémentations de l'interface Map pour un usage généraliste qui peuvent selon les besoins :

- maintenir un ordre des clés
- gérer des accès concurrents

Ordre des clés	Pas d'accès concurrent	Gestion des accès concurrents
Aucun	HashMap	Hashtable ConcurrentHashMap
Trié	TreeMap	ConcurrentSkipListMap
Fixe	LinkedHashMap	

Pour une Map triée, il faut utiliser la classe TreeMap s'il n'y a pas d'utilisation concurrente sinon il faut utiliser la classe ConcurrentSkipListMap.

Trois classes sont utilisables dans un contexte généraliste et non concurrentiel. Le critère de choix est essentiellement l'ordre de tri des éléments que la collection doit utiliser :

- HashMap : aucun ordre précis pour les éléments qui doivent avoir l'implémentation de leurs méthodes hashCode() et equals() correctement codées
- TreeMap : l'ordre naturel des éléments ou l'ordre défini par l'instance de type Comparator associée à la collection
- LinkedHashMap : l'ordre des éléments est leur ordre d'insertion

Si la collection peut être utilisée de manière concurrente, plusieurs classes peuvent être mises en oeuvre :

- Hashtable : les méthodes de cette classe étant synchronized, les performances ne sont pas trop bonnes sous forte concurrence. De plus, certaines opérations qui nécessitent l'invocation de deux ou plusieurs méthodes ne sont pas atomiques
- ConcurrentHashMap : les verrous sont uniquement posés sur les éléments en cours de modification et la méthode get() n'est pas bloquante
- ConcurrentSkipListMap : maintient l'ordre de ses éléments et propose un parcours rapide au prix d'un surcoût lors de l'insertion

Le tableau ci-dessous compare les performances relatives de trois opérations permettant d'obtenir des données de plusieurs instances de type Map.

	get()	containsKey()	next()
HashMap	O(1)	O(1)	O(h/n)
LinkedHashMap	O(1)	O(1)	O(1)
IdentityHashMap	O(1)	O(1)	O(h/n)
EnumMap	O(1)	O(1)	O(1)
TreeMap	O(log n)	O(log n)	O(log n)
ConcurrentHashMap	O(1)	O(1)	O(h/n)
ConcurrentSkipListMap	O(log n)	O(log n)	O(1)

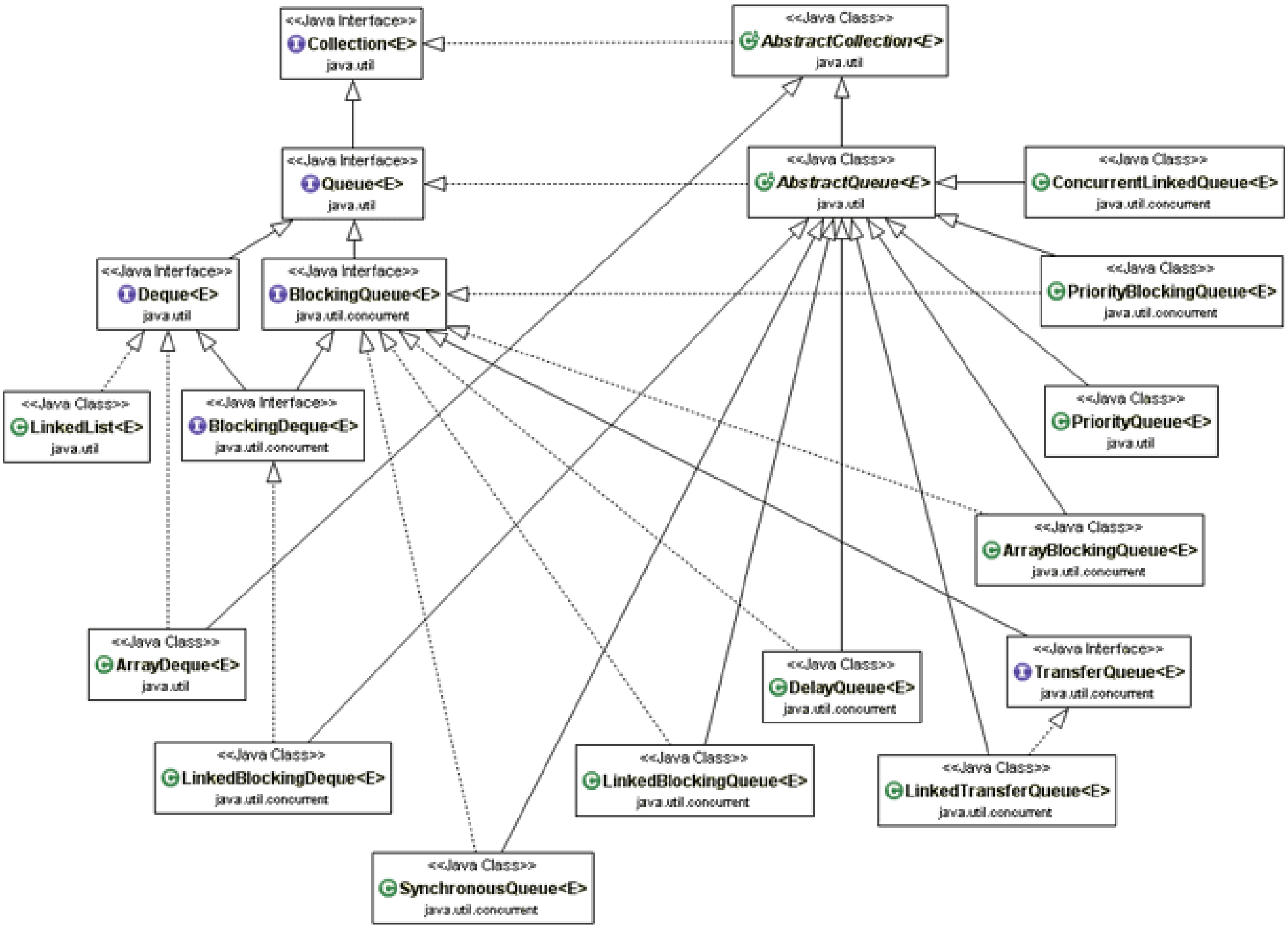
Dans le tableau ci-dessus, h représente la capacité de la collection et n le nombre d'éléments.

14.6. Les collections de type Queue : les files

Une Queue est une collection qui stocke des éléments dans un certain ordre avant d'être consommés pour être traités.

La plupart des implémentations proposées par le framework Collection utilise l'ordre FIFO (First In, First Out) mais l'ordre peut être différent.

La majorité des implémentations sont dans le package java.util.concurrent



14.6.1. L'interface Queue

L'interface Queue, ajoutée dans Java 5, définit les fonctionnalités pour une file d'objets (une collection qui permet de stocker des éléments avant leur traitement).

Une file propose trois opérations standard :

- ajouter un élément
- obtenir un élément
- consulter le prochain élément disponible : cette opération ne le retire pas de la collection

L'interface Queue propose des méthodes pour deux comportements différents en cas d'échec de ses opérations par exemple si la collection est vide ou pleine : le renvoie d'un booléen qui indique le succès de l'opération ou la levée d'une exception.

L'interface Queue définit plusieurs méthodes :

Méthode	Rôle
E element()	Consulter le premier élément disponible sans le retirer de la collection. Cette méthode lève une exception si la collection est vide
boolean offer(E o)	Ajouter l'élément dans la collection. Le booléen indique si l'ajout a réussi ou non
E peek()	Consulter le premier élément disponible sans le retirer de la collection. Cette méthode renvoie null si la collection est vide
E poll()	Obtenir le premier élément et le retirer de la file. Cette méthode renvoie null si la collection est vide
E remove()	Obtenir le premier élément et le retirer de la file. Cette méthode lève une exception si la collection est vide

Les méthodes pour ajouter un élément à la fin de la collection ou obtenir l'élément au début de la collection peuvent être classées en deux catégories selon qu'elles lèvent une exception ou retournent une valeur spéciale en cas d'échec :

	Lever une exception	Retourner une valeur spéciale
Ajouter un élément à la fin	add()	offer()
Obtenir et retirer le premier élément	remove()	poll()
Obtenir sans le retirer le premier élément	element()	peek()

La méthode add() est héritée de l'interface collection : elle renvoie un booléen qui ne peut valoir false que si l'ajout est impossible car l'élément est déjà présent dans la collection. Dans les autres cas d'erreurs, la méthode add() doit lever une exception. C'est la différence avec la méthode offer() qui ne lève pas d'exceptions.

Il n'y a que la classe LinkedList qui permet l'ajout d'éléments null dans les implémentations de l'interface Queue proposées par l'API Collections.

14.6.2. La classe AbstractQueue

La classe abstraite AbstractQueue, ajoutée à Java 5.0, est la classe mère de la plupart des collections de type Queue. Ces collections ne doivent pas accepter de valeur null.

Elle implémente les interfaces Collection, Iterable et Queue.

L'API Collections propose plusieurs classes filles : ArrayBlockingQueue, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, PriorityBlockingQueue, PriorityQueue et SynchronousQueue.

14.6.3. L'interface TransferQueue

L'interface TransferQueue, ajoutée dans Java 7, définit les fonctionnalités d'une collection dans laquelle les producteurs peuvent attendre qu'un consommateur reçoive un élément avant de pouvoir ajouter un nouvel élément dans la collection.

L'interface TransferQueue hérite de l'interface BlockingQueue.

Méthode	Rôle
int getWaitingConsumerCount()	Retourner une estimation du nombre de consommateurs qui attendent de recevoir un élément (en ayant invoqué les méthodes take() ou poll() avec un timeout)
boolean hasWaitingConsumer()	Retourner une booléen si au moins un consommateur attend de recevoir un élément (en ayant invoqué les méthodes take() ou poll() avec un timeout)
void transfer(E e)	Transférer un élément à un consommateur de manière bloquante (attente jusqu'à ce que l'élément soit consommé)
boolean tryTransfer(E e)	Transférer un élément à un consommateur : cet élément doit être immédiatement consommé si possible. Renvoie un booléen qui précise si l'élément est consommé
boolean tryTransfer(E e, long timeout, TimeUnit unit)	Transférer un élément à un consommateur : cet élément doit être consommé si possible dans le timeout précisé en paramètre. Renvoie un booléen qui précise si l'élément est consommé

Lors de l'utilisation d'une collection de type TransfertQueue, il est possible de choisir si l'ajout d'un nouvel élément doit être bloquant en utilisant la méthode transfert() ou non bloquant en utilisant la méthode put().

Si des éléments sont déjà dans la collection l'invocation de la méthode transfer() sera bloquante jusqu'à ce que tous les éléments aient été consommés.

Java 7 ne propose qu'une seule implémentation de cette interface : LinkedTransfertQueue.

14.6.4. La classe LinkedTransferQueue

La classe LinkedTransferQueue, ajoutée dans Java 7, est une implémentation d'une collection de type TransferQueue qui utilise en interne une LinkedList.

Elle présente plusieurs caractéristiques :

- c'est une queue de type FIFO
- la taille de la collection n'est pas bornée
- elle utilise des opérations de type CAS pour être non bloquante

Elle hérite de la classe AbstractQueue et implémente l'interface TransferQueue. Elle implémente toutes les méthodes optionnelles des interfaces Collection et Iterator.

La classe LinkedTransferQueue possède deux constructeurs :

Constructeur	Rôle
LinkedTransferQueue()	Créer une collection vide
LinkedTransferQueue(Collection<? Extends E> c)	Créer une collection qui va contenir les éléments de la collection fournie en paramètre

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.concurrent.LinkedTransferQueue;

public class TestLinkedTransferQueue {
  private static final int NB_TACHES = 5;
  public static void main(final String[] args) {
    final LinkedTransferQueue<MaTache> queue = new
      LinkedTransferQueue<MaTache>();

    final Thread producteur = new Thread(new Runnable() {
      @Override
      public void run() {
        for (int i = 1; i <= NB_TACHES; i++) {
          MaTache maTache = new MaTache(NB_TACHES - i, "Tache " + i);
          try {
            System.out.println("Ajout de " + maTache);
            queue.transfer(maTache);
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
        }
      }
    }, "Producteur");

    Thread consommateur = new Thread(new Runnable() {
      @Override
      public void run() {
        while (true) {
          MaTache maTache = null;
          try {
            maTache = queue.take();
            System.out.println("Traitement de " + maTache);
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
        }
      }
    }, "Consommateur");

    producteur.start();
    consommateur.start();
  }
}
```

Résultat :

```
Ajout de MaTache [priorite=4, description=Tache 1]
Traitement de MaTache [priorite=4, description=Tache 1]
Ajout de MaTache [priorite=3, description=Tache 2]
Ajout de MaTache [priorite=2, description=Tache 3]
Traitement de MaTache [priorite=3, description=Tache 2]
Traitement de MaTache [priorite=2, description=Tache 3]
Ajout de MaTache [priorite=1, description=Tache 4]
Traitement de MaTache [priorite=1, description=Tache 4]
Ajout de MaTache [priorite=0, description=Tache 5]
Traitement de MaTache [priorite=0, description=Tache 5]
```

Le temps d'exécution de la méthode size() n'est pas constant : toute la collection doit être parcourue pour déterminer le nombre d'éléments.

Les opérations groupées (addAll(), removeAll(), retainAll(), containsAll(), equals() et toArray()) n'offrent aucune garantie de s'exécuter de manière atomique. Par exemple, le parcours des éléments de la collection durant leur exécution peut ne voir qu'une partie des éléments impactés.

14.6.5. La classe PriorityQueue

La classe PriorityQueue, ajoutée à Java 1.5, hérite de la classe AbstractQueue. Les éléments de la collection sont ordonnés soit par leur ordre naturel soit par l'utilisation par la collection d'une instance de type Comparator fournie en paramètre du constructeur.

Si la collection utilise l'ordre naturel alors tous les éléments qu'elle contient doivent implémenter l'interface Comparable sinon une exception de type ClassCastException est levée.

Les opérations pour obtenir un élément (poll(), remove(), peek() et element()) renvoie le premier élément de la collection. Le premier élément de la collection est le plus petit selon l'ordre de classement des éléments par la collection.

Une collection de type PriorityQueue ne peut pas contenir d'élément null.

Elle implémente toutes les méthodes optionnelles des interfaces Collection et Iterator.

La taille d'une collection de type PriorityQueue n'est pas bridée mais elle possède une capacité interne initiale qui correspond à la taille du tableau dans lequel les éléments vont être stockés. La taille de ce tableau peut évoluer selon le nombre d'éléments contenu dans la collection.

La classe PriorityQueue possède plusieurs constructeurs :

Constructeur	Rôle
PriorityQueue()	Collection qui utilise l'ordre naturel avec une capacité initiale de 11 éléments
PriorityQueue(Collection<? extends E> c)	Collection initialisée avec les éléments de la collection fournie en paramètre qui utilise leur ordre naturel sauf si la collection est de type PriorityQueue ou SortedSet. Dans ce cas l'ordre de la collection est préservé
PriorityQueue(int initialCapacity)	Collection qui utilise l'ordre naturel dont la capacité initiale est fournie en paramètre
PriorityQueue(int initialCapacity, Comparator<? super E> comparator)	Collection qui utilise l'ordre du Comparator et la capacité initiale fournie en paramètre
PriorityQueue(PriorityQueue<? extends E> c)	Collection initialisée avec les éléments de la collection fournie en paramètre, en respectant l'ordre de celle-ci
PriorityQueue(SortedSet<? extends E> c)	Collection initialisée avec les éléments de la collection fournie en paramètre, en respectant l'ordre de celle-ci

L'obtention d'un élément se fait dans l'ordre dans lequel la collection gère ses éléments : soit en utilisant l'ordre naturel des objets (en implémentant l'interface Comparable) soit en utilisant l'instance de type Comparator fournie au constructeur qui a créé l'instance de la collection.

Une collection de type PriorityQueue accepte d'avoir des doublons ou des éléments qui possèdent la même priorité. Il n'y a aucune garantie sur l'ordre d'obtention de ces éléments : la seule solution est d'être suffisamment discriminant dans l'algorithme de comparaison utilisé par la collection pour déterminer l'ordre des éléments.

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.PriorityQueue;

public class TestPriorityQueue {
  public static void main(final String[] args) {
    PriorityQueue<MaTache> queue = new PriorityQueue<MaTache>();
    for (int i = 1; i <= 5; i++) {
      MaTache element = new MaTache(6 - i, "Tache" + i);
      System.out.println("insertion element " + element);
      queue.add(element);
    }

    System.out.println("taille de la queue=" + queue.size());
    for (int i = 1; i <= 5; i++) {
      MaTache element = queue.poll();
      System.out.println(element);
    }
  }
}
```

Résultat :

```
insertion element MaTache [priorite=5, description=Tache1]
insertion element MaTache [priorite=4, description=Tache2]
insertion element MaTache [priorite=3, description=Tache3]
insertion element MaTache [priorite=2, description=Tache4]
insertion element MaTache [priorite=1, description=Tache5]
taille de la queue=5
MaTache [priorite=1, description=Tache5]
MaTache [priorite=2, description=Tache4]
MaTache [priorite=3, description=Tache3]
MaTache [priorite=4, description=Tache2]
MaTache [priorite=5, description=Tache1]
```

Les Iterator de la classe PriorityQueue sont de type fail-fast. L'ordre de parcours des éléments avec un Iterator obtenu en invoquant la méthode iterator() n'est pas garanti.

La classe PriorityQueue n'est pas thread-safe : si plusieurs threads doivent pouvoir modifier cette collection, il faut utiliser la classe PriorityBlockingQueue.

14.6.6. La classe ConcurrentLinkedQueue

La classe ConcurrentLinkedQueue, ajoutée à Java 5, est une collection de type FIFO implémentée sous la forme d'une LinkedList.

La classe ConcurrentLinkedQueue possède plusieurs caractéristiques :

- l'ordre des éléments est de type FIFO (First In, First Out) : un nouvel élément est toujours ajouté à la fin de la collection et l'élément obtenu est le premier. Ainsi, le premier élément est celui qui est le plus ancien dans la collection et le dernier élément est celui qui est le plus récent dans la collection.
- la collection gère les accès concurrents qui sont faits sur elle
- elle ne permet pas l'ajout d'un élément null
- la taille de la collection n'est pas bridée.

L'implémentation de la classe ConcurrentLinkedQueue utilise un algorithme de type CAS pour gérer les opérations concurrentes.

La classe ConcurrentLinkedQueue possède deux constructeurs :

Constructeur	Rôle
ConcurrentLinkedQueue()	Créer une collection vide
ConcurrentLinkedQueue(Collection<? Extends E>)	Créer une collection avec les éléments de la collection fournie en paramètre. Les éléments sont insérés dans l'ordre du parcours des éléments de la collection avec son Iterator.

Cette classe implémente toutes les méthodes optionnelles de l'interface Queue et les instances de type Iterator obtenues implémentent toutes les méthodes optionnelles de l'interface Iterator.

Un Iterator obtenu à partir de la collection permet le parcours des éléments de la collection à un moment donné ou au moment de la création de l'Iterator. Ce parcours peut se faire en concurrence avec des modifications dans le contenu de la collection. Ce parcours ne lèvera jamais d'exceptions de type ConcurrentModificationException et un même élément ne pourra être obtenu qu'une seule fois lors du parcours.

Le temps d'exécution de la méthode size() n'est pas constant car elle doit parcourir les éléments pour déterminer le nombre d'éléments contenus dans la collection. La valeur renvoyée peut alors être inexacte si une modification du contenu de la collection est réalisée de manière concurrente lors de ce parcours.

L'exécution de manière atomique des opérations de type bulk comme addAll(), removeAll(), retainAll(), containsAll(), equals() et toArray() n'est pas garantie. Il est par exemple possible que le parcours avec un Iterator ne voit qu'une partie des éléments qui sont en cours d'ajout par la méthode addAll() dont l'exécution se fait de manière concurrente.

14.6.7. Les files utilisables par les deux bouts (Deque)

Le terme anglais deque vient de la contraction de «double ended queue». Une Deque est une file dans laquelle il est possible de réaliser des opérations à ses deux extrémités.

14.6.7.1. L'interface java.util.Deque

L'interface java.util.Deque, ajoutée dans Java 6, définit des fonctionnalités permettant l'ajout et la suppression d'éléments par les deux bouts dans une file.

L'interface Deque hérite de l'interface Queue. Une collection de type Deque diffère en plusieurs points d'une collection de type Queue :

- contrairement à une Queue dans laquelle un élément ne peut être ajouté qu'à la fin et consulté ou retiré du début de la liste, une Deque permet d'ajouter ou de retirer un élément en début ou en fin de liste
- utilisé comme une Queue, une Deque est toujours de type FIFO
- il n'est pas possible de gérer une priorité dans les éléments d'une Deque

Méthode	Rôle
void addFirst(E e)	Insérer un nouvel élément au début de la collection
void addLast(E e)	Insérer un nouvel élément à la fin de la collection
void push(E e)	Insérer un nouvel élément au début de la collection
boolean removeFirstOccurrence(Object o)	Supprimer la première occurrence d'un objet dans la collection quelque soit la position de cet élément. Elle renvoie un booléen qui précise si l'opération a réussie
boolean removeLastOccurrence(Object o)	Supprimer la dernière occurrence d'un objet dans la collection quelque soit la position de cet élément. Elle renvoie un booléen qui précise si l'opération a réussie
Iterator<E> descendingIterator()	Renvoyer un itérateur qui permet le parcours de la fin vers le début de la collection
boolean offerFirst(E e)	Insérer un nouvel élément au début de la collection. Elle renvoie un booléen qui précise si l'opération a réussi
boolean offerLast(E e)	Insérer un nouvel élément à la fin de la collection. Elle renvoie un booléen qui précise si l'opération a réussi
E peekFirst()	Obtenir le premier élément de la liste sans le retirer de la collection
E peekLast()	Obtenir le dernier élément de la liste sans le retirer de la collection
E pollFirst()	Obtenir le premier élément de la liste et le retirer de la collection

E pollLast()	Obtenir le dernier élément de la liste et le retirer de la collection
E getFirst()	Obtenir le premier élément de la liste sans le retirer de la collection
E getLast()	Obtenir le dernier élément de la liste sans le retirer de la collection
E removeFirst()	Supprimer le premier élément de la collection
E removeLast()	Supprimer le dernier élément de la collection
E pop()	Obtenir le dernier élément de la liste et le retirer de la collection

L'interface définit des méthodes pour manipuler les éléments de la collection par les deux bouts : à chacune de ces actions, une méthode est proposée pour lever une exception ou renvoyer une valeur particulière en cas d'échec de son exécution.

	Début de la collection		Fin de la collection	
	Lever une exception	Renvoyer une valeur	Lever une exception	Renvoyer une valeur
Ajout	addFirst()	offerFirst()	addLast()	offerLast()
Retirer	removeFirst()	poolFirst()	removeLast()	poolLast()
Consulter	getFirst()	peekFirst()	getLast()	peekLast()

Il n'est pas possible d'accéder à un élément particulier de la collection hormis le premier et le dernier.

Il n'est pas recommandé d'insérer la valeur null dans une collection de type Deque essentiellement parce que la valeur null est retournée par plusieurs méthodes pour indiquer qu'elles ont échouées.

Il est préférable d'utiliser les méthodes offerFirst() et offerLast() pour ajouter un élément car elles permettent de gérer les cas ou l'ajout n'est pas possible.

Les méthodes removeFirstOccurence() et removeLastOccurence() agissent comme la méthode remove() mais elles précisent le sens de la recherche de l'élément à supprimer.

Une collection de type Deque peut être utilisé dans le mode FIFO (First In First Out) pour agir comme une file : dans ce cas, les éléments sont insérés à la fin de la collection et sont retirés du début de la collection. Plusieurs méthodes de l'interface Queue sont équivalentes aux méthodes de l'interface Deque :

Méthode de l'interface Queue	Méthode de l'interface Deque
add(e)	addLast(e)
offer(e)	offerLast(e)
remove()	removeFirst()
poll()	pollFirst()
element()	getFirst()
peek()	peekFirst()

Une collection de type Deque peut être utilisée dans le mode LIFO (Last In First Out) pour agir comme une pile : dans ce cas, les éléments sont insérés et retirés uniquement en tête de la collection. Il est d'ailleurs recommandé d'utiliser une instance de type Deque plutôt qu'une collection de type Stack pour implémenter une pile.

Trois méthodes facilitent l'utilisation de la collection de type Deque comme une pile :

- la méthode pop() est équivalente à la méthode removeFirst()
- la méthode push() est équivalente à la méthode addFirst()
- la méthode peek() est équivalente à la méthode peekFirst()

L'API Collections propose plusieurs implémentations de l'interface Deque : ArrayDeque, ConcurrentLinkedDeque, LinkedBlockingDeque, LinkedList.

14.6.7.2. La classe ArrayDeque

La classe ArrayDeque, ajoutée dans Java 6, est une implémentation de l'interface Deque sous la forme d'un tableau dont les dimensions peuvent évoluer.

La classe ArrayDeque est l'implémentation de choix pour une file de type FIFO.

Les performances lors de l'ajout d'un élément en début ou en fin de la collection restent constantes.

La classe ArrayDeque présente plusieurs caractéristiques :

- elle n'est pas thread-safe : elle ne propose aucune gestion des accès concurrents
- elle n'impose pas de restriction sur sa taille et peut grossir selon les besoins
- elle ne permet pas d'ajouter un élément null

La classe ArrayDeque propose plusieurs constructeurs :

Constructeur	Rôle
ArrayDeque()	Créer une collection vide avec une capacité initiale de 16 éléments
ArrayDeque(Collection<? extends E> c)	Créer une collection initialisée avec les éléments de la collection fournie en paramètre. L'ordre d'insertion utilisé est celui de l'Iterator de la collection
ArrayDeque(int numElements)	Créer une collection vide avec la capacité initiale fournie en paramètre

La classe ArrayDeque implémente toutes les méthodes optionnelles des interfaces Collection et Iterator.

La classe ArrayDeque peut remplacer la classe Stack pour une utilisation comme pile et la classe LinkedList pour une utilisation comme file.

Les Iterator obtenus à partir de la collection sont de type fail-fast : si une modification est effectuée dans le contenu de la collection durant le parcours avec l'Iterator alors une exception de type ConcurrentModificationException peut être levée sauf si cette modification est faite avec la méthode remove() de l'Iterator.

14.6.8. Les files d'attente

Il est parfois utile pour des questions de performances d'implémenter le modèle de conception producteur/consommateur (producer/consumer). Dans ce modèle, un ou plusieurs threads sont utilisés pour produire des objets et un ou plusieurs autres threads sont utilisés pour les consommer.

Avant Java 5, l'implémentation de ce modèle devait être faite manuellement, le plus souvent en utilisant les méthodes wait() et notify() sur un objet partagé en fonction de l'état de la file d'attente.

Le modèle producteur/consommateur définit trois acteurs :

- un producteur qui produit des éléments
- un consommateur qui consomme les éléments
- une file d'attente qui permet de stocker les éléments échangés

Ce modèle permet de réduire le couplage entre la production d'éléments et leur traitement. Une file d'attente est utilisée pour gérer les échanges des éléments entre le producteur et le consommateur.

Le modèle permet une coordination et une collaboration entre un ou plusieurs producteurs et un ou plusieurs consommateurs.

Ce modèle présente plusieurs avantages :

- le producteur et le consommateur sont développés séparément : ils ont juste besoin de connaître le type de l'élément échangé
- le producteur n'a pas besoin de connaitre le ou les consommateurs et inversement
- le producteur et le consommateur peuvent effectuer leurs traitements à différentes vitesses : la file d'attente se charge alors de faire le tampon
- la séparation des rôles permet de réduire le couplage en rendant le code de chacun plus facile à maintenir

La file d'attente gère les accès de manière concurrente et bloquante : si la file est pleine, alors l'opération d'ajout attend qu'un élément soit consommé et inversement si la file est vide, l'opération d'obtention d'un élément attend qu'un nouvel élément soit ajouté.

Ce modèle permet de gérer le flux des éléments et de s'adapter selon les besoins :

- en utilisant plusieurs consommateurs si le temps de traitement est plus long que le temps de création
- en utilisant plusieurs producteurs si le temps de création est plus long que le temps de traitement ou s'il existe plusieurs sources de création
- en utilisant plusieurs producteurs et plusieurs consommateurs

Il est important en cas de forte charge sur les échanges d'éléments de surveiller la capacité de la file d'attente pour éviter que celle-ci ne devienne un goulet d'étranglement. Comme les accès concurrents sont gérés par la file d'attente, il n'y a, par exemple, aucun inconvénient à ajouter un ou plusieurs autres consommateurs.

L'interface BlockingQueue facilite la mise en oeuvre de motifs de conception en proposant directement des méthodes bloquantes pour ajouter ou retirer des éléments de la file. L'API Collections propose plusieurs implémentations de l'interface BlockingQueue qui permettent de répondre à différents besoins.

14.6.8.1. L'interface java.util.concurrent.BlockingQueue

Une collection de type BlockingQueue stocke des éléments et gère la façon dont un élément est ajouté ou retiré éventuellement de manière bloquante lorsque la collection est pleine ou vide.

Les objets de type BlockingQueue permettent facilement d'échanger des objets entre des threads sans avoir à gérer explicitement la synchronisation des échanges par exemple en utilisant les méthodes wait() et notifyAll().

L'interface BlockingQueue, ajoutée dans Java 5, hérite des interfaces Collection, Queue et Iterable. Elle définit notamment des opérations :

- qui attendent que la collection soit non vide pour pouvoir obtenir un élément
- qui attendent que la collection possède l'espace nécessaire pour ajouter un nouvel élément

L'interface BlockingQueue propose pour certaines méthodes des paramètres pour indiquer un temps d'attente avant de retourner un échec plutôt que d'attendre indéfiniment lors de l'ajout ou l'obtention d'un élément dans la collection. Ceci peut permettre de gérer des situations de blocages.

La principale utilisation des implémentations de l'interface BlockingQueue est pour mettre en oeuvre le paradigme producer/consumer mais les méthodes de l'interface Collection sont aussi implémentées. Il est par exemple possible d'invoquer la méthode remove() même si son invocation devrait être rare.

L'interface BlockingQueue définit plusieurs méthodes :

Méthode	Rôle
boolean add(E o)	Ajouter un nouvel élément. Renvoie true en cas de succès sinon lève une exception de type IllegalStateException
int drainTo(Collection< ? super E> c)	Retirer les éléments de la collection pour les copier dans celle fournie en paramètre

int drainTo(Collection < ? super E> c, int maxElements)	Retirer au plus le nombre d'éléments de la collection pour les copier dans celle fournie en paramètre
boolean offer(E o)	Ajouter un nouvel élément si possible : le booléen indique le succès de l'opération
boolean offer(E o, long timeout, TimeUnit timeUnit)	Ajouter un nouvel élément. A la fin du timeout, elle renvoie false si l'élément n'a pu être ajouté à la collection
E poll(long timeout, TimeUnit timeUnit)	Obtenir et retirer le premier élément de la queue en attendant au plus le timeout fourni en paramètre. Elle renvoie false si aucun élément ne peut être retiré de la collection à la fin du timeout
void put(E)	Ajouter un élément, la méthode est bloquante tant que la collection est pleine
int remainingCapacity()	Renvoyer le nombre théorique de nouveaux éléments que peut accepter la collection avant de bloquer l'ajout. Elle renvoie Integer.MAX_VALUE si la capacité n'est pas limitée
E take()	Renvoyer un élément, la méthode est bloquante tant que la collection est vide

Les méthodes put() et take() sont bloquantes :

- la méthode take() attend qu'un nouvel élément soit inséré dans la collection si celle-ci est vide
- la méthode put() attend qu'un élément soit enlevé si la collection est pleine

Les collections de type BlockingQueue ne permettent pas l'ajout d'un élément null : dans ce cas, une exception de type NullPointerException est levée.

Une collection de type BlockingQueue peut avoir :

- une taille maximale déterminée (bounded) : lorsque la capacité maximale est initialisée et que le nombre maximum d'objets est ajouté, l'invocation de la méthode put() pour ajouter un élément supplémentaire est bloquante
- une taille maximale non définie (unbounded) : la capacité maximale n'est pas initialisée, sa valeur est MAX_VALUE.

Les fonctionnalités de l'interface BlockingQueue sont prévues pour être exécutées dans un contexte concurrent (multithread). Les implémentations de type BlockingQueue sont thread-safe : il n'y a aucun soucis pour avoir plusieurs producteurs et plusieurs consommateurs sur la file.

L'interface BlockingQueue hérite de l'interface Queue pour ajouter le support d'opérations bloquantes. Les opérations d'ajout et d'obtention d'un élément dans la collection proposent quatre comportements :

- lever une exception en cas d'échec : la méthode lève une exception si l'opération n'a pas été immédiatement exécutée
- renvoyer une valeur particulière en cas d'échec : la méthode renvoie une valeur particulière si l'opération n'a pas été immédiatement exécutée
- bloquante : l'invocation de la méthode est bloquée indéfiniment tant que l'opération n'a pas pu être réalisée
- bloquante avec un timeout : l'invocation de la méthode est bloquée tant que l'opération n'a pas pu être réalisée ou que le délai du timeout n'est pas écoulé. La méthode renvoie une valeur qui permet de savoir si l'opération a été exécutée ou non

	Lever une exception	Renvoie une valeur spéciale	Bloquante	Bloquante avec timeout
Ajouter	add(e)	offer(e)	put(e)	offer(e, time, unit)
Retirer	remove()	poll()	take()	poll(time, unit)
Consulter	element()	peek()	-	-

Attention : l'aspect bloquant de certaines fonctionnalités des implémentations de l'interface BlockingQueue peut induire des problèmes de contention et donc limiter la montée en charge.

Il n'est pas possible d'ajouter un élément null dans une collection de type BlockingQueue : l'invocation des méthodes add(), offer() et put() avec null en paramètre lève une exception de type NullPointerException.

Les implémentations de type BlockingQueue ne proposent en standard aucun mécanisme qui permettrait d'empêcher l'ajout de nouveaux éléments. Ainsi chaque thread qui consomme des éléments de la queue doit proposer son propre mécanisme pour s'arrêter. Le ou les threads qui ajoutent des éléments dans la collection doivent mettre en oeuvre ce mécanisme pour permettre l'arrêt des threads de consommation. Ce mécanisme peut par exemple mettre en oeuvre un élément particulier (poison object) qui, une fois consommé et identifié par le thread, lui permettra de se terminer proprement.

14.6.8.2. La classe java.util.concurrent.ArrayBlockingQueue

La classe ArrayBlockingQueue implémente l'interface BlockingQueue en utilisant un tableau d'objets.

La classe ArrayBlockingQueue utilise le mode de fonctionnement FIFO (First In First Out) : le premier élément de la file est le plus ancien et le dernier élément est le plus récent.

Son implémentation utilise un tableau ce qui lui impose une taille maximale. La capacité maximale d'un ArrayBlockingQueue doit donc être obligatoirement fixée en paramètre de l'invocation du constructeur et ne peut plus être changée ultérieurement.

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.Date;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class ArrayBlockingQueueMain {
  private static final int NB_ELEMENTS = 5;

  public static void main(final String[] args) {
    final BlockingQueue<String> queue = new ArrayBlockingQueue<String>(2);
    Thread producteur = new Thread(new Runnable() {
      @Override
      public void run() {
        for (int i = 1; i <= NB_ELEMENTS; i++) {
          String element = "Element " + i;
          try {
            System.out.println(new Date() + " insertion element " +
              element);
            queue.put(element);
          } catch (InterruptedException ie) {
            ie.printStackTrace();
          }
        }
      }
    }, "Producteur");

    producteur.start();
    Thread consommateur = new Thread(new Runnable() {
      @Override
      public void run() {
        for (int i = 1; i <= NB_ELEMENTS; i++) {
          try {
            String element = queue.take();
            System.out.println(new Date() + " obtention element : " +
              element);
            Thread.sleep(2000);
          } catch (InterruptedException ie) {
            ie.printStackTrace();
          }
        }
      }
    }, "Consommateur");

    consommateur.start();
  }
}
```

Résultat :

```
Mon Mar 11 18:27:54 CET 2013 insertion element Element 1
Mon Mar 11 18:27:54 CET 2013 insertion element Element 2
Mon Mar 11 18:27:54 CET 2013 insertion element Element 3
Mon Mar 11 18:27:54 CET 2013 obtention element : Element 1
Mon Mar 11 18:27:54 CET 2013 insertion element Element 4
Mon Mar 11 18:27:56 CET 2013 obtention element : Element 2
Mon Mar 11 18:27:56 CET 2013 insertion element Element 5
Mon Mar 11 18:27:58 CET 2013 obtention element : Element 3
Mon Mar 11 18:28:00 CET 2013 obtention element : Element 4
Mon Mar 11 18:28:02 CET 2013 obtention element : Element 5
```

14.6.8.3. La classe java.util.concurrent.LinkedBlockingQueue

La classe LinkedBlockingQueue implémente l'interface BlockingQueue en utilisant une LinkedList pour stocker les éléments en interne.

Il est possible de préciser une taille maximale pour le nombre d'éléments que peut contenir la collection : par défaut, la capacité maximale d'une LinkedBlockingQueue est fixée à la valeur Integer.MAX_VALUE sauf si une valeur différente est fournie en paramètre du constructeur.

Dans une LinkedBlockingQueue, les éléments sont stockés selon le mode FIFO (First In, First Out) : le premier élément de la liste est celui qui est dans la file depuis le plus longtemps. Le dernier élément de la liste est celui qui est arrivé le plus récemment dans la file.

L'avantage d'une LinkedBlockingQueue par rapport à une ArrayBlockingQueue est de ne pas être obligé de limiter la taille de la collection : les producteurs ne sont pas bloqués en attendant que la file se vide. Ceci peut cependant engendrer d'autres difficultés comme un manque de mémoire si la collection se remplit beaucoup plus vite qu'elle ne se vide.

Une LinkedBlockingQueue offre généralement de meilleurs performances qu'une ArrayBlockingQueue notamment si la file doit contenir de nombreux objets mais elle requiert plus d'objets en mémoire.

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

public class LinkedBlockingQueueMain {
  public static void main(final String[] args) {
    final BlockingQueue<String> queue = new LinkedBlockingQueue<String>();
    Thread producteur = new Thread(new Runnable() {
      @Override
      public void run() {
        for (int i = 1; i <= 5; i++) {
          String element = "Element " + i;
          try {
            System.out.println("insertion element " + element);
            queue.put(element);
          } catch (InterruptedException ie) {
            ie.printStackTrace();
          }
        }
      }
    }, "Producteur");
    producteur.start();

    Thread consommateur = new Thread(new Runnable() {
      @Override
      public void run() {
        for (int i = 1; i <= 5; i++) {
          try {
```

```
      String element = queue.take();
      System.out.println("element : " + element);
      Thread.sleep(2000);
     } catch (InterruptedException ie) {
      ie.printStackTrace();
     }
    }
   }
  }, "Consommateur");

  consommateur.start();
 }
}
```

Les éléments sont consommés dans l'ordre FIFO.

Résultat :

```
insertion element Element 1
insertion element Element 2
insertion element Element 3
insertion element Element 4
insertion element Element 5
element : Element 1
element : Element 2
element : Element 3
element : Element 4
element : Element 5
```

14.6.8.4. La classe java.util.concurent.PriorityBlockingQueue

La classe PriorityBlockingQueue est une collection de type Queue qui renvoie en premier les éléments les plus prioritaires.

La classe PriorityBlockingQueue implémente l'interface BlockingQueue, Queue, Collection et Iterable.

Elle implémente toutes les méthodes optionnelles des interfaces Collection et Iterator.

Le parcours des éléments en utilisant la méthode iterator() se fait dans un ordre quelconque.

Par défaut, la capacité maximale d'un PriorityBlockingQueue est fixée à la valeur Integer.MAX_VALUE sauf si une valeur différente est fournie en paramètre du constructeur

Il n'est pas possible d'ajouter un élément null dans la collection.

La classe PriorityBlockingQueue stocke ses éléments de manière ordonnée en utilisant une instance de Comparator fournie en paramètre du constructeur ou à défaut en utilisant la méthode de l'interface Comparable implémentée par les éléments de la collection.

Si aucune instance de Comparator n'est fournie en paramètre du constructeur, alors les éléments ajoutés dans la collection doivent implémenter l'interface Comparable sinon une exception de type ClassCastException est levée.

La classe PriorityBlockingQueue n'impose rien concernant des éléments qui possèdent la même priorité : c'est l'implémentation de la méthode compareTo() qui peut gérer ce cas selon les besoins.

Exemple :

```
package fr.jmdoudoux.dej.collections;

public class MaTache implements Comparable<MaTache> {
 private final int    priorite;
 private final String description;

 public MaTache(final int priorite, final String description) {
  this.priorite = priorite;
```

```
    this.description = description;
  }

  @Override
  public int compareTo(final MaTache tache) {
    Integer prioriteCourante = getPiorite();
    Integer prioriteAutre = tache.getPiorite();
    if (prioriteCourante != prioriteAutre) {
      return prioriteCourante.compareTo(prioriteAutre);
    } else {
      return getDescription().compareTo(tache.getDescription());
    }
  }

  @Override
  public boolean equals(final Object obj) {
    if (this == obj) {
      return true;
    }
    if (obj == null) {
      return false;
    }
    if (getClass() != obj.getClass()) {
      return false;
    }
    MaTache other = (MaTache) obj;
    if (description == null) {
      if (other.description != null) {
        return false;
      }
    } else if (!description.equals(other.description)) {
      return false;
    }
    if (priorite != other.priorite) {
      return false;
    }
    return true;
  }

  public String getDescription() {
    return description;
  }

  public int getPiorite() {
    return priorite;
  }

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result
        + ((description == null) ? 0 : description.hashCode());
    result = prime * result + priorite;
    return result;
  }

  @Override public String toString() {
    return "MaTache [priorite=" + priorite + ", description=" + description
        + "]";
  }
}
```

La classe PriorityBlockingQueue agit comme la classe PriorityQueue avec une gestion des accès concurrents.

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.concurrent.BlockingQueue;
import java.util.concurrent.PriorityBlockingQueue;
```

```
public class PriorityBlockingQueueMain {
  public static void main(final String[] args) {
    final BlockingQueue<MaTache> queue = new PriorityBlockingQueue<MaTache>();
    Thread producteur = new Thread(new Runnable() {
      @Override
      public void run() {
        for (int i = 1; i <= 5; i++) {
          MaTache element = new MaTache(6 - i, "Tache " + i);
          try {
            System.out.println("insertion element " + element);
            queue.put(element);
          } catch (InterruptedException ie) {
            ie.printStackTrace();
          }
        }
        System.out.println("taille de la queue=" + queue.size());
      }
    }, "Producteur");

    producteur.start();
    try {
      // on attend que tous les éléments soient insérés
      // uniquement pour vérifier l'ordre d'obtention
      producteur.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    MaTache elt = queue.peek();
    System.out.println("peek=" + elt);

    Thread consommateur = new Thread(new Runnable() {
      @Override
      public void run() {
        for (int i = 1; i <= 5; i++) {
          try {
            MaTache element = queue.take();
            System.out.println("element : " + element);
          } catch (InterruptedException ie) {
            ie.printStackTrace();
          }
        }
      }
    }, "Consommateur");
    consommateur.start();
  }
}
```

Résultat :

```
insertion element MaTache [priorite=5,
description=Tache 1]
insertion element MaTache [priorite=4, description=Tache 2]
insertion element MaTache [priorite=3, description=Tache 3]
insertion element MaTache [priorite=2, description=Tache 4]
insertion element MaTache [priorite=1, description=Tache 5]
taille de la queue=5 peek=MaTache [priorite=1, description=Tache 5]
element : MaTache [priorite=1, description=Tache 5]
element : MaTache [priorite=2, description=Tache 4]
element : MaTache [priorite=3, description=Tache 3]
element : MaTache [priorite=4, description=Tache 2]
element : MaTache [priorite=5, description=Tache 1]
```

Si l'on supprime le temps d'attente de la fin du thread de production, les tâches obtenues tiennent toujours compte de l'ordre des éléments présents dans la collection au moment où l'on souhaite obtenir un élément.

Résultat :

```
peek=null
insertion element MaTache [priorite=5, description=Tache 1]
insertion element MaTache [priorite=4, description=Tache 2]
element : MaTache [priorite=5, description=Tache 1]
element : MaTache [priorite=4, description=Tache 2]
insertion element MaTache [priorite=3, description=Tache 3]
insertion element MaTache [priorite=2, description=Tache 4]
insertion element MaTache [priorite=1, description=Tache 5]
element : MaTache [priorite=3, description=Tache 3]
taille de la queue=1 element : MaTache [priorite=1, description=Tache 5]
element : MaTache [priorite=2, description=Tache 4]
```

14.6.8.5. La classe java.util.concurrent.DelayQueue

La classe DelayQueue, ajoutée à Java 1.5, permet de conserver un élément jusqu'à l'expiration d'un délai.

La classe DelayQueue implémente l'interface BlockingQueue

Les éléments insérés dans une DelayQueue doivent implémenter l'interface java.util.concurrent.Delayed. Cette interface ne définit qu'une seule méthode :

Méthode	Rôle
long getDelay(TimeUnit timeUnit)	Renvoyer le délai à attendre avant que l'élément ne soit retirable de la collection

Si la valeur retournée par la méthode getDelay() est négative ou nulle alors le délai est considéré comme expiré et l'élément est immédiatement retirable. Le paramètre de type TimeUnit permet de préciser l'unité de temps dans laquelle la valeur doit être retournée.

L'interface Delayed hérite de l'interface Comparable<Delayed>.

Chaque implémentation doit donc redéfinir la méthode compareTo() qui est utilisée par le DelayQueue pour déterminer l'ordre de renvoie d'un élément.

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.Date;
import java.util.concurrent.Delayed;
import java.util.concurrent.TimeUnit;

public class MonElementDelayed implements Delayed {
  private final String donnees;
  private final long   momentDeLiberation;
  public MonElementDelayed(final String donnees, final long delai) {
    this.donnees = donnees;
    this.momentDeLiberation = System.currentTimeMillis() + delai;
  }

  @Override
  public int compareTo(final Delayed o) {
    int resultat = -1;
    if (o instanceof MonElementDelayed) {
      MonElementDelayed med = (MonElementDelayed) o;
      if (this.momentDeLiberation < med.momentDeLiberation) {
        resultat = -1;
      } else {
        if (this.momentDeLiberation > med.momentDeLiberation) {
          resultat = 1;
        } else {
          resultat = 0;
        }
      }
    }
    return resultat;
  }

  @Override
  public boolean equals(final Object obj) {
    if (this == obj) {
      return true;
    }
    if (obj == null) {
      return false;
    }
    if (getClass() != obj.getClass()) {
      return false;
    }
    MonElementDelayed other = (MonElementDelayed) obj;
    if (donnees == null) {
      if (other.donnees != null) {
        return false;
      }
    } else if (!donnees.equals(other.donnees)) {
      return false;
    }
    if (momentDeLiberation != other.momentDeLiberation) {
      return false;
    }
    return true;
  }

  @Override
  public long getDelay(final TimeUnit unit) {
    long diff = momentDeLiberation - System.currentTimeMillis();
    return unit.convert(diff, TimeUnit.MILLISECONDS);
  }

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((donnees == null) ? 0 : donnees.hashCode());
    result = prime * result
        + (int) (momentDeLiberation ^ (momentDeLiberation >>> 32));
    return result;
  }

  @Override
  public String toString() {
    return "donnees='" + donnees + "', momentDeLiberation="
        + new Date(momentDeLiberation);
  }
}
```

Attention : l'implémentation de la méthode compareTo() doit tenir compte du délai restant lié à l'objet. Cette implémentation nest donc pas cohérente avec la méthode equals() ce qui exclut l'utilisation des objets de type Delayed dans certaines collections notamment celles de type SortedXXX.

Les implémentations de plusieurs méthodes de la classe DelayQueue assurent les comportements spécifiques de cette collection :

Méthode	Rôle
int drainTo(Collection<? super E> c)	Retirer tous les éléments dont le délai est expiré et les ajouter dans la collection fournie en paramètre
int drainTo(Collection<? super E> c, int maxElements)	Retirer au plus le nombre d'éléments dont le délai est expiré et les ajouter dans la collection fournie en paramètre
Iterator<E> iterator()	Renvoyer un Iterator permettant le parcours sur tous les éléments expirés ou non
boolean offer(E e, long timeout, TimeUnit unit)	Ajouter un élément dans la collection. Les paramètres timeout et unit sont ignorés
E peek()	Obtenir le prochain élément que son délai soit expiré ou non sans le retirer de la collection.
E poll()	

	Renvoyer le premier élément dont le délai est expiré et le supprimer de la collection. Elle renvoie toujours null tant que le délai d'attente d'au moins un élément n'est pas atteint.
E poll(long timeout, TimeUnit unit)	Identique à la méthode poll() mais attend au besoin le délai fourni en paramètre
int remainingCapacity()	Renvoyer toujours la valeur Integer.MAX_VALUE car la taille de la collection ne peut pas être limitée explicitement
boolean remove(Object o)	Retirer un élément de la collection que son délai soit expiré ou non
int size()	Renvoyer le nombre total d'éléments dans la collection qu'ils soient expirés ou non
E take()	Renvoyer le premier élément dont le délai est expiré et le supprimer de la collection. Son invocation est bloquante tant que le délai d'aucun élément n'est atteint
Object[] toArray()	Renvoyer un tableau de tous les éléments de la collection

La taille d'un DelayQueue ne peut pas être limitée : l'ajout d'un nouvel élément en utilisant la méthode put() n'est donc jamais bloquante. Par contre, l'invocation de la méthode take() est bloquante tant que le délai d'attente d'un élément n'est pas atteint (dans ce cas, la valeur retournée par la méthode getDelay() est inférieure ou égale à zéro).

Si plusieurs éléments ont leur délai d'attente atteint lors de l'invocation de la méthode take() alors celle-ci renvoie l'élément dont le délai d'attente est le plus dépassé.

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.Date;
import java.util.concurrent.DelayQueue;

public class DelayQueueMain {
  public static void main(final String[] args) {
    DelayQueue<MonElementDelayed> queue = new
      DelayQueue<MonElementDelayed>();
    for (int i = 1; i <= 5; i++) {
      MonElementDelayed element = new MonElementDelayed("Message" + i, 2000 * i);
      System.out.println("insertion element " + element);
      queue.put(element);
    }

    System.out.println("taille de la queue=" + queue.size());
    try {
      for (int i = 1; i < 5; i++) {
        MonElementDelayed element = queue.take();
        System.out.println("element obtenu le" + new Date());
        System.out.println("  " + element);
      }
    } catch (InterruptedException ie) {
      ie.printStackTrace();
    }
  }
}
```

Résultat :

```
insertion element donnees='Message1',
momentDeLiberation=Fri Mar 01 22:34:04 CET 2013
insertion element donnees='Message2', momentDeLiberation=Fri Mar 01 22:34:06 CET 2013
insertion element donnees='Message3', momentDeLiberation=Fri Mar 01 22:34:08 CET 2013
insertion element donnees='Message4', momentDeLiberation=Fri Mar 01 22:34:10 CET 2013
insertion element donnees='Message5', momentDeLiberation=Fri Mar 01 22:34:12 CET 2013
taille de la queue=5 element obtenu leFri Mar 01 22:34:04 CET 2013
  donnees='Message1', momentDeLiberation=Fri Mar 01 22:34:04 CET 2013
element obtenu leFri Mar 01 22:34:06 CET 2013
  donnees='Message2', momentDeLiberation=Fri Mar 01 22:34:06 CET 2013
```

```
element obtenu leFri Mar 01 22:34:08 CET 2013
  donnees='Message3', momentDeLiberation=Fri Mar 01 22:34:08 CET 2013
element obtenu leFri Mar 01 22:34:10 CET 2013
  donnees='Message4', momentDeLiberation=Fri Mar 01 22:34:10 CET 2013
```

14.6.8.6. La classe java.util.concurrrent.SynchronousQueue

La classe SynchronousQueue, ajoutée à Java 1.5 implémente l'interface BlockingQueue pour proposer un moyen facile d'échanger un élément entre deux threads et de synchroniser ces échanges.

Elle ne possède pas de capacité de stockage : elle ne peut contenir qu'un seul élément au plus. L'insertion d'un nouvel élément en invoquant la méthode put() est bloquée jusqu'à ce que l'élément inséré soit retiré. De façon similaire, si la collection est vide alors l'invocation de la méthode take() reste bloquée jusqu'à ce qu'un nouvel élément soit ajouté dans la collection.

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.concurrent.BlockingQueue;

public class MonProducer implements Runnable {
  private final BlockingQueue<String> queue;

  public MonProducer(final BlockingQueue<String> queue) {
    this.queue = queue;
  }

  @Override
    public void run() {
      try {
      for (int i = 1; i < 6; i++) {
        queue.put("Message" + i);
        System.out.println("Producer envoie Message" + i);
      }
      queue.put("Stop");
      } catch (InterruptedException ie) {
      ie.printStackTrace();
    }
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.concurrent.BlockingQueue;

public class MonConsumer implements Runnable {

  private final BlockingQueue<String> queue;

  public MonConsumer(final BlockingQueue<String> queue) {
    this.queue = queue;
  }

  @Override
    public void run() {
      try {
        String msg = null;
        boolean stop = false;
        while (!stop) {
          msg = queue.take();
          if (msg.equals("Stop")) {
            stop = true;
          } else {
            Thread.sleep(2000);
            System.out.println("Consumer traite " + msg);
          }
        }
     } catch (InterruptedException ie) {
       ie.printStackTrace();
     }
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.collections;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.SynchronousQueue;

public class SynchronousQueueMain {

  public static void main(final String[] args) {
    BlockingQueue<String> queue = new SynchronousQueue<String>();
      (new Thread(new MonProducer(queue))).start();

    try {
      Thread.sleep(1000);
    } catch (InterruptedException ie) {
      ie.printStackTrace();
    }

    (new Thread(new MonConsumer(queue))).start();

  }
}
```

Résultat :

```
Producer envoie Message1
Consumer traite Message1
Producer envoie Message2
Consumer traite Message2
Producer envoie Message3
Consumer traite Message3
Producer envoie Message4
Consumer traite Message4
Producer envoie Message5
Consumer traite Message5
```

Remarque : les temps d'attente dans l'exemple sont uniquement présents pour faciliter la compréhension du fonctionnement de la classe SynchronousQueue.

14.6.8.7. L'interface java.util.concurrent.BlockingDeque

L'interface BlockingDeque, ajoutée à Java 6.0, implémente les interfaces Deque et BlockingQueue avec des opérations bloquantes :

- qui attendent que la collection soit non vide pour pouvoir obtenir un élément
- qui attendent que la collection possède l'espace nécessaire pour ajouter un nouvel élément

L'utilisation d'un BlockingDeque est particulièrement intéressante :

- si un thread peut à la fois produire et consommer des éléments dans une file d'attente.
- si plusieurs threads doivent pouvoir ajouter ou retirer des éléments au début ou à la fin de la collection

Le comportement des méthodes de l'interface BlockingQueue d'un BlockingDeque est de respecter le mode de fonctionnement prévu par cette interface

Méthode de l'interface BlockingQueue	Méthode invoquée

add()	addLast()
offer()	offerLast()
put()	putLast()
remove()	removeFirst()
poll()	pollFirst()
take()	takeFirst()
element()	getFirst()
peek()	peekFirst()

L'interface BlockingDeque propose quatre comportements différents pour les opérations d'ajout, de retrait et de consultation d'un élément dans la collection si celles-ci ne peuvent pas être réalisées immédiatement :

- lever une exception si l'opération n'est pas réalisable immédiatement
- renvoie une valeur particulière si l'opération n'est pas réalisable immédiatement
- l'invocation de la méthode est bloquée jusqu'à ce que l'opération soit réalisée
- l'invocation de la méthode est bloquée jusqu'à ce que l'opération soit réalisée ou qu'un timeout soit atteint. La méthode renvoie une valeur qui précise si l'opération a été exécutée

	Lève une exception	Renvoie une valeur	Bloquant	Time out

Ajout	addFirst()
	addLast()
		offerFirst()
		offerLast()
			putFirst()
			putLast()
				offerFirst(timeout)
				offerLast(timeout)

Retrait	removeFirst()
	removeLast()
		pollFirst()
		pollLast()
			takeFirst()
			takeLast()
				pollFirst(timeout)
				pollLast(timeout)

Consultation	getFirst()
	getLast()
		peekFirst()
		peekLast()		

L'API Collections propose une seule implémentation de l'interface BlockingDeque : la classe java.util.concurrent.LinkedBlockingDeque.

14.6.8.8. La classe LinkedBlockingDeque

La classe LinkedBlockingDeque, ajoutée à Java 6.0, implémente l'interface BlockingDeque.

La classe LinkedBlockingDeque utilise en interne une liste chaînée de type LinkedList pour stocker ses éléments.

Un des constructeurs de la classe LinkedBlockingDeque attend en paramètre une valeur qui précise la capacité maximale de la collection.

Exemple :

```
package fr.jmdoudoux.dej.collections;

import java.util.Date;
import java.util.concurrent.BlockingDeque;
import java.util.concurrent.LinkedBlockingDeque;

public class TestLinkedBlockingDequeMain {
  private static final int NB_ELEMENTS = 5;
  public static void main(final String[] args) {
    // final BlockingDeque<String> queue = new LinkedBlockingDeque<String>();
    final BlockingDeque<String> queue = new LinkedBlockingDeque<String>(2);
    Thread producteur = new Thread(new Runnable() {
      @Override
      public void run() {
        for (int i = 1; i <= NB_ELEMENTS; i++) {
          String element = "Element " + i;
          try {
            System.out.println(new Date() + " insertion element " + element);
            queue.putFirst(element);
          } catch (InterruptedException ie) {
            ie.printStackTrace();
          }
        }
      }
    }, "Producteur");

    producteur.start();
    Thread consommateur = new Thread(new Runnable() {
      @Override
      public void run() {
        for (int i = 1; i <= NB_ELEMENTS; i++) {
          try {
            String element = queue.takeLast();
            System.out.println(new Date() + " obtention element : " + element);
            Thread.sleep(2000);
          } catch (InterruptedException ie) {
            ie.printStackTrace();
          }
        }
      }
    }, "Consommateur");

    consommateur.start();
  }
}
```

Résultat :

```
Fri Mar 15 21:06:34 CET 2013 insertion element
Element 1 Fri Mar 15 21:06:34 CET 2013 insertion element
Element 2 Fri Mar 15 21:06:34 CET 2013 insertion element
Element 3 Fri Mar 15 21:06:34 CET 2013 obtention element :
Element 1 Fri Mar 15 21:06:34 CET 2013 insertion element
Element 4 Fri Mar 15 21:06:36 CET 2013 insertion element
Element 5 Fri Mar 15 21:06:36 CET 2013 obtention element :
Element 2 Fri Mar 15 21:06:38 CET 2013 obtention element :
Element 3 Fri Mar 15 21:06:40 CET 2013 obtention element :
Element 4 Fri Mar 15 21:06:42 CET 2013 obtention element :
Element 5
```

Les performances d'insertion et de suppression d'un élément sont similaires à celles d'une collection de type LinkedBlockingQueue puisqu'elles utilisent toutes les deux une LinkedList pour la gestion de leurs éléments.

14.6.9. Le choix d'une implémentation de type Queue

Plusieurs implémentations de type Queue et BlockingQueue sont particulièrement utiles pour gérer des échanges d'objets entre threads par exemple en mettant en oeuvre le motif de conception producteur/consommateur.

Plusieurs implémentations de type Queue ne peuvent être utilisées que pour des cas bien particuliers :

- si la collection doit gérer ses éléments par ordre de priorité et que les accès concurrents n'ont pas besoin d'être gérés alors il faut utiliser la classe PriorityQueue
- si la collection doit gérer ses éléments par ordre de priorité et que les accès concurrents ont besoin d'être gérés alors il faut utiliser la classe PriorityBlockingQueue
- si la collection doit gérer un élément avant qu'un élément ne soit retirable alors il faut utiliser la classe DelayQueue

- la classe SynchronousQueue est particulièrement utile pour échanger un objet entre deux threads (elle n'a aucune capacité de stockage)

Le choix d'une implémentation d'une collection de type Queue pour un usage général doit se faire selon les fonctionnalités qu'elle doit supporter :

- la gestion des accès concurrents ou non
- les opérations sont bloquantes ou non
- la taille de la collection est bornée ou non

Si la collection de type Queue n'a pas besoin de gérer d'accès concurrents alors il faut utiliser la classe LinkedList.

Si les accès concurrents doivent être gérés alors il faut utiliser la classe ConcurrentLinkedList.

Si la taille de la collection de type Queue doit être bornée, alors il faut utiliser la classe ArrayBlockingQueue.

	Critère	Taille limitée	Taille non limitée
Bloquante	Aucun	ArrayBlockingQueue	LinkedBlockingQueue
Priorité		PriorityBlockingQueue	
Delai		DelayQueue	
Transfert	SynchronousQueue LinkedTransferQueue		
Deque		LinkedBlockingDeque	
Non-bloquante	Thread-safe		ConcurrentLinkedQueue
Non thread-safe		LinkedList	
Non thread-safe, priorité		PriorityQueue	
Non thread-safe, Deque	ArrayDeque		

14.7. Le tri des collections

L'ordre de tri est défini grâce à deux interfaces :

- Comparable
- Comparator

14.7.1. L'interface Comparable

Tous les objets qui doivent définir un ordre naturel utilisé par le tri d'une collection doivent implémenter cette interface.

Cette interface ne définit qu'une seule méthode : int compareTo(Object).

Cette méthode doit renvoyer :

- une valeur entière négative si l'objet courant est inférieur à l'objet fourni
- une valeur entière positive si l'objet courant est supérieur à l'objet fourni
- une valeur nulle si l'objet courant est égal à l'objet fourni

Les classes wrappers, String et Date implémentent cette interface.

14.7.2. L'interface Comparator

Cette interface représente un ordre de tri quelconque. Elle est utile pour permettre le tri d'objets qui n'implémentent pas l'interface Comparable ou pour définir un ordre de tri différent de celui défini avec Comparable (l'interface Comparable représente un ordre naturel : il ne peut y en avoir qu'un)

Cette interface ne définit qu'une seule méthode : int compare(Object, Object).

Cette méthode compare les deux objets fournis en paramètres et renvoie :

- une valeur entière négative si le premier objet est inférieur au second
- une valeur entière positive si le premier objet est supérieur au second
- une valeur nulle si les deux objets sont égaux

14.8. Les algorithmes

La classe Collections propose plusieurs méthodes statiques pour effectuer des opérations sur des collections. Ces traitements sont polymorphiques car ils demandent en paramètre un objet qui implémente une interface et retournent une collection.

Méthode	Rôle
void copy(List, List)	Copier tous les éléments de la seconde liste dans la première
Enumeration enumeration(Collection)	Renvoyer un objet Enumeration pour parcourir la collection
Object max(Collection)	Renvoyer le plus grand élément de la collection selon l'ordre naturel des éléments
Object max(Collection, Comparator)	Renvoyer le plus grand élément de la collection selon l'ordre précisé par l'objet Comparator
Object min(Collection)	Renvoyer le plus petit élément de la collection selon l'ordre naturel des éléments
Object min(Collection, Comparator)	Renvoyer le plus petit élément de la collection selon l'ordre précisé par l'objet Comparator
void reverse(List)	Inverser l'ordre de la liste fournie en paramètre
void shuffle(List)	Réordonner tous les éléments de la liste de façon aléatoire
void sort(List)	Trier la liste dans un ordre ascendant selon l'ordre naturel des éléments
void sort(List, Comparator)	Trier la liste dans un ordre ascendant selon l'ordre précisé par l'objet Comparator

Si la méthode sort(List) est utilisée, il faut obligatoirement que les éléments inclus dans la liste implémentent tous l'interface Comparable sinon une exception de type ClassCastException est levée.

La classe Collections propose aussi plusieurs méthodes pour obtenir une version multithread ou non modifiable des principales interfaces des collections : Collection, List, Map, Set, SortedMap, SortedSet

- XXX synchronizedXXX(XXX) pour obtenir une version multithread des objets implémentant l'interface XXX
- XXX unmodifiableXXX(XXX) pour obtenir une version non modifiable des objets implémentant l'interface XXX

Exemple (code Java 1.2) :

```
import java.util.*;

public class TestUnmodifiable{
    public static void main(String args[]) {
      List list = new LinkedList();
      list.add("1");
      list.add("2");
      list = Collections.unmodifiableList(list);

      list.add("3");
    }
}
```

Résultat :

```
C:\>java TestUnmodifiable
Exception in thread "main" java.lang.UnsupportedOperationException
        at java.util.Collections$UnmodifiableCollection.add(Unknown Source)
        at TestUnmodifiable.main(TestUnmodifiable.java:13)
```

L'utilisation d'une méthode synchronizedXXX() renvoie une instance de l'objet qui supporte la synchronisation pour les opérations d'ajout et de suppression d'éléments. Pour le parcours de la collection avec un objet Iterator, il est nécessaire de synchroniser le bloc de code utilisé pour le parcours. Il est important d'inclure aussi dans ce bloc l'appel à la méthode pour obtenir l'objet de type Iterator utilisé pour le parcours.

Exemple (code Java 1.2) :

```
import java.util.*;

public class TestSynchronized{
    public static void main(String args[]) {
      List maList = new LinkedList();

      maList.add("1");
      maList.add("2");
      maList.add("3");
      maList = Collections.synchronizedList(maList);

      synchronized(maList) {
          Iterator i = maList.iterator();
          while (i.hasNext())
               System.out.println(i.next());
          }
     }
}
```

14.9. Les exceptions du framework

L'exception de type UnsupportedOperationException est levée lorsqu'une opération optionnelle n'est pas supportée par l'objet qui gère la collection.

L'exception ConcurrentModificationException est levée lors du parcours d'une collection avec un objet Iterator et que cette collection subit une modification structurelle.

15. Les flux d'entrée/sortie

Niveau : 🟩🟩🟨⬜⬜⬜ Intermédiaire

Un programme a souvent besoin d'échanger des informations, que ce soit pour recevoir des données d'une source ou pour envoyer des données vers un destinataire.

La source et la destination de ces échanges peuvent être de natures multiples : un fichier, une socket réseau, un autre programme, etc ...

De la même façon, la nature des données échangées peut être diverse : du texte, des images, du son, etc ...

Ce chapitre contient plusieurs sections :

♦ La présentation des flux
♦ Les classes de gestion des flux
♦ Les flux de caractères
♦ Les flux d'octets
♦ La classe File
♦ Les fichiers à accès direct
♦ La classe java.io.Console

15.1. La présentation des flux

Les flux (streams en anglais) permettent d'encapsuler ces processus d'envoi et de réception de données. Les flux traitent toujours les données de façon séquentielle.

En Java, les flux peuvent être divisés en plusieurs catégories :

- les flux d'entrée (input stream) et les flux de sortie (output stream)
- les flux de traitement de caractères et les flux de traitement d'octets

Java définit des flux pour lire ou écrire des données mais aussi des classes qui permettent de faire des traitements sur les données du flux. Ces classes doivent être associées à un flux de lecture ou d'écriture et sont considérées comme des filtres. Par exemple, il existe des filtres qui permettent de mettre les données traitées dans un tampon (buffer) pour les traiter par lots.

Toutes ces classes sont regroupées dans le package java.io.

15.2. Les classes de gestion des flux

Ce qui déroute dans l'utilisation de ces classes, c'est leur nombre et la difficulté de choisir celle qui convient le mieux en fonction des besoins. Pour faciliter ce choix, il faut comprendre la dénomination des classes : cela permet de sélectionner la ou les classes adaptées aux traitements à réaliser.

Le nom des classes se compose d'un préfixe et d'un suffixe. Il y a quatre suffixes possibles en fonction du type de flux (flux d'octets ou de caractères) et du sens du flux (entrée ou sortie).

	Flux d'octets	Flux de caractères
Flux d'entrée	InputStream	Reader
Flux de sortie	OutputStream	Writer

Il existe donc quatre hiérarchies de classes qui encapsulent des types de flux particuliers. Ces classes peuvent être séparées en deux séries de deux catégories différentes : les classes de lecture et d'écriture et les classes permettant la lecture de caractères ou d'octets.

- les sous-classes de Reader sont des types de flux en lecture sur des ensembles de caractères
- les sous-classes de Writer sont des types de flux en écriture sur des ensembles de caractères
- les sous-classes de InputStream sont des types de flux en lecture sur des ensembles d'octets
- les sous-classes de OutputStream sont des types de flux en écriture sur des ensembles d'octets

Pour le préfixe, il faut distinguer les flux et les filtres. Pour les flux, le préfixe contient la source ou la destination selon le sens du flux.

Préfixe du flux	source ou destination du flux
ByteArray	tableau d'octets en mémoire
CharArray	tableau de caractères en mémoire
File	fichier
Object	objet
Pipe	pipeline entre deux threads
String	chaîne de caractères

Pour les filtres, le préfixe contient le type de traitement qu'il effectue. Les filtres n'existent pas obligatoirement pour des flux en entrée et en sortie.

Type de traitement	Préfixe de la classe	En entrée	En sortie
Mise en tampon	Buffered	Oui	Oui
Concaténation de flux	Sequence	Oui pour flux d'octets	Non
Conversion de données	Data	Oui pour flux d'octets	Oui pour flux d'octets
Numérotation des lignes	LineNumber	Oui pour les flux de caractères	Non
Lecture avec remise dans le flux des données	PushBack	Oui	Non
Impression	Print	Non	Oui
Sérialisation	Object	Oui pour flux d'octets	Oui pour flux d'octets
Conversion octets/caractères	InputStream / OutputStream	Oui pour flux d'octets	Oui pour flux d'octets

- Buffered : ce type de filtre permet de mettre les données du flux dans un tampon. Il peut être utilisé en entrée et en sortie
- Sequence : ce filtre permet de fusionner plusieurs flux.
- Data : ce type de flux permet de traiter les octets sous forme de type de données
- LineNumber : ce filtre permet de numéroter les lignes contenues dans le flux
- PushBack : ce filtre permet de remettre des données lues dans le flux

- Print : ce filtre permet de réaliser des impressions formatées
- Object : ce filtre est utilisé par la sérialisation
- InputStream / OuputStream : ce filtre permet de convertir des octets en caractères

La package java.io définit ainsi plusieurs classes :

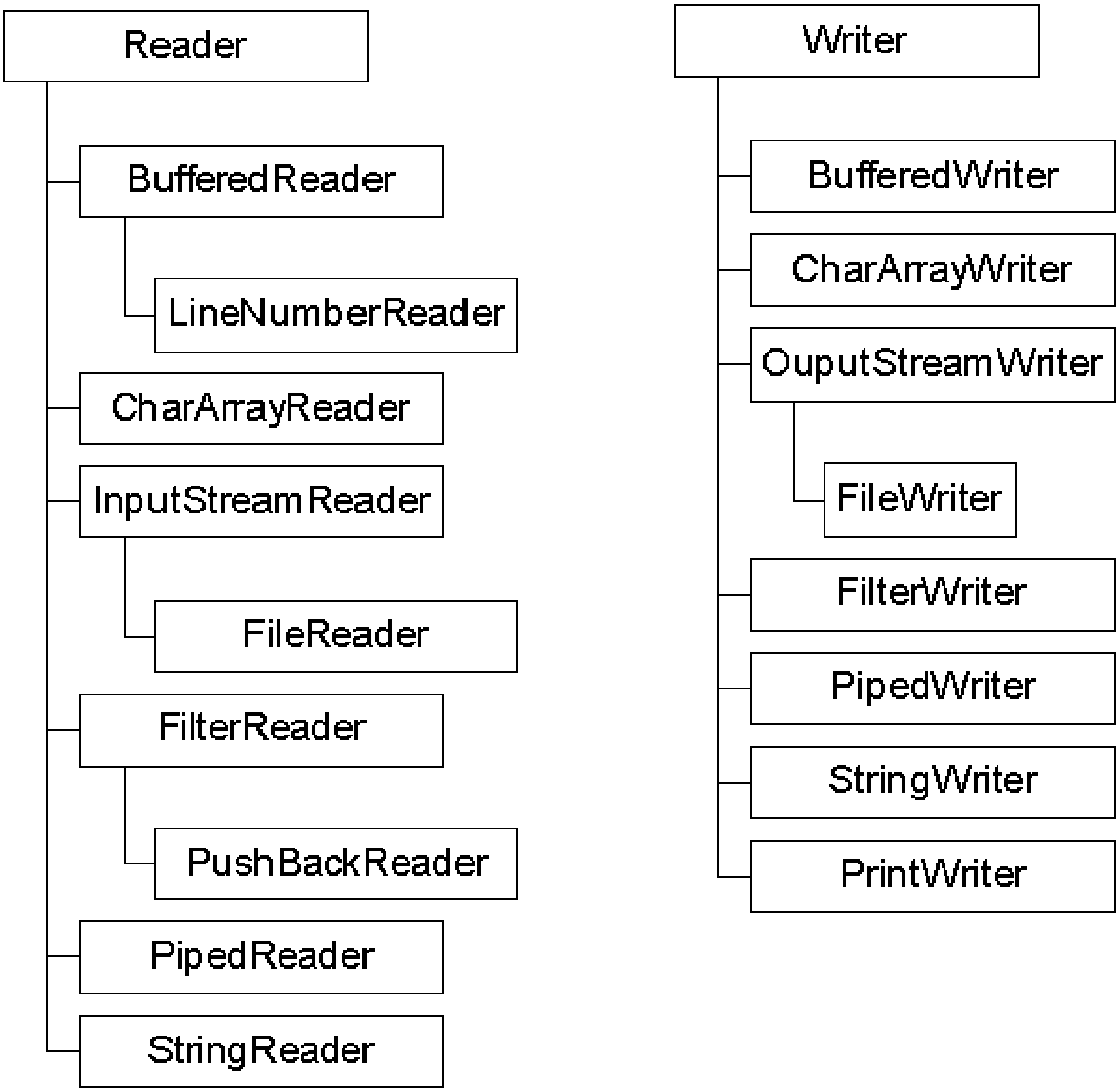
	Flux en lecture	Flux en sortie
Flux de caractères	BufferedReader	BufferedWriter
	CharArrayReader	CharArrayWriter
	FileReader	FileWriter
	InputStreamReader	OutputStreamWriter
	LineNumberReader	
	PipedReader	PipedWriter
	PushbackReader	
	StringReader	StringWriter
Flux d'octets	BufferedInputStream	BufferedOutputStream
	ByteArrayInputStream	ByteArrayOutputStream
	DataInputStream	DataOuputStream
	FileInputStream	FileOutputStream
	ObjectInputStream	ObjetOutputStream
	PipedInputStream	PipedOutputStream
		PrintStream
	PushbackInputStream	
	SequenceInputStream	

15.3. Les flux de caractères

Ils transportent des données sous forme de caractères : Java les gèrent avec le format Unicode qui code les caractères sur 2 octets.

Ce type de flux a été ajouté à partir du JDK 1.1.

Les classes qui gèrent les flux de caractères héritent d'une des deux classes abstraites Reader ou Writer. Il existe de nombreuses sous-classes pour traiter les flux de caractères.



15.3.1. La classe Reader

C'est une classe abstraite qui est la classe mère de toutes les classes qui gèrent des flux de caractères en lecture.

Cette classe définit plusieurs méthodes :

Méthodes	Rôles
boolean markSupported()	indique si le flux supporte la possibilité de marquer des positions
boolean ready()	indique si le flux est prêt à être lu
close()	ferme le flux et libère les ressources qui lui étaient associées
int read()	renvoie le caractère lu ou -1 si la fin du flux est atteinte.
int read(char[])	lire plusieurs caractères et les mettre dans un tableau de caractères
int read(char[], int, int)	lire plusieurs caractères. Elle attend en paramètre : un tableau de caractères qui contiendra les caractères lus, l'indice du premier élément du tableau qui recevra le premier caractère et le nombre de caractères à lire. Elle renvoie le nombre de caractères lus ou -1 si aucun caractère n'a été lu. Le tableau de caractères contient les caractères lus.
long skip(long)	saute autant de caractères dans le flux que la valeur fournie en paramètre. Elle renvoie le nombre de caractères sautés.

mark()	permet de marquer une position dans le flux
reset()	retourne dans le flux à la dernière position marquée

15.3.2. La classe Writer

C'est une classe abstraite qui est la classe mère de toutes les classes qui gèrent des flux de caractères en écriture.

Cette classe définit plusieurs méthodes :

Méthodes	Rôles
close()	ferme le flux et libère les ressources qui lui étaient associées
write(int)	écrire le caractère en paramètre dans le flux.
write(char[])	écrire le tableau de caractères en paramètre dans le flux.
write(char[], int, int)	écrire plusieurs caractères. Elle attend en paramètres : un tableau de caractères, l'indice du premier caractère et le nombre de caractères à écrire.
write(String)	écrire la chaîne de caractères en paramètre dans le flux
write(String, int, int)	écrire une portion d'une chaîne de caractères. Elle attend en paramètre : une chaîne de caractères, l'indice du premier caractère et le nombre de caractères à écrire.

15.3.3. Les flux de caractères avec un fichier

Les classes FileReader et FileWriter permettent de gérer des flux de caractères avec des fichiers.

15.3.3.1. Les flux de caractères en lecture sur un fichier

Il faut instancier un objet de la classe FileReader. Cette classe hérite de la classe InputStreamReader et possède plusieurs constructeurs qui peuvent tous lever une exception de type FileNotFoundException:

Constructeur	Rôle
FileInputReader(String)	Créer un flux en lecture vers le fichier dont le nom est précisé en paramètre.
FileInputReader(File)	Idem mais le fichier est précisé avec un objet de type File

Exemple (code Java 1.1) :

```
FileReader fichier = new FileReader("monfichier.txt");
```

Il existe plusieurs méthodes de la classe FileReader qui permettent de lire un ou plusieurs caractères dans le flux. Toutes ces méthodes sont héritées de la classe Reader et peuvent toutes lever l'exception IOException.

Une fois les traitements sur le flux terminés, il faut libérer les ressources qui lui sont allouées en utilisant la méthode close().

15.3.3.2. Les flux de caractères en écriture sur un fichier

Il faut instancier un objet de la classe FileWriter qui hérite de la classe OuputStreamWriter. Cette classe possède plusieurs constructeurs :

Constructeur	Rôle
FileWriter(String)	Si le nom du fichier précisé n'existe pas alors le fichier sera créé. S'il existe et qu'il contient des données celles-ci seront écrasées
FileWriter(File)	Idem mais le fichier est précisé avec un objet de la classe File
FileWriter(String, boolean)	Le booléen permet de préciser si les données seront ajoutées au fichier (valeur true) ou écraseront les données existantes (valeur false)

Exemple (code Java 1.1) :

```
FileWriter fichier = new FileWriter ("monfichier.dat");
```

Il existe plusieurs méthodes de la classe FileWriter héritées de la classe Writer qui permettent d'écrire un ou plusieurs caractères dans le flux.

Une fois les traitements sur le flux terminés, il faut libérer les ressources qui lui sont allouées en utilisant la méthode close().

15.3.4. Les flux de caractères tamponnés avec un fichier.

Pour améliorer les performances des flux sur un fichier, la mise en tampon des données lues ou écrites permet de traiter un ensemble de caractères représentant une ligne plutôt que de traiter les données caractères par caractères. Le nombre d'opérations est ainsi réduit.

Les classes BufferedReader et BufferedWriter permettent de gérer des flux de caractères tamponnés avec des fichiers.

15.3.4.1. Les flux de caractères tamponnés en lecture avec un fichier

Il faut instancier un objet de la classe BufferedReader. Cette classe possède plusieurs constructeurs qui peuvent tous lever une exception de type FileNotFoundException:

Constructeur	Rôle
BufferedReader(Reader)	le paramètre fourni doit correspondre au flux à lire.
BufferedReader(Reader, int)	l'entier en paramètre permet de préciser la taille du buffer. Il doit être positif sinon une exception de type IllegalArgumentException est levée.

Exemple (code Java 1.1) :

```
BufferedReader fichier  = new BufferedReader(new FileReader("monfichier.txt"));
```

Il existe plusieurs méthodes de la classe BufferedReader héritées de la classe Reader qui permettent de lire un ou plusieurs caractères dans le flux. Toutes ces méthodes peuvent lever une exception de type IOException. La classe BufferedReader définit une méthode supplémentaire pour la lecture :

Méthode	Rôle
String readLine()	lire une ligne de caractères dans le flux. Une ligne est une suite de caractères qui se termine par un retour chariot '\r' ou un saut de ligne '\n' ou les deux.

Elle possède plusieurs méthodes pour gérer le flux hérité de la classe Reader.

Une fois les traitements sur le flux terminés, il faut libérer les ressources qui lui sont allouées en utilisant la méthode close().

Exemple (code Java 1.1) :

```
import java.io.*;

public class TestBufferedReader {
  protected String source;

  public TestBufferedReader(String source) {
    this.source = source;
    lecture();
  }

  public static void main(String args[]) {
    new TestBufferedReader("source.txt");
  }

  private void lecture() {
    try {
      String ligne ;
      BufferedReader fichier = new BufferedReader(new FileReader(source));

      while ((ligne = fichier.readLine()) != null) {
          System.out.println(ligne);
      }

      fichier.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

15.3.4.2. Les flux de caractères tamponnés en écriture avec un fichier

Il faut instancier un objet de la classe BufferedWriter. Cette classe possède plusieurs constructeurs :

Constructeur	Rôle
BufferedWriter(Writer)	le paramètre fourni doit correspondre au flux dans lequel les données sont écrites.
BufferedWriter(Writer, int)	l'entier en paramètre permet de préciser la taille du buffer. Il doit être positif sinon une exception IllegalArgumentException est levée.

Exemple (code Java 1.1) :

```
BufferedWriter fichier = new BufferedWriter( new FileWriter("monfichier.txt"));
```

Il existe plusieurs méthodes de la classe BufferedWriter héritées de la classe Writer qui permettent de lire un ou plusieurs caractères dans le flux.

La classe BufferedWriter possède plusieurs méthodes pour gérer le flux :

Méthode	Rôle
flush()	vide le tampon en écrivant les données dans le flux.
newLine()	écrire un séparateur de ligne dans le flux

Une fois les traitements sur le flux terminés, il faut libérer les ressources qui lui sont allouées en utilisant la méthode close().

Exemple (code Java 1.1) :

```
import java.io.*;
import java.util.*;

public class TestBufferedWriter {
  protected String destination;

  public TestBufferedWriter(String destination) {
    this.destination = destination;
    traitement();
  }

  public static void main(String args[]) {
    new TestBufferedWriter("print.txt");
  }

  private void traitement() {
    try {
      String ligne ;
      int nombre = 123;
      BufferedWriter fichier = new BufferedWriter(new FileWriter(destination));

      fichier.write("bonjour tout le monde");
      fichier.newLine();
      fichier.write("Nous sommes le "+ new Date());
      fichier.write(", le nombre magique est " + nombre);

      fichier.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

15.3.4.3. La classe PrintWriter

Cette classe permet d'écrire dans un flux des données formatées.

Cette classe possède plusieurs constructeurs :

Constructeur	Rôle
PrintWriter(Writer)	Le paramètre fourni précise le flux. Le tampon est automatiquement vidé.
PrintWriter(Writer, boolean)	Le booléen permet de préciser si le tampon doit être automatiquement vidé
PrintWriter(OutputStream)	Le paramètre fourni précise le flux. Le tampon est automatiquement vidé.
PrintWriter(OutputStream, boolean)	Le booléen permet de préciser si le tampon doit être automatiquement vidé

Exemple (code Java 1.1) :

```
PrintWriter fichier = new PrintWriter( new FileWriter("monfichier.txt"));
```

Il existe de nombreuses méthodes de la classe PrintWriter qui permettent d'écrire un ou plusieurs caractères dans le flux en les formatant. Les méthodes write() sont héritées de la classe Writer qui définit plusieurs méthodes pour envoyer des données formatées dans le flux :

- print(...)

  Plusieurs surcharges de la méthode print() acceptent des données de différents types pour les convertir en caractères et les écrire dans le flux
- println()

  Cette méthode permet de terminer la ligne courante dans le flux en y écrivant un saut de ligne.

- println (...)

  Plusieurs surcharges de la méthode println() acceptent des données de différents types pour les convertir en caractères et les écrire dans le flux avec une fin de ligne.

La classe PrintWriter possède plusieurs méthodes pour gérer le flux :

Méthode	Rôle
close()	Ferme le tampon et libérer les ressources associées
boolean checkError()	Vide le tampon et renvoie true si une exception est levée lors de l'utilisation du flux sous-jacent
flush()	Vide le tampon en écrivant les données dans le flux.

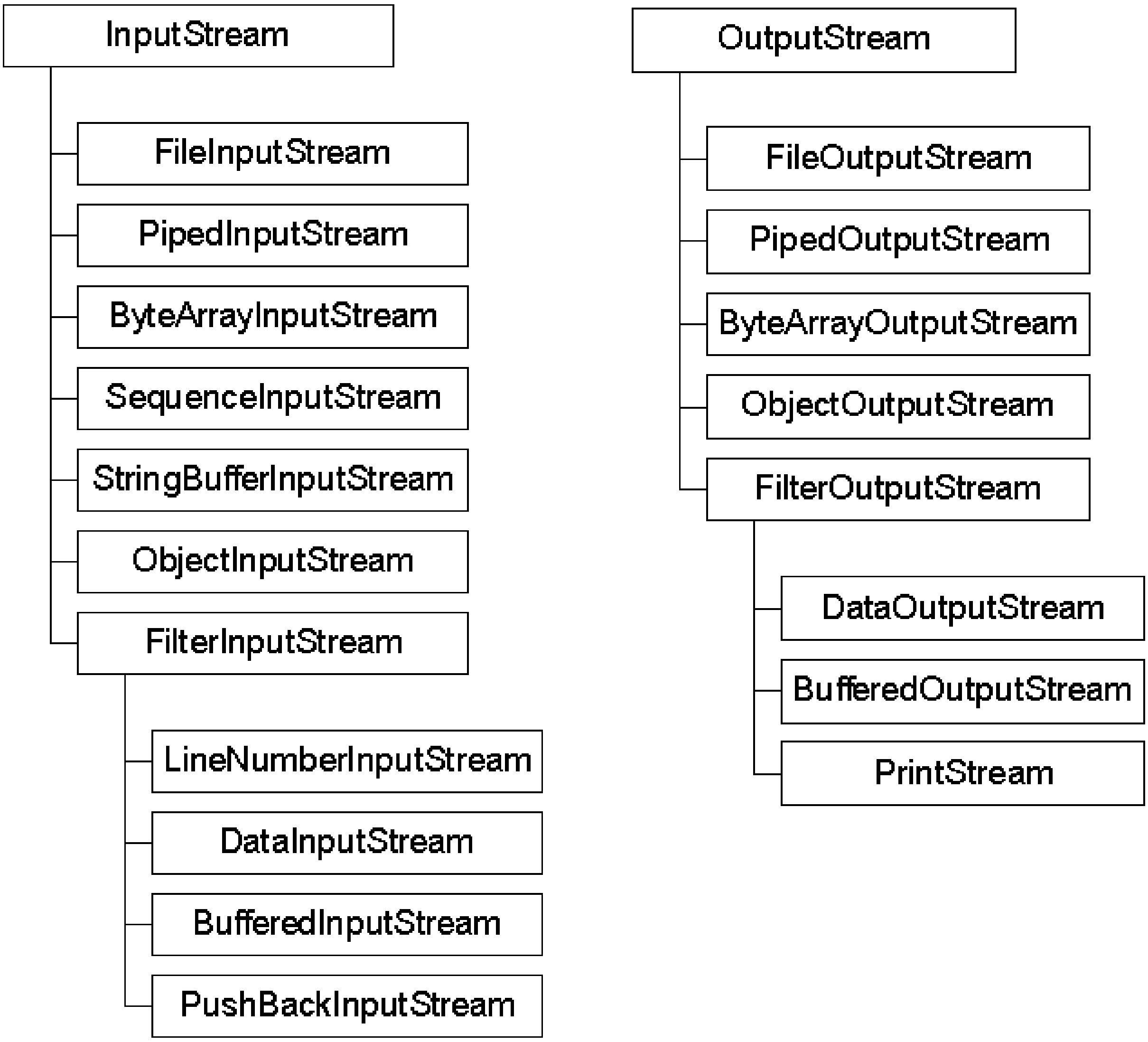
Exemple (code Java 1.1) :

```
import java.io.*;
import java.util.*;

public class TestPrintWriter {
  protected String destination;

  public TestPrintWriter(String destination) {
    this.destination = destination;
    traitement();
  }

  public static void main(String args[]) {
    new TestPrintWriter("print.txt");
  }

  private void traitement() {
    try {
      String ligne ;
      int nombre = 123;
      PrintWriter fichier = new PrintWriter(new FileWriter(destination));

      fichier.println("bonjour tout le monde");
      fichier.println("Nous sommes le "+ new Date());
      fichier.println("le nombre magique est " + nombre);

      fichier.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

15.4. Les flux d'octets

Ils transportent des données sous forme d'octets. Les flux de ce type sont capables de traiter toutes les données.

Les classes qui gèrent les flux d'octets héritent d'une des deux classes abstraites InputStream ou OutputStream. Il existe de nombreuses sous-classes pour traiter les flux d'octets.



15.4.1. Les flux d'octets avec un fichier.

Les classes FileInputStream et FileOutputStream permettent de gérer des flux d'octets avec des fichiers.

15.4.1.1. Les flux d'octets en lecture sur un fichier

Il faut instancier un objet de la classe FileInputStream. Cette classe possède plusieurs constructeurs qui peuvent tous lever l'exception FileNotFoundException:

Constructeur	Rôle
FileInputStream(String)	Ouvre un flux en lecture sur le fichier dont le nom est donné en paramètre
FileInputStream(File)	Idem mais le fichier est précisé avec un objet de type File

Exemple (code Java 1.1) :

```
FileInputStream fichier = new FileInputStream("monfichier.dat");
```

Il existe plusieurs méthodes de la classe FileInputStream qui permettent de lire un ou plusieurs octets dans le flux. Toutes ces méthodes peuvent lever l'exception IOException.

- int read()

  Cette méthode envoie la valeur de l'octet lu ou -1 si la fin du flux est atteinte.

  Exemple (code Java 1.1) :

  ```
  int octet = 0;
  while (octet !=  -1 ) {
     octet = fichier.read();
  }
  ```

- int read(byte[], int, int)

  Cette méthode lit plusieurs octets. Elle attend en paramètre : un tableau d'octets qui contiendra les octets lus, l'indice de élément du tableau qui recevra le premier octet et le nombre d'octets à lire.

  Elle renvoie le nombre d'octets lus ou -1 si aucun octet n'a été lu.

La classe FileInputStream possède plusieurs méthodes pour gérer le flux :

Méthode	Rôle
long skip(long)	saute autant d'octets dans le flux que la valeur fournie en paramètre. Elle renvoie le nombre d'octets sautés.
close()	ferme le flux et libère les ressources qui lui étaient associées
int available()	retourne une estimation du nombre d'octets qu'il est encore possible de lire dans le flux

Une fois les traitements sur le flux terminés, il faut libérer les ressources qui lui sont allouées en utilisant la méthode close().

15.4.1.2. Les flux d'octets en écriture sur un fichier

Il faut instancier un objet de la classe FileOutputStream. Cette classe possède plusieurs constructeurs :

Constructeur	Rôle
FileOutputStream(String)	Si le fichier précisé n'existe pas, il sera créé. Si il existe et qu'il contient des données celles-ci seront écrasées.
FileOutputStream(String, boolean)	Le booléen permet de préciser si les données seront ajoutées au fichier (valeur true) ou écraseront les données existantes (valeur false)

Exemple (code Java 1.1) :

```
FileOuputStream fichier = new FileOutputStream("monfichier.dat");
```

Il existe plusieurs méthodes de la classe FileOutputStream qui permettent d'écrire un ou plusieurs octets dans le flux.

Méthode	Rôle
write(int)	Cette méthode écrit l'octet en paramètre dans le flux
write(byte[])	Cette méthode écrit plusieurs octets. Elle attend en paramètre : un tableau d'octets qui contient les octets à écrire : tous les éléments du tableau sont écrits
write(byte[], int, int)	

	Cette méthode écrit plusieurs octets. Elle attend en paramètre : un tableau d'octets qui contient les octets à écrire, l'indice du premier élément du tableau d'octets à écrire et le nombre d'octets à écrire

Une fois les traitements sur le flux terminés, il faut libérer les ressources qui lui sont allouées en utilisant la méthode close().

Exemple (code Java 1.1) :

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopieFichier {

  public static void main(String args[]) {
    try {
      copierFichier("source.txt", "copie.txt");
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  public static void copierFichier(String source, String destination) throws IOException {
    FileInputStream fis = null;
    FileOutputStream fos = null;

    try {
      byte buffer[] = new byte[1024];
      int taille = 0;

      fis = new FileInputStream(source);
      fos = new FileOutputStream(destination);
      while ((taille = fis.read(buffer)) != -1) {
        System.out.println(taille);
        fos.write(buffer, 0, taille);
      }
    } finally {
      if (fis != null) {
        try {
          fis.close();
        } catch (IOException e) {
        }
      }
      if (fos != null) {
        try {
          fos.close();
        } catch (IOException e) {
        }
      }
    }
  }
}
```

Remarque : cet exemple est fourni à titre indicatif pour démontrer l'utilisation de l'API. Il est préférable d'utiliser l'API Apache Commons IO ou la méthode copy() de la classe Files de Java 7 plutôt que de réécrire une méthode de copie d'un fichier.

15.4.2. Les flux d'octets tamponnés avec un fichier.

Pour améliorer les performances des flux sur un fichier, la mise en tampon des données lues ou écrites permet de traiter un ensemble d'octets plutôt que de traiter les données octet par octet. Le nombre d'opérations est ainsi réduit.

15.5. La classe File

Les fichiers et les répertoires sont encapsulés dans la classe File du package java.io. Il n'existe pas de classe pour traiter les répertoires car ils sont considérés comme des fichiers. Une instance de la classe File est une représentation logique d'un fichier ou d'un répertoire qui peut ne pas exister physiquement sur le disque.

Si le fichier ou le répertoire existe, de nombreuses méthodes de la classe File permettent d'obtenir des informations sur le fichier. Sinon plusieurs méthodes permettent de créer des fichiers ou des répertoires. Voici une liste des principales méthodes :

Méthode	Rôle
boolean canRead()	Indiquer si le fichier peut être lu
boolean canWrite()	Indiquer si le fichier peut être modifié
boolean createNewFile()	Java 1.2 Créer un nouveau fichier vide
File createTempFile(String, String)	Java 1.2 Créer un nouveau fichier dans le répertoire par défaut des fichiers temporaires. Les deux arguments sont le nom et le suffixe du fichier
File createTempFile(String, String, File)	Créer un nouveau fichier temporaire. Les trois arguments sont le nom, le suffixe du fichier et le répertoire
boolean delete()	Détruire le fichier ou le répertoire. Le booléen indique le succès de l'opération
deleteOnExit()	Java 1.2 Demander la suppression du fichier à l'arrêt de la JVM
boolean exists()	Indique si le fichier existe physiquement
String getAbsolutePath()	Renvoyer le chemin absolu du fichier
String getPath	Renvoyer le chemin du fichier
boolean isAbsolute()	Indiquer si le chemin est absolu
boolean isDirectory()	Indiquer si le fichier est un répertoire
boolean isFile()	Indiquer si l'objet représente un fichier
long length()	Renvoyer la longueur du fichier
String[] list()	Renvoyer la liste des fichiers et répertoires contenus dans le répertoire
boolean mkdir()	Créer le répertoire
boolean mkdirs()	Créer le répertoire avec création des répertoires manquants dans l'arborescence du chemin
boolean renameTo()	Renommer le fichier

Depuis la version 1.2 du J.D.K., de nombreuses fonctionnalités ont été ajoutées à cette classe :

- la création de fichiers temporaires (createNewFile, createTempFile, deleteOnExit)
- la gestion des attributs "caché" et "lecture seule" (isHidden, isReadOnly)
- des méthodes qui renvoient des objets de type File au lieu du type String (getParentFile, getAbsoluteFile, getCanonicalFile, listFiles)
- une méthode qui renvoie le fichier sous forme d'URL (toURL)

Exemple (code Java 1.1) :

```
import java.io.*;
```

```
public class TestFile {
  protected String nomFichier;
  protected File fichier;

  public TestFile(String nomFichier) {
    this.nomFichier = nomFichier;
    fichier = new File(nomFichier);
    traitement();
  }

  public static void main(String args[]) {
    new TestFile(args[0]);
  }

  private void traitement() {

    if (!fichier.exists()) {
      System.out.println("le fichier "+nomFichier+" n'existe pas");
      System.exit(1);
    }

    System.out.println(" Nom du fichier    : "+fichier.getName());
    System.out.println(" Chemin du fichier : "+fichier.getPath());
    System.out.println(" Chemin absolu     : "+fichier.getAbsolutePath());
    System.out.println(" Droit de lecture  : "+fichier.canRead());
    System.out.println(" Droit d'ecriture  : "+fichier.canWrite());

    if (fichier.isDirectory() ) {
      System.out.println(" contenu du repertoire ");
      String fichiers[] = fichier.list();
      for(int i = 0; i > fichiers.length; i++) System.out.println("  "+fichiers[i]);
    }
  }
}
```

Exemple (code Java 1.2) :

```
import java.io.*;

public class TestFile_12 {
  protected String nomFichier;
  protected File fichier;

  public TestFile_12(String nomFichier) {
    this.nomFichier = nomFichier;
    fichier = new File(nomFichier);
    traitement();
  }

  public static void main(String args[]) {
    new TestFile_12(args[0]);
  }

  private void traitement() {

    if (!fichier.exists()) {
      System.out.println("le fichier "+nomFichier+"n'existe pas");
      System.exit(1);
    }

    System.out.println(" Nom du fichier    : "+fichier.getName());
    System.out.println(" Chemin du fichier : "+fichier.getPath());
    System.out.println(" Chemin absolu     : "+fichier.getAbsolutePath());
    System.out.println(" Droit de lecture  : "+fichier.canRead());
    System.out.println(" Droit d'ecriture  : "+fichier.canWrite());

    if (fichier.isDirectory() ) {
      System.out.println(" contenu du repertoire ");
      File fichiers[] = fichier.listFiles();
      for(int i = 0; i > fichiers.length; i++) {

        if (fichiers[i].isDirectory())
          System.out.println("  ["+fichiers[i].getName()+"]");
        else
          System.out.println("  "+fichiers[i].getName());
      }
    }
  }
}
```

15.6. Les fichiers à accès direct

Les fichiers à accès direct permettent un accès rapide à un enregistrement contenu dans un fichier. Le plus simple pour utiliser un tel type de fichier est qu'il contienne des enregistrements de taille fixe mais ce n'est pas obligatoire. Il est possible dans un tel type de fichier de mettre à jour directement un de ses enregistrements.

La classe RamdonAccessFile encapsule les opérations de lecture/écriture d'un tel fichier. Elle implémente les interfaces DataInput et DataOutput.

Elle possède deux constructeurs qui attendent en paramètres le fichier à utiliser (sous la forme d'un nom de fichier ou d'un objet de type File qui encapsule le fichier) et le mode d'accès.

Le mode est une chaîne de caractères qui doit être égale à «r» ou «rw» selon que le mode est lecture seule ou lecture/écriture.

Ces deux constructeurs peuvent lever les exceptions suivantes :

- FileNotFoundException si le fichier n'est pas trouvé
- IllegalArgumentException si le mode n'est pas «r» ou «rw»
- SecurityException si le gestionnaire de sécurité empêche l'accès aux fichiers dans le mode précisé

La classe RandomAccessFile possède de nombreuses méthodes writeXXX() pour écrire des types primitifs dans le fichier.

Exemple :

```
package fr.jmdoudoux.dej;

import java.io.RandomAccessFile;

public class TestRandomAccesFile {
  public static void main(String[] args) {
    try {
      RandomAccessFile monFichier = new RandomAccessFile("monfichier.dat", "rw");
      for (int i = 0; i < 10; i++) {
        monFichier.writeInt(i * 100);
      }
      monFichier.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Elle possède aussi de nombreuses classes readXXX() pour lire des données primitives dans le fichier.

Exemple :

```
package fr.jmdoudoux.dej;

import java.io.RandomAccessFile;

public class TestRandomAccesFile {
  public static void main(String[] args) {
    try {
      RandomAccessFile monFichier = new RandomAccessFile("monfichier.dat", "rw");
      for (int i = 0; i < 10; i++) {
         System.out.println(monFichier.readInt());
      }
      monFichier.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
  0
  100
  200
  300
  400
  500
  600
  700
  800
  900
```

Pour naviguer dans le fichier, la classe utilise un pointeur qui indique la position dans le fichier où les opérations de lecture ou de mise à jour doivent être effectuées. La méthode getFilePointer() permet de connaître la position de ce pointeur et la méthode seek() permet de le déplacer.

La méthode seek() attend en paramètre un entier long qui réprésente la position, dans le fichier, précisée en octets. La première position commence à zéro.

Exemple : lecture de la sixième données

```
package fr.jmdoudoux.dej;

import java.io.RandomAccessFile;

public class TestRandomAccesFile {
  public static void main(String[] args) {
    try {
      RandomAccessFile monFichier = new RandomAccessFile("monfichier.dat", "rw");
      // 5 représente le sixième enregistement puisque le premier commence à 0
      // 4 est la taille des données puisqu'elles sont des entiers de type int
      // (codé sur 4 octets)
      monFichier.seek(5*4);
      System.out.println(monFichier.readInt());
      monFichier.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
500
```

15.7. La classe java.io.Console

La classe java.io.Console, ajoutée dans Java SE 6, permet un accès à la console du système d'exploitation pour permettre la saisie ou l'affichage de données. Cette nouvelle classe fait usage des flux de type Reader et Writer ce qui permet une gestion correcte des caractères.

La classe System possède une méthode console() qui permet d'obtenir une instance de la classe Console.

La méthode printf() permet de formater et d'afficher des données.

La méthode readLine() permet la saisie d'une ligne de données dont les caractères sont affichés sur la console.

La méthode readPassword() est identique à la méthode readLine() mais les caractères saisis ne sont pas affichés sur la console.

Exemple :

```
package fr.jmdoudoux.dej.java6;

public class TestConsole {

        public static void main(String args[]) {
            String string = "La façade nécessaire";
            System.out.println(string);
            System.console().printf("%s%n", string);
    }

}
```

Résultat :

```
C:\java\TestJava6\classes>java fr.jmdoudoux.dej.java6.TestConsole
La faþade nÚcessaire
La façade nécessaire
```

16. NIO 2

Chapitre 16

Niveau : 🟩🟩🟨⬜⬜⬜ Intermédiaire

L'API NIO 2 a été développée sous la JSR 203 et a été ajoutée au JDK dans la version 7 de Java SE. NIO 2 est une API plus moderne et plus complète pour l'accès au système de fichiers. Son but est en partie de remplacer la classe File de la très ancienne API IO.

NIO 2 propose d'étendre les fonctionnalités relatives aux entrées/sorties : l'utilisation du système de fichiers de manière facile et les lectures/écritures asynchrones.

L'API FileSystem simplifie grandement la manipulation de fichiers et répertoires d'un système de fichiers et ajoute des fonctionnalités attendues depuis longtemps. La nouvelle API de gestion et d'accès au système de fichiers est contenue dans le package java.nio.file et ses sous-packages.

Parmi les nouvelles fonctionnalités proposées par NIO2, on peut trouver :

- le support des liens physiques et symboliques s'ils sont pris en charge par le système de fichiers
- la gestion des attributs sur les fichiers des systèmes Dos et POSIX
- Le support de notifications en cas de changement dans le contenu d'un répertoire (ajout, suppression, modification d'un fichier du répertoire) en utilisant l'API WatchService
- le support du parcours d'un répertoire avec la possibilité de filtrer les fichiers obtenus
- l'utilisation de channels asynchrones avec lesquels les opérations de lecture/écriture sont réalisées en utilisant un pool de threads
- l'ajout de fonctionnalités de base comme la copie ou le déplacement de fichiers
- l'utilisation de fabriques pour permettre à l'API d'être extensible : il est par exemple possible de créer sa propre implémentation d'un système de fichiers. Une implémentation permettant de gérer les fichiers zip est d'ailleurs fournie en standard.

NIO2 est probablement l'API de Java 7 qui sera la plus utilisée par les développeurs tant elle facilite la mise en oeuvre de fonctionnalités courantes d'entrées/sorties sur un système de fichiers.

Ce chapitre contient plusieurs sections :

- ♦ Les entrées/sorties avec Java
- ♦ Les principales classes et interfaces
- ♦ L'interface Path
- ♦ Glob
- ♦ La classe Files
- ♦ Le parcours du contenu de répertoires
- ♦ L'utilisation de systèmes de gestion de fichiers
- ♦ La lecture et l'écriture dans un fichier
- ♦ Les liens et les liens symboliques
- ♦ La gestion des attributs
- ♦ La gestion des unités de stockages
- ♦ Les notifications de changements dans un répertoire
- ♦ La gestion des erreurs et la libération des ressources
- ♦ L'interopérabilité avec le code existant

16.1. Les entrées/sorties avec Java

Depuis les débuts de Java, l'API java.io est essentiellement composée de classes et d'interfaces pour réaliser des opérations sur les flux d'octets ou de caractères. Seule la classe File permet des opérations sur les fichiers et les répertoires du système de fichiers.

L'utilisation du système de fichiers se fait donc en utilisant l'API java.io et notamment la classe java.io.File qui présente de nombreux inconvénients :

- Plusieurs méthodes ne lèvent pas une exception en cas de problème mais renvoient un booléen. Ceci ne respecte pas ce qu'il est possible de faire pour gérer les erreurs avec Java, rend l'API incohérente et ne permet pas de connaître l'origine du problème mais seulement de savoir que la fonctionnalité a échouée
- La méthode rename() n'a pas le même comportement sur toutes les plates-formes
- Il n'y a pas de réel support pour les liens symboliques (symbolic links)
- Les métadonnées (attributs de permissions, propriétaire, sécurité, ...) sont peu ou mal supportées
- Certaines fonctionnalités de base sont absentes de l'API comme la copie ou le déplacement d'un fichier
- Certaines fonctionnalités sont peu performantes comme par exemple la méthode listFiles() avec un répertoire contenant de nombreux fichiers
- ...

La classe java.io.File existe depuis Java 1.0 mais elle a évoluée dans plusieurs versions de Java :

Version	Méthodes ajoutées
1.1	getCanonicalPath()
1.2	getParentFile(), getAbsoluteFile(),getCanonicalFile(), toURL(), isHidden(), createNewFile(), deleteOnExit(), listFiles(), setLastModified(), setReadOnly(), listRoots(), createTempFile(), compareTo()
1.4	création à partir d'une URI, toURI()
6	setWritable(), setReadable(), setExecutable(), canExecute(), getTotalSpace(), getFreeSpace(), getUsableSpace()

L'API NIO a été introduite dans Java 1.4 : elle propose entre autres l'utilisation de channels, buffers et charsets notamment pour permettre de réaliser des opérations de lectures/écritures non bloquantes (non blocking I/O).

L'API NIO 2 est une API plus moderne qui propose plusieurs caractéristiques :

- la séparation des responsabilités : un chemin (Path) représente un élément du système de fichiers (FileSystem) stocké dans un système de stockage (FileStorage) et est manipulé en utilisant la classe Files
- la gestion de toutes les erreurs se fait avec des exceptions
- l'utilisation de fabriques permet de créer les différentes instances de l'API et de la rendre extensible

La classe java.io.File n'est pas deprecated mais à partir de Java 7, il est recommandé d'utiliser les classes et interfaces de l'API NIO 2 dans la mesure du possible : ceci doit être le cas dans les nouveaux développements d'autant que, pour faciliter l'intégration dans le code existant, il existe des fonctionnalités pour convertir un objet de type File en un objet de type Path et vice versa.

16.2. Les principales classes et interfaces

NIO 2 repose sur plusieurs classes et interfaces dont les principales sont :

- Path : encapsule un chemin dans le système de fichiers
- Files : contient des méthodes statiques pour manipuler les éléments du système de fichiers
- FileSystemProvider : service provider qui interagit avec le système de fichiers sous-jacent
- FileSystem : encapsule un système de fichiers

- FileSystems : fabrique qui permet de créer une instance de FileSystem

Ces classes et interfaces sont regroupées dans le package java.nio.file et ses sous-packages :

- java.nio.file
- java.nio.file.attribute

L'interface Path décrit les fonctionnalités d'une classe qui encapsule un chemin sur le système de fichiers. Ce chemin est représenté sous la forme d'une séquence de noms qui compose la hiérarchie des répertoires du chemin. Cette séquence peut inclure le nom d'un fichier ou d'un répertoire comme dernier élément mais pas obligatoirement car un objet de type Path peut simplement encapsuler un sous-chemin.

Les méthodes de l'interface Path permettent uniquement de manipuler les éléments qui composent le chemin : elles n'ont aucune action sur le système de fichiers sous-jacent du chemin.

La classe FileStorage encapsule un système de stockage de fichiers. Elle permet d'obtenir des informations sur le système de stockage comme l'espace total ou l'espace libre. Une instance de type FileStorage est obtenue en invoquant la méthode Files.getFileStore() en lui passant en paramètre un objet de type Path encapsulant un élément du système de stockage.

La classe FileSystem est une fabrique pour créer des objets relatifs à un système de fichiers. La méthode getPath() permet d'obtenir une instance d'un chemin dans le système de fichiers. La méthode getFileStores() permet d'obtenir une collection de tous les systèmes de stockage utilisables.

La classe FileSystems permet de créer des objets de type FileSystem. La méthode statique getDefault() permet d'obtenir une instance du FileSystem par défaut. La classe FileSystems permet aussi de créer des instances personnalisées de classes de type FileSystem.

16.3. L'interface Path

La classe Path est une des interfaces principales de NIO 2 : elle encapsule tout ou partie d'un chemin vers un élément du système de fichiers de manière dépendante du système d'exploitation sous-jacent.

Le chemin peut concerner plusieurs types d'éléments :

- Un fichier
- Un répertoire
- Un lien symbolique : permet de faire référence à un fichier ou un autre répertoire
- Un sous-chemin

Ce chemin peut être absolu (le chemin contient une racine) ou relatif (en combinaison avec le chemin courant pour obtenir le chemin absolu). La représentation d'un chemin dépend du système de fichiers sous-jacent : par exemple, tous les systèmes d'exploitation n'utilisent pas tous le même séparateur entre les éléments d'un chemin. Un objet de type Path encapsule le chemin d'un élément du système de fichiers composé d'un ensemble d'éléments organisés de façon hiérarchique grâce à un séparateur spécifique au système.

Une instance de type Path encapsule les informations sur le chemin permettant de localiser un fichier ou un répertoire dans un système de fichiers. Elle peut contenir la racine ou le nom du fichier mais aucun des deux n'est obligatoire : elle peut contenir un sous-chemin ou uniquement le nom du fichier.

Le chemin d'un fichier ou d'un répertoire encapsulé dans une instance de type Path n'a pas forcément d'existence physique sur le système de fichiers sous-jacents.

Les instances de type Path sont immuables et utilisables dans un contexte multithread.

Elle hérite de plusieurs interfaces : Comparable<Path>, Iterable<Path> et Watchable.

La classe Path possède plusieurs méthodes qui peuvent être utilisées pour obtenir des informations sur le chemin, accéder aux éléments du chemin, convertir le chemin ou extraire des sous-chemins, ... Ces méthodes traitent le chemin lui-même mais sont sans action sur le système de fichiers sous-jacents. Aucune méthode ne concerne la gestion des extensions des fichiers.

16.3.1. L'obtention d'une instance de type Path

Il n'est pas possible de créer une instance de type Path sans utiliser une fabrique ou un helper qui invoque une fabrique.

Il existe plusieurs manières de créer un objet de type Path :

- invoquer la méthode getPath() d'une instance de type FileSystem
- invoquer la méthode Paths.get() qui invoque la méthode FileSystems.getDefault().getPath()
- invoquer la méthode toPath() sur un objet de type java.io.File

La méthode getPath() de la classe FileSystem permet d'obtenir une instance de type Path.

Exemple (code Java 7) :

```
Path chemin = FileSystems.getDefault()
  .getPath("C:/Users/jm/AppData/Local/Temp/monfichier.txt");
```

La classe Paths est un helper qui permet de créer facilement des instances de type Path : c'est une fabrique proposant deux surcharges de sa méthode get() qui attendent respectivement en paramètres un nombre variable d'objets de type String qui sont les éléments du chemin ou une URI.

Exemple (code Java 7) :

```
Path chemin1 = Paths.get("C:/Users/jm/AppData/Local/Temp/monfichier.txt");

Path chemin2 = Paths.get(URI.create("file:///C:/Users/jm/AppData/Local/Temp/monfichier.txt"));

Path chemin3 = Paths.get(System.getProperty("java.io.tmpdir"), "monfichier.txt");
```

Le chemin précisé peut utiliser le séparateur du système sous-jacent.

Exemple (code Java 7) :

```
Path chemin1 = Paths.get("C:\\Users\\jm\\AppData\\Local\\Temp\\monfichier.txt");
```

16.3.2. L'obtention d'éléments du chemin

Une instance de type Path stocke les éléments de la hiérarchie du chemin sous une forme séquentielle, l'élément le plus haut dans la hiérarchie (après la racine) ayant l'index 0 et l'élément le plus bas ayant l'index n-1, n étant le nombre d'éléments du chemin.

L'interface Path propose plusieurs méthodes pour retrouver un élément particulier ou un sous-chemin composé de plusieurs éléments en utilisant les index.

Méthode	Rôle
String getFileName()	Retourner le nom du dernier élément du chemin. Si le chemin concerne un fichier alors c'est le nom du fichier qui est retourné
Path getName(int index)	Retourner l'élément du chemin dont l'index est fourni en paramètre. Le premier élément possède l'index 0
int getNameCount()	Retourner le nombre d'éléments du chemin
Path getParent()	Retourner le chemin parent ou null s'il n'existe pas (dans ce cas, le chemin correspond à une racine)
Path getRoot()	Retourner la racine d'un chemin absolu (par exemple C:\ sous Dos ou / sous Unix) ou null pour un chemin relatif

String toString()	Retourner le chemin sous la forme d'une chaîne de caractères
Path subPath(int beginIndex, int endIndex)	Retourner un sous-chemin correspondant aux deux index fournis en paramètres

Exemple (code Java 7) :

```
    Path path = Paths.get("C:/Users/jm/AppData/Local/Temp/monfichier.txt");
    System.out.println("toString()     = " + path.toString());
    System.out.println("getFileName()  = " + path.getFileName());
    System.out.println("getRoot()      = " + path.getRoot());
    System.out.println("getName(0)     = " + path.getName(0));
    System.out.println("getNameCount() = " + path.getNameCount());
    System.out.println("getParent()    = " + path.getParent());
    System.out.println("subpath(0,3)   = " + path.subpath(0,3));
```

Résultat :

```
toString()     = C:\Users\jm\AppData\Local\Temp\monfichier.txt
getFileName()  = monfichier.txt
getRoot()      = C:\
getName(0)     = Users
getNameCount() = 6
getParent()    = C:\Users\jm\AppData\Local\Temp
subpath(0,3)   = Users\jm\AppData
```

Le chemin peut aussi être relatif.

Exemple (code Java 7) :

```
    Path path = Paths.get("jm/AppData/Local/Temp/monfichier.txt");
    System.out.println("toString()     = " + path.toString());
    System.out.println("getFileName()  = " + path.getFileName());
    System.out.println("getRoot()      = " + path.getRoot());
    System.out.println("getName(0)     = " + path.getName(0));
    System.out.println("getNameCount() = " + path.getNameCount());
    System.out.println("getParent()    = " + path.getParent());
    System.out.println("subpath(0,3)   = " + path.subpath(0, 3));
```

Résultat :

```
toString()     = jm\AppData\Local\Temp\monfichier.txt
getFileName()  = monfichier.txt
getRoot()      = null
getName(0)     = jm
getNameCount() = 5
getParent()    = jm\AppData\Local\Temp
subpath(0,3)   = jm\AppData\Local
```

Une instance de type Path implémente l'interface Iterator qui permet de réaliser une itération sur les éléments du chemin.

Exemple (code Java 7) :

```
    Path path = Paths.get("C:/Users/jm/AppData/Local/Temp/monfichier.txt");
    for (Path name : path) {
      System.out.println(name);
    }
```

Résultat :

```
Users
jm
AppData
Local
Temp
monfichier.txt
```

16.3.3. La manipulation d'un chemin

L'interface Path propose plusieurs méthodes pour manipuler les chemins :

Méthode	Rôle
Path normalize()	Nettoyer le chemin en supprimant les éléments « . » et « .. » qu'il contient
Path relativize(Path other)	Retourner le chemin relatif à celui fourni en paramètres
Path resolve(Path)	Combiner deux chemins

La méthode normalize() permet d'expliciter un chemin en éliminant les éléments comme « . » et « .. »

Exemple (code Java 7) :

```
    Path path = Paths.get("C:/Users/jm/AppData/Local/Temp/./monfichier.txt");

    System.out.println("normalize()   = " + path.normalize());
    path = Paths.get("C:/Users/admin/  ./../jm/AppData/Local/Temp/./monfichier.txt");

    System.out.println("normalize()   = " + path.normalize());
```

Résultat :

```
normalize()   = C:\Users\jm\AppData\Local\Temp\monfichier.txt
normalize()   = C:\Users\jm\AppData\Local\Temp\monfichier.txt
```

La méthode normalize() effectue une opération purement syntaxique : elle ne vérifie pas dans le système de fichiers le chemin qu'elle produit.

La méthode resolve() permet de combiner deux chemins. Elle attend en paramètre un chemin partiel qui ne doit pas commencer par un élément racine du système de fichiers. Si le chemin fourni en paramètre contient un élément racine, alors la méthode resolve() renvoie le chemin fourni en paramètre.

Exemple (code Java 7) :

```
    Path path = Paths.get("C:/Users/jm/AppData/Local/");
    Path nouveauPath = path.resolve("Temp/monfichier.txt");
    System.out.println(nouveauPath);
    nouveauPath = path.resolve("C:/Temp");
    System.out.println(nouveauPath);
```

Résultat :

```
C:\Users\jm\AppData\Local\Temp\monfichier.txt
C:\Temp
```

La méthode Path.relativize() permet d'obtenir le chemin relatif à celui encapsulé dans l'instance de type Path. Ceci permet de définir le chemin relatif entre deux Path du système de fichiers.

La méthode relativize() effectue l'inverse de la méthode resolve() : elle ajoute au besoin dans le chemin qu'elle renvoie des éléments ./ ou ../

Exemple (code Java 7) :

```
    Path path1 = Paths.get("C:/Users/jm");
    Path path2 = Paths.get("C:/Users/test");
    Path path1VersPath2 = path1.relativize(path2);
    System.out.println(path1VersPath2);
    Path path2VersPath1 = path2.relativize(path1);
    System.out.println(path2VersPath1);
```

Résultat :

```
..\test
..\jm
```

Dans cet exemple, les deux chemins ont le même répertoire père : le résultat de l'invocation de la méthode relativize() renvoie simplement un chemin qui remonte au répertoire père et descend au répertoire cible.

Exemple (code Java 7) :

```
    Path path1 = Paths.get("C:/");
    Path path2 = Paths.get("C:/Users/test");
    Path path1VersPath2 = path1.relativize(path2);
    System.out.println(path1VersPath2);
    Path path2VersPath1 = path2.relativize(path1);
    System.out.println(path2VersPath1);
```

Résultat :

```
Users\test
..\..
```

Une exception est levée si un chemin relatif et un chemin absolu sont utilisés lors de l'invocation de la méthode relativize().

Exemple (code Java 7) :

```
    Path path1 = Paths.get("test");
    Path path2 = Paths.get("C:/Users/test");
    Path path1VersPath2 = path1.relativize(path2);
    System.out.println(path1VersPath2);
    Path path2VersPath1 = path2.relativize(path1);
    System.out.println(path2VersPath1);
```

Résultat :

```
Exception in thread "main" java.lang.IllegalArgumentException: 'other' is different type
of Path
      at sun.nio.fs.WindowsPath.relativize(Unknown Source)
      at sun.nio.fs.WindowsPath.relativize(Unknown Source)
      at fr.jmdoudoux.dej.nio2.TestNIO2.testRelativize3(TestNIO2.java:33)
      at fr.jmdoudoux.dej.nio2.TestNIO2.main(TestNIO2.java:9)
```

16.3.4. La comparaison de chemins

Une instance de type Path redéfinit la méthode equals() pour permettre de tester l'égalité de l'instance avec une autre instance.

L'interface Path hérite de l'interface Comparable, ce qui permet de trier des objets de type Path.

L'interface Path propose également des méthodes permettant de comparer le début ou la fin de deux chemins

Méthode	Rôle
int compareTo(Path other)	Comparer le chemin avec celui fourni en paramètre
boolean endsWith(Path other)	Comparer la fin du chemin avec celui fourni en paramètre
boolean endsWith(String other)	Comparer la fin du chemin avec celui fourni en paramètre

boolean startsWith(Path other)	Comparer le début du chemin avec celui fourni en paramètre
boolean startsWith(String other)	Comparer le début du chemin avec celui fourni en paramètre

Attention : une instance de type Path est dépendante du système de fichiers : il n'est donc pas possible de comparer deux instances de type Path associées à deux systèmes de fichiers différents.

L'interface Path propose les méthodes startsWith() et endsWith() qui permettent respectivement de tester si le chemin commence ou se termine par la chaîne de caractères fournie en paramètre.

Exemple (code Java 7) :

```
    Path path1 = Paths.get("C:/Users/jm");
    Path path2 = Paths.get("C:/");

    System.out.println(path1.startsWith("C:/"));
    System.out.println(path1.startsWith("C:/Users"));
    System.out.println(path1.startsWith(path2));
    System.out.println(path1.startsWith("C:"));
    System.out.println(path1.startsWith("Users"));
    System.out.println(path1.startsWith("/Users"));
```

Résultat :

```
true
true
true
false
false
false
```

16.3.5. La conversion d'un chemin

Les chemins encapsulés dans une instance de type Path ne sont pas toujours complets ou linéaires : par exemple un chemin relatif ne possède pas de racine ou un chemin peut contenir un lien symbolique qui fera dévier le cheminement lors de l'accès à la ressource encapsulée par le chemin.

L'interface Path propose donc plusieurs méthodes pour convertir un chemin.

Méthode	Rôle
Path toAbsolutePath()	Retourner le chemin absolu du chemin
Path toRealPath(LinkOption...)	Retourner le chemin physique du Path notamment en résolvant les liens symboliques selon les options fournies. Peut lever une exception si le fichier n'existe pas ou s'il ne peut pas être accédé
URI toUri()	Retourner le chemin sous la forme d'une URI

La méthode Path.toAbsolutePath() permet d'obtenir le chemin absolu du chemin encapsulé dans l'instance de type Path.

La méthode toRealPath() renvoie un chemin dans lequel les liens symboliques du chemin fourni en paramètre ont été résolus par rapport au système de fichiers.

Exemple (code Java 7) :

```
    path = Paths.get("C:/Users/jm/AppData/Local/Temp/monfichier.txt");
    System.out.println("toUri()          = " + path.toUri());
    path = Paths.get("src/monfichier.txt");
    System.out.println("toAbsolutePath() = " + path.toAbsolutePath());
    try {
      System.out.println("toRealPath()     = " + path.toRealPath(LinkOption.NOFOLLOW_LINKS));
    } catch (IOException ex) {
      ex.printStackTrace();
    }
```

Résultat :

```
toUri()           = file:///C:/Users/jm/AppData/Local/Temp/monfichier.txt
toAbsolutePath() = C:\Users\jm\Documents\NetBeansProjects\JavaApplication1\src\monfichier.txt
toRealPath()      = C:\Users\jm\Documents\NetBeansProjects\JavaApplication1\src\monfichier.txt
```

16.4. Glob

Un glob est un pattern qui est appliqué sur des noms de fichiers ou de répertoires : c'est une version simplifiée des expressions régulières adaptée aux noms d'éléments d'un système de fichiers.

Plusieurs méthodes de la classe Files attendent un glob en paramètre.

L'interface PathMatcher définit une méthode pour des objets dont le but est de réaliser des comparaisons sur des chemins.

Méthode	Rôle
Boolean matches(Path path)	Renvoie une booléen qui précise si le chemin correspond au pattern

Pour obtenir une instance de type PathMatcher, il faut invoquer la méthode getPathMatcher() de la classe FileSystem qui attend en paramètre une chaîne de caractères précisant la syntaxe et le pattern.

Exemple (code Java 7) :

```
      PathMatcher matcher = FileSystems.getDefault().getPathMatcher("glob:*.java");
      if (matcher.matches(path)) {
        System.out.println(path);
      }
```

La définition d'un glob utilise une syntaxe qui lui est propre :

Motif	Rôle
*	Aucun ou plusieurs caractères
**	Aucun ou plusieurs sous-répertoires
?	Un caractère quelconque
{}	Un ensemble de motifs exemple : {htm, html}
[]	Un ensemble de caractères. Exemple : [A-Z] : toutes les lettres majuscules [0-9] : tous les chiffres [a-z,A-Z] : toutes les lettres indépendamment de la casse Chaque élément de l'ensemble est séparé par un caractère virgule Le caractère - permet de définir une plage de caractères A l'intérieur des crochets, les caractères *, ? et / ne sont pas interprétés
\	

	Il permet d'échapper des caractères pour éviter qu'ils ne soient interprétés. Il sert notamment à échapper le caractère \ lui-même
Les autres caractères	Ils se représentent eux-mêmes sans être interprétés

Exemples :

Glob	Explication
*.html	tous les fichiers ayant l'extension .html
???	trois caractères quelconques
[0-9]	tous les fichiers qui contiennent au moins un chiffre
*.{htm, html}	tous les fichiers dont l'extension est htm ou html
I*.java	tous les fichiers dont le nom commence par un i majuscule et possède une extension .java

Chaque implémentation de type FileSystem permet d'obtenir une instance de type PathMatcher en utilisant la méthode getPathMatcher() qui attend en paramètre un objet de type String contenant la syntaxe et le motif.

Exemple (code Java 7) :

```
      String pattern = "glob:*.{text}";
      PathMatcher matcher = FileSystems.getDefault().getPathMatcher("glob:" + pattern);
```

Le paramètre contient la syntaxe du motif suivi du caractère deux-points et du motif qui sera utilisé pour vérifier la correspondance. Dans l'exemple ci-dessus, la syntaxe utilisée est de type glob.

La syntaxe glob est simple mais il est aussi possible d'utiliser une expression régulière en précisant la syntaxe regex.

Une implémentation peut proposer le support d'autres syntaxes. Il est aussi possible de définir sa propre implémentation de l'interface PathMatcher.

L'interface PathMatcher ne possède qu'une seule méthode nommée matches() qui attend en paramètre un objet de type Path et renvoie un booléen.

Exemple (code Java 7) :

```
      PathMatcher matcher = FileSystems.getDefault().getPathMatcher("glob:*.{java,class}");
      Path filename = ...;
      if (matcher.matches(filename)) {
          System.out.println(filename);
      }
```

Il faut être vigilent lors de la définition du motif utilisé par le glob car le motif s'applique sur l'ensemble du chemin.

Exemple (code Java 7) :

```
  public static void testGlob() throws IOException {
    final Path file1 = Paths.get("C:/java/test/test.java");
    final Path file2 = Paths.get("C:/java/test/test.txt");
    final Path file3 = file1.getFileName();

    String pattern = "glob:**/*.{java,class}";
    System.out.println("Pattern " + pattern);

    PathMatcher matcher = FileSystems.getDefault().getPathMatcher(pattern);
    System.out.println(file1 + " " + matcher.matches(file1));
    System.out.format("%-22s %b\n", file2, matcher.matches(file2));
    System.out.format("%-22s %b\n", file3, matcher.matches(file3));
    System.out.println("");

    pattern = "glob:*.java";
    System.out.println("Pattern " + pattern);
    matcher = FileSystems.getDefault().getPathMatcher(pattern);
    System.out.println(file1 + " " + matcher.matches(file1));
    System.out.format("%-22s %b\n", file3, matcher.matches(file3));
  }
```

Résultat :

```
Pattern glob:**/*.{java,class}
C:\java\test\test.java true
C:\java\test\test.txt  false
test.java              false
Pattern glob:*.java
C:\java\test\test.java false
test.java              true
```

16.5. La classe Files

La classe java.nio.file.Files est un helper qui contient une cinquantaine de méthodes statiques permettant de réaliser des opérations sur des fichiers ou des répertoires dont le chemin est encapsulé dans un objet de type Path.

La classe Files permet de réaliser des opérations de base sur les fichiers et les répertoires : création, ouverture, suppression, test d'existence, changement des permissions, ...

Ces méthodes concernent notamment :

- La création d'éléments : createDirectory(), createFile(), createLink(), createSymbolicLink(), createTempFile(), createTempDirectory(), ...
- La manipulation d'éléments : delete(), move(), copy(), ...
- L'obtention du type d'un élément : isRegularFile(), isDirectory(), probeContentType(), ...
- L'obtention de métadonnées et la gestion des permissions : getAttributes(), getPosixFilePermissions(), isReadable(), isWriteable(), size(), getFileAttributeView(), ...

NIO 2 propose une API qui facilite la manipulation des éléments du système de fichiers pour par exemple créer, supprimer, déplacer, renommer ou copier un fichier. La manipulation des fichiers et des répertoires est assurée par la classe java.nio.file.Files..

Les méthodes de la classe Files attendent généralement en paramètre au moins une instance de type Path. Certaines méthodes de la classe Files effectuent des opérations atomiques qui doivent être réalisées dans leur entièreté ou pas du tout : elles réussissent ou échouent.

16.5.1. Les vérifications sur un fichier ou un répertoire

La classe Files propose deux méthodes pour vérifier l'existence d'un élément dans le système de fichier :

Méthode	Rôle
boolean exists(Path)	vérifier l'existence sur le système de fichiers de l'élément dont le chemin est encapsulé dans le paramètre de type Path fourni
boolean notExists(Path)	vérifier que l'élément dont le chemin est encapsulé dans l'instance de type Path fournie en paramètre n'existe pas sur le système de fichiers

Lors d'un test d'existence d'une instance de type Path, le résultat peut avoir plusieurs valeurs :

- L'existence de l'élément est vérifiée
- L'inexistence de l'élément est vérifiée

- La vérification n'a pas pu être réalisée car le statut de l'élément est inconnu : c'est par exemple le cas si l'élément n'est pas accessible

La vérification n'a pas pu être réalisée si les méthodes exists() et notExists() pour une même instance de type Path renvoient toutes les deux false.

Attention : !Files.exists(path) n'est donc pas équivalent à Files.notExists(path)

La classe Files propose plusieurs méthodes pour vérifier les droits d'accès ou le type d'un élément de type Path :

Méthode	Rôle
boolean isReadable(Path path)	Retourner true si le fichier peut être lu
boolean isWritable(Path path)	Retourner true si le fichier peut être modifié
boolean isHidden(Path path)	Retourner true si le fichier est caché
boolean isExecutable(Path path)	Retourner true si le fichier est exécutable
boolean isRegularFile(Path path)	Retourner true si l'objet encapsulé dans le Path est un fichier
boolean isDirectory(Path path)	Retourner true si l'objet encapsulé dans le Path est un répertoire
boolean isSymbolicLink(Path path)	Retourner true si l'objet encapsulé dans le Path est un lien symbolique

Exemple (code Java 7) :

```
  public static void testAttributs() throws IOException {
    Path monFichier = Paths.get("C:/java/temp/monfichier.txt");
    boolean estLisible = Files.isRegularFile(monFichier) &
                         Files.isReadable(monFichier);
    System.out.println(monFichier + " est lisible : "+estLisible);
  }
```

Résultat :

```
C:\java\temp\monfichier.txt est lisible : true
```

La classe Files propose aussi plusieurs méthodes pour faire d'autres vérifications sur des éléments de type Path.

Méthode	Rôle
isSamePath(Path, Path)	Comparer les deux instances de Path pour déterminer si elles correspondent aux mêmes éléments dans le système de fichiers. Ceci est pratique si l'un des deux Path est un lien symbolique.

Exemple (code Java 7) :

```
  public static void sontIdentiques(String cheminCible, String cheminLien)
      throws IOException {
    Path lien = Paths.get(cheminLien);
    Path cible = Paths.get(cheminCible);
    if (Files.isSameFile(lien, cible)) {
      System.out.println("Fichiers identiques");
    } else {
      System.out.println("Fichiers différents");
    }
  }
```

16.5.2. La création d'un fichier ou d'un répertoire

L'API permet la création de fichiers, de répertoires permanents ou temporaires en utilisant plusieurs méthodes de la classe File :

Méthode	Rôle
Path createFile(Path path, FileAttribute<?>... attrs)	Créer un fichier dont le chemin est encapsulé par l'instance de type Path fournie en paramètre
Path createDirectory(Path dir, FileAttribute<?>... attrs)	Créer un répertoire dont le chemin est encapsulé par l'instance de type Path fournie en paramètre
Path createDirectories(Path dir, FileAttribute<?>... attrs)	Créer dans le répertoire dont le chemin est fourni en paramètre un sous-répertoire avec les attributs fournis
Path createTempDirectory(Path dir, String prefix, FileAttribute<?>... attrs)	Créer dans le répertoire dont le chemin est fourni en paramètre un sous-répertoire temporaire dont le nom utilisera le préfixe fourni
Path createTempDirectory(String prefix, FileAttribute<?>... attrs)	Créer dans le répertoire temporaire par défaut du système, un sous-répertoire temporaire dont le nom utilisera la préfixe fourni
Path createTempFile(Path dir, String prefix, String suffix, FileAttribute<?>... attrs)	Créer dans le répertoire dont le chemin est fourni en paramètre un fichier temporaire dont le nom utilisera le préfixe fourni
Path createTempFile(String prefix, String suffix, FileAttribute<?>... attrs)	Créer dans le répertoire temporaire par défaut du système un fichier temporaire dont le nom utilisera le préfixe et le suffixe fournis

La méthode Files.createFile() permet de créer un fichier dont le chemin est encapsulé dans son paramètre de type Path.

La méthode createFile() attend en paramètres un objet de type Path et un varargs de type FileAttribute< ?> qui permet de préciser les attributs du fichier créé.

Exemple (code Java 7) :

```
Pathfichier=Paths.get("/home/jm/test.txt");
Set<PosixFilePermission> perms=PosixFilePermissions.fromString("rw-rw-rw-");
FileAttribute<Set<PosixFilePermission>>attr=PosixFilePermissions.asFileAttribute(perms);
Files.createFile(fichier,attr);
```

Si le chemin est uniquement fourni en paramètre de la méthode createFile(), le fichier est créé avec les attributs par défaut du système.

Exemple (code Java 7) :

```
Path monFichier = Paths.get("C:/temp/monfichier.txt");
Path file = Files.createFile(monFichier);
```

Par défaut, une exception de type FileAlreadyExistsException est levée si le fichier à créer existe déjà.

La méthode createTempFile() permet de créer un fichier temporaire.

Elle possède deux surcharges :

Méthode	Rôle
createTempFile(Path dir, String prefix, String suffix, FileAttribute< ?>... attrs)	Créer un fichier temporaire dans le répertoire dont le chemin est fourni en paramètre
createTempFile(String prefix, String suffix, FileAttribute< ?>... attrs)	Créer un fichier temporaire dans le répertoire par défaut du système

Les deux surcharges attendent en paramètres un préfixe et un suffixe qui seront utilisés pour déterminer le nom du fichier et les attributs à utiliser lors de la création du fichier. Le préfixe et le suffixe peuvent être null : s'ils sont fournis, ils seront utilisés par l'implémentation de manière spécifique pour déterminer le nom du fichier. Le format du nom du fichier créé est dépendant de la plate-forme.

Exemple (code Java 7) :

```
public static void testCreateTempFile() throws IOException {
  Path tempFile = Files.createTempFile("monapp_", ".tmp");
  System.out.format("Fichier créé : %s%n", tempFile);
}
```

Résultat :

```
Fichier créé : C:\DOCUME~1\jm\LOCALS~1\Temp\monapp_242180026059597956.tmp
```

La méthode createDirectory() permet de créer un répertoire : elle attend en paramètre un objet de type Path qui encapsule le chemin ou le sous-chemin du répertoire et un varargs de type FileAttribute< ?> qui permet de préciser les attributs du nouveau répertoire.

Si aucun attribut n'est fourni en paramètre, alors le répertoire est créé avec les attributs par défaut du système.

Exemple (code Java 7) :

```
public static void testCreateDirectory() throws IOException {
  Path monRepertoire = Paths.get("C:/temp/mon_repertoire");
  Path file = Files.createDirectory(monRepertoire);
}
```

Si le répertoire à créer existe déjà alors une exception de type FileAlreadyExistsException est levée.

La méthode createDirectory() ne permet que de créer un seul sous-répertoire : le chemin ou le sous-chemin fourni ne doit donc correspondre qu'à un nouveau sous-répertoire à créer dans un répertoire existant. Dans le cas contraire, une exception de type NoSuchFileException est levée.

Exemple (code Java 7) :

```
public static void testCreateDirectory() throws IOException {
  Path monRepertoire = Paths.get("C:/temp/niveau1/niveau2/mon_repertoire");
  Path file = Files.createDirectory(monRepertoire);
}
```

Résultat :

```
java.nio.file.NoSuchFileException:
C:\temp\niveau1\niveau2\mon_repertoire
      at sun.nio.fs.WindowsException.translateToIOException(Unknown Source)
      at sun.nio.fs.WindowsException.rethrowAsIOException(Unknown Source)
      at sun.nio.fs.WindowsException.rethrowAsIOException(Unknown Source)
      at sun.nio.fs.WindowsFileSystemProvider.createDirectory(Unknown Source)
      at java.nio.file.Files.createDirectory(Unknown Source)
      at fr.jmdoudoux.dej.nio2.TestNIO2.testCreateDirectory(TestNIO2.java:199)
      at fr.jmdoudoux.dej.nio2.TestNIO2.main(TestNIO2.java:28)
```

Pour créer toute l'arborescence fournie dans le chemin, incluant la création d'un ou plusieurs sous-répertoires manquants dans l'arborescence, il faut utiliser la méthode createDirectories().

Exemple (code Java 7) :

```
public static void testCreateDirectories() throws IOException {
  Path monRepertoire = Paths.get("C:/temp/niveau1/niveau2/mon_repertoire");
  Path file = Files.createDirectories(monRepertoire);
}
```

Pour créer un répertoire temporaire, il faut utiliser la méthode createTempDirectory() qui possède deux surcharges :

- createTempDirectory(Path dir, String prefix, FileAttribute<?>... attrs)
- createTempDirectory(String prefix, FileAttribute<?>... attrs)

La surcharge qui attend en paramètre un objet de type Path permet de préciser le sous-répertoire dans lequel le répertoire temporaire va être créé. La seconde surcharge crée le sous-répertoire temporaire dans le répertoire temporaire par défaut du système d'exploitation.

Le paramètres varargs de type FileAttributs<?> permet de préciser les attributs qui seront associés au nouveau répertoire. Si aucun attribut n'est précisé alors ce sont les attributs par défaut du système qui seront utilisés.

Le paramètre prefix, qui peut être null, sera utilisé de manière dépendante de l'implémentation pour construire le nom du répertoire.

Exemple (code Java 7) :

```
public static void testCreateTempDirectory() throws IOException {
  Path repertoireTemp = Files.createTempDirectory(null);
  System.out.println(repertoireTemp);
  repertoireTemp = Files.createTempDirectory("monApp_");
  System.out.println(repertoireTemp);
}
```

Résultat :

```
C:\DOCUME~1\jm\LOCALS~1\Temp\2626334559178550265
C:\DOCUME~1\jm\LOCALS~1\Temp\monApp_404075526480225045
```

16.5.3. La copie d'un fichier ou d'un répertoire

Ecrire sa propre méthode pour une fonctionnalité aussi basique que la copie d'un fichier ne présente pas beaucoup d'intérêt. Il est préférable d'utiliser une bibliothèque tierce comme Apache Commons IO ou Google Guava car cette fonctionnalité n'est pas proposée par l'API Java Core avant Java 7.

La classe Files propose plusieurs surcharges de la méthode copy() pour copier un fichier ou un répertoire.

Méthode	Rôle
Path copy(Path source, Path target, CopyOption... options)	Copier un élément avec les options précisées
long copy(InputStream in, Path target, CopyOption... options)	Copier tous les octets d'un flux de type InputStream vers un fichier
long copy(Path source, OutputStream out)	Copier tous les octets d'un fichier dans un flux de type OutputStream

La méthode Files.copy() permet de copier un fichier dont les chemins source et cible sont encapsulés dans ses deux paramètres de type Path.

Exemple (code Java 7) :

```
Path monFichier =
Paths.get("C:\\temp\\monfichier.txt");
Path monFichierCopie = Paths.get("C:\\temp\\monfichier - copie.txt");
Path file = Files.copy(monFichier, monFichierCopie);
```

Une surcharge de la méthode copy() permet de préciser les options de copie du fichier en utilisant son troisième paramètre qui est un varargs de type CopyOption.

Plusieurs valeurs des énumérations StandardCopyOption et LinkOption qui implémentent l'interface CopyOption peuvent être utilisées avec la méthode copy() :

Valeur	Rôle
StandardCopyOption.COPY_ATTRIBUTES	La copie se fait en conservant les attributs du fichier : ceux-ci sont dépendants du système sous-jacent
StandardCopyOption.REPLACE_EXISTING	Remplacer le fichier cible s'il existe. Si le chemin cible est un répertoire non vide, une exception de type FileAlreadyExistsException est levée
LinkOption.NOFOLLOW_LINKS	Ne pas suivre les liens symboliques. Si le chemin à copier est un lien symbolique, c'est le lien lui-même qui est copié

Exemple (code Java 7) :

```
import static java.nio.file.StandardCopyOption.*;

// ...

Path monFichier = Paths.get("C:\\temp\\monfichier.txt");
Path monFichierCopie = Paths.get("C:\\temp\\monfichier - copie.txt");
Path file = Files.copy(monFichier, monFichierCopie, REPLACE_EXISTING);
```

Faute d'option indiquée, une exception est levée si le fichier cible existe déjà. La copie échoue si la destination existe sauf si l'option StandardCopyOption.REPLACE_EXISTING est utilisée.

La copie d'un lien symbolique duplique sa cible si l'option LinkOption.NOFOLLOW_LINKS est utilisée : dans ce cas, c'est le lien lui-même qui est copié.

Si l'option StandardCopyOption.ATOMIC_MOVE est utilisée avec la méthode copy(), alors une exception de type UnsupportedOperationException est levée.

Attention : il est possible d'utiliser la méthode copy() sur un répertoire cependant, le répertoire sera créé sans que les fichiers et les sous-répertoires ne le soient : quoi que contienne le répertoire, la méthode copy ne créé qu'un répertoire vide. Pour copier le contenu du répertoire, il faut parcourir son contenu et copier chacun des éléments un par un.

La méthode copy() possède deux surcharges qui permettent d'utiliser respectivement un objet de type InputStream comme source et un objet de type OutputStream comme cible.

Exemple (code Java 7) :

```
public static void copierFichier2() throws IOException {
  Path cible = Paths.get("c:/java/test/monfichier_copie.txt");
  URI uri = new File("c:/java/test/monfichier.txt").toURI();
  try (InputStream in = uri.toURL().openStream()) {
    Files.copy(in, cible);
  }
}
```

16.5.4. Le déplacement d'un fichier ou d'un répertoire

Avant Java 7, la méthode rename() de la classe java.io.File ne fonctionnait pas sur tous les systèmes d'exploitation et généralement pas au travers du réseau. Bien que peu performante, la solution la plus sûre était de copier chaque octet du fichier source puis de supprimer ce fichier.

La méthode Files.move() permet de déplacer ou de renommer un fichier dont les chemins source et cible sont encapsulés dans ses deux paramètres de type Path.

Méthode	Rôle
move(Path source, Path target, CopyOption... options)	Déplacer ou renommer un élément avec les options précisées

Exemple (code Java 7) :

```
Path monFichier = Paths.get("C:\\temp\\monfichier.txt");
Path monFichierCopie = Paths.get("C:\\temp\\monfichier.old");
Path file = Files.move(monFichier, monFichierCopie);
```

Les options de déplacement du fichier peuvent être précisées en utilisant son troisième paramètre de type CopyOption.

Plusieurs valeurs de l'énumération StandardCopyOption qui implémente l'interface CopyOption peuvent être utilisées avec la méthode move() :

Valeur	Rôle
StandardCopyOption.REPLACE_EXISTING	Remplacement du fichier s'il existe
StandardCopyOption.ATOMIC_MOVE	Assure que le déplacement est réalisé sous la forme d'une opération atomique. Si l'atomicité de l'opération ne peut être garantie alors une exception de type AtomicMoveNotSupportedException est levée

Exemple (code Java 7) :

```
Path monFichier = Paths.get("C:\\temp\\monfichier.txt");
Path monFichierCopie = Paths.get("C:\\temp\\monfichier.old");
Path file = Files.move(monFichier, monFichierCopie, REPLACE_EXISTING, COPY_ATTRIBUTES);
```

Si la méthode move() est invoquée avec l'option StandardCopyOption.COPY_ATTRIBUTES alors une exception de type UnsupportedOperationException est levée.

Par défaut, l'invocation de la méthode move() dont le chemin cible existe déjà lève une exception de type FileAlreadyExistException. Pour écraser le fichier existant, il faut utiliser l'option StandardCopyOption.REPLACE_EXISTING.

Si le chemin source est un lien alors c'est le lien lui-même et non sa cible qui est déplacé.

Si les chemins cible et source fournis en paramètres de la méthode move() sont identiques alors l'invocation de la méthode n'a aucun effet.

Exemple (code Java 7) :

```
  public static void testMove() throws IOException {
    Path source = Paths.get("C:/java/temp/monfichier.txt");
    Path cible = Paths.get("C:/java/temp/monfichier.txt");
    Files.move(source, cible);
  }
```

La méthode move() peut être utilisée sur un répertoire vide ou sur un répertoire non vide dont la cible est sur le même système de fichiers. Dans ce cas le répertoire est simplement renommé et il n'est pas nécessaire de déplacer récursivement le contenu du répertoire.

Exemple (code Java 7) :

```
  public static void testMoveRepertoireVide() throws IOException {
    Path source = Paths.get("C:/java/temp/mon_repertoire");
    Path cible = Paths.get("C:/temp/mon_repertoire_copie");
    Files.move(source, cible);
  }
```

Exemple (code Java 7) :

```
  public static void testRenommerRepertoire() throws IOException {
    Path source = Paths.get("C:/java/temp/mon_repertoire");
    Path cible = source.resolveSibling("mon_repertoire_copie");
    Files.move(source, cible);
  }
```

Si le répertoire cible existe déjà, même vide, alors une exception de type FileAlreadyExistsException est levée. Pour forcer le remplacement, il faut utiliser l'option REPLACE_EXISTING.

Exemple (code Java 7) :

```
  public static void testMoveRepertoireVide() throws IOException {
    Path source = Paths.get("C:/java/temp/mon_repertoire");
    Path cible = Paths.get("C:/java/temp/mon_repertoire_copie");
    Files.move(source, cible, StandardCopyOption.REPLACE_EXISTING);
  }
```

Si le répertoire cible existe et n'est pas vide, alors une exception de type DirectoryNotEmptyException est levée.

Une exception de type AtomicNotSupportedException est levée si le déplacement du répertoire implique deux systèmes de fichiers différents entre la cible et la source et que l'option ATOMIC_MOVE est utilisée.

Exemple (code Java 7) :

```
    // déplacer un fichier dans une autre unité de stockage
    source = Paths.get("c:/temp/cible.txt");
    cible = Paths.get("s:/cible.txt");
    try {
      Files.move(source, cible, ATOMIC_MOVE);
    } catch (final IOException ioe) {
      ioe.printStackTrace();
    }
```

Résultat :

```
java.nio.file.AtomicMoveNotSupportedException:
c:\temp\cible.txt -> s:\cible.txt: Impossible de déplacer le fichier vers un
lecteur de disque différent.
      at sun.nio.fs.WindowsFileCopy.move(WindowsFileCopy.java:296)
      at sun.nio.fs.WindowsFileSystemProvider.move(WindowsFileSystemProvider.java:286)
      at java.nio.file.Files.move(Files.java:1339)
```

Les répertoires vides peuvent être déplacés. Si le répertoire n'est pas vide alors il est possible de le déplacer à condition que son contenu n'est pas besoin de l'être : ceci dépend du système d'exploitation sous-jacent qui peut simplement renommer le répertoire si celui-ci reste sur la même unité de stockage.

Sur la plupart des systèmes, le déplacement d'un répertoire vers une cible sur le même système de stockage se fait simplement en modifiant des entrées dans la table d'allocations des fichiers.

Par contre, le déplacement vers une autre unité de stockage implique forcément le déplacement du contenu du répertoire.

Pour tout autre problème lors de l'invocation de la méthode move(), comme pour toute opération d'entrée/sortie, une erreur peut survenir : dans ce cas, la méthode lève une exception de type IOException.

L'exécution de la méthode move() se fait de manière synchrone et bloquante.

Par défaut, lors de la copie ou du déplacement d'un fichier :

- la copie échoue si le fichier cible existe déjà
- les attributs du fichier peuvent être conservés entièrement, partiellement ou pas du tout
- lors de la copie d'un lien symbolique, c'est la cible du lien qui est copiée et non le lien lui-même

- lors du déplacement d'un lien symbolique, le lien lui-même est déplacé mais le fichier cible n'est pas déplacé
- un répertoire est déplacé seulement s'il est vide ou si le déplacement consiste simplement à le renommer

16.5.5. La suppression d'un fichier ou d'un répertoire

L'API permet la suppression de fichiers, de répertoires ou de liens en utilisant l'une des deux méthodes de la classe Files :

Méthode	Rôle
void delete(Path path)	Supprimer un élément du système de fichiers
boolean deleteIfExist(Path path)	Supprimer un élément du système de fichiers s'il existe

La méthode Files.delete() permet de supprimer un fichier dont le chemin est encapsulé dans son paramètre de type Path. Elle lève une exception si la suppression échoue. Par exemple, une exception de type NoSuchFileException est levée si le fichier à supprimer n'existe pas dans le système de fichiers.

La suppression d'un lien symbolique supprime le lien mais ne supprime pas le fichier cible.

La suppression d'un répertoire échoue si le répertoire n'est pas vide.

Exemple (code Java 7) :

```
Path path = Paths.get("c:/java/test.txt");
try {
  Files.delete(path);
} catch (NoSuchFileException nsfe) {
  System.err.println("Fichier ou repertoire " + path + " n'existe pas");
} catch (DirectoryNotEmptyException dnee) {
  System.err.println("Le repertoire " + path + " n'est pas vide");
} catch (IOException ioe) {
  System.err.println("Impossible de supprimer " + path + " : " + ioe);
}
```

La méthode deleteIfExist() permet de supprimer un élément du système de fichiers sans lever d'exception si celui-ci n'existe pas.

Exemple (code Java 7) :

```
Path path = Paths.get("c:/java/test.txt");
try {
  Files.deleteIfExists(path);
} catch (DirectoryNotEmptyException dnee) {
  System.err.println("Le repertoire " + path + " n'est pas vide");
} catch (IOException ioe) {
  System.err.println("Impossible de supprimer " + path + " : " + ioe);
}
```

16.5.6. L'obtention du type de fichier

NIO2 propose une fonctionnalité pour obtenir le type du contenu d'un fichier en utilisant la méthode probeContentType() de la classe Files

Méthode	Rôle
String probeContentType(Path path)	Retourner le type du contenu du fichier dont le chemin est passé en paramètre

Exemple (code Java 7) :

```
package fr.jmdoudoux.dej.nio2;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class TestNIO2 {

  public static void main(String[] args) {
    try {
      Path source = Paths.get("c:/java/temp/monfichier.txt");
      testProbeContent(source);
      source = Paths.get("c:/java/temp/monfichier.bin");
      testProbeContent(source);
      source = Paths.get("c:/java/temp/monfichier");
      testProbeContent(source);
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  public static void testProbeContent(Path fichier) throws IOException {
    String type = Files.probeContentType(fichier);
    if (type == null) {
      System.out.println("Impossible de déteminer le type du fichier :"
         + fichier);
    } else {
      System.out.println("le fichier " + fichier + " est du type : " + type);
    }
  }
}
```

Résultat :

```
le fichier c:\java\temp\monfichier.txt
est du type : text/plain
Impossible de déteminer le type du
fichier : c:\java\temp\monfichier.bin
Impossible de déteminer le type du
fichier : c:\java\temp\monfichier
```

La méthode probeContentType() renvoie null si le type de contenu ne peut pas être déterminé.

Si le type a pu être déterminé, il est renvoyé sous la forme d'une chaîne de caractères dont le contenu respecte la norme MIME (Multipurpose Internet Mail Extensions) définit par la RFC 2045.

L'implémentation de cette méthode est dépendante de la plate-forme : sa fiabilité n'est donc pas garantie.

Il est possible de fournir une implémentation du type FileTypeDetector pour déterminer le type du contenu d'un fichier.

Si aucune implémentation de type FileTypeDetector ne peut déterminer le type, alors la méthode probeContentType() va demander au système de déterminer le type du contenu.

Pour définir sa propre implémentation, il faut créer une classe qui hérite de la classe abstraite FileTypeDetector et redéfinir sa méthode abstraite probeContentType() qui attend en paramètre un objet de type Path et renvoie une chaîne de caractères.

L'implémentation doit avoir un constructeur sans argument.

L'enregistrement de FileTypeDetector doit se faire un utilisant le service Provider de la JVM : le nom pleinement qualifié de la classe doit être dans un fichier java.nio.file.spi.FileTypeDetector contenu dans le sous-répertoire META-INF/services.

La détermination du type du contenu est généralement spécifique au système d'exploitation sous-jacent : utilisation de l'extension, de métadonnées dans un fichier associé ou lecture de tout ou partie du contenu du fichier.

16.6. Le parcours du contenu de répertoires

Les solutions proposées par NIO2 pour le parcours du contenu d'un répertoire remplacent avantageusement les méthodes list() et listfiles() de la classe java.io.File. Ces méthodes offraient de piètres performances notamment avec des répertoires contenant de nombreux fichiers et consommaient beaucoup de ressources.

NIO2 propose plusieurs solutions pour parcourir le contenu d'un répertoire : elles sont plus complexes à mettre en oeuvre par rapport à la classe java.io.File mais sont aussi beaucoup plus performantes surtout avec des répertoires qui contiennent de nombreux fichiers.

16.6.1. Le parcours d'un répertoire

Il est possible d'utiliser une instance de l'interface java.nio.file.DirectoryStream qui permet de parcourir un répertoire en réalisant une itération sur les éléments qu'il contient.

La méthode newDirectoryStream() de la classe Files attend en paramètre un objet de type Path qui correspond au répertoire à parcourir et permet d'obtenir une instance de type DirectoryStream<Path>.

La méthode iterator() retourne une instance d'un itérateur sur les éléments du répertoire : fichiers, liens, sous-répertoires, ...

L'itération sur les éléments permet de meilleures performances et une consommation réduite en ressources pour obtenir les mêmes résultats que l'invocation des méthodes list() et listFiles() de la classe java.io.File.

Attention : il est très important d'invoquer la méthode close() de l'instance de type DirectoryStream pour libérer les ressources utilisées.

Exemple (code Java 7) :

```
  public static void testDirectoryStream() throws IOException {
    Path jdkPath = Paths.get("C:/Program Files/Java/jdk1.7.0_02");
    DirectoryStream<Path> stream = Files.newDirectoryStream(jdkPath);
    try {
      Iterator<Path> iterator = stream.iterator();
      while(iterator.hasNext()) {
        Path p = iterator.next();
        System.out.println(p);
      }
    } finally {
      stream.close();
    }
  }
```

Résultat :

```
C:\Program Files\Java\jdk1.7.0\bin
C:\Program Files\Java\jdk1.7.0\COPYRIGHT
C:\Program Files\Java\jdk1.7.0\db
C:\Program Files\Java\jdk1.7.0\demo
C:\Program Files\Java\jdk1.7.0\include
C:\Program Files\Java\jdk1.7.0\jre
C:\Program Files\Java\jdk1.7.0\lib
C:\Program Files\Java\jdk1.7.0\LICENSE
C:\Program Files\Java\jdk1.7.0\README.html
C:\Program Files\Java\jdk1.7.0\register.html
C:\Program Files\Java\jdk1.7.0\register_ja.html
C:\Program Files\Java\jdk1.7.0\register_zh_CN.html
C:\Program Files\Java\jdk1.7.0\release
C:\Program Files\Java\jdk1.7.0\sample
C:\Program Files\Java\jdk1.7.0\src.zip
C:\Program Files\Java\jdk1.7.0\THIRDPARTYLICENSEREADME.txt
```

L'ordre dans lequel les éléments sont fournis lors de l'itération n'est pas garanti. Des éléments spécifiques à certains systèmes ne sont pas retournés dans l'itération : c'est par exemple le cas des éléments « . » (le répertoire courant) et « .. » (le répertoire parent) sur un système de type Unix.

Attention : l'implémentation de l'interface Iterable de l'instance de type DirectoryStream ne propose pas le support de la méthode remove() et son invocation lève une exception de type UnsupportedOperationException.

Exemple (code Java 7) :

```
  public static void testDirectoryStream() throws IOException {
    Path jdkPath = Paths.get("C:/Program Files/Java/jdk1.7.0_02");
    DirectoryStream<Path> stream = Files.newDirectoryStream(jdkPath);
    try {
      Iterator<Path> iterator = stream.iterator();
      while(iterator.hasNext()) {
        Path p = iterator.next();
        System.out.println(p);
        Iterator.remove();
      }
    } finally {
      stream.close();
    }
  }
```

Résultat :

```
C:\Program Files\Java\jdk1.7.0\bin
Exception in thread "main"
java.lang.UnsupportedOperationException
      at sun.nio.fs.WindowsDirectoryStream$WindowsDirectoryIterator.remove(Unknown
Source)
      at fr.jmdoudoux.dej.nio2.TestNIO2.testDirectoryStream(TestNIO2.java:138)
      at fr.jmdoudoux.dej.nio2.TestNIO2.main(TestNIO2.java:25)
```

L'interface DirectoryStream hérite des interfaces Closeable et Iterable. Il est donc pratique de déclarer l'instance de type DirectoryStream<Path> dans une instruction try avec ressources qui se chargera d'invoquer automatiquement sa méthode close(). Le parcours des éléments peut se faire dans une instruction for.

Exemple (code Java 7) :

```
  public static void utilisationDirectoryStream() throws IOException {
    Path jdkPath = Paths.get("C:/Program Files/Java/jdk1.7.0");
    try (DirectoryStream<Path> stream = Files.newDirectoryStream(jdkPath)) {
      for (Path entry : stream) {
        System.out.println(entry);
      }
    }
  }
```

Si une exception est levée durant l'itération, alors elle est encapsulée dans une exception unchecked de type DirectoryIteratorException.

Exemple (code Java 7) :

```
  public static void testDirectoryStream3() {
    Path jdkPath = Paths.get("C:/Program Files/Java/jdk1.7.0_02");
    try (DirectoryStream<Path> stream = Files.newDirectoryStream(jdkPath)) {
      for (Path entry : stream) {
        System.out.println(entry);
      }
    } catch (IOException | DirectoryIteratorException e) {
      e.printStackTrace();
    }
  }
```

Il est aussi possible de fournir un paramètre qui est une chaîne de caractères au format glob pour filtrer la liste des éléments retournés en fonction de leurs noms.

Exemple (code Java 7) :

```
  public static void utilisationDirectoryStream() throws IOException {
    Path jdkPath = Paths.get("C:/Program Files/Java/jdk1.7.0");
    try (DirectoryStream<Path> stream = Files.newDirectoryStream(jdkPath,"*.{zip,html}")) {
      for (Path entry : stream) {
        System.out.println(entry);
      }
    }
  }
```

Résultat :

```
C:\Program Files\Java\jdk1.7.0\README.html
C:\Program Files\Java\jdk1.7.0\register.html
C:\Program Files\Java\jdk1.7.0\register_ja.html
C:\Program Files\Java\jdk1.7.0\register_zh_CN.html
C:\Program Files\Java\jdk1.7.0\src.zip
```

Attention : il n'est possible de n'obtenir qu'un seul itérateur d'une même instance de type DirectoryStream. Une seconde invocation de la méthode iterator() lève une exception de type IllegalStateException.

Exemple (code Java 7) :

```
  public static void utilisationDirectoryStream() throws IOException {
    Path jdkPath = Paths.get("C:/Program Files/Java/jdk1.7.0_02");
    try (DirectoryStream<Path> stream = Files.newDirectoryStream(jdkPath)) {
      for (Path entry : stream) {
        System.out.println(entry);
        Iterator<Path> secondIterator = stream.iterator();
      }
    }
  }
```

Résultat :

```
Exception in thread "main"
java.lang.IllegalStateException: Iterator already obtained
      at sun.nio.fs.WindowsDirectoryStream.iterator(Unknown Source)
      at fr.jmdoudoux.dej.nio2.TestNIO2.utilisationDirectoryStream(TestNIO2.java:134)
      at fr.jmdoudoux.dej.nio2.TestNIO2.main(TestNIO2.java:24)
```

Il est possible de définir un filtre qui sera appliqué sur chacun des éléments du répertoire pour déterminer s'il doit être retourné ou non lors du parcours.

Pour cela, il faut créer une instance de type DirectoryStream.Filter<Path> et la fournir en paramètre à la méthode newDirectoryStream(). Le code du filtre doit se trouver dans la méthode accept() qui prend en paramètre un objet de type Path et renvoie un boolean qui est le résultat de l'application du filtre.

Exemple (code Java 7) :

```
  public static void utilisationDirectoryStreamAvecFiltre() throws IOException {
    Path jdkPath = Paths.get("C:/Program Files/Java/jdk1.7.0_02");
    DirectoryStream.Filter<Path> filtre = new DirectoryStream.Filter<Path>() {
      public static final long HUIT_MEGABYTES = 8*1024*1024;

      @Override
      public boolean accept(Path element) throws IOException {
        return Files.size(element) >= HUIT_MEGABYTES;
      }
    };

    try (DirectoryStream<Path> stream = Files.newDirectoryStream(jdkPath, filtre)) {
      for (Path entry : stream) {
        System.out.println(entry);
      }
    }
  }
```

Résultat :

```
  C:\Program Files\Java\jdk1.7.0_02\src.zip
```

16.6.2. Le parcours d'une hiérarchie de répertoires

La méthode Files.walkFileTree() permet de parcourir la hiérarchie d'un ensemble de répertoires en utilisant le motif de conception visiteur. Ce type de parcours peut être utilisé pour rechercher, copier, déplacer, supprimer, ... des éléments de la hiérarchie parcourue.

Il faut écrire une classe qui implémente l'interface java.nio.file.FileVisitor<T>. Cette interface définit des méthodes qui seront des callbacks lors du parcours de la hiérarchie.

Méthode	Rôle
FileVisitResult postVisitDirectory(T dir, IOException exc)	Le parcours sort d'un répertoire qui vient d'être parcouru ou une exception est survenue durant le parcours
FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)	Le parcours rencontre un répertoire, cette méthode est invoquée avant de parcourir son contenu
FileVisitResult visitFile(T file, BasicFileAttributes attrs)	Le parcours rencontre un fichier
FileVisitResult visitFileFailed(T file, IOException exc)	La visite d'un des fichiers durant le parcours n'est pas possible et une exception a été levée

Il est possible de contrôler les traitements du parcours en utilisant les objets de type FileVisitResult retournés par les méthodes de l'interface FileVisitor.

Les méthodes de l'interface FileVisitor renvoient toutes une valeur qui appartient à l'énumération FileVisitResult. Cette valeur permet de contrôler le processus de parcours de l'arborescence :

- CONTINUE : poursuite du parcours
- TERMINATE : arrêt immédiat du parcours
- SKIP_SUBTREE : inhibe le parcours de la sous-arborescence. Si la méthode preVisitDirectory() renvoie cette valeur, le parcours du répertoire est ignoré
- SKIP_SIBLING : inhibe le parcours des répertoires frères. Si la méthode preVisitDirectory() renvoie cette valeur alors le répertoire n'est pas parcouru et la méthode postVisitDirectory() n'est pas invoquée. Si la méthode postVisitDirectory() renvoie cette valeur, alors les autres répertoires frères qui n'ont pas encore été parcourus sont ignorés

Exemple (code Java 7) :

```
public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs) {
  if (dir.getFileName().toString().equals("target")) {
    return SKIP_SUBTREE;
  }
  return CONTINUE;
}
```

L'exemple ci-dessous parcourt l'arborescence et s'arrête dès que le fichier test.txt est trouvé.

Exemple (code Java 7) :

```
public FileVisitResult visitFile(Path file, BasicFileAttributes attr) {
  if (file.getFileName().equals("test.txt")) {
    System.out.println("Fichier trouve");
    return TERMINATE;
  }
  return CONTINUE;
}
```

L'API propose la classe java.nio.file.SimpleFileVisitor qui est une implémentation de l'interface FileVisitor. Le plus simple est donc de créer une classe fille qui hérite de la classe SimpleFileVisitor et de redéfinir les méthodes utiles selon les besoins.

L'exemple ci-dessous affiche tous les fichiers .java en ignorant les répertoires target.

Exemple (code Java 7) :

```
  public static void testWalkFileTree() throws IOException {
    final Path repertoire = Paths.get("C:/java/projets");
    Files.walkFileTree(repertoire, new SimpleFileVisitor<Path>() {

      @Override
      public FileVisitResult visitFile(final Path file,
          final BasicFileAttributes attrs) throws IOException {
        final String nom = file.getFileName().toString();
        System.out.println("Fichier : " + nom);
        return FileVisitResult.CONTINUE;
      }

      @Override
      public FileVisitResult preVisitDirectory(final Path dir,
      final BasicFileAttributes attrs) throws IOException {
        FileVisitResult result = FileVisitResult.CONTINUE;
        System.out.println("Répertoire : " + dir);
        return result;
      }
    });
  }
```

Pour lancer le parcours de la hiérarchie d'un répertoire, il faut utiliser la méthode walkFileTree() de la classe Files qui propose deux surcharges :

- Path walkFileTree(Path start, FileVisitor<? super Path> visitor)
- Path walkFileTree(Path start, Set<FileVisitOption> options, int maxDepth, FileVisitor<? super Path> visitor)

La première surcharge attend en paramètres le chemin du répertoire qui doit être parcouru et une instance de type FileVisitor qui va encapsuler les traitements du parcours.

La seconde surcharge attend deux paramètres supplémentaires qui permettent de préciser des options sous la forme d'un ensemble de type FileVisitOption et un entier qui permet de limiter le niveau de profondeur du parcours dans la hiérarchie.

L'énumération FileVisitOption ne contient que la valeur FOLLOW_LINKS qui permet de demander de suivre les liens rencontrés lors du parcours. Par défaut, les liens symboliques ne sont pas suivis par le WalkFileTree. Pour suivre les liens symboliques, il faut préciser l'utilisation de l'option FOLLOW_LINKS.

Exemple (code Java 7) :

```
final Path repertoire = Paths.get("C:/java/projets");

EnumSet<FileVisitOption> options = EnumSet.of(FileVisitOption.FOLLOW_LINKS);

Files.walkFileTree(repertoire, options, Integer.MAX_VALUE, new
SimpleFileVisitor<Path>() {
  // ...
});
```

Si l'option FOLLOW_LINK est utilisée, le walkFileTree est capable de détecter les références circulaires lors du parcours. Dans ce cas, la méthode visitFileFailed() sera invoquée et elle aura une exception de type FileSystemLoopException en paramètre.

Exemple (code Java 7) :

```
      @Override
      public FileVisitResult visitFileFailed(Path file, IOException ioe) {
        if (ioe instanceof FileSystemLoopException) {
          System.err.println("Reference circulaire detectee : " + file);
        } else {
          ioe.printStackTrace();
        }
        return FileVisitResult.CONTINUE;
      }
```

Important : il n'est pas possible de présumer de l'ordre de parcours des répertoires.

Si les traitements modifient le système de fichiers, il est important de faire particulièrement attention dans l'implémentation du FileVisitor. Par exemple :

- Si le parcours est utilisé pour supprimer une sous-arborescence, il est nécessaire de supprimer les fichiers contenus par un répertoire avant de supprimer le répertoire lui-même.
- Si le parcours est utilisé pour copier une sous-arborescence, il faut créer le sous-répertoire avant de copier les fichiers qu'il doit contenir

16.6.3. Les opérations récursives

Les fonctionnalités offertes par la classe Files ne s'appliquent pas de manière récursive : il est nécessaire de parcourir l'arborescence en utilisant une des deux techniques ci-dessus pour réaliser des opérations sur un répertoire.

Par exemple, la méthode size() de la classe Files ne s'applique que sur un fichier. Pour déterminer la taille d'un répertoire (en fait la taille des fichiers qu'il contient), il faut écrire du code qui va parcourir son contenu et cumuler les tailles des fichiers qu'il contient.

Exemple (code Java 7) :

```
  public static long getDirectorySize(final Path repertoire) throws IOException {
    final AtomicLong size = new AtomicLong();
    if (!Files.isDirectory(repertoire)) {
      throw new IllegalArgumentException(
          "Le chemin n'est pas celui d'un répertoire");
    }
    Files.walkFileTree(repertoire, new SimpleFileVisitor<Path>() {

      @Override
      public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
          throws IOException {
        if (Files.isRegularFile(file)) {
          size.addAndGet(attrs.size());
        }
        return FileVisitResult.CONTINUE;
      }

      @Override
      public FileVisitResult preVisitDirectory(Path dir,
          BasicFileAttributes attrs) throws IOException {
        FileVisitResult resultat = FileVisitResult.CONTINUE;
        if (!dir.equals(repertoire)) {
          size.addAndGet(getDirectorySize(dir));
          resultat = FileVisitResult.SKIP_SUBTREE;
        }
        return resultat;
      }
    });
    return size.get();
  }
```

Il est possible de télécharger séparément les exemples du JDK : plusieurs de ces exemples situés dans le sous-répertoire sample/nio/file concernent des fonctionnalités utilisant des opérations récursives avec l'API NIO2.

16.7. L'utilisation de systèmes de gestion de fichiers

Un système de gestion de fichiers est encapsulé par un objet de type FileSystem qui permet de créer des objets qui pourront interagir avec lui.

Il faut utiliser la fabrique FileSystems pour obtenir une instance de type FileSystem.

16.7.1. La classe FileSystems

La classe FileSystems est une fabrique pour obtenir des instances de type FileSystem.

La méthode getDefault() renvoie une instance de type FileSystem qui encapsule le système de fichiers de la JVM.

La méthode getFileSystem() renvoie une instance de type FileSystem qui encapsule le système de fichiers dont l'URI est fourni en paramètre.

Plusieurs surcharges de la méthode newFileSystem() permettent de créer une instance spécifique de type FileSystem.

16.7.2. La classe FileSystem

La classe FileSystem encapsule un système de fichiers. C'est essentiellement une fabrique d'instances d'objets dépendants du système encapsulé notamment : Path, PathMatcher, FileStores, WatchService, ...

Pour obtenir une instance de la classe FileSystem qui encapsule le système de fichiers par défaut, il faut utiliser la méthode getDefault() de la classe FileSystems.

Les systèmes de fichiers n'utilisent pas tous le même séparateur dans les chemins de leurs éléments : par exemple, Windows utilise le caractère antislash, les systèmes de type Unix utilisent le caractère slash, ...

Pour connaître le séparateur utilisé par le système, il est possible d'invoquer la méthode getSeparator() de la classe FileSystem.

Exemple (code Java 7) :

```
private static void testGetSeparator() {
  String separator = FileSystems.getDefault().getSeparator();
  System.out.println(separator);
}
```

Résultat :

```
\
```

La méthode getRootDirectories() permet d'obtenir un objet de type Iterable<Path> qui permet d'obtenir les éléments racine du système de fichiers par défaut.

Exemple (code Java 7) :

```
public static void getRootDirectories() throws IOException {
  Iterable<Path> dirs = FileSystems.getDefault().getRootDirectories();
  for (Path name: dirs) {
    System.err.println(name);
  }
}
```

Résultat :

```
C:\
```

16.7.3. La création d'une implémentation de FileSystem

La classe FileSystem est extensible.

Il est par exemple possible de développer ses propres implémentations permettant d'offrir différentes vues d'un système de fichiers (cacher des fichiers sensibles, accès en lecture seule à tous les éléments du système, ...).

Il faut créer une classe qui hérite de la classe FileSystemProvider et une classe qui hérite de la classe FileSystem.

Exemple (code Java 7) :

```
public class MonFileSystem extends FileSystem {
  // ...
}
```

La prise en compte du FileSystem se fait en utilisant le service Provider de la JVM. Il faut donc packager le Filesystem dans une archive de type jar contenant un sous-répertoire META-INF/services avec un fichier java.nio.file.spi.FileSystemProvider listant les noms pleinement qualifiés des sous-classes de type FileSystemProvider.

L'implémentation d'un FileSystem n'a pas besoin d'être liée à un «vrai» système de fichiers.

16.7.4. Une implémentation de FileSystem pour les fichiers Zip

L'implémentation du JDK propose en standard une implémentation spéciale de la classe FileSystem pour faciliter la manipulation de fichiers compressés au format ZIP. Son utilisation rend la manipulation d'archives de type zip beaucoup plus aisée que l'utilisation des classes du package java.util.zip.

Il faut utiliser la fabrique FileSystems pour créer une instance de type FileSystem en invoquant la méthode newFileSystem() et en lui passant en paramètre une instance de type Path qui encapsule le chemin de l'archive à manipuler.

Il est alors possible d'utiliser cette instance de FileSystem pour obtenir des chemins contenus dans l'archive puisque l'archive est vue elle-même comme un système de fichiers particulier. L'utilisation de ces chemins se fait de la même manière que pour les chemins obtenus d'une instance de type FileSystem encapsulant un système de fichiers du système d'exploitation.

L'exemple ci-dessous affiche le contenu d'un fichier contenu dans une archive de type jar.

Exemple (code Java 7) :

```
  public static void testZip() throws IOException {
    // Path de l'archive
    final Path jarfile = Paths.get("c:/java/test/archive.jar");

    // création d'une instance de FileSystem pour gérer les zip
    final FileSystem fs = FileSystems.newFileSystem(jarfile, null);
    // Path du fichier à accéder dans l'archive
    final Path mf = fs.getPath("META-INF", "MANIFEST.MF");

    // lecture et affichage du fichier contenu dans l'archive
    try (BufferedReader readBuffer = Files.newBufferedReader(mf,
      Charset.defaultCharset())) {
      String ligne = "";
      while ((ligne = readBuffer.readLine()) != null) {
        System.out.println(ligne);
      }
    }
  }
```

L'extraction d'un fichier d'une archive de type zip se fait simplement en invoquant la méthode copy() de la classe Files en lui passant en paramètres une instance de type Path du chemin dans l'archive et une instance de type Path du chemin cible.

L'exemple ci-dessous extrait un fichier contenu dans une archive de type jar.

Exemple (code Java 7) :

```
  public static void testZip() throws IOException {
    // Path de l'archive
    final Path jarfile = Paths.get("c:/java/test/archive.jar");

    // creation d'une instance de FileSystem pour gérer les zip
    final FileSystem fs = FileSystems.newFileSystem(jarfile, null);

    // Path du fichier cible
    final Path cible = Paths.get("c:/java/test/MANIFEST.MF");
    Files.deleteIfExists(cible);

    // extraire l'élément de l'archive
    Files.copy(fs.getPath("/META-INF/MANIFEST.MF"), cible);
    if (Files.exists(cible)) {
      System.out.println("fichier " + cible.getFileName() +
        " extrait de l'archive " + jarfile);
    }
  }
```

Pour créer une archive de type zip vide, il faut créer une instance de type FileSystem en utilisant la méthode newFileSystem() et en lui passant en paramètre :

- une URI du chemin de l'archive dont le protocole est jar:file:
- une collection de type Map qui contienne une occurrence ayant pour clé "create" et pour valeur "true"

L'exemple ci-dessous crée une archive de type zip vide.

Exemple (code Java 7) :

```
  private static FileSystem creerZipFileSystem(Path zipFile) throws IOException {
    final URI uri = URI.create("jar:file:" + zipFile.toUri().getPath());

    final Map<String, String> env = new HashMap<>();
    env.put("create", "true");
    return FileSystems.newFileSystem(uri, env);
  }
```

L'ajout d'un fichier dans une archive se fait en utilisant la méthode copy() de la classe Files avec comme paramètres le chemin de la source et le chemin dans l'archive.

L'exemple ci-dessous ajoute un nouveau fichier dans une nouvelle archive de type zip.

Exemple (code Java 7) :

```
  public static void testAjouterZip() throws IOException {
    final Path pathZip = Paths.get("c:/java/test/monarchive.zip");

    Files.deleteIfExists(pathZip);

    // important : invoquer la méthode close() du FS
    try (FileSystem fs = creerZipFileSystem(pathZip)) {
      Path source = Paths.get("c:/java/test/monfichier.txt");
      Path dest = fs.getPath("/", "monfichier.txt");
      Files.copy(source, dest, StandardCopyOption.COPY_ATTRIBUTES);
    }
  }
```

Pour que le fichier soit correctement ajouté, il est important d'invoquer la méthode close() sur l'instance de type FileSystem qui encapsule l'archive. Dans l'exemple ci-dessus, cette invocation est assurée par l'utilisation d'un try-with-resource.

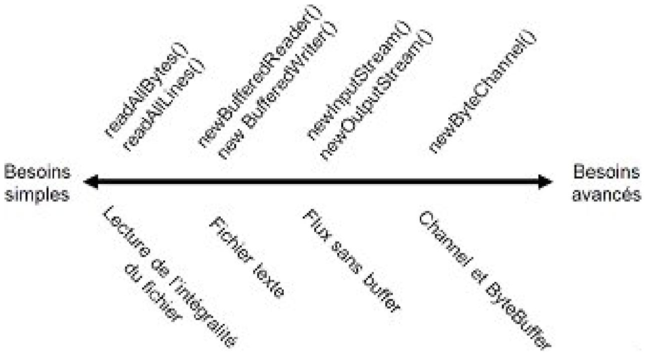
16.8. La lecture et l'écriture dans un fichier

La lecture et l'écriture dans un fichier se font toujours de la même façon avec NIO2 mais l'API propose des méthodes utilitaires pour faciliter le travail.

La gestion des opérations de types entrées/sorties a évolué au fur et à mesure des versions de Java.

IO	NIO	NIO2
Java 1.0 et 1.1	Java 1.4 (JSR 151)	Java 7 (JSR 203)
Synchrone bloquant	Synchrone non bloquant	Asynchrone non bloquant
File InputStream OutputStream Reader (Java 1.1) Writer (Java 1.1) Socket RandomAccessFile	FileChannel SocketChannel ServerSocketChannel (Charset, Selector, ByteBuffer)	Path AsynchronousFileChannel AsynchronousByteChannel AsynchronousSocketChannel AsynchronousServerSocketChannel SeekableByteChannel

La classe Files propose plusieurs méthodes pour faciliter la lecture ou l'écriture de fichiers et de flux selon les besoins allant des plus simples aux plus complexes.



Les méthodes readAllBytes() et readAllLines() permettent de lire l'intégralité du contenu d'un fichier respectivement d'octets et texte. Deux surcharges de la méthode write() permettent d'écrire l'intégralité d'un fichier. Ces méthodes sont à réserver pour de petits fichiers.

Les méthodes newBufferedReader() et newBufferedWriter() sont des helpers pour faciliter la création d'objets de types BufferedReader et BufferedWriter permettant la lecture et l'écriture de fichiers de type texte en utilisant un tampon.

Les méthodes newInputStream() et newOutputStream() sont des helpers pour faciliter la création d'objets permettant la lecture et l'écriture de fichiers d'octets.

Ces quatre méthodes sont des helpers pour créer des objets du package java.io.

La méthode newByteChannel() est un helper pour créer un objet de type SeekableByteChannel.

La classe FileChannel propose des fonctionnalités avancées sur l'utilisation d'un fichier (verrous, mapping direct à une zone de la mémoire, ...) : cette classe a été enrichie pour fonctionner avec NIO2.

16.8.1. Les options d'ouverture d'un fichier

L'énumération StandardOpenOption implémente l'interface OpenOption et définit les options d'ouverture standard d'un fichier :

Valeur	Rôle
APPEND	Si le fichier est ouvert en écriture alors les données sont ajoutées au fichier. Cette option doit être utilisée avec les options CREATE ou WRITE
CREATE	Créer un nouveau fichier s'il n'existe pas sinon le fichier est ouvert
CREATE_NEW	Créer un nouveau fichier : si le fichier existe déjà alors une exception est levée
DELETE_ON_CLOSE	Supprimer le fichier lorsque son flux associé est fermé : cette option est utile pour des fichiers temporaires
DSYNC	Demander l'écriture synchronisée des données dans le système de stockage sous-jacent (pas d'utilisation des tampons du système)
READ	Ouvrir le fichier en lecture
SPARSE	Indiquer au système que le fichier est clairsemé ce qui peut lui permettre de réaliser certaines optimisations si l'option est supportée par le système de fichiers (c'est notamment le cas avec NTFS)
SYNC	Demander l'écriture synchronisée des données et des métadonnées dans le système de stockage sous-jacent
TRUNCATE_EXISTING	Si le fichier existe et qu'il est ouvert en écriture alors il est vidé. Cette option doit être utilisée avec l'option WRITE
WRITE	Ouvrir le fichier en écriture

Ces options sont utilisables avec toutes les méthodes qui ouvrent des fichiers. Elles ne sont pas toutes mutuellement exclusives.

16.8.2. La lecture et l'écriture de l'intégralité d'un fichier

La classe Files propose les méthodes readAllLines() et readAllBytes() qui renvoient respectivement une collection de type List<String> et un tableau d'octets contenant l'intégralité d'un fichier texte ou binaire. Bien sûr l'utilisation de ces méthodes est à réserver pour des fichiers de petites tailles.

La méthode readAllLines() de la classe Files permet de lire l'intégralité d'un fichier et de renvoyer son contenu sous la forme d'une collection de chaînes de caractères.

Exemple (code Java 7) :

```
List<String> lignes =  Files.readAllLines(
  FileSystems.getDefault().getPath("monfichier.txt"), StandardCharsets.UTF_8);
for (String ligne : lignes)
  System.out.println(ligne);
```

La méthode readAllLines() attend en paramètre un objet de type Path qui encapsule le chemin du fichier à lire et un objet de type Charset qui précise le jeu d'encodage de caractères du fichier. Elle s'occupe d'ouvrir le fichier, lire le contenu et fermer le flux.

La méthode readAllBytes() de la classe Files permet de lire l'intégralité d'un fichier et renvoyer son contenu sous la forme d'un tableau d'octets.

Exemple (code Java 7) :

```
Path file = FileSystems.getDefault().getPath("monfichier.bin");
byte[] contenu = Files.readAllBytes(file);
```

La méthode write() permet d'écrire le contenu d'un fichier. Elle possède deux surcharges :

- Path write(Path path, byte[] bytes, OpenOption... options)
- Path write(Path path, Iterable<? extends CharSequence> lines, Charset cs, OpenOption... options)

Exemple (code Java 7) :

```
    final Path pathSource = Paths.get("c:/java/source.txt");
    final Path pathCible = Paths.get("c:/java/cible.txt");
    final List<String> lignes = Files.readAllLines(pathSource, Charset.defaultCharset());
    Files.write(pathCible, lignes, Charset.defaultCharset());
```

Exemple (code Java 7) :

```
    final Path pathSource = Paths.get("c:/java/source.bin");
    final Path pathCible = Paths.get("c:/java/cible.bin");
    // lire et écrire tout le fichier
    final byte[] bytes = Files.readAllBytes(pathSource);
    Files.write(pathCible, bytes);
```

16.8.3. La lecture et l'écriture bufférisées d'un fichier

Avant Java 7, pour lire un fichier avec un tampon, il fallait invoquer le constructeur de la classe BufferedReader en lui passant en paramètre un objet de type Reader.

Exemple :

```
BufferedReader in = new BufferedReader(new FileReader("monfichier.txt"));
```

A partir de Java 7, il est possible d'utiliser la méthode newBufferedReader() de la classe Files.

Exemple (code Java 7) :

```
BufferedReader in = Files.newBufferedReader(Paths.get("monfichier.txt"),
  Charset.forName("UTF-8"));
```

Le résultat est quasiment le même mais il est nécessaire de préciser le jeu d'encodage des caractères. La classe FileReader utilise toujours le jeu par défaut du système. Même si ce n'est pas une bonne pratique, il est possible d'obtenir ce jeu d'encodage de caractères en invoquant la méthode java.nio.charset.Charset.defaultCharset().

La méthode newBufferedReader() de la classe Files renvoie un objet de type BufferedReader qui permet de lire le fichier dont le chemin et le jeu de caractères d'encodage sont fournis en paramètres.

Exemple (code Java 7) :

```
  public static void testNewBufferedReader() throws IOException {
    Path sourcePath = Paths.get("C:/java/temp/monfichier.txt");
    try (BufferedReader reader = Files.newBufferedReader(sourcePath,
        StandardCharsets.UTF_8)) {
      String line = null;
      while ((line = reader.readLine()) != null) {
        System.out.println(line);
      }
    }
  }
```

La méthode newBufferedReader() ouvre un fichier de type texte pour des lectures avec un tampon. Elle retourne un objet de type BufferedReader.

Exemple (code Java 7) :

```
    Path fichier = Paths.get("monfichier.txt");
    Charset charset = Charset.forName("US-ASCII");
    try (BufferedReader reader = Files.newBufferedReader(fichier, charset)) {
      String line = null;
      while ((line = reader.readLine()) != null) {
        System.out.println(line);
      }
    } catch (IOException ioe) {
      ioe.printStacktrace();
    }
```

La méthode newBufferedWriter() ouvre un fichier de type texte pour des écritures avec un tampon. Elle retourne un objet de type BufferedWriter.

Exemple (code Java 7) :

```
    Path fichier = Paths.get("monfichier.txt");
    Charset charset = Charset.forName("US-ASCII");
    String contenu = "Contenu du fichier";
    try (BufferedWriter writer = Files.newBufferedWriter(fichier, charset)) {
      writer.write(contenu, 0, contenu.length());
    } catch (IOException ioe) {
      ioe.printStacktrace();
    }
```

16.8.4. La lecture et l'écriture d'un flux d'octets

Les méthodes newInputStream() et newOutputStream() permettent d'obtenir une instance de type InputStream et une instance de type OutputStream sur le fichier dont le chemin est fourni en paramètre :

Méthode	Rôle
InputStream newInputStream(Path path, OpenOption... options)	Créer un objet de type InputStream
OutputStream newOutputStream(Path path, OpenOption... options)	Créer un objet de type OutputStream

Les méthodes newInputStream() et newOutputStream() attendent en paramètres un objet de type Path et un varargs de type OpenOption.

La méthode newInputStream() ouvre un fichier pour des lectures sans tampon. Elle retourne un objet de type InputStream.

Exemple (code Java 7) :

```
public static void testNewInputStream() throws IOException {
  Path path = Paths.get("c:/java/test/monfichier.txt");
  try (InputStream in = Files.newInputStream(path);
      BufferedReader reader = new BufferedReader(new InputStreamReader(in))) {
    String line = null;
    while ((line = reader.readLine()) != null) {
      System.out.println(line);
    }
  } catch (IOException x) {
    System.err.println(x);
  }
}
```

La méthode newOutputStream() ouvre un fichier pour des écritures sans tampon. Elle retourne un objet de type OutputStream. Si aucun paramètre de type OpenOption n'est précisé, la méthode va utiliser les paramètres CREATE et TRUNCATE_EXISTING par défaut (créer le fichier s'il n'existe pas et le vider s'il existe).

Exemple (code Java 7) :

```
public static void testNewOutputStream() throws IOException {
  Path path = Paths.get("c:/java/test/monfichier.txt");
  try (OutputStream out = Files.newOutputStream(path,
      StandardOpenOption.TRUNCATE_EXISTING, StandardOpenOption.WRITE)) {
    out.write('X');
  }
}
```

16.8.5. La lecture et l'écriture d'un fichier avec un channel

L'API Java NIO propose de réaliser des opérations d'entrées/sorties utilisant des channels et des tampons (ByteBuffer) ce qui améliore les performances par rapport à l'API Java IO.

Par défaut les flux de java.io lisent ou écrivent uniquement un octet ou un caractère à la fois.

Les opérations de lectures/écritures de java.nio utilisent un tampon (ByteBuffer). L'interface ByteChannel propose des fonctionnalités de base pour de telles lectures ou écritures.

La méthode newByteChannel() de la classe Files renvoie une instance d'un channel NIO de type SeekableByteChannel. Elle possède deux surcharges :

- SeekableByteChannel newByteChannel(Path path, OpenOption... options)
- SeekableByteChannel newByteChannel(Path path, Set<? extends OpenOption> options, FileAttribute<?>... attrs)

Ces deux surcharges permettent d'ouvrir ou de créer un fichier et de lui associer un channel en fonction des paramètres d'ouverture de type OpenOption fournis. Par défaut le channel est ouvert en lecture (option READ).

Exemple (code Java 7) :

```
  final Path path = Paths.get("C:/java/test/fichier.bin");

  Files.deleteIfExists(path);

  try (SeekableByteChannel sbc = Files.newByteChannel(path,
       StandardOpenOption.WRITE, StandardOpenOption.SYNC)) {

     // ...
```

```
    }
```

L'interface java.nio.channels.SeekableByteChannel ajoute à l'interface ByteChannel la possibilité de gérer une position dans le channel, de vider un channel et d'obtenir la taille du fichier associé au channel. Cela permet de se déplacer dans le channel pour réaliser une opération de lecture ou d'écriture sans avoir à parcourir les données jusqu'à la position désirée. Un SeekableByteChannel est donc un channel qui possède des fonctionnalités similaires à celles proposées par la classe java.io.RandomAccessFile.

L'interface SeekableByteChannel hérite des interfaces : AutoCloseable, ByteChannel, Channel, Closeable, ReadableByteChannel et WritableByteChannel.

Elle propose plusieurs méthodes pour permettre de se déplacer dans le fichier avant de réaliser une opération de lecture ou d'écriture.

Méthode	Rôle
long position()	Retourner la position courante dans le channel
SeekableByteChannel position(long newPosition)	Changer la position dans le channel
int read(ByteBuffer dst)	Lire un ensemble d'octets du channel dans le tampon fourni en paramètre. Retourne le nombre d'octets lus ou -1 si la fin du channel est atteinte
long size()	Retourner la taille en octets du flux auquel le channel est connecté
SeekableByteChannel truncate(long size)	Tronquer le contenu de l'élément sur lequel le channel est connecté à la taille fournie en paramètre. Cela permet de redimensionner la taille du flux associé au channel avec la valeur fournie en paramètre
int write(ByteBuffer src)	Ecrire les octets fournis en paramètre à la position courante dans le channel

La méthode read() tente une lecture pour remplir le nombre d'octets du tampon passé en paramètre. Elle renvoie -1 si la fin du flux est atteinte. La position courante dans le channel est augmentée de la taille des données lues.

La méthode write() écrit les octets du tampon passé en paramètre à partir de la position courante dans le channel. Si le fichier est ouvert avec l'option APPEND, alors la position courante est située à la fin du fichier. Elle renvoie le nombre d'octets écrits. La position courante dans le channel est augmentée de la taille des données écrites.

La surcharge de la méthode position() qui attend un paramètre de type long permet de déplacer la position courante dans le channel. Elle renvoie le channel lui-même pour permettre un chaînage des appels de cette méthode. La taille du flux connecté au channel n'est pas modifiée si la valeur fournie en paramètre est supérieure à sa taille totale. L'utilisation de cette méthode n'est pas recommandée avec un channel ouvert avec l'option APPEND.

La méthode truncate() permet de réduire la taille totale du flux connecté au channel. Si la taille fournie en paramètre est inférieure à la taille totale courante, alors les octets entre la taille fournie et la taille totale sont perdus. Si la taille fournie est supérieure ou égale à la taille du flux connecté au channel alors l'invocation de la méthode n'a aucun effet. Une implémentation de cette interface peut interdire l'utilisation de cette méthode si le channel est ouvert avec l'option APPEND.

Exemple (code Java 7) :

```
    final ByteBuffer donneesBonjour = ByteBuffer.wrap("Bonjour".getBytes());
    final ByteBuffer donneesBonsoir = ByteBuffer.wrap("Bonsoir".getBytes());

    final Path path = Paths.get("C:/java/test/fichier.bin");

    Files.deleteIfExists(path);
    try (FileChannel fileChannel = FileChannel.open(path,
         StandardOpenOption.CREATE, StandardOpenOption.WRITE,
         StandardOpenOption.SYNC)) {
      fileChannel.position(100);
      fileChannel.write(donneesBonjour);
    }
    try (SeekableByteChannel sbc = Files.newByteChannel(path,
         StandardOpenOption.WRITE, StandardOpenOption.SYNC)) {
      sbc.position(200);
      sbc.write(donneesBonsoir);
    }
```

La méthode Files.newByteChannel() permet de créer une instance de type SeekableByteChannel. Si le fichier connecté au channel est sur le système de fichiers par défaut, il est possible de caster l'objet retourné en un objet de type FileChannel.

La classe abstraite FileChannel propose des fonctionnalités avancées à utiliser sur un channel connecté à un fichier :

- des octets peuvent être lus ou écrits sans modifier la position courante dans le channel
- une région du fichier peut être mappée directement en mémoire (cette fonctionnalité est intéressante pour manipuler de gros fichiers)
- l'écriture de données peut être forcée pour être faite directement sur le système de stockage afin d'éviter une perte de données en cas de crash du système
- une région du fichier peut être verrouillée pour empêcher l'accès par d'autres applications

16.9. Les liens et les liens symboliques

Il existe deux types de liens :

- liens physiques (hard links) : ils permettent de faire référence à un élément physique du système de fichiers qui doit exister. Si le fichier cible est modifié alors le lien est aussi modifié.
- liens symboliques (symbolic links) : ils permettent de faire référence à un autre élément du système de fichiers. Si l'élément cible est supprimé alors le lien existe toujours mais il est invalide.

La classe Files propose deux méthodes pour créer des liens physiques et des liens symboliques.

Méthode	Rôle
Path createSymbolicLink(Path link, Path target, FileAttribute<?>... attrs)	Créer un lien symbolique vers un élément
Path createLink(Path link, Path existing)	Créer un lien physique

16.9.1. La création d'un lien physique

Les liens physiques (hard links) possèdent quelques restrictions :

- le fichier cible doit exister
- le fichier cible doit être sur la même partition
- il possède les mêmes attributs que le fichier cible

Pour créer un lien, il faut invoquer la méthode createLink() de la classe Files qui attend en paramètres deux objets de type Path : le premier est le chemin du lien, le second est le chemin du fichier cible qui s'il n'existe pas lèvera une exception de type NoSuchFileException.

Exemple (code Java 7) :

```
  public static void testCreateLink() throws IOException {
    Path lien = Paths.get("c:/java/test/monlien.lnk");
    Path cible = Paths.get("c:/java/test/monfichier.txt");
    Files.createLink(lien, cible);
  }
```

16.9.2. La création d'un lien symbolique

La méthode createSymbolicLink() de la classe Files permet de créer un lien symbolique. Le premier paramètre de type Path est le chemin du lien symbolique. Le second paramètre de type Path est le chemin vers le fichier ou le répertoire cible. Le paramètre de type varargs FileAttributes permet de préciser les options du lien qui seront utilisées lors de sa création.

Exemple (code Java 7) :

```
Path lien = Paths.get("/home/jm/monlien");
Path cible = Paths.get("/home/jm/monfichier.txt");
Files.createSymbolicLink(lien, cible);
if (Files.isSameFile(lien, cible)) {
  System.out.println("Identique");
} else {
  System.out.println("Non identique");
}
```

L'utilisation des liens symboliques est conditionnée par le fait que le système d'exploitation sous-jacent propose un support de ces liens. Si le système sous-jacent ne supporte pas les liens symboliques, une exception de type UnsupportedOperationException est levée lors de l'invocation de la méthode createSymbolicLink().

Exemple sous Windows XP

Exemple (code Java 7) :

```
   public static void testSymbolicLink() {
    Path newLink = Paths.get("C:/test_link");
    Path target = Paths.get("C:/Users/test");
    try {
      Files.createSymbolicLink(newLink, target);
    } catch (IOException ioe) {
      ioe.printStackTrace();
    } catch (UnsupportedOperationException uoe) {
      // Le systeme de fichiers ne supporte pas les liens symboliques.
      uoe.printStackTrace();
    }
  }
```

Le support des liens symboliques est aussi contrôlé par un SecurityManager en utilisant l'option LinkPermission("symbolic") : leur support est désactivé par défaut. Une exception de type SecurityException peut donc être levée si un SecurityManager est utilisé et que les droits adéquats ne sont pas activés.

16.9.3. L'utilisation des liens et des liens symboliques

La méthode toRealPath() de l'interface Path permet de retourner un chemin dont les liens symboliques contenus dans le chemin sont résolus.

La méthode isSymbolicLink() de l'interface Path permet de déterminer si l'élément précisé par le chemin est un lien symbolique ou non.

La méthode readSymbolicLink() de la classe Files renvoie le chemin de la cible du lien symbolique ou lève une exception de type NotLinkException si l'élément dont le chemin fourni en paramètre n'est pas un lien symbolique.

La suppression d'un lien se fait en utilisant la méthode delete() de la classe Files : dans ce cas, c'est le lien qui est supprimé et non le fichier cible.

Certaines méthodes de la classe Files attendent en paramètre un varargs de type LinkOption. L'option LinkOption.NOFOLLOW_OPTIONS permet de demander de ne pas suivre les liens pour réaliser l'action demandée.

16.10. La gestion des attributs

Les éléments d'un système de fichiers possèdent des métadonnées généralement nommées attributs : le type d'éléments (fichier, répertoire, lien), la taille, la date de création et de modification, les permissions d'utilisation, ... Le nombre de ces métadonnées et la façon dont elles sont gérées sont dépendants du système d'exploitation.

NIO 2 permet de gérer les permissions sur les fichiers. Malheureusement, ces permissions sont dépendantes du système de fichiers sous-jacent. NIO 2 propose des classes dédiées pour chaque système de fichiers supporté qui sont regroupées dans le package java.nio.file.attribute.

L'accès aux métadonnées a été enrichi avec NIO 2 : certains attributs de base sont accessibles par la classe Files d'autres sont accessibles au travers de vues.

L'implémentation par défaut propose plusieurs vues pour les principaux types de système d'exploitation :

- Basic : cette vue est commune à tous les systèmes d'exploitation
- Dos : cette vue est dédiée aux systèmes d'exploitation Windows
- Posix : cette vue est dédiée aux systèmes d'exploitation de type Unix like avec notamment une gestion sur des permissions adaptées à ce type de système

Il est aussi possible qu'une implémentation spécifique soit fournie par un tiers ou encore, de développer sa propre implémentation.

16.10.1. La gestion individuelle des attributs

La classe Files propose plusieurs méthodes pour obtenir individuellement certains de ces attributs pour un élément dont le chemin est fourni en paramètre.

Méthode	Rôle
boolean isDirectory(Path, LinkOption)	Renvoyer un booléen qui précise si l'élément est un répertoire
boolean isRegularFile(Path, LinkOption...)	Renvoyer un booléen qui précise si l'élément est un fichier
boolean isSymbolicLink(Path)	Renvoyer un booléen qui précise si l'élément est un lien symbolique
boolean isHidden(Path)	Renvoyer un booléen qui précise si l'élément est caché
FileTime getLastModifiedTime(Path, LinkOption...)	Renvoyer la date/heure de dernière modification de l'élément
Path setLastModifiedTime(Path, FileTime)	Modifier la date de dernière modification de l'élément
UserPrincipal getOwner(Path, LinkOption...)	Renvoyer le propriétaire du fichier
Path setOwner(Path, UserPrincipal)	Modifier le propriétaire du fichier
Set<PosixFilePermission> getPosixFilePermissions(Path, LinkOption...)	Renvoyer les droits d'un élément d'un système de type Unix
Path setPosixFilePermissions(Path, Set<PosixFilePermission>)	Modifier les droits d'un élément d'un système de type Unix
Object getAttribute(Path, String, LinkOption...)	Obtenir la valeur d'un attribut de l'élément
Path setAttribute(Path, String, Object, LinkOption...)	Modifier la valeur d'un attribut de l'élément
boolean isExecutable()	Renvoyer un booléen qui précise si l'élément peut être exécuté
boolean isReadable()	Renvoyer un booléen qui précise si l'élément peut être lu

boolean isWritable()	Renvoyer un booléen qui précise si l'élément peut être modifié
long size(Path)	Renvoyer la taille en octets d'un fichier

Il est possible d'utiliser la méthode getOwner(Path) de la classe Files pour obtenir un objet de type UserPrincipal qui encapsule le propriétaire du fichier.

Exemple (code Java 7) :

```
  public static void testGetOwner() throws IOException {
    Path fichier = Paths.get("C:/java/temp/monfichier.txt");
    UserPrincipal owner = Files.getOwner(fichier);
    System.out.println(owner);
  }
```

Résultat :

```
THINKPAD_X60S\jm (User)
```

16.10.2. La gestion de plusieurs attributs

Si l'application a besoin de plusieurs attributs d'un même élément, il est plus efficace d'utiliser une des surcharges de la méthode readAttributes() qui renvoie un objet encapsulant des attributs d'une même famille. Les performances peuvent être dégradées si le système de fichiers est consulté plusieurs fois pour obtenir des attributs.

Méthode	Rôle
Map<String, Object> readAttributes(Path, String, LinkOption...)	Renvoyer une collection d'attributs lus en une seule opération
<A extends BasicFileAttributes> A readAttributes(Path, Class<A>, LinkOption...)	Renvoyer un objet qui encapsule les attributs lus en une seule opération. Le type de cet objet est précisé en paramètre

Exemple (code Java 7) :

```
  public static void lectureBasicAttributs() {
    Path monFichier = Paths.get("C:/Users/jm/AppData/Local/Temp/monfichier.txt");
    BasicFileAttributes basicAttrs;
    try {
      basicAttrs = Files.readAttributes(monFichier, BasicFileAttributes.class);

      System.out.println("creationTime     = " + basicAttrs.creationTime());
      System.out.println("lastAccessTime   = " + basicAttrs.lastAccessTime());
      System.out.println("lastModifiedTime = " + basicAttrs.lastModifiedTime());
      System.out.println("isDirectory      = " + basicAttrs.isDirectory());
      System.out.println("isOther          = " + basicAttrs.isOther());
      System.out.println("isRegularFile    = " + basicAttrs.isRegularFile());
      System.out.println("isSymbolicLink   = " + basicAttrs.isSymbolicLink());
      System.out.println("size             = " + basicAttrs.size());
      System.out.println("fileKey          = " + basicAttrs.fileKey());
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
```

Résultat :

```
creationTime    = 2011-07-19T14:12:07.916077Z
lastAccessTime  = 2011-07-19T14:12:07.916077Z
lastModifiedTime = 2011-07-23T16:39:05.957393Z
isDirectory     = false
isOther         = false
isRegularFile   = true
```

```
isSymbolicLink  = false
size            = 16
fileKey         = null
```

Pour obtenir une instance de type BasicFileAttributes, il faut invoquer la méthode readAttributes() de la classe Files en lui passant en paramètre le chemin du fichier et une instance de type Class pour la classe BasicFileAttributes. Il est aussi possible de préciser des options sous la forme d'un varargs de l'énumération de type LinkOption.

La valeur LinkOption.NOFOLLOW_LINKS indique de ne pas suivre les liens symboliques.

La méthode readAttributes() permet de lire en une seule opération plusieurs attributs encapsulés dans l'objet retourné lors de son invocation, ce qui est plus efficace que de lire ces attributs un par un.

Les attributs creationTime, lastModifiedTime et lastAccessTime encapsulés dans la classe BasicFilesAttributes sont de type java.nio.file.attribute.FileTime qui encapsule un horodatage.

Il est possible de créer une instance de la classe FileTime en utilisant les méthodes :

- from(long, TimeUnit) : créer une instance à partir de la valeur et de l'unité fournies en paramètre
- fromMillis(long) : créer une instance à partir du nombre de millisecondes fourni en paramètre

Exemple (code Java 7) :

```
  public static void testSetLastModifiedTime() throws IOException {
    Path fichier = Paths.get("c:/java/test/monfichier.txt");
    long currentTime = System.currentTimeMillis();
    FileTime ft = FileTime.fromMillis(currentTime);
    Files.setLastModifiedTime(fichier, ft);
  }
```

La méthode fileKey() renvoie un objet qui encapsule une clé unique du fichier dans le système de fichiers si celui-ci supporte cette fonctionnalité sinon elle renvoie null.

16.10.3. L'utilisation des vues

Les différents types de systèmes de fichiers possèdent des attributs communs mais possèdent aussi des attributs spécifiques. La notion de vue regroupe plusieurs attributs ce qui permet d'obtenir ces attributs en une fois. L'API propose en standard plusieurs vues qui sont spécialisées :

- BasicFileAttributeView : propose une vue qui contient des attributs communs à tous les systèmes de fichiers
- DosFileAttributeView : propose une vue qui permet un support des quatre attributs spécifiques à un système de fichiers de type DOS (readonly, hidden, system et archive)
- PosixFileAttributeView : propose une vue qui permet un support des attributs spécifiques à un système de fichiers de type Posix notamment la gestion des droits pour le propriétaire, le groupe et les autres utilisateurs.
- FileOwnerAttributeView : propose une vue qui permet une gestion du propriétaire de l'élément qui correspond par défaut à celui qui a créé l'élément
- AclFileAtributeView : propose une vue qui permet le support de la gestion des droits de type ACL
- UserDefinedFileAttributeView : propose une vue qui permet le support de métadonnées spécifiques à un système de fichiers

Une vue peut permettre un accès en lecture seule aux données ou permettre leur mise à jour.

Un système de fichiers ne peut être supporté que par la BasicFileAttributeView ou être supporté par plusieurs vues. Un système de fichiers peut même proposer une ou plusieurs vues spécifiques qui ne sont pas fournies en standard par l'API.

Pour obtenir une vue spécifique, il faut utiliser la méthode getFileAttributeView() de la classe Files en précisant le type de la vue souhaitée.

Exemple (code Java 7) :

```
  public static void testBasicFileAttributeView() throws IOException {
    Path path = Paths.get("c:/java/test/monfichier.txt");
    BasicFileAttributeView basicView = Files.getFileAttributeView(path,
        BasicFileAttributeView.class);
    if (basicView != null) {
      BasicFileAttributes basic = basicView.readAttributes();

      System.out.println("isRegularfile    " + basic.isRegularFile());
      System.out.println("isDirectory      " + basic.isDirectory());
      System.out.println("isSymbolicLink   " + basic.isSymbolicLink());
      System.out.println("isOther          " + basic.isOther());
      System.out.println("size             " + basic.size());
      System.out.println("creationTime     " + basic.creationTime());
      System.out.println("lastAccesstime   " + basic.lastAccessTime());
      System.out.println("lastModifiedTime " + basic.lastModifiedTime());
    }
  }
```

Les informations de la vue basic peuvent aussi être obtenues en utilisant la classe Files : cependant l'utilisation de la vue permet d'obtenir toutes les informations avec un seul accès à l'élément du système d'exploitation.

16.10.4. La gestion des permissions DOS

La classe DosFileAttributes encapsule les attributs d'un élément d'un système de fichiers de type DOS : read only, hidden, archive et system.

Exemple (code Java 7) :

```
  public static void testDosFileAttributes() throws IOException {
    Path fichier = Paths.get("C:/java/temp/monfichier.txt");
    try {
      DosFileAttributes attr = Files.readAttributes(fichier,
          DosFileAttributes.class);
      System.out.println("isReadOnly = " + attr.isReadOnly());
      System.out.println("isHidden   = " + attr.isHidden());
      System.out.println("isArchive  = " + attr.isArchive());
      System.out.println("isSystem   = " + attr.isSystem());
    } catch (UnsupportedOperationException ueo) {
      ueo.printStackTrace();
    }
  }
```

Résultat :

```
isReadOnly = false
isHidden   = false
isArchive  = true
isSystem   = false
```

Il est aussi possible d'utiliser les méthodes getAttribute() et setAttribute() de la classe Files. L'inconvénient de ces méthodes est que l'attribut concerné est fourni sous la forme d'une chaîne de caractères. Celle-ci doit être composée du nom de la vue suivi du caractère deux points suivi du nom de l'attribut.

Exemple (code Java 7) :

```
  public static void testGetFileAttribute() throws IOException {
    Path fichier = Paths.get("C:/java/temp/monfichier.txt");
    try {
      System.out.println("isReadOnly = " +
        Files.getAttribute(fichier,"dos:readonly", LinkOption.NOFOLLOW_LINKS));
      System.out.println("isHidden   = " +
        Files.getAttribute(fichier,"dos:hidden", LinkOption.NOFOLLOW_LINKS));
      System.out.println("isArchive  = " +
        Files.getAttribute(fichier,"dos:archive",LinkOption.NOFOLLOW_LINKS));
      System.out.println("isSystem   = " +
        Files.getAttribute(fichier,"dos:system", LinkOption.NOFOLLOW_LINKS));
    } catch (UnsupportedOperationException ueo) {
      ueo.printStackTrace();
    }
  }
```

Si le nom de l'attribut fourni en paramètre n'est pas supporté alors une exception de type IllegalArgumentException est levée.

Exemple (code Java 7) :

```
  public static void testGetFileAttribute() throws IOException {
    Path fichier = Paths.get("C:/java/temp/monfichier.txt");
    try {
      System.out.pr intln("isReadOnly = " +
        Files.getAttribute(fichier,"dos:readolny", LinkOption.NOFOLLOW_LINKS));
    } catch (UnsupportedOperationException ueo) {
      ueo.printStackTrace();
    }
  }
```

Résultat :

```
Exception
in thread "main" java.lang.IllegalArgumentException: 'readolny' not
recognized
  at sun.nio.fs.AbstractBasicFileAttributeView$AttributesBuilder.<init>(Unknown Source)
  at sun.nio.fs.AbstractBasicFileAttributeView$AttributesBuilder.create(Unknown Source)
  at sun.nio.fs.WindowsFileAttributeViews$Dos.readAttributes(Unknown Source)
  at sun.nio.fs.AbstractFileSystemProvider.readAttributes(Unknown Source)
  at java.nio.file.Files.readAttributes(Unknown Source)
  at java.nio.file.Files.getAttribute(Unknown Source)
  at fr.jmdoudoux.dej.nio2.TestNIO2.testGetFileAttribute(TestNIO2.java:385)
  at fr.jmdoudoux.dej.nio2.TestNIO2.main(TestNIO2.java:57)
```

La méthode setAttribute() de la classe Files permet de modifier un attribut d'un élément du système de fichiers.

Exemple (code Java 7) :

```
  public static void testSetFileAttribute() throws IOException {
    Path fichier = Paths.get("C:/java/temp/monfichier.txt");
    try {
      Files.setAttribute(fichier,"dos:hidden", false);
    } catch (UnsupportedOperationException ueo) {
      ueo.printStackTrace();
    }
  }
```

16.10.5. La gestion des permissions Posix

La gestion des permissions de type Posix se fait sur trois niveaux : propriétaire, groupe et autres utilisateurs.

Avant Java 7, la modification des attributs d'un fichier sur système POSIX devait se faire en utilisant la méthode System.exec() ou en invoquant une méthode native.

Avec NIO 2, il faut utiliser les classes PosixFilePermission et PosixFilePermissions pour gérer les permissions des systèmes de fichiers respectant la norme POSIX.

Exemple (code Java 7) :

```
Path monFichier = Paths.get("/tmp/monfichier.txt");
```

```
Set<PosixFilePermission>filePermissions    =
  PosixFilePermissions.fromString("rw-rw-r--");
FileAttribute<Set<PosixFilePermission>> fileAttribute     =
  PosixFilePermissions.asFileAttribute(filePermissions);
Files.createFile(monFichier, fileAttribute);
```

Attention : les attributs réellement positionnés sur le fichier peuvent être différents en fonction de règles définies sur le système de fichiers comme par exemple l'utilisation d'un umask sous un système de type Unix.

L'interface PosixFileAttributes qui hérite de l'interface BasicFileAttributes propose des méthodes pour obtenir le propriétaire, le groupe de l'élément du système de fichiers et les permissions.

Méthode	Rôle
UserPrincipal owner()	Renvoyer le propriétaire
GroupPrincipal()	Renvoyer le groupe
Set<PosixFilePermission> permissions()	Renvoyer les permissions de lecture/écriture/exécution du propriétaire, du groupe et des autres

Exemple (code Java 7) :

```
    Path fichier = Paths.get("/home/jm/test.txt");
    PosixFileAttributes attrs = Files.readAttributes(fichier, PosixFileAttributes.class);
    UserPrincipal owner = attrs.owner();
    GroupPrincipal group = attrs.group();
    System.out.println("Le fichier appartient à " + owner + ":" + group);
```

L'énumération PosixFilePermission contient des valeurs pour gérer les droits de lecture, écriture et exécution pour le propriétaire, le groupe et les autres : OWNER_READ, OWNER_WRITE, OWNER_EXECUTE, GROUP_READ, GROUP_WRITE, GROUP_EXECUTE, OTHERS_READ, OTHERS_WRITE, OTHERS_EXECUTE.

Les permissions sont encapsulées dans une collection de type Set d'éléments de type PosixFilePermission.

Exemple (code Java 7) :

```
    PosixFilePermission[] permissionsArray = {
    PosixFilePermission.OWNER_READ, PosixFilePermission.OWNER_WRITE,
    PosixFilePermission.GROUP_READ, PosixFilePermission.GROUP_WRITE };
    Set<PosixFilePermission> newPermissions = new HashSet<>(
      Arrays.asList(permissionsArray));
```

Les gestions des permissions peut se faire en manipulant directement la collection.

Exemple (code Java 7) :

```
    Set<PosixFilePermission> permissions = attributes.permissions();
    permissions.add(PosixFilePermission.OTHERS_READ);
    permissions.remove(PosixFilePermission.GROUP_WRITE);
    Files.setPosixFilePermissions(path, permissions);
```

La classe Files propose la méthode getPosixFilePermissions(Path, LinkOption ...) qui renvoie une collection de type Set<PosixFilePermission> encapsulant les permissions de lecture/écriture/exécution du propriétaire, du groupe et des autres pour l'élément dont le chemin est fourni en paramètre.

La classe PosixFilePermissions propose des méthodes pour faciliter la manipulation d'un ensemble de permissions.

Méthode	Rôle

static FileAttribute<Set<PosixFilePermission>> asFileAttribute(Set<PosixFilePermission> perms)	Créer une instance de type FileAttribute qui encapsule l'ensemble des permissions fournies en paramètre
static Set<PosixFilePermission> fromString(String perms)	Renvoyer un ensemble de permissions à partir d'une chaîne de caractères au format rwxrwxrwx
static String toString(Set<PosixFilePermission> perms)	Renvoyer une représentation de l'ensemble des permissions sous la forme d'une chaîne de caractères au format rwxrwxrwx

La méthode toString() de la classe PosixFilePermissions renvoie une chaîne de caractères qui représente les permissions.

Exemple (code Java 7) :

```
Path fichier = Paths.get("/home/jm/test.txt");
PosixFileAttributes attrs = Files.readAttributes(fichier, PosixFileAttributes.class);
Set<PosixFilePermission> permissions = attrs.permissions();
System.out.println(PosixFilePermissions.toString(permissions));
```

Inversement, la méthode fromString() permet de renvoyer une collection de permissions à partir de leur représentation sous la forme d'une chaîne de caractères.

Exemple (code Java 7) :

```
Path fichier = Paths.get("/home/jm/test.txt");
Set<PosixFilePermission> perms = PosixFilePermissions.fromString("rw-rw-rw-");
FileAttribute<Set<PosixFilePermission>> attr =
  PosixFilePermissions.asFileAttribute(perms);
Files.createFile(fichier, attr);
```

La méthode setPosixFilePermission(Path, Set<PosixFilePermission>) de la classe Files permet de modifier les permissions sur un élément du système de fichiers dont le chemin est fourni en paramètre sous réserve que les droits actuels sur le fichier le permettent.

Exemple (code Java 7) :

```
Set<PosixFilePermission> permissions = PosixFilePermissions.fromString("rw-rw-r--");
Files.setPosixFilePermissions(fichier, permissions);
```

16.11. La gestion des unités de stockages

Les fichiers et les répertoires contenus dans un système de fichiers sont stockés dans un périphérique de stockage. Ces systèmes de stockages peuvent être des unités physiques sous la forme de disques (disque dur, SSD, ...) ou des unités logiques (partitions sur un disque, ...).

La classe java.nio.file.FileStore encapsule un système de stockage.

Le point d'entrée d'un système de stockage est dépendant du système d'exploitation :

- Sous Windows : c'est un volume désigné par une lettre suivie du caractère « : », les lettres A et B sont réservées aux lecteurs de disquettes, la lettre C est la partition de boot, les autres lettres sont attribuées aux autres partitions, disques ou systèmes de stockage externes
- Sous Unix : c'est un point de montage qui correspond à un répertoire dans le système de fichiers

Pour obtenir une instance de la classe FileStore qui encapsule le système de stockage, il faut utiliser la méthode getFileStore() de la classe Files en lui passant en paramètres une instance de type Path qui encapsule un élément du système de fichiers correspondant au système de stockage.

La méthode getFileStores() de la classe FileSystem permet d'obtenir une instance de type Iterable<FileStore> qui contient tous les systèmes de stockage accessibles.

Exemple (code Java 7) :

```
    Iterable<FileStore> fileStores = FileSystems.getDefault().getFileStores();
    for (FileStore fileStore : fileStores) {
      System.out.println(fileStore);
      System.out.println("name : "+ fileStore.name() + ", type : "
          + fileStore.type());
    }
```

La méthode supportsFileAttributView() permet de vérifier si une vue relative aux méta-données est supportée ou non par le FileStore.

Exemple (code Java 7) :

```
    for (FileStore store : FileSystems.getDefault().getFileStores()) {
      System.out.println(store);
      System.out.println("Support BasicFileAttribute : "
          + store.supportsFileAttributeView(BasicFileAttributeView.class));
      System.out.println("Support DosFileAttribute : "
          + store.supportsFileAttributeView(DosFileAttributeView.class));
      System.out.println("Support PosixFileAttribute : "
          + store.supportsFileAttributeView(PosixFileAttributeView.class));
    }
```

La classe FileStore possède aussi plusieurs méthodes pour obtenir des informations concernant la taille du système de stockage :

- sur l'espace totale avec la méthode getTotalSpace()
- sur l'espace disponible avec la méthode getUsableSpace()
- sur l'espace non alloué avec la méthode getUnallocatedSpace().

Exemple (code Java 7) :

```
    final int UN_GIGA = 1024 * 1024 * 1024;
    for (FileStore store : FileSystems.getDefault().getFileStores()) {
      try {
        long total = store.getTotalSpace() / UN_GIGA;
        long used = (store.getTotalSpace() - store.getUnallocatedSpace()) / UN_GIGA;
        long avail = store.getUsableSpace() / UN_GIGA;
        System.out.format("%-20s total=%5dGo used=%5dGo avail=%5dGo%n", store,
            total, used, avail);
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
```

16.12. Les notifications de changements dans un répertoire

Avant Java 7, pour obtenir des notifications lorsque les éléments d'un répertoire étaient modifiés, il fallait développer son propre mécanisme de polling ou utiliser une bibliothèque comme JPathWatch ou JNotify.

Un polling sur le contenu du répertoire permet de savoir si une modification est intervenue dans les fichiers d'un répertoire : ceci consiste à rechercher des modifications de façon périodique en vérifiant le statut de tous les fichiers du répertoire par rapport à leur précédent état.

Java 7 propose l'API WatchService qui offre cette fonctionnalité en standard : NIO2 propose la classe WatchService qui permet d'obtenir des événements sur des actions réalisées sur un répertoire surveillé du système de fichiers. L'API WatchService est performante mais elle n'est pas récursive.

L'utilisation de l'API WatchService pour obtenir des notifications requiert la mise en oeuvre de plusieurs étapes :

- créer une instance de type WatchService
- enregistrer cette instance auprès du répertoire concerné en précisant le type de notifications auquel on souhaite s'abonner (création, modification, suppression). Un objet de type WatchKey est obtenu suite à cet enregistrement
- utiliser une boucle pour obtenir les événements encapsulés dans un objet de type WatchKey
- utiliser l'objet de type WatchKey : il faut parcourir et traiter les événements qu'il contient
- chaque objet de type WatchKey doit être réinitialisé
- une fois que l'objet WatchService n'est plus utile, il est préférable d'invoquer sa méthode close() pour libérer les ressources natives utilisées

16.12.1. La surveillance d'un répertoire

L'implémentation de la classe WatchService s'appuie généralement sur le mécanisme d'événements sous-jacent du système d'exploitation (ChangeNotification sous Windows, inotify sous Linux, FSEvents sous Mac OS X). Si un tel mécanisme n'existe pas alors l'implémentation va utiliser un mécanisme de polling. Dans tous les cas, cette implémentation est spécifique à chaque JVM et système d'exploitation.

Pour obtenir une instance de type WatchService, il faut invoquer la méthode newWatchService() de la classe FileSystem.

Exemple (code Java 7) :

```
  WatchService watchService = FileSystems.getDefault().newWatchService();
```

Un objet de type WatchService peut s'utiliser sur un objet qui implémente l'interface Watchable. L'interface Path hérite de l'interface Watchable. L'interface Watchable définit deux surcharges de la méthode register() qui attendent en paramètre une instance de type WatchService et les types d'événements qui doivent être capturés.

Il faut donc créer une instance de type Path qui encapsule le chemin du répertoire que l'on souhaite surveiller. La surveillance d'un répertoire se fait en enregistrant l'objet de type WatchService auprès de l'objet de type Path qui encapsule le chemin du répertoire.

Exemple (code Java 7) :

```
      final Path dir = Paths.get("c:/java/test");

      WatchKey key = dir.register(watcher,
         StandardWatchEventKinds.ENTRY_CREATE,
         StandardWatchEventKinds.ENTRY_DELETE,
         StandardWatchEventKinds.ENTRY_MODIFY);
```

La méthode register() attend en paramètre un objet de type WatchService et un ensemble de varargs de type WatchEvent.Kind qui permet de préciser les types d'événements à revecoir. La méthode register() attend donc en paramètre l'instance de type WatchService et accepte plusieurs types événements définis dans la classe java.nio.file.StandardWatchEventKinds.

Les types d'événements concernant les modifications dans un répertoire sont définis dans la classe StandardWatchEventKinds sous la forme de champs statiques de type WatchEvent.Kind<Path>.

WatchEvent.Kind<Path> ENTRY_CREATE	un nouvel élément est créé ou renommé dans le répertoire
WatchEvent.Kind<Path> ENTRY_MODIFY	un élément du répertoire est modifié
WatchEvent.Kind<Path> ENTRY_DELETE	un élément du répertoire est supprimé ou renommé. Les modifications/suppressions du répertoire lui-même ne sont pas concernées
WatchEvent.Kind<Object> OVERFLOW	indique qu'un ou plusieurs événements peuvent avoir été perdus ou manqués

Lors de l'enregistrement d'un répertoire, il faut préciser les types d'événements auxquels on souhaite s'abonner. Les événements de type OVERFLOW sont reçus automatiquement : il n'est pas nécessaire de préciser le type OVERFLOW lors de l'enregistrement.

La méthode register() renvoie un objet de type WatchKey qui encapsule l'enregistrement du chemin avec l'objet de type WatchService.

L'interface WatchKey définit les méthodes d'un jeton qui représente l'enregistrement d'un objet WatchService sur un objet de type Watchable.

Un objet de type WatchKey reste valide jusqu'à ce que :

- Il soit annulé en invoquant sa méthode cancel()
- L'objet de type Watchable n'existe plus
- La méthode close() de l'objet WatchService() est invoquée

Un objet de type WatchKey possède un état qui peut prendre plusieurs valeurs :

- ready : l'objet peut recevoir de nouveaux événements. C'est l'état de l'objet lors de sa création
- signaled : l'objet possède un ou plusieurs événements à traiter. Pour revenir à l'état ready, il faut invoquer la méthode reset()
- invalid : l'objet n'est plus actif. Cet état est obtenu en invoquant sa méthode cancel(), en invoquant la méthode close() de l'objet de type WatchService ou si le répertoire n'est plus accessible

Un objet de type WatchKey encapsule le résultat de l'enregistrement du WatchService sur l'objet de type Path.

Après l'enregistrement, l'objet de type WatchKey est dans l'état ready et y reste jusqu'à ce que :

- la méthode cancel() de l'objet WatchKey soit invoquée
- la méthode close() de l'objet WatchService soit invoquée
- le répertoire n'est plus accessible

Les objets de type WatchKey sont thread-safe.

Pour arrêter l'émission des événements, il faut invoquer la méthode cancel() de la classe WatchKey ou la méthode close() de la classe WatchService.

La méthode watchable() de la classe WatchKey renvoie un objet de type Path qui encapsule le chemin du répertoire sur lequel l'abonnement aux notifications a été réalisé.

16.12.2. L'obtention des événements

La réception des événements ne se fait pas par un mécanisme asynchrone comme enregistrer un callback de type listener : il est nécessaire de créer son propre polling pour obtenir les événements.

Le traitement des événements doit ainsi se faire dans un thread dédié pour ne pas bloquer le thread courant.

Lorsqu'un changement est détecté, l'état de l'objet WatchKey passe à signaled. Pour obtenir le ou les événements non traités liés à ces changements, il faut invoquer la méthode poll() ou take() de l'objet WatchService :

Méthode	Rôle
poll()	Retourne le prochain WatchKey ou null si aucun n'est présent
poll(long timeout, TimeUnit unit)	Retourne le prochain WatchKey en attendant le temps fourni en paramètre sous la forme d'une durée et d'une unité sinon retourne null
take()	Retourne le prochain WatchKey en attendant indéfiniment jusqu'à ce qu'un ou plusieurs événements soient disponibles

Il faut utiliser une boucle qui invoque l'une de ces méthodes pour obtenir les événements à traiter.

Exemple (code Java 7) :

```
      while(true) {
         WatchKey key = watchService.take();
         // ...
      }
```

Si un changement est détecté dans un ou plusieurs éléments du répertoire, alors l'état de l'instance du WatchKey passe à « signaled » et l'événement est mis dans une queue pour traitement.

Exemple (code Java 7) :

```
      boolean running = true;
      // ...
      while (running) {
          try {
              // key = watcher.take();
              key = watcher.poll(1000, TimeUnit.MILLISECONDS);

              if (key != null) {
                  for (final WatchEvent<?> event : key.pollEvents()) {
                      final Path name = (Path) event.context();
                      System.out.format(event.kind() + " " + "%s\n", name);
                  }
                  key.reset();
              }
          } catch (final InterruptedException e) {
              e.printStackTrace();
          }
      }
```

La méthode pollEvents() de l'interface WatchKey permet d'obtenir tous les événements qui sont stockés dans l'objet.

Il est important d'invoquer la méthode reset() de l'interface WatchKey pour permettre de remettre son état à ready : elle renvoie un booléen qui précise si l'objet de type WatchKey est toujours valide et actif. L'invocation de la méthode reset() sur un objet de type WatchKey annulé ou déjà dans l'état ready n'a aucun effet.

Attention : lorsqu'un événement est reçu, il n'y a aucune garantie que l'opération qui est à l'orgine de l'événement soit terminée.

16.12.3. Le traitement des événements

Le ou les événements sont encapsulés dans un objet qui implémente l'interface WatchKey. Pour obtenir les événements, il faut invoquer la méthode pollEvents() de la classe WatchKey qui renvoie une collection de type List<WatchEvent< ?>>. Cette méthode supprime de l'objet WatchKey les événements qu'elle renvoie.

Il faut itérer sur les éléments de la collection pour traiter chacun des événements encapsulés.

Un objet de type WatchEvent<?> est typé avec un type qui sera utilisé comme contexte de l'événement.

Exemple (code Java 7) :

```
WatchEvent<Path> evenement = (WatchEvent<Path>) event;
Path chemin = evenement.context();
```

Un événement obtenu par un objet de type WatchService est encapsulé dans un objet qui implémente l'interface WatchEvent<T> qui possède trois méthodes :

Méthode	Rôle
T context()	Renvoyer le contexte de l'événement
int count()	Retourner le nombre d'occurences de l'événement
WatchEvent.Kind<T> kind	Renvoyer le type de l'événement

Pour chaque événement à traiter, il est possible de connaître :

- le type de l'événement en invoquant la méthode kind() de l'objet de type WatchEvent
- en invoquant la méthode context() de l'objet de type WatchEvent, le chemin relatif au répertoire enregistré qui est encapsulé dans le Path sur lequel l'événement (création, suppression ou mise à jour) a eu lieu
- le chemin du répertoire concerné (c'est notamment pratique si plusieurs répertoires ont été enregistrés) en invoquant la méthode watchable() de l'objet de type WatchKey

La méthode kind() permet d'obtenir le type de l'événement sous la forme d'une interface de type WatchEvent.Kind<T>.

La méthode count() permet de savoir combien de fois l'événement a été émis.

La méthode context() permet de renvoyer un objet qui encapsule le contexte associé à l'événement.

Exemple (code Java 7) :

```
    for (final WatchEvent<?> event : key.pollEvents()) {
      final Path name = (Path) event.context();
      System.out.format(event.kind() + " " + "%s\n", name);
    }
    key.reset();
```

Attention : une fois que les événements ont été traités, il est important de remettre l'objet de type WatchKey dans l'état ready en invoquant sa méthode reset(). Si la méthode reset() renvoie false, alors l'objet de type WatchKey n'est plus valide et il faut donc interrompre les traitements d'écoute des événements.

16.12.4. Un exemple complet

Cette section propose un exemple complet de mise en oeuvre de l'API WatchService.

Exemple (code Java 7) :

```
package fr.jmdoudoux.dej.nio2;

import static java.nio.file.StandardCopyOption.ATOMIC_MOVE;
import static java.nio.file.StandardCopyOption.COPY_ATTRIBUTES;
import static java.nio.file.StandardCopyOption.REPLACE_EXISTING;

import java.io.IOException;
import java.nio.file.FileSystems;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardWatchEventKinds;
import java.nio.file.WatchEvent;
import java.nio.file.WatchKey;
import java.nio.file.WatchService;
import java.util.concurrent.TimeUnit;

public class TestWatcherService {

  public static void main(final String[] args) {
    final Path source = Paths.get("c:/java/test/fichier.txt");
    final Path copie = Paths.get("c:/java/test/fichier_copie.txt");
    final Path renomme = Paths.get("c:/java/test/fichier_nouveau.txt");
    final MonWatcher monWatcher = new MonWatcher();
    monWatcher.start();

    try {
      Thread.sleep(1000);
      System.out.println("Copie " + source + " -> " + copie);
      Files.copy(source, copie, REPLACE_EXISTING, COPY_ATTRIBUTES);

      Thread.sleep(2000);
      System.out.println("Deplacement " + copie + " -> " + renomme);
      Files.move(copie, renomme, REPLACE_EXISTING, ATOMIC_MOVE);

      Thread.sleep(2000);
      System.out.println("Supression fichier " + renomme);
      Files.deleteIfExists(renomme);

      Thread.sleep(5000);
    } catch (final IOException ioe) {
      ioe.printStackTrace();
    } catch (final InterruptedException e) {
      e.printStackTrace();
    }

    monWatcher.setRunning(false);
  }
}

class MonWatcher extends Thread {

  private boolean running = true;

  public boolean isRunning() {
    return running;
  }

  public void setRunning(final boolean running) {
    this.running = running;
  }

  @Override
  public void run() {
    WatchService watcher;
    try {
      watcher = FileSystems.getDefault().newWatchService();
      final Path dir = Paths.get("c:/java/test");

      WatchKey key = dir.register(watcher,
          StandardWatchEventKinds.ENTRY_CREATE,
          StandardWatchEventKinds.ENTRY_DELETE,
          StandardWatchEventKinds.ENTRY_MODIFY);

      while (running) {
        try {
          // key = watcher.take();
          key = watcher.poll(1000, TimeUnit.MILLISECONDS);
        } catch (final InterruptedException e) {
          e.printStackTrace();
        }

        if (key != null) {
          for (final WatchEvent<?> event : key.pollEvents()) {
            final Path name = (Path) event.context();
            System.out.format(event.kind() + " " + "%s\n", name);
          }
          boolean reset = key.reset();
          if (!reset) {
            running = false;
          }
        }
      }
    } catch (final IOException ioe) {
      ioe.printStackTrace();
    }
  }
}
```

Résultat :

```
Copie c:\java\test\fichier.txt -> c:\java\test\fichier_copie.txt
ENTRY_CREATE fichier_copie.txt
ENTRY_MODIFY fichier_copie.txt
Deplacement c:\java\test\fichier_copie.txt -> c:\java\test\fichier_nouveau.txt
ENTRY_MODIFY fichier_copie.txt
ENTRY_DELETE fichier_copie.txt
ENTRY_CREATE fichier_nouveau.txt
ENTRY_MODIFY fichier_nouveau.txt
Supression fichier c:\java\test\fichier_nouveau.txt
ENTRY_DELETE fichier_nouveau.txt
```

16.12.5. L'utilisation et les limites de l'API WatchService

L'API WatchService permet d'être notifié des changements qui surviennent sur les éléments d'une entité, par exemple sur un répertoire d'un système de fichiers.

Cette fonctionnalité est intéressante mais elle présente quelques limites qu'il est important de connaitre :

- Aucun événements n'est émis concernant les sous-répertoires du répertoire observé : dans ce cas, il faut parcourir les sous-répertoires et enregistrer un objet de type WatchService sur chacun d'entre-eux.
- Les performances et l'ordre des événements sont dépendants de l'implémentation.
- Lorsqu'un evénement est reçu, il n'y a pas de garantie que les traitements à l'origine de l'événement soient terminés.

16.13. La gestion des erreurs et la libération des ressources

Lors d'opérations d'entrées-sorties de nombreuses erreurs inattendues peuvent survenir, par exemple un fichier qui n'existe pas, un manque de droit d'accès, une erreur de lecture, ...

Toutes ces erreurs sont encapsulées dans une exception de type IOException ou d'un de ses sous-types. Toutes les méthodes qui réalisent des opérations d'entrées-sorties peuvent lever ces exceptions.

Avant Java 7, les opérations de type I/O devaient être utilisées dans un bloc try et les exceptions pouvant être levées, traitées dans des blocs catch. La fermeture des flux devait être assurée dans un bloc finally pour garantir son exécution dans tous les cas.

Exemple :

```
Charset charset = Charset.forName("UTF-8");
String contenu = "Bonjour";
BufferedWriter writer = null;
try {
    writer = Files.newBufferedWriter(file, charset);
    writer.write(contenu, 0, contenu.length());
} catch (IOException ioe) {
    ioe.printStackTrace();
} finally {
    if (writer != null) {
        writer.close();
    }
}
```

A partir de Java SE 7, il est préférable d'utiliser l'opérateur try-with-ressources pour assurer la libération automatique des ressources et la gestion des exceptions.

Exemple (code Java 7) :

```
      Charset charset = Charset.forName("UTF-8");
      String contenu = "Bonjour";
      try (BufferedWriter writer = Files.newBufferedWriter(file, charset)) {
          writer.write(contenu, 0, contenu.length());
      } catch (IOException ioe) {
          ioe.printStackTrace();
      }
```

Plusieurs exceptions héritent de l'exception FileSystemException qui hérite elle-même de l'exception IOException.

La classe FileSystemException encapsule plusieurs attributs qui sont des chaînes de caractères:

- file : le nom du fichier impliqué
- message : un message détaillé sur l'exception
- reason : la raison pour laquelle l'opération a échoué
- otherFile : renvoie le nom d'un second fichier impliqué

Exemple (code Java 7) :

```
  public static void copierFichier() {
    Path source = Paths.get("c:/java/test/monfichier.txt");
    Path cible = Paths.get("c:/java/test/monfichier_copie.txt");
    try {
      Files.copy(source, cible);
    } catch (FileAlreadyExistsException e) {
      System.err.format("Copie impossible : le fichier %s existe déjà", e.getFile());
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
```

Résultat :

```
Copie impossible : le fichier
c:\java\test\monfichier_copie.txt existe déjà
```

De nombreuses ressources utilisées par l'API NIO 2 telles que les channels ou les flux implémentent l'interface java.io.Closeable. Ceci permet leur prise en compte par l'opérateur try-with-ressource qui invoque leur méthode close() et libère ainsi les ressources devenues inutiles.

Java 7 propose une fonctionnalité nommée Automatic Resource Management ou ARM. L'ARM propose de réduire la quantité de code à produire par le développeur pour gérer une ressource et surtout pour libérer les ressources qui lui sont associées.

Des langages comme C, C++ ou Delphi, offrent aux développeurs un contrôle total sur l'allocation et la désallocation mémoire des objets créés en utilisant des opérateurs comme malloc, free, new, delete, ...

Contrairement à eux, Java ne propose pas de contrôle sur le processus de désallocation des ressources d'un objet. La JVM propose un mécanisme nommé garbage collection ou ramasse-miettes qui assure la libération des ressources mémoires des objets qui ne sont plus utilisés.

Il est possible de demander à la JVM de forcer l'exécution du ramasse-miettes en utilisant les méthodes System.gc() ou Runtime.getRuntime.gc() : ce ne sont que des suggestions de demandes que la JVM n'est pas obligée de suivre.

Il n'est pas recommandé d'utiliser ces méthodes dans son code et dans tous les cas la logique des traitements ne doit pas reposer sur ces méthodes.

La gestion de la mémoire par la JVM, notamment grâce au garbage collector, a grandement amélioré la productivité des développeurs et la fiabilité des applications. Cependant le ramasse-miettes n'est pas capable de faire seul la libération des ressources notamment dans le cas de ressources natives fournies par le système d'exploitation sous-jacent de la JVM. Ce type de ressources doit être libéré explicitement par le développeur qui doit invoquer la méthode adéquate généralement dans un bloc try/finally.

Exemple :

```
package fr.jmdoudoux.dej.java7;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;

public class MaClasse {
  public MaClasse() {
  }

  public static void main(String[] args) {
    InputStream file = null;
    try {
      file = new FileInputStream(new File("test.bin"));
      byte fileContent[] = new byte[(int) file.available()];
      file.read(fileContent);
    } catch (IOException ioe) {
      ioe.printStackTrace();
    } finally {
      try {
        file.close();
      } catch (IOException ioe) {
        // traitement de l'exception au besoin
      }
    }
  }
}
```

L'invocation de la méthode close() dans la clause finally doit être faite dans un bloc try/catch car elle peut lever une exception de type IOException.

Il est possible de déclarer plusieurs ressources dans un bloc try : leurs méthodes close() seront toutes invoquées même si une de ces invocations lève une exception durant sont exécution.

Si une exception de type IOException est levée dans les traitements du bloc try et une autre dans le bloc finally générée par le compilateur pour fermer les ressources, c'est toujours l'exception du bloc try qui sera propagée.

Il est possible d'avoir des informations sur l'exception masquée en utilisant la méthode getSuppressed() de la classe Throwable.

Il n'est généralement pas pratique d'utiliser en même temps les instructions catch et finally avec l'instruction try. Il est préférable d'utiliser simplement un bloc finally avec le try et de laisser la gestion des exceptions à un niveau supérieur.

Exemple :

```
  public void maMethode() throws IOException {
    try {
      // ...
    } finally {
      // ...
    }
  }
```

16.14. L'interopérabilité avec le code existant

Les objets de type Path obtenus sur le système de fichiers par défaut sont interopérables avec des objets de type java.io.File. Les objets de type Path obtenus sur d'autres systèmes de fichiers peuvent ne pas être interopérables.

Pour faciliter le portage de code utilisant l'API java.io vers NIO2, la classe java.io.File propose la méthode toPath() qui crée une instance de type Path à partir des informations encapsulées dans l'instance de type File.

Exemple (code Java 7) :

```
Path input = file.toPath();
```

Il est ainsi facile de bénéficier des fonctionnalités offertes par NIO2 sans avoir à tout réécrire.

Exemple :

```
file.delete();
```

Il est possible de réécrire cette portion de code en utilisant NIO2.

Exemple (code Java 7) :

```
Path fp = file.toPath();
Files.delete(fp);
```

Inversement, la classe Path propose la méthode toFile() permettant de créer une instance de la classe java.io.File qui correspond aux informations encapsulées dans l'instance de type Path.

16.14.1. L'équivalence des fonctionnalités entre java.io et NIO2

Comme l'API NIO2 est une nouvelle API, il n'y a pas de correspondance directe entre les deux API mais le tableau ci-dessous fournit un résumé de l'équivalence des principales fonctionnalités.

Fonctionnalité	java.io	NIO 2
Encapsuler un chemin	java.io.File	java.nio.file.Path
Vérifier les permissions	File.canRead(), File.canCrite() et File.canExecute()	Files.isReadable(), Files.isWritable() et Files.isExecutable().
Vérifier le type d'élément	File.isDirectory(), File.isFile()	Files.isDirectory(Path, LinkOption...), Files.isRegularFile(Path, LinkOption...),
Taille d'un fichier	File.length()	Files.size(Path)
Obtenir ou modifier la date de dernière mise à jour	File.lastModified() , File.setLastModified(long)	Files.getLastModifiedTime(Path, LinkOption...), Files.setLastModifiedTime(Path, FileTime)
Modifier les attributs	File.setExecutable(), File.setReadable(), File.setReadOnly(), File.setWritable()	Files.setAttribute(Path, String, Object, LinkOption...)
Déplacer un fichier	File.renameTo()	Files.move()
Supprimer un fichier	File.delete()	Files.delete()
Créer un fichier	File.createNewFile()	Files.createFile()
	File.deleteOnExit()	Option DELETE_ON_CLOSE à utiliser sur la méthode createFile()
Créer un fichier temporaire	File.createTempFile()	Files.createTempFile(Path, String, FileAttributes<?>), Files.createTempFile(Path,

		String, String, FileAttributes<?>)
Tester l'existence d'un fichier	File.exists	Files.exists() ou Files.notExists()
Obtenir le chemin absolu	File.getAbsolutePath() ou File.getAbsoluteFile()	Path.toAbsolutePath()
	File.getCanonicalPath() ou File.getCanonicalFile()	Path.toRealPath() ou Path.normalize()
Convertir en URI	File.toURI()	Path.toURI()
L'élément est-il caché ?	File.isHidden()	Files.isHidden()
Obtenir le contenu d'un répertoire	File.list() ou File.listFiles()	Path.newDirectoryStream()
Créer un répertoire	File.mkdir() ou File.mkdirs()	Path.createDirectory()
Obtenir le contenu du répertoire racine	File.listRoots()	FileSystem.getRootDirectories()
	File.getTotalSpace()	FileStore.getTotalSpace()
	File.getFreeSpace()	FileStore.getUnallocatedSpace()
	File.getUsableSpace()	FileStore.getUsableSpace()

Il est possible d'obtenir plus de détails à l'url :

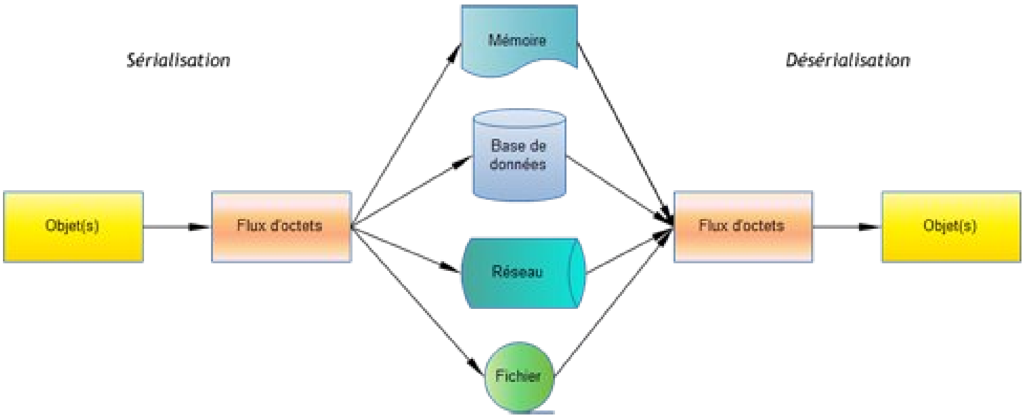
https://docs.oracle.com/javase/tutorial/essential/io/legacy.html

17. La sérialisation

Chapitre 17

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

La sérialisation est un procédé introduit dans le JDK version 1.1 qui permet de rendre un objet ou un graphe d'objets de la JVM persistant pour stockage ou échange et vice versa. Cet objet est mis sous une forme sous laquelle il pourra être reconstitué à l'identique. Ainsi il pourra être stocké sur un disque dur ou transmis au travers d'un réseau pour le créer dans une autre JVM. C'est le procédé qui est utilisé, par exemple, par RMI. La sérialisation est aussi utilisée par les beans pour sauvegarder leurs états.



Au travers de ce mécanisme, Java fournit une façon facile, transparente et standard de réaliser cette opération : ceci permet de facilement mettre en place un mécanisme de persistance. Il est de ce fait inutile de créer un format particulier pour sauvegarder et relire un objet. Le format utilisé est indépendant du système d'exploitation. Ainsi, un objet sérialisé sur un système peut être réutilisé par un autre système pour récréer l'objet.

La sérialisation peut s'appliquer facilement à quasiment tous les objets. Cependant, toutes les classes du JDK ne sont pas sérialisables : notamment les classes qui sont liées à des éléments du système ne le sont pas (Thread, OutputStream et ses sous-classes, Socket, Image, ...)

L'implémentation de son propre mécanisme de sérialisation est probablement complexe et coûteux. A contrario, le mécanisme de sérialisation proposé par défaut en Java est relativement simple à mettre en oeuvre. Cette relative simplicité de mise en oeuvre permet d'utiliser la sérialisation pour différents besoins :

- stocker des objets momentanément inutilisés (généralement la sérialisation est utilisée par les serveurs d'applications Java EE lors de passivation d'un EJB par exemple)
- sauvegarder une configuration
- échanger des objets entre plusieurs JVM
- rendre persistants des objets (dans un fichier ou une base de données par exemple)
- créer une copie intégrale d'un objet (deep copy)

La sérialisation n'est cependant pas spécialement conçue pour persister un objet durant une longue période même si cela peut être fait : au contraire elle est plutôt conçue pour être utilisée sur des périodes temporelles courtes (échange réseau, mise en cache, persistance temporaire, ...)

Différentes solutions sont proposées pour sérialiser/désérialiser un objet.

Ce chapitre contient plusieurs sections :

- ♦ La sérialisation standard
- ♦ La documentation d'une classe sérialisable
- ♦ La sérialisation et la sécurité
- ♦ La sérialisation en XML

17.1. La sérialisation standard

La sérialisation d'un objet permet d'envoyer dans un flux les informations sur la classe et l'état d'un objet pour permettre de le récréer ultérieurement. Ces informations permettent de restaurer l'état de l'objet même dans une classe différente qui dans ce cas doit être compatible. Elle permet donc de transformer l'état d'un objet pour permettre sa persistance en dehors de la JVM ou de l'échanger en utilisant le réseau.

L'opération inverse qui consiste à créer une nouvelle instance à partir du résultat d'une sérialisation s'appelle la désérialisation.

La sérialisation doit faire face à plusieurs problématiques notamment :

- la gestion des versions des objets
- la portabilité du résultat de la sérialisation
- les objets sérialisés font généralement référence à d'autres objets qui doivent aussi être sérialisés en même temps pour permettre de maintenir l'état de leurs relations.
- la gestion des références dans un graphe d'objets sérialisés surtout si un même objet est référencé plusieurs fois dans le graphe
- elle est généralement peut performante car elle utilise l'introspection
- certains objets ne peuvent pas être sérialisés notamment ceux dépendants du système d'exploitation : threads, fichiers, ...
- son utilisation peut engendrer des problèmes de sécurité : sérialisation de données sensibles, désérialisation d'un objet dont la source est inconnue, ...

Il existe plusieurs formats de sérialisation appartenant à deux grandes familles :

- formats binaires : c'est le format par défaut
- formats textes : ils sont plus portables car ils utilisent généralement une structuration standard (XML, JSON, ...), peuvent être facilement modifiés et consomment plus de ressources pour être traités

L'ajout d'un attribut à l'objet est automatiquement pris en compte lors de la sérialisation. Attention toutefois, la désérialisation de l'objet doit se faire avec la classe qui a été utilisée pour la sérialisation ou une classe compatible.

Tous les objets ne sont pas sérialisables : généralement ce sont des objets qui ont des références sur des éléments du système d'exploitation (threads, fichiers, ...).

Le mécanisme de sérialisation par défaut ignore les champs static ou transient.

La sérialisation utilise un mécanisme qui ne sérialise qu'une fois un objet même si le graphe d'objets à sérialiser possède plusieurs références sur celui-ci. Ceci permet lors de la désérialisation que toutes les références sur l'objet dans le graphe pointent bien sur la bonne instance.

Pour pouvoir être sérialisée, une classe doit implémenter l'interface java.io.Serializable ou l'interface java.io.Externalizable

17.1.1. La sérialisation binaire

La sérialisation binaire est une fonctionnalité introduite à partir de Java 1.1. Le format de la sérialisation est spécifique à la JVM mais il ne dépend pas du système d'exploitation.

La sérialisation de membres de types primitifs est facile puisqu'il suffit de prendre leur valeur. Lorsque le membre est un objet c'est plus compliqué car il n'est pas possible de simplement prendre la référence. Lorsque l'objet sera désérialisé, les références vont changer : sérialiser la référence est inutile dans la mesure où la référence n'a de sens que dans le contexte d'exécution d'une seule instance d'une JVM. Il est donc nécessaire de sérialiser récursivement chaque objet du graphe.

Une énumération est sérialisée uniquement sous la forme de son nom.

Plusieurs éléments de la plate-forme Java agissent durant la mise en oeuvre de la sérialisation :

- le mot clé transient
- l'interface Serializable qui est un marqueur indiquant que la classe peut être sérialisée
- la classe ObjectOutputStream qui permet de sérialiser un objet
- la classe ObjectInputStream qui permet de désérialiser un objet précédemment sérialisé
- l'interface Externalizable dont l'implémentation des méthodes permet de gérer finement la sérialisation et la désérialisation d'une classe en implémentant son propre mécanisme.

Le choix de rendre une classe sérialisable est facile à court terme par contre cela peut impliquer des difficultés à long terme notamment en limitant les possibilités de changement dans la classe. La sérialisation de la classe devient publique comme son interface : l'ajout ou la suppression d'un champ peut modifier le comportement du mécanisme de sérialisation par défaut.

Par exemple, par défaut si une classe Serializable ne déclare pas explicitement un champ serialVersionUID alors le compilateur va l'ajouter et calculer une valeur qui prend en compte les différents éléments qui composent la classe.

Exemple :

```
import java.io.Serializable;

public class MonBean implements Serializable {

  private String champ1;

  public String getChamp1() {
    return this.champ1;
  }

  public void setChamp1(final String champ1) {
    this.champ1 = champ1;
  }

  @Override
  public String toString() {
    return "MonBean [champ1=" + this.champ1 + "]";
  }
}
```

Si la classe est modifiée, le champ serialVersionUID sera recalculé avec une version différente. Les données des sérialisations précédentes lèveront une InvalidClassException

Exemple :

```
import java.io.Serializable;

public class MonBean implements Serializable {

  private String champ1;
  private String champ2;

  public String getChamp1() {
    return this.champ1;
  }

  public String getChamp2() {
    return this.champ2;
  }

  public void setChamp2(final String champ2) {
    this.champ2 = champ2;
  }

  public void setChamp1(final String champ1) {
    this.champ1 = champ1;
  }

  @Override
  public String toString() {
    return "MonBean [champ1=" + this.champ1 + ", champ2=" + this.champ2 + "]";
  }
}
```

Résultat :

```
java.io.InvalidClassException: MonBean; local class incompatible: stream classdesc
serialVersionUID = -386157042668049345, local class serialVersionUID = -608800936009283713
        at java.io.ObjectStreamClass.initNonProxy(ObjectStreamClass.java:562)
        at java.io.ObjectInputStream.readNonProxyDesc(ObjectInputStream.java:1582)
        at java.io.ObjectInputStream.readClassDesc(ObjectInputStream.java:1495)
        at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1731)
        at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1328)
        at java.io.ObjectInputStream.readObject(ObjectInputStream.java:350)
        at SerDeserMonBean.main(SerDeserMonBean.java:22)
```

La sérialisation d'une classe interne n'est pas recommandée car elle possède une référence implicite sur son instance englobante. Cette référence est créée par le compilateur de manière dépendante de l'implémentation par son fournisseur. Ceci peut nuire à la portabilité. De plus, comme les classes internes ne peuvent pas avoir de contructeur par défaut, elles ne peuvent pas implémenter l'interface Externalizable.

17.1.1.1. L'interface Serializable

Cette interface ne définit aucune méthode mais permet simplement de marquer une classe comme pouvant être sérialisée.

Tout objet qui doit être sérialisé grâce au mécanisme par défaut doit implémenter cette interface ou une de ses classes mères doit l'implémenter.

Si l'on tente de sérialiser un objet qui n'implémente pas l'interface Serializable, une exception java.io.NotSerializableException est levée.

La classe ci-dessous sera utilisée dans certains exemples de ce chapitre.

Exemple :

```
public class Personne implements java.io.Serializable {
  private String nom    = "";
  private String prenom = "";
  private int    taille = 0;

  public Personne(final String nom, final String prenom, final int taille) {
    this.nom = nom;
    this.taille = taille;
    this.prenom = prenom;
  }

  public String getNom() {
    return this.nom;
  }

  public void setNom(final String nom) {
    this.nom = nom;
  }

  public int getTaille() {
    return this.taille;
  }
  public void setTaille(final int taille) {
    this.taille = taille;
  }

  public String getPrenom() {
    return this.prenom;
  }

  public void setPrenom(final String prenom) {
    this.prenom = prenom;
  }
}
```

Tous les objets du graphe doivent être sérializable. Si ce n'est pas le cas, plusieurs solutions sont possibles :

- créer une classe fille qui implémente l'interface Serializable et utiliser cette sous-classe. Ce n'est pas toujours possible notamment si la classe est déclarée finale.
- marquer le membre avec le mot clé transient pour qu'il soit ignoré lors de la sérialisation. Si l'instance est requise, il est possible d'utiliser les méthodes readObject() et writeObject() pour respectivement sérialiser l'état de l'instance et recréer une instance lors de la désérialisation manuellement

Tous les objets ne peuvent pas être sérialisés : c'est notamment le cas de certains objets qui sont dépendants du système (exemple : les threads, les flux, ...). Ceci explique pourquoi la classe Object n'implémente pas l'interface Serializable.

17.1.1.2. La classe ObjectOutputStream

La classe ObjectOutputStream permet de sérialiser un objet ou un graphe d'objets. Cette sérialisation parcours le graphe d'objets pour les sérialiser les uns après les autres en tenant compte des éventuelles références déjà sérialisées. Par défaut, chaque objet qui est référencé par l'objet sérialisé est aussi sérialisé.

Elle implémente plusieurs interfaces : Closeable (depuis Java 5), DataOutput, Flushable (depuis Java 5), ObjectOutput, ObjectStreamConstants et AutoCloseable (depuis Java 7).

Elle ne possède qu'un constructeur public :

Constructeur	Rôle
ObjectOutputStream(OutputStream out)	Créer une instance qui va écrire le résultat de la sérialisation dans le flux fourni en paramètre

Le constructeur de la classe ObjectOutputStream attend en paramètre un flux de type OutputStream dans lequel les données de la sérialisation seront envoyées.

Exemple :

```
    ObjectOutputStream oos = null;

    try {
      final FileOutputStream fichier = new FileOutputStream("mon_objet.ser");
      oos = new ObjectOutputStream(fichier);
      // ...
    } catch (final java.io.IOException e) {
      e.printStackTrace();
    } finally {
      try {
        if (oos != null) {
          oos.flush();
          oos.close();
        }
      } catch (final IOException ex) {
        ex.printStackTrace();
        }
    }
```

La classe ObjectOutputStream propose de nombreuses méthodes pour permettre d'ajouter au flux de la sérialisation des objets sérialisables, des types primitifs, des tableaux, des chaînes de caractères, ... :

Méthode	Rôle
public final void writeObject(Object obj)	Sérialiser un objet
public void writeUnshared(Object obj)	
public void defaultWriteObject()	Sérialiser un objet en utilisant les règles par défaut. Ne peut être invoquée que dans le corps de la méthode writeObject() sinon une exception de type NotActiveException est levée
public PutField putFields()	Obtenir les différents champs qui seront inclus dans le flux
public writeFields()	
public void reset()	Réinitialiser l'état des objets utilisés par la classe
protected void annotateClass(Class cl)	
protected void writeClassDescriptor(ObjectStreamClass desc)	Ecrire dans le flux une représentation de l' ObjectStreamClass fournie en paramètre
protected Object replaceObject(Object obj)	Permettre une substitution de l'objet par une autre instance de remplacement
protected boolean enableReplaceObject(boolean enable)	Activer la possibilité de remplacer l'objet écrit par une autre instance
protected void writeStreamHeader()	Ecrire les premiers octets du flux notamment la valeur du magic stream et le numéro de version du protocole
public void write(int data)	Ecrire un octet
public void write(byte b[])	Ecrire un ensemble d'octets
public void write(byte b[], int off, int len)	Ecrire un ensemble d'octets
public void flush()	Vider le tampon
protected void drain()	Similaire à flush mais les actions ne vont pas jusqu'au système
public void close()	Fermer le flux
public void writeBoolean(boolean data) public void writeByte(int data) public void writeShort(int data) public void writeChar(int data) public void writeInt(int data) public void writeLong(long data) public void writeFloat(float data) public void writeDouble(double data) public void writeBytes(String data) public void writeChars(String data)	Ecrire une donnée primitive dans le bloc de données (block data)
public void writeUTF(String data)	Ecrire une chaîne de caractères encodée en UTF-8 modifié
public void useProtocolVersion(int version)	Préciser le numéro de version du protocole de sérialisation utilisé
protected writeObjectOverride()	

La méthode writeObject() sérialise le graphe d'objets dont l'objet racine est fourni en paramètre.

Chaque classe fille d'une classe sérialisable peut redéfinir la méthode writeObject() : dans ce cas, les traitements de la méthode ne doivent généralement concerner que les champs de la classe elle-même.

Pour être sérialisée, la classe d'un objet doit implémenter l'interface Serializable.

Par défaut, le mécanisme de sérialisation d'un objet écrit dans le flux binaire :

- la classe de l'objet
- les valeurs des champs

Par défaut, tous les champs d'un objet sont sérialisés sauf :

- ceux qui sont déclarés static
- ceux qui sont déclarés avec le mot clé transient

Tous les champs doivent pouvoir être sérialisés (si le type d'un champ est une classe, celle-ci doit implémenter l'interface Serializable). Les champs qui ne peuvent pas être sérialisés doivent être marqués avec le mot clé transient.

Exemple :

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class SerializerPersonne {

  public static void main(final String argv[]) {
    final Personne personne = new Personne("Dupond", "Jean", 175);
    ObjectOutputStream oos = null;

    try {
      final FileOutputStream fichier = new FileOutputStream("personne.ser");
      oos = new ObjectOutputStream(fichier);
      oos.writeObject(personne);
      oos.flush();
    } catch (final java.io.IOException e) {
      e.printStackTrace();
    } finally {
      try {
        if (oos != null) {
          oos.flush();
          oos.close();
        }
      } catch (final IOException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

On définit un fichier avec la classe FileOutputStream. On instancie un objet de la classe ObjectOutputStream en lui fournissant en paramètre le fichier : ainsi, le résultat de la sérialisation sera envoyé dans le fichier.

On appelle la méthode writeObject() en lui passant en paramètre l'objet à sérialiser. On appelle la méthode flush() pour vider le tampon dans le fichier et la méthode close() pour terminer l'opération.

Lors de ces opérations une exception de type IOException peut être levée si un problème intervient avec le fichier.

Après l'exécution de cet exemple, un fichier nommé « personne.ser » est créé. On peut visualiser son contenu mais surtout ne pas le modifier car sinon il serait corrompu. En effet, les données contenues dans ce fichier ne sont pas toutes au format caractères.

La classe ObjectOutputStream contient aussi plusieurs méthodes qui permettent de sérialiser des types élémentaires : writeInt(), writeDouble(), writeFloat(), ...

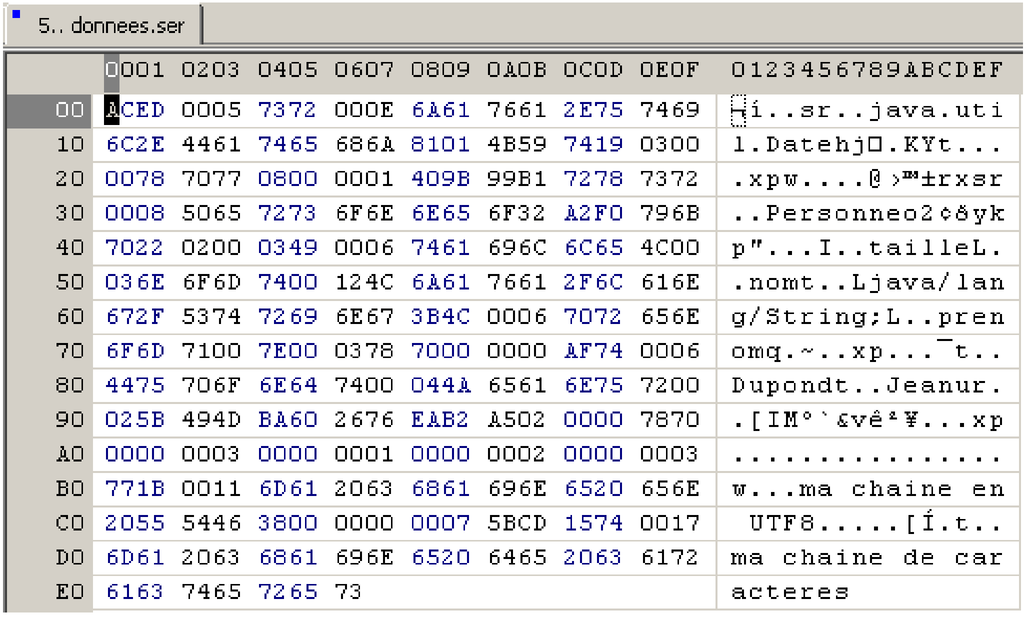
Il est possible dans un même flux d'écrire plusieurs objets les uns à la suite des autres. Ainsi plusieurs objets peuvent être sérialisés. Dans ce cas, il faut faire attention de relire les objets dans leur ordre d'écriture.

Exemple :

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class SerializerDonnees {

  public static void main(final String argv[]) {
    final Personne personne = new Personne("Dupond", "Jean", 175, "1234");
    ObjectOutputStream oos = null;

    try {
      final FileOutputStream fichier = new FileOutputStream("donnees.ser");
      oos = new ObjectOutputStream(fichier);
      oos.writeObject(new java.util.Date());
      oos.writeObject(personne);
      final int[] tableau = { 1, 2, 3 };
      oos.writeObject(tableau);
      oos.writeUTF("ma chaine en UTF8");
      oos.writeLong(123456789);
      oos.writeObject("ma chaine de caracteres");

      oos.flush();
    } catch (final java.io.IOException e) {
      e.printStackTrace();
    } finally {
      try {
        if (oos != null) {
          oos.flush();
          oos.close();
        }
      } catch (final IOException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

Certaines informations apparaissent en clair dans le fichier contenant les données de la sérialisation visualisé dans un éditeur hexadécimal :

- Le type des objets sérialisés
- Les chaînes de caractères



```
    0001 0203 0405 0607 0809 0A0B 0C0D 0E0F  0123456789ABCDEF
00  ACED 0005 7372 000E 6A61 7661 2E75 7469  □i..sr..java.uti
10  6C2E 4461 7465 686A 8101 4B59 7419 0300  l.Datehj□.KYt...
20  0078 7077 0800 0001 409B 99B1 7278 7372  .xpw....@›™±rxsr
30  0008 5065 7273 6F6E 6E65 6F32 A2F0 796B  ..Personneo2¢ðyk
40  7022 0200 0349 0006 7461 696C 6C65 4C00  p"...I..tailleL.
50  036E 6F6D 7400 124C 6A61 7661 2F6C 616E  .nomt..Ljava/lan
60  672F 5374 7269 6E67 3B4C 0006 7072 656E  g/String;L..pren
70  6F6D 7100 7E00 0378 7000 0000 AF74 0006  omq.~..xp...¯t..
80  4475 706F 6E64 7400 044A 6561 6E75 7200  Dupondt..Jeanur.
90  025B 494D BA60 2676 EAB2 A502 0000 7870  .[IM°`&vê²¥...xp
A0  0000 0003 0000 0001 0000 0002 0000 0003  ................
B0  771B 0011 6D61 2063 6861 696E 6520 656E  w...ma chaine en
C0  2055 5446 3800 0000 0007 5BCD 1574 0017   UTF8.....[Í.t..
D0  6D61 2063 6861 696E 6520 6465 2063 6172  ma chaine de car
E0  6163 7465 7265 73                        acteres
```

A partir des informations lues, il est possible de réécrire du code qui sera capable de lire le fichier.

17.1.1.3. La classe ObjectInputStream

La classe ObjectInputStream a pour but de désérialiser un objet précédemment sérialisé.

Elle implémente plusieurs interfaces : Closeable (depuis Java 5), DataInput (depuis Java 5), ObjectInput, ObjectStreamConstants et AutoCloseable (depuis Java 7).

Elle ne possède qu'un seul constructeur public qui attend en paramètre un objet de type InputStream qui encapsule le flux dans lequel les données sérialisées seront lues.

Elle possède de nombreuses méthodes :

Méthode	Rôle
Object readObject()	Désérialiser un objet
Object readUnshared()	
void defaultReadObject()	Désérialiser un objet en utilisant les règles par défaut. Ne peut être invoquée que dans le corps de la méthode readObject() sinon une exception de type NotActiveException est levée
GetField readFields()	Lire les champs de l'objet sérialisé
void registerValidation(ObjectInputValidation obj, int priority)	Enregistrer un callback qui validera l'objet désérialisé. La priorité permet de gérer l'ordre d'exécution des callbacks : ils sont exécutés dans leur ordre de priorité décroissante. Cette méthode ne peut être invoquée que dans une méthode readObject() sinon une exception de type NotActiveException est levée
ObjectStreamClass readClassDescriptor()	
Class resolveClass(ObjectStreamClass v)	
Object resolveObject(Object obj)	Renvoyer une autre instance que celle créée lors de la désérialisation
boolean enableResolveObject(boolean enable)	Activer la possibilité de remplacer l'objet lu par une autre instance
void readStreamHeader()	Lire les premiers octets du flux pour vérifier la valeur du magic stream et le numéro de version du protocole. Si ceux-ci sont erronés alors une exception de type StreamCorruptedMismatch est levée
int read()	Lire un octet
int read(byte[] data, int offset, int length)	Lire un ensemble d'octets
int available()	Retourner le nombre d'octets qui peuvent être lus dans le flux
void close()	Fermer le flux
boolean readBoolean() byte readByte() int readUnsignedByte() short readShort() int readUnsignedShort() char readChar() int readInt() long readLong() float readFloat() double readDouble()	Lecture d'une donnée primitive dans le bloc de données (block data)
void readFully(byte[] data)	

void readFully(byte[] data, int offset, int size)	
int skipBytes(int len)	Ignorer dans le flux les prochains octets dont le nombre est précisé en paramètre
String readLine()	Deprecated
String readUTF()	Lire une chaîne de caractères encodée en UTF-8 modifié

La méthode readObject() renvoie une instance de type Object qu'il est nécessaire de caster vers le type présumé. Pour être sûr du type, il est possible d'effectuer un test sur le type de l'objet retourné en utilisant l'opérateur instanceof.

La JVM ne peut désérialiser un objet que si elle peut charger la classe pour en créer une instance : c'est la raison pour laquelle la méthode readObject() peut lever l'exception ClassNotFoundException.

Exemple :

```
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class DeSerializerPersonne {
  public static void main(final String argv[]) {

    ObjectInputStream ois = null;

    try {
      final FileInputStream fichier = new FileInputStream("personne.ser");
      ois = new ObjectInputStream(fichier);
      final Personne personne = (Personne) ois.readObject();
      System.out.println("Personne : ");
      System.out.println("nom : " + personne.getNom());
      System.out.println("prenom : " + personne.getPrenom());
      System.out.println("taille : " + personne.getTaille());
    } catch (final java.io.IOException e) {
      e.printStackTrace();
    } catch (final ClassNotFoundException e) {
      e.printStackTrace();
    } finally {
      try {
        if (ois != null) {
          ois.close();
        }
      } catch (final IOException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

Résultat :

```
Personne :
nom : Dupond
prenom : Jean
taille : 175
```

On crée un objet de la classe FileInputStream qui représente le fichier contenant l'objet sérialisé puis un objet de type ObjectInputStream en lui passant le fichier en paramètre. Un appel à la méthode readObject() retourne l'objet avec un type Object. Un cast est nécessaire pour obtenir le type de l'objet. La méthode close() permet de terminer l'opération et libérer les ressources.

Si la classe a changé entre le moment où elle a été sérialisée et le moment où elle est désérialisée, une exception est levée. Exemple : la classe Personne est modifiée et recompilée

Résultat :

```
java.io.InvalidClassException: Personne;
local class incompatible: stream classdesc serialVersionUID =
503025058333454277, local class serialVersionUID = 1565267035218362193
      at java.io.ObjectStreamClass.initNonProxy(ObjectStreamClass.java:562)
      at java.io.ObjectInputStream.readNonProxyDesc(ObjectInputStream.java:1582)
      at java.io.ObjectInputStream.readClassDesc(ObjectInputStream.java:1495)
      at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1731)
      at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1328)
      at java.io.ObjectInputStream.readObject(ObjectInputStream.java:350)
      at DeSerializerPersonne.main(DeSerializerPersonne.java:13)
```

Une exception de type StreamCorruptedException peut être levée si le fichier a été corrompu par exemple en le modifiant avec un éditeur.

Exemple : les 2 premiers octets du fichier personne.ser ont été modifiés avec un éditeur hexa

Résultat :

```
java.io.StreamCorruptedException:
InputStream does not contain a serialized object
        at java.io.ObjectInputStream.readStreamHeader(ObjectInputStream.java:731)
        at java.io.ObjectInputStream.<init>(ObjectInputStream.java:165)
        at DeSerializerPersonne.main(DeSerializerPersonne.java:8)
```

Exemple : le nom est modifié dans le fichier personne.ser

Résultat :

```
java.io.StreamCorruptedException: invalid type code: 64
        at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1355)
        at java.io.ObjectInputStream.defaultReadFields(ObjectInputStream.java:1946)
        at java.io.ObjectInputStream.readSerialData(ObjectInputStream.java:1870)
        at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1752)
        at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1328)
        at java.io.ObjectInputStream.readObject(ObjectInputStream.java:350)
        at DeSerializerPersonne.main(DeSerializerPersonne.java:13)
```

Une exception de type ClassNotFoundException peut être levée si l'objet est désérialisé vers une classe qui n'existe plus ou pas au moment de l'exécution.

Résultat :

```
java.lang.ClassNotFoundException: Personne
        at java.io.ObjectInputStream.inputObject(ObjectInputStream.java:981)
        at java.io.ObjectInputStream.readObject(ObjectInputStream.java:369)
        at java.io.ObjectInputStream.readObject(ObjectInputStream.java:232)
        at DeSerializerPersonne.main(DeSerializerPersonne.java:9)
```

La classe ObjectInputStream possède de la même façon que la classe ObjectOutputStream des méthodes pour lire des données de types primitives : readInt(), readDouble(), readFloat(), ...

Lors de la désérialisation, le constructeur de l'objet n'est jamais utilisé.

Exemple :

```
public class Personne implements java.io.Serializable {
  private String nom    = "";
  private String prenom = "";
  private int    taille = 0;

  public Personne(final String nom, final String prenom, final int taille) {
    this.nom = nom;
    this.taille = taille;
    this.prenom = prenom;
    System.out.println("invocation du constructeur");
  }

  public String getNom() {
    return this.nom;

  }

  public int getTaille() {
    return this.taille;
  }

  public String getPrenom() {
    return this.prenom;
  }
}
```

Résultat :

```
Personne :
nom : Dupond
prenom : Jean
taille : 175
```

17.1.1.4. Le mot clé transient

Le mot clé transient permet de préciser qu'une variable d'instance ne doit pas être prise en compte lors de la sérialisation de l'état d'un objet.

Le contenu des attributs est visible dans le flux dans lequel est sérialisé l'objet. Il est ainsi possible pour toute personne ayant accès au flux de voir le contenu de chaque attribut même si ceux-ci sont private, ce qui peut poser des problèmes de sécurité surtout si les données sont sensibles.

Java introduit le mot clé transient qui précise que l'attribut qu'il qualifie ne doit pas être inclus dans un processus de sérialisation et donc de désérialisation.

Exemple :

```
public class Personne implements java.io.Serializable {
  private String           nom        = "";
  private String           prenom     = "";
  private int              taille     = 0;
  private transient String codeSecret = "";

  public Personne(final String nom, final String prenom, final int taille,
    final String codeSecret) {
    this.nom = nom;
    this.taille = taille;
    this.prenom = prenom;
    this.codeSecret = codeSecret;
  }

  public String getNom() {
    return this.nom;
  }

  public int getTaille() {
    return this.taille;
  }

  public String getPrenom() {
    return this.prenom;
  }

  public String getCodeSecret() {
    return this.codeSecret;
  }
}
```

Exemple :

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class SerDeserPersonne {

  public static void main(final String[] args) {
    Personne personne = new Personne("Dupond", "Jean", 175, "1234");
    ObjectOutputStream oos = null;
    ObjectInputStream ois = null;

    try {
      final FileOutputStream fichierOut = new FileOutputStream("personne.ser");
      oos = new ObjectOutputStream(fichierOut);
      oos.writeObject(personne);
      oos.flush();
      final FileInputStream fichierIn = new FileInputStream("personne.ser");
      ois = new ObjectInputStream(fichierIn);
      personne = (Personne) ois.readObject();
      System.out.println("Personne : ");
      System.out.println("nom : " + personne.getNom());
      System.out.println("prenom : " + personne.getPrenom());
      System.out.println("taille : " + personne.getTaille());
      System.out.println("codeSecret : " + personne.getCodeSecret());
    } catch (final java.io.IOException e) {
      e.printStackTrace();
    } catch (final ClassNotFoundException e) {
      e.printStackTrace();
    } finally {
      try {
        if (ois != null) {
          ois.close();
        }
        if (oos != null) {
          oos.close();
        }
      } catch (final IOException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

Résultat :

```
Personne :
nom : Dupond
prenom : Jean
taille : 175
codeSecret : null
```

Lors de la désérialisation, les champs transient sont initialisés avec la valeur null. L'objet recréé doit donc gérer cet état pour éviter d'avoir des exceptions de type NullPointerException.

Il est aussi pratique de marquer transient des champs d'un type qui n'est pas sérialisable.

Exemple :

```
public class Personne implements java.io.Serializable {
  private String nom        = "";
  private String prenom     = "";
  private int    taille     = 0;
  private Thread monThread;

  public Personne() {
  }

  public Personne(final String nom, final String prenom, final int taille) {
    this.nom = nom;
    this.taille = taille;
    this.prenom = prenom;
    this.monThread = new Thread();
  }

  public String getNom() {
    return this.nom;
  }

  public int getTaille() {
    return this.taille;
  }

  public String getPrenom() {
    return this.prenom;
  }
}
```

Résultat :

```
java.io.NotSerializableException: java.lang.Thread
        at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1164)
        at java.io.ObjectOutputStream.defaultWriteFields(ObjectOutputStream.java:1518)
        at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:1483)
        at java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.java:1400)
        at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1158)
        at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:330)
        at SerializerPersonne.main(SerializerPersonne.java:14)
```

Si une propriété de classe n'est pas sérialisable alors, on peut la marquer avec le mot clé transient pour qu'elle soit ignorée lors de la sérialisation.

Exemple :

```
public class Personne implements java.io.Serializable {
  private String           nom        = "";
  private String           prenom     = "";
  private int              taille     = 0;
  private transient Thread monThread;

  public Personne() {
  }

  public Personne(final String nom, final String prenom, final int taille) {
    this.nom = nom;
    this.taille = taille;
    this.prenom = prenom;
    this.monThread = new Thread();
  }

  public String getNom() {
    return this.nom;
  }

  public int getTaille() {
    return this.taille;
  }

  public String getPrenom() {
    return this.prenom;
  }
}
```

17.1.1.5. La gestion des versions d'une classe sérialisable

La gestion de la version d'une classe est importante car il est possible que la classe dans laquelle les données sont désérialisées soit différente de la classe de l'instance sérialisée.

La sérialisation inclut le numéro de version de la classe. Lors de la désérialisation, le numéro de version sérialisé est comparé au numéro de version de la classe : si ces numéros sont différents alors une exception de type InvalidClassException est levée.

Chaque classe qui implémente l'interface Serializable possède un numéro de version stocké dans un champ de type long nommé serialVersionUID.

Ce champ doit être déclaré static et final. Il est préférable de déclarer le champ serialVersionUID private car il y a normalement peut d'intérêt à être hérité puisqu'il concerne la déclaration de la classe elle-même.

Exemple :

```
private static final long serialVersionUID = 1L;
```

Si le champ serialVersionUID n'est pas explicitement précisé dans la classe, alors le compilateur va l'ajouter automatiquement en calculant une valeur. La spécification du langage Java ne précise pas comment cette valeur doit être calculée : l'algorithme utilisé peut être différent selon l'implémentation du compilateur proposé par le fournisseur du JDK. Il est donc recommandé de gérer explicitement le numéro de version pour maximiser la compatibilité.

Par exemple, avec le compilateur de Sun/Oracle, le serialVersionUID est le résultat du calcul de la signature avec SHA-1 d'un ensemble d'octets qui reprend les principales caractéristiques de la classe (nom de la classe, modificateurs d'accès, noms des interfaces implémentées, constructeurs non privés, méthodes non privées, champs non static ni transient, ...). Un décalage de bits est appliqué sur les deux premiers octets de la signature pour obtenir la valeur du serialVersionUID.

A partir de Java 5, le compilateur signale un warning pour les classes qui implémentent l'interface Serializable et qui ne définissent pas explicitement le serialVersionUID.

La valeur du champ serialVersionUID est toujours incluse dans les données contenant le résultat de la sérialisation. Il est utilisé lors de la désérialisation pour vérifier que les données de la sérialisation sont compatibles, relativement aux règles de la sérialisation, avec la classe chargée. Lors de la désérialisation cette valeur est comparée avec celle du champ serialVersionUID de la classe correspondante. Si les deux valeurs sont différentes alors la désérialisation lève une exception de type InvalidClassException.

Il est possible que la classe ait été modifiée entre le moment où une de ses instances a été sérialisée et le moment où une nouvelle instance est crée par désérialisation. C'est donc une bonne pratique de définir explicitement un champ serialVersionUID dans toutes les classes Serializable : ceci permet de garder le control sur la compatibilité des changements dans la classe.

Si une classe sérialisable ne déclare pas explicitement le champ serialVersionUID, une valeur par défaut utilisant différents éléments de la classe est calculée par le compilateur et lui est affectée. Comme la valeur par défaut est relativement sensible aux changements dans la classe, il est préférable de définir explicitement la valeur du serialVersionUID pour garder le contrôle sur la compatibilité :

- entre les changements dans la classe
- entre les différentes implémentations du compilateur Java

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.Serializable;

public class NumeroVersion implements Serializable {
  private final String majeur;
  private final String mineur;

  public NumeroVersion(final String majeur, final String mineur) {
    this.majeur = majeur;
    this.mineur = mineur;
  }

  public String getVersion() {
    return this.majeur + "." + this.mineur;
  }
}
```

La classe est sérialisable mais ne définit pas explicitement le serialVersionUID.

Il n'y a aucun souci pour sérialiser une instance de cet objet.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class SerializerNumeroVersion {

  public static void main(final String argv[]) {
    final NumeroVersion version = new NumeroVersion("1", "0");
    ObjectOutputStream oos = null;

    try {
      final FileOutputStream fichier = new FileOutputStream("numeroversion.ser");
      oos = new ObjectOutputStream(fichier);
      oos.writeObject(version);
      oos.flush();
    } catch (final java.io.IOException e) {
      e.printStackTrace();
    } finally {
      try {
        if (oos != null) {
          oos.flush();
          oos.close();
        }
      } catch (final IOException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

La classe est modifiée pour ajouter un champ revision.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.Serializable;

public class NumeroVersion implements Serializable {
  private final String majeur;
  private final String mineur;
  private final String revision;

  public NumeroVersion(final String majeur, final String mineur) {
    this(majeur, mineur, null);
  }

  public NumeroVersion(final String majeur, final String mineur, final String revision) {
    this.majeur = majeur;
    this.mineur = mineur;
    this.revision = revision;
  }
  public String getVersion() {
    return this.majeur + "." + this.mineur + this.revision;
  }
}
```

Si le fichier contenant la sérialisation de la précédente version de la classe est désérialisé, alors une exception de type InvalidClassException est levée.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class DeSerializerNumeroVersion {
  public static void main(final String argv[]) {

    ObjectInputStream ois = null;

    try {
      final FileInputStream fichier = new FileInputStream("numeroversion.ser");
      ois = new ObjectInputStream(fichier);
      final NumeroVersion version = (NumeroVersion) ois.readObject();

      System.out.println("version : " + version.getVersion());

    } catch (final java.io.IOException e) {
      e.printStackTrace();
    } catch (final ClassNotFoundException e) {
      e.printStackTrace();
    } finally {
      try {
        if (ois != null) {
          ois.close();
        }
      } catch (final IOException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

Résultat :

```
java.io.InvalidClassException: fr.jmdoudoux.dej.serialisation.NumeroVersion;
local class incompatible: stream classdesc serialVersionUID = -3291283991370239935,
local class serialVersionUID = 5449821435975484475
        at java.io.ObjectStreamClass.initNonProxy(ObjectStreamClass.java:560)
        at java.io.ObjectInputStream.readNonProxyDesc(ObjectInputStream.java:1580)
        at java.io.ObjectInputStream.readClassDesc(ObjectInputStream.java:1493)
        at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1729)
        at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1326)
        at java.io.ObjectInputStream.readObject(ObjectInputStream.java:348)
        at fr.jmdoudoux.dej.serialisation.DeSerializerNumeroVersion.main
(DeSerializerNumeroVersion.java:15)
```

Pour gérer ce type de cas, en supposant que les deux versions de la classe soient compatibles, il est nécessaire de définir dans la première version une valeur explicite pour le champ serialVersionUID.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.Serializable;

public class NumeroVersion implements Serializable {

  private static final long serialVersionUID = 1L;

  private String            majeur;
  private String            mineur;

  public NumeroVersion(final String majeur, final String mineur) {
    this.majeur = majeur;
    this.mineur = mineur;
  }

  public String getVersion() {
    return this.majeur + "." + this.mineur;
  }

  public String getMajeur() {
    return this.majeur;
  }

  public String getMineur() {
    return this.mineur;
  }

  public void setMajeur(final String majeur) {
    this.majeur = majeur;
  }

  public void setMineur(final String mineur) {
    this.mineur = mineur;
  }
}
```

Dans la nouvelle version de la classe, il faut définir la même valeur pour le champ serialVersionUID et gérer le fait que la valeur du champ révision soit null.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.Serializable;

public class NumeroVersion implements Serializable {

  private static final long serialVersionUID = 1L;

  private final String      majeur;
  private final String      mineur;
  private final String      revision;

  public NumeroVersion(final String majeur, final String mineur) {
    this(majeur, mineur, null);
  }

  public NumeroVersion(final String majeur, final String mineur, final String revision) {
    this.majeur = majeur;
    this.mineur = mineur;
    this.revision = revision;
  }

  public String getVersion() {
    return this.majeur + "." + this.mineur + (this.revision == null ? "" : "."
       + this.revision);
  }
}
```

Il ne faut changer la valeur explicite d'un serialVersionUID que si les modifications faites dans la classe la rendent incompatible avec la version précédente.

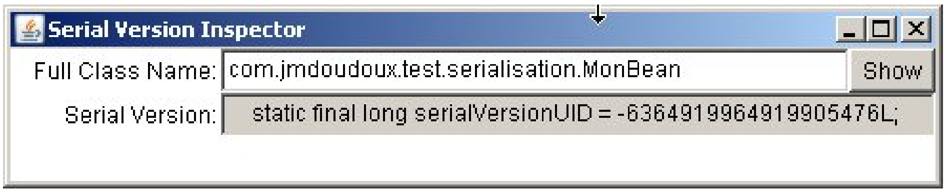
La plupart des IDE propose une fonctionnalité pour calculer et ajouter la propriété serialVersionUID dans le code source de la classe.

Les JDK de Sun/Oracle proposent l'outil serialver qui permet d'obtenir le serialVersionIUD d'une classe.

Résultat :

```
C:\java\eclipse\workspace\TestSerialisation\bin>serialver
use: serialver [-classpath classpath] [-show] [classname...]
```

L'option - show permet d'ouvrir une petite interface graphique qui permet d'afficher le serialVersionUID d'une classe



Full Class Name: com.jmdoudoux.test.serialisation.MonBean

Serial Version: static final long serialVersionUID = -6364919964919905476L;

Résultat :

```
C:\java\eclipse\workspace\TestSerialisation\bin>serialver
-show
```

Il suffit de passer le nom pleinement qualifié de la classe. Elle doit se trouver dans le classpath : l'option -classpath permet de le préciser au besoin.

Résultat :

```
C:\java\eclipse\workspace\TestSerialisation\bin>serialver
fr.jmdoudoux.dej.serialisation.MonBean
fr.jmdoudoux.dej.serialisation.MonBean:    static final long serialVersionUID =
-6364919964919905476L;
```

17.1.1.6. La compatibilité des versions de classes sérialisées

Il est fréquent de devoir gérer la compatibilité ascendante voire descendante pour permettre à une nouvelle version de la classe d'un objet d'être créée à partir des données sérialisées d'une précédente version ou éventuellement dans des cas plus rares de pouvoir créer une instance d'une précédente version à partir des données sérialisées d'une version plus récente de la classe de l'objet.

Certaines évolutions d'une classe peuvent avoir un impact sur la compatibilité des données sérialisées avec le mécanisme par défaut et ainsi ne plus garantir l'interopérabilité entre deux versions.

Il est donc important, lors de l'utilisation de la sérialisation, de prendre en compte la compatibilité des versions de la classe.

La sérialisation par défaut de Java peut gérer automatiquement plusieurs évolutions dans une classe sans que cela empêche la désérialisation de données de la version précédente de la classe (sous réserve que le serialVersionUID reste le même dans les deux versions de la classe)

Les changements qui impliquent une incompatibilité lors de l'utilisation de la sérialisation par défaut sont :

- suppression de champs : la lecture d'un flux ne contenant pas le champ par une ancienne version de la classe va induire que le champ a sa valeur par défaut et potentiellement changer le comportement des traitements
- changer la hiérarchie des classes en intervertissant des classes mère/fille
- ajouter le modificateur static à un champ (d'un point de vue de la sérialisation par défaut cela revient à supprimer le champ)
- ajouter le modificateur transient à un champ (d'un point de vue de la sérialisation par défaut cela revient à supprimer le champ)

- changer le type d'une donnée primitive : la donnée ne sera pas lue avec le bon type
- remplacer l'implémentation de l'interface Serializable par Externalizable et vice versa
- supprimer l'implémentation de l'interface Serializable ou Externalizable
- transformer une classe en énumération et vice versa
- ajouter les méthodes readResolve() et writeReplace() et que leur comportement utilise des objets incompatibles avec la version précédente
- modifier dans les méthodes writeObject() et readObject() la prise en compte des champs par défaut (ajouter ou retirer l'invocation des méthodes defaultWritebject() et defaultReadObject())

Les changements qui maintiennent la compatibilité sont :

- ajouter des champs : dans ce cas, le champ non lu du flux sérialisé sera initialisé avec sa valeur par défaut
- ajouter les méthodes writeObjet() et readObject() sous réserve que les méthodes defaultWritebject() et defaultReadObject() soient invoquées avant l'écriture/la lecture de données additionnelles
- changer le modificateur d'accès d'un champ (public, package, protected et private)
- retirer le modificateur static d'un champ (d'un point de vue de la sérialisation par défaut cela revient à ajouter un champ)
- retirer le modificateur transient d'un champ (d'un point de vue de la sérialisation par défaut cela revient à ajouter un champ)
- implémenter l'interface Serializable (d'un point de vue de la sérialisation par défaut cela revient à ajouter un type) : les classes mères non sérialisables doivent avoir un constructeur par défaut
- retirer les méthodes writeObject() et readObject() : dans ce cas, les éventuelles données optionnelles rajoutées sont ignorées
- ajouter un type dans la hiérarchie de la classe : les champs de cette classe seront initialisés avec leur valeur par défaut
- supprimer un type dans la hiérarchie de la classe : un objet du type est tout de même créé au cas où une référence est faite dessus mais les valeurs des champs sont ignorées

17.1.1.7. Des points particuliers

Plusieurs points particuliers doivent être pris en compte lors de la mise en oeuvre de la sérialisation/désérialisation.

Les membres static d'un objet ne sont jamais sérialisés car ils ne concernent pas l'état d'un objet mais de tous les objets d'une même classe. Par exemple, une classe définit un membre static. Trois instances de cette classe sont sérialisées à des moments où la valeur du membre static est différente. Lors de la désérialisation, qu'elle serait la bonne valeur pour le membre static ?

Attention, lorsqu'un objet est désérialisé, le constructeur de sa classe n'est pas invoqué et les variables d'instances ne sont pas initialisées avec les valeurs qui pourraient leur être assignées.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.Serializable;

public class MaClasseTransient implements Serializable {

  transient int valeur = 1234;

  public MaClasseTransient() {
    super();
    System.out.println("Invocation du constructeur");
  }

  public int getValeur() {
    return this.valeur;
  }

  public void setValeur(final int valeur) {
    this.valeur = valeur;
  }
}
```

Dans ce cas, la valeur du champ transient est celle par défaut selon son type.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class SerDeserMaClasseTransient {
  public static void main(final String[] args) {
    MaClasseTransient maClasse = new MaClasseTransient();
    ObjectOutputStream oos = null;
    ObjectInputStream ois = null;

    try {
      System.out.println("Serialisation");
      final FileOutputStream fichierOut = new FileOutputStream("maclassetransient.ser");
      oos = new ObjectOutputStream(fichierOut);
      oos.writeObject(maClasse);
      oos.flush();

      System.out.println("Deserialisation");
      final FileInputStream fichierIn = new FileInputStream("maclassetransient.ser");
      ois = new ObjectInputStream(fichierIn);
      maClasse = (MaClasseTransient) ois.readObject();
      System.out.println("MaClasseTransient valeur : " + maClasse.getValeur());
    } catch (final java.io.IOException e) {
      e.printStackTrace();
    } catch (final ClassNotFoundException e) {
      e.printStackTrace();
    } finally {
      try {
        if (ois != null) {
          ois.close();
        }
        if (oos != null) {
          oos.close();
        }
      } catch (final IOException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

Résultat :

```
Invocation du constructeur
Serialisation
Deserialisation
MaClasseTransient valeur : 0
```

Si une classe mère implémente l'interface Serializable alors toutes les classes filles en héritent : il est donc inutile d'implémenter explicitement l'interface Serializable pour une classe fille. De fait, il n'est pas possible de facilement savoir si une classe est sérialisable ou non uniquement en regardant son code source : il est nécessaire de savoir si une classe mère implémente l'interface Serializable. La seule exception est si la classe hérite directement de la classe Object puisque celle-ci n'implémente pas l'interface Serializable.

L'héritage peut aussi avoir un impact sur la sérialisation notamment lorsqu'une classe fille est sérialisable mais qu'aucune de ses classes mères l'est.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;
```

```
public class MaClasseMere {

  protected String nom = null;

  public String getNom() {
    return this.nom;
  }

  public void setNom(final String nom) {
    this.nom = nom;
  }

  public MaClasseMere() {
    System.out.println("Invocation contructeur MaClasseMere");
  }

  public MaClasseMere(final String nom) {
    super();
    this.nom = nom;
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.Serializable;

public class MaClasseFille extends MaClasseMere implements Serializable {

  private int valeur = 0;

  public MaClasseFille() {
    System.out.println("Invocation contructeur MaClasseFille");
  }

  public MaClasseFille(final String nom, final int valeur) {
    super(nom);
    this.valeur = valeur;
  }

  public int getValeur() {
    return this.valeur;
  }

  public void setValeur(final int valeur) {
    this.valeur = valeur;
  }
}
```

Dans ce cas, l'état des propriétés de la classe mère est ignoré lors de la sérialisation : la valeur par défaut de ces propriétés est alors celle par défaut selon leur type.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class SerDeserMaClasseFille {
  public static void main(final String[] args) {

    MaClasseFille maClasseFille = new MaClasseFille("nom1", 123);
    ObjectOutputStream oos = null;
    ObjectInputStream ois = null;

    try {
      System.out.println("Serialisation");
      final FileOutputStream fichierOut = new FileOutputStream("maclassefille.ser");
      oos = new ObjectOutputStream(fichierOut);
      oos.writeObject(maClasseFille);
      oos.flush();

      System.out.println("Deserialisation");
      final FileInputStream fichierIn = new FileInputStream("maclassefille.ser");
      ois = new ObjectInputStream(fichierIn);
      maClasseFille = (MaClasseFille) ois.readObject();
      System.out.println("MaClasseFille : ");
      System.out.println("nom : " + maClasseFille.getNom());
      System.out.println("taille : " + maClasseFille.getValeur());
    } catch (final java.io.IOException e) {
      e.printStackTrace();
    } catch (final ClassNotFoundException e) {
      e.printStackTrace();
    } finally {
      try {
        if (ois != null) {
          ois.close();
        }
        if (oos != null) {
          oos.close();
        }
      } catch (final IOException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

Résultat :

```
MaClasseFille :
nom : null
taille
: 123
```

Comme la classe mère n'est pas sérialisable, son état n'est pas pris en compte lors de la sérialisation/désérialisation même si ses champs sont hérités et accessibles dans la classe fille. Lors de la désérialisation, le constructeur de la classe mère est invoqué puisqu'elle n'est pas sérialisable : les valeurs des champs de la classe mère sont alors ceux par défaut selon leur type ou leur initialisation. Si la classe mère ne possède pas de constructeur par défaut alors une exception de type InvalidClassException est levée.

Dans ce cas, si la classe mère ne peut pas être modifiée pour implémenter l'interface Serializable, il est nécessaire de personnaliser la sérialisation pour inclure manuellement l'état des propriétés de la classe mère.

Une fois qu'une des classes de la hiérarchie implémente l'interface Serializable, toutes ses classes filles sont obligatoirement sérializable. Pour rendre une de ces classes filles non sérialisable, il faut personnaliser la sérialisation pour cette classe en définissant les méthodes writeObject() et readObject() pour qu'elle lève une exception de type NotSerializableException.

Les interfaces de l'API Collection ne sont pas sérialisables mais les classes concrètes qui les implémentent le sont. Il est très important que tous les éléments qui sont ajoutés à une collection qui doit être sérialisée soient sérialisables.

17.1.1.8. La vérification d'un objet désérialisé

Il est possible d'effectuer une validation des champs d'un objet désérialisé en faisant implémenter l'interface ObjectInputValidation par sa classe.

L'interface ObjectInputValidation ne définit qu'une seule méthode :

Méthode	Rôle

void validateObject()	Valider les valeurs des champs de l'objet

Si les traitements de validation échouent, alors la méthode validateObject() doit lever une exception de type InvalidObjectException.

Pour être invoquée, une instance de type ObjectInputValidation doit être enregistrée en utilisant la méthode registerValidation() de la classe ObjectInputStream. Celle-ci attend en paramètres l'instance de type ObjectInputValidation et un entier qui correspond à la priorité d'invocation.

Les règles de gestion des priorités sont :

- Zéro est une bonne valeur par défaut
- Plus la valeur est élevée, plus tardivement la validation est invoquée
- Les validations de même priorité sont invoquées dans un ordre quelconque

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.IOException;
import java.io.InvalidObjectException;
import java.io.ObjectInputStream;
import java.io.ObjectInputValidation;
import java.io.Serializable;

public class MaClasseValidation implements Serializable, ObjectInputValidation {
  private String nom;

  public String getNom() {
    return this.nom;
  }

  public void setNom(final String nom) {
    this.nom = nom;
  }

  @Override
  public void validateObject() throws InvalidObjectException {
    if (this.nom == null) {
      throw new InvalidObjectException("Le champ nom ne doit pas être vide");
    }
  }

  private void readObject(final ObjectInputStream in) throws IOException,
    ClassNotFoundException {
    in.registerValidation(this, 0);
    in.defaultReadObject();
  }
}
```

Résultat :

```
Serialisation
Deserialisation
java.io.InvalidObjectException: Le champ nom ne doit pas être vide
        at fr.jmdoudoux.dej.serialisation.MaClasseValidation.validateObject
(MaClasseValidation.java:23)
        at java.io.ObjectInputStream$ValidationList$1.run(ObjectInputStream.java:2210)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.io.ObjectInputStream$ValidationList.doCallbacks(ObjectInputStream.java:2206)
        at java.io.ObjectInputStream.readObject(ObjectInputStream.java:355)
        at fr.jmdoudoux.dej.serialisation.SerDeserMaClasseValidation.main
(SerDeserMaClasseValidation.java:26)
```

17.1.1.9. Les exceptions liées à la sérialisation

L'API Serialization définit et utilise plusieurs exceptions qui héritent toutes de la classe ObjectStreamException. Cette classe est elle-même une classe fille de la classe IOException.

Exception	Rôle
ObjectStreamException	Classe mère des exceptions liées à la sérialisation
InvalidClassException	Levée lorsque qu'un objet ne peut pas être désérialisé à cause de sa classe : • le numéro de version de la classe ne correspond pas à celui des données sérialisées • le type d'un champ primitif ne correspond pas à celui d'une donnée sérialisée • la classe implémente l'interface Externalizable mais ne possède pas de constructeur par défaut accessible • La classe implémente l'interface Serializable mais une classe mère n'est pas sérialisable et ne possède pas de constructeur par défaut accessible
NotSerializableException	La classe n'est pas sérialisable
StreamCorruptedException	Le flux qui contient les données sérialisées est corrompu ou invalide (lecture de données sérialisées au format v2 avec un JDK inférieur à 1.1.5)
NotActiveException	La sérialisation n'est pas active
InvalidObjectException	La validation d'un objet désérialisé a échoué
OptionalDataException	La lecture de données primitives a échoué
WriteAbortedException	Une erreur est survenue durant l'écriture du flux

17.1.2. La sérialisation personnalisée

Le mécanisme de sérialisation par défaut de Java peut ne pas répondre à tous les besoins : dans ce cas, il est possible de personnaliser la façon dont un objet est sérialisé/désérialisé.

Dans certains cas, il est nécessaire d'utiliser cette personnalisation, par exemple :

- pour contrôler les instances utilisées
- pour tenir compte des données sensibles
- pour gérer la sérialisation de différentes versions de la classe
- pour améliorer les performances notamment avec des versions anciennes de Java

Plusieurs mécanismes sont proposés en standard pour personnaliser la sérialisation/désérialisation :

- définir explicitement la liste des champs à inclure dans la sérialisation
- définir les méthodes writeObject() et readObject()
- définir les méthodes writeReplace() et readResolve()
- implémenter l'interface Externalizable

Remarque : la sérialisation/désérialisation d'une énumération ne peut pas être personnalisée : les méthodes writeObject(), readObject(), readObjectNoData(), writeReplace() et readResolve() sont ignorées par le mécanisme de sérialisation. Les champs serialPersistentFields et serialVersionUID sont ignorés d'autant que ce dernier a toujours la valeur 0L.

17.1.2.1. La définition des champs à sérialiser

Le mécanisme de sérialisation par défaut ignore les champs transient et static.

A partir de Java 1.2, il est possible de préciser explicitement la liste des champs qui devront être pris en compte lors de la sérialisation/désérialisation en définissant un champ qui est un tableau de type java.io. ObjectStreamField.

Ce champ doit obligatoirement se nommer serialPersistentFields et doit être déclaré avec les modificateurs private, static et final.

Une instance de type ObjectStreamField est créée en passant le nom du champ et son type en paramètre du constructeur.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.ObjectStreamField;
import java.io.Serializable;

public class MonBean implements Serializable {

  private String                           champ1;
  private String                           champ2;

  private static final ObjectStreamField[] serialPersistentFields =
    { new ObjectStreamField("champ1", String.class) };

  public String getChamp1() {
    return this.champ1;
  }

  public String getChamp2() {
    return this.champ2;
  }

  public void setChamp2(final String champ2) {
    this.champ2 = champ2;
  }

  public void setChamp1(final String champ1) {
    this.champ1 = champ1;
  }

  @Override
  public String toString() {
    return "MonBean [champ1=" + this.champ1 + ", champ2=" + this.champ2 + "]";
  }
}
```

Les champs qui ne sont pas inclus dans la sérialisation seront initialisés avec leur valeur par défaut lors de la désérialisation.

Résultat :

```
MonBean [champ1=valeur1, champ2=null]
```

Attention : l'utilisation d'un champ serialPersistentFields remplace purement et simple le mécanisme de recherche par défaut des champs à sérialiser. Ainsi un champ marqué transient mais défini dans le champ serialPersistentFields sera sérialisé.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.ObjectStreamField;
import java.io.Serializable;

public class MonBean implements Serializable {

  private transient String                 champ1;
  private transient String                 champ2;

  private static final ObjectStreamField[] serialPersistentFields = {
      new ObjectStreamField("champ1", String.class),
      new ObjectStreamField("champ2", String.class)};
  public String getChamp1() {
    return this.champ1;
  }

  public String getChamp2() {
    return this.champ2;
  }

  public void setChamp2(final String champ2) {
    this.champ2 = champ2;
  }

  public void setChamp1(final String champ1) {
    this.champ1 = champ1;
  }

  @Override
  public String toString() {
    return "MonBean [champ1=" + this.champ1 + ", champ2=" + this.champ2 + "]";
  }
}
```

Résultat :

```
MonBean [champ1=valeur1, champ2=valeur2]
```

Si le champ serialPersistentFields est null, n'est pas un tableau de type ObjectStreamField ou n'est pas déclaré private static final alors il est ignoré par le mécanisme de sérialisation.

L'utilisation du champ serialPersistentFields n'est pas possible dans une classe interne pour des champs qui ne sont pas static.

17.1.2.2. Les méthodes writeObject() et readObject()

Les mécanismes de sérialisation/désérialisation par défaut ne sont pas toujours adaptés à certains besoins spécifiques qui requièrent une personnalisation des actions réalisées.

Pour cela, il est possible de définir les méthodes writeObject() et readObject() dans la classe à sérialiser.

Comme défini dans les spécifications de l'API Serialization, la signature de ces deux méthodes doit obligatoirement être :

- private void readObject(ObjectInputStream ois) throws IOException, ClassNotFoundException
- private void writeObject(ObjectOutptStream oos) throws IOException

Lors de la sérialisation/désérialisation d'un objet de la classe, ces deux méthodes seront invoquées en remplacement du mécanisme standard.

Cette section propose plusieurs cas d'utilisation de la personnalisation en utilisant les méthodes writeObject() et readObject().

Le premier cas permet de ne pas sérialiser un champ qui peut par exemple contenir des données sensibles comme un mot de passe.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

public class MaClasse implements Serializable {
  private String champ1;
  private String champ2;

  public String getChamp1() {
    return this.champ1;
  }

  public String getChamp2() {
    return this.champ2;
  }

  public void setChamp2(final String champ2) {
    this.champ2 = champ2;
  }

  public void setChamp1(final String champ1) {
    this.champ1 = champ1;
  }

  @Override
  public String toString() {
    return "MaClasse [champ1=" + this.champ1 + ", champ2=" + this.champ2 + "]";
  }

  private void readObject(final ObjectInputStream ois) throws IOException,
    ClassNotFoundException {
    this.champ1 = (String) ois.readObject();
  }

  private void writeObject(final ObjectOutputStream oos) throws IOException {
    oos.writeObject(this.champ1);
  }

}
```

Les traitements réalisés par ces deux méthodes doivent être synchronisés : l'ordre d'ajout des attributs de l'objet dans la méthode writeObject() doit être le même que l'ordre de lecture des attributs dans la méthode readObject(). Si ce n'est pas le cas, généralement une exception de type java.io.OptionalDataException est levée.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class TestReadWriteObject {

  public static void main(final String[] args) {
    MaClasse maClasse = new MaClasse();
    maClasse.setChamp1("valeur1");
    maClasse.setChamp2("valeur2");

    ObjectOutputStream oos = null;
    ObjectInputStream ois = null;

    try {
      final FileOutputStream fichierOut = new FileOutputStream("maclasse.ser");
      oos = new ObjectOutputStream(fichierOut);
      oos.writeObject(maClasse);
      oos.flush();

      final FileInputStream fichierIn = new FileInputStream("maclasse.ser");
      ois = new ObjectInputStream(fichierIn);
      maClasse = (MaClasse) ois.readObject();

      System.out.println("" + maClasse);
    } catch (final java.io.IOException e) {
      e.printStackTrace();
    } catch (final ClassNotFoundException e) {
      e.printStackTrace();
    } finally {
      try {
        if (ois != null) {
          ois.close();
        }
        if (oos != null) {
          oos.close();
        }
      } catch (final IOException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

Résultat :

```
MaClasse [champ1=valeur1, champ2=null]
```

Les champs des classes mères qui implémentent l'interface Serializable seront utilisés par le mécanisme de sérialisation. Les méthodes readObject() et writeObject() ne doivent dans ce cas prendre en compte que les champs de la classe elle-même.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.Serializable;

public class MaClasseParent implements Serializable {

  private String champ3;

  public String getChamp3() {
    return this.champ3;
  }

  public void setChamp3(final String champ3) {
    this.champ3 = champ3;
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class MaClasse extends MaClasseParent {

  private String champ1;
  private String champ2;

  public String getChamp1() {
    return this.champ1;
  }

  public String getChamp2() {
    return this.champ2;
  }

  public void setChamp2(final String champ2) {
    this.champ2 = champ2;
  }
  public void setChamp1(final String champ1) {
    this.champ1 = champ1;
  }

  @Override
  public String toString() {
    return "MaClasse [champ1=" + this.champ1 + ", champ2=" + this.champ2 + ",
    champ3=" + getChamp3() + "]";
  }

  private void readObject(final ObjectInputStream ois) throws IOException,
    ClassNotFoundException {
    this.champ1 = (String) ois.readObject();
    this.champ2 = (String) ois.readObject();
  }

  private void writeObject(final ObjectOutputStream oos) throws IOException {
    oos.writeObject(this.champ1);
    oos.writeObject(this.champ2);
  }

}
```

Résultat :

```
MaClasse [champ1=valeur1, champ2=valeur2, champ3=valeur3]
```

Il peut être, par exemple, utile d'utiliser le mécanisme de personnalisation pour permettre de sérialiser des champs privés d'une classe mère qui n'est pas sérialisable.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;


public class MaClasseParent {

  private String champ3;

  public String getChamp3() {
    return this.champ3;
  }

  public void setChamp3(final String champ3) {
    this.champ3 = champ3;
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.Serializable;

public class MaClasse extends MaClasseParent implements Serializable {

  private String champ1;
  private String champ2;

  public String getChamp1() {
    return this.champ1;
  }

  public String getChamp2() {
    return this.champ2;
  }

  public void setChamp2(final String champ2) {
    this.champ2 = champ2;
  }
  public void setChamp1(final String champ1) {
    this.champ1 = champ1;
  }

  @Override
  public String toString() {
    return "MaClasse [champ1=" + this.champ1 + ", champ2=" + this.champ2 + ",
    champ3=" + getChamp3() + "]";
  }
}
```

Si on sérialise/désérialise cette classe avec les mécanismes par défaut, le champ privé de la classe mère n'est pas pris en compte

Résultat :

```
MaClasse [champ1=valeur1, champ2=valeur2, champ3=null]
```

Pour forcer la sérialisation de ce champs, il faut personnaliser l'opération de sérialisation en implémentant les méthodes readObject() et writeObject().

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

public class MaClasse extends MaClasseParent implements Serializable {

  private String champ1;
  private String champ2;

  public String getChamp1() {
    return this.champ1;
  }

  public String getChamp2() {
    return this.champ2;
  }

  public void setChamp2(final String champ2) {
    this.champ2 = champ2;
  }

  public void setChamp1(final String champ1) {
    this.champ1 = champ1;
  }

  @Override
  public String toString() {
    return "MaClasse [champ1=" + this.champ1 + ", champ2=" + this.champ2 + ",
    champ3=" + getChamp3() + "]";
  }

  private void readObject(final ObjectInputStream ois) throws IOException,
    ClassNotFoundException {
    this.champ1 = (String) ois.readObject();
    this.champ2 = (String) ois.readObject();
    setChamp3((String) ois.readObject());
  }

  private void writeObject(final ObjectOutputStream oos) throws IOException {
    oos.writeObject(this.champ1);
    oos.writeObject(this.champ2);
    oos.writeObject(getChamp3());
  }

}
```

Résultat :

```
MaClasse
[champ1=valeur1, champ2=valeur2, champ3=valeur3]
```

Pour utiliser le mécanisme de sérialisation par défaut, il est possible d'invoquer les méthodes :

- defaultReadobject() de la classe ObjectInputStream
- defautWriteObject() de la classe ObjectOutputStream

L'utilisation de ces méthodes est par exemple pratique lorsque la personnalisation de la sérialisation/désérialisation consiste simplement à vérifier l'intégrité des données désérialisées ou à réaliser des opérations pré/post sérialisation/désérialisation.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

public class MaPosition implements Serializable {

  private final int x;
  private final int y;

  public MaPosition(final int x, final int y) {
    this.x = x;
    this.y = y;
    verifierIntegrite();
  }

  public int getX() {
    return this.x;
  }

  public int getY() {
    return this.y;
  }

  private void readObject(final ObjectInputStream o) throws IOException,
    ClassNotFoundException {
    o.defaultReadObject();
    verifierIntegrite();
  }

  private void writeObject(final ObjectOutputStream o) throws IOException {
    o.defaultWriteObject();
  }

  protected void verifierIntegrite() {
    if ((this.x < 0) || (this.y < 0)) {
      throw new IllegalArgumentException(
        "Violation des contraintes d'integrite x=" + this.x + ", y=" + this.y);
    }
  }
}
```

Exemple :

```
Exception in thread "main" java.lang.IllegalArgumentException: Violation des contraintes
 d'integrite x=-1, y=-1
        at fr.jmdoudoux.dej.serialisation.MaPosition.verifierIntegrite(MaPosition.java:38)
        at fr.jmdoudoux.dej.serialisation.MaPosition.<init>(MaPosition.java:16)
        at fr.jmdoudoux.dej.serialisation.SerDeserMaPosition.main(SerDeserMaPosition.java:12)
```

Ceci peut permettre de valider les données au cas où celles-ci auraient été modifiées. Il est aussi possible d'utiliser l'interface ObjectInputValidation qui est la solution standard pour valider des données désérialisées.

17.1.2.3. La méthode readObjectNoData()

Depuis Java 1.4, il est possible de définir la méthode readObjectNoData() dans une classe qui implémente l'interface Serializable. Celle-ci sera invoquée si des données à désérialiser ne comportent rien concernant cette classe : c'est par exemple le cas si la hiérarchie de classes de l'objet sérialisé a changé entre la sérialisation d'un objet de cette classe et sa désérialisation.

Par exemple, une classe est sérialisable et une de ses instances est sérialisée dans un fichier.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.Serializable;

public class ClasseFille implements Serializable {

  private int valeur = 0;

  public int getValeur() {
    return this.valeur;
  }

  public void setValeur(final int valeur) {
    this.valeur = valeur;
  }

}
```

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.Serializable;

public class ClasseFille implements Serializable {

  private int valeur = 0;

  public int getValeur() {
    return this.valeur;
  }

  public void setValeur(final int valeur) {
    this.valeur = valeur;
  }

}
```

Le fichier contient les données de la classe sérialisées. Il n'y a aucun souci pour désérialiser le contenu de ce fichier.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
```

```
public class DeSerializerClasseFille {

  public static void main(final String argv[]) {
    ObjectInputStream ois = null;
    try {
      final FileInputStream fichier = new FileInputStream("classefille.ser");
      ois = new ObjectInputStream(fichier);
      final ClasseFille classeFille = (ClasseFille) ois.readObject();
      System.out.println(classeFille);
    } catch (final java.io.IOException e) {
      e.printStackTrace();
    } catch (final ClassNotFoundException e) {
      e.printStackTrace();
    } finally {
      try {
        if (ois != null) {
          ois.close();
        }
      } catch (final IOException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

La classe ClasseFille est modifiée pour hériter d'une classe ClasseMere. La désérialisation ne pose pas de problème : dans ce cas, les champs de la classe mère seront initialisées avec leurs valeurs par défaut.

Si elle est sérialisable, il est possible de définir la méthode readObjectNoData() qui alors sera invoquée lors de la désérialisation.

La signature de cette méthode doit être :

```
  private void readObjectNoData() throws ObjectStreamException;
```

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.InvalidObjectException;
import java.io.Serializable;

public class ClasseMere implements Serializable {
  private String nom;

  public String getNom() {
    return this.nom;
  }

  public void setNom(final String nom) {
    this.nom = nom;
  }

  private void readObjectNoData() throws InvalidObjectException {
    throw new InvalidObjectException("Donnees serialisee manquante");
  }
}
```

Résultat :

```
java.io.InvalidObjectException: Donnees serialisee manquante
    at fr.jmdoudoux.dej.serialisation.ClasseMere.readObjectNoData(ClasseMere.java:18)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
    at java.lang.reflect.Method.invoke(Method.java:597)
    at java.io.ObjectStreamClass.invokeReadObjectNoData(ObjectStreamClass.java:999)
    at java.io.ObjectInputStream.readSerialData(ObjectInputStream.java:1886)
    at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1756)
    at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1326)
    at java.io.ObjectInputStream.readObject(ObjectInputStream.java:348)
    at fr.jmdoudoux.dej.serialisation.DeSerializerClasseFille.main
        (DeSerializerClasseFille.java:15)
```

Dans l'exemple ci-dessus, si les données de l'objet sont manquantes alors une exception est levée. Il est aussi possible d'initialiser les données des champs de la classe.

Ceci est particulièrement utile par exemple si les champs de la classe mère possèdent des invariants qui ne seraient pas respectés à cause des valeurs par défaut.

17.1.2.4. Les méthodes writeReplace() et readResolve()

Lors de l'utilisation du mécanisme de sérialisation, il est parfois nécessaire d'avoir un contrôle sur l'instance obtenue ou à sérialiser. C'est notamment le cas si la classe est un singleton.

Ce cas d'utilisation met en oeuvre les méthodes writeReplace et readResolve().

La méthode readResolve() permet d'avoir un contrôle direct sur le type et l'instance retournés lors de la désérialisation. Elle doit avoir la signature suivante :

Object readResolve() throws ObjectStreamException;

La classe ObjectInputStream vérifie si la classe possède une méthode readResolve() avec cette signature et si c'est le cas elle l'invoque à la place du mécanisme standard. L'objet retourné par la méthode readResolve() doit cependant être compatible avec la classe de l'objet sérialisé sinon une exception de type ClassCastException est levée.

La méthode writeReplace() permet de remplacer l'instance de l'objet qui sera sérialisé. Elle doit avoir la signature suivante :

Object writeReplace() throws ObjectStreamException;

La classe ObjectInputStream vérifie si la classe possède une méthode writeReplace() avec cette signature et si c'est le cas elle l'invoque pour obtenir l'instance à sérialiser. Le type de cette instance doit être compatible avec la classe de l'objet sérialisé sinon une exception de type ClassCastException est levée.

Ces deux méthodes sont utilisables pour des classes qui implémentent Serializable ou Externalizable.

Les méthodes readResolve() et WriteReplace() peuvent avoir n'importe quel modificateur d'accès.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.Serializable;

public final class MonSingleton implements Serializable {

  private static final long   serialVersionUID = -1572447373762477721L;

  private static MonSingleton instance         = new MonSingleton();

  public static MonSingleton getlnstance() {
    return MonSingleton.instance;
  }

  private MonSingleton() {
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class SerializerSingleton {

  public static void main(final String[] args) {
    try {
      final MonSingleton singleton1 = MonSingleton.getlnstance();
      System.out.println(singleton1);
      final FileOutputStream fos = new FileOutputStream("singleton.ser");
      final ObjectOutputStream oos = new ObjectOutputStream(fos);
      try {
        oos.writeObject(singleton1);
        oos.flush();
      } finally {
        try {
          oos.close();
        } finally {
          fos.close();
        }
      }

      final FileInputStream fis = new FileInputStream("singleton.ser");
      final ObjectInputStream ois = new ObjectInputStream(fis);
      try {
        final MonSingleton singleton2 = (MonSingleton) ois.readObject();
        System.out.println(singleton2);
      } finally {
        try {
          ois.close();
        } finally {
          fis.close();
        }
      }
    } catch (final ClassNotFoundException cnfe) {
      cnfe.printStackTrace();
    } catch (final IOException ioe) {
      ioe.printStackTrace();
    }
  }
}
```

Résultat :

```
fr.jmdoudoux.dej.serialisation.MonSingleton@addbf1
fr.jmdoudoux.dej.serialisation.MonSingleton@c17164
```

Lors de la désérialisation, une nouvelle instance du singleton est créée ce qui rompt le contrat posé par le motif de conception puisque deux instances existent dans la JVM.

Pour résoudre ce problème, il faut définir la méthode readResolve() dans la classe à sérialiser.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.ObjectStreamException;
import java.io.Serializable;

public final class MonSingleton implements Serializable {

  private static final long   serialVersionUID = -1572447373762477721L;

  private static MonSingleton instance         = new MonSingleton();

  public static MonSingleton getlnstance() {
    return MonSingleton.instance;
  }

  private MonSingleton() {
  }

  protected Object readResolve() throws ObjectStreamException {
    return MonSingleton.getlnstance();
  }
}
```

Résultat :

```
fr.jmdoudoux.dej.serialisation.MonSingleton@addbf1
fr.jmdoudoux.dej.serialisation.MonSingleton@addbf1
```

Ce mécanisme peut aussi permettre d'utiliser des proxies au lieu de la classe à sérialiser/désérialiser.

17.1.2.5. L'interface Externalizable

L'implémentation de l'interface Externalizable permet d'avoir un contrôle très fin sur les opérations de sérialisation et désérialisation lorsque les mécanismes de sérialisation par défaut ne répondent pas au besoin.

L'interface Externalizable hérite de l'interface Serializable. Elle définit deux méthodes :

Méthode	Rôle
void readExternal(ObjectInput in)	Désérialiser de manière personnalisée l'objet à partir du flux passé en paramètre. Les méthodes de l'objet de type DataInput permettent de lire des valeurs primitives et la méthode readObject() de la classe ObjectInput permet de lire et créer des objets
void writeExternal(ObjectOutput out)	Sérialiser de manière personnalisée l'état de l'objet dans le flux passé en paramètre. Les méthodes de l'objet de type DataOutput permettent d'écrire des valeurs primitives et la méthode writeObject() de la classe ObjectOutput permet d'écrire des objets

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.i o.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;
import java.util.Date;

public class Personne implements java.io.Externalizable {
  private String           nom        = "";
  private String           prenom     = "";
  private int              taille     = 0;
  private Date             dateNaiss  = null;
  private transient String codeSecret = "";

  public Personne() {
  }

  public Personne(final String nom, final String prenom, final int taille, final String
codeSecret, final Date dateNaiss) {
    this.nom = nom;
    this.taille = taille;
    this.prenom = prenom;
    this.codeSecret = codeSecret;
    this.dateNaiss = dateNaiss;
  }

  public String getNom() {
    return this.nom;
  }

  public void setNom(final String nom) {
    this.nom = nom;
  }

  public int getTaille() {
    return this.taille;
  }

  public void setTaille(final int taille) {
    this.taille = taille;
  }

  public String getPrenom() {
    return this.prenom;
  }

  public void setPrenom(final String prenom) {
    this.prenom = prenom;
  }

  public String getCodeSecret() {
    return this.codeSecret;
  }

  public Date getDateNaiss() {
    return this.dateNaiss;
  }

  public void setDateNaiss(final Date dateNaiss) {
    this.dateNaiss = dateNaiss;
  }

  @Override public void writeExternal(final ObjectOutput out) throws IOException {
  }

  @Override public void readExternal(final ObjectInput in) throws IOException,
  ClassNotFoundException {
  }
}
```

La sérialisation et la désérialisation se font en utilisant les classes ObjetOutputStream et ObjectInputStream.

Résultat :

```
Personne :
nom :
prenom :
taille : 0
codeSecret :
```

Par défaut, la sérialisation d'un objet qui implémente cette interface ne prend en compte aucun attribut de l'objet. Seul le type de la classe est par défaut écrit dans le flux lors de la sérialisation : l'écriture de l'état de l'objet et sa restauration sont de la responsabilité du développeur.

Exemple :

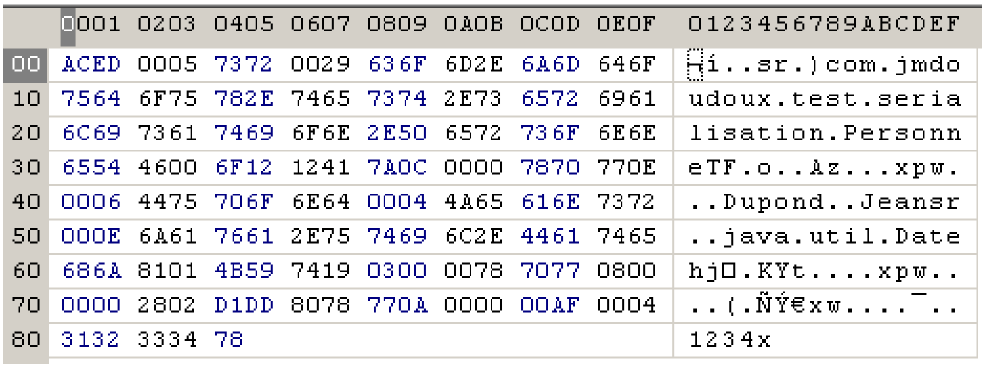
```
package fr.jmdoudoux.dej.serialisation;

import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;
import java.util.Date;

public class Personne implements java.io.Externalizable {
  private String           nom        = "";
  private String           prenom     = "";
  private int              taille     = 0;
  private Date             dateNaiss  = null;
  private transient String codeSecret = "";

  public Personne() {
  }

  // ... getters et setters

  @Override
  public void writeExternal(final ObjectOutput out) throws IOException {
    out.writeUTF(this.nom);
    out.writeUTF(this.prenom);
    out.writeObject(this.dateNaiss);
    out.writeInt(this.taille);
    out.writeUTF(this.codeSecret);
  }

  @Override
  public void readExternal(final ObjectInput in) throws IOException, ClassNotFoundException {
    this.nom = in.readUTF();
    this.prenom = in.readUTF();
    this.dateNaiss = (Date) in.readObject();
    this.taille = in.readInt();
    this.codeSecret = in.readUTF();
  }
}
```

Remarque : le mot clé transient est inutile avec une classe qui implémente l'interface Externalizable

Résultat :

```
Personne :
nom : Dupond
prenom : Jean
taille : 175
date naissance :
Fri Jun 13 00:00:00 CET 1975
codeSecret : 1234
```

Les données du flux de sérialisation sont facilement exploitables même si c'est un format binaire. Dans l'exemple, ci-dessus le code secret apparait en clair puisque c'est une chaîne de caractères.



```
    0001 0203 0405 0607 0809 0A0B 0C0D 0E0F   0123456789ABCDEF
00  ACED 0005 7372 0029 636F 6D2E 6A6D 646F   □i..sr.)com.jmdo
10  7564 6F75 782E 7465 7374 2E73 6572 6961   udoux.test.seria
20  6C69 7361 7469 6F6E 2E50 6572 736F 6E6E   lisation.Personn
30  6554 4600 6F12 1241 7A0C 0000 7870 770E   eTF.o..Az...xpw.
40  0006 4475 706F 6E64 0004 4A65 616E 7372   ..Dupond..Jeansr
50  000E 6A61 7661 2E75 7469 6C2E 4461 7465   ..java.util.Date
60  686A 8101 4B59 7419 0300 0078 7077 0800   hj□.KYt....xpw..
70  0000 2802 D1DD 8078 770A 0000 00AF 0004   ..(.ÑÝ€xw....¯..
80  3132 3334 78                              1234x
```

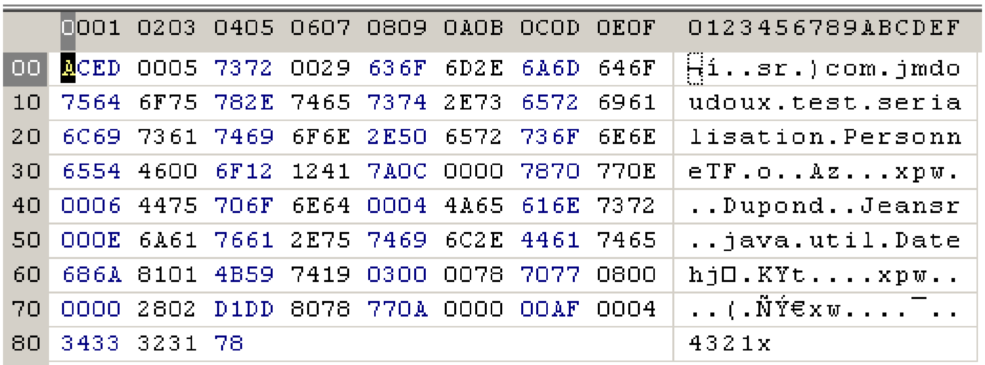
Le fait d'avoir le contrôle total sur les opérations de sérialisation/désérialisation permet par exemple de modifier les valeurs de certaines données de l'état de l'objet.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;
import java.util.Date;
```

```
public class Personne implements java.io.Externalizable {
  private String           nom        = "";
  private String           prenom     = "";
  private int              taille     = 0;
  private Date             dateNaiss  = null;
  private transient String codeSecret = "";

  public Personne() {
  }

  // ... getters et setters

  @Override
  public void writeExternal(final ObjectOutput out) throws IOException {
    out.writeUTF(this.nom);
    out.writeUTF(this.prenom);
    out.writeObject(this.dateNaiss);
    out.writeInt(this.taille);
    out.writeUTF(new StringBuilder(this.codeSecret).reverse().toString());
  }

  @Override
  public void readExternal(final ObjectInput in) throws IOException, ClassNotFoundException {
    this.nom = in.readUTF();
    this.prenom = in.readUTF();
    this.dateNaiss = (Date) in.readObject();
    this.taille = in.readInt();
    this.codeSecret = new StringBuilder(in.readUTF()).reverse().toString();
  }
}
```

Le résultat est le même mais la valeur contenue dans le flux n'est plus utilisable directement.



```
    0001 0203 0405 0607 0809 0A0B 0C0D 0E0F  0123456789ABCDEF
00  ACED 0005 7372 0029 636F 6D2E 6A6D 646F  ¬i..sr.)com.jmdo
10  7564 6F75 782E 7465 7374 2E73 6572 6961  udoux.test.seria
20  6C69 7361 7469 6F6E 2E50 6572 736F 6E6E  lisation.Personn
30  6554 4600 6F12 1241 7A0C 0000 7870 770E  eTF.o..Az...xpw.
40  0006 4475 706F 6E64 0004 4A65 616E 7372  ..Dupond..Jeansr
50  000E 6A61 7661 2E75 7469 6C2E 4461 7465  ..java.util.Date
60  686A 8101 4B59 7419 0300 0078 7077 0800  hj□.KYt....xpw..
70  0000 2802 D1DD 8078 770A 0000 00AF 0004  ..(.ÑÝ€xw....¯..
80  3433 3231 78                             4321x
```

Cet exemple est simpliste car il inverse simplement d'ordre des caractères de la chaîne : en réalité, il serait nécessaire d'utiliser un mécanisme de chiffrement/déchiffrement beaucoup plus robuste.

Lors de la désérialisation d'un objet Externalizable, une nouvelle instance est créée en invoquant le constructeur par défaut puis la méthode readExternal() est invoquée. Une classe qui implémente l'interface Externalizable doit donc obligatoirement proposer un constructeur par défaut, sinon une exception de type InvalidClassException est levée.

Résultat :

```
Caused by: java.io.InvalidClassException: Personne; no valid constructor
        at java.io.ObjectStreamClass.<init>(ObjectStreamClass.java:471)
        at java.io.ObjectStreamClass.lookup(ObjectStreamClass.java:310)
        at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1114)
        at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:330)
        at SerDeserPersonne.main(SerDeserPersonne.java:16)
```

L'obligation d'avoir un constructeur par défaut explique la raison pour laquelle une classe interne ne peut pas mettre en oeuvre le mécanisme utilisant l'interface Externalizable.

Lors de l'utilisation de l'interface Externalizable, les mécanismes de sérialisation/désérialisation mis en oeuvre doivent tenir compte de l'état des membres hérités, des valeurs par défaut, des membres transient et static , ...

Par exemple, une classe mère possède un champ et implémente l'interface Externalizable.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;

public class MaClasseMere implements Externalizable {

  public static final long serialVersionUID = 1;

  private String nom = null;

  public MaClasseMere() {
  }

  public MaClasseMere(final String nom) {
    super();
    this.nom = nom;
  }

  public String getNom() {
    return this.nom;
  }

  public void setNom(final String nom) {
    this.nom = nom;
  }

  @Override
  public void writeExternal(final ObjectOutput out) throws IOException {
    out.writeUTF(this.nom);
  }

  @Override
  public void readExternal(final ObjectInput in) throws IOException, ClassNotFoundException {
    this.nom = in.readUTF();
  }
}
```

La classe fille hérite de la classe mère et ajoute un nouveau champ de type int.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

public class MaClasseFille extends MaClasseMere {

  public static final long serialVersionUID = 1;

  private int valeur = 0;

  public MaClasseFille() {
  }

  public MaClasseFille(final String nom, final int valeur) {
    super(nom);
    this.valeur = valeur;
  }

  public int getValeur() {
    return this.valeur;
  }

  public void setValeur(final int valeur) {
    this.valeur = valeur;
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class SerDeserMaClasseFille {

  public static void main(final String[] args) {

    MaClasseFille maClasseFille = new MaClasseFille("nom1", 123);
    ObjectOutputStream oos = null;
    ObjectInputStream ois = null;

    try {
      final FileOutputStream fichierOut = new FileOutputStream("maclassefille.ser");
      oos = new ObjectOutputStream(fichierOut);
      oos.writeObject(maClasseFille);
      oos.flush();

      final FileInputStream fichierIn = new FileInputStream("maclassefille.ser");
      ois = new ObjectInputStream(fichierIn);
      maClasseFille = (MaClasseFille) ois.readObject();
      System.out.println("MaClasseFille : ");
      System.out.println("nom : " + maClasseFille.getNom());
      System.out.println("taille : " + maClasseFille.getValeur());
    } catch (final java.io.IOException e) {
      e.printStackTrace();
    } catch (final ClassNotFoundException e) {
      e.printStackTrace();
    } finally {
      try {
        if (ois != null) {
          ois.close();
        }
        if (oos != null) {
          oos.close();
        }
      } catch (final IOException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

Résultat :

```
MaClasseFille :
nom : nom1
taille : 0
```

La classe fille doit redéfinir les méthodes writeExternal() et readExternal() en invoquant leur équivalent de la classe mère.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;
```

```
public class MaClasseFille extends MaClasseMere {

  public static final long serialVersionUID = 1;

  private int valeur = 0;

  public MaClasseFille() {
  }

  public MaClasseFille(final String nom, final int valeur) {
    super(nom);
    this.valeur = valeur;
  }

  public int getValeur() {
    return this.valeur;
  }

  public void setValeur(final int valeur) {
    this.valeur = valeur;
  }

  @Override
  public void writeExternal(final ObjectOutput out) throws IOException {
    super.writeExternal(out);
    out.writeInt(this.valeur);
  }

  @Override
  public void readExternal(final ObjectInput in) throws IOException, ClassNotFoundException {
    super.readExternal(in);
    this.valeur = in.readInt();
  }
}
```

Résultat :

```
MaClasseFille :
nom : nom1
taille : 123
```

Il est possible que la classe mère n'implémente pas l'interface Externalizable et qu'il ne soit pas possible de la modifier.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

public class MaClasseMere {

  public static final long serialVersionUID = 1;

  protected String nom = null;

  public String getNom() {
    return this.nom;
  }

  public void setNom(final String nom) {
    this.nom = nom;
  }

  public MaClasseMere() {
  }

  public MaClasseMere(final String nom) {
    super();

    this.nom = nom;
  }
}
```

Dans ce cas, la classe fille doit implémenter l'interface Externalizable et redéfinir les méthodes writeExternal() et readExternal() pour tenir compte de l'état des membres de la classe fille et de la classe mère.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;

public class MaClasseFille extends MaClasseMere implements Externalizable {

  public static final long serialVersionUID = 1;

  private int valeur = 0;

  public MaClasseFille() {
  }

  public MaClasseFille(final String nom, final int valeur) {
    super(nom);
    this.valeur = valeur;
  }

  public int getValeur() {
    return this.valeur;
  }

  public void setValeur(final int valeur) {
    this.valeur = valeur;
  }

  @Override
  public void writeExternal(final ObjectOutput out) throws IOException {
    out.writeUTF(this.nom);
    out.writeInt(this.valeur);
  }

  @Override
  public void readExternal(final ObjectInput in) throws IOException, ClassNotFoundException {
    this.nom = in.readUTF();
    this.valeur = in.readInt();
  }
}
```

Lors de la sérialisation, les méthodes writeExternal() et readExternal() sont utilisées prioritairement aux méthodes writeObjet() et readObject().

La sérialisation en utilisant un Externalizable est généralement plus performante que la sérialisation standard car cette dernière requièrt l'utilisation de l'introspection pour déterminer les éléments à inclure.

En contrepartie, l'utilisation d'un Externalizable est moins sécurisée car les méthodes à redéfinir sont publiques alors que les méthodes de la sérialisation standard sont privées.

17.1.2.6. Les différences entre Serializable et Externalizable

Bien que ces deux interfaces soient utilisées pour sérialiser/désérialiser un objet, il existe plusieurs différences entre l'utilisation des interfaces Serializable et Externalizable :

- l'interface Serializable est un marqueur, il n'y a donc pas de méthode à implémenter par défaut. L'interface Externalizable hérite de Serializable et définit deux méthodes

- l'utilisation de Serializable implique l'utilisation du mécanisme de sérialisation par défaut (les mécanismes utilisés sont contrôlés par la plate-forme Java) alors que l'utilisation d'Externalizable implique l'utilisation de mécanismes de sérialisation personnalisés (Les mécanismes utilisés sont contrôlés par le développeur)
- l'utilisation d'Externalizable peut permettre de faciliter la compatibilité en cas de changement dans la classe mais en contrepartie chaque changement de la classe peut impliquer un changement dans le code des méthodes readExternal() et writeExternal()
- l'utilisation d'Externalizable est généralement plus performante car l'utilisation de Serializable pour une sérialisation standard implique l'utilisation de l'introspection. Ceci est particulièrement vrai pour les anciennes versions de la JVM
- les méthodes readExternal() et writeExternal() de l'interface Externalizable annulent et remplacent les méthodes readObject() et writeObject() de l'interface Serializable si elles existent

17.2. La documentation d'une classe sérialisable

L'outil javadoc propose de prendre en compte la documentation d'une classe sérialisable.

Son utilisation est importante pour permettre de documenter comment la classe est sérialisée actuellement : cela facilite le travail lors de la création d'une sous-classe ou lors d'évolutions dans la classe.

Javadoc propose, depuis sa version 1.2, trois tags dédiés à la documentation de la manière avec laquelle une classe est sérialisée :

Tag	Rôle
@serial	Ce tag s'utilise dans les commentaires Javadoc d'un champ qui est sérialisé par défaut
@serialField	Ce tag s'utilise dans les commentaires Javadoc du champ serialPersistentFields de type ObjectStreamField[] pour décrire les différentes données qui sont sérialisées. Il faut utiliser un tag par données
@serialData	Ce tag s'utilise dans les commentaires Javadoc des méthodes writeObject(), readObject(), writeExternal(), readExternal(), writeReplace() et readResolve() pour décrire quelles données sont sérialisées et dans quel ordre

Le tag @serial s'utilise dans les commentaires Javadoc d'un champ sérialisé par défaut.

Sa syntaxe est :

@serial [*field-description]* [include | exclude]

La description est optionnelle : elle permet d'expliquer le sens du champ et sa liste de valeurs acceptables.

Les arguments include et exclude permettent de préciser pour une classe ou package s'il doit être ajouté dans la page serialized-form. Par défaut :

- une classe public ou protected qui implémente l'interface Serializable est ajoutée dans la page serialized-form sauf si son package ou elle-même est marqué avec @serial exclude
- une classe private ou package-private qui implémente l'interface Serializable n'est pas ajoutée dans la page serialized-form sauf si son package ou elle-même est marqué avec @serial include

Le tag @serial au niveau de la classe est prioritaire sur celui au niveau du package.

Le doclet standard émet un warning sur les champs private d'une classe sérialisable qui ne sont pas marqués @serial.

Le tag @serialField permet de décrire un champ sérialisé encapsulé dans un objet de type ObjectStreamField. Il s'utilise dans les commentaires du champ serialPersistentFields de type ObjectStreamField[]. Il faut utiliser un tag @serialField pour chaque champ.

Sa syntaxe est :

@serialField field-name field-type field-description

Exemple :

```
/**
 * @serialField champ1 String description du champ1
 * @serialField champ2 String description du champ2
 */
private static final ObjectStreamField[] serialPersistentFields =
  { new ObjectStreamField("champ1", String.class),
    new ObjectStreamField("champ2", String.class)};
```

@serialData *data-description*

Le tag @serialData s'utilise pour décrire les données qui sont manuellement ajoutées dans les données sérialisées.

Il s'utilise sur les méthodes writeObject(), readObject(), writeExternal(), readExternal(), writeReplace() et readResolve().

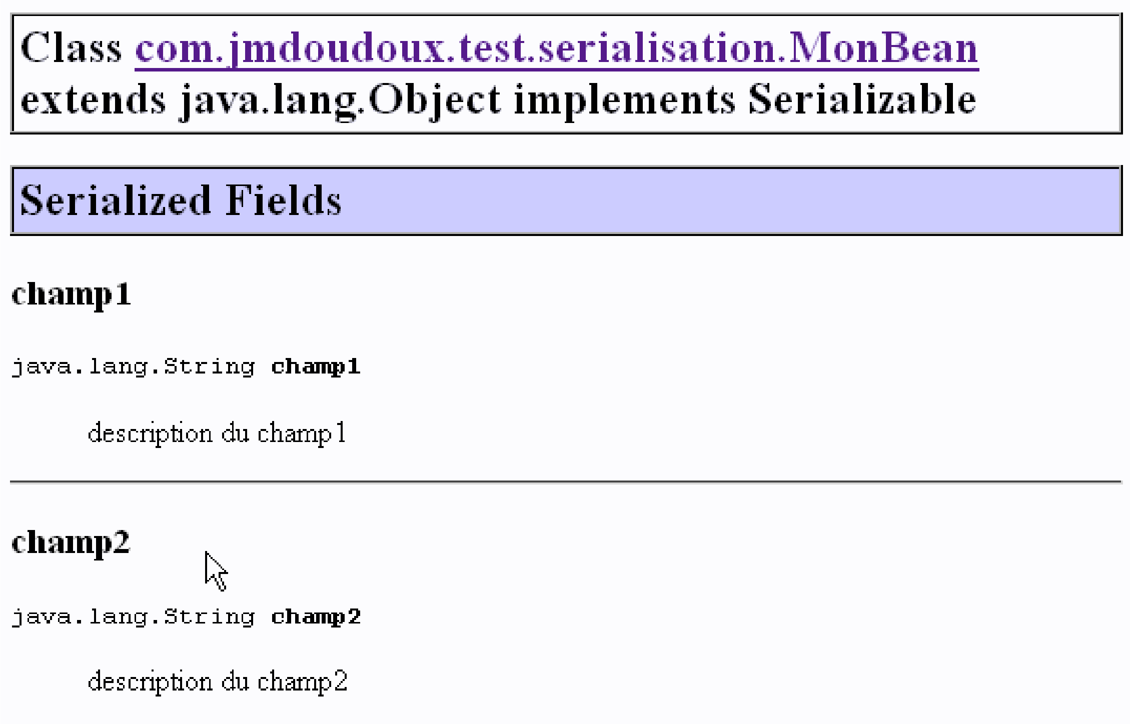
La description permet de préciser les données et l'ordre dans lequel elles sont ajoutées dans les données optionnelles.

Si un nouveau champ sérialisable est ajouté, il est possible d'utiliser le tag Javadoc @since pour préciser depuis quelle version.

Lors de la génération de la documentation Javadoc, une page dédié à la sérialisation, nommée serialized-form.html est générée. Elle contient toutes les classes qui implémentent Serializable ou Externalizable et pour chacune propose une description des champs sérialisés, des données optionnelles et des méthodes relatives à la sérialisation.

Il n'y a pas de lien proposé dans la barre de navigation vers cette page : il faut ouvrir la page d'une classe sérialisable et cliquer sur le lien « Serialized form » dans la section « See also ».

Le doclet par défaut utilise les métadonnées de la classe et les informations fournies par les tags @serial, @serialFields et @serialData pour générer la page.



Class com.jmdoudoux.test.serialisation.MonBean
extends java.lang.Object implements Serializable

Serialized Fields

champ1

java.lang.String **champ1**

description du champ1

champ2

java.lang.String **champ2**

description du champ2

17.3. La sérialisation et la sécurité

La sérialisation n'est pas sécurisée : même si le format par défaut est un format binaire, ce format est connu et documenté. Le contenu des données binaires peut assez facilement permettre de définir une classe qui permettra de lire le contenu du résultat de la sérialisation.

Un simple éditeur hexadécimal permet d'obtenir les valeurs des différents champs sérialisés même ceux qui sont déclarés privés.

Il ne faut pas sérialiser de données sensibles par le processus de sérialisation standard car cela rend public ces données. Une fois un objet sérialisé, les mécanismes d'encapsulation de Java ne sont plus mis en oeuvre : il est possible d'accéder aux champs private par exemple. Il faut soit :

- exclure ces champs de la sérialisation
- encrypter/décrypter ces données avec une personnalisation de la sérialisation
- encrypter tous les résultats de la sérialisation en utilisant la classe javax.crypto.SealedObject ou la classe java.security.SignedObject qui implémente l'interface Serializable. Elles permettent d'encrypter et de signer un objet
- sérialiser une instance d'un proxy qui ne contient pas ces données

Le mécanisme de désérialisation permet de créer de nouvelles instances : tous les contrôles qui sont faits dans le constructeur doivent aussi être faits lors de la désérialisation.

17.4. La sérialisation en XML

A partir de Java 1.4, le JDK permet de sérialiser un objet Java en XML plutôt que sous un format binaire en utilisant les classes XMLEncoder et XMLDecoder.

La sérialisation binaire requiert une analogie entre les méthodes readObject() et la méthode writeObject() correspondante. La sérialisation XML repose sur une approche différente qui se base sur l'API publique d'une classe. C'est la raison pour laquelle l'utilisation de cette fonctionnalité ne peut s'appliquer par défaut que sur des objets qui respectent la convention JavaBeans.

La sérialisation XML présente quelques avantages :

- une meilleure portabilité notamment dans le cas d'échanges entre JVM de différents fournisseurs
- facilité de traitements et d'échanges sur le réseau grâce à l'utilisation du format XML
- le processus de sérialisation ne tient pas compte des champs qui ont leur valeur par défaut

Elle a aussi plusieurs inconvénients :

- ne peut s'utiliser par défaut que sur des objets qui respectent la convention JavaBeans (constructeur par défaut, getter/setter pour tous les attributs, ...)
- la taille des données sérialisées est plus importante que leur équivalent binaire

Attention, en Java 5, les énumérations ne sont pas supportées par les classes XMLEncoder et XMLDecoder.

17.4.1. La classe XMLEncoder

La classe java.beans.XMLEncoder permet de sérialiser un objet en XML.

La classe XMLEncoder permet de sérialiser l'état d'un JavaBean dans un document XML encodé en UTF-8. La sérialisation XML ne prend en compte que les champs pour lesquels il existe un getter et un setter public. Le mécanisme de sérialisation XML optimise le contenu du document XML en omettant les champs dont la valeur est celle par défaut.

La classe XMLEncoder possède deux constructeurs :

Constructeur	Rôle

XMLEncoder(OutputStream out)	Créer une nouvelle instance qui utilise le flux en paramètre pour écrire le résultat de la sérialisation
XMLEncoder(OutputStream out, String charset, boolean declaration, int indentation)	Créer une nouvelle instance qui utilise le flux en paramètre pour écrire le résultat de la sérialisation en précisant le charset, un booléen qui précise si le document XML doit contenir la déclaration et la taille de l'indentation.

Le constructeur qui attend plusieurs paramètres a été ajouté dans Java 7

Exemple (code Java 1.4) :

```
package fr.jmdoudoux.dej.serialisation;

import java.beans.XMLEncoder;
import java.io.FileOutputStream;
import java.util.Date;

public class SerializerPersonneXML {

  public static void main(final String argv[]) {
    final Personne personne = new Personne("Dupond", "Jean", 175, "1234", new Date());
    XMLEncoder encoder = null;

    try {
      encoder = new XMLEncoder(new BufferedOutputStream(
        new FileOutputStream("personne.xml")));
      encoder.writeObject(personne);
      encoder.flush();
    } catch (final java.io.IOException e) {
      e.printStackTrace();
    } finally {
      if (encoder != null) {
        encoder.close();
      }
    }
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0_45" class="java.beans.XMLDecoder">
 <object class="fr.jmdoudoux.dej.serialisation.Personne">
  <void property="dateNaiss">
   <object class="java.util.Date">
    <long>1391419419836</long>
   </object>
  </void>
  <void property="nom">
   <string>Dupond</string>
  </void>
  <void property="prenom">
   <string>Jean</string>
  </void>
  <void property="taille">
   <int>175</int>
  </void>
 </object>
</java>
```

La structure du document XML généré utilise plusieurs conventions :

- le tag racine est le tag <java>. La propriété version permet de préciser la version de la plate-forme Java utilisée pour générer le document. La propriété class permet de préciser le nom pleinement qualifié de la classe
- chaque propriété est définie dans un tag dédié
- un tag void désigne un élément qui est passé en argument de l'élément englobant

- un tag object désigne un objet qui est passé en argument de l'élément englobant en invoquant un setter
- les valeurs sont encapsulées dans des tags selon leur type (<int>, <string>, <long>, ...)
- les attributs id et idref permettent de définir et faire référence à d'autres éléments

La classe XMLEncoder clone le graphe d'objets à sérialiser pour enregistrer les opérations nécessaires et ainsi pouvoir créer le document XML Lors de la sérialisation, la classe XMLEncoder applique un algorithme qui permet d'éviter de sérialiser les propriétés qui ont leur valeur par défaut : ceci permet de rendre le document XML plus compact.

Exemple (code Java 1.4) :

```
package fr.jmdoudoux.dej.serialisation;

import java.beans.XMLEncoder;
import java.io.FileOutputStream;

public class SerializerPersonneXML {

  public static void main(final String argv[]) {
    final Personne personne = new Personne();

    XMLEncoder encoder = null;

    try {
      encoder = new XMLEncoder(new BufferedOutputStream(new FileOutputStream("personne.xml")));
      encoder.writeObject(personne);
      encoder.flush();
    } catch (final java.io.IOException e) {
      e.printStackTrace();
    } finally {
      if (encoder != null) {
        encoder.close();
      }
    }
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0_45" class="java.beans.XMLDecoder">
 <object class="fr.jmdoudoux.dej.serialisation.Personne"/>
</java>
```

17.4.2. La classe XMLDecoder

La classe java.beans.XMLDecoder permet de désérialiser un objet à partir d'un document XML généré avec la classe XMLEncoder.

Elle possède plusieurs constructeurs :

Constructeur	Rôle
XMLDecoder(InputStream in)	Créer un nouveau décodeur pour désérialiser le document XML lu du flux en paramètre
XMLDecoder(InputStream in, Object owner)	Créer un nouveau décodeur pour désérialiser le document XML lu du flux en paramètre
XMLDecoder(InputStream in, Object owner, ExceptionListener exceptionListener)	Créer un nouveau décodeur pour désérialiser le document XML lu du flux en paramètre
XMLDecoder(InputStream in, Object owner, ExceptionListener exceptionListener, ClassLoader cl)	Créer un nouveau décodeur pour désérialiser le document XML lu du flux en paramètre (depuis Java 1.5)
XMLDecoder(InputSource is)	Créer un nouveau décodeur pour désérialiser le document XML lu du flux en paramètre (depuis Java 7)

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.beans.XMLDecoder;
import java.io.BufferedInputStream;
import java.io.FileInputStream;

public class DeserializerPersonneXML {

  public static void main(final String argv[]) {
    XMLDecoder decoder = null;

    try {
      decoder = new XMLDecoder(new BufferedInputStream(new FileInputStream("personne.xml")));
      final Personne personne = (Personne) decoder.readObject();
      System.out.println(personne);
    } catch (final Exception e) {
      e.printStackTrace();
    } finally {
      if (decoder != null) {
        decoder.close();
      }
    }
  }
}
```

Résultat :

```
Personne
[nom=Dupond, prenom=Jean, taille=175, dateNaiss=Wed Feb 12 20:45:45 CET 2013]
```

L'utilisation de la sérialisation XML n'est possible par défaut que sur des objets dont la classe respecte la convention Javabeans.

Notamment une exception est levée si la classe ne possède pas de constructeur par défaut.

Exemple :

```
java.lang.InstantiationException: fr.jmdoudoux.dej.serialisation.Personne
Continuing ...
java.lang.Exception: XMLEncoder: discarding statement XMLEncoder.writeObject(Personne);
Continuing ...
```

Dans ce cas, le fichier XML est créé mais il contient uniquement le prologue et le tag racine :

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0_45" class="java.beans.XMLDecoder">
</java>
```

Seuls les champs qui possèdent un getter et un setter sont pris en compte lors des opérations de sérialisation et désérialisation.

17.4.3. La personnalisation de la sérialisation

La classe XMLEncoder utilise une instance de type PersistenceDelegate pour sérialiser un objet en XML. Si aucune instance de type PersistenceDelegate n'est explicitement fournie, alors la classe XMLEncoder utilise une instance de type DefaultPersistenceDelegate qui implémente la stratégie de sérialisation par défaut.

17.4.3.1. La classe PersistenceDelegate

La classe abstraite java.beans.PersistenceDelegate a pour rôle d'exprimer l'état d'une instance au travers de l'utilisation de ses méthodes publiques. Au lieu de réaliser ses actions dans la classe de l'instance à sérialiser, comme c'est fait pour la sérialisation binaire, la classe XMLEncoder délègue ces traitements à une instance de type PersitanceDelegate.

La classe PersistenceDelegate permet de contrôler certaines étapes de la sérialisation :

- création d'une instance de l'objet par invocation d'un constructeur ou d'une fabrique
- déterminer si une instance peut être transformée en une autre
- initialisation de l'instance de l'objet

Elle possède plusieurs méthodes :

Méthode	Rôle
protected void initialize(Class<?> type, Object oldInstance, Object newInstance, Encoder out)	Initialiser la nouvelle instance éventuellement à partir des données de la première instance fournie en paramètre
protected abstract Expression instantiate(Object oldInstance, Encoder out)	Renvoyer une instance de type Expression correspondant à l'objet fourni en paramètre
protected boolean mutatesTo(Object oldInstance, Object newInstance)	Renvoyer un booléen qui précise s'il est possible de créer une copie de la première instance et appliquer des opérations pour obtenir la seconde instance
void writeObject(Object oldInstance, Encoder out)	Réaliser les opérations déléguées relatives à la sérialisation de l'instance fournie en paramètre

17.4.3.2. La classe DefaultPersistenceDelegate

La classe java.beans.DefaultPersistenceDelegate hérite de la classe java.beans.PersistenceDelegate pour en proposer une implémentation concrète.

Cette classe est utilisée par défaut pour des classes qui doivent respecter la convention Javabeans notamment la présence d'un constructeur par défaut et la présence de getter/setter pour les propriétés.

Elle possède deux constructeurs :

Constructeur	Rôle
DefaultPersistenceDelegate()	
DefaultPersistenceDelegate(String[] constructorPropertyNames)	Créer une instance qui par défaut va invoquer le constructeur avec les valeurs des propriétés de l'objet à traiter en paramètre

Elle redéfinit plusieurs méthodes :

Méthode	Rôle
protected void initialize(Class<?> type, Object oldInstance, Object newInstance, Encoder out)	Initialiser les valeurs des champs de la nouvelle instance avec celles de l'instance fournie en paramètres en utilisant leurs getter/setter grâce à l'introspection
protected Expression instantiate(Object oldInstance, Encoder out)	Envoyer une instance de type Expression dont le nom de méthode est "new" pour préciser que c'est le constructeur qui doit être invoqué
protected boolean mutatesTo(Object oldInstance, Object newInstance)	Si au moins un constructeur est défini et la méthode equals() redéfinie alors renvoie la valeur de l'invocation de oldInstance.equals(newInstance)

17.4.3.3. Empêcher la sérialisation d'un attribut

Pour empêcher la sérialisation en XML d'un champ, il est inutile de lui ajouter le modificateur transient car il n'est pas pris en compte dans ce cas.

Il faut soit :

- ne pas proposer de getter ou de setter sur le champ : cette solution est rarement utilisable
- utiliser un PropertyDescriptor du champ concerné pour mettre la valeur de la propriété transient à true.

Il faut utiliser l'API Introspection pour obtenir l'instance de type BeanInfo de la classe du bean. La méthode getPropertyDescriptors() permet d'obtenir un tableau de type PropertyDescriptor, une occurrence pour chaque champ. Il faut itérer sur ce tableau pour trouver l'occurrence du champ concerné. Enfin, il faut invoquer sa méthode setValue() en lui passant en paramètre « transient » et Boolean.TRUE.

Exemple (code Java 5.0) :

```java
package fr.jmdoudoux.dej.serialisation;

import java.beans.BeanInfo;
import java.beans.IntrospectionException;
import java.beans.Introspector;
import java.beans.PropertyDescriptor;
import java.beans.XMLEncoder;
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;

public class SerializerMonBean {

  public static void main(final String[] args) {
    final MonBean monBean = new MonBean();
    monBean.setChamp1("valeur1");
    monBean.setChamp2("valeur2");
    XMLEncoder encoder = null;

    try {
      final BeanInfo info = Introspector.getBeanInfo(MonBean.class);
      final PropertyDescriptor[] propertyDescriptors = info.getPropertyDescriptors();
      for (final PropertyDescriptor descriptor : propertyDescriptors) {
        if (descriptor.getName().equals("champ2")) {
          descriptor.setValue("transient", Boolean.TRUE);
          break;
        }
      }

      encoder = new XMLEncoder(new BufferedOutputStream(new FileOutputStream("monbean.xml")));
      encoder.writeObject(monBean);
      encoder.flush();
    } catch (final java.io.IOException e) {
      e.printStackTrace();
    } catch (final IntrospectionException e) {
      e.printStackTrace();
    } finally {
      if (encoder != null) {
        encoder.close();
      }
    }
  }
}
```

Résultat :

```xml
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0_43" class="java.beans.XMLDecoder">
 <object class="fr.jmdoudoux.dej.serialisation.MonBean">
  <void property="champ1">
   <string>valeur1</string>
  </void>
 </object>
</java>
```

Plutôt que de modifier le BeanInfo par défaut associé à la classe, il est possible de définir explicitement cette classe.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.beans.IntrospectionException;
import java.beans.PropertyDescriptor;
import java.beans.SimpleBeanInfo;

public class MonBeanBeanInfo extends SimpleBeanInfo {

  private PropertyDescriptor[] descriptors;

  public MonBeanBeanInfo() {
    try {
      final PropertyDescriptor champ1Descriptor = new PropertyDescriptor("champ1",
        MonBean.class);
      final PropertyDescriptor champ2Descriptor = new PropertyDescriptor("champ2",
        MonBean.class);
      champ2Descriptor.setValue("transient", Boolean.TRUE);

      this.descriptors = new PropertyDescriptor[] { champ1Descriptor, champ2Descriptor };
    } catch (final IntrospectionException ex) {
      ex.printStackTrace();
    }
  }

  @Override
  public PropertyDescriptor[] getPropertyDescriptors() {
    return this.descriptors;
  }
}
```

Les traitements pour sérialiser une instance sont alors plus simple.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.beans.XMLEncoder;
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;

public class SerializerMonBean {

  public static void main(final String[] args) {
    final MonBean monBean = new MonBean();
    monBean.setChamp1("valeur1");
    monBean.setChamp2("valeur2");
    XMLEncoder encoder = null;

    try {
      encoder = new XMLEncoder(new BufferedOutputStream(new FileOutputStream("monbean.xml")));
      encoder.writeObject(monBean);
      encoder.flush();
    } catch (final java.io.IOException e) {
      e.printStackTrace();
    } finally {
      if (encoder != null) {
        encoder.close();
      }
    }
  }
}
```

17.4.3.4. Sérialiser des attributs sans accesseur/modificateur standard

Parfois, un champ possède un accesseur et un modificateur mais leurs noms ne respectent pas la convention JavaBean.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

public class MonBean {

  private String champ1;
  private String champ2;

  public String getChamp1() {
    return this.champ1;
  }

  public String obtenirChamp2() {
    return this.champ2;
  }

  public void modifierChamp2(final String champ2) {
    this.champ2 = champ2;
  }

  public void setChamp1(final String champ1) {
    this.champ1 = champ1;
  }

  @Override
  public String toString() {
    return "MonBean [champ1=" + this.champ1 + ", champ2=" + this.champ2 + "]";
  }
}
```

Lors de la sérialisation en XML d'une instance de cette classe, le champ2 est ignoré puisse qu'il ne possède pas de getter/setter standard.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.beans.XMLEncoder;
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;

public class SerializerMonBean {

  public static void main(final String[] args) {
    final MonBean monBean = new MonBean();
    monBean.setChamp1("valeur1");
    monBean.modifierChamp2("valeur2");
    XMLEncoder encoder = null;

    try {
      encoder = new XMLEncoder(new BufferedOutputStream(new FileOutputStream("monbean.xml")));
      encoder.writeObject(monBean);
      encoder.flush();
    } catch (final java.io.IOException e) {
      e.printStackTrace();
    } finally {
      if (encoder != null) {
        encoder.close();
      }
    }
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0_43" class="java.beans.XMLDecoder">
 <object class="fr.jmdoudoux.dej.serialisation.MonBean">
  <void property="champ1">
   <string>valeur1</string>
  </void>
 </object>
</java>
```

Pour forcer l'API à utiliser l'accesseur et le modificateur non standard, il faut les préciser dans le PropertyDescriptor du champ correspondant encapsulé dans la classe de type BeanInfo liée à la classe du bean.

La classe doit implémenter l'interface BeanInfo ou hériter de la classe SimpleBeanInfo. Son nom doit obligatoirement être composé du nom de la classe suivi de BeanInfo

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.beans.IntrospectionException;
import java.beans.PropertyDescriptor;
import java.beans.SimpleBeanInfo;

public class MonBeanBeanInfo extends SimpleBeanInfo {

  private PropertyDescriptor[] descriptors;

  public MonBeanBeanInfo() {
    try {

      final PropertyDescriptor champ1Descriptor = new PropertyDescriptor("champ1",
        MonBean.class);
      final PropertyDescriptor champ2Descriptor = new PropertyDescriptor("champ2",
        MonBean.class, "obtenirChamp2", "modifierChamp2");

      this.descriptors = new PropertyDescriptor[] { champ1Descriptor, champ2Descriptor };
    } catch (final IntrospectionException ex) {
      ex.printStackTrace();
    }
  }

  @Override
  public PropertyDescriptor[] getPropertyDescriptors() {
    return this.descriptors;
  }
}
```

Les traitements pour sérialiser en XML une instance de la classe reste classique.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0_43" class="java.beans.XMLDecoder">
 <object class="fr.jmdoudoux.dej.serialisation.MonBean">
  <void property="champ1">
   <string>valeur1</string>
  </void>
  <void method="modifierChamp2">
   <string>valeur2</string>
  </void>
 </object>
</java>
```

Dans le fichier XML contenant le résultat de la sérialisation, le tag qui correspond au champ champ2 contient une propriété method dont la valeur est le nom de la méthode définie comme étant le modificateur du champ. C'est cette méthode qui sera invoquée lors de la désérialisation pour initialiser la valeur du champ.

17.4.3.5. Sérialiser une classe sans constructeur par défaut

Par défaut la sérialisation XML requiert un constructeur par défaut dans la classe de l'instance à traiter. La convention JavaBean impose elle-même un tel constructeur. Cependant toutes les classes n'en sont pas forcement pourvues soit parce qu'elles définissent explicitement d'autres constructeurs soit parce que l'obtention d'une nouvelle instance passe par une fabrique.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

public class MonBean {

  private String champ1;
  private String champ2;

  private MonBean() {
  }

  public static MonBean creerInstance() {
    return new MonBean();
  }

  public String getChamp1() {
    return this.champ1;
  }

  public String getChamp2() {
    return this.champ2;
  }

  public void setChamp2(final String champ2) {
    this.champ2 = champ2;
  }

  public void setChamp1(final String champ1) {
    this.champ1 = champ1;
  }

  @Override
  public String toString() {
    return "MonBean [champ1=" + this.champ1 + ", champ2=" + this.champ2 + "]";
  }
}
```

Dans ce cas, la sérialisation XML par défaut échoue car elle ne trouve pas le constructeur par défaut en utilisant l'introspection.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.beans.XMLEncoder;
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;

public class SerializerMonBean {

  public static void main(final String[] args) {
    final MonBean monBean = MonBean.creerInstance();
    monBean.setChamp1("valeur1");
    monBean.setChamp2("valeur2");
    XMLEncoder encoder = null;

    try {
      encoder = new XMLEncoder(new BufferedOutputStream(new FileOutputStream("monbean.xml")));
      encoder.writeObject(monBean);
      encoder.flush();
    } catch (final java.io.IOException e) {
      e.printStackTrace();
    } finally {
      if (encoder != null) {
        encoder.close();
      }
    }
  }
}
```

Résultat :

```
java.lang.IllegalAccessException: Class sun.reflect.misc.Trampoline can not access a member
of class fr.jmdoudoux.dej.serialisation.MonBean with modifiers "private"
Continuing ...
java.lang.Exception: XMLEncoder: discarding statement XMLEncoder.writeObject(MonBean);
Continuing ...
```

Pour indiquer qu'il faut utiliser autre chose que le constructeur par défaut pour obtenir une instance, il faut définir un PersistenceDelegate personnalisé.

Il faut redéfinir la méthode instanciate() pour qu'elle renvoie une instance de type java.beans.Expression. à laquelle on a précisé le nom de la méthode à invoquer pour obtenir une nouvelle instance.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.beans.DefaultPersistenceDelegate;
import java.beans.Encoder;
import java.beans.Expression;
import java.beans.XMLEncoder;
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;

public class SerializerMonBean {

  public static void main(final String[] args) {
    final MonBean monBean = MonBean.creerInstance();
    monBean.setChamp1("valeur1");
    monBean.setChamp2("valeur2");
    XMLEncoder encoder = null;

    try {
      encoder = new XMLEncoder(new BufferedOutputStream(new FileOutputStream("monbean.xml")));
      encoder.setPersistenceDelegate(MonBean.class, new DefaultPersistenceDelegate() {
        @Override
        protected Expression instantiate(final Object oldInstance, final Encoder out) {
          return new Expression(oldInstance, MonBean.class, "creerInstance", null);
        }
      });

      encoder.writeObject(monBean);
      encoder.flush();
    } catch (final java.io.IOException e) {
      e.printStackTrace();
    } finally {
      if (encoder != null) {
        encoder.close();
      }
    }
  }
}
```

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0_43" class="java.beans.XMLDecoder">
 <object class="fr.jmdoudoux.dej.serialisation.MonBean" method="creerInstance">
```

```
  <void property="champ1">
   <string>valeur1</string>
  </void>
  <void property="champ2">
   <string>valeur2</string>
  </void>
 </object>
</java>
```

Dans le fichier XML contenant le résultat de la sérialisation, le nom de la méthode à invoquer est fourni comme valeur à l'attribut method.

Il est fréquent que la fabrique ou le constructeur à invoquer requière des paramètres.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

public class MonBean {

  private String champ1;
  private String champ2;

  private MonBean() {
  }

  public static MonBean creerInstance(final String champ1, final String champ2) {
    final MonBean resultat = new MonBean();
    resultat.setChamp1(champ1);
    resultat.setChamp2(champ2);
    return resultat;
  }

  public String getChamp1() {
    return this.champ1;
  }

  public String getChamp2() {
    return this.champ2;
  }

  public void setChamp2(final String champ2) {
    this.champ2 = champ2;
  }

  public void setChamp1(final String champ1) {
    this.champ1 = champ1;
  }

  @Override
  public String toString() {
    return "MonBean [champ1=" + this.champ1 + ", champ2=" + this.champ2 + "]";
  }
}
```

Dans cas, il faut extraire les valeurs à passer en paramètre et les fournir sous la forme d'un tableau d'objets en paramètre du constructeur de la classe Expression.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.beans.DefaultPersistenceDelegate;
import java.beans.Encoder;
import java.beans.Expression;
import java.beans.XMLEncoder;
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;
```

```
public class SerializerMonBean {

  public static void main(final String[] args) {
    final MonBean monBean = MonBean.creerInstance("valeur1", "valeur2");
    XMLEncoder encoder = null;

    try {
      encoder = new XMLEncoder(new BufferedOutputStream(
        new FileOutputStream("monbean.xml")));
      encoder.setPersistenceDelegate(MonBean.class, new DefaultPersistenceDelegate() {
        @Override
        protected Expression instantiate(final Object oldInstance, final Encoder out) {
          final String valeur1 = ((MonBean) oldInstance).getChamp1();
          final String valeur2 = ((MonBean) oldInstance).getChamp2();
          return new Expression(oldInstance, MonBean.class, "creerInstance",
            new Object[] { valeur1, valeur2 });
        }

      });

      encoder.writeObject(monBean);
      encoder.flush();
    } catch (final java.io.IOException e) {
      e.printStackTrace();
    } finally {
      if (encoder != null) {
        encoder.close();
      }
    }
  }
}
```

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0_43" class="java.beans.XMLDecoder">
 <object class="fr.jmdoudoux.dej.serialisation.MonBean" method="creerInstance">
  <string>valeur1</string>
  <string>valeur2</string>
 </object>
</java>
```

17.4.3.6. Les arguments du constructeur sont des champs de la classe

Parfois, la classe à sérialiser ne possède que des constructeurs qui attendent des paramètres.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

public class MonBean {

  private String champ1;
  private String champ2;

  public MonBean(final String champ1, final String champ2) {
    this.champ1 = champ1;
    this.champ2 = champ2;
  }

  public String getChamp1() {
    return this.champ1;
  }

  public String getChamp2() {
    return this.champ2;
  }

  public void setChamp2(final String champ2) {
    this.champ2 = champ2;
  }
  public void setChamp1(final String champ1) {
    this.champ1 = champ1;
  }

  @Override
  public String toString() {
    return "MonBean [champ1=" + this.champ1 + ", champ2=" + this.champ2 + "]";
  }
}
```

Pour sérialiser en XML une instance de cette classe, il est nécessaire de fournir une nouvelle instance de DefaultPersistenceDelegate en lui passant en paramètres un tableau de chaînes de caractères qui va contenir le nom de chacun des champs à fournir en paramètre du constructeur.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.beans.DefaultPersistenceDelegate;
import java.beans.XMLEncoder;
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;

public class SerializerMonBean {

  public static void main(final String[] args) {
    final MonBean monBean = new MonBean("valeur1", "valeur2");
    XMLEncoder encoder = null;

    try {
      encoder = new XMLEncoder(new BufferedOutputStream(
        new FileOutputStream("monbean.xml")));
      encoder.setPersistenceDelegate(MonBean.class,
        new DefaultPersistenceDelegate(new String[] { "champ1", "champ2" }));

      encoder.writeObject(monBean);
      encoder.flush();
    } catch (final java.io.IOException e) {
      e.printStackTrace();
    } finally {
      if (encoder != null) {
        encoder.close();
      }
    }
  }
}
```

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0_43" class="java.beans.XMLDecoder">
 <object class="fr.jmdoudoux.dej.serialisation.MonBean">
  <string>valeur1</string>
  <string>valeur2</string>
 </object>
</java>
```

17.4.3.7. Les arguments du constructeur ne sont pas des champs de la classe

Dans le cas où les arguments à passer au constructeur ne sont pas des attributs de la classe, il faut définir un objet de type PersistenceDelegate dans lequel la méthode instantiate() est redéfinie.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.beans.Encoder;
import java.beans.Expression;
import java.beans.PersistenceDelegate;
import java.beans.XMLEncoder;
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;

public class SerializerMonBean {

  public static void main(final String[] args) {
    final MonBean monBean = new MonBean("valeur1", "valeur2");
    XMLEncoder encoder = null;

    try {
      encoder = new XMLEncoder(new BufferedOutputStream(
        new FileOutputStream("monbean.xml")));
      encoder.setPersistenceDelegate(MonBean.class, new PersistenceDelegate() {
        @Override
        protected Expression instantiate(final Object oldInstance, final Encoder out) {
          return new Expression(oldInstance, oldInstance.getClass(), "new",
            new Object[] { "valeur3", "valeur4" });
        }
      });

      encoder.writeObject(monBean);
      encoder.flush();
    } catch (final java.io.IOException e) {
      e.printStackTrace();
    } finally {
      if (encoder != null) {
        encoder.close();
      }
    }
  }
}
```

Il faut préciser la chaîne de caractères « new » comme nom de méthode pour indiquer que c'est le constructeur qui doit être invoqué.

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0_43" class="java.beans.XMLDecoder">
 <object class="fr.jmdoudoux.dej.serialisation.MonBean">
  <string>valeur3</string>
  <string>valeur4</string>
 </object>
</java>
```

17.4.3.8. La gestion des exceptions

Par défaut, les classes XMLEncoder et XMLDecoder interceptent les exceptions levées durant leurs traitements.

Il peut cependant être nécessaire d'être informé de la levée d'une exception pour permettre sa gestion.

La méthode setExceptionListener() de la classe XMLEncoder permet d'enregistrer un listener pour la gestion des exceptions.

Le listener est de type ExceptionListener : il ne déclare qu'une seule méthode exceptionThrown() qui prend en paramètre l'exception levée.

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.beans.ExceptionListener;
import java.beans.XMLEncoder;
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;
import java.util.Date;

public class SerializerPersonneXML {

  public static void main(final String argv[]) {
    final Personne personne = new Personne("Dupond", "Jean", 175, "1234", new Date());
    XMLEncoder encoder = null;

    try {
      encoder = new XMLEncoder(new BufferedOutputStream(new FileOutputStream("personne.xml")));
      encoder.setExceptionListener(new ExceptionListener() {
        @Override
        public void exceptionThrown(final Exception ex) {
          ex.printStackTrace();
        }
      });

      encoder.writeObject(personne);
      encoder.flush();
    } catch (final java.io.IOException e) {
      e.printStackTrace();
    } finally {
      if (encoder != null) {
        encoder.close();
      }
    }
  }
}
```

Résultat :

```
java.lang.InstantiationException: fr.jmdoudoux.dej.serialisation.Personne
    at java.lang.Class.newInstance0(Class.java:340)
    at java.lang.Class.newInstance(Class.java:308)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
    at java.lang.reflect.Method.invoke(Method.java:597)
    at sun.reflect.misc.Trampoline.invoke(MethodUtil.java:37)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
    at java.lang.reflect.Method.invoke(Method.java:597)
    at sun.reflect.misc.MethodUtil.invoke(MethodUtil.java:244)
    at java.beans.Statement.invokeInternal(Statement.java:239)
    at java.beans.Statement.access$000(Statement.java:39)
    at java.beans.Statement$2.run(Statement.java:140)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.beans.Statement.invoke(Statement.java:137)
    at java.beans.Expression.getValue(Expression.java:98)
    at java.beans.Encoder.getValue(Encoder.java:85)
    at java.beans.Encoder.get(Encoder.java:200)
    at java.beans.PersistenceDelegate.writeObject(PersistenceDelegate.java:94)
    at java.beans.Encoder.writeObject(Encoder.java:54)
    at java.beans.XMLEncoder.writeObject(XMLEncoder.java:257)
    at java.beans.Encoder.writeExpression(Encoder.java:279)
    at java.beans.XMLEncoder.writeExpression(XMLEncoder.java:372)
    at java.beans.PersistenceDelegate.writeObject(PersistenceDelegate.java:97)
    at java.beans.Encoder.writeObject(Encoder.java:54)
    at java.beans.XMLEncoder.writeObject(XMLEncoder.java:257)
    at java.beans.Encoder.writeObject1(Encoder.java:206)
    at java.beans.Encoder.cloneStatement(Encoder.java:219)
    at java.beans.Encoder.writeStatement(Encoder.java:250)
    at java.beans.XMLEncoder.writeStatement(XMLEncoder.java:331)
    at java.beans.XMLEncoder.writeObject(XMLEncoder.java:260)
    at fr.jmdoudoux.dej.serialisation.SerializerPersonneXML.main
(SerializerPersonneXML.java:24)
java.lang.Exception: XMLEncoder: discarding statement XMLEncoder.writeObject(Personne);
    at java.beans.XMLEncoder.writeStatement(XMLEncoder.java:344)
    at java.beans.XMLEncoder.writeObject(XMLEncoder.java:260)
    at fr.jmdoudoux.dej.serialisation.SerializerPersonneXML.main
(SerializerPersonneXML.java:24)
Caused by: java.lang.RuntimeException: failed to evaluate: <unbound>=Class.new();
```

```
    at java.beans.Encoder.getValue(Encoder.java:89)
    at java.beans.Encoder.get(Encoder.java:200)
    at java.beans.PersistenceDelegate.writeObject(PersistenceDelegate.java:94)
    at java.beans.Encoder.writeObject(Encoder.java:54)
    at java.beans.XMLEncoder.writeObject(XMLEncoder.java:257)
    at java.beans.Encoder.writeExpression(Encoder.java:279)
    at java.beans.XMLEncoder.writeExpression(XMLEncoder.java:372)
    at java.beans.PersistenceDelegate.writeObject(PersistenceDelegate.java:97)
    at java.beans.Encoder.writeObject(Encoder.java:54)
    at java.beans.XMLEncoder.writeObject(XMLEncoder.java:257)
    at java.beans.Encoder.writeObject1(Encoder.java:206)
    at java.beans.Encoder.cloneStatement(Encoder.java:219)
    at java.beans.Encoder.writeStatement(Encoder.java:250)
    at java.beans.XMLEncoder.writeStatement(XMLEncoder.java:331)
        ... 2 more
```

Pour associer un gestionnaire d'exceptions à une instance de type XMLDecoder, il faut soit :

- fournir l'instance de type ExceptionListener en paramètre du constructeur
- invoquer la méthode setExceptionListener avec l'instance de type ExceptionListener en paramètre

Exemple :

```
package fr.jmdoudoux.dej.serialisation;

import java.beans.ExceptionListener;
import java.beans.XMLDecoder;
import java.io.BufferedInputStream;
import java.io.FileInputStream;

public class DeserializerPersonneXML {

  public static void main(final String argv[]) {
    XMLDecoder decoder = null;

    try {
      decoder = new XMLDecoder(new BufferedInputStream(new FileInputStream("personne.xml")));
      decoder.setExceptionListener(new ExceptionListener() {
        @Override
        public void exceptionThrown(final Exception ex) {
          ex.printStackTrace();
        }
      });
      final Personne personne = (Personne) decoder.readObject();
      System.out.println(personne);
    } catch (final Exception e) {
      e.printStackTrace();
    } finally {
      if (decoder != null) {
        decoder.close();
      }
    }
  }
}
```

18. L'interaction avec le réseau

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Les échanges avec le réseau sont devenus omniprésents dans les applications et entre les applications. Ils permettent notamment :

- un accès à un serveur comme une base de données
- d'invoquer des services distants
- de développer des applications web
- d'échanger des données entres applications

La plupart de ces fonctionnalités sont mises en oeuvre grâce à des API de haut niveau mais celles-ci utilisent généralement des API de bas niveau pour interagir avec le réseau.

Depuis son origine, Java fournit plusieurs classes et interfaces destinées à faciliter l'utilisation du réseau par programmation en reposant sur les sockets. Celles-ci peuvent être mises en oeuvre pour réaliser des échanges utilisant le protocole réseau IP avec les protocoles de transport TCP ou UDP. Les échanges se font entre deux parties : un client et un serveur.

Ce chapitre contient plusieurs sections :

♦ L'introduction aux concepts liés au réseau
♦ Les adresses internet
♦ L'accès aux ressources avec une URL
♦ L'utilisation du protocole TCP
♦ L'utilisation du protocole UDP
♦ Les exceptions liées au réseau
♦ Les interfaces de connexions au réseau

18.1. L'introduction aux concepts liés au réseau

Le modèle OSI (Open System Interconnection) propose un découpage en sept couches des différents composants qui permettent la communication sur un réseau.

Couche	Représentation physique ou logicielle
Application	Netscape ou Internet Explorer ou une application
Présentation	Windows, Mac OS ou Unix
Session	WinSock, MacTCP
Transport	TCP / UDP
Réseau	IP
Liaison	PPP, Ethernet

Physique	Les câbles et les cartes électroniques

Le protocole IP est un protocole de niveau réseau qui permet d'échanger des paquets d'octets appelés datagrammes. Ce protocole ne garantit pas l'arrivée à bon port des messages. Cette fonctionnalité peut être implémentée par la couche supérieure, comme par exemple avec TCP. Un datagramme IP est l'unité de transfert à ce niveau. Cette série d'octets contient les informations du message, un en-tête (adresses source et de destination, ...) mais aussi des informations de contrôle. Ces informations permettent aux routeurs de faire transiter les paquets sur l'internet.

La couche de transport est implémentée dans les protocoles UDP ou TCP. Elle permet la communication entre des applications sur des machines distantes.

La notion de service permet à une même machine d'assurer plusieurs communications simultanément.

Le système des sockets est le moyen de communication interprocessus développé pour l'Unix Berkeley (BSD). Il est actuellement implémenté sur tous les systèmes d'exploitation utilisant TCP/IP. Une socket est le point de communication par lequel un thread peut émettre ou recevoir des informations et ainsi elle permet la communication entre deux applications à travers le réseau.

La communication se fait sur un port particulier de la machine. Le port est une entité logique qui permet d'associer un service particulier à une connexion. Un port est identifié par un entier de 1 à 65535. Par convention les 1024 premiers sont réservés pour des services standard (80 : HTTP, 21 : FTP, 25: SMTP, ...)

Java prend en charge deux protocoles : TCP et UDP.

Les classes et interfaces utiles au développement réseau sont regroupées dans le package java.net.

18.2. Les adresses internet

Une adresse internet permet d'identifier de façon unique une machine sur un réseau. Cette adresse pour le protocole I.P. est sous la forme de quatre octets séparés chacun par un point. Chacun de ces octets appartient à une classe selon l'étendue du réseau.

Pour faciliter la compréhension humaine, un serveur particulier appelé DNS (Domaine Name Service) est capable d'associer un nom à une adresse I.P.

18.2.1. La classe InetAddress

Une adresse internet est composée de quatre octets séparés chacun par un point.

Un objet de la classe InetAddress représente une adresse Internet. Cette classe contient des méthodes pour lire une adresse, la comparer avec une autre ou la convertir en chaîne de caractères. Elle ne possède pas de constructeur : il faut utiliser certaines méthodes statiques de la classe pour obtenir une instance de cette classe.

La classe InetAdress encapsule aussi des fonctionnalités pour utiliser les adresses internet.

Méthode	Rôle
InetAddress getByName(String)	Renvoie l'adresse internet associée au nom d'hôte fourni en paramètre
InetAddress[] getAllByName(String)	Renvoie un tableau des adresses internet associées au nom d'hôte fourni en paramètre
InetAddress getLocalHost()	Renvoie l'adresse internet de la machine locale
byte[] getAddress()	Renvoie un tableau contenant les 4 octets de l'adresse internet
String getHostAddress()	Renvoie l'adresse internet sous la forme d'une chaîne de caractères

String getHostName()	Renvoie le nom du serveur

Exemple :

```
import java.net.InetAddress;
import java.net.UnknownHostException;

public class TestNet1 {

  public static void main(String[] args) {
        try {
      InetAddress adrLocale = InetAddress.getLocalHost();
      System.out.println("Adresse locale = "+adrLocale.getHostAddress());
      InetAddress adrServeur = InetAddress.getByName("java.sun.com");
      System.out.println("Adresse Sun = "+adrServeur.getHostAddress());
      InetAddress[] adrServeurs = InetAddress.getAllByName("www.microsoft.com");
      System.out.println("Adresses Microsoft : ");
      for (int i = 0; i < adrServeurs.length; i++) {
        System.out.println("     "+adrServeurs[i].getHostAddress());
      }
    } catch (UnknownHostException e) {
          e.printStackTrace();
        }

  }
}
```

Résultat :

```
Adresse locale = 192.166.23.103
Adresse Sun = 192.18.97.71
Adresses Microsoft :
     207.46.249.27
     207.46.134.155
     207.46.249.190
     207.46.134.222
     207.46.134.190
```

18.3. L'accès aux ressources avec une URL

Une URL (Uniform Resource Locator) ou localisateur de ressource uniforme est une chaîne de caractères qui désigne une ressource précise accessible par Internet ou Intranet. Une URL est donc une référence à un objet dont le format dépend du protocole utilisé pour accéder à la ressource.

Dans le cas du protocole http, l'URL est de la forme :

http://<serveur>:<port>/<chemin>?<param1>&<param2>

Elle se compose du protocole (HTTP), d'une adresse IP ou du nom de domaine du serveur de destination, avec éventuellement un port, un chemin d'accès vers un fichier ou un nom de service et éventuellement des paramètres sous la forme clé=valeur.

Dans le cas du protocole ftp, l'URL est de la forme :

ftp://<user>:<motdepasse>@<serveur>:<port>/<chemin>

Dans le cas d'un e-mail, l'URL est de la forme

mailto:<email>

Dans le cas d'un fichier local, l'URL est de la forme :

file://<serveur>/<chemin>

Elle se compose de la désignation du serveur (non utilisé dans le cas du système de fichiers local) et du chemin absolu de la ressource.

18.3.1. La classe URL

Un objet de cette classe encapsule une URL : la validité syntaxique de l'URL est assurée mais l'existence de la ressource représentée par l'URL ne l'est pas.

Exemple d'URL :

http: //www.test.fr :80 /images/image.gif

Dans l'exemple, http représente le protocole, www.test.fr représente le serveur, 80 représente le port, /images/image.gif représente la ressource.

Le nom du protocole indique au navigateur le protocole qui doit être utilisé pour accéder à la ressource. Il existe plusieurs protocoles sur internet : http, ftp, smtp ...

L'identification du serveur est l'information qui désigne une machine sur le réseau, identifiée par une adresse IP. Cette adresse s'écrit sous la forme de quatre entiers séparés par un point. Une machine peut se voir affecter un nom logique (hostname) composé d'un nom de machine (ex : www), d'un nom de sous domaine (ex : toto) et d'un nom de domaine (ex :fr). Chaque domaine possède un serveur de nom (DNS : Domain Name Server) chargé d'effectuer la correspondance entre les noms logiques et les adresses IP.

Le numéro de port désigne le service. En mode client/serveur, un client s'adresse à un programme particulier (le service) qui s'exécute sur le serveur. Le numéro de port identifie ce service. Cette information est facultative dans l'URL : par exemple si aucun numéro n'est précisé dans une url, un browser dirige sa demande vers un port standard : par défaut, le service http est associé au port 80, le service ftp au port 21, etc ...

L'identification de la ressource indique le chemin d'accès de celle-ci sur le serveur.

Le constructeur de la classe lève une exception du type MalformedURLException si la syntaxe de l'URL n'est pas correcte.

Les objets créés sont constants et ne peuvent plus être modifiés par la suite.

Exemple :

```
URL pageURL = null;
try {
  pageURL = new URL(getDocumentBase( ), "http://www.javasoft.com");
} catch (MalformedURLException mue) {}
```

La classe URL possède des getters pour obtenir les différents éléments qui la composent : getProtocole(), getHost(), getPort(), getFile().

La méthode openStream() ouvre un flux de données en entrée pour lire la ressource et renvoie un objet de type InputStream.

La méthode openConnection() ouvre une connexion vers la ressource et renvoie un objet de type URLConnexion

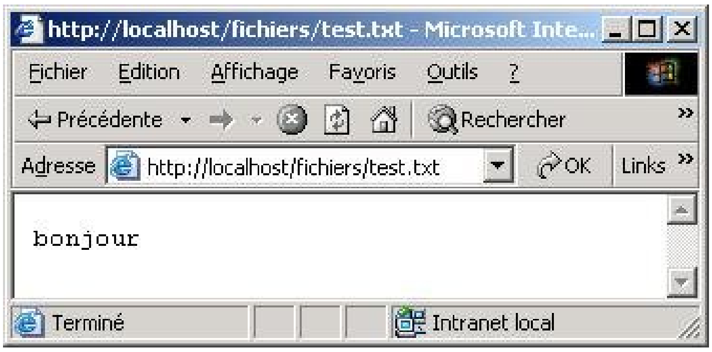
18.3.2. La classe URLConnection

Cette classe abstraite encapsule une connexion vers une ressource désignée par une URL pour obtenir un flux de données ou des informations sur la ressource.

Exemple :

```
import java.net.*;
import java.io.*;

public class TestURLConnection {

        public static void main(String[] args) {

                String donnees;

                try {

                        URL monURL = new URL("http://localhost/fichiers/test.txt");

                        URLConnection connexion = monURL.openConnection();
                        InputStream flux = connexion.getInputStream();

                        int donneesALire = connexion.getContentLength();

                        for(;donneesALire != 0; donneesALire--)
                        System.out.print((char)flux.read());

                        // Fermeture de la connexion
                        flux.close();

                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

Pour cet exemple, le fichier test.txt doit être accessible via le serveur web dans le répertoire "fichiers".


http://localhost/fichiers/test.txt - Microsoft Inte...
Fichier Edition Affichage Favoris Outils ?
Précédente
Rechercher
Adresse
http://localhost/fichiers/test.txt
OK
Links
Terminé
Intranet local

bonjour

18.3.3. La classe URLEncoder

Cette classe est une classe utilitaire qui propose la méthode statique encode() pour encoder une URL. Elle remplace notamment les espaces par un signe "+" et les caractères spéciaux par un signe "%" suivi du code du caractère.

Exemple :

```
import java.net.*;

public class TestEncodeURL {

  public static void main(String[] args) {
        String url = "http://www.test.fr/images perso/mon image.gif";
        System.out.println(URLEncoder.encode(url));
  }
}
```

Résultat :

```
http%3A%2F%2Fwww.test.fr%2Fimages+perso%2Fmon+image.gif
```

Depuis le JDK 1.4, il existe une version surchargée de la méthode encode() qui nécessite le passage d'un paramètre supplémentaire : une chaîne de caractères qui précise le format d'encodage des caractères. Cette méthode remplace l'ancienne méthode encode() qui est dépréciée. Elle peut lever une exception du type UnsupportedEncodingException.

Exemple (JDK 1.4) :

```
import java.io.UnsupportedEncodingException;
import java.net.*;

public class TestEncodeURL {

  public static void main(String[] args) {
        try {
          String url = "http://www.test.fr/images perso/mon image.gif";
          System.out.println(URLEncoder.encode(url,"UTF-8"));
          System.out.println(URLEncoder.encode(url,"UTF-16"));
    } catch (UnsupportedEncodingException e) {
      e.printStackTrace();
        }
  }
}
```

Exemple :

```
http%3A%2F%2Fwww.test.fr%2Fimages+perso%2Fmon+image.gif
http%FE%FF%00%3A%00%2F%00%2Fwww.test.fr%FE%FF%00%2Fimages+perso%FE%FF%00%2Fmon+image.gif
```

18.3.4. La classe HttpURLConnection

Cette classe qui hérite de URLConnection encapsule une connexion utilisant le protocole HTTP.

Exemple :

```
import java.net.*;
import java.io.*;

public class TestHttpURLConnection {

  public static void main(String[] args) {
    HttpURLConnection connexion = null;

   try {
    URL url = new URL("http://java.sun.com");

    System.out.println("Connexion a l'url ...");
    connexion = (HttpURLConnection) url.openConnection();

    connexion.setAllowUserInteraction(true);
    DataInputStream in = new DataInputStream(connexion.getInputStream());

      if (connexion.getResponseCode() != HttpURLConnection.HTTP_OK) {
        System.out.println(connexion.getResponseMessage());
      } else {
        while (true) {
          System.out.print((char) in.readUnsignedByte());
        }
      }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
      connexion.disconnect();
      }
    System.exit(0);
  }
}
```

18.4. L'utilisation du protocole TCP

TCP est un protocole qui permet une connexion de type point à point entre deux applications. C'est un protocole fiable qui garantit la réception dans l'ordre d'envoi des données. En contre-partie, ce protocole offre de moins bonnes performances mais c'est le prix à payer pour la fiabilité.

TCP utilise la notion de port pour permettre à plusieurs applications d'exploiter ce même protocole.

Dans une liaison entre deux ordinateurs, l'un des deux joue le rôle de serveur et l'autre celui de client.

18.4.1. La classe ServerSocket

La classe ServerSocket est utilisée côté serveur : elle attend simplement les appels du ou des clients. C'est un objet du type Socket qui prend en charge la transmission des données.

Cette classe représente la partie serveur du socket. Un objet de cette classe est associé à un port sur lequel il va attendre les connexions d'un client. Généralement, à l'arrivée d'une demande de connexion, un thread est lancé pour assurer le dialogue avec le client sans bloquer les connexions des autres clients.

La classe ServerSocket possède plusieurs constructeurs dont les principaux sont :

Constructeur	Rôle
ServerSocket()	Constructeur par défaut
ServerSocket(int)	Créer une socket sur le port fourni en paramètre
ServerSocket(int, int)	Créer une socket sur le port et avec la taille maximale de la file fournis en paramètres

Tous ces constructeurs peuvent lever une exception de type IOException.

La classe ServerSocket possède plusieurs méthodes :

Méthode	Rôle
Socket accept()	Attendre une nouvelle connexion
void close()	Fermer la socket

Si un client tente de communiquer avec le serveur, la méthode accept() renvoie une socket qui encapsule la communication avec ce client.

Le mise en oeuvre de la classe ServerSocket suit toujours la même logique :

- créer une instance de la classe ServerSocket en précisant le port en paramètre
- définir une boucle sans fin contenant les actions ci-dessous
- appelle de la méthode accept() qui renvoie une socket lors d'une nouvelle connexion
- obtenir un flux en entrée et en sortie à partir de la socket
- écrire les traitements à réaliser

Exemple (code Java 1.2) :

```
import java.net.*;
import java.io.*;

public class TestServeurTCP {
  final static int port = 9632;

  public static void main(String[] args) {
    try {
      ServerSocket socketServeur = new ServerSocket(port);
      System.out.println("Lancement du serveur");

      while (true) {
        Socket socketClient = socketServeur.accept();
        String message = "";

        System.out.println("Connexion avec : "+socketClient.getInetAddress());

        // InputStream in = socketClient.getInputStream();
        // OutputStream out = socketClient.getOutputStream();

        BufferedReader in = new BufferedReader(
          new InputStreamReader(socketClient.getInputStream()));
        PrintStream out = new PrintStream(socketClient.getOutputStream());
        message = in.readLine();
        out.println(message);

        socketClient.close();
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

L'inconvénient de ce modèle est qu'il ne peut traiter qu'une connexion à la fois. Pour pouvoir traiter plusieurs connexions simultanément, il faut créer un nouveau thread contenant les traitements à réaliser sur la socket.

Exemple :

```
import java.net.*;
import java.io.*;

public class TestServeurThreadTCP extends Thread {

  final static int port = 9632;
  private Socket socket;

  public static void main(String[] args) {
    try {
      ServerSocket socketServeur = new ServerSocket(port);
      System.out.println("Lancement du serveur");
      while (true) {
        Socket socketClient = socketServeur.accept();
        TestServeurThreadTCP t = new TestServeurThreadTCP(socketClient);
        t.start();
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public TestServeurThreadTCP(Socket socket) {
    this.socket = socket;
  }

  public void run() {
    traitements();
  }

  public void traitements() {
    try {
      String message = "";

      System.out.println("Connexion avec le client : " + socket.getInetAddress());

      BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
      PrintStream out = new PrintStream(socket.getOutputStream());
      message = in.readLine();
      out.println("Bonjour " + message);
      socket.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

18.4.2. La classe Socket

Les sockets implémentent le protocole TCP (Transmission Control Protocol). La classe contient les méthodes de création des flux d'entrée et de sortie correspondants. Les sockets constituent la base des communications par le réseau.

Comme les flux Java sont transformés en format TCP/IP, il est possible de communiquer avec l'ensemble des ordinateurs qui utilisent ce même protocole. La seule condition importante au niveau du système d'exploitation est qu'il soit capable de gérer ce protocole.

Cette classe encapsule la connexion à une machine distante par le réseau. Elle gère la connexion, l'envoi de données, la réception de données et la déconnexion.

La classe Socket possède plusieurs constructeurs dont les principaux sont :

Constructeur	Rôle
Socket()	Constructeur par défaut
Socket(String, int)	Créer une socket sur la machine dont le nom et le port sont fournis en paramètres
Socket(InetAddress, int)	Créer une socket sur la machine dont l'adresse IP et le port sont fournis en paramètres

La classe Socket possède de nombreuses méthodes dont les principales sont :

Méthode	Rôle
InetAddress getInetAddress()	Renvoie l'adresse I.P. à laquelle la socket est connectée
void close()	Fermer la socket
InputStream getInputStream()	Renvoie un flux en entrée pour recevoir les données de la socket
OutputStream getOutputStream()	Renvoie un flux en sortie pour émettre les données de la socket
int getPort()	Renvoie le port utilisé par la socket

Le mise en oeuvre de la classe Socket suit toujours la même logique :

- créer une instance de la classe Socket en précisant la machine et le port en paramètres
- obtenir un flux en entrée et en sortie
- écrire les traitements à réaliser

Exemple :

```
import java.net.*;
import java.io.*;

public class TestClientTCP {
  final static int port = 9632;

  public static void main(String[] args) {

    Socket socket;
    DataInputStream userInput;
    PrintStream theOutputStream;

    try {
      InetAddress serveur = InetAddress.getByName(args[0]);
      socket = new Socket(serveur, port);

      BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
      PrintStream out = new PrintStream(socket.getOutputStream());

      out.println(args[1]);
      System.out.println(in.readLine());

    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

18.5. L'utilisation du protocole UDP

UDP est un protocole basé sur IP qui permet une connexion de type point à point ou de type multipoint. C'est un protocole qui ne garantit pas que les données arriveront dans l'ordre d'émission. En contre-partie, ce protocole offre de bonnes performances car il est très rapide mais à réserver à des tâches peu importantes.

Pour assurer les échanges, UDP utilise la notion de port, ce qui permet à plusieurs applications d'utiliser UDP sans que les échanges interfèrent les uns avec les autres. Cette notion est similaire à la notion de port utilisée par TCP.

UDP est utilisé dans de nombreux services "standard" tels que echo (port 7), DayTime (13), etc ...

L'échange de données avec UDP se fait avec deux sockets, l'une sur le serveur, l'autre sur le client. Chaque socket est caractérisée par une adresse internet et un port.

Pour utiliser le protocole UDP, Java définit deux classes DatagramSocket et DatagramPacket.

18.5.1. La classe DatagramSocket

Cette classe crée un Socket qui utilise le protocole UDP (Unreliable Datagram Protocol) pour émettre ou recevoir des données.

Cette classe possède plusieurs constructeurs :

Constructeur	Rôle
DatagramSocket()	Créer une socket attachée à toutes les adresses IP de la machine et à un des ports libres sur la machine
DatagramSocket(int)	Créer une socket attachée à toutes les adresses IP de la machine et au port précisé en paramètre
DatagramSocket(int, InetAddress)	Créer une socket attachée à l'adresse IP et au port précisés en paramètres

Tous les constructeurs peuvent lever une exception de type SocketException : en particulier, si le port précisé est déjà utilisé lors de l'instanciation de l'objet DatagramSocket, une exception de type BindException est levée. Cette exception hérite de SocketException.

La classe DatagramSocket définit plusieurs méthodes :

Méthode	Rôle
close()	Fermer la socket et ainsi libérer le port
receive(DatagramPacket)	Recevoir des données

send(DatagramPacket)	Envoyer des données
int getPort()	Renvoyer le port associé à la socket
void setSoTimeout(int)	Préciser un timeout d'attente pour la réception d'un message.

Par défaut, un objet DatagramSocket ne possède pas de timeout lors de l'utilisation de la méthode receive(). La méthode bloque donc l'exécution de l'application jusqu'à la réception d'un packet de données. La méthode setSoTimeout() permet de préciser un timeout en millisecondes. Une fois ce délai écoulé sans réception d'un paquet de données, la méthode lève une exception du type SocketTimeoutException.

18.5.2. La classe DatagramPacket

Cette classe encapsule une adresse internet, un port et les données qui sont échangées grâce à un objet de type DatagramSocket. Elle possède plusieurs constructeurs pour encapsuler des paquets émis ou reçus.

Constructeur	Rôle
DatagramPacket(byte tampon[], int taille)	Encapsule des paquets en réception dans un tampon
DatagramPacket(byte port[], int taille, InetAddress adresse, int port)	Encapsule des paquets en émission à destination d'une machine

Cette classe propose plusieurs méthodes pour obtenir ou mettre à jour les informations sur le paquet encapsulé.

Méthode	Rôle
InetAddress getAddress ()	Renvoyer l'adresse du serveur
byte[] getData()	Renvoyer les données contenues dans le paquet
int getPort ()	Renvoyer le port
int getLength ()	Renvoyer la taille des données contenues dans le paquet
setData(byte[])	Mettre à jour les données contenues dans le paquet

Le format des données échangées est un tableau d'octets, il faut donc correctement initialiser la propriété length qui représente la taille du tableau pour un paquet émis et utiliser cette propriété pour lire les données dans un paquet reçu.

18.5.3. Un exemple de serveur et de client

L'exemple suivant est très simple : un serveur attend un nom d'utilisateur envoyé sur le port 9632. Dès qu'un message lui est envoyé, il renvoie à son expéditeur "bonjour" suivi du nom reçu du client, de son IP et du numéro du port.

Exemple : le serveur

```
import java.io.*;
import java.net.*;

public class TestServeurUDP {

  final static int port = 9632;
  final static int taille = 1024;
  static byte buffer[] = new byte[taille];

  public static void main(String argv[]) throws Exception {
    DatagramSocket socket = new DatagramSocket(port);
    String donnees = "";
    String message = "";
    int taille = 0;

    System.out.println("Lancement du serveur");
    while (true) {
      DatagramPacket paquet = new DatagramPacket(buffer, buffer.length);
      DatagramPacket envoi = null;
      socket.receive(paquet);

      System.out.println("\n"+paquet.getAddress());
      taille = paquet.getLength();
      donnees = new String(paquet.getData(),0, taille);
      System.out.println("Donnees reçues = "+donnees);

      message = "Bonjour "+donnees;
      System.out.println("Donnees envoyees = "+message);
      envoi = new DatagramPacket(message.getBytes(),
        message.length(), paquet.getAddress(), paquet.getPort());
      socket.send(envoi);
    }
  }
}
```

Exemple : le client

```
import java.io.*;
import java.net.*;

public class TestClientUDP {

  final static int port = 9632;
  final static int taille = 1024;
  static byte buffer[] = new byte[taille];

  public static void main(String argv[]) throws Exception {
    try {
      InetAddress serveur = InetAddress.getByName(argv[0]);
      int length = argv[1].length();
      byte buffer[] = argv[1].getBytes();
      DatagramSocket socket = new DatagramSocket();
      DatagramPacket donneesEmises = new DatagramPacket(buffer, length, serveur, port);
      DatagramPacket donneesRecues = new DatagramPacket(new byte[taille], taille);

      socket.setSoTimeout(30000);
      socket.send(donneesEmises);
      socket.receive(donneesRecues);

      System.out.println("Message : " + new String(donneesRecues.getData(),
        0, donneesRecues.getLength()));
      System.out.println("de : " + donneesRecues.getAddress() + ":" +
        donneesRecues.getPort());
    } catch (SocketTimeoutException ste) {
      System.out.println("Le delai pour la reponse a expire");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

L'utilisation du client nécessite de fournir en paramètres l'adresse internet du serveur et le nom de l'utilisateur.

Exécution du client :

```
C:\>java TestClientUDP www.test.fr "Michel"
java.net.UnknownHostException: www.test.fr: www.test.fr
        at java.net.InetAddress.getAllByName0(InetAddress.java:948)
        at java.net.InetAddress.getAllByName0(InetAddress.java:918)
        at java.net.InetAddress.getAllByName(InetAddress.java:912)
        at java.net.InetAddress.getByName(InetAddress.java:832)
        at TestClientUDP.main(TestClientUDP.java:12)
```

```
C:\>java TestClientUDP 192.168.25.101 "Michel"
Le delai pour la reponse a expire

C:\>java TestClientUDP 192.168.25.101 "Michel"
Message : Bonjour Michel
de : /192.168.25.101:9632
```

18.6. Les exceptions liées au réseau

Le package java.net définit plusieurs exceptions :

Exception	Rôle / Définition
BindException	Connection au port local impossible : le port est peut être déjà utilisé
ConnectException	Connection à une socket impossible : aucun serveur n'écoute sur le port précisé
MalformedURLException	L'URL n'est pas valide
NoRouteToHostException	Connection à l'hôte impossible : un firewall empêche la connexion
ProtocolException	Une erreur est survenue au niveau du protocle sous-jacent (TCP par exemple)
SocketException	Une erreur est survenue au niveau du protocle sous-jacent
SocketTimeoutException	Délai d'attente pour la réception ou l'émission des données écoulé
UnknownHostException	L'adresse IP de l'hôte n'a pas pu être trouvée
UnknownServiceException	Une erreur est survenue au niveau de la couche service : par exemple, le type MIME retourné est incorrect ou l'application tente d'écrire sur une connexion en lecture seule
URISyntaxException	La syntaxe de l'URI utilisée est invalide

18.7. Les interfaces de connexions au réseau

Le J2SE 1.4 ajoute une nouvelle classe qui encapsule les interfaces de connexions aux réseaux et qui permet d'obtenir la liste des interfaces de connexions de la machine. Cette classe est la classe NetworkInterface.

Une interface de connexions au réseau se caractérise par un nom court, une désignation et une liste d'adresses IP. La classe possède des getters sur chacun de ses éléments :

Méthode	Rôle
String getName()	Renvoie le nom court de l'interface
String getDisplayName()	Renvoie la désignation de l'interface
Enumeration getInetAddresses()	Renvoie une énumération d'objets InetAddress contenant la liste des adresses IP associées à l'interface

Cette classe possède une méthode statique getNetwotkInterfaces() qui renvoie une énumération contenant des objets de type NetworkInterface encapsulant les différentes interfaces présentes dans la machine.

Exemple :

```
import java.net.*;
import java.util.*;

public class TestNetworkInterface {

  public static void main(String[] args) {
    try {
      TestNetworkInterface.getLocalNetworkInterface();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  private static void getLocalNetworkInterface() throws SocketException,
    NoClassDefFoundError {
    Enumeration interfaces = NetworkInterface.getNetworkInterfaces();

    while (interfaces.hasMoreElements()) {
      NetworkInterface ni;
      Enumeration adresses;

      ni = (NetworkInterface) interfaces.nextElement();

      System.out.println("Network interface : ");
      System.out.println("  nom court    = " + ni.getName());
      System.out.println("  désignation  = " + ni.getDisplayName());

      adresses = ni.getInetAddresses();
      while (adresses.hasMoreElements()) {
        InetAddress ia = (InetAddress) adresses.nextElement();
        System.out.println("  adresse I.P. = " + ia);
      }
    }
  }

}
```

Résultat :

```
Network interface :
  nom court    = MS TCP Loopback interface
  désignation  = lo
  adresse I.P. = /127.0.0.1
Network interface :
  nom court    = Carte Realtek Ethernet à base RTL8029(AS)(Générique)
  désignation  = eth0
  adresse I.P. = /169.254.166.156
Network interface :
  nom court    = WAN (PPP/SLIP) Interface
  désignation  = ppp0
  adresse I.P. = /193.251.70.245<
```

19. L'internationalisation

Chapitre 19

Niveau : 🟩🟩🟨⬜⬜⬜ Intermédiaire

La localisation consiste à adapter un logiciel aux caractéristiques locales de l'environnement d'exécution telles que la langue et la monnaie. Le plus gros du travail consiste à traduire toutes les phrases et les mots. Les classes nécessaires sont incluses dans le package java.util.

Ce chapitre contient plusieurs sections :

- Les objets de type Locale
- La classe ResourceBundle
- Un guide pour réaliser la localisation : propose plusieurs solutions pour internationaliser une application.

19.1. Les objets de type Locale

Un objet de type Locale identifie une langue et un pays donné.

19.1.1. La création d'un objet Locale

Exemple (code Java 1.1) :

```
locale_US = new Locale("en","US");
locale_FR = new Locale("fr","FR");
```

Le premier paramètre est le code langue (deux caractères minuscules conformes à la norme ISO-639 : exemple "de" pour l'allemand, "en" pour l'anglais, "fr" pour le français, etc ...)

Le deuxième paramètre est le code pays (deux caractères majuscules conformes à la norme ISO-3166 : exemple : "DE" pour l'Allemagne, "FR" pour la France,"US" pour les Etats Unis, etc ...). Ce paramètre est obligatoire : si le pays n'a pas besoin d'être précisé, il faut fournir une chaîne vide.

Exemple (code Java 1.1) :

```
Locale locale = new Locale("fr", "");
```

Un troisième paramètre peut permettre de préciser d'avantage la localisation par exemple la plate-forme d'exécution (il ne respecte aucun standard car il ne sera défini que dans l'application qui l'utilise) :

Exemple (code Java 1.1) :

```
Locale locale_unix = new Locale("fr","FR", "UNIX");
Locale locale_windows = new Locale("fr","FR", "WINDOWS");
```

Ce troisième paramètre est optionnel.

La classe Locale définit des constantes pour certaines langues et pays :

Exemple (code Java 1.1) : ces deux lignes sont équivalentes

```
Locale locale_1 = Locale.JAPAN;
Locale locale_2 = new Locale("ja", "JP");
```

Lorsque l'on précise une constante représentant une langue alors le code pays n'est pas défini.

Exemple (code Java 1.1) : ces deux lignes sont équivalentes

```
Locale locale_3 = Locale.JAPANESE;
Locale locale_2 = new Locale("ja", "");
```

Il est possible de rendre la création d'un objet Locale dynamique :

Exemple (code Java 1.1) :

```
static public void main(String[] args) {
   String langue = new String(args[0]);
   String pays = new String(args[1]);
   locale = new Locale(langue, pays);
}
```

Cet objet ne sert que d'identifiant qu'il faut passer, par exemple, à des objets de type ResourceBundle qui eux possèdent le nécessaire pour réaliser la localisation. En fait, la création d'un objet Locale pour un pays donné ne signifie pas que l'on va pouvoir l'utiliser.

19.1.2. L'obtention de la liste des Locales disponibles

La méthode getAvailableLocales() permet de connaître la liste des Locales reconnues par une classe sensible à l'internationalisation

Exemple (code Java 1.1) : avec la classe DateFormat

```
import java.util.*;
import java.text.*;

public class Available {
   static public void main(String[] args) {
      Locale liste[] = DateFormat.getAvailableLocales();
      for (int i = 0; i < liste.length; i++)
      {
         System.out.println(liste[i].toString());
         // toString retourne le code langue et le code pays séparé d'un souligné
      }
   }
}
```

La méthode Locale.getDisplayName() peut être utilisée à la place de toString() pour obtenir les noms du code langue et du code pays.

19.1.3. L'utilisation d'un objet Locale

Il n'est pas obligatoire de se servir du même objet Locale avec les classes sensibles à l'internationalisation.

Cependant la plupart des applications utilisent l'objet Locale par défaut qui est initialisé par la machine virtuelle avec les paramètres de la machine hôte. La méthode Locale.getDefault() permet de connaître cet objet Locale.

19.2. La classe ResourceBundle

Il est préférable de définir un ResourceBundle pour chaque catégorie d'objet (exemple un par fenêtre) : ceci rend le code plus clair et plus facile à maintenir, évite d'avoir des ResourceBundle trop importants et réduit l'espace mémoire utilisé car chaque ressource n'est chargée que lorsque l'on en a besoin.

19.2.1. La création d'un objet ResourceBundle

Conceptuellement, chaque ResourceBundle est un ensemble de sous-classes qui partagent la même racine de nom.

Exemple :

```
TitreBouton
TitreBouton_de
TitreBouton_en_GB
TitreBouton_fr_FR_UNIX
```

Pour sélectionner le ResourceBundle approprié il faut utiliser la méthode ResourceBundle.getBundle().

Exemple (code Java 1.1) :

```
Locale locale = new Locale("fr", "FR");
ResourceBundle messages = ResourceBundle.getBundle("TitreBouton", locale);
```

Le premier argument contient le type d'objet à utiliser (la racine du nom de cet objet).

Le second argument de type Locale permet de déterminer quel fichier sera utilisé : il ajoute le code pays et le code langue séparés par des soulignés de la racine du nom.

Si la classe désignée par l'objet Locale n'existe pas, alors la méthode getBundle() recherche celle qui se rapproche le plus. L'ordre de recherche sera le suivant :

Exemple :

```
TitreBouton_fr_CA_UNIX
TitreBouton_fr_FR
TitreBouton_fr
TitreBouton_en_US
TitreBouton_en
TitreBouton
```

Si aucune n'est trouvée alors getBundle lève une exception de type MissingResourceException.

19.2.2. Les sous-classes de ResourceBundle

La classe abstraite ResourceBundle possède deux sous-classes : PropertyResourceBundle et ListResourceBundle.

La classe ResourceBundle est une classe flexible : le passage d'un PropertyResourceBundle à ListResourceBundle se fait sans impact sur le code. La méthode getBundle() recherche le ResourceBundle désiré qu'il soit dans un fichier .class ou propriétés.

19.2.2.1. L'utilisation de PropertyResourceBundle

Un PropertyResourceBundle est rattaché à un fichier propriétés. Ces fichiers propriétés ne font pas partie du code source java. Ils ne peuvent contenir que des chaines de caractères. Pour stocker d'autres objets, il faut utiliser des objets ListResourceBundle.

La création d'un fichier propriétés est simple : c'est un fichier texte qui contient des paires clé-valeur. La clé et la valeur sont séparées par un signe =. Chaque paire doit être sur une ligne séparée.

Exemple (code Java 1.1) :

```
texte_suivant = suivant
texte_precedent = precedent
```

Le nom du fichier propriétés par défaut se compose de la racine du nom suivi de l'extension .properties.

Exemple : TitreBouton.properties.

Dans une autre langue, anglais par exemple, le fichier s'appellerait : TitreBouton_en.properties

Exemple (code Java 1.1) :

```
texte_suivant = next
texte_precedent = previous
```

Les clés sont les mêmes, seule la traduction change.

Le nom de fichier TitreBouton_fr_FR.properties contient la racine (Titrebouton), le code langue (fr) et le code pays (FR).

19.2.2.2. L'utilisation de ListResourceBundle

La classe ListResourceBundle gère les ressources sous forme d'une liste encapsulée dans un objet. Chaque ListResourceBundle est donc rattaché à un fichier .class. On peut y stocker n'importe quel objet spécifique à la localisation.

Les objets ListResourceBundle contiennent des paires clé-valeur. La clé doit être une chaîne qui caractérise l'objet. La valeur est un objet de n'importe quelle classe.

Exemple (code Java 1.1) :

```
class TitreBouton_fr extends ListResourceBundle {
   public Object[][] getContents() {
      return contents;
   }
   static final Object[][] contents = {
      {"texte_suivant", "Suivant"},
      {"texte_precedent", "Precedent"},
   };
}
```

19.2.3. L'obtention d'un texte d'un objet ResourceBundle

La méthode getString() retourne la valeur de la clé précisée en paramètre.

Exemple (code Java 1.1) :

```
String message_1 = messages.getString("texte_suivant");
String message_2 = TitreBouton.getString("texte_suivant");
```

19.3. Un guide pour réaliser la localisation

19.3.1. L'utilisation d'un ResourceBundle avec un fichier propriétés

Il faut toujours créer le fichier propriété par défaut. Le nom de ce fichier commence avec le nom de base du ResourceBundle et se termine avec le suffixe .properties

Exemple (code Java 1.1) :

```
#Exemple de fichier propriété par défaut (TitreBouton.properties)
texte1 = suivant
texte2 = precedent
texte3 = quitter
```

Les lignes de commentaires commencent par un #. Les autres lignes contiennent les paires clé-valeur. Une fois le fichier défini, il ne faut plus modifier la valeur de la clé qui pourrait être appelée dans un programme.

Pour ajouter le support d'autres langues, il faut créer des fichiers propriétés supplémentaires qui contiendront les traductions. Les fichiers ne diffèreront que par les valeurs associées aux clés, ces dernières restant identiques entre les fichiers.

Exemple (code Java 1.1) :

```
#Exemple de fichier propriété en anglais (TitreBouton_en.properties)
texte1 = next
texte2 = previous
texte3 = quit
```

Lors de la programmation, il faut créer un objet Locale. Il est possible de créer un tableau qui contient la liste des Locale disponibles en fonction des fichiers propriétés créés.

Exemple (code Java 1.1) :

```
Locale[] locales = { Locale.GERMAN, Locale.ENGLISH };
```

Dans cet exemple, l'objet Locale.ENGLISH correspond au fichier TitreBouton_en.properties. L'objet Locale.GERMAN ne possédant pas de fichier propriétés défini, le fichier par défaut sera utilisé.

Il faut créer l'objet ResourceBundle en invoquant la méthode getBundle de l'objet Locale.

Exemple (code Java 1.1) :

```
ResourceBundle titres = ResourceBundle.getBundle("TitreBouton", locales[1]);
```

La méthode getBundle() recherche en premier une classe qui correspond au nom de base, si elle n'existe pas alors elle recherche un fichier de propriétés. Lorsque le fichier est trouvé, elle retourne un objet PropertyResourceBundle qui contient les paires clé-valeur du fichier

Pour retrouver la traduction d'un texte, il faut utiliser la méthode getString() d'un objet ResourceBundle

Exemple (code Java 1.1) :

```
String valeur = titres.getString(key);
```

Lors du débogage, il peut être utile d'obtenir la liste des paires d'un objet ResourceBundle. La méthode getKeys() retourne un objet Enumeration qui contient toutes les clés de l'objet.

Exemple (code Java 1.1) :

```
ResourceBundle titres =ResourceBundle.getBundle("TitreBouton", locales[1]);
Enumeration cles = titres.getKeys();
while (cles.hasMoreElements()) {
   String cle = (String)cles.nextElement();
   String valeur = titres.getString(cle);
   System.out.println("cle = " + cle +
      ", " +   "valeur = " + valeur);
}
```

19.3.2. Des exemples de classes utilisant PropertiesResourceBundle

Exemple (code Java 1.1) : Sources de la classe I18nProperties

```
/*
Test d'utilisation de la classe PropertiesResourceBundle
pour internationaliser une application
13/02/99
*/

import java.util.*;

/**
 * Description de la classe I18nProperties
 *
 * @version       0.10 13 fevrier 1999
 * @author        Jean Michel DOUDOUX
 */
public class I18nProperties {
  /**
   * Constructeur de la classe
   */
  public I18nProperties() {

    String texte;
    Locale locale;
    ResourceBundle res;

    System.out.println("Locale par defaut : ");
    locale = Locale.getDefault();
    res = ResourceBundle.getBundle("I18nPropertiesRessources", locale);
    texte = (String)res.getObject("texte_suivant");
    System.out.println("texte_suivant = "+texte);
    texte = (String)res.getObject("texte_precedent");
    System.out.println("texte_precedent = "+texte);

    System.out.println("\nLocale anglaise : ");
    locale = new Locale("en","");
    res = ResourceBundle.getBundle("I18nPropertiesRessources", locale);
    texte = (String)res.getObject("texte_suivant");
    System.out.println("texte_suivant = "+texte);
    texte = (String)res.getObject("texte_precedent");
    System.out.println("texte_precedent = "+texte);

    System.out.println("\nLocale allemande : "+
       "non définie donc utilisation locale par defaut ");
    locale = Locale.GERMAN;
    res = ResourceBundle.getBundle("I18nPropertiesRessources", locale);
    texte = (String)res.getObject("texte_suivant");
    System.out.println("texte_suivant = "+texte);
    texte = (String)res.getObject("texte_precedent");
    System.out.println("texte_precedent = "+texte);
  }

 /**
   * Pour tester la classe
   *
   * @param      args[]         arguments passes au programme
   */
  public static void main(String[] args) {
    I18nProperties i18nProperties = new I18nProperties();
  }

}
```

Exemple (code Java 1.1) : Contenu du fichier I18nPropertiesRessources.properties

```
texte_suivant=suivant
texte_precedent=Precedent
```

Exemple (code Java 1.1) : Contenu du fichier I18nPropertiesRessources_en.properties

```
texte_suivant=next
texte_precedent=previous
```

Exemple (code Java 1.1) : Contenu du fichier I18nPropertiesRessources_en_US.properties

```
texte_suivant=next
texte_precedent=previous
```

19.3.3. L'utilisation de la classe ListResourceBundle

Il faut créer autant de sous-classes de ListResourceBundle que de langues désirées : ceci va générer un fichier .class pour chacune des langues .

Exemple (code Java 1.1) :

```
TitreBouton_fr_FR.class
TitreBouton_en_EN.class
```

Le nom de la classe doit contenir le nom de base plus le code langue et le code pays séparés par des soulignés. A l'intérieur de la classe, un tableau à deux dimensions est initialisé avec les paires clé-valeur. Les clés sont des chaines qui doivent être identiques dans toutes les classes des différentes langues. Les valeurs peuvent être des objets de n'importe quel type.

Exemple (code Java 1.1) :

```
import java.util.*;

public class TitreBouton_fr_FR extends ListResourceBundle {
   public Object[][] getContents() {
      return contents;
   }

   private Object[][] contents = {
      { "texte_suivant", new String(" suivant ")},
      { "Numero", new Integer(4) }
   };
}
```

Il faut définir un objet de type Locale

Il faut créer un objet de type ResourceBundle en appelant la méthode getBundle() de la classe Locale

Exemple (code Java 1.1) :

```
ResourceBundle titres=ResourceBundle.getBundle("TitreBouton", locale);
```

La méthode getBundle() recherche une classe qui commence par TitreBouton et qui est suivie par le code langue et le code pays précisés dans l'objet Locale passé en paramètre

La méthode getObject() permet d'obtenir la valeur de la clé passée en paramètres. Dans ce cas une conversion est nécessaire.

Exemple (code Java 1.1) :

```
Integer valeur = (Integer)titres.getObject("Numero");
```

19.3.4. Des exemples de classes utilisant ListResourceBundle

Exemple (code Java 1.1) : Sources de la classe I18nList

```
/*
Test d'utilisation de la classe ListResourceBundle
pour internationaliser une application
13/02/99
*/

import java.util.*;

/**
 * Description de la classe I18nList
 *
 * @version       0.10 13 fevrier 1999
 * @author        Jean Michel DOUDOUX
 */
public class I18nList {
  /**
   * Constructeur de la classe
   */
  public I18nList() {

    String texte;
    Locale locale;
    ResourceBundle res;

    System.out.println("Locale par defaut : ");
    locale = Locale.getDefault();
    res = ResourceBundle.getBundle("I18nListRessources", locale);
    texte = (String)res.getObject("texte_suivant");
    System.out.println("texte_suivant = "+texte);
    texte = (String)res.getObject("texte_precedent");
    System.out.println("texte_precedent = "+texte);

    System.out.println("\nLocale anglaise : ");
    locale = new Locale("en","");
    res = ResourceBundle.getBundle("I18nListRessources", locale);
    texte = (String)res.getObject("texte_suivant");
    System.out.println("texte_suivant = "+texte);
    texte = (String)res.getObject("texte_precedent");
    System.out.println("texte_precedent = "+texte);

    System.out.println("\nLocale allemande : "+
       "non définie donc utilisation locale par defaut ");
    locale = Locale.GERMAN;
    res = ResourceBundle.getBundle("I18nListRessources", locale);
    texte = (String)res.getObject("texte_suivant");
    System.out.println("texte_suivant = "+texte);
    texte = (String)res.getObject("texte_precedent");
    System.out.println("texte_precedent = "+texte);
  }

 /**
   * Pour tester la classe
   *
   * @param      args[]         arguments passes au programme
   */
  public static void main(String[] args) {
    I18nList i18nList = new I18nList();
  }

}
```

Exemple (code Java 1.1) : Sources de la classe I18nListRessources

```
/*
test d'utilisation de la classe ListResourceBundle pour
internationaliser une application
13/02/99
*/

import java.util.*;

/**
 * Ressource contenant les traductions françaises
 * langue par defaut de l'application
 *
 * @version       0.10 13 fevrier 1999
 * @author        Jean Michel DOUDOUX
 *
 */

public class I18nListRessources extends ListResourceBundle {
  public Object[][] getContents() {
    return contents;
  }

   //tableau des mots clés et des valeurs

   static final Object[][] contents = {
   {"texte_suivant", "Suivant"},
   {"texte_precedent", "Precedent"},
  };
}
```

Exemple (code Java 1.1) : Sources de la classe I18nListRessources_en

```
/*
test d'utilisation de la classe ListResourceBundle pour
internationaliser une application
13/02/99
*/

import java.util.*;

/**
 * Ressource contenant les traductions anglaises
 *
 * @version       0.10 13 fevrier 1999
 * @author        Jean Michel DOUDOUX
 *
 */
public class I18nListRessources_en extends ListResourceBundle {
  public Object[][] getContents() {
    return contents;
  }

  //tableau des mots clés et des valeurs

  static final Object[][] contents = {
    {"texte_suivant", "Next"},
    {"texte_precedent", "Previous"},
  };
}
```

Exemple (code Java 1.1) : Sources de la classe I18nListRessources_en_US

```
/*
test d'utilisation de la classe ListResourceBundle pour
internationaliser une application
13/02/99
*/
import java.util.*;

/**
 * Ressource contenant les traductions américaines
 *
 * @version 0.10 13 fevrier 1999
 * @author Jean Michel DOUDOUX
 *
 */
public class I18nListRessources_en_US extends ListResourceBundle {
  public Object[][] getContents() {
    return contents;
  }

  //tableau des mots clés et des valeurs

  static final Object[][] contents = {
    {"texte_suivant", "Next"},
    {"texte_precedent", "Previous"},
  };
}
```

19.3.5. La création de sa propre classe fille de ResourceBundle

La troisième solution consiste à créer sa propre sous-classe de ResourceBundle et à surcharger la méthode handleGetObject().

Exemple (code Java 1.1) :

```
abstract class MesRessources extends ResourceBundle {

   public Object handleGetObject(String cle)  {
      if(cle.equals(" texte_suivant "))
         return " Suivant " ;
      if(cle.equals(" texte_precedent "))
         return "Precedent " ;
         return null ;
   }
}
```

Attention : la classe ResourceBundle contient deux méthodes abstraites : handleGetObjects() et getKeys(). Si l'une des deux n'est pas définie alors il faut définir la sous-classe avec le mot clé abstract.

Il faut créer autant de sous-classes que de Locale désirées : il suffit simplement d'ajouter dans le nom de la classe le code langue et le code pays avec éventuellement le code variant.

Exemple (code Java 1.1) : Sources de la classe I18nResource

```
/*
Test d'utilisation d'un sous classement
de la classe ResourceBundle pour
internationaliser une application
13/02/99
*/
import java.util.*;
/**
 * Description de la classe I18nResource
 *
 * @version        0.10 13 fevrier 1999
 * @author         Jean Michel DOUDOUX
 */
public class I18nResource {
  /**
   * Constructeur de la classe
   */
  public I18nResource() {

    String texte;
    Locale locale;
    ResourceBundle res;


    System.out.println("Locale par defaut : ");
    res = ResourceBundle.getBundle("I18nResourceBundle");
    texte = (String)res.getObject("texte_suivant");
    System.out.println("texte_suivant = "+texte);
    texte = (String)res.getObject("texte_precedent");
    System.out.println("texte_precedent = "+texte);


    System.out.println("\nLocale anglaise : ");
    locale = new Locale("en","");
    res = ResourceBundle.getBundle("I18nResourceBundle", locale);
    texte = (String)res.getObject("texte_suivant");
    System.out.println("texte_suivant = "+texte);
    texte = (String)res.getObject("texte_precedent");
    System.out.println("texte_precedent = "+texte);

    System.out.println("\nLocale allemande : "+
       "non définie donc utilisation locale par defaut ");
    locale = Locale.GERMAN;
    res = ResourceBundle.getBundle("I18nResourceBundle", locale);
    texte = (String)res.getObject("texte_suivant");
    System.out.println("texte_suivant = "+texte);
    texte = (String)res.getObject("texte_precedent");
    System.out.println("texte_precedent = "+texte);
  }

 /**
   * Pour tester la classe
   *
   * @param      args[]         arguments passés au programme
   */
  public static void main(String[] args) {
    I18nResource i18nResource = new I18nResource();
  }

}
```

Exemple (code Java 1.1) : Sources de la classe I18nResourceBundle

```
/*
Test d'utilisation de la derivation de la classe ResourceBundle pour
internationaliser une application
13/02/99
*/

import java.util.*;

/**
 * Description de la classe I18nResourceBundle
 * C'est la classe contenant la locale par défaut
 * Contient les traductions de la locale française (langue par defaut)
 * Elle hérite de ResourceBundle
 *
 * @version        0.10 13 février 1999
 * @author         Jean Michel DOUDOUX
 */
public class I18nResourceBundle extends ResourceBundle {
   protected Vector table;
   public I18nResourceBundle() {
      super();
      table = new Vector();
      table.addElement("texte_suivant");
      table.addElement("texte_precedent");
   }

   public Object handleGetObject(String cle)  {
      if(cle.equals(table.elementAt(0))) return "Suivant" ;
      if(cle.equals(table.elementAt(1))) return "Precedent" ;
      return null ;
   }

   public Enumeration getKeys()  {
      return table.elements();
   }

}
```

Exemple (code Java 1.1) : Sources de la classe I18nResourceBundle_en

```
/*
Test d'utilisation de la derivation de la classe ResourceBundle pour
internationaliser une application
13/02/99
*/

import java.util.*;

/**
 * Description de la classe I18nResourceBundle_en
 * Contient les traductions de la locale anglaise
 * Elle hérite de la classe contenant la locale par défaut
 *
 * @version       0.10 13 février 1999
 * @author        Jean Michel DOUDOUX
 */
public class I18nResourceBundle_en extends I18nResourceBundle {
   public Object handleGetObject(String cle)  {
      if(cle.equals(table.elementAt(0))) return "Next" ;
      if(cle.equals(table.elementAt(1))) return "Previous" ;
      return null ;
   }
}
```

Exemple (code Java 1.1) : Sources de la classe I18nResourceBundle_fr_FR

```
/*
Test d'utilisation de la dérivation de la classe ResourceBundle pour
internationaliser une application
13/02/99
*/

import java.util.*;

/**
 * Description de la classe I18nResourceBundle_fr_FR
 * Contient les traductions de la locale française
 * Elle hérite de la classe contenant la locale par défaut
 *
 * @version       0.10 13 février 1999
 * @author        Jean Michel DOUDOUX
 */
public class I18nResourceBundle_fr_FR extends I18nResourceBundle {

   /**
    *
    * Retourne toujours null car la locale francaise correspond
    * a la locale par defaut
    *
    */
   public Object handleGetObject(String cle)  {
      return null ;
    }
}
```

20. Les composants Java beans

Chapitre 20

Niveau : 🟩🟩🟨⬜⬜⬜ Intermédiaire

Les JavaBeans sont des composants réutilisables introduits par le JDK 1.1. De nombreuses fonctionnalités ont ensuite été ajoutées pour développer des caractéristiques de ces composants. Les JavaBeans sont couramment appelés beans tout simplement.

Les beans sont prévus pour pouvoir interagir avec d'autres beans au point de pouvoir développer une application simplement en assemblant des beans avec un outil graphique dédié. Sun fournit gratuitement un tel outil : le B.D.K. (Bean Development Kit).

Ce chapitre contient plusieurs sections :

- La présentation des Java beans
- Les propriétés.
- Les méthodes
- Les événements
- L'introspection
- Paramétrage du bean (Customization)
- La persistance
- La diffusion sous forme de jar

20.1. La présentation des Java beans

Des composants réutilisables sont des objets autonomes qui doivent pouvoir être facilement assemblés entres eux pour créer un programme.

Microsoft propose la technologie ActiveX pour définir des composants mais celle-ci est spécifiquement destinée aux plates-formes Windows.

Les Java beans proposés par Sun reposent bien sûr sur Java et de fait en possèdent toutes les caractéristiques : indépendance de la plate-forme, taille réduite du composant, ...

La technologie JavaBeans propose de simplifier et faciliter la création et l'utilisation de composants.

Les JavaBeans possèdent plusieurs caractéristiques :

- la persistance : elle permet grâce au mécanisme de sérialisation de sauvegarder l'état d'un bean pour le restaurer. Ainsi si on assemble plusieurs beans pour former une application, on peut la sauvegarder.
- la communication, grâce à des événements, qui utilise le modèle des écouteurs introduit par Java 1.1
- l'introspection : ce mécanisme permet de découvrir de façon dynamique l'ensemble des éléments qui composent le bean (attributs, méthodes et événements) sans avoir le code source.
- la possibilité de paramétrer le composant : les données du paramétrage sont conservées dans des propriétés.

Ainsi, les beans sont des classes Java qui doivent respecter un certain nombre de règles :

- ils doivent posséder un constructeur sans paramètre. Celui-ci devra initialiser l'état du bean avec des valeurs par défauts.
- ils peuvent définir des propriétés : celles-ci sont identifiées par des méthodes dont le nom et la signature sont normalisés
- ils devraient implémenter l'interface serialisable : ceci est obligatoire pour les beans qui possèdent une partie graphique pour permettre la sauvegarde de leur état
- ils définissent des méthodes utilisables par les composants extérieurs : elles doivent être public et prévoir une gestion des accès concurrents
- ils peuvent émettre des événements en gérant une liste d'écouteurs qui s'y abonnent grâce à des méthodes dont les noms sont normalisés

Le type de composants le plus adapté est le composant visuel. D'ailleurs, les composants des classes A.W.T. et Swing pour la création d'interfaces graphiques sont tous des beans. Mais les beans peuvent aussi être des composants non visuels pour prendre en charge les traitements.

20.2. Les propriétés.

Les propriétés contiennent des données qui gèrent l'état du composant : elles peuvent être de type primitif ou être un objet.

Il existe quatre types de propriétés :

- les propriétés simples
- les propriétés indexées (indexed properties)
- les propriétés liées (bound properties)
- les propriétés liées avec contraintes (Constrained properties)

20.2.1. Les propriétés simples

Les propriétés sont des variables d'instance du bean qui possèdent des méthodes particulières pour lire et modifier leur valeur. La normalisation de ces méthodes permet à des outils de déterminer de façon dynamique quelles sont les propriétés du bean. L'accès à ces propriétés doit se faire grâce à ces méthodes. Ainsi la variable qui stocke la valeur de la propriété ne doit pas être déclarée public mais les méthodes d'accès à cette variable doivent bien sûr l'être.

Le nom de la méthode de lecture d'une propriété doit obligatoirement commencer par "get" suivi par le nom de la propriété dont la première lettre doit être une majuscule. Une telle méthode est souvent appelée "getter" ou "accesseur" de la propriété. La valeur retournée par cette méthode doit être du type de la propriété.

Exemple (code Java 1.1) :

```
private int longueur;

public int getLongueur () {
  return longueur;
}
```

Pour les propriétés booléennes, une autre convention peut être utilisée : la méthode peut commencer par «is» au lieu de « get ». Dans ce cas, la valeur de retour est obligatoirement de type boolean.

Le nom de la méthode permettant la modification d'une propriété doit obligatoirement commencer par « set » suivi par le nom de la propriété dont la première lettre doit être une majuscule. Une telle méthode est souvent appelée « setter ». Elle ne retourne aucune valeur et doit avoir en paramètre une variable du type de la propriété qui contiendra sa nouvelle valeur. Elle devra assurer la mise à jour de la valeur de la propriété en effectuant éventuellement des contrôles et/ou des traitements (par exemple le rafraîchissement pour un bean visuel dont la propriété affecte l'affichage).

Exemple (code Java 1.1) :

```
private int longueur ;

public void setLongueur (int longueur) {
  this.longueur = longueur;
}
```

Une propriété peut n'avoir qu'un getter et pas de setter : dans ce cas, la propriété n'est utilisable qu'en lecture seule.

Le nom de la variable d'instance qui contient la valeur de la propriété n'est pas obligatoirement le même que le nom de la propriété

Il est préférable d'assurer une gestion des accès concurrents dans ses méthodes de lecture et de mise à jour des propriétés par exemple en déclarant ces méthodes synchronized.

Les méthodes du beans peuvent directement manipuler en lecture et écriture la variable d'instance qui stocke la valeur de la propriété, mais il est préférable d'utiliser le getter et le setter.

20.2.2. Les propriétés indexées (indexed properties)

Ce sont des propriétés qui possèdent plusieurs valeurs stockées dans un tableau.

Pour ces propriétés, il faut aussi définir des méthodes « get » et « set » dont il convient d'ajouter un paramètre de type int représentant l'index de l'élément du tableau.

Exemple (code Java 1.1) :

```
private float[] notes = new float[5];

public float getNotes (int i ) {
  return notes[i];
}

public void setNotes (int i, float notes) {
  this.notes[i] = notes;
}
```

Il est aussi possible de définir des méthodes « get » et « set » permettant de lire et de mettre à jour tout le tableau.

Exemple (code Java 1.1) :

```
private float[] notes = new float[5];

public float[] getNotes () {
  return notes;
}

public void setNotes (float[] notes) {
  this.notes = notes;
}
```

20.2.3. Les propriétés liées (Bound properties)

Il est possible d'informer d'autres composants du changement de la valeur d'une propriété d'un bean. Les JavaBeans peuvent mettre en place un mécanisme qui permet pour une propriété d'enregistrer des composants qui seront informés du changement de la valeur de la propriété.

Ce mécanisme peut être mis en place grâce à un objet de la classe PropertyChangeSupport qui permet de simplifier la gestion de la liste des écouteurs et de les informer des changements de valeur d'une propriété. Cette classe définit les méthodes addPropertyChangeListener() pour enregistrer un composant désirant être informé du changement de la valeur de la propriété et removePropertyChangeListener() pour supprimer un composant de la liste.

La méthode firePropertyChange() permet d'informer tous les composants enregistrés du changement de la valeur de la propriété.

Le plus simple est que le bean hérite de la classe PropertyChangeSupport si possible car les méthodes addPropertyChangeListener() et removePropertyChangeListener() seront directement héritées.

Si ce n'est pas possible, il est obligatoire de définir les méthodes addPropertyChangeListener() et removePropertyChangeListener() dans le bean qui appelleront les méthodes correspondantes de l'objet PropertyChangeSupport.

Exemple (code Java 1.1) :

```
import java.io.Serializable;
import java.beans.*;
public class MonBean03 implements Serializable {
  protected int valeur;

  PropertyChangeSupport changeSupport;

  public MonBean03(){
    valeur = 0;

    changeSupport = new PropertyChangeSupport(this);
  }

  public synchronized void setValeur(int val) {
    int oldValeur = valeur;
    valeur = val;

    changeSupport.firePropertyChange("valeur",oldValeur,valeur);
  }
  public synchronized int getValeur() {
    return valeur;
  }
  public synchronized void addPropertyChangeListener(PropertyChangeListener listener) {
    changeSupport.addPropertyChangeListener(listener);
  }

  public synchronized void removePropertyChangeListener(PropertyChangeListener listener) {
    changeSupport.removePropertyChangeListener(listener);
  }

}
```

Les composants qui désirent être enregistrés doivent obligatoirement implémenter l'interface PropertyChangeListener et définir la méthode propertyChange() déclarée par cette interface.

La méthode propertyChange() reçoit en paramètre un objet de type PropertyChangeEvent qui représente l'événement. Les méthodes propertyChange() de tous les objets enregistrés sont appelées. La méthode propertyChange() reçoit en paramètre un objet de type PropertyChangeEvent qui contient plusieurs informations :

- l'objet source : le bean dont la valeur d'une propriété a changé
- le nom de la propriété sous forme d'une chaîne de caractères
- l'ancienne valeur sous forme d'un objet de type Object
- la nouvelle valeur sous forme d'un objet de type Object

Pour les traitements, il est souvent nécessaire d'utiliser un cast pour transmettre ou utiliser les objets qui représentent l'ancienne et la nouvelle valeur.

Méthode	Rôle
public Object getSource()	retourne l'objet source
public Object getNewValue()	retourne la nouvelle valeur de la propriété
public Object getOldValue()	retourne l'ancienne valeur de la propriété

public String getPropertyName	retourne le nom de la propriété modifiée

Exemple (code Java 1.1) : un programme qui créé le bean et lui associe un écouteur

```
import java.beans.*;
import java.util.*;
public class TestMonBean03 {
  public static void main(String[] args) {
    new TestMonBean03();
  }

  public TestMonBean03() {
    MonBean03 monBean = new MonBean03();

    monBean.addPropertyChangeListener( new PropertyChangeListener() {
      public void propertyChange(PropertyChangeEvent event) {
        System.out.println("propertyChange : valeur = "+ event.getNewValue());
      }
    } );

    System.out.println("valeur = " + monBean.getValeur());
    monBean.setValeur(10);
    System.out.println("valeur = " + monBean.getValeur());

  }
}
```

Résultat :

```
C:\tutorial\sources exemples>java TestMonBean03
valeur = 0
propertyChange : valeur = 10
valeur = 10
```

Pour supprimer un écouteur de la liste du bean, il suffit d'appeler la méthode removePropertyChangeListener() en lui passant en paramètre une référence sur l'écouteur.

20.2.4. Les propriétés liées avec contraintes (Constrained properties)

Ces propriétés permettent à un ou plusieurs composants de mettre un veto sur la modification de la valeur de la propriété.

Comme pour les propriétés liées, le bean doit gérer une liste de composants « écouteurs » qui souhaitent être informés d'un changement possible de la valeur de la propriété. Si un composant désire s'opposer à ce changement de valeur, il lève une exception pour en informer le bean.

Les écouteurs doivent implémenter l'interface VetoableChangeListener qui définit la méthode vetoableChange().

Avant le changement de la valeur, le bean appelle cette méthode vetoableChange() de tous les écouteurs enregistrés. Elle possède en paramètre un objet de type PropertyChangeEvent qui contient : le bean, le nom de la propriété, l'ancienne et la nouvelle valeur.

Si un écouteur veut s'opposer à la mise à jour de la valeur, il lève une exception de type java.beans.PropertyVetoException. Dans ce cas, le bean ne change pas la valeur de la propriété : ces traitements sont à la charge du programmeur avec notamment la gestion de la capture et du traitement de l'exception dans un bloc try/catch.

La classe VetoableChangeSupport permet de simplifier la gestion de la liste des écouteurs et de les informer du futur changement de valeur d'une propriété. Son utilisation est similaire à celle de la classe PropertyChangeSupport.

Pour ces propriétés, pour que les traitements soient complets il faut implémenter le code pour gérer et traiter les écouteurs qui souhaitent connaître les changements de valeur effectifs de la propriété (voir les propriétés liées).

Exemple (code Java 1.1) :

```
import java.io.Serializable;
import java.beans.*;

public class MonBean04 implements Serializable {
  protected int oldValeur;
  protected int valeur;

  PropertyChangeSupport changeSupport;
  VetoableChangeSupport vetoableSupport;

  public MonBean04(){
    valeur = 0;
    oldValeur = 0;

    changeSupport = new PropertyChangeSupport(this);
    vetoableSupport = new VetoableChangeSupport(this);
  }


  public synchronized void setValeur(int val) {
    oldValeur = valeur;
    valeur = val;

    try {
      vetoableSupport.fireVetoableChange("valeur",new Integer(oldValeur),
              new Integer(valeur));
    } catch(PropertyVetoException e) {
      System.out.println("MonBean, un veto est emis :  "+e.getMessage());
      valeur = oldValeur;
    }
    if ( valeur != oldValeur ) {
      changeSupport.firePropertyChange("valeur",oldValeur,valeur);
    }

  }
  public synchronized int getValeur() {
    return valeur;
  }
  public synchronized void addPropertyChangeListener(PropertyChangeListener listener) {
    changeSupport.addPropertyChangeListener(listener);
  }

  public synchronized void removePropertyChangeListener(PropertyChangeListener listener) {
    changeSupport.removePropertyChangeListener(listener);
  }
  public synchronized void addVetoableChangeListener(VetoableChangeListener listener) {
    vetoableSupport.addVetoableChangeListener(listener);
  }
  public synchronized void removeVetoableChangeListener(VetoableChangeListener listener) {
    vetoableSupport.removeVetoableChangeListener(listener);
  }

}
```

Exemple (code Java 1.1) : un programme qui teste le bean. Il émet un veto si la nouvelle valeur de la propriété est supérieure à 100.

```
import java.beans.*;
import java.util.*;
public class TestMonBean04 {
  public static void main(String[] args) {
    new TestMonBean04();
  }

  public TestMonBean04() {
    MonBean04 monBean = new MonBean04();

    monBean.addPropertyChangeListener( new PropertyChangeListener() {
      public void propertyChange(PropertyChangeEvent event) {
        System.out.println("propertyChange : valeur = "+ event.getNewValue());
      }
    } );
    monBean.addVetoableChangeListener( new VetoableChangeListener() {

      public void vetoableChange(PropertyChangeEvent event) throws PropertyVetoException {
        System.out.println("vetoableChange : valeur = " + event.getNewValue());
        if( ((Integer)event.getNewValue()).intValue() > 100 )
          throw new PropertyVetoException("valeur superieure a 100",event);
      }
    } );

    System.out.println("valeur = " + monBean.getValeur());
    monBean.setValeur(10);
    System.out.println("valeur = " + monBean.getValeur());
    monBean.setValeur(200);
    System.out.println("valeur = " + monBean.getValeur());

  }
}
```

Exemple (code Java 1.1) :

```
C:\tutorial\sources exemples>java TestMonBean04
valeur = 0
vetoableChange : valeur = 10
propertyChange : valeur = 10
valeur = 10
vetoableChange : valeur = 200
vetoableChange : valeur = 10
MonBean, un veto est emis :  valeur superieure a 100
valeur = 10
```

20.3. Les méthodes

Toutes les méthodes publiques sont visibles de l'extérieur et peuvent donc être appelées.

20.4. Les événements

Pour dialoguer, Les beans utilisent les événements définis dans le modèle par délégation introduit par le J.D.K. 1.1. Par respect de ce modèle, le bean est la source et les autres composants qui souhaitent être informés sont nommés « Listeners » ou « écouteurs » et doivent s'enregistrer auprès du bean qui maintient la liste des composants enregistrés.

Il est nécessaire de définir les méthodes qui vont permettre de gérer la liste des écouteurs désirant recevoir l'événement. Il faut définir deux méthodes :

- public void addXXXListener(XXXListener li) pour enregistrer l'écouteur li
- public void removeXXXListener (XXXListener li) pour enlever l'écouteur li de la liste

L'objet de type XXXListener doit obligatoirement implémenter l'interface java.util.EventListener et son nom doit terminer par « Listener ».

Les événements peuvent être mono ou multi écouteurs.

Pour les événements mono écouteurs, la méthode addXXXListener() doit indiquer dans sa signature qu'elle est susceptible de lever l'exception java.util.TooManyListenersException si un écouteur tente de s'enregistrer et qu'il y en a déjà un présent.

20.5. L'introspection

L'introspection est un mécanisme qui permet de déterminer de façon dynamique les caractéristiques d'une classe et donc d'un bean. Les caractéristiques les plus importantes sont les propriétés, les méthodes et les événements. Le principe de l'introspection permet à Sun d'éviter de rajouter des éléments au langage pour définir ces caractéristiques.

L'API JavaBean définit la classe java.beans.Introspector qui facilite et standardise la recherche des propriétés, méthodes et événements du bean. Cette classe possède des méthodes pour analyser le bean et retourner un objet de type BeanInfo contenant les informations trouvées.

La classe Introspector utilise deux techniques pour retrouver ces informations :

1. un objet de type BeanInfo, s'il y en a un défini par les développeurs du bean
2. les mécanismes fournis par l'API reflexion pour extraire les entités qui respectent leurs modèles (design pattern) respectifs.

Il est donc possible de définir un objet BeanInfo qui sera directement utilisé par la classe Introspector. Cette définition est utile si le bean ne respecte pas certains modèles (design patterns) ou si certaines entités héritées ne doivent pas être utilisables. Dans ce cas, le nom de cette classe doit obligatoirement respecter le modèle XXXBeanInfo ou XXX est le nom du bean correspondant. La classe Introspector recherche une classe respectant ce modèle.

Si une classe BeanInfo est définie pour un bean, une classe qui hérite du bean n'est pas obligée d'en définir une. Dans ce cas, la classe Introspector utilise les informations du BeanInfo de la classe mère et ajoute les informations retournées par l'API Reflection sur le bean.

Sans classe BeanInfo associée au bean, les méthodes de la classe Introspector utilisent les techniques d'introspection pour analyser le bean.

20.5.1. Les modèles (design patterns)

La classe Introspector utilise l'API reflection pour déterminer les informations sur le bean et utilise en même temps un ensemble de modèles sur chacune des entités propriétés, méthodes et événements.

Pour déterminer les propriétés, la classe Introspector recherche les méthodes getXxx(), setXxx() et isXxx() où Xxx représente le nom de la propriété dont la première lettre est en majuscule. La première lettre du nom de la propriété est remise en minuscule sauf si les deux premières lettres de la propriété sont en majuscules.

Pour déterminer les méthodes, la classe Introspector va rechercher toutes les méthodes publiques.

Pour déterminer les événements, la classe Introspector recherche les méthodes addXxxListener() et removeXxxListener(). Si les deux sont présentes, elle en déduit que l'événement xxx est défini dans le bean. Comme pour les propriétés, la première lettre du nom de l'événement est mise en minuscule.

20.5.2. La classe BeanInfo

La classe BeanInfo contient des informations sur un bean et possède plusieurs méthodes pour les obtenir.

La méthode getBeanInfo() prend en paramètre un objet de type Class qui représente la classe du bean et elle renvoie des informations sur la classe et toutes ses classes mères.

Une version surchargée de la méthode accepte deux objets de type Class : le premier représente le bean et le deuxième représente une classe appartenant à la hiérarchie du bean. Dans ce cas, la recherche d'informations s'arrêtera juste avant d'arriver à la classe précisée en deuxième argument.

Exemple : obtenir des informations sur le bean uniquement (sans informations sur ses super-classes)

Exemple (code Java 1.1) :

```
Class monBeanClasse = Class.forName("monBean");
BeanInfo bi = Introspector.getBeanInfo(monBeanClasse, monBeanClasse.getSuperclass());
```

La méthode getBeanDescriptor() permet d'obtenir des informations générales sur le bean en renvoyant un objet de type BeanDescriptor()

La méthode getPropertyDescriptors() permet d'obtenir un tableau d'objets de type PropertyDescriptor qui contiennent les caractéristiques d'une propriété. Plusieurs méthodes permettent d'obtenir ces informations.

Exemple (code Java 1.1) :

```
    PropertyDescriptor[] propertyDescriptor = bi.getPropertyDescriptors();
    for (int i=0; i<propertyDescriptor.length; i++) {
      System.out.println(" Nom propriete    : " +
                        propertyDescriptor[i].getName());
      System.out.println(" Type propriete   : "
                + propertyDescriptor[i].getPropertyType());
      System.out.println(" Getter propriete : "
                  + propertyDescriptor[i].getReadMethod());
      System.out.println(" Setter propriete : "
                 + propertyDescriptor[i].getWriteMethod());
    }
```

La méthode getMethodDescriptors() permet d'obtenir un tableau d'objets de type MethodDescriptor. Cette classe fournit plusieurs méthodes pour extraire les informations des objets contenus dans le tableau.

Exemple (code Java 1.1) :

```
    MethodDescriptor[] methodDescriptor = bi.getMethodDescriptors();
    for (int i=0; i < methodDescriptor.length; i++) {
      System.out.println(" Methode : "+methodDescriptor[i].getName());
    }
```

La méthode getEventSetDescriptors() permet d'obtenir un tableau d'objets de type EventSetDescriptor qui contient les caractéristiques d'un événement. Plusieurs méthodes permettent d'obtenir ces informations.

Exemple (code Java 1.1) :

```
    EventSetDescriptor[] unEventSetDescriptor = bi.getEventSetDescriptors();
    for (int i = 0; i < unEventSetDescriptor.length; i++) {
      System.out.println(" Nom evt             : "
                      + unEventSetDescriptor[i].getName());
      System.out.println(" Methode add evt     : " +
             unEventSetDescriptor[i].getAddListenerMethod());
      System.out.println(" Methode remove evt  : " +
          unEventSetDescriptor[i].getRemoveListenerMethod());
      methodDescriptor = unEventSetDescriptor[i].getListenerMethodDescriptors();
      for (int j = 0; j < methodDescriptor.length; j++) {
          System.out.println(" Event Type: " + methodDescriptor[j].getName());
      }
    }
```

Exemple (code Java 1.1) :

```
import java.io.*;
import java.beans.*;
import java.lang.reflect.*;

public class BeanIntrospection {

  static String nomBean;

  public static void main(String args[]) throws Exception {
    nomBean = args[0];
    new BeanIntrospection();
  }

  public BeanIntrospection() throws Exception {
    Class monBeanClasse = Class.forName(nomBean);
    MethodDescriptor[] methodDescriptor;

    BeanInfo bi = Introspector.getBeanInfo(monBeanClasse, monBeanClasse.getSuperclass());
    BeanDescriptor unBeanDescriptor = bi.getBeanDescriptor();
    System.out.println("Nom du bean    : " + unBeanDescriptor.getName());
    System.out.println("Classe du bean : " + unBeanDescriptor.getBeanClass());
    System.out.println("");

    PropertyDescriptor[] propertyDescriptor = bi.getPropertyDescriptors();
    for (int i=0; i<propertyDescriptor.length; i++) {
      System.out.println(" Nom propriete    : " +
                          propertyDescriptor[i].getName());
      System.out.println(" Type propriete   : "
                × + propertyDescriptor[i].getPropertyType());
      System.out.println(" Getter propriete : "
                  + propertyDescriptor[i].getReadMethod());
      System.out.println(" Setter propriete : "
                 + propertyDescriptor[i].getWriteMethod());
    }
    System.out.println("");
    methodDescriptor = bi.getMethodDescriptors();
    for (int i=0; i < methodDescriptor.length; i++) {
      System.out.println(" Methode : "+methodDescriptor[i].getName());
    }
    System.out.println("");
    EventSetDescriptor[] unEventSetDescriptor = bi.getEventSetDescriptors();
    for (int i = 0; i < unEventSetDescriptor.length; i++) {
      System.out.println(" Nom evt             : "
                        + unEventSetDescriptor[i].getName());
      System.out.println(" Methode add evt     : " +
             unEventSetDescriptor[i].getAddListenerMethod());
      System.out.println(" Methode remove evt  : " +
          unEventSetDescriptor[i].getRemoveListenerMethod());
      methodDescriptor = unEventSetDescriptor[i].getListenerMethodDescriptors();
      for (int j = 0; j < methodDescriptor.length; j++) {
          System.out.println(" Event Type: " + methodDescriptor[j].getName());
      }
    }
    System.out.println("");
  }
}
```

20.6. Paramétrage du bean (Customization)

Il est possible de développer un éditeur de propriétés spécifique pour permettre de personnaliser la modification des paramètres du bean.

La suite de cette section sera développée dans une version future de ce document

20.7. La persistance

Les propriétés du bean doivent pouvoir être sauvegardées pour être restituées ultérieurement. Le mécanisme utilisé est la sérialisation. Pour permettre d'utiliser ce mécanisme, le bean doit implémenter l'interface Serializable.

20.8. La diffusion sous forme de jar

Pour diffuser un bean sous forme de jar, il faut définir un fichier manifest.

Ce fichier doit obligatoirement contenir un attribut Name: qui contient le nom complet de la classe (incluant le package) et un attribut Java-Bean: valorisé à True.

Exemple de fichier manifest pour un bean :

```
    Name: MonBean.class
    Java-Bean: True
```

La suite de cette section sera développée dans une version future de ce document

21. Le logging

Chapitre 21

Niveau : 🟩🟩🟨⬜⬜⬜ Intermédiaire

Le logging consiste à ajouter des traitements dans les applications pour permettre l'émission et le stockage de messages suite à des événements.

Le logging est utile pour tous les types d'applications en permettant par exemple de conserver une trace des exceptions qui sont levées dans l'application et des différents événements anormaux ou normaux liés à l'exécution de l'application.

Le logging permet de gérer des messages émis par une application durant son exécution et de permettre leur exploitation immédiate ou a posteriori. Ces messages sont d'ailleurs très utiles lors de la mise au point d'une application ou lors de son exploitation pour comprendre son fonctionnement ou résoudre une anomalie.

Ce chapitre contient plusieurs sections :

- ♦ La présentation du logging
- ♦ Log4j
- ♦ L'API logging
- ♦ Jakarta Commons Logging (JCL)
- ♦ D'autres API de logging

21.1. La présentation du logging

Le logging est une activité technique utile et nécessaire dans une application pour :

- Déboguer : pratique lorsque la mise en oeuvre d'un débogueur n'est pas facile.
- Obtenir des traces d'exécution (démarrage/arrêt, informations, avertissements, erreurs d'exécution, ...
- Faciliter la recherche d'une source d'anomalie (stacktrace, ...)
- Comprendre ou vérifier le flux des traitements exécutés : traces des entrées/sorties dans les méthodes, affichage de la pile d'appels, ...
- ...

L'importance du logging croît avec la taille et la complexité de l'application qui l'utilise.

Une API de logging fait généralement intervenir trois composants principaux :

- Logger : invoqué pour émettre grâce au framework un message généralement avec un niveau de gravité associé
- Formatter : utilisé pour formater le contenu du message
- Appender : utilisé pour envoyer le message à une cible de stockage (console, fichier, base de données, email, ...)

Le logging doit faire partie intégrante des fonctionnalités d'une application. Bien sûr le niveau de gravité des messages n'est pas le même en développement et en production mais le code de l'application doit rester le même. Seule la configuration du logging doit changer dans les différents environnements.

Généralement la configuration peut être externalisée dans un fichier ce qui rend l'utilisation de l'API plus souple et flexible.

La modification de la configuration du logging en cours d'exécution de l'application (soit dynamiquement soit par rechargement de la configuration) est importante pour permettre d'avoir couramment un niveau de log acceptable et, au besoin, un niveau de log plus fin sans devoir relancer l'application.

Les API de logging ont plusieurs inconvénients :

- Il faut définir avec précision les messages à ajouter dans les journaux et la pertinence des informations qu'ils contiennent
- Il faut définir avec précision le niveau de gravité des messages
- L'utilisation d'une API de logging peut dégrader les performances d'une application

Le logging est particulièrement important dans une application notamment côté serveur mais une utilisation à outrance ou une mauvaise utilisation de cette fonctionnalité peut dégrader les performances générales de l'application.

Les frameworks de logging sont conçus pour limiter la consommation en ressources nécessaires à leur mise en oeuvre mais cette consommation existe tout de même et croît naturellement avec le nombre de messages émis.

L'utilisation d'une API de Logging implique donc une surcharge de consommation de ressources (CPU, mémoire, ...) mais elle se justifie par l'apport des informations fournies en cas de problème sous réserve que ces informations aient été judicieusement choisies.

21.1.1. Des recommandations lors de la mise en oeuvre

Voici quelques règles pour une bonne mise en oeuvre du logging :

- Chaque message doit contenir la date/heure d'émission et la classe émettrice
- Ne jamais utiliser de System.out pour afficher des messages mais utiliser une API de Logging
- Ne jamais utiliser la méthode printStackTrace() de la classe Exception pour afficher des messages mais utiliser une API de Logging
- Eviter les messages émis trop fréquemment (par exemple dans une boucle avec un nombre important d'itérations ou dans une méthode fréquemment invoquée, ...)
- Utiliser le niveau de gravité en adéquation avec le message

Pour des traces d'exécution, il est pratique d'émettre un message en début d'une méthode qui affiche les paramètres en entrée et un message à la fin de la méthode avec la valeur de retour

Il est fortement recommandé d'utiliser une API de logging plutôt que d'utiliser la méthode System.out.println() pour plusieurs raisons :

- Une API de logging permet un contrôle sur le format des messages en proposant un format standard pouvant inclure des données telles que la date/heure, la classe, le thread, ...
- Une API de logging permet de gérer différentes cibles de stockage des messages
- Une API de logging permet de modifier à l'exécution le niveau de gravité des messages pris en compte

Sur des applications utilisées par plusieurs utilisateurs, par exemple une application web, il peut être très utile de faire figurer dans le message une identité sur le responsable de l'action (par exemple, l'adresse IP d'une requête http).

21.1.2. Les différents frameworks

De nombreux frameworks existent pour mettre en oeuvre le logging dont :

- Log4j
- Java Logging
- Jlog
- Protomatter
- SLF4J
- LogBack

Log4j du groupe Apache Jakarta est sûrement l'API la plus répandue et la plus populaire.

Les qualités de Log4j notamment sa simplicité de mise en oeuvre, ses fonctionnalités, sa fiabilité et son évolutivité lui permettent d'être le standard de facto pour le logging.

Depuis la version 1.4 du JDK, Java intègre une API de logging qui est le standard officiel pour le logging. Légèrement moins riche en fonctionnalités que Log4J, elle présente l'avantage d'être fournie dans les API de base.

Afin de faciliter l'utilisation du logging, le groupe Jakarta a développé un wrapper nommé JCL (JakartaCommon Logging) qui permet d'utiliser de façon transparente Log4j ou l'API Logging du JDK en utilisant le tronc commun de ces deux API.

21.2. Log4j

Log4j est un projet open source distribué sous la licence Apache Software initialement créé par Ceki Gï¿½lcï¿½ et maintenu par le groupe Jakarta. Cette API permet aux développeurs d'utiliser et de paramétrer un système de gestion de journaux (logs). Il est possible de fournir les paramètres dans un fichier de configuration ce qui rend sa configuration facile et souple. Log4j est compatible avec le JDK 1.1. et supérieur.

Log4j gère plusieurs niveaux de gravités et les messages peuvent être envoyés dans plusieurs flux : un fichier sur disque, le journal des événements de Windows, une connexion TCP/IP, une base de données, un message JMS, etc ...

Log4j utilise trois composants principaux pour assurer l'envoi de messages selon un certain niveau de gravité et contrôler à l'exécution le format et la ou les cibles de destination des messages :

- Category/Logger : ces classes permettent de gérer les messages associés à un niveau de gravité
- Appenders : ils représentent les flux qui vont recevoir les messages de log
- Layouts : ils permettent de formater le contenu des messages de log

Ces trois types de composants sont utilisés ensemble pour émettre des messages vers différentes cibles de stockage.

Ceci permet au framework de déterminer les messages qui doivent être loggués, la façon de les formater et vers quelle cible les messages seront envoyés.

La popularité de Log4J est largement liée à sa facilité d'utilisation, ses nombreuses fonctionnalités extensibles et sa fiabilité. Comme le logging n'est jamais une fonctionnalité principale d'une application, Log4j se veut facile à mettre en oeuvre.

Les principales caractéristiques de Log4j sont :

- Utilisation d'une hiérarchie de loggers basée sur leurs noms
- Support en standard de plusieurs niveaux de gravité
- Configuration externalisable dans un fichier au format .properties ou XML
- Thread-safe
- Optimisé pour réduire les temps de traitements
- Prise en charge des exceptions associables aux messages
- Support de nombreuses cibles de destination des messages
- Extensible

Un autre avantage de log4J est de pouvoir être utilisé avec toutes les versions du JDK depuis la 1.1.

L'externalisation de la configuration de Log4j dans un fichier externe permet de modifier la configuration des traitements de logging sans avoir à modifier le code source de l'application.

La hiérarchie des loggers permet un contrôle très fin de la granularité des messages ce qui réduit le volume de données des logs.

Log4j propose en standard plusieurs destinations de stockage des messages : fichiers, gestion d'événements Windows, Syslog Unix, base de données, email, message JMS, ...

L'API Log4j est regroupée dans plusieurs packages :

Package	Rôle
org.apache.log4j	Contient les principales classes et interfaces
org.apache.log4j.spi	System Programming Interface pour étendre Log4j
org.apache.log4j.chainsaw	Application Swing Chainsaw pour visualiser les logs formatées par un XMLLayout ou émises par un SocketAppender
org.apache.log4j.config	Classes pour la gestion des propriétés des composants
org.apache.log4j.helpers	Utilitaires
org.apache.log4j.jdbc	Classes pour stocker les messages dans une base de données
org.apache.log4j.jmx	Classes pour permettre la configuration de Log4j grâce à JMX
org.apache.log4j.lf5	Application Swing Log Force 5 pour visualiser les logs
org.apache.log4j.net	Classes pour envoyer les messages à travers le réseau (JMS, SMTP, Sockets, ...)
org.apache.log4j.nt	Classes pour envoyer les messages dans le système de gestion des événements de Windows
org.apache.log4j.or	Utilitaires pour formater des objets
org.apache.log4j.performance	Classes de tests des performances
org.apache.log4j.xml	Classes pour permettre la configuration de Log4j avec un fichier XML
org.apache.log4j.varia	Classes diverses

Le site officiel de Log4j est à l'url : logging.apache.org/log4j/

Log4j est disponible dans trois versions majeures :

- 1.2 : c'est la version stable courante
- 1.3 : cette version est abandonnée
- 2.0 : c'est la future version en cours de développement

21.2.1. Les premiers pas

Cette section fournit des informations et un premier exemple pour la mise en oeuvre de Log4J.

21.2.1.1. L'installation

Il faut télécharger le fichier apache-log4j-1.2.xx.zip à l'url logging.apache.org/log4j/1.2/download.html

Il suffit ensuite de décompresser l'archive dans un répertoire du système. L'archive contient entre autres les sources, la documentation, des exemples et la bibliothèque log4j-1.2.x.jar.

21.2.1.2. Les principes de mise en oeuvre

Pour utiliser Log4j, il suffit d'ajouter le fichier log4j-1.2.x.jar dans le classpath de l'application.

Il faut définir un fichier de configuration : configuration des loggers, définition des appenders, association des appenders aux loggers avec un layout.

Dans le code source des classes, il faut :

- obtenir une instance du logger relative à la classe
- utiliser l'API pour émettre un message associé à un niveau de gravité

21.2.1.3. Un exemple de mise en oeuvre

Cette section va mettre en oeuvre Log4j dans un exemple très simple.

Il faut créer un fichier log4j.properties stocké dans le classpath de l'application : ce fichier contient la configuration de Log4j pour l'application.

Exemple :

```
log4j.rootLogger=DEBUG, stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d [%-5p] (%F:%M:%L) %m%n
```

Cette configuration définit le niveau de gravité DEBUG pour le logger racine et lui associe un logger nommé arbitrairement stdout. Par héritage, tous les loggers de l'application vont hériter de cette configuration.

L'appender nommé stdout est de type ConsoleAppender : il envoie les messages sur la console standard.

Un layout personnalisé est associé à l'appender nommé stdout pour formater les messages. Chaque séquence commençant par le caractère % sera remplacée dynamiquement par sa valeur correspondante. Par exemple : %d correspond à la date/heure, %p au niveau de gravité, %m le message, %n une nouvelle ligne, ...

Pour mettre en oeuvre l'API dans le code source, il faut tout d'abord obtenir une instance du logger à utiliser en utilisant la méthode getLogger() de la classe Logger.

Chaque message est émis en utilisant la méthode correspondant au niveau de gravité choisi de la classe Logger.

Exemple :

```
package fr.jmdoudoux.dej.log4j;

import org.apache.log4j.Logger;

public class TestLog4j1 {

  private static Logger logger = Logger.getLogger(TestLog4j1.class);

  public static void main(String[] args) {
    logger.debug("msg de debogage");
    logger.info("msg d'information");
    logger.warn("msg d'avertissement");
    logger.error("msg d'erreur");
    logger.fatal("msg d'erreur fatale");
  }
}
```

L'exécution de cette classe permet d'afficher sur la console les différents messages

Résultat :

```
2008-06-08 10:16:21,546 [DEBUG] (TestLog4j1.java:main:13) msg de debogage
2008-06-08 10:16:21,546 [INFO ] (TestLog4j1.java:main:14) msg d'information
2008-06-08 10:16:21,546 [WARN ] (TestLog4j1.java:main:15) msg d'avertissement
2008-06-08 10:16:21,546 [ERROR] (TestLog4j1.java:main:16) msg d'erreur
2008-06-08 10:16:21,546 [FATAL] (TestLog4j1.java:main:17) msg d'erreur fatale
```

Une simple modification du fichier de configuration permet de changer le niveau de gravité des messages pris en compte. Par exemple en remplaçant DEBUG par ERROR

La réexécution de la classe qui n'a pas été modifiée et donc pas recompilée permet d'afficher sur la console uniquement les messages dont la gravité est supérieure ou égale à ERROR.

Résultat :

```
2008-06-08 10:18:47,530 [ERROR] (TestLog4j1.java:main:13) msg d'erreur
2008-06-08 10:18:47,530 [FATAL] (TestLog4j1.java:main:14) msg d'erreur fatale
```

21.2.2. La gestion des logs avec les versions antérieures à la 1.2

Les versions antérieures à la 1.2 de Log4J utilisaient les classes Category pour gérer les messages et la Priority pour encapsuler les niveaux de gravité.

21.2.2.1. Les niveaux de gravités : la classe Priority

Log4j gère des priorités pour permettre à une instance de la classe Category de déterminer si le message sera envoyé dans le log ou non. Il existe cinq priorités qui possèdent un ordre hiérarchique croissant :

- DEBUG
- INFO
- WARN
- ERROR
- FATAL

La classe org.apache.log4j.Priority encapsule ces priorités.

Chaque Category est associée à une priorité qui peut être changée dynamiquement. La catégorie détermine si un message doit être envoyé dans le log en comparant sa priorité avec la priorité du message. Si celle-ci est supérieure ou égale à la priorité de la Category, alors le message est envoyé vers la cible de destination du log.

La méthode setPropriety() de la classe Category permet de préciser la priorité.

Si aucune priorité n'est donnée à une catégorie, elle "hérite" de la priorité de la première catégorie renseignée trouvée en remontant dans la hiérarchie.

Exemple : soit trois catégories
root associée à la priorité INFO
categorie1 nommée "org" sans priorité particulière
categorie2 nommée "org.moi" associée à la priorité ERROR
categorie3 nommée "org.moi.projet" sans priorité particulière

Une demande d'émission de message avec la priorité DEBUG sur categorie1 n'est pas traitée car la priorité INFO héritée est supérieure à DEBUG.
Une demande avec la priorité WARN sur categorie1 est traitée car la priorité INFO héritée est inférieure à WARN .
Une demande avec la priorité DEBUG sur categorie3 n'est pas traitée car la priorité ERROR héritée est supérieure à DEBUG.
Une demande avec la priorité FATAL sur categorie3 est traitée car la priorité ERROR héritée est inférieure à FATAL.
En fait dans l'exemple, aucune demande avec la priorité DEBUG ne sera traitée.

Au niveau applicatif, il est possible d'interdire le traitement d'une priorité et de celle inférieure en utilisant le code suivant : Category.getDefaultHierarchy().disable(). Il faut fournir la priorité à la méthode disable().

Il est possible d'annuler ce traitement dynamiquement en positionnant la propriété système log4j.disableOverride.

21.2.2.2. La classe Category

La classe org.apache.log4j.Category détermine si un message doit être envoyé dans le ou les logs qui lui sont associés.

Chaque Category possède un nom qui est sensible à la casse. Pour créer une instance de la classe Category il faut utiliser la méthode statique getInstance() qui attend en paramètre le nom de la Category. Si une Category existe déjà avec le nom fourni, alors la méthode getInstance() renvoie l'instance existante.

Il est pratique de fournir le nom complet de la classe comme nom de la Category dans laquelle elle est instanciée mais ce n'est pas une obligation. Il est ainsi possible de créer une hiérarchie spécifique différente de celle de l'application, par exemple basée sur des aspects fonctionnels. L'inconvénient d'associer le nom de la classe au nom de la catégorie est qu'il faut instancier un objet Category dans chaque classe : le plus pratique est de déclarer cet objet static.

Exemple :

```
public class Classe1 {
  static Category category = Category.getInstance(Classe1.class.getName());
  ...
}
```

La méthode log(Priority, Object) permet de demander l'émission d'un message associé au niveau de gravité fourni en paramètre. Plusieurs méthodes sont des raccourcis qui évitent d'avoir à préciser le niveau de gravité car celui utilisé sera automatiquement celui associé à la méthode (debug(Object), info(Object), warn(Object), error(Object), fatal(Object)).

Toutes ces méthodes possèdent une surcharge qui attend en paramètre supplémentaire un objet de type Throwable. Ces méthodes ajouteront automatiquement au message la pile d'appels (stacktrace) de l'exception.

La demande est traitée en fonction de la hiérarchie de la Category et de la priorité du message.

Pour éviter d'éventuels traitements inutiles de création du message, il est possible d'utiliser la méthode isEnabledFor(Priority) pour savoir si la catégorie prend en compte la priorité ou non.

Exemple :

```
import org.apache.log4j.*;

public class TestIsEnabledFor {

  static Category cat1 = Category.getInstance(TestIsEnabledFor.class.getName());

  public static void main(String[] args) {
    int i=1;
    int[] occurrence={10,20,30};

    BasicConfigurator.configure();

    cat1.setPriority(Priority.WARN) ;
    cat1.warn("message de test");

    if(cat1.isEnabledFor(Priority.INFO)) {
      System.out.println("traitement du message de priorité INFO");
      cat1.info("La valeur de l'occurrence "+i+" = " + String.valueOf(occurrence[i]));
    }
    if(cat1.isEnabledFor(Priority.WARN)) {
      System.out.println("traitement du message de priorité WARN");
      cat1.warn("La valeur de l'occurrence "+i+" = " + String.valueOf(occurrence[i]));
    }
  }
}
```

Résultat :

```
0 [main] WARN TestIsEnabledFor  - message de test
traitement du message de priorit_ WARN
50 [main] WARN TestIsEnabledFor  - La valeur de l'occurrence 1 = 20
```

Le nom de la Category permet d'établir une hiérarchie dans les Category : ce nom est composé de mots séparés par un caractère point comme pour les packages. D'ailleurs par simplicité et par convention c'est le nom pleinement qualifié de la classe qui est utilisé.

Il existe toujours une catégorie racine créée par Log4J : pour obtenir une instance de cette Category, il faut utiliser la méthode getRoot() de la classe Category car elle ne possède pas de nom.

La méthode getInstance() de la classe Category renvoie toujours la même instance pour un même nom de catégorie. Si cette instance n'existe pas alors la méthode la créée sinon elle retourne celle existante.

Le message n'est pris en compte que si son niveau de gravité est supérieur ou égal à celui de la catégorie.

Par défaut, une Category hérite du niveau de gravité de sa Category mère selon la hiérarchie des catégories basée sur leurs noms. Ceci est possible car la Category racine à un niveau de gravité par défaut initialisé à DEBUG.

Par exemple, la catégorie fr.jmdoudoux.dej.log4j hérite des caractéristiques de la catégorie fr.jmdoudoux.dej.

Il est possible d'associer un niveau de gravité à la Category de façon statique en utilisant la méthode setPriority().

21.2.2.3. La hiérarchie dans les catégories

Le nom de la catégorie permet de la placer dans une hiérarchie dont la racine est une catégorie spéciale nommée root qui est créée par défaut sans nom.

La classe Category possède une méthode statique getRoot() pour obtenir la catégorie racine.

La hiérarchie des noms est établie grâce à la notation par point comme pour les packages. D'ailleurs par convention, le nom de la catégorie correspond généralement au nom pleinement qualifié de la classe qui va utiliser la catégorie.

Exemple : soit trois catégories
categorie1 nommée "org"
categorie2 nommée "org.moi"
categorie3 nommée "org.moi.projet"

Categorie3 est fille de categorie2, elle-même fille de categorie1.

Cette relation hiérarchique est importante car la configuration établie pour une catégorie est automatiquement propagée par défaut aux catégories enfants.

L'ordre de la création des catégories de la hiérarchie ne doit pas obligatoirement respecter l'ordre de la hiérarchie. Celle-ci est constituée au fur et à mesure de la création des catégories.

21.2.3. La gestion des logs à partir de la version 1.2

Les classes Category et Priority sont déclarées deprecated et sont remplacées respectivement par les classes Logger et Level qui en héritent.

21.2.3.1. Les niveaux de gravité : la classe Level

A partir de la version 1.2 de Log4j, la classe Priority ne doit plus être utilisée : il est préférable d'utiliser sa classe fille Level.

Attention la classe Priority n'est pas marquée deprecated car la classe Level en hérite.

La classe org.apache.log4j.Level encapsule donc un niveau de gravité.

Log4j définit plusieurs niveaux de gravité en standard et possédant un ordre hiérarchique :

- TRACE : correspond à des messages de traces d'exécution (depuis la version 1.2.12)
- DEBUG : correspond à des messages de débogage
- INFO : correspond à des messages d'information
- WARN : correspond à des messages d'avertissement
- ERROR : correspond à des messages d'erreur
- FATAL : correspond à des messages liés à un arrêt imprévu de l'application

Deux autres niveaux particuliers sont définis et utilisés dans la configuration :

- OFF : aucun niveau de gravité n'est pris en compte
- ALL : tous les niveaux de gravité sont pris en compte

Il est possible de définir ses propres niveaux de gravité en créant une classe qui hérite de la classe Level.

Le choix du niveau de gravité associé à un message est très important. Voici quelques exemples d'utilisation selon chaque niveau de gravité :

Niveau de gravité	Exemple d'utilisation
TRACE	Entrée et sortie de méthodes
DEBUG	Affichage de valeur de données
INFO	Chargement d'un fichier de configuration, début et fin d'exécution d'un traitement long
WARN	Erreur de login, données invalides
ERROR	Toutes les exceptions capturées qui n'empêchent pas l'application de fonctionner
FATAL	Indisponibilité d'une base de données, toutes les exceptions qui empêchent l'application de fonctionner

21.2.3.2. La classe Logger

A partir de la version 1.2 de Log4j, la classe Category ne doit plus être utilisée : il est préférable d'utiliser sa classe fille Logger.

Attention la classe Category n'est pas marquée deprecated car la classe Logger en hérite.

La classe org.apache.log4j.Logger permet donc comme la classe Category de demander l'envoi d'un message dans le système de logs. Un logger compare son niveau de gravité avec celui du message : si ce dernier est supérieur ou égal à celui du logger alors le message est traité.

Un logger est associé à un ou plusieurs appenders : si le message est à traiter, celui-ci est envoyé par le logger à ses appenders.

La classe Logger héritant de la classe Category, elle possède toutes ses méthodes notamment celles permettant l'émission d'un message. L'émission de messages se fait donc en utilisant la méthode log() ou une des méthodes utilisant implicitement un niveau de gravité (debug(), info(), warn(), error(), fatal()).

Exemple : les deux lignes de code sont équivalentes

Exemple :

```
logger.log(Level.INFO, "mon message");
logger.info("mon message");
```

Pour obtenir une instance de la classe Logger, il faut utiliser sa méthode statique getLogger(). Cette méthode attend en paramètre le nom du logger.

Exemple :

```
package fr.jmdoudoux.dej.log4j;

import org.apache.log4j.Logger;

public class MaClasse {
  private static final Logger logger = Logger.getLogger("fr.jmdoudoux.dej.log4j.MaClasse");
}
```

Comme généralement ce nom correspond au nom pleinement qualifié de la classe, une version surchargée de la méthode getLogger() attend en paramètre un objet de type Class pour en extraire le nom.

Exemple :

```
package fr.jmdoudoux.dej.log4j;

import org.apache.log4j.Logger;

public class MaClasse {
  private static final Logger logger = Logger.getLogger(MaClasse.class);
}
```

La méthode getLogger() permet de s'assurer que pour un même nom cela soit toujours la même instance qui est retournée.

Exemple :

```
package fr.jmdoudoux.dej.log4j;

import org.apache.log4j.Logger;

public class TestLog4j9 {

  public static void main(String[] args) {
    Logger loggerA = Logger.getLogger("fr.jmdoudoux.dej.log4j");
    Logger loggerB = Logger.getLogger("fr.jmdoudoux.dej.log4j");

    System.out.println("loggerA == loggerB : "+(loggerA==loggerB));
  }
}
```

Résultat :

```
loggerA == loggerB : true
```

Le nom de chaque Logger permet de définir une hiérarchie pour permettre de faciliter leur configuration. Cette hiérarchie sur les noms repose sur l'utilisation du caractère point comme pour les packages. Il est dès lors pratique d'utiliser le nom pleinement qualifié de la classe comme nom de logger pour une classe.

Le nom des logger est sensible à la casse.

La hiérarchie commence toujours par un Logger fournit par Log4j : le RootLogger. Pour obtenir une instance de ce logger racine, il faut utiliser la méthode getRootLogger() de la classe Logger.

Le rootLogger a deux caractéristiques distinctives par rapport aux autres loggers :

- Il existe toujours
- Il n'a pas de nom

Lors de la création de l'instance d'un Logger, la hiérarchie est parcourue pour déterminer le Logger le plus proche de la hiérarchie. A défaut, ce sont les caractéristiques du rootLogger qui sont attribuées au nouveau Logger.

L'ordre de création des loggers n'a pas d'importance : il n'est pas obligatoire de créer les 1oggers dans leur ordre hiérarchique

Chaque Logger et chaque message possèdent un niveau de gravité. Le Logger compare son niveau de gravité avec celui du message : si le niveau de gravité du message est égal ou supérieur au niveau de gravité du Logger, alors le message est traité par le framework sinon il est ignoré.

Exemple : le message ne sera jamais pris en compte

Exemple :

```
Logger logger = Logger.getLogger("fr.jmdoudoux.dej.log4j");
logger.setLevel(Level.INFO);
logger.debug("mon message");
```

Chaque logger est associé à un niveau de gravité soit directement soit indirectement par héritage du niveau de gravité de son père dans la hiérarchie. Si le logger ne possède par de niveau de gravité explicite alors c'est celui de son ancêtre le plus proche dans la hiérarchie des loggers.

Comme le logger racine à un niveau de gravité par défaut, cela implique qu'un logger à toujours un niveau de gravité qui lui est associé.

Si aucun logger ne possède de niveau de gravité explicite dans la hiérarchie alors le niveau du logger racine (rootLogger) est utilisé. Le rootLogger est toujours définit avec un niveau de gravité qui par défaut est debug.

Il est possible de configurer un logger par programmation.

Il est possible d'associer de façon statique un niveau de gravité au logger en utilisant la méthode setLevel(). Il est cependant préférable d'utiliser la configuration dynamique en utilisant un fichier de configuration qui permet de modifier les paramètres sans modifier le code source.

21.2.3.3. La migration de Log4j antérieure à 1.2 vers 1.2

La migration de l'utilisation des classes Category vers Logger et Priority vers Level peut généralement être faite grâce à un rechercher/remplacer dans le code source :

Rechercher	Remplacer par
Category.getInstance	Logger.getLogger
Category.getRoot	Logger.getRootLogger
Category	Logger
Priority	Level

21.2.4. Les Appender

La cible de destination des messages est encapsulée dans un ou plusieurs objets de type Appender.

L'interface org.apache.log4j.Appender désigne un flux qui représente le log et se charge de l'envoi de messages formatés dans le flux. Le formatage proprement dit est réalisé par un objet de type Layout. Ce layout peut être fourni dans le constructeur adapté ou par la méthode setLayout().

Une Category ou un Logger peuvent avoir plusieurs appenders. Si la Category ou le Logger décident de traiter la demande d'un message, le message est envoyé à chacun des appenders. Pour ajouter manuellement un appender à une Category, il suffit d'utiliser la méthode addAppender() qui attend en paramètre un objet de type Appender.

L'interface Appender est directement implémentée par la classe abstraite AppenderSkeleton.

Cette classe est la classe mère de toutes les classes fournies avec Log4j pour représenter un type de log. Log4J propose plusieurs appenders en standard :

- AsyncAppender : messages envoyés vers différents appenders de façon périodique et asynchrone
- ExternalyRolledFileAppender : messages envoyés sur un fichier à rotation à la réception d'un message dédié sur une socket et envoi d'un accusé de traitement
- JDBCAppender : messages envoyés dans une base de données (attention à son utilisation)
- JMSAppender : messages envoyés vers une destination utilisant JMS
- LF5Appender : messages envoyés sur une application Swing dédiée
- NTEventLogAppender : messages envoyés dans le log des événements système sur Windows à partir de NT
- NullAppender : messages ignorés
- SMTPAppender : messages envoyés par mail
- SocketAppender : messages envoyés dans une socket
- SocketHubAppender : messages envoyés dans plusieurs sockets
- SyslogAppender : messages envoyés dans le démon syslog d'un système Unix
- TelnetAppender : messages envoyés dans une socket en lecture seule facilement consultable avec l'outil telnet
- WriterAppender : cette classe possède deux classes filles : ConsoleAppender et FileAppender.
- ConsoleAppender : messages envoyés sur la console
- FileAppender : messages envoyés dans un fichier. La classe FileAppender possède deux classes filles : DailyRollingAppender et RollingFileAppender
- DailyRollingFileAppender : messages envoyés dans un fichier à rotation périodique (pas obligatoirement journalière)
- RollingFileAppender : messages envoyés dans un fichier à rotation selon sa taille

Pour créer un appender par programmation, il suffit d'instancier un objet d'une de ces classes.

Chaque appender possède des paramètres de configuration dédiés.

Comme un Logger peut avoir plusieurs appenders, un même message peut être envoyé vers plusieurs appenders selon la configuration. La méthode addAppender() de la classe Logger permet d'ajouter manuellement un appender au logger.

Comme pour les niveaux de gravité, les appenders d'une catégorie ou d'un logger sont hérités implicitement par défaut de la hiérarchie des loggers.

Il est possible d'inhiber cet héritage pour une partie de la hiérarchie en utilisant la méthode setAdditivity() avec le paramètre false sur l'instance du logger concerné. Ce logger et sa hiérarchie descendante n'hériteront pas des caractéristiques de leur parent.

Exemple :

```
package fr.jmdoudoux.dej.log4j;

import java.io.IOException;

import org.apache.log4j.ConsoleAppender;
import org.apache.log4j.FileAppender;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.SimpleLayout;
import org.apache.log4j.xml.XMLLayout;

public class TestLog4j10 {

  public static void main(
      String[] args) {
    Logger logRoot = Logger.getRootLogger();
    ConsoleAppender ca = new ConsoleAppender();
    ca.setName("console");
    ca.setLayout(new SimpleLayout());
    ca.activateOptions();
    logRoot.addAppender(ca);
    logRoot.setLevel(Level.DEBUG);

    logRoot.debug("message 1");

    Logger log = Logger.getLogger(TestLog4j10.class);

    log.setAdditivity(false);
    try {
      FileAppender fa = new FileAppender(new XMLLayout(), "c:/log.txt");
      fa.setName("FichierLog");
      log.addAppender(fa);
    } catch (IOException e) {
      e.printStackTrace();
    }

    log.debug("message 2");

    Logger logTest = Logger.getLogger("fr.jmdoudoux.dej.log4j");
    logTest.debug("message 3");
  }
}
```

Résultat dans la console :

```
DEBUG - message 1
DEBUG - message 3
```

Résultat dans le fichier de log :

```
<log4j:event logger="fr.jmdoudoux.dej.log4j.TestLog4j10" timestamp="1231923298709"
      level="DEBUG" thread="main">
<log4j:message><![CDATA[message 2]]></log4j:message>
</log4j:event>
```

Avant qu'ils ne puissent être utilisés, la plupart des appenders nécessitent un appel à leur méthode activateOptions() lorsqu'ils sont configurés par programmation.

Il est possible de définir son propre appender en définissant une classe qui implémente l'interface Appender ou qui hérite de la classe AppenderSkeleton.

21.2.4.1. AsyncAppender

La classe org.apache.log4j.AsyncAppender envoie les messages vers différents appenders de façon périodique et asynchrone. Cet appender utilise son propre thread.

Cet appender n'est configurable que dans un fichier de configuration au format XML.

Un tag fils <appender-ref> permet de préciser un appender vers lequel les messages seront envoyés. L'attribut ref permet de préciser le nom de l'appender concerné.

L'attribut bufferSize permet de préciser le nombre de messages qui seront stockés dans le tampon.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration debug="false"
  xmlns:log4j="http://jakarta.apache.org/log4j/">
  <appender name="console" class="org.apache.log4j.ConsoleAppender">
    <param name="Target" value="System.out" />
    <layout class="org.apache.log4j.SimpleLayout" />
  </appender>
  <appender class="org.apache.log4j.FileAppender" name="file">
    <param name="file" value="c:/monapp.log" />
    <layout class="org.apache.log4j.SimpleLayout" />
  </appender>
  <appender class="org.apache.log4j.AsyncAppender" name="async">
    <param name="bufferSize" value="2" />
    <appender-ref ref="file" />
    <appender-ref ref="console" />
  </appender>
  <root>
    <level value="info" />
    <appender-ref ref="async" />
  </root>
</log4j:configuration>
```

21.2.4.2. JDBCAppender

La classe org.apache.log4j.jdbc.JDBCAppender envoie les messages dans une base de données.

Cet appender possède plusieurs propriétés notamment pour préciser les paramètres de connexion à la base de données.

Nom	Rôle
BufferSize	Nombre de messages stockés dans le tampon avant l'insertion dans la base de données
Driver	Pilote JDBC pour l'accès à la base de données
Url	Url de connexion à la base de données
Password	Mot de passe de connexion
User	Utilisateur de connexion
Sql	Requête SQL pour insérer une occurrence dans la base de données

La propriété Sql permet de définir la requête SQL qui permet l'insertion des informations sur le message dans la base de données. La requête doit utiliser les séquences utilisées par le layout PatternLayout

Exemple :

```
INSERT INTO log(dthr, niveau, message) VALUES('%d', '%p', '%m');".
```

Attention : l'utilisation de cet appender fourni par Log4J n'est pas recommandée. Pour plus d'informations consultez la documentation de l'API.

21.2.4.3. JMSAppender

La classe org.apache log4j.net.JMSAppender envoie les message vers une destination JMS.

21.2.4.4. LF5Appender

La classe org.apache.log4j.lf5.LF5Appender envoie les messages sur une application Swing dédiée.

21.2.4.5. NTEventLogAppender

La classe org.apache.log4j.nt.NTEventLogAppender envoie les messages dans le log des événements système sur Windows à partir de Windows NT

21.2.4.6. NullAppender

La classe org.apache.log4j.varia.NullAppender ignore les messages qui lui sont envoyés.

La seule propriété d'un NullAppender est :

Nom	Rôle
Threshold	Limiter les messages pris en compte par l'appender à ceux dont le niveau de gravité est supérieur ou égal à celui de threshold. Ceci vient en plus du niveau de gravité associé au logger. Héritée de AppenderSkeleton

21.2.4.7. SMTPAppender

La classe org.apache.log4j.net.SMTPAppender envoie les messages par mail.

La classe SMTPAppender possède plusieurs attributs :

Nom	Rôle
BufferSize	Nombre de messages inclus dans un mail
SMTPHost	Nom de la machine qui héberge le serveur SMTP
From	Email de l'émetteur du mail
To	Email du ou des destinataires du mail
Subject	Sujet du mail
Cc	Email du ou des destinataires en copie du mail
Bcc	Email du ou des destinataires en copie cachée du mail
SMTPPassword	Mot de passe
SMTPUsername	Utilisateur

Par défaut, seuls les messages avec un niveau de gravité supérieur ou égal à ERROR sont traités par cet appender.

Cet appender requiert les bibliothèques JavaBeans Activation Framework et JavaMail pour fonctionner.

21.2.4.8. SocketAppender

La classe org.apache.log4j.net.SocketAppender envoie les messages dans une socket utilisant TCP/IP.

Les données envoyées sont des objets de type LoggingEvent sérialisés.

La classe SocketAppender possède plusieurs attributs :

Nom	Rôle
LocationInfo	Booléen qui précise si des informations de localisation sont envoyées. Par défaut la valeur est false.

Port	Port de la machine hôte à utiliser.
RemoteHost	Chaîne de caractères qui précise la machine hôte

La méthode activateOptions() permet de réaliser la connexion.

Exemple :

```
package fr.jmdoudoux.dej.log4j;

import org.apache.log4j.Logger;
import org.apache.log4j.SimpleLayout;
import org.apache.log4j.net.SocketAppender;

public class TestLog4j18 {
  static Logger logger = Logger.getLogger(TestLog4j18.class);

  public static void main(
      String args[]) {
    SocketAppender appender = null;
    try {
      appender = new SocketAppender();
      appender.setPort(10256);
      appender.setRemoteHost("localhost");
      appender.setLocationInfo(true);
      appender.setLayout(new SimpleLayout());
      appender.activateOptions();
    } catch (Exception e) {
      e.printStackTrace();
    }
    logger.addAppender(appender);

    while (true) {
      System.out.println("envoie log");
      logger.debug("msg de debogage");
      logger.info("msg d'information");
      logger.warn("msg d'avertissement");
      logger.error("msg d'erreur");
      logger.fatal("msg d'erreur fatale");
      try {
        Thread.sleep(5000);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
}
```

La configuration dans le fichier properties est similaire aux autres appenders.

Exemple :

```
log4j.appender.socket=org.apache.log4j.net.SocketAppender
log4j.appender.socket.RemoteHost=localhost
log4j.appender.socket.Port=10256
log4j.appender.socket.LocationInfo=true
```

La configuration dans le fichier XML est similaire aux autres appenders.

Exemple :

```
...
  <appender name="socket" class="org.apache.log4j.net.SocketAppender">
    <param name="Port" value="10256"/>
    <param name="RemoteHost" value="localhost"/>
    <param name="LocationInfo" value="true"/>
  </appender>
```

```
...
```

21.2.4.9. SocketHubAppender

La classe org.apache.log4j.net.SocketHubAppender envoie les messages dans plusieurs sockets.

21.2.4.10. SyslogAppender

La classe org.apache.log4j.net.SyslogAppender envoie les messages dans le démon syslog d'un système Unix

21.2.4.11. TelnetAppender

La classe org.apache.log4j.net.TelnetAppender envoie les messages dans une socket en lecture seule facilement consultable avec l'outil telnet.

21.2.4.12. WriterAppender

La classe org.apache.log4j.WriterAppender possède deux classes filles : ConsoleAppender et FileAppender. La classe FileAppender possède, elle aussi, deux classes filles : DailyRollingAppender et RollingFileAppender.

Elles possèdent plusieurs propriétés dont :

Nom	Rôle	Valeur par défaut
Encoding	Préciser le jeu de caractères à utiliser.	null
ImmediateFlush	Préciser si le tampon doit être vidé à chaque opération (pas de stockage dans un tampon).	true

Exemple :

```
package fr.jmdoudoux.dej.log4j;

import java.io.FileOutputStream;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.WriterAppender;
import org.apache.log4j.xml.XMLLayout;

public class TestLog4j11 {

  public static void main(
      String[] args) {
    Logger logRoot = Logger.getRootLogger();

    WriterAppender appender = null;
    try {
      appender = new WriterAppender(new XMLLayout(), new FileOutputStream("c:/malog.txt"));

    } catch (Exception e) {
      e.printStackTrace();
    }

    logRoot.addAppender(appender);
    logRoot.setLevel(Level.DEBUG);

    logRoot.debug("mon message");
  }
}
```

Résultat : le contenu du fichier c:\malog.txt

```
<log4j:event logger="root" timestamp="1219683683344" level="DEBUG" thread="main">
<log4j:message><![CDATA[mon message]]></log4j:message>
</log4j:event>
```

21.2.4.13. ConsoleAppender

La classe org.apache.log4j.ConsoleAppender envoie les messages sur la console : soit sur la sortie standard (System.out) par défaut soit vers la sortie d'erreurs (System.err).

Les propriétés d'un ConsoleAppender sont :

Nom	Rôle	Valeur par défaut
Encoding	Préciser le jeu de caractères à utiliser. Héritée de WriterAppender	null
ImmediateFlush	Envoyer les messages immédiatement vers la console (pas de mise dans un tampon). Héritée de WriterAppender	true
Target	System.out ou System.err	System.out
Threshold	Limiter les messages pris en compte par l'appender à ceux dont le niveau de gravité est supérieur ou égal à celui de threshold. Ceci vient en plus du niveau de gravité associé au logger. Héritée de AppenderSkeleton	

Exemple :

```
package fr.jmdoudoux.dej.log4j;

import org.apache.log4j.ConsoleAppender;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.SimpleLayout;

public class TestLog4j12 {

  public static void main(
      String[] args) {
    Logger logRoot = Logger.getRootLogger();
    ConsoleAppender ca = new ConsoleAppender();
    ca.setName("console");
    ca.setLayout(new SimpleLayout());
    ca.activateOptions();
    logRoot.addAppender(ca);
    logRoot.setLevel(Level.DEBUG);

    logRoot.info("mon message");
  }
}
```

Résultat :

```
2008-06-15 10:22:02,925 [INFO ] (TestLog4j12.java:main:20) mon message
INFO - mon message
```

21.2.4.14. FileAppender

La classe org.apache.log4j.FileAppender envoie les messages dans un fichier.

Les propriétés d'un FileAppender sont :

Nom	Rôle	Valeur par défaut
ImmediateFlush	Envoyer les messages immédiatement vers le fichier (pas de stockage dans un tampon). Héritée de WriterAppender	true
Threshold	Limiter les messages pris en compte par l'appender à ceux dont le niveau de gravité est supérieur ou égal à celui de threshold. Ceci vient en plus du niveau de gravité associé au logger. Héritée de AppenderSkeleton	
Append	Ajouter le message à la fin du fichier ou remplacer le contenu du fichier	True
Encoding	Jeu de caractères utilisé pour l'encodage	
BufferedIO	Préciser si un tampon doit être utilisé	False
BufferSize	Préciser la taille du tampon s'il est utilisé	
File	Nom du fichier	

Exemple :

```
package fr.jmdoudoux.dej.log4j;

import org.apache.log4j.FileAppender;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.SimpleLayout;

public class TestLog4j13 {

  public static void main(
      String[] args) {
    Logger logRoot = Logger.getRootLogger();

    FileAppender appender = null;
    try {
      appender = new FileAppender();

      appender.setLayout(new SimpleLayout());
      appender.setFile("c:/app_log.txt");
      appender.activateOptions();
      logRoot.addAppender(appender);
      logRoot.setLevel(Level.DEBUG);

      logRoot.info("mon message");

    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

21.2.4.15. DailyRollingFileAppender

La classe org.apache.log4j.DailyRollingFileAppender envoie les messages dans un fichier à rotation périodique (qui n'est pas obligatoirement journalière).

Les propriétés d'un DailyRollingFileAppender sont :

Nom	Rôle	Valeur par défaut
ImmediateFlush	Envoyer les messages immédiatement vers le fichier (pas de mise dans un tampon). Héritée de WriterAppender	true
Threshold	Limiter les messages pris en compte par l'appender à ceux dont le niveau de gravité est supérieur ou égal à celui de threshold. Ceci vient en plus du niveau de gravité associé au logger. Héritée de AppenderSkeleton	

Append	Ajouter le message à la fin du fichier ou remplacer le contenu du fichier. Héritée de FileAppender	True
Encoding	Préciser le jeu de caractères utilisé pour l'encodage. Héritée de WriterAppender	
BufferedIO	Préciser si un tampon doit être utilisé. Héritée de FileAppender	False
BufferSize	Préciser la taille du tampon s'il est utilisé. Héritée de FileAppender	
File	Nom du fichier. Héritée de FileAppender	
DatePattern	Définir la périodicité de rotation et le suffixe des noms des fichiers créés à chaque rotation	

La valeur de la propriété DatePattern suit le format utilisé par la classe SimpleDateFormat.

Exemple :

'.'yyyy-MM: rotation chaque mois

'.'yyyy-ww: rotation chaque semaine

'.'yyyy-MM-dd: rotation chaque jour à minuit

'.'yyyy-MM-dd-a: rotation chaque jour à midi et à minuit

'.'yyyy-MM-dd-HH: rotation chaque heure

Exemple :

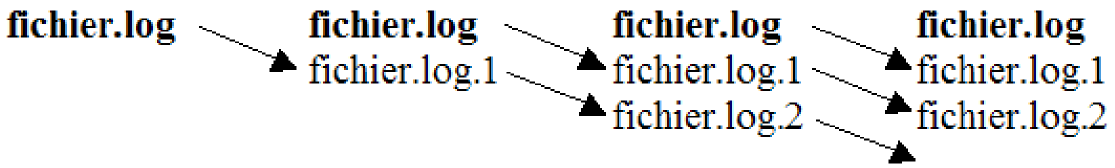
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration debug="false"
  xmlns:log4j="http://jakarta.apache.org/log4j/">
  <appender name="LoggerFile"
    class="org.apache.log4j.DailyRollingFileAppender">
    <param name="File"
      value="c:/monapp.log" />
    <param name="DatePattern" value="'.'yyyy-MM-dd" />
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
        value="%d{yyyy-MM-dd HH:mm:ss.SSS} %-8p [%t]:%C - %m%n" />
    </layout>
  </appender>
  <root>
    <level value="info" />
    <appender-ref ref="LoggerFile" />
  </root>
</log4j:configuration>
```

21.2.4.16. RollingFileAppender

La classe org.apache.log4j.DailyRollingFileAppender envoie les messages dans un fichier à rotation selon sa taille.

Le fichier est créé et rempli avec les différents messages. Une fois que la taille du fichier a atteint celle précisée, le fichier est renommé avec le suffixe .1 et le fichier est recréé. Une fois qu'il est de nouveau rempli, le fichier avec le suffixe .1 est renommé avec .2, le fichier est renommé avec le suffixe .1 et un nouveau fichier est créé.

Si le fichier le plus ancien possède un suffixe supérieur à celui précisé, alors il est supprimé.



Les propriétés d'un RollingFileAppender sont :

Nom	Rôle	Valeur par défaut
ImmediateFlush	Envoyer les messages immédiatement vers le fichier (pas de stockage dans un tampon). Héritée de WriterAppender	true
Threshold	Limiter les messages pris en compte par l'appender à ceux dont le niveau de gravité est supérieur ou égal à celui de threshold. Ceci vient en plus du niveau de gravité associé au logger. Héritée de AppenderSkeleton	
Append	Ajouter le message à la fin du fichier ou remplacer le contenu du fichier. Héritée de FileAppender	True
Encoding	Préciser le jeu de caractères utilisé pour l'encodage. Héritée de WriterAppender	
BufferedIO	Préciser si un tampon doit être utilisé. Héritée de FileAppender	False
BufferSize	Préciser la taille du tampon s'il est utilisé. Héritée de FileAppender	
File	Nom du fichier. Héritée de FileAppender	
MaxFileSize	Taille maximale du fichier avant sa rotation. La taille peut être fournie en KB, MB ou GB Exemple : 1024KB	
MaxIndexBackup	Indiquer le nombre de fichiers de sauvegarde conservés. Une fois ce nombre dépassé, le fichier de sauvegarde le plus ancien est supprimé	

21.2.4.17. ExternalyRolledFileAppender

La classe org.apache.log4j.ExternalyRollingFileAppender envoie les messages dans un fichier à rotation déclenchée par la réception dans une socket de la chaîne de caractères "RollOver" en respectant la casse .

L'appender envoie en retour un accusé de traitement ou d'erreur par la socket.

La classe ExternalyRolledFileAppender possède plusieurs attributs :

Nom	Rôle	Valeur par défaut
ImmediateFlush	Envoyer les messages immédiatement vers le fichier (pas de stockage dans un tampon). Héritée de WriterAppender	true
Threshold	Limiter les messages pris en compte par l'appender à ceux dont le niveau de gravité est supérieur ou égal à celui de threshold. Ceci vient en plus du niveau de gravité associé au logger. Héritée de AppenderSkeleton	
Append	Ajouter le message à la fin du fichier ou remplacer le contenu du fichier. Héritée de FileAppender	True
Encoding	Préciser le jeu de caractères utilisé pour l'encodage. Héritée de WriterAppender	
BufferedIO	Préciser si un tampon doit être utilisé. Héritée de FileAppender	False
BufferSize	Préciser la taille du tampon s'il est utilisé. Héritée de FileAppender	
File	Nom du fichier. Héritée de FileAppender	
MaxFileSize	Taille maximale du fichier avant sa rotation. La taille peut être fournie en KB, MB ou GB	

	Exemple : 1024KB	
MaxIndexBackup	Indiquer le nombre de fichiers de sauvegarde conservés. Une fois ce nombre dépassé, le fichier de sauvegarde le plus ancien est supprimé	
Port	Port d'écoute utilisé par la socket	

21.2.5. Les layouts

Ces composants représentés par la classe org.apache.log4j.Layout permettent de définir le format du message avant son envoi vers ses cibles de destination. Un layout est associé à un Appender lors de son instanciation.

Il existe plusieurs layouts définis par log4j :

- HTMLLayout : formate le message en HTML dans un tableau contenant les colonnes (date/heure, niveau de gravité, thead, logger et message)
- PatternLayout : layout le plus puissant puisqu'il permet de préciser le format du message grâce à un motif
- SimpleLayout : layout le plus simple qui ne contient que le niveau de gravité et le message
- XMLLayout : formate le message en XML

Il est possible de créer ses propres layouts en dérivant de la classe Layout.

21.2.5.1. SimpleLayout

La classe org.apache.log4j.SimpleLayout formate le message de façon basique en incluant :

- le niveau de gravité
- la chaîne de caractères " - "
- le message

Exemple :

```
package fr.jmdoudoux.dej.log4j;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.SimpleLayout;
import org.apache.log4j.FileAppender;

public class TestLog4j8 {
  static Logger logger = Logger.getLogger(TestLog4j8.class);

  public static void main(
      String args[]) {
    SimpleLayout layout = new SimpleLayout();

    FileAppender appender = null;
    try {
      appender = new FileAppender(layout, "c:/monapp.log", false);
    } catch (Exception e) {
      e.printStackTrace();
    }

    logger.addAppender(appender);
    logger.setLevel((Level) Level.DEBUG);

    logger.debug("msg de debogage");
    logger.info("msg d'information");
    logger.warn("msg d'avertissement");
    logger.error("msg d'erreur");
    logger.fatal("msg d'erreur fatale");
  }
}
```

Résultat : le fichier monapp.log

```
DEBUG - msg de debogage
INFO - msg d'information
WARN - msg d'avertissement
ERROR - msg d'erreur
FATAL - msg d'erreur fatale
```

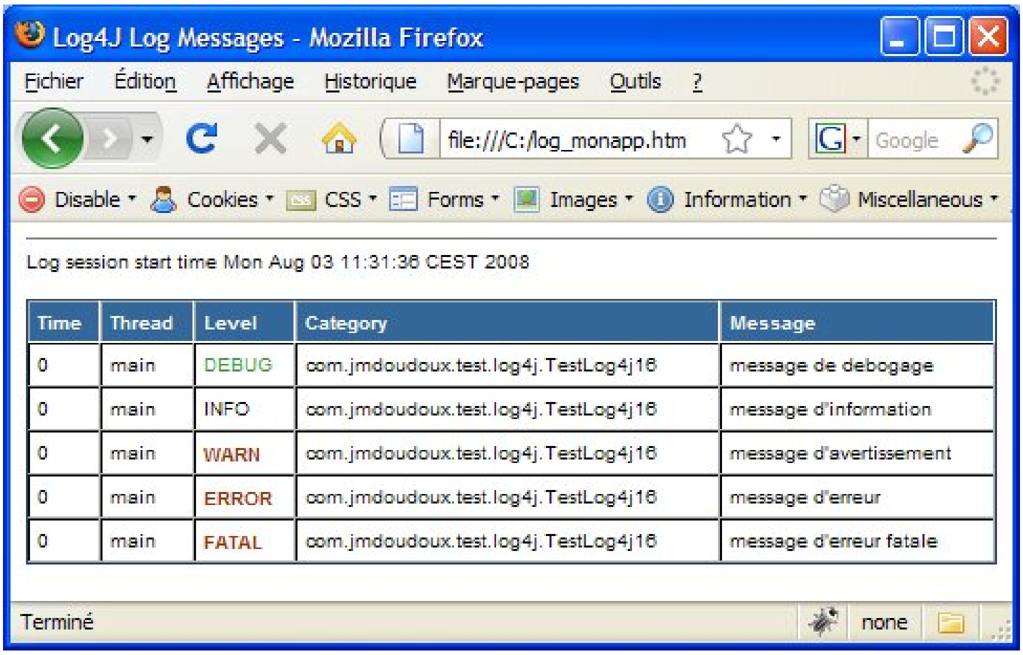
21.2.5.2. HTMLLayout

La classe org.apache.log4j.HTMLLayout formate les messages dans un tableau HTML.

Propriété	Rôle	Valeur par défaut
LocationInfo	Inclure des informations sur la classe	False
Title	Précise le titre de la page web	Log4j Log Messages

Exemple :

```
package fr.jmdoudoux.dej.log4j;

import java.io.*;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.HTMLLayout;
import org.apache.log4j.WriterAppender;

public class TestLog4j16 {
  static Logger logger = Logger.getLogger(TestLog4j16.class);

  public static void main(
      String args[]) {
    HTMLLayout layout = new HTMLLayout();
    WriterAppender appender = null;
    try {
      FileOutputStream output = new FileOutputStream("c:/log_monapp.htm");
      appender = new WriterAppender(layout, output);
    } catch (Exception e) {
      e.printStackTrace();
    }
    logger.addAppender(appender);
    logger.setLevel((Level) Level.DEBUG);

    logger.debug("msg de debogage");
    logger.info("msg d'information");
    logger.warn("msg d'avertissement");
    logger.error("msg d'erreur");
    logger.fatal("msg d'erreur fatale");
  }
}
```


Log4J Log Messages - Mozilla Firefox
Fichier Édition Affichage Historique Marque-pages Outils ?
file:///C:/log_monapp.htm
Google
Disable Cookies CSS Forms Images Information Miscellaneous
Terminé
none

Log session start time Mon Aug 03 11:31:36 CEST 2008

Time	Thread	Level	Category	Message
0	main	DEBUG	com.jmdoudoux.test.log4j.TestLog4j16	message de debogage
0	main	INFO	com.jmdoudoux.test.log4j.TestLog4j16	message d'information
0	main	WARN	com.jmdoudoux.test.log4j.TestLog4j16	message d'avertissement
0	main	ERROR	com.jmdoudoux.test.log4j.TestLog4j16	message d'erreur
0	main	FATAL	com.jmdoudoux.test.log4j.TestLog4j16	message d'erreur fatale

21.2.5.3. XMLLayout

La classe org.apache.log4j.XMLLayout formate les messages en XML.

Exemple :

```
<log4j:event logger="fr.jmdoudoux.dej.log4j.TestLog4j10"
          timestamp="1219683683344" level="DEBUG" thread="main">
<log4j:message><![CDATA[mon message]]></log4j:message>
</log4j:event>
```

21.2.5.4. PatternLayout

Le PatternLayout permet de préciser le format du message grâce à un motif dont certaines séquences seront dynamiquement remplacées par leurs valeurs correspondantes à l'exécution.

Les séquences commencent par un caractère % suivi d'une lettre :

Motif	Rôle
%c	Le nom de la catégorie ou du logger qui a émis le message
%C	Le nom de la classe qui a émis le message : l'utilisation de ce motif est coûteuse en ressources
%d	Le timestamp de l'émission du message. Il est possible de fournir un format pour la date/heure en utilisant les motifs de la classe SimpleDateFormat. Exemple : %d{dd MMM yyyy HH:MM:ss } Pour améliorer les performances, il est possible d'utiliser des formateurs de dates en précisant ABSOLUTE, RELATIVE ou ISO8601 Exemple : %d{ABSOLUTE} Sans format précisé, c'est le format défini dans la norme ISO8601 qui est utilisé.
%m	Le message
%n	Un saut de ligne dépendant de la plate-forme

%p	Le niveau de gravité du message
%r	Le nombre de millisecondes écoulées entre le lancement de l'application et l'émission du message
%t	Le nom du thread
%x	NDC (Nested Diagnostic Context) du thread. Ceci est particulièrement utile pour les applications de type web.
%%	Le caractère %
%L	Le numéro de ligne dans le code émettant le message : l'utilisation de ce motif est coûteuse en ressources
%F	Le nom du fichier émettant le message : l'utilisation de ce motif est coûteuse en ressources
%M	Le nom de la méthode émettant le message : l'utilisation de ce motif est coûteuse en ressources
%l	Des informations sur l'origine du message dans le code source (C'est un raccourci dépendant de la JVM qui correspond généralement à %C.%M(%F:%L)): l'utilisation de ce motif est coûteuse en ressources

Il est possible de préciser le formatage de chaque motif grâce à un alignement et/ou une troncature. Dans le tableau ci-dessous, le caractère # représente une des lettres du tableau ci-dessus, n représente un nombre de caractères.

Motif	Rôle
%#	Aucun formatage (par défaut)
%n#	Alignement à droite, des blancs sont ajoutés si la taille du motif est inférieure à n caractères
%-n#	Alignement à gauche, des blancs sont ajoutés si la taille du motif est inférieure à n caractères
%.n	Tronque le motif s'il est supérieur à n caractères
%-n.n#	Alignement à gauche, taille du motif obligatoirement de n caractères (troncature ou complément avec des blancs)

Le motif par défaut du PatternLayout est %m%n.

Le motif permet donc une grande souplesse dans le formatage du message.

Exemple :

```
package fr.jmdoudoux.dej.log4j;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.PatternLayout;
import org.apache.log4j.ConsoleAppender;

public class TestLog4j15 {
  static Logger logger = Logger.getLogger(TestLog4j15.class);

  public static void main(String args[]) {

    StringBuilder motif = new StringBuilder();
    motif.append("Date/heure : %d{yyyy-MM-dd HH:mm:ss.SSS} %n");
    motif.append("Classe emettrice : %C %n");
    motif.append("Localisation : %l %n");
    motif.append("Message: %m %n");
    motif.append("%n");

    PatternLayout layout = new PatternLayout(motif.toString());
    ConsoleAppender appender = new ConsoleAppender(layout);

    logger.addAppender(appender);
    logger.setLevel((Level) Level.DEBUG);

    logger.debug("msg de debogage");
    logger.info("msg d'information");
    logger.warn("msg d'avertissement");
    logger.error("msg d'erreur");
    logger.fatal("msg d'erreur fatale");
  }
}
```

Résultat :

```
Date/heure : 2008-08-03 11:26:13.705
Classe emettrice : fr.jmdoudoux.dej.log4j.TestLog4j15
Localisation : fr.jmdoudoux.dej.log4j.TestLog4j15.main(TestLog4j15.java:26)
Message: msg de debogage

Date/heure : 2008-08-03 11:26:13.705
Classe emettrice : fr.jmdoudoux.dej.log4j.TestLog4j15
Localisation : fr.jmdoudoux.dej.log4j.TestLog4j15.main(TestLog4j15.java:27)
Message: msg d'information

Date/heure : 2008-08-03 11:26:13.705
Classe emettrice : fr.jmdoudoux.dej.log4j.TestLog4j15
Localisation : fr.jmdoudoux.dej.log4j.TestLog4j15.main(TestLog4j15.java:28)
Message: msg d'avertissement

Date/heure : 2008-08-03 11:26:13.705
Classe emettrice : fr.jmdoudoux.dej.log4j.TestLog4j15
Localisation : fr.jmdoudoux.dej.log4j.TestLog4j15.main(TestLog4j15.java:29)
Message: msg d'erreur

Date/heure : 2008-08-03 11:26:13.705
Classe emettrice : fr.jmdoudoux.dej.log4j.TestLog4j15
Localisation : fr.jmdoudoux.dej.log4j.TestLog4j15.main(TestLog4j15.java:30)
Message: msg d'erreur fatale
```

Voici un exemple de configuration dans un fichier de configuration XML.

Exemple :

```
...
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
        value="%d{yyyy-MM-dd HH:mm:ss.SSS} %-8p [%t]:%C - %m%n" />
    </layout>
...
```

Résultat :

```
2008-08-03 09:42:19.342 DEBUG  [main]:fr.jmdoudoux.dej.log4j.TestLog4j1 - msg de debogage
2008-08-03 09:42:19.342 INFO  [main]:fr.jmdoudoux.dej.log4j.TestLog4j1 - msg d'information
2008-08-03 09:42:19.342 WARN  [main]:fr.jmdoudoux.dej.log4j.TestLog4j1 - msg d'avertissement
2008-08-03 09:42:19.342 ERROR [main]:fr.jmdoudoux.dej.log4j.TestLog4j1 - msg d'erreur
2008-08-03 09:42:19.342 FATAL [main]:fr.jmdoudoux.dej.log4j.TestLog4j1 - msg d'erreur fatale
```

21.2.6. L'externalisation de la configuration

Log4j peut être entièrement configuré directement dans le code de l'application.

Attention : dans ce cas, la configuration d'un appender nécessite généralement l'appel à la méthode activateOptions() de l'instance de l'Appender pour qu'elle soit prise en compte.

Cependant Log4j est généralement mis en oeuvre avec une externalisation de sa configuration pour que ses paramètres ne soient pas codés en dur. Les paramètres sont alors modifiables sans avoir à recompiler le code de l'application ce qui offre plus de souplesse dans l'utilisation de Log4j.

La configuration de Log4j commence par la définition du ou des appenders qui seront utilisés. Il faut ensuite définir le ou les loggers en leur associant au besoin un ou plusieurs appenders. Pour que Log4j fonctionne, il faut au minimum associer un appender au logger racine.

Log4j propose deux formats pour externaliser sa configuration :

- fichier properties : les informations sont fournies sous la forme de paires clé=valeur. Le nom de ce fichier est par défaut log4j.properties
- fichier XML : les informations sont fournies dans un document XML. Le nom de ce fichier est par défaut log4j.xml

Le format XML est plus verbeux mais il est mieux structuré. De plus, certaines fonctionnalités ne sont configurables que dans ce format. C'est donc le format dont l'utilisation est recommandée.

Dans les fichiers de configuration, la valeur d'une propriété peut être initialisée avec une variable d'environnement de la JVM en utilisant la syntaxe ${nom_propriété}

21.2.6.1. Les principes généraux

L'initialisation de Log4j n'a besoin d'être réalisée qu'une seule fois de préférence au lancement de l'application.

La configuration suit la même logique que celle des loggers : il est inutile de définir tous les loggers puisque le principe d'héritage permet automatiquement à un logger d'obtenir les caractéristiques de son ascendant le plus proche pour lequel une configuration particulière a été précisée.

Exemple :

Logger	Niveau de gravité	Affectation par
rootLogger	error	assignation (debug par défaut)
com	error	héritage
fr.jmdoudoux	error	héritage
fr.jmdoudoux.dej	info	assignation
fr.jmdoudoux.dej.log4j	info	héritage
fr.jmdoudoux.dej.log4j.MaClasse	debug	assignation

Ceci peut permettre de très finement régler le niveau de gravité des différents éléments qui composent une application que ce soit dans les classes de l'application ou d'une bibliothèque tierce.

La configuration au niveau des appenders utilisés suit aussi une logique hiérarchique qui n'est pas de l'héritage mais une additivité. Un appender défini dans un logger s'ajoute à ou aux appenders déjà définis dans les loggers de la hiérarchie mère.

21.2.6.1.1. Le mécanisme de recherche de la configuration

Log4j propose par défaut un mécanisme de recherche de sa configuration. Log4j recherche un fichier de configuration dans le classpath car il utilise un classLoader pour cette tâche.

Ce mécanisme de recherche peut être désactivé en positionnant à true la propriété système log4j.defaultInitOverride. Ceci doit être utilisé si le chargement de la configuration est fait manuellement dans l'application.

La propriété système log4j.configuration peut être utilisée pour préciser le nom du fichier de configuration.

Par défaut, Log4j recherche dans le classpath un fichier nommé log4j.xml. Si ce fichier n'est pas trouvé, Log4j recherche un fichier nommé log4j.properties.

Log4j utilise un objet de type org.apache.log4j.spi.Configurator pour charger la configuration.

La propriété log4j.configuratorClass permet de préciser explicitement la classe à utiliser pour charger la configuration.

Par défaut, Log4j utilise un objet de type DomConfigurator pour charger un fichier au format XML sinon c'est un objet de type ProperyConfigurator qui est utilisé pour charger le fichier properties.

Vu le mécanisme par défaut proposé par Log4j, le plus simple est donc de nommer son fichier de configuration log4j.xml ou log4j.properties selon le format de configuration utilisé et de mettre le fichier dans le classpath.

21.2.6.2. Le chargement explicite d'une configuration

Si le mode de fonctionnement par défaut ne répond pas aux besoins, il est possible de demander explicitement le chargement d'une configuration.

Pour effectuer ce chargement, l'API fournit plusieurs classes qui implémentent l'interface Configurator. La classe BasicConfigurator est la classe mère des classes PropertyConfigurator (pour la configuration par un fichier de propriétés) et DOMConfigurator (pour la configuration par un fichier XML).

21.2.6.2.1. La classe BasicConfigurator

La classe org.apache.log4j.BasicConfigurator permet de créer une configuration basique.

Avant la version 1.2 de Log4j, la classe BasicConfigurator permettait de configurer la catégorie root avec des valeurs par défaut. L'appel à la méthode configure() ajoutait à la catégorie root la priorité DEBUG et un ConsoleAppender vers la sortie standard (System.out) associé à un PatternLayout (TTCC_CONVERSION_PATTERN qui est une constante définie dans la classe PatternLayout).

Exemple :

```
import org.apache.log4j.*;

public class TestBasicConfigurator {
  static Category cat = Category.getInstance(TestBasicConfigurator.class.getName()) ;

  public static void main(String[] args) {
    BasicConfigurator.configure();
    cat.info("Mon message");
  }
}
```

Résultat :

```
0 [main] INFO TestBasicConfigurator  - Mon message
```

A partir de la version 1.2 de Log4j, la méthode configure() instancie une configuration où le rootLogger utilise un appender de type ConsoleAppender et un motif PatternLayout.TTCC_CONVERSION_PATTERN pour cet appender. Le niveau de gravité associé est DEBUG par défaut.

Exemple avec Log4j 1.2 :

Exemple :

```
package fr.jmdoudoux.dej.log4j;

import org.apache.log4j.Logger;
import org.apache.log4j.BasicConfigurator;

public class TestLog4j14 {
  static Logger logger = Logger.getLogger(TestLog4j14.class);

  public static void main(
      String[] args) {
```

```
    BasicConfigurator.configure();

    logger.info("debut");
    System.out.println("traitement");
    logger.debug("maValeur");
    logger.info("fin");
  }
}
```

Résultat :

```
0 [main] INFO
fr.jmdoudoux.dej.log4j.TestLog4j14  -
debut
traitement
0 [main] DEBUG
fr.jmdoudoux.dej.log4j.TestLog4j14  -
maValeur
0 [main] INFO
fr.jmdoudoux.dej.log4j.TestLog4j14  -
fin
```

21.2.6.2.2. La classe PropertyConfigurator

La classe org.apache.log4j.PorpertyConfigurator lit un fichier de configuration au format properties et instancie la configuration correspondante.

La classe PropertyConfigurator permet de configurer Log4j à partir d'un fichier de propriétés ce qui évite la recompilation de classes pour modifier la configuration. La méthode configure() qui attend en paramètre un nom de fichier permet de charger la configuration.

Exemple :

```
import org.apache.log4j.*;

public class TestLogging6 {
   static Category cat = Category.getInstance(TestLogging6.class.getName());

   public static void main(String[] args) {
      PropertyConfigurator.configure("logging6.properties");
      cat.info("Mon message");
   }
}
```

Exemple : le fichier loggin6.properties de configuration de Log4j

```
# Affecte a la catégorie root la priorité DEBUG et un appender nommé CONSOLE_APP
log4j.rootCategory=DEBUG, CONSOLE_APP
# l'appender CONSOL_APP est associé à la console
log4j.appender.CONSOLE_APP=org.apache.log4j.ConsoleAppender
# CONSOLE_APP utilise un PatternLayout qui affiche : le nom du thread, la priorité,
# le nom de la catégorie et le message
log4j.appender.CONSOLE_APP.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE_APP.layout.ConversionPattern= [%t] %p %c - %m%n
```

Résultat :

```
C:\>java TestLogging6
[main] INFO TestLogging6 - Mon message
```

21.2.6.2.3. La classe DOMConfigurator

La classe DOMConfigurator permet de configurer Log4j à partir d'un fichier XML ce qui évite aussi la recompilation de classes pour modifier la configuration. La méthode configure() qui attend un nom de fichier permet de charger la configuration. Cette méthode nécessite un parser XML de type DOM compatible avec l'API JAXP.

Le fichier de configuration au format XML doit respecter la DTD log4j.dtd fournie dans la bibliothèque Log4j.

Exemple :

```
import org.apache.log4j.*;
import org.apache.log4j.xml.*;

public class TestLogging7 {
  static Category cat = Category.getInstance(TestLogging7.class.getName());

  public static void main(String[] args) {
    try {
      DOMConfigurator.configure("logging7.xml");
    } catch (Exception e) {
      e.printStackTrace();
    }
    cat.info("Mon message");
  }
}
```

Exemple : le fichier loggin7.xml de configuration de log4j

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
  <appender name="CONSOLE_APP" class="org.apache.log4j.ConsoleAppender">
     <layout class="org.apache.log4j.PatternLayout">
     <param name="ConversionPattern" value="[%t] %p %c - %m%n"/>
         </layout>
  </appender>
  <root>
    <priority value ="DEBUG" />
    <appender-ref ref="CONSOLE_APP" />
  </root>
</log4j:configuration>
```

Résultat :

```
C:\j2sdk1.4.0-rc\bin>java TestLogging7
[main] INFO TestLogging7 - Mon message
```

21.2.6.3. Les formats des fichiers de configuration

Le fichier de configuration permet :

- de définir les caractéristiques des différents loggers (ou categories)
- de définir les appenders et leur associer un layout
- d'associer un ou plusieurs appenders à un ou aux loggers (ou categories)

Il est préférable d'utiliser un fichier de configuration plutôt que de configurer les entités de Log4j dans le code source car cette dernière solution implique une modification du code et une recompilation pour être prise en compte.

Le fichier de configuration permet basiquement de définir le niveau de gravité minimum à traiter, les flux de sorties (Appender) et le format des messages (Layout).

Deux formats de fichiers de configuration sont proposés par Log4j :

- fichier properties : fichier texte dans lequel la configuration est fournie sous la forme de paires clé=valeur
- fichier xml : la configuration est fournie dans un document xml

L'ordre de déclaration des loggers dans le fichier de configuration n'est pas imposé mais il est préférable de conserver un ordre hiérarchique pour en faciliter la lecture et la compréhension.

Généralement le fichier de configuration est lu et utilisé au lancement de l'application.

Chaque appender possède ses propres propriétés de configuration.

Celles-ci sont définies de façons différentes selon le format du fichier de configuration :

- Dans un fichier properties : log4j.appender.nom_appender.nom_propriété=valeur
- Dans un fichier xml : en utilisant le tag fils <param> du tag <appender>

21.2.6.4. La configuration par fichier properties

La configuration utilisant un fichier properties est historiquement la plus ancienne : de nombreuses applications utilisant Log4j la mettent encore en oeuvre.

Comme pour tous fichiers properties, les lignes qui commencent par un caractère dièse sont des lignes de commentaires et sont donc ignorées.

Plusieurs options de configuration générale peuvent être définies dans le fichier de configuration :

Options	Description
log4j.debug	Booléen qui précise si Log4j doit fournir des informations de debogage sur ses activités de chargement du fichier de configuration. La valeur par défaut est false.
log4j.disable	Précise le niveau de gravité minimum des messages pour être traités par tous les loggers/categorys. Remarque : l'option log4j.disableOverrider doit obligatoirement être positionnée à false.
log4j.disableOverride	Doit être positionnée à true pour utiliser l'option log4j.disable. La valeur par défaut est false.

La clé log4j.threshold permet de préciser un niveau minimum de gravité pour tous les loggers ou category définis indépendamment du niveau spécifié pour chacun d'eux.

Remarque : l'ordre de déclaration des clés dans le fichier n'a pas d'importance pour la bonne mise en oeuvre de Log4j mais il est cependant recommandé d'utiliser un ordre logique pour faciliter la compréhension du paramétrage.

Une category est définie en utilisant une clé de la forme :

log4j.category.nom_category

Un logger est défini en utilisant une clé de la forme :

log4j.logger.nom_logger

La category racine est configurée en utilisant une clé de la forme :

log4j.categoryLogger

Le logger racine est configuré en utilisant une clé de la forme :

log4j.rootLogger

La valeur de ces clés est de la forme niveau_gravité, nom_appender1, nom_appender2, ...

Exemple :

```
# le niveau de gravité debug est associe à la catégorie racine avec deux
# appenders nommés A1 et A2
log4j.rootCategory=DEBUG, A1, A2
```

Exemple :

```
# le niveau de gravité debug est associe au logger racine avec deux
# appenders nommés A1 et A2
log4j.rootLogger=DEBUG, A1, A2
```

Remarque : chaque appender doit avoir un nom unique

Le niveau de gravité est optionnel mais dans le cas où il n'est pas fourni, il est impératif de laisser la virgule entre le signe = et le nom du premier appender.

Exemple :

```
# le niveau de gravité debug (par défaut) est associe au logger racine
# avec un appender nommé A1
   log4j.rootLogger=, A1
```

Un appender est défini en utilisant une clé de la forme :

log4j.appender.nom_appender

La valeur de cette clé est le nom pleinement qualifié de la classe qui encapsule l'appender.

Exemple :

```
      # l'appender nommé A1 est de type ConsoleAppender
      log4j.appender.A1=org.apache.log4j.ConsoleAppender
      # l'appender nommé A2 est de type RollingFileAppender
      log4j.appender.A2=org.apache.log4j.RollingFileAppender
```

Le layout d'un appender est précisé en utilisant une clé de la forme :

log4j.appender.nom_appender.layout

La valeur de cette clé est le nom pleinement qualifié de la classe qui encapsule le layout.

Exemple :

```
   log4j.appender.A1.layout=org.apache.log4j.PatternLayout
```

Une propriété d'un layout est précisée en utilisant une clé de la forme :

log4j.appender.nom_appender.layout.nom_propriété

La valeur sera fournie à la propriété correspondante par introspection.

Exemple :

```
   log4j.appender.A1.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} [%-5p] %c- %m%n
   log4j.appender.A2.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} [%-5p] %c- %m%n
   log4j.appender.A2.File=monapp.log
   log4j.appender.A2.MaxFileSize=1024KB
   log4j.appender.A2.MaxBackupIndex=2
```

Pour modifier le niveau de gravité pris en compte par un logger il faut utiliser une clé de la forme log4j.logger.nom_logger. La valeur de cette clé doit être le niveau de gravité minimum qui sera traité par le logger.

Exemple :

```
   log4j.logger.fr.jmdoudoux.dej=INFO
```

Il est possible de supprimer l'additivité des appenders d'un logger en utilisant une clé de la forme log4j.additivity.nom_logger. La valeur est un booléen qui précise l'additivité des appenders (la valeur par défaut est true, il faut mettre false pour la supprimer).

Exemple :

```
   log4j.additivity.fr.jmdoudoux.dej=false
```

Il est possible de fournir comme valeur d'une clé la valeur d'une propriété système définie dans la JVM. Pour obtenir la valeur d'une de ces propriétés, il suffit d'utiliser la syntaxe ${nom_de_la_propriete}

21.2.6.5. La configuration par un fichier XML

La structure des données contenues dans le fichier XML est organisée en plusieurs parties définies dans la DTD log4j.dtd et comprend :

- la définition et la configuration des appenders
- la définition et la configuration des loggers
- la configuration du logger racine

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration debug="false"
  xmlns:log4j="http://jakarta.apache.org/log4j/">
  <appender name="console" class="org.apache.log4j.ConsoleAppender">
    <param name="Target" value="System.out" />
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
        value="%d{yyyy-MM-dd HH:mm:ss.SSS} %-8p [%t]:%C - %m%n" />
    </layout>
  </appender>
  <root>
    <appender-ref ref="console" />
  </root>
</log4j:configuration>
```

Si le fichier n'est pas valide alors une exception est levée durant sa lecture et son traitement.

Exemple :

```
log4j:WARN Fatal parsing error 12 and column 5
log4j:WARN The element type "param" must be terminated by the matching end-tag "</param>".
log4j:ERROR Could not parse url
[file:/C:/Documents%20and%20Settings/jmd/workspace/TestLog4j/bin/log4j.xml].
org.xml.sax.SAXParseException: The element type "param" must be terminated
by the matching end-tag "</param>".
```

Les différents éléments qui composent le fichier de configuration sont détaillés dans les sections suivantes.

21.2.6.5.1. Le format du fichier de configuration XML

Le fichier de configuration commence par un prologue et une déclaration de la DTD.

La structure du document xml qui va contenir la configuration de Log4j est définie dans la DTD log4j.dtd.

Cette DTD est fournie dans la bibliothèque log4j.jar dans le package org.apache.log4j.xml

Exemple :

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
```

La DTD définit l'élément racine comme suit :

Exemple :

```
<!ELEMENT log4j:configuration (renderer*, appender*,plugin*, (category|logger)*,root?,
                               (categoryFactory|loggerFactory)?)>
```

L'élément racine est le tag <configuration> associé à l'espace de nommage log4j.

Exemple :

```
<log4j:configuration xmlns:log4j='http://jakarta.apache.org/log4j/'>
</log4j:configuration>
```

Le tag <configuration> peut avoir

- 0 ou plusieurs tags fils <renderer>
- 0 ou plusieurs tags fils <appender>
- 0 ou plusieurs tags fils <plugin>
- 0 ou plusieurs tags fils <category> ou <logger>
- 0 ou 1 tag fils <root>
- 0 ou 1 tag fils <categoryFactory> ou <loggerFactory>

La définition des éléments doit impérativement respecter cet ordre défini dans la DTD.

La DTD définit trois attributs pour le tag <configuration>.

Exemple :

```
<!ATTLIST log4j:configuration
  xmlns:log4j             CDATA #FIXED "http://jakarta.apache.org/log4j/"
  threshold               (all|trace|debug|info|warn|error|fatal|off|null) "null"
  debug                   (true|false|null)  "null"
  reset                   (true|false) "false">
```

Le tag racine est le tag <configuration> qui possède trois attributs :

- threshold : précise le niveau de gravité minimum pour qu'un message soit pris en compte par un logger indépendamment du niveau de gravité associé à ce logger
- debug : la valeur true permet de demander à Log4j de fournir des informations de débogage sur son exécution. La valeur par défaut "null" permet de demander l'utilisation de la valeur interne de Log4j
- reset :

L'attribut debug est particulièrement utile pour comprendre l'utilisation du fichier de configuration et résoudre d'éventuel problème dans son contenu.

21.2.6.5.2. La configuration d'un appender

La configuration d'un appender se fait en utilisant un tag <appender>.

La DTD définit l'élément appender comme suit :

Exemple :

```
<!ELEMENT appender (errorHandler?, param*,
      rollingPolicy?, triggeringPolicy?, connectionSource?,
      layout?, filter*, appender-ref*)>
<!ATTLIST appender
  name         CDATA   #REQUIRED
  class        CDATA   #REQUIRED
>
```

Le tag <appender> possède deux attributs obligatoires :

- name : nom unique dans la configuration de l'appender permettant d'y faire référence
- class : nom pleinement qualifié de la classe qui encapsule l'appender

Le tag fils facultatif <param> permet de fournir un paramètre à l'appender. Chaque appender possède ses propres paramètres. Le tag <param> permet de fournir des valeurs aux propriétés de l'appender dont le nom correspond à l'attribut name et la valeur à l'attribut value.

Exemple :

```
<appender name="console" class="org.apache.log4j.ConsoleAppender">
  <param name="Target" value="System.out"/>
</appender>
```

Le tag facultatif layout permet de préciser le layout associé à l'appender. Le tag <layout> possède l'attribut obligatoire class qui précise le nom pleinement qualifié de la classe qui encapsule le layout.

Exemple :

```
<appender name="console" class="org.apache.log4j.ConsoleAppender">
  <layout class="org.apache.log4j.SimpleLayout"/>
</appender>
```

Des paramètres peuvent aussi être fournis au layout en utilisant un ou plusieurs tags fils <param>. Chaque layout possède ses propres propriétés.

Exemple :

```
<appender name="console" class="org.apache.log4j.ConsoleAppender">
  <layout class="org.apache.log4j.PatternLayout">
    <param name="ConversionPattern" value="=%d{yyyy-MM-dd HH:mm:ss} [%-5p] %c- %m%n" />
  </layout>
</appender>
```

21.2.6.5.3. La configuration des Loggers

La configuration d'un logger se fait en utilisant un tag <logger>.

La DTD définit l'élément logger comme suit :

Exemple :

```
<!ELEMENT logger (level?,appender-ref*)>
<!ATTLIST logger
  name         CDATA   #REQUIRED
  additivity   (true|false) "true"
>
```

Le tag <logger> possède un attribut obligatoire :

- name : précise le nom du logger, généralement correspondant à un nom de package ou de classe pleinement qualifié

Le tag <logger> possède un attribut facultatif :

- additivity : précise si l'additivité des appenders doit être poursuivie ou non. La valeur par défaut est true

Le tag logger peut avoir deux types de tag fils : <level> et <appender-ref>

Le tag facultatif level permet de préciser le niveau de gravité associé au logger. L'attribut value permet de préciser ce niveau de gravité.

Exemple :

```
<logger name="fr.jmdoudoux.dej.monapp">
  <level value="info"/>
</logger>
```

Le tag <appender-ref> permet d'associer un nouvel appender au logger en plus de ceux associés par additivité des loggers de hiérarchie supérieure. L'attribut ref permet de préciser le nom de l'appender concerné. La valeur de cet attribut doit correspondre à une valeur d'un attribut name d'un appender défini dans la configuration.

Un tag <logger> peut avoir aucun, un ou plusieurs tags <appender-ref> puisqu'un logger peut avoir plusieurs appenders.

Exemple :

```
<logger name="fr.jmdoudoux.dej.monapp">
  <appender-ref ref="console" />
  <appender-ref ref="journal" />
</logger>
```

21.2.6.5.4. La configuration du logger racine

La configuration du logger racine se fait en utilisant un tag <root>.

La DTD définit l'élément logger comme suit :

Exemple :

```
<!ELEMENT root (param*, (priority|level)?, appender-ref*)>
```

Sa configuration est similaire à celle des loggers sauf que le tag <root> ne possède pas d'attribut.

Exemple :

```
<root>
  <priority value ="info" />
  <appender-ref ref="console"/>
</root>
```

21.2.6.5.5. Les seuils et les filtres pour les appenders

La propriété threshold permet de définir un seuil minimum de niveau de gravité des messages traités par l'appender.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration debug="false"
  xmlns:log4j="http://jakarta.apache.org/log4j/">
  <appender name="console" class="org.apache.log4j.ConsoleAppender">
```

```
    <param name="threshold" value="ERROR" />
    <param name="Target" value="System.out" />
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
        value="%d{yyyy-MM-dd HH:mm:ss.SSS} %-8p [%t]:%C - %m%n" />
    </layout>
  </appender>
  <root>
    <appender-ref ref="console" />
  </root>
</log4j:configuration>
```

Résultat :

```
2008-08-03 10:03:11.895 ERROR [main]:fr.jmdoudoux.dej.log4j.TestLog4j1 - msg d'erreur
2008-08-03 10:03:11.910 FATAL [main]:fr.jmdoudoux.dej.log4j.TestLog4j1 - msg d'erreur fatale
```

Log4j propose un autre mécanisme plus puissant pour filtrer les messages traités par un appender : les filtres.

Log4j propose plusieurs filtres en standard :

- LevelMatchFilter : filtre uniquement les messages ayant un niveau de gravité particulier
- LevelRangeFilter : filtre pour les messages ayant un niveau de gravité compris entre un niveau minimum et un niveau maximum
- DenyAllFilter : filtre pour refuser tous les messages

Les filtres ne peuvent être utilisés que dans une configuration par un fichier xml.

Le filtre LevelMatchFilter possède plusieurs paramètres :

Paramètres	Rôle
levelToMatch	Précise le niveau de gravité du message pour qu'il soit traité
acceptOnMatch	Booléen qui indique si le message est traité (true) ou rejeté (false) par le filtre

Le filtre LevelMatchFilter traite les messages qui correspondent au filtre mais laisse passer ceux qui ne correspondent pas. Ainsi pour ignorer ces messages, il est nécessaire d'appliquer en plus un filtre de type DenyAllFilter.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration debug="false"
  xmlns:log4j="http://jakarta.apache.org/log4j/">
  <appender name="console" class="org.apache.log4j.ConsoleAppender">
    <param name="Target" value="System.out" />
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
        value="%d{yyyy-MM-dd HH:mm:ss.SSS} %-8p [%t]:%C - %m%n" />
    </layout>
    <filter class="org.apache.log4j.varia.LevelMatchFilter">
      <param name="levelToMatch" value="ERROR" />
    </filter>
    <filter class="org.apache.log4j.varia.DenyAllFilter"/>
  </appender>
  <root>
    <appender-ref ref="console" />
  </root>
</log4j:configuration>
```

Résultat :

```
2008-08-03 10:14:49.203 ERROR    [main]:fr.jmdoudoux.dej.log4j.TestLog4j1 - msg d'erreur
```

Si le filtre DenyAllFilter n'est pas utilisé alors tous les messages sont traités.

Résultat :

```
2008-08-03 10:19:28.612 DEBUG [main]:fr.jmdoudoux.dej.log4j.TestLog4j1 - msg de debogage
2008-08-03 10:19:28.612 INFO  [main]:fr.jmdoudoux.dej.log4j.TestLog4j1 - msg d'information
2008-08-03 10:19:28.612 WARN  [main]:fr.jmdoudoux.dej.log4j.TestLog4j1 - msg d'avertissement
2008-08-03 10:19:28.612 ERROR [main]:fr.jmdoudoux.dej.log4j.TestLog4j1 - msg d'erreur
2008-08-03 10:19:28.612 FATAL [main]:fr.jmdoudoux.dej.log4j.TestLog4j1 - msg d'erreur fatale
```

Le filtre LevelRangeFilter possède plusieurs paramètres :

Paramètres	Rôle
levelMin	Précise le niveau de gravité minimal du message pour qu'il soit traité
levelMax	Précise le niveau de gravité maximal du message pour qu'il soit traité
acceptOnMatch	true : le message est traité sans appliquer les autres filtres false : si le niveau de gravité est hors de la plage, alors le message est ignoré sinon le message est soumis aux autres filtres

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration debug="false"
  xmlns:log4j="http://jakarta.apache.org/log4j/">
  <appender name="console" class="org.apache.log4j.ConsoleAppender">
    <param name="Target" value="System.out" />
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
        value="%d{yyyy-MM-dd HH:mm:ss.SSS} %-8p [%t]:%C - %m%n" />
    </layout>
    <filter class="org.apache.log4j.varia.LevelRangeFilter">
      <param name="levelMin" value="INFO" />
      <param name="levelMax" value="ERROR" />
    </filter>
  </appender>
  <root>
    <appender-ref ref="console" />
  </root>
</log4j:configuration>
```

Résultat :

```
2008-08-03 10:44:46.636 INFO  [main]:fr.jmdoudoux.dej.log4j.TestLog4j1 - msg d'information
2008-08-03 10:44:46.636 WARN  [main]:fr.jmdoudoux.dej.log4j.TestLog4j1 - msg d'avertissement
2008-08-03 10:44:46.636 ERROR [main]:fr.jmdoudoux.dej.log4j.TestLog4j1 - msg d'erreur
```

Avec les filtres, il est par exemple possible de définir un appender qui traite les messages de debogage et un appender qui traite les autres messages.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd" >
<log4j:configuration>
  <appender name="fichierLog"
    class="org.apache.log4j.RollingFileAppender">
    <param name="maxFileSize" value="1024KB" />
    <param name="maxBackupIndex" value="2" />
    <param name="File" value="c:/monapp.log" />
    <param name="threshold" value="info" />
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
```

```
        value="%d{yyyy-MM-dd HH:mm:ss.SSS} %-8p [%t]:%C - %m%n" />
    </layout>
  </appender>

  <appender name="fichierDebug"
    class="org.apache.log4j.RollingFileAppender">
    <param name="maxFileSize" value="1024KB" />
    <param name="maxBackupIndex" value="2" />
    <param name="File" value="c:/monapp_debug.log" />
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
        value="%d{yyyy-MM-dd HH:mm:ss.SSS} %-8p [%t]:%C - %m%n" />
    </layout>
    <filter class="org.apache.log4j.varia.LevelMatchFilter">
      <param name="levelToMatch" value="DEBUG" />
    </filter>
    <filter class="org.apache.log4j.varia.DenyAllFilter"/>
  </appender>

  <root>
    <priority value="debug"></priority>
    <appender-ref ref="fichierLog" />
    <appender-ref ref="fichierDebug" />
  </root>
</log4j:configuration>
```

21.2.6.6. log4j.xml versus log4j.properties

La configuration par fichier properties est moins verbeuse que par fichier XML.

Certaines fonctionnalités ne sont pas supportées en utilisant la configuration par properties comme l'utilisation des Filters ou des ErrorHandlers. Certains appenders ne sont configurables que par fichier XML.

21.2.6.7. La conversion du format properties en format XML

La conversion d'un fichier de configuration au format properties en un fichier de configuration au format XML doit se faire manuellement.

Voici un premier exemple simple.

Exemple :

```
log4j.rootLogger=info, console
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.Target=System.out
log4j.appender.console.layout=org.apache.log4j.SimpleLayout
```

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd" >
<log4j:configuration>
  <appender name="console" class="org.apache.log4j.ConsoleAppender">
    <layout class="org.apache.log4j.SimpleLayout" />
  </appender>
  <root>
    <priority value="info" />
    <appender-ref ref="console" />
  </root>
</log4j:configuration>
```

Le second exemple ci-dessous utilise deux appenders.

Exemple :

```
log4j.rootLogger=debug, console, fichier

log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss.SSS} %-8p [%t]:%C - %m%n

log4j.appender.fichier=org.apache.log4j.RollingFileAppender
log4j.appender.fichier.File=c:/monapp.log
log4j.appender.fichier.MaxFileSize=1024KB
log4j.appender.fichier.MaxBackupIndex=2
log4j.appender.fichier.layout=org.apache.log4j.PatternLayout
log4j.appender.fichier.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss.SSS} %-8p [%t]:%C - %m%n
```

Exemple :

```
<?xml version="1.0" encoding="UTF-8" ?>
!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
  <appender name="console" class="org.apache.log4j.ConsoleAppender">
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
        value="%d{yyyy-MM-dd HH:mm:ss.SSS} %-8p [%t]:%C - %m%n" />
    </layout>
  </appender>
  <appender name="fichier"
    class="org.apache.log4j.RollingFileAppender">
    <param name="file" value="c:/monapp.log" />
    <param name="MaxFileSize" value="1024KB" />
        <param name="MaxBackupIndex" value="2" /></
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
        value="%d{yyyy-MM-dd HH:mm:ss.SSS} %-8p [%t]:%C - %m%n" />
    </layout>
  </appender>
  <root>
    <priority value="debug" />
    <appender-ref ref="console" />
    <appender-ref ref="fichier" />
  </root>
</log4j:configuration></pre>
```

21.2.7. La mise en oeuvre avancée

Cette section présente quelques fonctionnalités avancées de Log4j.

21.2.7.1. La lecture des logs

La consultation des logs peut ne pas être facile si elle doit être réalisée en temps réel ou si le volume de messages est très important.

21.2.7.1.1. La lecture pendant l'exécution du programme

Si l'application écrit régulièrement dans le fichier, un simple bloc note n'est pas suffisant pour lire les messages arrivés après l'ouverture du fichier.

Sous Unix, la commande tail est particulièrement utile car elle permet de visualiser les n dernières lignes d'un fichier alors que celui-ci est en train de grossir.

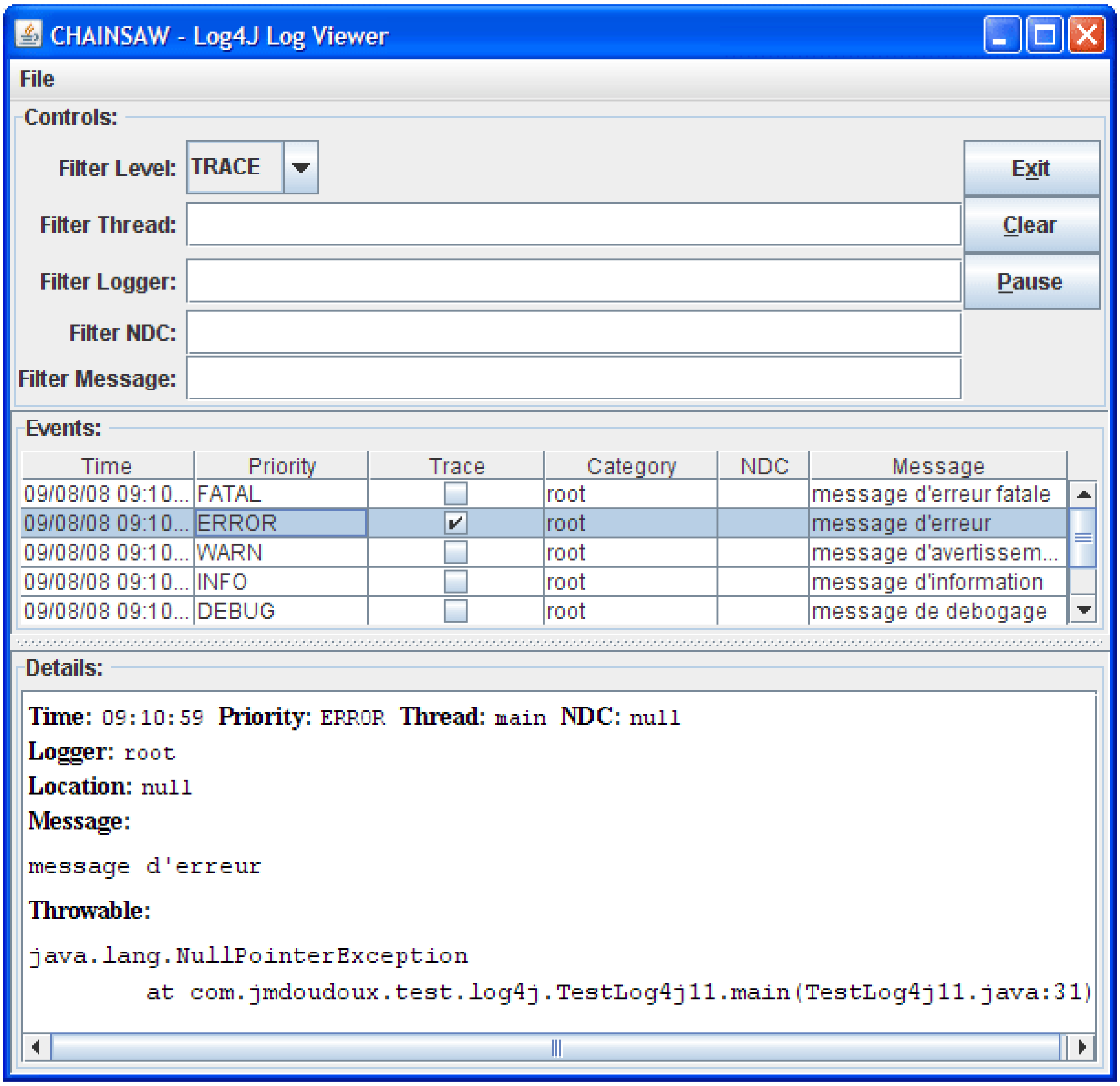
21.2.7.1.2. L'application Chainsaw

Log4j propose en standard une application graphique nommée chainsaw qui permet de visualiser des logs formatées avec un layout XMLLayout ou envoyées par un SocketAppender.

Pour exécuter Chainsaw, il faut exécuter la classe org.apache.log4j.chainsaw.Main

Exemple :

```
C:\>java -cp C:/java/apache-log4j-1.2.15/log4j-1.2.15.jar org.apache.log4j.chainsaw.Main
```



Filter Level: TRACE
Filter Thread:
Filter Logger:
Filter NDC:
Filter Message:

Time	Priority	Trace	Category	NDC	Message
09/08/08 09:10...	FATAL	☐	root		message d'erreur fatale
09/08/08 09:10...	ERROR	☑	root		message d'erreur
09/08/08 09:10...	WARN	☐	root		message d'avertissem...
09/08/08 09:10...	INFO	☐	root		message d'information
09/08/08 09:10...	DEBUG	☐	root		message de debogage

Time: 09:10:59 **Priority:** ERROR **Thread:** main **NDC:** null
Logger: root
Location: null
Message:

message d'erreur

Throwable:

```
java.lang.NullPointerException
        at com.jmdoudoux.test.log4j.TestLog4j11.main(TestLog4j11.java:31)
```

Pour consulter un fichier XML qui contient des logs formatées avec un XMLLayout, il faut utiliser l'option "Load File" du menu "File".

La partie "Controls" propose plusieurs filtres : il suffit de saisir les caractères recherchés et le filtre est appliqué au fur et à mesure de la saisie.

ChainSaw est aussi très pratique pour consulter les logs envoyées par un SocketAppender.

Il faut définir une variable d'environnement de la JVM nommée chainsaw.port pour préciser le port à écouter.

Exemple :

```
C:\>java -cp C:/java/apache-log4j-1.2.15/log4j-1.2.15.jar -Dchainsaw.port=10256
org.apache.log4j.chainsaw.Main
```

21.2.7.2. Les variables d'environnement

Log4j utilise plusieurs variables d'environnement de la JVM pour éventuellement modifier certains comportements.

Variable	Rôle
log4j.debug	Fournir des informations de débogage lors de la recherche de la configuration et de son chargement
log4j.configuration	Permet de préciser le nom du fichier properties qui contient la configuration. Ce fichier doit être dans le classpath
log4j.defaultInitOverride	Booléen qui permet de demander d'inhiber la recherche de la configuration. La valeur par défaut est false.

21.2.7.3. L'internationalisation des messages

La classe Category et par héritage la classe Logger proposent deux surcharges de la méthode l7dlog() qui permettent l'émission de messages internationalisés (l7d est le raccourci de localized).

Avant la première utilisation de la méthode l7dlog, il est nécessaire de préciser quel ResourceBundle doit être utilisé en invoquant la méthode setResourceBundle().

Les méthodes l7dlog() attendent en paramètres le niveau de gravité, la clé du message dans le resourceBundle et un objet de type Throwable. La seconde surcharge attend aussi un tableau d'objets qui seront insérés à leurs emplacements définis dans les valeurs de clés.

Exemple :

```
package fr.jmdoudoux.dej.log4j;

import java.util.Locale;
import java.util.ResourceBundle;

import org.apache.log4j.ConsoleAppender;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.SimpleLayout;

public class TestLog4j19 {
  static Logger logger = Logger.getLogger(TestLog4j19.class);

  public static void main(
      String args[]) {
    Logger logRoot = Logger.getRootLogger();
    ConsoleAppender ca = new ConsoleAppender();
    ca.setName("console");
    ca.setLayout(new SimpleLayout());
    ca.activateOptions();
    logRoot.addAppender(ca);
    logRoot.setLevel(Level.DEBUG);

    Locale locale = new Locale("fr", "FR");
    ResourceBundle messages = ResourceBundle.getBundle("MessagesLog", locale);
    logger.setResourceBundle(messages);

    logger.l7dlog(Level.DEBUG, "MESSAGE", null);

    locale = new Locale("en", "EN");
    messages = ResourceBundle.getBundle("MessagesLog", locale);
    logger.setResourceBundle(messages);

    logger.l7dlog(Level.DEBUG, "MESSAGE", null);

  }
}
```

Il faut définir les fichiers properties qui seront utilisés par le ResourceBundle. Ces fichiers doivent être stockés dans le classpath.

Exemple : le fichier MessagesLog.properties

```
MESSAGE=mon message en français
```

Exemple : le fichier MessagesLog_en.properties

```
MESSAGE=my message in English
```

Résultat :

```
DEBUG - mon message en français
DEBUG - my message in English
```

21.2.7.4. L'initialisation de Log4j dans une webapp

L'utilisation de Log4J dans une webapp est détaillée dans une section du chapitre «Les servlets» de ce tutoriel.

21.2.7.5. La modification dynamique de la configuration

Cette section sera développée dans une version future de ce document

21.2.7.6. NDC/MDC

Log4J propose deux fonctionnalités qui permettent d'ajouter des données contextuelles dans les logs :

- NDC (Nested Diagnostic Context) : une pile est associée à chaque thread pour y stocker des données contextuelles qui seront insérées dans les traces
- MDC (Mapped Diagnostic Context) : une map est associée à chaque thread pour y stocker des données contextuelles qui seront insérées dans les traces

Les classes NDC et MDC sont utilisées pour stocker ces informations contextuelles relatives à l'application qui peuvent être utilisées lors de la journalisation des messages. Les classes NDC et MDC proposent des méthodes statiques qui permettent de gérer des données contextuelles liées au ThreadLocal du thread courant.

Un cas typique d'utilisation est de pouvoir ajouter dans les logs d'une webapp L'identité de l'utilisateur qui est à l'origine de la trace.

La classe org.apache.log4j.NDC propose un contexte composé de chaînes de caractères qui pourront toutes être ajoutées dans le journal si le format le demande.

NDC utilise une pile pour stocker des informations contextuelles. Le NDC est stocké dans le ThreadLocal de chaque thread.

La classe NDC propose plusieurs méthodes :

Méthode	Rôle
void push(String)	Ajouter une chaîne de caractères dans le contexte

String pop(String)	Retirer du contexte le dernier élément ajouté
void remove()	Retirer le contexte du thread local
void clear()	Vider le contenu de la pile

Exemple :

```
import org.apache.log4j.Logger;
import org.apache.log4j.NDC;

public class TestNDC {
  private static Logger LOGGER = Logger.getLogger(TestNDC.class);

  public static void main(String[] args) {
    NDC.push("Partie1");
    LOGGER.info("debut des traitements");
    NDC.push("Etape1");

    LOGGER.info("traitement 1.1");
    NDC.pop();
    NDC.push("Etape2");
    LOGGER.info("traitement 1.2");

    NDC.remove();
    LOGGER.info("fin des traitements");
  }
}
```

Pour insérer le contenu du NDC dans la trace, il faut utiliser la variable %x dans le pattern. Lorsqu'un message est ajouté dans le journal et que l'option %x est utilisée par le motif du layout, alors tout le contenu courant du NDC est ajouté dans le journal.

Résultat :

```
log4j.rootLogger=INFO, stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p - [%x] %m%n
```

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd" >
<log4j:configuration>
  <appender name="stdout" class="org.apache.log4j.ConsoleAppender">
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%d{yyyy-MM-dd HH:mm:ss} %-5p - [%x] %m%n" />
    </layout>
  </appender>
  <root>
    <priority value="info"></priority>
    <appender-ref ref="stdout" />
  </root>
</log4j:configuration>
```

Toutes les informations contenues dans le NDC sont ajoutées dans le journal.

Résultat :

```
2013-03-10 15:17:45 INFO  - [Partie1] debut des traitements
2013-03-10 15:17:45 INFO  - [Partie1 Etape1] traitement 1.1
2013-03-10 15:17:45 INFO  - [Partie1 Etape2] traitement 1.2
2013-03-10 15:17:45 INFO  - [] fin des traitements
```

Les opérations réalisées sur le NDC affectent uniquement le thread courant.

Il est recommandé de limiter au strict nécessaire les informations contenues dans le NDC pour maintenir la lisibilité des logs.

Il est de la responsabilité de l'application de gérer correctement le contenu de la pile selon le contexte.

Il faut faire attention en utilisant le NDC car celui-ci maintient une référence sur le threadlocal ce qui empêche le ramasse-miettes de libérer la mémoire même si le thread est terminé. Pour éviter ce type de soucis, il est nécessaire d'invoquer la méthode remove() de la classe NDC.

La classe org.apache.log4j.MDC associe une Map pour chaque thread dans laquelle les données contextuelles sont stockées avec le nom de la donnée comme clé associée à sa valeur.

L'utilisation de MDC est plus souple que NDC car il permet de sélectionner le ou les éléments du contexte qui doivent apparaître dans le journal.

La gestion des éléments contenus dans le MDC se fait en utilisant les méthodes put() pour ajouter, get() pour obtenir et remove() pour supprimer un élément.

Exemple :

```
import org.apache.log4j.Logger;
import org.apache.log4j.MDC;

public class TestMDC {
  private static Logger LOGGER = Logger.getLogger(TestNDC.class);
  public static void main(String[] args) {
    MDC.put("user","jm");
    MDC.put("partie","Partie1");
    MDC.put("etape","Etape1");
    LOGGER.info("debut des traitements");
    MDC.put("etape","Etape2");
    LOGGER.info("fin des traitements");
  }
}
```

Pour insérer un élément contenu dans le MDC dans le journal, il faut utiliser la variable %X dans le pattern suivie du nom de la clé entre accolades.

Résultat :

```
log4j.rootLogger=INFO, stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p -
  [%X{partie} %X{etape}] %m%n
```

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM
"log4j.dtd" >
<log4j:configuration>
  <appender name="stdout" class="org.apache.log4j.ConsoleAppender">
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="**%d{yyyy-MM-dd HH:mm:ss}%-5p -
        [%X{partie} %X{etape}] %m%n" />
    </layout>
  </appender>
  <root>
    <priority value="info"></priority>
    <appender-ref ref="stdout"/>
  </root>
</log4j:configuration>
```

Seuls les éléments dont la clé est précisée dans le pattern sont insérés dans le journal.

Résultat :

```
2013-03-10 15:38:14 INFO  - [Partie1 Etape1] debut des traitements
2013-03-10 15:38:14 INFO  - [Partie1 Etape2] fin des traitements
```

L'utilisation de MDC requiert un JDK 1.2 minimum.

Les threads fils héritent automatiquement du MDC du thread parent.

Le choix d'utiliser NDC ou MDC dépend des besoins. Si les informations contextuelles sont imbriquées alors le NDC est le meilleur choix sinon il faut utiliser le MDC.

Avec les deux solutions, il est nécessaire de prendre en compte correctement la réinitialisation des données contextuelles : par exemple dans une application web, les données étant stockées dans le ThreadLocal, elles sont communes pour toutes les requêtes http qui sont traitées par le processeur associé à ce ThreadLocal.

Dans une application web, il est fréquent de stocker des informations contextuelles, comme l'utilisateur, l'adresse IP ou l'url, dans le NDC ou le MDC dans un point d'entrée du traitement des requêtes http, par exemple en utilisant un filtre.

21.2.8. Des best practices

Cette section fournit quelques conseils pour la mise en oeuvre de Log4j.

21.2.8.1. Le choix du niveau de gravité des messages

Le choix du niveau de gravité de chaque message émis est très important.

Voici quelques exemples d'utilisation de chaque niveau de gravité :

- fatal : messages concernant un arrêt imprévu de l'application
- error : messages d'erreurs nécessitant une analyse par exemple les exceptions levées
- warn : message d'avertissement
- info : messages d'information par exemple le démarrage ou l'arrêt de l'application, la connexion à une ressource, ...
- trace : messages pour suivre le flow d'exécution
- debug : messages de débogage par exemple pour obtenir la valeur de variables, ...

Hors de l'environnement de développement, le niveau de gravité minimum des messages doit être info. Le niveau debug n'est à utiliser que dans l'environnement de développement ou à utiliser temporairement pour des besoins spécifiques dans les autres environnements.

21.2.8.2. L'amélioration des performances

Log4j a été développé dans le souci de réduire au minimum le surcoût de son utilisation.

Cependant le logging a nécessairement un coût et ce coût peut devenir important si certaines précautions ne sont pas prises par le développeur.

Il est nécessaire de limiter le coût d'émission d'un message dont la construction est complexe surtout si ce dernier sera ignoré par le logger.

Exemple :

```
logger.debug("valeur="+valeur+" , i="+i+" ,next="+next);
```

Dans cet exemple, même si le niveau de gravité du logger est supérieur à debug, le coût de l'émission du message contiendra la création du message par concaténation des différentes valeurs.

Pour limiter ce coût, il est préférable de conditionner l'émission du message par un test préalable sur le niveau de gravité pris en charge par le logger lors de l'exécution du traitement.

Les classes Category et Logger proposent des méthodes pour effectuer ces tests.

Exemple :

```
if (logger.isDebugEnabled()) {
  logger.debug("valeur="+valeur+" , i="+i+" ,next="+next);
}
```

Avec ce test, le message n'est construit que s'il est pris en compte par le logger. L'inconvénient de ce test est qu'il est réalisé deux fois : une fois par la méthode isDebugEnabled() et une autre fois par la méthode debug(). Cependant ce surcoût est beaucoup moins important que la création du message.

Les temps de traitement de Log4j sont obligatoirement dépendants de l'utilisation qui en est faite dans l'application notamment :

- plus il a de messages émis plus les traitements sont longs : par exemple, il faut éviter d'envoyer un message dans une boucle
- plus les niveaux de gravité associés à un appender sont bas dans la hiérarchie, plus le nombre de messages à traiter est important
- plus il y a d'appenders plus le temps de traitement d'un message est important

Lors de l'utilisation d'un layout de type PatternLayout, l'utilisation de certains motifs sont connus pour être gourmands en temps de traitement. Même si les informations de ces motifs sont particulièrement utiles, il faut tenir compte de leur temps de traitement lors de leur utilisation.

Pour économiser de la mémoire, il est préférable de déclarer les loggers en tant que variables statiques.

Exemple :

```
private static Logger LOGGER = Logger.getLogger(MaClasse.class);
```

21.2.8.3. D'autres recommandations

Lorsqu'une erreur doit être journalisée, il faut toujours utiliser la surcharge qui accepte l'exception pour permettre d'ajouter la stacktrace dans le journal.

LOGGER.error("Erreur dans les traitements", e);

Dans un contexte multithread, comme dans une webapp par exemple, il est important d'ajouter le nom du thread dans le pattern des messages en utilisant %t.

21.3. L'API logging

L'usage de fonctionnalités de logging dans les applications est tellement répendu que SUN a décidé de développer sa propre API et de l'intégrer au JDK à partir de la version 1.4.

Cette API a été proposée à la communauté sous la Java Specification Request numéro 047 (JSR-047).

Le but est de proposer un système qui puisse être exploité facilement par toutes les applications.

L'API repose sur plusieurs classes principales et une interface:

- Logger : cette classe permet d'envoyer des messages dans le système de log
- LogRecord : cette classe encapsule le message
- Handler : cette classe représente la destination des messages
- Formatter : cette classe permet de formater le message avant son envoi vers la destination
- Filter : cette interface, dont le but est de déterminer si le message sera enregistré, doit être implémentée par les classes désireuses de filtrer les messages
- Level : cette classe représente le niveau de gravité du message
- LogManager : cette classe est un singleton qui permet de gérer l'état des Loggers

Un logger possède un ou plusieurs Handler qui sont des entités recevant les messages. Chaque Handler peut avoir un filtre associé en plus du filtre associé au Logger.

Chaque message possède un niveau de sévérité représenté par la classe Level.

21.3.1. La classe LogManager

La classe LogManager encapsule la configuration et les loggers de l'API de logging.

Cette classe est un singleton qui propose la méthode getLogManager() pour obtenir l'unique référence sur un objet de ce type.

Cet objet permet :

- de maintenir une liste de Logger désignée par un nom unique
- de lire et conserver la configuration de l'API

Pour réaliser ces actions, la classe LogManager possède plusieurs méthodes dont les principales sont :

Méthode	Rôle
boolean addLogger(Logger)	Ajouter un logger : cette méthode renvoie simplement false si le logger existe déjà avec le même nom, sinon elle ajoute le logger et renvoie true
Logger getLogger(String)	Obtenir un logger à partir de son nom
Enumeration getLoggerName()	Obtenir une énumération de tous les noms des logger
String getProperty(String)	Obtenir la valeur d'une propriété de la configuration
void readConfiguration()	Relire la configuration
LogManager getLogManager()	Renvoyer l'instance unique de cette classe
void reset()	Réinitialiser la configuration

La méthode getLogger() de la classe LogManager permet d'instancier un nouvel objet Logger si aucun Logger possédant le nom passé en paramètre n'a déjà été défini sinon elle renvoie l'instance existante.

Le LogManager conserve les instances de type Logger sous la forme de références faibles : il est donc possible qu'elle ne renvoie pas toujours la même instance pour un même nom.

Par défaut, la liste des Loggers contient toujours un Logger nommé global qui peut être facilement utilisé.

21.3.2. La classe Logger

La classe Logger est la classe qui se charge d'envoyer les messages dans la log. Un Logger est identifié par un nom qui est habituellement le nom qualifié de la classe dans laquelle le Logger est utilisé. Ce nom permet de gérer une hiérarchie de Logger. Cette gestion est assurée par le LogManager. Cette hiérarchie permet d'appliquer des modifications sur un Logger ainsi qu'à toute sa "descendance".

La méthode statique getLogger() de la classe Logger est un helper qui permet d'obtenir une instance de type Logger pour le nom fourni en paramètre.

Exemple :

```
package fr.jmdoudoux.dej.logging;

import java.util.logging.Logger;

public class TestLogging {

  private static Logger LOGGER = Logger.getLogger(TestLogging.class.getName());

  public static void main(String[] args) {
    LOGGER.info("Lancement de l'application");
  }
}
```

Il est aussi possible de créer des Logger anonymes.

La méthode getLogger() est un délégué qui demande au LogManager de lui donner l'instance de type Logger correspondant au nom passé en paramètre.

La classe Logger se charge d'envoyer les messages aux Handler enregistrés sous la forme d'un objet de type LogRecord. Par défaut, ces Handlers sont ceux enregistrés dans le LogManager. L'envoi des messages est conditionné par la comparaison du niveau de sévérité du message avec celui associé au Logger.

La classe Logger possède de nombreuses méthodes pour émettre des messages, notamment :

Méthode	Rôle
void addHandler(Handler handler)	Ajouter un Handler qui va recevoir les messages émis par le Logger
void config(String msg)	Emettre un message avec le niveau de gravité CONFIG
void entering(String sourceClass, String sourceMethod)	Emettre un message pour l'entrée dans une méthode
void entering(String sourceClass, String sourceMethod, Object param1)	Emettre un message pour l'entrée dans une méthode avec son premier paramètre
void entering(String sourceClass, String sourceMethod, Object[] params)	Emettre un message pour l'entrée dans une méthode avec un tableau des paramètres passés à la méthode
void exiting(String sourceClass, String sourceMethod)	Emettre un message de retour d'une méthode
void exiting(String sourceClass, String sourceMethod, Object result)	Emettre un message de retour d'une méthode avec la valeur de retour
void fine(String msg)	Emettre un message avec le niveau de gravité FINE
void finer(String msg)	Emettre un message avec le niveau de gravité FINER
void finest(String msg)	Emettre un message avec le niveau de gravité FINEST
static Logger getAnonymousLogger()	Créer un Logger anonyme
static Logger getAnonymousLogger(String resourceBundleName)	Créer un Logger anonyme

Filter getFilter()	Obtenir le filtre associé au Logger
Handler[] getHandlers()	Obtenir les Handlers associés au Logger
Level getLevel()	Obtenir le niveau minimum associé au Logger
static Logger getLogger(String name)	Obtenir une instance du Logger pour le nom fourni en paramètre
static Logger getLogger(String name, String resourceBundleName)	Obtenir une instance du Logger pour le nom fourni en paramètre
String getName()	Renvoyer le nom du Logger
Logger getParent()	Renvoyer le Logger père
ResourceBundle getResourceBundle()	Renvoyer le ResourceBundle associé au Logger pour la Locale courante
String getResourceBundleName()	Renvoyer le nom du RessourceBundle associé au Logger
void info(String msg)	Emettre un message avec le niveau de gravité INFO
boolean isLoggable(Level level)	Vérifier si un message avec le niveau passé en paramètre sera pris en compte par le Logger ou non
void log(Level level, String msg)	Emettre un message
void log(Level level, String msg, Object param1)	Emettre un message avec un objet en paramètre
void log(Level level, String msg, Object[] params)	Emettre un message avec un tableau d'objets en paramètre
void log(Level level, String msg, Throwable thrown)	Emettre un message avec un objet de type Throwable
void log(LogRecord record)	Emettre un message
void removeHandler(Handler handler)	Retirer un Handler associé au Logger
void setFilter(Filter newFilter)	Définir le filtre associé au Logger
void setLevel(Level newLevel)	Définir le niveau minimum d'un message émis par le Logger
void setParent(Logger parent)	Définir le Logger père
void severe(String msg)	Emettre un message avec le niveau de gravité SEVERE
void warning(String msg)	Emettre un message avec le niveau de gravité WARNING
void throwing(String sourceClass, String sourceMethod, Throwable thrown)	Emettre un message avec le niveau de gravité WARNING

Plusieurs méthodes sont définies pour chaque niveau de sévérité. Plutôt que d'utiliser la méthode log() en précisant le niveau de sévérité, il est possible d'utiliser la méthode dont le nom correspondant au niveau de sévérité.

La méthode log() possède plusieurs surcharges : le framework tente de déterminer dynamiquement les noms de la classe et de la méthode.

La méthode logp() possède plusieurs surcharges qui permettent de fournir des informations précises sur l'origine de l'émission du message notamment le nom de la classe, le nom de la méthode et le message.

Les différentes surcharges de la méthode logrb() permettent en plus de préciser le nom d'un ResourceBundle à utiliser.

Les méthodes entering() et exiting() sont très utiles pour faciliter le débogage.

21.3.3. La classe Level

Chaque message est associé à un niveau de sévérité représenté par un objet de type Level. Cette classe définit 7 niveaux de sévérité :

- Level.SEVERE : intialisée avec la valeur 1000
- Level.WARNING : intialisée avec la valeur 900
- Level.INFO : intialisée avec la valeur 800
- Level.CONFIG : intialisée avec la valeur 700
- Level.FINE : intialisée avec la valeur 500
- Level.FINER : intialisée avec la valeur 400
- Level.FINEST : intialisée avec la valeur 300

La valeur Level.OFF, initialisée avec la valeur Integer.MAX_VALUE, permet de désactiver toutes les actions de logging de l'API. La valeur Level.ALL, initialisée avec la valeur Integer.MIN_VALUE, permet de logger tous les messages quelque soit leur niveau.

Il est possible de définir des niveaux personnalisés en créant des classes filles : ces nouveaux niveaux doivent impérativement avoir une valeur unique.

Exemple :

```
    LOGGER.setLevel(Level.INFO);
```

La classe Level propose plusieurs méthodes :

Méthode	Rôle
String getName()	Obtenir le nom du niveau
String getLocalizedName()	Obtenir le nom du niveau dans la langue de la JVM
int intValue()	Obtenir la valeur du niveau
Level parse(String)	Obtenir le niveau à partir de son nom (méthode statique)

21.3.4. La classe LogRecord

La classe java.util.logging.LogRecord permet de passer des requêtes de logging à un Handler.

Elle possède un constructeur qui attend un objet de type Level et un message de type String et plusieurs méthodes qui sont des getters/setters pour des propriétés notamment :

Méthode	Rôle
Level getLevel()	Obtenir le niveau de sévérité du message
String getLoggerName()	Obtenir le nom du Logger
String getMessage()	Obtenir le message brut, avant le formatage et la localisation
long getMillis()	Obtenir le timestamp (nombre de millisecondes depuis 1970)
Object[] getParameters()	Obtenir les paramètres du message
ResourceBundle getResourceBundle()	Obtenir le ResourceBundle
String getResourceBundleName()	Obtenir le nom du ResourceBundle
long getSequenceNumber()	Obtenir le numéro de séquence
String getSourceClassName()	Obtenir le nom de la classe ayant émis le message
String getSourceMethodName()	Obtenir le nom de la méthode à l'origine du message
int getThreadID()	Obtenir l'identifiant du thread ayant émis le message

Throwable getThrown()	Obtenir l'objet de type Throwable associé au message
void setLevel(Level level)	Définir le niveau de sévérité du message
void setLoggerName(String name)	Définir le nom du Logger
void setMessage(String message)	Définir le message brut, avant le formatage et la localisation
void setMillis(long millis)	Définir le timestamp
void setParameters(Object[] parameters)	Définir les paramètres du message
void setResourceBundle(ResourceBundle bundle)	Définir le ResourceBundle
void setResourceBundleName(String name)	Définir le nom du ResourceBundle
void setSequenceNumber(long seq)	Définir le numéro de séquence
void setSourceClassName(String sourceClassName)	Définir le nom de la classe émettant le message
void setSourceMethodName(String sourceMethodName)	Définir le nom de la méthode émettant le message
void setThreadID(int threadID)	Définir l'identifiant du thread ayant émis le message
void setThrown(Throwable thrown)	Définir l'objet de type Throwable associé au message

Les données contenues dans un objet de type LogRecord sont utilisées par les filtres et les formateurs lors de l'émission d'un message vers les handlers.

21.3.5. La classe Handler

Un Handler reçoit un message d'un logger et l'envoie vers une cible. Un logger peut être associé à plusieurs Handler.

La classe java.util.logging.Handler possède plusieurs méthodes :

Méthode	Rôle
void close()	Fermer le Handler et libérer les éventuelles ressources utilisées
void flush()	Vider le tampon
String getEncoding()	Renvoyer le jeu d'encodage de caractères utilisé par le Handler
ErrorManager getErrorManager()	Renvoyer le gestionnaire d'erreurs associé au Handler
Filter getFilter()	Renvoyer le filtre associé au Handler
Formatter getFormatter()	Renvoyer le formateur associé au Handler
Level getLevel()	Renvoyer le niveau minimum des messages traités par le Handler
boolean isLoggable(LogRecord record)	Vérifier si le message peut être traité par le Handler
void setEncoding(String encoding)	Définir le jeu d'encodage de caractères utilisé par le Handler
void setErrorManager(ErrorManager em)	Définir le gestionnaire d'erreurs associé au Handler
void setFilter(Filter newFilter)	Définir le filtre associé au Handler
void setFormatter(Formatter newFormatter)	Définir le formateur associé au Handler
void setLevel(Level newLevel)	Définir le niveau minimum d'un message pour être pris en compte par le Handler

Le framework propose plusieurs classes filles qui représentent différentes destinations pour émettre les messages :

- StreamHandler : envoie des messages dans un flux de sortie
- ConsoleHandler : envoie des messages sur la sortie standard d'erreurs
- FileHandler : envoie des messages sur un fichier
- SocketHandler : envoie des messages dans une socket réseau
- MemoryHandler : envoie des messages dans un tampon en mémoire

Exemple :

```
package fr.jmdoudoux.dej.logging;

import java.io.IOException;
import java.util.logging.FileHandler;
import java.util.logging.Handler;
import java.util.logging.Logger;

public class TestLogging {

  private static Logger LOGGER = Logger.getLogger(TestLogging.class.getName());

  public static void main(String[] args) {
    Handler fh;
    try {
      fh = new FileHandler("TestLogging.log");
      LOGGER.addHandler(fh);
    } catch (SecurityException e) {
      LOGGER.severe("Impossible d'associer le FileHandler");
    } catch (IOException e) {
      LOGGER.severe("Impossible d'associer le FileHandler");
    }
    LOGGER.info("Lancement de l'application");
  }
}
```

Il est possible de désactiver un handler simplement en invoquant sa méthode setLevel() avec la valeur Level.OFF en paramètre. Pour le réactiver, il faut invoquer la méthode setLevel() avec le niveau désiré.

21.3.5.1. La classe FileHandler

La classe java.util.logging.FileHandler permet d'écrire des messages dans un fichier. Il est possible de définir une rotation sur plusieurs fichiers : dès que le fichier atteint une certaine taille, le fichier est fermé et un nouveau fichier est créé. Les fichiers précédents sont renommés en utilisant un suffixe numérique. Le formateur par défaut est une instance de type XMLFormatter.

La classe FileHandler possède plusieurs constructeurs :

Constructeur	Rôle
FileHandler()	
FileHandler(String pattern)	Précise le motif du nom du fichier
FileHandler(String pattern, boolean append)	Précise le motif du nom du fichier et si le fichier existant doit être complété
FileHandler(String pattern, int limit, int count)	Précise le motif du nom du fichier, la taille maximale des fichiers et le nombre de fichiers à conserver lors de la rotation
FileHandler(String pattern, int limit, int count, boolean append)	Précise le motif du nom du fichier, la taille maximale des fichiers, le nombre de fichiers à conserver lors de la rotation et si le fichier existant doit être complété

Le fonctionnement d'un FileHandler peut être configuré en utilisant plusieurs propriétés :

Propriété	Rôle
java.util.logging.FileHandler.level	Définir le niveau de sévérité par défaut du Handler (Level.ALL par défaut)
java.util.logging.FileHandler.filter	Définir le nom de la classe de type Filter associée au Handler (aucun par défaut)
java.util.logging.FileHandler.formatter	Définir le nom de la classe de type Formatter associée au Handler (java.util.logging.XMLFormatter par défaut)
java.util.logging.FileHandler.encoding	Définir le jeu d'encodage de caractères à utiliser (par défaut celui de la plate-forme)
java.util.logging.FileHandler.limit	Définir la taille maximale du fichier en octets. La valeur zéro précise qu'il n'y a pas de limite (0 par défaut)
java.util.logging.FileHandler.count	Définir le nombre maximum de fichiers lors des rotations (1 par défaut)
java.util.logging.FileHandler.pattern	Définir un motif pour le nom du fichier (la valeur par défaut est "%h/java%u.log")
java.util.logging.FileHandler.append	Définir si les messages doivent être ajoutés à un fichier existant avec la valeur true, avec la valeur false (par défaut), si le fichier doit être écrasé

La valeur par défaut d'une propriété est utilisée si aucune valeur n'est explicitement précisée ou si la valeur précisée est invalide.

Le motif pour le nom du fichier est une chaîne de caractères qui peut contenir une ou plusieurs séquences qui seront remplacées dynamiquement à l'exécution par leurs valeurs respectives.

Motif	Rôle
/	Le séparateur de chemin du système
%t	Le répertoire temporaire du système
%h	La valeur de la propriété système user.home
%g	Le numéro généré pour chaque fichiers lors de la rotation des fichiers. Chaque fichier utilise un numéro dont le premier est 0
%u	Un nombre unique permettant de gérer les conflits. 0 par défaut, ce nombre est incrémenté tant que le fichier est utilisé par un processus jusqu'à ce que le fichier soit utilisable. Le fichier doit être stocké sur un disque local
%%	Le caractère %

Si le motif ne contient pas de %g et que le nombre de fichiers est supérieur à 1 alors un nombre unique sera ajouté à la fin du nom du fichier précédé d'un caractère point.

21.3.6. L'interface Filter

L'interface java.util.logging.Filter définit une seule méthode isLoggable(LogRecord) qui retourne un booléen. Cette méthode permet de déterminer si le message doit être loggué ou non. Si la méthode renvoie false alors l'objet de type LogRecord est ignoré.

Un objet de type Filter peut être associé à un Logger ou à un Handler. La méthode setFilter() de la classe Logger permet de lui associer un filtre.

Pour créer son propre filtre, il suffit de créer une classe qui implémente l'interface Filter.

21.3.7. La classe Formatter

La classe Formatter permet de formater un message. Une instance de type Formatter est associée à chaque Handler.

Le framework propose deux implémentations :

- SimpleFormatter : pour formater l'enregistrement sous forme de chaînes de caractères
- XMLFormatter : pour formater l'enregistrement en XML

XMLFormatter utilise une DTD particulière. Le tag racine est <log>. Chaque enregistrement est inclus dans un tag <record>. Chaque tag <record> possède plusieurs tags fils :

Tag	Rôle
Date	Date et heure d'émission du message
Millis	Timestamp de l'émission du message
Sequence	
Logger	Nom du Logger utilisé pour émettre le message
Level	Niveau de sévérité du message
Class	Nom de la classe
Method	Nom de la méthode
Thread	Numéro du thread
Message	Contenu du message

Il est possible de créer son propre formateur en créant une classe fille de la classe Formatter et en redéfinissant les méthodes format(), getHead() et getTail().

Chaque Handler est associé à une instance de type Formatter. La méthode setFormatter() de la classe Handler permet d'associer un autre formateur.

21.3.8. Le fichier de configuration

Un fichier particulier au format Properties permet de préciser des paramètres de configuration pour le système de log tels que le niveau de sévérité géré par un Logger particulier et sa descendance, les paramètres de configuration des Handlers, etc...

Il est possible de préciser le niveau de sévérité pris en compte par tous les Logger :

.level = niveau

Il est possible de définir les handlers par défaut :

handlers = java.util.logging.FileHandler

Pour préciser d'autres handlers, il faut les séparer par des virgules.

Pour définir le niveau de sévérité d'un Handler, il suffit de le préciser :

java.util.logging.FileHandler.level = niveau

Un fichier par défaut est défini avec les autres fichiers de configuration dans le répertoire lib du JRE. Ce fichier se nomme logging.properties.

Il est possible de préciser un fichier particulier précisant son nom dans la propriété système java.util.logging.config.file exemple : java -Djava.util.logging.config.file=monLogging.properties

21.4. Jakarta Commons Logging (JCL)

Le projet Jakarta Commons propose un sous-projet nommé Logging qui encapsule l'usage de plusieurs systèmes de logging et facilite ainsi leur utilisation dans les applications. Ce n'est pas un autre système de log mais il propose un niveau d'abstraction qui permet sans changer le code d'utiliser indifféremment n'importe lequel des systèmes de logging supportés. Son utilisation est d'autant plus pratique qu'il existe plusieurs systèmes de log dont aucun des plus répandus, Log4j et l'API logging du JDK 1.4 n'est dominant.

Le grand intérêt de cette bibliothèque est donc de rendre l'utilisation d'un système de log dans le code indépendant de l'implémentation de ce système. JCL encapsule l'utilisation de Log4j, l'API logging du JDK 1.4 et LogKit.

De nombreux projets du groupe Jakarta tels que Tomcat ou Struts utilisent cette bibliothèque. La version de JCL utilisée dans cette section est le 1.0.3

Le package, contenu dans le fichier commons-logging-1.0.3.zip peut être téléchargé sur le site https://commons.apache.org/logging/. Il doit ensuite être décompressé dans un répertoire du système d'exploitation.

Pour utiliser la bibliothèque, il faut ajouter le fichier dans le classpath.

L'inconvénient d'utiliser cette bibliothèque est qu'elle n'utilise que le dénominateur commun des systèmes de log qu'elle supporte : ainsi certaines caractéristiques d'un système de log particulier ne pourront être utilisées avec cette API .

La bibliothèque propose une fabrique qui renvoie, en fonction du paramètre précisé, un objet qui implémente l'interface Log. La méthode statique getLog() de la classe LogFactory permet d'obtenir cet objet : elle attend en paramètre soit un nom sous la forme d'une chaîne de caractères soit un objet de type Class dont le nom sera utilisé. Si un objet de type log possédant ce nom existe déjà alors c'est cette instance qui est renvoyée par la méthode sinon c'est une nouvelle instance qui est retournée. Ce nom représente la catégorie pour le système log utilisé, si celui-ci supporte une telle fonctionnalité.

Par défaut, la méthode getLog() utilise les règles suivantes pour déterminer le système de log à utiliser :

- Si la bibliothèque Log4j est incluse dans le classpath de la JVM alors celle-ci sera utilisée par défaut par la bibliothèque Commons Logging.
- Si le JDK 1.4 est utilisé et que Log4j n'est pas trouvé alors le système utilisé par défaut est celui fourni en standard avec le JDK (java.util.logging)
- Si aucun de ces systèmes de log n'est trouvé, alors JCL utilise un système de log basic fourni dans la bibliothèque : SimpleLog. La configuration de ce système se fait dans un fichier nommé simplelog.properties

Il est possible de forcer l'usage d'un système de log particulier en précisant la propriété org.apache.commons.logging.Log à la machine virtuelle.

Pour complètement désactiver le système de log, il suffit de fournir la valeur org.apache.commons.logging.impl.NoOpLog à la propriété org.apache.commons.logging.Log de la JVM. Attention dans ce cas, plus aucun log ne sera émis.

Il existe plusieurs niveaux de gravité que la bibliothèque tentera de faire correspondre au mieux avec le système de log utilisé.

21.5. D'autres API de logging

Il existe d'autres API de logging dont voici une liste non exhaustive :

Produit	URL
Lumberjack	http://javalogging.sourceforge.net/
Javalog	https://sourceforge.net/projects/javalog/

Simple Logging Facade for Java (SLF4J)	www.slf4j.org/
Logback	logback.qos.ch/
Blitz4j	https://github.com/Netflix/blitz4j

22. L'API Stream

Chapitre 22

Niveau : 🟩🟩🟨⬜⬜⬜ Intermédiaire

En programmation fonctionnelle, on décrit le résultat souhaité mais pas comment on obtient le résultat. Ce sont les fonctionnalités sous-jacentes qui se chargent de réaliser les traitements requis en tentant de les exécuter de manière optimisée.

Ce mode de fonctionnement est similaire à SQL : le langage SQL permet d'exprimer une requête mais c'est le moteur de la base de données qui choisir la meilleure manière d'obtenir le résultat décrit. Comme avec SQL, la manière dont on exprime le résultat peut influencer la manière dont le résultat va être obtenu notamment en termes de performance.

Java 8 propose l'API Stream pour mettre en oeuvre une approche de la programmation fonctionnelle sachant que Java est et reste un langage orienté objet.

Le concept de Stream existe déjà depuis longtemps dans l'API I/O, notamment avec les interfaces InputStream et OutputStream. Il ne faut pas confondre l'API Stream de Java 8 avec les classes de type xxxStream de Java I/O. Les streams de Java I/O permettent de lire ou écrire des données dans un flux (sockets, fichiers, ...). Le concept de Stream de Java 8 est différent du concept de flux (stream) de l'API I/O même si l'approche de base est similaire : manipuler un flux d'octets ou de caractères pour l'API I/O et manipuler un flux de données pour l'API Stream. Cette dernière repose sur le concept de flux (stream en anglais) qui est une séquence d'éléments.

L'API Stream facilite l'exécution de traitements sur des données de manière séquentielle ou parallèle. Les Streams permettent de laisser le développeur se concentrer sur les données et les traitements réalisés sur cet ensemble de données sans avoir à se préoccuper de détails techniques de bas niveau comme l'itération sur chacun des éléments ou la possibilité d'exécuter ces traitements de manière parallèle.

L'API Stream de Java 8 propose une approche fonctionnelle dans les développements avec Java. Elle permet de décrire de manière concise et expressive un ensemble d'opérations dont le but est de réaliser des traitements sur les éléments d'une source de données. Cette façon de faire est complètement différente de l'approche itérative utilisée dans les traitements d'un ensemble de données avant Java 8.

Ceci permet au Stream de pouvoir :

- Optimiser les traitements exécutés grâce au lazyness et à l'utilisation d'opérations de type short-circuiting qui permettent d'interrompre les traitements avant la fin si une condition est atteinte
- Exécuter certains traitements en parallèle à la demande du développeur

L'API Stream permet de réaliser des opérations fonctionnelles sur un ensemble d'éléments. De nombreuses opérations de l'API Stream attendent en paramètre une interface fonctionnelle ce qui conduit naturellement à utiliser les expressions lambdas et les références de méthodes dans la définition des Streams. Un Stream permet donc d'exécuter des opérations standards dont les traitements sont exprimés grâce à des expressions lambdas ou des références de méthodes.

Un Stream permet d'exécuter une agrégation d'opérations de manière séquentielle ou en parallèle sur une séquence d'éléments obtenus à partir d'une source dans le but d'obtenir un résultat.

Exemple (code Java 8) :

```
    double tailleMoyenneDesHommes = employes
        .stream()
        .filter(e -> e.getGenre() == Genre.HOMME)
        .mapToInt(e -> e.getTaille())
        .average()
        .orElse(0);
```

Dans l'exemple ci-dessus, une collection d'éléments de type Employe est utilisée comme source pour un Stream qui va effectuer des opérations de type filter-map-reduce pour calculer la moyenne de la taille des personnes de sexe masculin. La moyenne calculée par la méthode average() est une opération dite de réduction.

Le code utilisé est :

- Concis et clair car il repose sur l'utilisation d'opérations prédéfinies
- Exprimé de manière déclarative (c'est une description du résultat attendu) plutôt que de manière impérative (c'est une description des différentes étapes nécessaires à l'exécution des traitements)

Les éléments traités par le Stream sont fournis par une source qui peut être de différents types :

- Une collection
- Un tableau
- Un flux I/O
- Une chaîne de caractères
- ...

Avec l'API Stream il est possible de déclarer de manière concise des traitements sur une source de données qui seront exécutés de manière séquentielle ou parallèle. Le traitement en parallèle d'un ensemble de données requiert plusieurs opérations :

- La séparation des données en différents paquets plus petits (forking)
- L'exécution dans un thread dédié des traitements sur chaque élément d'un paquet, généralement en utilisant un pool de threads pour limiter la quantité de ressources utilisées
- La combinaison des résultats de chaque paquet pour obtenir le résultat final (joining)

Java 7 propose le framework Fork/Join pour faciliter la mise en oeuvre de ces opérations en parallèle. L'API Stream utilise ce framework pour l'exécution des traitements en parallèle.

L'utilisation de l'API Stream possède donc plusieurs avantages :

- L'exécution, sur un ensemble de données, de traitements définis de manière déclarative
- La quantité de code à produire pour obtenir un traitement similaire reposant sur sa propre itération est réduite
- La possibilité d'exécuter ses traitements en parallèle

Ce chapitre contient plusieurs sections :

♦ Le besoin de l'API Stream
♦ Le pipeline d'opérations d'un Stream
♦ Les opérations intermédiaires
♦ Les opérations terminales
♦ Les Collectors
♦ Les Streams pour des données primitives
♦ L'utilisation des Streams avec les opérations I/O
♦ Le traitement des opérations en parallèle
♦ Les optimisations réalisées par l'API Stream
♦ Les Streams infinis
♦ Le débogage d'un Stream
♦ Les limitations de l'API Stream
♦ Quelques recommandations sur l'utilisation de l'API Stream

22.1. Le besoin de l'API Stream

Il est fréquent dans les applications de devoir manipuler un ensemble de données. Pour stocker cet ensemble de données, Java propose, depuis sa version 1.1, l'API Collections.

Différentes évolutions ont permis de rendre de plus en plus expressif le code pour traiter ces données. Celles-ci vont être illustrées dans les exemples ci-dessous à l'aide d'un petit algorithme qui va calculer la somme des valeurs inférieures à 10 dans une collection contenant les 12 premiers entiers.

Avec Java 5, il est possible d'utiliser un Iterator avec les generics.

Exemple (code Java 5.0) :

```
    List<Integer> entiers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12);
    Iterator<Integer> it = entiers.iterator();
    long somme = 0;
    while (it.hasNext()) {
      int valeur = it.next();
      if (valeur < 10) {
        somme += valeur;
      }
    }
    System.out.println(somme);
```

Java 5 a aussi introduit une nouvelle syntaxe de l'instruction for qui permet de faciliter l'écriture du code requis pour parcourir une collection

Exemple (code Java 5.0) :

```
   long somme = 0;
   for (int valeur : entiers) {
     if (valeur < 10) {
       somme += valeur;
     }
   }
   System.out.println(somme);
```

Dans les exemples précédents, le parcours des éléments de la collection est fait à la main sous la forme d'une itération externe (external iteration) qui contient la logique de traitements. Finalement le code à produire pour réaliser cette tâche relativement simple est assez important.

Java 8 propose la méthode forEach() dans l'interface Collection qui attend en paramètre une interface fonctionnelle de type Consumer.

Exemple (code Java 8) :

```
    LongAdder somme = new LongAdder();
    entiers.forEach(valeur -> {
      if (valeur < 10) {
        somme.add(valeur);
      }
    });
    System.out.println(somme);
```

Dans les exemples précédents, le code de l'algorithme est exécuté séquentiellement. Il serait beaucoup plus complexe et verbeux pour arriver à le faire exécuter en parallèle (en imaginant que le volume de données à traiter soit beaucoup plus important).

L'ajout du support de traitement en parallèle dans l'API Collections aurait été très compliqué : le choix a été fait de définir une nouvelle API permettant de :

- Faciliter l'exécution de traitements sur un ensemble de données

- Réduire la quantité de code nécessaire pour le faire
- Permettre d'exécuter des opérations en parallèle de manière très simple

Pour offrir une solution à cette problématique et proposer la possibilité de simplifier la réalisation d'opérations plus ou moins complexes sur des données, Java 8 propose l'API Stream.

Exemple (code Java 8) :

```
    long somme = entiers.stream()
                        .filter(v -> v <10)
                        .mapToInt(i -> i)
                        .sum();
    System.out.println(somme);
```

Elle permet notamment très facilement d'exécuter ces traitements en parallèle.

Exemple (code Java 8) :

```
    long somme = entiers.parallelStream()
                        .filter(v -> v < 10)
                        .mapToInt(i -> i)
                        .sum();
    System.out.println(somme);
```

L'approche proposée par Java 8 est de laisser l'API réaliser l'itération (internal iteration).

L'utilisation de l'API Stream permet de se concentrer sur l'essentiel (la logique des traitements) et de l'exprimer sous une forme expressive et succincte. L'exemple de traitement tient maintenant sur une ligne de code.

Ainsi avant Java 8, la seule manière de traiter les éléments d'une collection est d'itérer sur ceux-ci généralement en utilisant un Iterator explicitement en utilisant une boucle while ou implicitement en utilisant une boucle de type for each. Cette approche est celle de la programmation impérative avec laquelle il est nécessaire de coder toutes les étapes de l'algorithme à utiliser. Généralement cela implique une écriture verbeuse du code de ces traitements puisque celui-ci détaille ce que doivent faire les fonctionnalités. Dans la plupart des cas, il serait moins verbeux de décrire les résultats attendus et de laisser l'application exécuter à sa manière les fonctionnalités correspondantes.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.ArrayList;
import java.util.List;

public class TestBoucle {

  public static void main(String[] args) {
    List<Personne> personnes = new ArrayList<>(6);
    personnes.add(new Personne("p1", Genre.HOMME, 176));
    personnes.add(new Personne("p2", Genre.HOMME, 190));
    personnes.add(new Personne("p3", Genre.FEMME, 172));
    personnes.add(new Personne("p4", Genre.FEMME, 162));
    personnes.add(new Personne("p5", Genre.HOMME, 176));
    personnes.add(new Personne("p6", Genre.FEMME, 168));

    long total = 0;
    int nbPers = 0;
    for (Personne personne : personnes) {
      if (personne.getGenre() == Genre.FEMME) {
        nbPers++;
        total += personne.getTaille();
      }
    }
    double resultat = (double) total / nbPers;
    System.out.println("Taille moyenne des femmes = " + resultat);
  }
}
```

Les traitements de l'exemple précédent peuvent être réécrits en utilisant l'API Stream.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.ArrayList;
import java.util.List;

public class TestBoucle {

  public static void main(String[] args) {
    List<Personne> personnes = new ArrayList<>(6);
    personnes.add(new Personne("p1", Genre.HOMME, 176));
    personnes.add(new Personne("p2", Genre.HOMME, 190));
    personnes.add(new Personne("p3", Genre.FEMME, 172));
    personnes.add(new Personne("p4", Genre.FEMME, 162));
    personnes.add(new Personne("p5", Genre.HOMME, 176));
    personnes.add(new Personne("p6", Genre.FEMME, 168));

    resultat = personnes
      .stream()
      .filter(p -> p.getGenre() == Genre.FEMME)
      .mapToInt(p -> p.getTaille())
      .average()
      .getAsDouble();
    System.out.println("Taille moyenne des femmes = " + resultat);
  }
}
```

Dans l'exemple ci-dessus, les données traitées par le Stream subissent plusieurs opérations :

- Un filtre qui utilise le Predicat passé en paramètre sous la forme d'une expression Lambda pour ne conserver que les données concernant le genre féminin
- Le nouveau Stream obtenu est transformé en utilisant la méthode mapToInt() qui utilise l'interface fonctionnelle ToIntFunction passée en paramètre sous la forme d'une expression lambda pour obtenir un nouveau Stream qui ne contient que les tailles des personnes féminines sous la forme d'entiers
- La méthode d'agrégation average() du Stream de type IntStream obtenu permet de calculer la moyenne de ces valeurs entières
- Le résultat est une instance de type OptionalDouble dont la méthode getAsDouble() permet d'obtenir la valeur sous la forme d'une valeur flottante de type double si une valeur est présente

L'API Stream propose donc d'ajouter une manière plus expressive et une approche fonctionnelle au langage Java pour le traitement de données :

- Il n'y a plus de code pour itérer sur chacun des éléments
- Des opérations sont utilisées en leur passant en paramètre des expressions lambdas ou des références de méthode pour indiquer le détail de la tâche à accomplir

L'interface Stream<T> propose une liste définie d'opérations qu'il sera possible de réaliser. La plupart de ces opérations renvoient une instance de type Stream<T> pour permettre de combiner ces opérations.

22.1.1. L'API Stream

L'API Stream est contenue dans le package java.util.stream. Ce package est composée de 10 interfaces, deux classes et une énumération.

Les interfaces de l'API sont :

Interface	Description

BaseStream<T,S extends BaseStream<T,S>>	Interface de base pour les Streams
Collector<T,A,R>	Interface pour une opération de réduction qui accumule les éléments dans un conteneur mutable avec éventuellement une transformation du résultat un fois tous les éléments traités
DoubleStream	Un Stream dont la séquence est composée d'éléments de primitif double
DoubleStream.Builder	Un builder pour un DoubleStream
IntStream	Un Stream dont la séquence est composée d'éléments de primitif int
IntStream.Builder	Un builder pour un IntStream
LongStream	Un Stream dont la séquence est composée d'éléments de primitif long
LongStream.Builder	Un builder pour un LongStream
Stream<T>	Un stream dont la séquence est composée d'éléments de type T
Stream.Builder<T>	Un builder pour un Stream

L'interface BaseStream<T,S extends BaseStream<T,S>> est l'interface mère des principales interfaces utilisées :

- Stream<T> : interface permettant le traitement d'objets de type T de manière séquentielle ou parallèle
- IntStream : spécialisation pour traiter des données de type int
- LongStream : spécialisation pour traiter des données de type long
- DoubleStream : spécialisation pour traiter des données de type double

Si les éléments du Stream sont des données numériques alors il n'est pas recommandé d'utiliser un Stream<T> où T est de type Double, Long ou Integer. Les opérations sur ces objets vont nécessiter de l'autoboxing. Dans ces cas, il est préférable d'utiliser un DoubleStream, IntStream ou LongStream pour améliorer les performances des traitements de ces données et profiter de certaines opérations de réductions dédiés aux type primitifs tels que la somme ou la moyenne.

L'API définit plusieurs classes :

Classe	Description
Collectors	Propose plusieurs fabriques pour obtenir des implémentations de l'interface Collector qui implémentent des opérations de réduction communes
StreamSupport	Propose des méthodes de bas niveau pour créer des Streams à partir d'un Spliterator fourni en paramètre

22.1.2. Le rôle d'un Stream

Il est très fréquent de vouloir traiter des données stockées dans une collection. Cependant, l'exécution de traitements sur une collection présente plusieurs inconvénients :

- Elle requière de nombreuses lignes de code notamment quelques-unes pour gérer l'itération sur les différents éléments ou pour réaliser des opérations de base comme la somme de valeurs ou la recherche du plus petit ou plus grand élément
- Il n'est pas facile de faire exécuter ces traitements de manière parallèle : ceci pourrait cependant être particulièrement utile lors du traitement d'une très large quantité de données. Et les quantités de données à traiter augmente de manière croissante globalement dans le temps

Exemple (code Java 7) :

```
    List<Employe> employeMasculins = new ArrayList<>();
    for (Employe e : employes) {
      if (e.getGenre() == Genre.HOMME) {
        employeMasculins.add(e);
      }
    }

    Collections.sort(employeMasculins, new Comparator<Employe>() {
      @Override
      public int compare(Employe p1, Employe p2) {
        return p2.getTaille() - p1.getTaille();
      }
    });

    List<String> nomEmployes = new ArrayList<>();
    for (Personne p : employeMasculins) {
      nomEmployes.add(p.getNom());
    }
    for (String nom : nomEmployes) {
      System.out.println(nom);
   }
```

Un Stream peut être vu comme une abstraction qui permet de réaliser des opérations définies sur un ensemble de données.

Le but principal d'un Stream est de pouvoir décrire de manière expressive des traitements à exécuter sur un ensemble de données en limitant au maximum le code à produire pour y arriver. Globalement cela passe par une description de ce que l'on veut faire mais pas comment cela va se faire.

De la même manière que SQL permet de définir les données à obtenir et la façon dont on souhaite qu'elles soient restituées, les Streams permettent de décrire des opérations à réaliser sur un ensemble de données.

La déclaration d'un Stream se fait en plusieurs étapes :

- Création d'un Stream à partir d'une source : collection ou autre
- Définition de zéro, une ou plusieurs opérations intermédiaires qui renvoient toutes un Stream
- Définition d'une opération terminale qui va permettre d'obtenir le résultat des traitements et qui va clôturer le Stream

Exemple (code Java 8) :

```
    List<String> nomEmployes =
      employes.stream()
              .filter(e -> e.getGenre() == Genre.HOMME)
              .sorted(Comparator.comparingInt(Employe::getTaille)
                                .reversed())
              .map(Personne::getNom)
              .collect(Collectors.toList());

    for (String nom : nomEmployes) {
      System.out.println(nom);
    }
```

Le code Java 8 est plus compact car il ne contient que les informations indispensables à la bonne exécution des traitements :

- On obtient un Stream à partir de la collection contenant les données
- On applique plusieurs opérations filtre pour ne conserver sur les employés masculins, trie ces employés par taille décroissante et extraction des noms
- On les insère dans une collection de type List

De plus, comme la manière d'exécuter ces traitements est laissée à l'implémentation du Stream, il est possible que celui-ci les exécute en parallèle sur simple demande en remplaçant la méthode stream() par parallelStream().

Exemple (code Java 8) :

```
    List<String> nomEmployes =
      employes.parallelStream()
              .filter(e -> e.getGenre() == Genre.HOMME)
              .sorted(Comparator.comparingInt(Employe::getTaille)
                                .reversed())
              .map(Personne::getNom)
              .collect(Collectors.toList());

    for (String nom : nomEmployes) {
      System.out.println(nom);
    }
```

Un Stream parallèle va en interne paralléliser l'exécution du pipeline d'opérations en répartissant les données sur plusieurs threads pour utiliser les coeurs présents sur la machine.

22.1.3. Les concepts mis en oeuvre par les Streams

Une définition simple d'un Stream pourrait être : « l'application d'un ensemble d'opérations sur une séquence d'éléments issus d'une source dans le but d'obtenir un résultat ».

Cette définition met en avant plusieurs concepts :

- Un ensemble d'opérations : ce sont les traitements ordonnés qui seront exécutés. Chaque opération permet d'effectuer une tâche. Les opérations communes en programmation fonctionnelle sont proposées : filtre (filter), transformation (map), réduction (reduce), recherche (find), correspondance (match), tri (sorted), ...
- Pipeline d'opérations : de nombreuses opérations renvoient un Stream ce qui permet de les enchaîner et éventuellement de réaliser des optimisations
- Une séquence d'éléments : ce sont les données d'un certain type sur lesquelles les opérations vont être appliquées mais le Stream ne stocke pas ses données car elles sont obtenues à la demande
- Une source : elle permet de fournir les données à traiter au Stream comme une collection, un tableau ou tout autre ressource capable de fournir un Stream

De plus, un Stream utilise d'autres concepts lors de sa mise en oeuvre :

- Lazyness : les données sont consommées de manière tardive
- Short-circuiting : certaines opérations peuvent interrompre le traitement des éléments
- Parallélisation possible des traitements
- Internal iteration : c'est l'API qui réalise l'itération sur les données à traiter

22.1.3.1. Le mode de fonctionnement d'un Stream

Un Stream permet d'exécuter des opérations sur un ensemble de données obtenues à partir d'une source afin de générer un résultat.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.Arrays;
import java.util.List;

public class JavaApplication1 {

  public static void main(String[] args) {
    List<String> maListe = Arrays.asList("a1", "a2", "b2", "b1", "c1");
    maListe.stream()
           .filter(s -> s.startsWith("b"))
           .map(String::toUpperCase)
           .sorted()
           .forEach(System.out::println);
  }
}
```

Résultat :

```
B1
B2
```

Il existe deux types d'opérations : les opérations intermédiaires et les opérations terminales. L'ensemble des opérations effectuées par un Stream est appelé pipeline d'opérations (operation pipeline).

La plupart des opérations d'un Stream attendent en paramètre une interface fonctionnelle pour définir leur comportement. Ces paramètres peuvent ainsi être exprimés en utilisant une expression lambda ou une référence de méthode.

Une opération peut être avec ou sans état.

Il est important que :

- Les opérations exécutées par le Stream ne modifient pas la source de données utilisée par le Stream
- Les données de la source de données ne doivent pas être modifiées durant leur traitement par le Stream car cela pourrait avoir un impact sur leur exécution

22.1.3.2. Les opérations pour définir les traitements d'un Stream

Les opérations d'un Stream permettent de décrire des traitements, potentiellement complexes, à exécuter sur un ensemble de données. Elles sont définies dans l'interface java.util.stream.Stream grâce à de nombreuses méthodes.

Il existe deux catégories d'opérations :

- Opérations intermédiaires : elles peuvent être enchaînées car elles renvoient un Stream
- Opérations terminales : elles renvoient une valeur différente d'un Stream (ou pas de valeur) et ferme le Stream à la fin de leur exécution

Les opérations intermédiaires ne réalisent aucun traitement tant que l'opération terminale n'est invoquée. Elles sont dites lazy ce qui permettra éventuellement d'effectuer des optimisations dans leur exécution comme car exemple de réaliser celle-ci dans une même itération.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.Arrays;
import java.util.List;
import static java.util.stream.Collectors.toList;

public class TestStream2 {

  public static void main(String[] args) {
    List<Integer> nombres = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);

    List<Integer> troisPremierNombrePairAuCarre =
      nombres.stream()
             .filter(n -> {
                 System.out.println("filter " + n);
                 return n % 2 == 0;
              })
             .map(n -> {
                 System.out.println("map " + n);
                 return n * n;
              })
             .limit(3)
             .collect(toList());

    System.out.println("");
    troisPremierNombrePairAuCarre.forEach(System.out::println);
  }
}
```

Résultat :

```
filter 1
filter 2
map 2
filter 3
filter 4
map 4
filter 5
filter 6
map 6
4
16
36
```

Dans l'exemple ci-dessus, plusieurs optimisations sont mises en oeuvre par le Stream. Tous les éléments ne sont pas parcourus. Les opérations ne sont invoquées que le nombre de fois requis sur des éléments pour obtenir le résultat :

- L'opération de filtre n'est invoquée que 6 fois
- L'opération de transformation n'est invoquée que 3 fois

Ceci est possible car l'opération limit(3) ne renvoie qu'un Stream qui contient au maximum le nombre d'éléments précisé en paramètre.

Les opérations sont regroupées en une seule itération qui traite chaque élément en lui appliquant le pipeline d'opérations : les opérations n'itèrent pas individuellement sur les éléments du Stream.

Ceci permet d'optimiser au mieux les traitements à réaliser pour obtenir le résultat de manière efficiente.

L'utilisation d'un Stream implique généralement trois choses :

- Une source qui va alimenter le Stream avec les éléments à traiter
- Un ensemble d'opérations intermédiaires qui vont décrire les traitements à effectuer
- Une opération terminale qui va exécuter les opérations et produire le résultat

Les opérations proposées par un Stream sont des opérations communes :

- Pour filtrer des données
- Pour rechercher une correspondance avec des éléments
- Pour transformer des éléments
- Pour réduire les éléments et produire un résultat

Pour filtrer des données, un Stream propose plusieurs opérations :

- filter(Predicate) : renvoie un Stream qui contient les éléments pour lesquels l'évaluation du Predicate passé en paramètre vaut true
- distinct() : renvoie un Stream qui ne contient que les éléments uniques (elle retire les doublons). La comparaison se fait grâce à l'implémentation de la méthode equals()
- limit(n) : renvoie un Stream que ne contient comme éléments que le nombre fourni en paramètre
- skip(n) : renvoie un Stream dont les n premiers éléments sont ignorés

Pour rechercher une correspondance avec des éléments, un Stream propose plusieurs opérations :

- anyMatch(Predicate) : renvoie un booléen qui précise si l'évaluation du Predicate sur au moins un élément vaut true
- allMatch(Predicate) : renvoie un booléen qui précise si l'évaluation du Predicate sur tous les éléments vaut true
- noneMatch(Predicate) : renvoie un booléen qui précise si l'évaluation du Predicate sur tous les éléments vaut false
- findAny() : renvoie un objet de type Optional qui encapsule un élément du Stream s'il existe
- findFirst() : renvoie un objet de type Optional qui encapsule le premier élément du Stream s'il existe

Pour transformer des données, un Stream propose plusieurs opérations :

- map(Function) : applique la Function fournie en paramètre pour transformer l'élément en créant un nouveau
- flatMap(Function) : applique la Function fournie en paramètre pour transformer l'élément en créant zéro, un ou plusieurs éléments

Pour réduire les données et produire un résultat, un Stream propose plusieurs opérations :

- reduce() : applique une Function pour combiner les éléments afin de produire le résultat
- collect() : permet de transformer un Stream qui contiendra le résultat des traitements de réduction dans un conteneur mutable

22.1.4. La différence entre une collection et un Stream

Il est tentant de voir un Stream comme une collection. Les notions de collections et de Streams concernent toutes les deux une séquence d'éléments. De manière grossière, une collection permet de stocker cette séquence d'éléments et un Stream permet d'exécuter des opérations sur cette séquence d'éléments.

Bien que les Collections et les Streams semblent avoir des similitudes, ils ont des objectifs différents :

- Les collections permettent de gérer et de récupérer des données qu'elles stockent
- Les Streams assurent l'exécution lazy de traitements déclarés sur des données d'une source

Les Streams diffèrent donc de plusieurs manières des Collections :

- Un Stream n'est pas une structure qui stocke des données. Elle permet de traiter les différents éléments d'une source en appliquant différentes opérations
- Un Stream permet de mettre en oeuvre la programmation fonctionnelle : le résultat d'une opération ne doit pas modifier l'élément sur lequel elle opère ni aucun autre élément de la source. Par exemple, une opération de filtre ne doit pas retirer des éléments de la source mais créer un nouveau Stream qui contient uniquement les éléments filtrés
- De nombreuses opérations d'un Stream attendent en paramètre une expression lambda
- Il n'est pas possible d'accéder à un élément grâce à un index : seul le premier peut être obtenu ou il faut exporter le résultat sous la forme d'une collection ou d'un tableau
- Il est possible de paralléliser l'exécution des opérations. Cette exécution parallèle utilise alors le framework Fork/Join
- Un Stream n'a pas forcement de taille fixe : le nombre d'éléments à traiter peut potentiellement être infini. Ils peuvent consommer des données jusqu'à ce qu'une condition soit satisfaite : des méthodes comme limit() ou findFirst() peuvent alors permettre de définir une condition d'arrêt
- Une opération ne peut consommer qu'une seule fois un élément. Un nouveau Stream doit être recréer pour traiter de nouveau un élément comme le fait un Iterator
- Les Streams ne stockent pas de données. Ils transportent et utilisent les données issues d'une source qui les stockent ou les génèrent
- Les Streams exécutent un pipeline d'opérations sur les données de leur source
- Les Streams sont de nature fonctionnel : les opérations d'un Stream produisent des résultats mais ne devrait pas modifier les données de leur source
- Les opérations intermédiaires des Streams sont exécutées de manière lazy. Ce mode de fonctionnement permet d'effectuer des optimisations lors de l'exécution en un seul passage du pipeline d'opérations

Une collection est une structure qui stocke un ensemble de données en mémoire. Les traitements sur une collection, sans utiliser les Streams, nécessitent de réaliser une itération sur tout ou partie des éléments de la collection par exemple en utilisant l'instruction for.

Exemple (code Java 5.0) :

```
List<Personne> personnes = // ... code pour obtenir une liste de personnes
List<Long> ids = new ArrayList<>(personnes.size());
for(Personne p: personnes){
  ids.add(p.getId());
}
```

Avec un Stream, les opérations à réaliser sont décrites mais l'itération ou les itérations pour exécuter ces traitements sont réalisées en interne lors de l'exécution de la méthode terminale du Stream.

Exemple (code Java 8) :

```
    List<Personne> personnes = // ... code pour obtenir une liste de personnes
    List<Long> ids = personne.stream()
                             .map(Personne::getId)
                             .collect(toList());
```

Le code ci-dessus décrit simplement les opérations à réaliser par le Stream. La façon dont ces traitements seront exécutés est à la charge de l'implémentation du Stream.

Dans les deux exemples, le résultat est le même.

Les collections sont des structures en mémoire qui permettent de stocker des données : toutes les valeurs doivent être ajoutées à la collection avant de pouvoir les traiter. Un Stream ne stocke pas de données : il va obtenir les données à traiter à la demande à partir d'une source. Il effectue des traitements sur les données de cette source sur la base de l'exécution d'un pipeline d'opérations.

Le fait que l'API Stream réalise elle-même en interne l'itération sur les éléments de la source de données permet que la plupart des opérations soient exécutés de manière lazy. Ceci permet à l'API de réaliser éventuellement des optimisations.

Un Stream est un consommable : il consomme les données de la source une fois que son opération terminale est invoquée.

Un Stream peut assurer un traitement séquentiel ou en parallèle de ces traitements : le traitement en parallèle peut être très intéressant d'un point de vue performance notamment dans le cas du traitement d'un gros volume de données.

22.1.5. L'obtention d'un Stream

La création d'un Stream se fait nécessairement à partir d'une source de données. Cette source va permettre de fournir les différents éléments à la demande du Stream pour pouvoir les traiter.

Les Streams peuvent utiliser des données issues d'une source finie ou infinie d'éléments.

Il existe différentes façons de construire un Stream à partir de différentes sources :

- Collection
- Tableau
- Ensemble de données
- Fichier
- ...

Il est aussi possible d'utiliser une Function pour produire un nombre infini d'éléments comme source de données.

Chacune des classes pouvant servir de source propose une ou plusieurs méthodes pour créer un Stream :

Méthode	Source
stream() ou parallelStream() d'une Collection	Les éléments de la collection Stream<String> chaines = Arrays.asList("a1", "a2", "a3").stream();

Stream<T> Arrays.stream(T[])	Les éléments du tableau
	String[] valeurs = {"a1", "a2", "a3"};
	Stream<String> chaines = Arrays.stream(valeurs);

Stream<T> Stream.of(T)	L'élément
Stream<T> Stream.of(T.)	Les éléments passés en paramètre dans le varargs Stream<Integer> stream = Stream.of(1,2,3,4);
IntStream.range(int, int)	Les valeurs entières incluse entre les bornes inférieure et supérieure exclue fournies
IntStream.rangeClosed(int, int)	Les valeurs entières incluse entre les bornes inférieures et supérieures fournies
Stream<T> Stream.iterate(T, UnaryOperator<T>)	Des valeurs générées en appliquant successivement la fonction à partir de la valeur initiale fournie en paramètres Stream<Integer> stream = Stream.iterate(0, (i) -> i + 1);
Stream.empty()	Aucun élément
Stream.generate(Supplier<T>)	Un nombre infini d'éléments créés par le Supplier fourni en paramètre Stream<UUID> stream = Stream.generate(UUID::randomUUID);
Stream<String> BufferedReader.lines()	Les lignes d'un fichier texte
Stream<String> Files.lines	Les lignes d'un fichier texte
IntStream Random.ints()	Un nombre infini d'entiers aléatoires
IntStream BitSet.stream()	Les indices des bits positionnés à 1 dans le BitSet

Stream<String> Pattern.splitAsStream(CharSequence)	Les chaînes de caractères issues de l'application du motif
	String str = "chaine1,chaine2,chaine3";
	Pattern.compile(",")
	.splitAsStream(str)
	.forEach(System.out::println);

ZipFile.stream()
JarFile.stream()
	Les éléments contenus dans l'archive

IntStream String.chars	Les caractères de la chaîne Remarque le Stream obtenu est de type IntStream IntStream stream = "abcdef".chars();

Il est également possible que des sources soient proposées par des bibliothèques tierces ou soient développées par ses propres soins.

22.1.5.1. La création d'un Stream à partir de ses fabriques

L'interface Stream propose plusieurs fabriques pour créer un Stream.

La méthode of() de l'interface Stream attend en paramètre un varargs d'objets.

Exemple (code Java 8) :

```
    Stream<Integer> stream = Stream.of(new Integer[] { 1, 2, 3});
```

Attention, ce sont des objets qui sont attendus. Il n'est donc pas possible de fournir en paramètre un tableau de type entier primitif.

Exemple (code Java 8) :

```
    Stream<Integer> stream = Stream.of(new int[] { 1, 2, 3});
```

Résultat :

```
C:\java\workspace\TestStreams\src>javac com/jmdoudoux/dej/streams/TestStream.java
com\jmdoudoux\dej\streams\TestStream.java:9:
error: incompatible types: inference variable T has incompatible bounds
    Stream<Integer> stream1 = Stream.of(new int[] { 1, 2, 3 });
                                       ^
    equality constraints: Integer
    lower bounds: int[]
  where T is a type-variable:
    T extends Object declared in method <T>of(T)
1 error
```

Il est cependant possible de passer en paramètre de la méthode of() de l'interface IntStream un tableau d'entiers primitifs.

Exemple (code Java 8) :

```
    IntStream stream = IntStream.of(new int[] { 1, 2, 3});
```

Les méthodes iterate() et generate() de l'interface Stream attendent en paramètre une Function. Les éléments générés sont calculés à la demande en invoquant la fonction pour générer la prochaine valeur. Sans contrainte particulière, cette génération se poursuit à l'infini. Dans ce cas, on parle de Stream infini : c'est un Stream dont la source ne connait pas le nombre d'éléments qui sera traité à l'avance.

Par exemple, la méthode iterate() attend en paramètre une valeur initiale et un UnaryOperator qui sera invoqué pour générer chaque nouvelle valeur.

Exemple (code Java 8) :

```
    Stream<Integer> entiers = Stream.iterate(0, n -> n + 1);
    entiers.forEach(System.out::println);
```

Ce traitement va incrémenter une valeur entière et l'afficher sur la console.

Il est bien sûr possible de définir une condition d'arrêt des traitements du Stream basée sur le nombre d'éléments traités en utilisant la méthode limit() avec en paramètre le nombre souhaité.

Exemple (code Java 8) :

```
    Stream<Integer> entiers = Stream.iterate(0, n -> n + 1);
    entiers.limit(10)
           .forEach(System.out::println);
```

22.1.5.2. L'obtention d'un Stream à partir d'une collection

Les traitements sur les données d'une collection doivent être réalisés en itérant explicitement sur chacun des éléments. Avec les Streams l'itération est effectué par les traitements de l'API.

L'interface Collection propose deux méthodes pour obtenir un Stream dont la source sera la collection :

Méthode	Rôle
default Stream<E> stream()	

	Renvoyer un Stream dont les traitements seront exécutés de manière séquentielle sur les éléments de la collection
default Stream<E> parallelStream()	Renvoyer un stream dont les traitements sont exécutés en parallèle sur les éléments de la collection

Un Stream peut donc être obtenu à partir de n'importe quelle implémentation de l'interface Collection en utilisant la méthode stream().

Exemple (code Java 8) :

```
List<String> elements = new ArrayList<String>();
elements.add("element1");
elements.add("element2");
elements.add("element3");
Stream<String> stream = elements.stream();
```

22.2. Le pipeline d'opérations d'un Stream

Un Stream exécute une combinaison d'opérations pour former un pipeline.

Les opérations sont des méthodes définies dans l'interface Stream :

- Les méthodes intermédiaires : map(), mapToInt(), flatMap(), mapToDouble(), filter(), distinct(), sorted(), peek(), limit(), skip(), parallel(), sequential(), unordered()
- Les méthodes terminales : forEach(), forEachOrdered(), toArray(), reduce(), collect(), min(), max(), count(), anyMatch(), allMatch(), noneMatch(), findFirst(), findAny(), iterator()

La plupart des opérations d'un Stream attendent en paramètres une interface fonctionnelle qui sont généralement définies dans le package java.util.function. Cela permet d'utiliser une expression Lambda pour décrire les traitements qui seront réalisés par l'opération.

Le traitement de données par un Stream se fait en deux étapes :

- Configuration en invoquant des méthodes intermédiaires
- Exécution des traitements en invoquant la méthode terminale

Remarque : aucun traitement n'est effectué lors de l'invocation des méthodes intermédiaires. Ces méthodes sont dites lazy.

Dès qu'une méthode terminale est invoquée, le Stream est considéré comme consommé et plus aucune de ces méthodes ne peut être invoquée. Une méthode terminale peut produire une ou plusieurs valeurs ou opérer des effets de bord (forEach).

L'exemple ci-dessous créé une liste des 5 premières personnes dont le nom commence par un 'A'.

Exemple (code Java 8) :

```
List<String> prenomsCommencantParA = personnes
    .stream()
    .map(Personne::getNom)
    .filter(nom->nom.startsWith("a"))
    .limit(5)
    .collect(Collectors.toList());
```

Cet exemple utilise trois opérations intermédiaires :

- map() : transforme chaque élément de type Personne pour renvoyer uniquement le nom
- filter() : permet de limiter les éléments pour ne conserver que ceux qui commencent par 'A'
- limit() : arrête les traitements dès que le Stream contient 5 éléments

L'opération terminale collect() transforme les résultats de l'exécution dans une collection de type List.

Les opérations d'un Stream sont réparties en deux catégories :

- Les opérations intermédiaires (intermediate operation) : zéro, une ou plusieurs opérations intermédiaires par Stream. Elles produisent un Stream et sont toujours évaluées de manière lazy
- Les opérations terminales (terminal operation) : une seule opération terminale par Stream. Elles produisent une valeur ou des effets de bord

Une opération intermédiaire produit et renvoie un objet de type Stream. Ces opérations peuvent donc être chaînées pour définir le pipeline d'opérations.

Le traitement de ces méthodes n'est pas exécuté tant qu'une méthode terminale n'est pas invoquée. L'invocation d'une méthode intermédiaire permet simplement d'ajouter l'opération au pipeline : seule l'invocation de l'opération terminale va lancer la récupération des données et leur utilisation au travers du pipeline d'opérations.

Chaque opération intermédiaire :

- Renvoie un nouveau Stream : elle produit des éléments qui seront utilisés par l'opération suivante mais ne peut jamais produire le résultat final d'un Stream
- Opère de manière lazy : renvoie un nouveau Stream qui contiendra les éléments résultant de l'exécution de l'opération. Le parcours des éléments de la source ne commence pas avant l'invocation de la méthode terminale

La plupart de ces opérations intermédiaires d'un Stream fonctionnent en mode lazy : il diffère le plus tardivement ses traitements jusqu'à ce que les résultats soient nécessaires. Le mode lazy permet des optimisations, par exemple : il n'est pas nécessaire de parcourir tous les éléments pour trouver le premier qui correspondent à un critère donné. Ce type d'optimisation est particulièrement utile notamment si le nombre d'éléments à traiter est important

Il est important de comprendre que les opérations intermédiaires s'exécutent en mode lazy. Leur invocation ne provoque aucun traitement : c'est l'invocation de la méthode terminale qui provoque le traitement des données par les opérations du Stream.

Ce mode de fonctionnement possède plusieurs avantages :

- Généralement permet l'invocation des opérations en une passe
- Ceci est particulièrement utile lorsque la quantité de données à traiter par le Stream est importante
- L'API Stream peut optimiser les opérations exécutées notamment en utilisant des opérations de type short-circuiting. Ce type d'opération peut interrompre les traitements du Stream lorsqu'une condition est satisfaite

Pour illustrer ce comportement, l'exemple ci-dessous utilise un Stream sur une collection de prénoms pour les mettre en majuscule, ne conserver que ceux qui commencent par un 'A' et ne prendre que les deux premiers pour les afficher.

Exemple (code Java 8) :

```
  prenoms.stream()
         .map(String::toUpperCase)
         .filter(p -> p.startsWith("A"))
         .limit(2)
         .forEach(System.out::println);
```

Résultat :

```
ANDRE
ALBERT
```

Bien que l'opération limit() soit la dernière opération intermédiaire, elle a une influence sur l'exécution. Une fois que la condition est atteinte (deux prénoms commençant par A) alors les traitements effectués par le Stream sont interrompus. Ceci est possible car le pipeline d'opérations est appliqué sur chaque donnée.

Exemple (code Java 8) :

```
  public static void main(String[] args) {
    List<String> prenoms = Arrays.asList("andre", "benoit", "albert", "thierry", "alain",
      "jean");
    prenoms.stream()
           .map(p -> {
              afficher("map    ", p);
              return p.toUpperCase();})
           .filter(p -> {
                afficher("filter ", p);
                 return p.startsWith("A");})
           .peek(p -> {
              afficher("limit ", p);})
           .limit(2)
           .forEach(System.out::println);
  }

  public static void afficher(String message, String p) {
    System.out.println(message + " : " + p);
  }
```

Résultat :

```
map    : andre
filter : ANDRE
limit  : ANDRE
ANDRE
map    : benoit
filter : BENOIT
map    : albert
filter : ALBERT
limit  : ALBERT
ALBERT
```

Cette façon d'exécuter les traitements permet d'améliorer les performances notamment lorsque la quantité de donnée est importante. Au lieu de réaliser la transformation et le filtre sur toutes les données pour ne conserver que les trois premières, elles sont appliquées sur les données jusqu'à ce que la condition soit respectée.

Un pipeline d'opérations doit impérativement se terminer par une opération terminale.

Les opérations terminales lancent les traitements du Stream pour produire le résultat de l'exécution du pipeline d'opérations ou des effets de bords sur les données consommées :

- Renvoyer le résultat, généralement sous la forme d'une instance de type Optional<T> (elles ne renvoient jamais un Stream)
- Créer une collection qui contient les résultats
- Exécuter des traitements sur les résultats (méthode forEach())

Une fois qu'une méthode terminale est invoquée, elle lance les traitements et le Stream ne pourra alors plus être utilisé. Une fois que l'opération terminale est exécutée, le Stream est considéré comme consommé et il ne peut plus être utilisé. Si les données de la source doivent de nouveau être parcourues, il faut obligatoirement recréer un nouveau Stream à partir de la source.

Les opérations terminales ont pour rôle de terminer le pipeline d'opérations :

- Elles lancent l'exécution des traitements
- Elles produisent le résultat
- Et elles ferment le Stream

Certaines méthodes intermédiaires (limit(), skip(), ...) ou terminales (anyMatch(), allMatch(), noneMatch(), findFirst(), findAny(), ...) sont de type « short-circuiting » :

- Pour des opérations intermédiaires ce sont des opérations qui produisent un Stream dont le nombre d'éléments est fini à partir d'un Stream dont le nombre d'éléments est potentiellement infini
- Pour des opérations terminales ce sont des opérations qui peuvent se terminer dans un temps fini à partir d'un Stream dont le nombre d'éléments est potentiellement infini

Si le pipeline contient une opération de type short-circuiting, les méthodes intermédiaires suivantes ne seront exécutées que lorsque la méthode short circuit pourra être évaluée.

Avoir une opération de type short-circuit dans le pipeline du Stream est une condition nécessaire mais pas toujours suffisante pour permettre de terminer l'exécution d'un Stream dont le nombre d'éléments est infini.

22.3. Les opérations intermédiaires

Une opération intermédiaire renvoit toujours un Stream.

Le pipeline d'opérations d'un Stream peut avoir zéro, une ou plusieurs opérations intermédiaires.

Les opérations intermédiaires peuvent être réparties en deux groupes :

- Les opérations stateless ne conserve pas d'état lors du traitement d'un élément par rapport au précédent. Chaque élément est donc traité indépendamment des autres.
- Les opérations stateful conserve un état par rapport aux éléments traités précédemment lors du traitement d'un élément. Certaines opérations stateful doivent traiter l'ensemble des éléments avant de pouvoir créer leur résultat. Ceci peut avoir des conséquences lors de l'utilisation d'opérations stateful dans un Stream avec traitements en parallèle : cela peut nécessiter plusieurs itérations sur les données.

L'API Stream définit plusieurs opérations intermédiaires :

Opération		Rôle
filter	Stateless	Stream<T> filter(Predicate<? super T> predicate) Filtrer tous les éléments pour n'inclure dans le Stream de sortie que les éléments qui satisfont le Predicat
map	Stateless	<R> Stream<R> map(Function<? super T,? extends R> mapper) Renvoyer un Stream qui contient le résultat de la transformation de chaque élément de type T en un élément de type R
mapToxxx(Int, Long or Double)	Stateless	xxxStream mapToxxx(ToxxxFunction<? super T> mapper) Renvoyer un Stream qui contient le résultat de la transformation de chaque élément de type T en un type primitif xxx
flatMap	Stateless	<R> Stream<R> flatMap(Function<T,Stream<? extends R>> mapper) Renvoyer un Stream avec l'ensemble des éléments contenus dans les Stream<R> retournés par l'application de la Function sur les éléments de type T. Ainsi chaque élément de type T peut renvoyer zéro, un ou plusieurs éléments de type R.
flatMapToxxx (Int, Long or Double)		xxxStream flatMapToxxx(Function<? super T,? extends xxxStream> mapper) Renvoyer un Stream avec l'ensemble des éléments contenus dans les xxxStream retournés par l'application de la Function sur les éléments de type T. Ainsi chaque élément de type T peut renvoyer zéro, un ou plusieurs éléments de type primitif xxx.
distinct	Stateful	Stream<T> distinct() Renvoyer un Stream<T> dont les doublons ont été retirés. La détection des doublons se fait en invoquant la méthode equals()
sorted	Stateful	Stream<T> sorted() Stream<T> sorted(Comparator<? super T>) Renvoyer un Stream dont les éléments sont triés dans un certain ordre. La surcharge sans paramètre tri dans l'ordre naturel : le type T doit donc

		implémenter l'interface Comparable car c'est sa méthode compareTo() qui est utilisée pour la comparaison des éléments deux à deux La surcharge avec un Comparator l'utilise pour déterminer l'ordre de tri.
peek	Stateless	Stream<T> peek(Consumer <? super T>) Renvoyer les éléments du Stream et leur appliquer le Consumer fourni en paramètre
limit	Stateful Short-Circuiting	Stream<T> limit(long) Renvoyer un Stream qui contient au plus le nombre d'éléments fournis en paramètre
skip	Stateful	Stream<T> skip(long) Renvoyer un Stream dont les n premiers éléments ont été ignorés, n correspondant à la valeur fournie en paramètre
sequential		Stream<T> sequential() Renvoyer un Stream équivalent dont le mode d'exécution des opérations est séquentiel
parallel		Stream<T> parallel() Renvoyer un Stream équivalent dont le mode d'exécution des opérations est en parallèle
unordered		Stream<T> parallel() Renvoyer un Stream équivalent dont l'ordre des éléments n'a pas d'importance
onClose		Stream<T> onClose(Runnable) Renvoyer un Stream équivalent dont le handler fourni en paramètre sera exécuté à l'invocation de la méthode close(). L'ordre d'exécution de plusieurs handlers est celui de leur déclaration. Tous les handlers sont exécutés même si un handler précédent à lever une exception.

Les opérations sans état (stateless) comme map() ou filter() renvoient simplement le résultat de l'exécution sur l'élément courant dans le Stream pour être traité par l'opération suivante.

Les opérations avec état (stateful) peuvent avoir besoin de conserver des informations relatives au traitement des éléments précédent voir de tous les éléments pour pouvoir produire le Stream contenant les éléments à traiter par l'opération suivante.

22.3.1. Les méthodes map(), mapToInt(), mapToLong et mapToDouble()

Une opération de mapping permet de réaliser une transformation des éléments traités par le Stream.

Parfois, il est nécessaire de transformer les éléments d'un Stream. La méthode map() permet d'opérer une transformation en passant chacun des éléments du Stream à la Function fournie en paramètre. Le résultat est la production d'un nouveau Stream contenant le résultat de l'application de la fonction à chaque élément. C'est une transformation de type un pour un.

L'interface Stream<T> propose plusieurs opérations pour faire des transformations de type map.

Méthode	Rôle

<R> Stream<R> map(Function<? super T,? extends R> mapper)	Renvoyer un Stream qui contient le résultat de l'application de la fonction sur chaque élément du Stream. T est le type des éléments du Stream et R est le type des éléments résultat
DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)	Renvoyer un DoubleStream contenant le résultat de l'application de la fonction passé en paramètre à tous les éléments du Stream
IntStream mapToInt(ToIntFunction<? super T> mapper)	Renvoyer un IntStream contenant le résultat de l'application de la fonction passé en paramètre à tous les éléments du Stream
LongStream mapToLong(ToLongFunction<? super T> mapper)	Renvoyer un LongStream contenant le résultat de l'application de la fonction passé en paramètre à tous les éléments du Stream

La méthode map(Function< ? super T, ? super R>) attend en paramètre une Function ayant en argument un objet de type T et renvoie un objet de type R. Cette Function va être appliquée sur tous les éléments du Stream pour créer pour chacun un nouvel élément qui sera inclus dans le Stream en résultat. L'implémentation de l'interface fonctionnelle fournie en paramètre doit être :

- Sans état : aucune information relative aux éléments traités précédemment ne doit être conservée
- Ne pas modifier les éléments ou la source de données. Ceci est particulièrement vrai pour les Stream parallèles

Important :la modification de la source de données durant l'exécution du pipeline d'opérations peut conduire à un comportement ou un résultat non désiré.

Le type de la valeur de retour de l'opération map() peut être du même type que la valeur traitée ou d'un type différent.

Exemple (code Java 8) :

```
    Stream<String> chaines = Stream.of("aaa", "bbb", "ccc");
    chaines.map(chaine -> chaine.toUpperCase())
           .forEach(System.out::println);
```

Résultat :

```
AAA
BBB
CCC
```

Les traitements de transformations peuvent être le résultat de l'exécution d'une méthode indiquée grâce à une référence de méthode comme pour toute expression Lambda qui ne consiste qu'à invoquer une méthode.

Exemple (code Java 8) :

```
    Stream<String> chaines = Stream.of("aaa", "bbb", "ccc");
    chaines.map(String::toUpperCase)
           .forEach(System.out::println);
```

Résultat :

```
AAA
BBB
CCC
```

Cette méthode peut aussi être une méthode contenant des traitements métiers de l'application.

Exemple (code Java 8) :

```
    Long[] idEmployes = { 12345L, 67890L };
    List<Employe> listeEmployes = Stream
        .of(idEmployes)
        .map(EmployeeService::rechercherParId)
        .collect(Collectors.toList());
    System.out.println(listeEmployes);
```

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.Random;

public class EmployeeService {

  private static Random rnd = new Random();

  public static Employe rechercherParId(Long id) {
    return new Employe("nom " + id,
       (id % 2) == 0 ? Genre.HOMME : Genre.FEMME,
       150 + rnd.nextInt(50),
       1000 + rnd.nextInt(2000));
  }
}
```

Résultat :

```
[Employe [nom=nom 12345, genre=FEMME, taille=168,
salaire=2573.0], Employe [nom=nom 67890, genre=HOMME, taille=187, salaire=1374.0]]
```

La Function peut produire des éléments de types différents du type de l'élément traité.

Exemple (code Java 8) :

```
personnes.stream().map(Personne::getNom).forEach(System.out::println);
```

Dans l'exemple ci-dessus, la méthode stream() renvoie un Stream<Personne> sur lequel la méthode map() est appliquée. La Function qui lui est passée est une invocation de la méthode getNom() : la méthode map() renvoie un Stream<String> dont les éléments sont les noms des personnes. Enfin la méthode forEach() exécute le Consumer qui affiche tous les noms sur la console.

Plusieurs transformations peuvent être appliquées dans le pipeline d'opérations

Exemple (code Java 8) :

```
    Long[] idEmployes = { 12345L, 67890L };
    List<String> listeNomsEmployes = Stream
        .of(idEmployes)
        .map(EmployeeService::rechercherParId)
        .map(e -> e.getNom())
        .collect(Collectors.toList());
    System.out.println(listeNomsEmployes);
```

Résultat :

```
[nom 12345, nom 67890]
```

La méthode mapToInt() applique une Function à chaque élément pour produire un IntStream à la place d'un Stream<Integer>

La méthode mapToLong() est similaire à la méthode mapToInt() mais elle renvoie un LongStream.

La méthode mapToDouble() est similaire à la méthode mapToInt() mais elle renvoie un DoubleStream.

Lorsque le résultat de la transformation renvoie une valeur entière ou flottante, il est possible d'utiliser la méthode map() qui va renvoyer un Stream du type du wrapper de la valeur primitive.

Exemple (code Java 8) :

```
  Double[] valeurs = { 1.0, 2.0, 3.0, 4.0, 5.0 };
  Double[] valeursAuCarre = Stream.of(valeurs).map(v -> v * v).toArray(Double[]::new);
  System.out.println(Arrays.deepToString(valeursAuCarre));
```

Résultat :

```
[1.0, 4.0, 9.0, 16.0, 25.0]
```

Mais l'utilisation des méthodes mapToDouble(), mapToInt() et mapToLong() améliore les performances essentiellement car elles évitent des opérations coûteuses de boxing et unboxing. Ceci est particulièrement vrai pour très grande quantité d'éléments.

Exemple (code Java 8) :

```
  Double[] valeurs = { 1.0, 2.0, 3.0, 4.0, 5.0 };
  double[] valeursAuCarre = Stream.of(valeurs)
                                .mapToDouble(v -> v * v)
                                .toArray();
  System.out.println(Arrays.toString(valeursAuCarre));
```

Résultat :

```
[1.0, 4.0, 9.0, 16.0, 25.0]
```

22.3.2. La méthode flatMap()

Les opérations map() et flatMap() servent toutes les deux à réaliser des transformations mais il existe une différence majeure :

- La méthode map() renvoie un seul élément
- La méthode flatMap() renvoie un Stream qui peut contenir zéro, un ou plusieurs éléments

L'opération map() permet de transformer un élément du Stream mais elle possède une limitation : le résultat de la transformation doit obligatoirement produire un unique élément qui sera ajouté au Stream renvoyé.

L'exemple ci-dessous créé une paire de valeur (valeur -1 et valeur) à partir d'une collection de nombres. La valeur retournée est ainsi une List de List d'Integer.

Exemple (code Java 8) :

```
  List<Integer> nombres = Arrays.asList(1, 3, 5, 7, 9);
  List<List<Integer>> tuples =
      nombres.stream()
             .map(nombre -> Arrays.asList(nombre - 1, nombre))
             .collect(Collectors.toList());
  System.out.println(tuples);
```

Résultat :

```
[[0, 1], [2, 3], [4, 5], [6, 7], [8, 9]]
```

Le même exemple que précédemment mais utilisant une opération flatMap() renvoie une List d'Integer dont toutes les valeurs ont été mises à plat.

Exemple (code Java 8) :

```
  List<Integer> nombres = Arrays.asList(1, 3, 5, 7, 9);
  List<Integer> nombresDesTuples =
      nombres.stream()
```

```
             .flatMap(nombre -> Arrays.asList(nombre - 1, nombre).stream())
             .collect(Collectors.toList());
  System.out.println(nombresDesTuples);
```

Résultat :

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

L'opération flatMap() transforme chaque élément en un Stream d'autres éléments : ainsi chaque élément va être transformé en zéro, un ou plusieurs autres éléments. Le contenu des Streams issus du résultat de la transformation de chaque objet est agrégé pour constituer le Stream retourné par la méthode flatMap().

L'opération flatMap() attend en paramètre une Function qui renvoie un Stream. La Function est appliquée sur chaque élément qui produit chacun un Stream. Le résultat renvoyé par la méthode est un Stream qui est l'agréation de tous les Streams produits par les invocations de la Function.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.ArrayList;
import java.util.List;

public class Departement {
  private String nom;
  private List<Etudiant> etudiants = new ArrayList<>();

  public Departement(String nom) {
    super();
    this.nom = nom;
  }

  public String getNom() {
    return nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

  public List<Etudiant> getEtudiants() {
    return etudiants;
  }
}
```

Exemple (code Java 8) :

```
    departements.stream()
                .flatMap(d -> d.getEtudiants().stream())
                .forEach(System.out::println);
```

L'exemple ci-dessous affiche tous les mots d'un fichier texte.

Exemple (code Java 8) :

```
    try {
      Files.lines(Paths.get("fichier.txt"))
           .map(ligne -> ligne.split("\\s+"))
           .flatMap(Arrays::stream)
           .map(mot -> mot.replaceAll("[^A-Za-z]+", ""))
           .distinct()
           .forEach(System.out::println);
    } catch (IOException e) {
      e.printStackTrace();
    }
```

La méthode lines() renvoie un Stream<String> où chaque élément est une ligne du fichier. La première méthode map() renvoie un Stream<String[]> où chaque élément est un tableau des mots d'un ligne. La méthode flatMap() renvoie un Stream<String> où chaque élément est un mot du fichier

L'exemple ci-dessous créé un Stream à partir d'un Stream de List en utilisant la méthode flatMap().

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;

public class TestFlatMap {

  public static void main(String[] args) {
    Stream<List<String>> listesIngredients = Stream.of(Arrays.asList("carottes",
"poireaux", "navets"), Arrays.asList("eau"), Arrays.asList("sel", "poivre"));
    Stream<String> ingredients = listesIngredients.flatMap(strList -> strList.stream());
    ingredients.forEach(System.out::println);
  }
}
```

Résultat :

```
carottes
poireaux
navets
eau
sel
poivre
```

22.3.3. La méthode filter()

Il est fréquent de vouloir appliquer un filtre sur les éléments d'un Stream pour ne conserver qu'un sous-ensemble de ces éléments, ceux répondant à un certain critère. L'opération filter() permet de ne conserver que les éléments du Stream qui respecte une certaine condition.

L'interface Stream<T> propose la méthode filter() pour filtrer les éléments du stream :

Stream<T> filter(Predicate<? super T> predicate)

La méthode filter() attend en paramètre une interface fonctionnelle de type Predicate : elle permet de filter les éléments du Stream selon la condition fournie en paramètre par l'interface fonctionnelle de type Predicate. Le Predicate est appliqué sur chaque élément du Stream : s'il renvoie true, élément est ajouté dans le Stream de sortie, sinon l'élément est ignoré.

Exemple (code Java 8) :

```
    Listb<String> prenoms = Arrays.asList("andre", "benoit" "albert", "thierry", "alain",
      "jean");
    prenoms.stream()
           .filter(p -> p.startsWith("a"))
           .forEach(System.out::println);
```

Résultat :

```
andre
albert
alain
```

Il est possible d'utiliser plusieurs opérations filter() dans un même pipeline d'opérations.

Exemple (code Java 8) :

```
    List<String> prenoms = Arrays.asList("andre", "benoit","albert", "thierry", "alain",
      "jean");
    prenoms.stream()
           .filter(p -> p.startsWith("a"))
           .filter(p -> p.length() == 5)
           .forEach(System.out::println);
```

Résultat :

```
andre
alain
```

Il est cependant préférable de regrouper les filtres lorsque cela est possible.

Exemple (code Java 8) :

```
    List<String> prenoms = Arrays.asList("andre", "benoit","albert", "thierry", "alain",
      "jean");
    prenoms.stream()
           .filter(p -> p.startsWith("a") && (p.length() == 5))
           .forEach(System.out::println);
```

Résultat :

```
andre
alain
```

22.3.4. La méthode distinct()

L'opération distinct() permet de supprimer les doublons contenus dans un Stream. Les éléments contenus dans le Stream retournés sont donc uniques. La méthode distinct() n'attend aucun paramètre et renvoie un Steam avec les éléments dont les doublons ont été retirés.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.io.PrintStream;
import java.util.Arrays;
import java.util.List;

public class TestStream {

  public static void main(String[] args) {
    List<Integer> entiers = Arrays.asList(1, 2, 3, 1, 2, 3);
    entiers.stream()
           .distinct()
           .forEach(System.out::println);
  }
}
```

Résultat :

```
1
2
3
```

La détection des doublons est réalisée en utilisant la méthode equals() des éléments. Il est donc important que celle-ci soit correctement implémentée.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

public class MaClasse {

  private String nom;

  public MaClasse(String nom) {
    this.nom = nom;
  }

  public String getNom() {
    return nom;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj) {
      return true;
    }
    if (obj == null) {
      return false;
    }
    if (getClass() != obj.getClass()) {
      return false;
    }
    MaClasse other = (MaClasse) obj;
    if (nom == null) {
      if (other.nom != null) {
        return false;
      }
    } else if (!nom.equals(other.nom)) {
      return false;
    }
    return true;
  }

  // implémentation de la méthode hashCode() en correspondance
  // volontairement non fournie pour illustrer la problématique engendrée

  @Override
  public String toString() {
    return "MaClasse [nom=" + nom + "]";
  }
}
```

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.io.PrintStream;
import java.util.Arrays;
import java.util.List;

public class TestStream {

  public static void main(String[] args) {
    List<MaClasse> maClasses = Arrays.asList(new MaClasse("aaa"), new MaClasse("bbb"), new
MaClasse("aaa"));
    maClasses.stream()
             .distinct()
             .forEach(System.out::println);
  }
}
```

Résultat :

```
MaClasse [nom=aaa]
MaClasse [nom=bbb]
MaClasse [nom=aaa]
```

La méthode distinct() ne donne pas le résultat attendu bien que la méthode equals() soit implémentée. Cela est dû est fait qu'en Java, il faut implémenter la méthode hashCode() en correspondance avec l'implémentation de la redéfinition de la méthode equals().

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

public class MaClasse {

  private String nom;

  public MaClasse(String nom) {
    this.nom = nom;
  }

  public String getNom() {
    return nom;
  }

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = (prime * result) + ((nom == null) ? 0 : nom.hashCode());
    return result;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj) {
      return true;
    }
    if (obj == null) {
      return false;
    }
    if (getClass() != obj.getClass()) {
      return false;
    }
    MaClasse other = (MaClasse) obj;
    if (nom == null) {
      if (other.nom != null) {
        return false;
      }
    } else if (!nom.equals(other.nom)) {
      return false;
    }
    return true;
  }

  @Override
  public String toString() {
    return "MaClasse [nom=" + nom + "]";
  }
}
```

Résultat :

```
MaClasse [nom=aaa]
MaClasse [nom=bbb]
```

Cette opération intermédiaire est stateful : elle nécessite de stocker tous les éléments qui seront renvoyés : cela peut donc requérir une quantité de mémoire dépendante du nombre d'éléments traités par le Stream. Tous les éléments du Stream en entrée doivent être traités avant qu'un premier élement ne soit envoyés dans le Stream de retour.

Si le Stream est ordonné, l'élément conservé parmi les doublons est toujours le premier. Si le Stream n'est pas ordonné, l'élément conservé parmi les doublons n'est pas prédictible.

L'utilisation d'un Stream non ordonné ou l'utilisation de la méthode unordered() peut améliorer les performances lors de l'utilisation de distinct() dans un Stream parallèle, sous réserve que le contexte le permette. Si l'ordre doit être préservé, il arrive qu'un Stream séquentiel puisse être plus performant qu'un Stream parallèle équivalent.

22.3.5. La méthode limit()

L'opération intermédiaire limit() permet de limiter le nombre d'éléments contenu dans le Stream retourné. Le nombre d'éléments est passé en paramètre de la méthode limit().

C'est une opération de type short-circuit : dès que le nombre d'éléments contenus dans le Stream retourné est atteint, l'opération met fin à la consommation d'éléments de la source par le Stream.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.io.PrintStream;
import java.util.Arrays;
import java.util.List;

public class TestStream {

  public static void main(String[] args) {
    List<Integer> entiers = Arrays.asList(1, 2, 3, 4, 5, 6);
    entiers.stream()
           .limit(3)
           .forEach(System.out::println);
  }
}
```

Résultat :

```
1
2
3
```

22.3.6. La méthode skip()

L'opération skip() permet d'ignorer un certain nombre d'éléments du Stream. Les premiers éléments du Stream sont ignorés, les autres sont ajoutés dans le Stream retourné par la fonction. Si le nombre d'éléments fournis en paramètre est supérieur ou égal au nombre d'éléments du Stream, alors le Stream retourné est vide.

La méthode skip() attend en paramètre une valeur entière qui correspond au nombre d'éléments à ignorer.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.io.PrintStream;
import java.util.Arrays;
import java.util.List;

public class TestStream {

  public static void main(String[] args) {
    List<Integer> entiers = Arrays.asList(1, 2, 3, 4, 5, 6);
    entiers.stream()
           .skip(3)
           .forEach(System.out::println);
  }
}
```

Résultat :

```
4
```

```
5
6
```

22.3.7. La méthode sorted()

L'opération sorted() permet de trier les éléments du Stream : elle renvoie donc un Stream dont les éléments sont triés. Elle utilise un tampon qui doit avoir tous les éléments avant de pouvoir effectuer le tri.

Par défaut, la méthode sorted() sans paramètre utilise l'ordre naturel des éléments.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.Arrays;
import java.util.List;

public class TestStream {

  public static void main(String[] args) {
    List<String> prenoms = Arrays.asList("andre", "benoit", "albert", "thierry", "alain");
    prenoms.stream()
           .sorted()
           .forEach(System.out::println);
  }
}
```

Résultat :

```
alain
albert
andre
benoit
thierry
```

Une surcharge de la méthode sorted() attend en paramètre un Comparator qui sera utilisé pour comparer les éléments deux à deux lors du tri.

Exemple (code Java 8) :

```
   List<String> prenoms = Arrays.asList("andre", "benoit","albert", "thierry", "alain");
   prenoms.stream()
          .sorted(Comparator.reverseOrder())
          .forEach(System.out::println);
```

Résultat :

```
thierry
benoit
andre
albert
alain
```

Il est possible de construire des Comparator évolués en utilisant les méthodes static et par défaut de l'interface Comparator.

Exemple (code Java 8) :

```
   List<String> prenoms = Arrays.asList("andre", "benoit", "albert", "thierry", "alain");
   prenoms.stream()
          .sorted(Comparator.comparingInt(String::length)
                            .thenComparing(Comparator.naturalOrder())
                            .reversed())
          .forEach(System.out::println);
```

Résultat :

```
thierry
benoit
lbert
andre
alain
```

22.3.8. La méthode peek()

L'opération peek() renvoie un Stream contenant tous les éléments du Stream courant en appliquant le Consumer fournit en paramètre sur chacun des éléments.

Le but de cette opération est essentiellement le débogage, par exemple pour afficher l'élément en cours de traitement.

Exemple (code Java 8) :

```
   List<String> prenoms = Arrays.asList("andre", "benoit", "albert", "thierry", "alain",
      "jean");
   prenoms.stream()
          .map(p -> {
            afficher("map   ", p);
            return p.toUpperCase();
          })
          .filter(p -> {
             afficher("filter ", p);
             return p.startsWith("A");
          })
          .peek(p -> {
            afficher("limit ", p);
          })
          .limit(2)
          .forEach(System.out::println);
```

Résultat :

```
map : andre
filter : ANDRE
limit  : ANDRE
ANDRE
map    : benoit
filter : BENOIT
map    : albert
filter : ALBERT
limit  : ALBERT
ALBERT
ANDRE
```

Important : le Consumer ne devrait pas modifier l'élément ni la source du Stream.

22.3.9. Les méthodes qui modifient le comportement du Stream

Plusieurs méthodes sont des opérations intermédiaires qui modifient certaines caractéristiques du Stream et donc peuvent avoir un impact sur la manière dont le Stream va traiter les données.

La méthode sequential() permet de demander l'exécution des traitements du Stream en séquentiel dans un seul thread.

La méthode parallel() permet de demander l'exécution des traitements du Stream en parallèle en utilisant plusieurs threads du pool Fork/Join.

Les méthodes sequential() et parallel() sont des opérations intermédiaires : elles permettent donc simplement de configurer les traitements qui seront effectivement exécutés par l'invocation de la méthode terminale. Les traitements ainsi exécutés ne peuvent l'être dans leur intégralité qu'en séquentiel ou en parallèle même si les méthodes sequential() et parallel() sont invoqués dans le Stream.

Exemple (code Java 8) :

```
    Stream.of(1, 3, 4, 2)
          .parallel()
          .sorted()
          .peek((v) -> System.out.println(Thread.currentThread()
              .getName() + " " + v))
          .sequential()
          .forEach((v) -> System.out.println(Thread.currentThread()
             .getName() + " " + v));
```

Résultat :

```
main 1
main 1
main 2
main 2
main 3
main 3
main 4
main 4
```

C'est l'état indiqué par la dernière méthode sequential() ou parallel() qui sera utilisée pour déterminer le mode d'exécution des traitements du Stream.

Exemple (code Java 8) :

```
    Stream.of(1, 3, 4, 2)
          .sequential()
          .sorted()
          .peek((v) -> System.out.println(Thread.currentThread()
              .getName() + " " + v))
          .parallel()
          .forEach((v) -> System.out.println(Thread.currentThread()
              .getName() + " " + v));
```

Résultat :

```
ForkJoinPool.commonPool-worker-2
4
ForkJoinPool.commonPool-worker-3
2
main
3
ForkJoinPool.commonPool-worker-1
1
main
3
ForkJoinPool.commonPool-worker-3
2
ForkJoinPool.commonPool-worker-2
4
ForkJoinPool.commonPool-worker-1 1
```

La méthode unordered() permet de retirer le flag ORDERED pour indiquer au Stream que les données ne sont pas ordonnées.

Le flag ORDERED du Stream permet de préciser que les éléments sont ordonnées : il peut être positionné par la source (par exemple ArrayList ou un tableau) ou par une opération intermédiaire (par exemple sorted()). Une opération terminale peut ignorer le flag ORDERED (par exemple forEach()). La plupart des opérations intermédiaires traitent les éléments en respectant leur ordre.

Lors de l'exécution des opérations en séquentiel, avoir les éléments dans un certain ordre n'a pas d'influence sur les performances : il affecte uniquement le déterminisme du résultat en fournissant toujours le même avec la même source de données. Si les éléments ne sont pas ordonnés, différentes exécutions sur la même source peuvent donner des résultats différents.

Lors de l'exécution des opérations en parallèle, certaines opérations intermédiaires stateful ou terminales peuvent s'exécuter avec de meilleures performances (exemple distinct() ou Collectors.groupingBy()). Dans ce cas, si l'ordre des éléments n'est pas important, il peut être utile d'utiliser la méthode unordered() du Stream.

Par exemple, la méthode limit() est généralement peu coûteuse lorsque les traitements sont effectués en séquentiel mais peut être très coûteuse sur des données ordonnés avec une exécution en parallèle. Si cela n'altère pas les autres opérations, l'utilisation de la méthode unoredred() peut améliorer les performances dans des traitements en parallèle. Si l'ordre des éléments doit être conservé, l'exécution des traitements en séquentiel peut aussi améliorer les performances notamment celles de la méthode limit().

22.4. Les opérations terminales

Les traitements d'un Stream sont démarrés à l'invocation de son unique opération terminale.

Une seule opération terminale ne peut être invoquée sur un même Stream : une fois cela fait, le Stream ne sera plus utilisable.

Il n'est pas possible de réutiliser un Stream une fois que son opération terminale est invoquée. Le Stream est alors considéré comme consommé. Il faut obligatoirement obtenir un nouveau Stream à partir de la source pour chaque traitement. Il n'est possible que d'invoquer une seule fois un pipeline d'opérations sur un Stream sinon une exception de type IllegalStateException est levée.

Exemple (code Java 8) :

```
String[] fruits = { "orange", "citron", "pamplemousse", "banane","fraise",
        "groseille", "raisin", "pomme", "poire", "abricot", "cerise", "peche", "clementine" };

Stream<String> stream = Stream.of(fruits);
stream.filter(s -> s.startsWith("p"))
      .forEach(System.out::println);
try {
  stream.filter(s -> s.startsWith("c"))
        .forEach(System.out::println);
} catch (IllegalStateException e) {
  e.printStackTrace(System.out);
}
```

Cela ne pose pas de soucis pour créer un nouveau Stream à chaque fois puisque le Stream pointe simplement sur sa source de données mais ne les copie pas. Il est par exemple possible d'utiliser un Supplier pour créer des instances d'un Stream avec les mêmes éléments.

Exemple (code Java 8) :

```
Supplier<Stream<String>> streamSupplier = () -> Stream.of(fruits);
streamSupplier.get()
              .filter(s -> s.startsWith("p"))
              .forEach(System.out::println);
streamSupplier.get()
              .filter(s -> s.startsWith("c"))
              .forEach(System.out::println);
```

Contrairement aux opérations intermédiaires qui renvoient toujours un Stream, les opérations terminales renvoient une valeur qui correspond au résultat de l'exécution du pipeline d'opérations sur les données. Ce résultat peut être :

- Une valeur d'un type primitif
- Un élément ou une instance de type Optional

- Une collection ou un tableau d'éléments
- void

Il est possible qu'il n'y ait pas de résultat à l'issu des traitements d'un Stream. Java 8 propose la classe Optional qui encapsule une valeur ou l'absence de valeur.

Certaines opérations d'un Stream renvoie donc un objet de type java.util.Optional qui permet de préciser s'il y a un résultat ou non. C'est notamment le cas dans les opérations de réduction, des opérations de recherche (findXXX) ou de recherche de correspondance (XXXMatch) sur un Stream vide.

L'API Stream propose plusieurs opérations terminales dont les principales sont :

Méthode	Rôle
forEach	void forEach(Consumer<? super T> action) Exécuter le Consumer sur chacun des éléments du Stream
forEachOrdered	void forEachOrdered(Consumer<? super T> action) Exécuter le Consumer sur chacun des éléments du Stream en respectant l'ordre de éléments si le Stream en défini un
toArray	Object[] toArray() Renvoyer un tableau contenant les éléments du Stream <A> A[] toArray(IntFunction<A[]> generator) Renvoyer un tableau contenant les éléments du Stream : le tableau est créé par la fonction fournie
Reduce	Optional<T> reduce(BinaryOperator<T> accumulator) Réaliser une opération de réduction qui accumule les différents éléments du Stream grâce à la fonction fournie T reduce(T identity, BinaryOperator<T> accumulator) Réaliser une opération de réduction qui accumule à partir de la valeur fournie les différents éléments du Stream grâce à la fonction <U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner) Réaliser une opération de réduction avec les fonctions fournies en paramètres
collect	<R,A> R collect(Collector<? super T,A,R> collector) Réaliser une opération de réduction avec le Collector fourni en paramètre <R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner) Réaliser une opération de réduction avec les fonctions fournies en paramètres
min	Optional<T> min(Comparator<? super T> comparator) Renvoyer le plus petit élément du Stream selon le Comparator fourni
max	Optional<T> max(Comparator<? super T> comparator) Renvoyer le plus grand élément du Stream selon le Comparator fourni
count	long count() Renvoyer le nombre d'éléments contenu dans le Stream

anyMatch	boolean anyMatch(Predicate<? super T> predicate) Retourner un booléen qui indique si au moins un élément valide le Predicate
allMatch	boolean allMatch(Predicate<? super T> predicate) Retourner un booléen qui indique si tous les éléments valident le Predicate
noneMatch	boolean noneMatch(Predicate<? super T> predicate) Retourner un booléen qui indique si aucun élément ne valide le Predicate
findFirst	Optional<T> findFirst() Retourner un Optional qui encapsule le premier élément validant le Predicate s'il existe
findAny	Optional<T> findAny() Retourner un Optional qui encapsule élément validant le Predicate s'il existe
iterator	Iterator<T> iterator() Renvoyer un Iterator qui permet de réaliser une itération sur tous les éléments en dehors du Stream
spliterator	Spliterator<T> spliterator() Renvoyer un Spliterator pour les éléments du Stream

Certaines de ces méthodes sont de type short-circuiting, par exemple findFirst().

Il est possible d'exporter les éléments d'un Steam dans une collection ou un tableau en utilisant certaines de ses méthodes :

Méthode	Rôle

collect(Collectors.toList())	Obtenir une collection de type List qui contient les éléments du Stream
	Exemple (code Java 8) :
	`Stream<Integer> intStream = Stream.of(1,2,3);`
	`List<Integer> intList = intStream.collect(Collectors.toList());`
	`System.out.println(intList);`

toArray(TypeDonnees[]::new)	Obtenir un tableau qui contient les éléments du Stream
	Exemple (code Java 8) :
	`Stream<Integer> intStream = Stream.of(1, 2, 3);`
	`Integer[] intArray = intStream.toArray(Integer[]::new);`
	`System.out.println(Arrays.deepToString(intArray));`

Une opération de réduction permet de traiter les éléments du Stream de manière itérative pour produire un résultat unique. Des opérations de réduction typique sur des entiers sont par exemple le calcul de la somme ou de la moyenne ou bien encore la détermination de la plus petite ou la plus grande valeur.

Une opération de réduction utilise une valeur initiale (identity) pour la combiner avec le premier élément. Le résultat est ensuite combiné avec le second élément et ainsi de suite. Les traitements utilisés pour réaliser la combinaison sont fournis sous la forme d'une fonction (accumulator).

L'application répétée de cette combinaison permet de générer le résultat qui peut prendre plusieurs formes :

- Un résultat unique (la somme, la moyenne, la plus petite/grande valeur, ...)
- Une collection (List, Set, ...) qui contient les éléments
- Une Map qui contient des paires clé/valeur extraites des données du Stream

L'API Stream propose des opérations de réduction :

- Spécialisée : count(), max(), min(), ...
- Générique : reduce(), collect()

22.4.1. Les méthodes forEach() et forEachOrdered()

L'API Stream propose deux opérations pour exécuter des traitements sur les éléments du résultat de l'exécution des opérations intermédiaires :

- void forEach(Consumer<? super T> action) : cette opération exécute l'action passée en paramètre sur chacun des éléments du Stream. Le comportement de la méthode forEach() sur un Stream parallèle n'est pas déterministe : elle ne garantit pas dans ce cas le respect de l'ordre des éléments puisque ceux-ci sont traités par différents threads
- void forEachOrdered(Consumer<? super T> action) : cette opération est similaire à l'opération forEach() mais elle garantit l'ordre des éléments du Stream. Pour cela, elle n'utilise qu'un seul thread pour exécuter l'action sur tous les éléments. C'est l'API qui détermine quel unique thread exécute l'action sur chacun des éléments

Ces deux méthodes attendent en paramètre un java.util.function.Consumer qui contient l'action à réaliser sur chacun des éléments.

Remarque : il n'est pas possible d'utiliser les instructions break ou return dans l'expression lambda pour interrompre l'itération sur les éléments.

L'exemple ci-dessous illustre les différences lors de l'utilisation de ces méthodes :

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;

public class TestStream {

  public static void main(String[] args) {
    List<String> prenoms = Arrays.asList("andre", "benoit", "albert", "thierry", "alain");
    Consumer<String> afficherElement = s -> System.out.println(s + " - " +
      Thread.currentThread().getName());
    prenoms.stream().sorted().forEach(afficherElement);
    System.out.println();
    prenoms.parallelStream().sorted().forEach(afficherElement);
    System.out.println();
    prenoms.parallelStream().sorted().forEachOrdered(afficherElement);
  }
}
```

Résultat :

```
alain - main
albert - main
andre - main
benoit - main
thierry - main
andre - main
thierry - ForkJoinPool.commonPool-worker-3
benoit - main
alain - ForkJoinPool.commonPool-worker-1
```

```
albert - ForkJoinPool.commonPool-worker-2
alain - ForkJoinPool.commonPool-worker-2
albert - ForkJoinPool.commonPool-worker-3
andre - ForkJoinPool.commonPool-worker-3
benoit - ForkJoinPool.commonPool-worker-3
thierry - ForkJoinPool.commonPool-worker-3
```

Lors de l'invocation de la méthode forEach() sur un Stream séquentiel, l'ordre des éléments n'est préservé. Bien que la méthode forEach() ne respecte aucun ordre, le parcours d'un Stream séquentiel se fait dans un unique thread ce qui permet de conserver l'ordre des éléments.

Lors de l'invocation de la méthode forEach() sur un Stream parallèle, l'ordre des éléments est pas préservé. Ceci est lié au fait de l'utilisation de plusieurs threads pour exécuter l'action sur les différents éléments.

Pour respecter l'ordre des éléments dans un Stream parallèle, il est nécessaire d'utiliser la méthode forEachOrdered() qui va réaliser l'exécution de l'action dans un thread unique. Evidemment les performances de la méthode forEachOrdered() sont moins bonnes que celles de la méthode forEach().

Il faut éviter d'utiliser ces méthodes. C'est particulièrement vrai si celles-ci effectuent des traitements qui modifient une donnée.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.atomic.DoubleAdder;

public class TestEmployeStream {

  public static void main(String[] args) {
    List<Employe> employes = new ArrayList<>(6);
    employes.add(new Employe("e1", Genre.HOMME, 176, 1500));
    employes.add(new Employe("e2", Genre.HOMME, 190, 2700));
    employes.add(new Employe("e3", Genre.FEMME, 172, 1850));
    employes.add(new Employe("e4", Genre.FEMME, 162, 3300));
    employes.add(new Employe("e5", Genre.HOMME, 176, 1280));
    employes.add(new Employe("e6", Genre.FEMME, 168, 2850));

    DoubleAdder total = new DoubleAdder();
    employes.stream()
            .forEach(e -> total.add(e.getSalaire()));

    System.out.println("total salaire = " + total.doubleValue());
  }
}
```

Résultat :

```
total salaire = 13480.0
```

Attention : lors de l'utilisation sur un Stream parallèle, si l'action d'une opération forEach() modifie une ressource partagée, il est nécessaire de gérer explicitement les accès concurrents.

Cette approche n'est cependant pas fonctionnelle : il préférable d'utiliser une approche de type map / reduce. L'interface DoubleStream propose d'ailleurs une méthode sum() qui effectue ce traitement.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.ArrayList;
import java.util.List;
```

```
public class TestEmployeStream {

  public static void main(String[] args) {
    List<Employe> employes = new ArrayList<>(6);
    employes.add(new Employe("e1", Genre.HOMME, 176, 1500));
    employes.add(new Employe("e2", Genre.HOMME, 190, 2700));
    employes.add(new Employe("e3", Genre.FEMME, 172, 1850));
    employes.add(new Employe("e4", Genre.FEMME, 162, 3300));
    employes.add(new Employe("e5", Genre.HOMME, 176, 1280));
    employes.add(new Employe("e6", Genre.FEMME, 168, 2850));

    Double total = employes.stream()
                           .mapToDouble(e -> e.getSalaire())
                           .sum();

    System.out.println("total salaire = " + total);
  }
}
```

Résultat :

```
total salaire = 13480.0
```

Ces méthodes impliquent nécessairement des effets de bords puisqu'elle ne renvoie rien (void).

Pour respecter une approche fonctionnelle, il est préférable d'éviter de modifier la source de données ou ses éléments.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.ArrayList;
import java.util.List;

public class TestEmployeStream {

  public static void main(String[] args) {
    List<Employe> employes = new ArrayList<>(6);
    employes.add(new Employe("e1", Genre.HOMME, 176, 1500));
    employes.add(new Employe("e2", Genre.HOMME, 190, 2700));
    employes.add(new Employe("e3", Genre.FEMME, 172, 1850));
    employes.add(new Employe("e4", Genre.FEMME, 162, 3300));
    employes.add(new Employe("e5", Genre.HOMME, 176, 1280));
    employes.add(new Employe("e6", Genre.FEMME, 168, 2850));

    employes.stream()
            .forEach(e -> e.setSalaire(e.getSalaire() + (e.getSalaire() * 0.1)));
    employes.stream()
            .forEach(System.out::println);
  }
}
```

Résultat :

```
Employe [nom=e1, genre=HOMME, taille=176, salaire=1650.0]
Employe [nom=e2, genre=HOMME, taille=190, salaire=2970.0]
Employe [nom=e3, genre=FEMME, taille=172, salaire=2035.0]
Employe [nom=e4, genre=FEMME, taille=162, salaire=3630.0]
Employe [nom=e5, genre=HOMME, taille=176, salaire=1408.0]
Employe [nom=e6, genre=FEMME, taille=168, salaire=3135.0]
```

Une meilleure approche dans ce cas, pourrait être de ne pas utiliser un Stream mais d'utiliser la méthode forEach() de la collection.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;
```

```
import java.util.ArrayList;
import java.util.List;

public class TestEmployeStream {

  public static void main(String[] args) {
    List<Employe> employes = new ArrayList<>(6);
    employes.add(new Employe("e1", Genre.HOMME, 176, 1500));
    employes.add(new Employe("e2", Genre.HOMME, 190, 2700));
    employes.add(new Employe("e3", Genre.FEMME, 172, 1850));
    employes.add(new Employe("e4", Genre.FEMME, 162, 3300));
    employes.add(new Employe("e5", Genre.HOMME, 176, 1280));
    employes.add(new Employe("e6", Genre.FEMME, 168, 2850));

    employes.forEach(e -> { e.setSalaire(e.getSalaire() + (e.getSalaire() * 0.1));
                               System.out.println(e);
                    });
  }
}
```

Le résultat de l'exécution est le même que pour l'exemple précédent.

Les méthodes forEach() et forEachOrdered() sont des opérations terminales : une fois leur exécution terminée, le Stream est consommé et ne peux plus être utilisé. Il n'est donc pas possible d'invoquer deux fois ces méthodes sur un même Stream.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.stream.Stream;

public class TestStream {

  public static void main(String[] args) {
    Stream<Integer> nombres = Stream.of(1, 2, 3, 4, 5);
    nombres.forEach(i -> System.out.println(i));
    nombres.forEach(System.out::println);
  }
}
```

Résultat :

```
1
2
3
4
5
Exception
in thread "main" java.lang.IllegalStateException: stream has already
been operated upon or closed
      at java.util.stream.AbstractPipeline.sourceStageSpliterator(AbstractPipeline.java:279)
      at
java.util.stream.ReferencePipeline$Head.forEach(ReferencePipeline.java:580)
      at
fr.jmdoudoux.dej.streams.TestStream.main(TestStream.java:22)
```

Dans ce cas, il est possible de combiner les deux traitements dans la même expression lambda ou utiliser la méthode peek().

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.stream.Stream;

public class TestStream {
  public static void main(String[] args) {
    Stream<Integer> nombres = Stream.of(1, 2, 3, 4, 5);

    nombres.peek(i -> System.out.println(i))
           .forEach(System.out::println);
  }
}
```

Résultat :

```
1
1
2
2
3
3
4
4
5
5
```

22.4.2. La méthode collect()

Les opérations de réductions ne renvoient pas toujours un résultat sous la forme d'une valeur unique mais sous la forme d'une structure de données comme une collection par exemple. Dans ce cas, il faut utiliser l'opération collect() qui permet de réaliser une réduction dans un conteneur mutable.

La méthode collect() de l'interface Stream est une opération terminale qui permet de réaliser une agrégation mutable des éléments. Cette agrégation peut prendre différentes formes : stocker les éléments dans une structure de données telle qu'une collection, concaténer les éléments qui sont des chaînes de caractères, ...

L'opération collect() permet de transformer les éléments d'un Stream pour produire un résultat sous différentes formes telles que List, Set, Map, ... ou plus généralement sous la forme d'une réduction dans un conteneur mutable.

Méthode	Rôle
<R,A> R collect(Collector<? super T,A,R> collector)	Effectuer une opération de réduction mutable sur les éléments du Stream en utilisant le Collector fourni
<R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)	Effectuer une opération de réduction mutable sur les éléments du Stream en utilisant les fonctions fournies en paramètres

Les traitements réalisés par la méthode collect() peuvent être exécutés en parallèle sur plusieurs threads dans un Stream parallèle car chacun d'eux va utiliser sa propre instance du conteneur et la méthode de combinaison permet de fusionner le contenu des différents conteneurs.

La surcharge <R> collect(Supplier<R> resultSupplier, BiConsumer<R, T> accumulator, BiConsumer<R, R> combiner) de la méthode collect() attend trois paramètres :

- supplier : pour fournir une instance du conteneur vide
- accumulator : pour ajouter un élément dans le conteneur
- combiner : pour combiner deux conteneurs lors de l'utilisation du Stream en parallèle

L'exemple ci-dessous utilise la méthode collect() pour mettre tous les éléments du Stream dans une collection de type HashSet grâce aux trois fonctions fournies en paramètre :

- supplier : renvoie une nouvelle instance de type HashSet
- accumulator : ajoute l'élément dans la collection
- combiner : ajoute tous les éléments d'un HashSet dans l'autre

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.Arrays;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class TestCollectors {

  public static void main(String[] args) {
    List<String> elements = Arrays.asList("elem1", "elem2", "elem2", "elem3", "elem4");
    Set<String> ensemble = elements.stream()
                                   .collect(() -> new HashSet<String>(),
                                           (s, e) -> s.add(e),
                                           (s1, s2) -> s1.addAll(s2));
    System.out.println(ensemble);
  }
}
```

Résultat :

```
[elem1, elem2, elem3, elem4]
```

Evidemment, comme les expressions correspondent toutes à l'invocation d'une seule méthode, il est possible d'utiliser des références de méthodes pour obtenir le même résultat.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.Arrays;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class TestCollectors {

  public static void main(String[] args) {
    List<String> elements = Arrays.asList("elem1", "elem2", "elem2", "elem3", "elem");
    Set<String> ensemble = elements.stream()
                                  .collect(HashSet::new,
                                            HashSet::add,
                                            HashSet::addAll);
    System.out.println(ensemble);
  }
}
```

Résultat :

```
[elem1, elem2, elem3, elem4]
```

La mise en oeuvre de ces deux exemples est préférable à l'exemple ci-dessous.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.Arrays;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class TestCollectors {

  public static void main(String[] args) {
    List<String> elements = Arrays.asList("elem1", "elem2", "elem2", "elem3", "elem4");
    // NE PAS FAIRE COMME CELA
    Set<String> ensemble = new HashSet<>();
    elements.stream()
            .forEach(s -> ensemble.add(s));
    System.out.println(ensemble);
  }
}
```

Résultat :

```
[elem1, elem2, elem3, elem4]
```

Cet exemple fonctionne mais c'est un antipattern. Il fonctionne correctement uniquement en mono thread.

La méthode collect() peut aussi être utilisée pour effectuer une opération de réduction qui va permettre de combiner deux éléments pour obtenir un nouvel élément qui sera combiné à son tour à l'élément suivant et ainsi de suite jusqu'à ce que tous les éléments soient traités.

L'exemple ci-dessous concatène les éléments qui sont de type chaîne de caractères.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.Arrays;
import java.util.List;

public class TestCollectors {

  public static void main(String[] args) {
    List<String> elements = Arrays.asList("elem1", "elem2", "elem2", "elem3", "elem4");
    StringBuilder elmtSb =
      elements.stream()
              .collect(() -> new StringBuilder(),
                       (sb, s) -> sb.append(s),
                       (sb1, sb2) -> sb1.append(sb2));
    System.out.println(elmtSb.toString());
  }
}
```

Résultat :

```
elem1elem2elem2elem3elem4
```

Comme chaque fonction est l'invocation d'une seule méthode, il est possible d'utiliser des références de méthodes.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.Arrays;
import java.util.List;

public class TestCollectors {

  public static void main(String[] args) {
    List<String> elements = Arrays.asList("elem1", "elem2", "elem2", "elem3", "elem4");
    StringBuilder elmtSb =
      elements.stream()
              .collect(StringBuilder::new, StringBuilder::append, StringBuilder::append);
    System.out.println(elmtSb.toString());
  }
}
```

Résultat :

```
elem1elem2elem2elem3elem4
```

Fréquemment, il est nécessaire d'ajouter un séparateur entre chacun des éléments concaténés.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.Arrays;
import java.util.List;

public class TestCollectors {

  public static void main(String[] args) {
    List<String> elements = Arrays.asList("elem1", "elem2", "elem2", "elem3", "elem4");
    StringBuilder concat =
      elements.stream()
              .collect(() -> new StringBuilder(), (sb, s) -> sb.append(";")
              .append(s),(sb1, sb2) -> sb1.append(sb2));
    System.out.println(concat.toString());
  }
}
```

Résultat :

```
;elem1;elem2;elem2;elem3;elem4
```

Evidemment cette solution simple possède l'inconvénient d'avoir le séparateur en trop en première ou dernière position selon l'ordre de concaténation du séparateur et de l'élément courant. Il est alors possible d'écrire un bloc de code pour conditionner l'ajouter du séparateur ou non.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.Arrays;
import java.util.List;

public class TestCollectors {

  public static void main(String[] args) {
    List<String> elements = Arrays.asList("elem1", "elem2", "elem2", "elem3", "elem4");
    StringBuilder concat =
      elements.stream()
              .collect(() -> new StringBuilder(), (sb, s) -> {
                       if (sb.length() != 0) {
                         sb.append(";");
                       }
                         sb.append(s);
                       } , (sb1, sb2) -> sb1.append(sb2));
    System.out.println(concat.toString());
  }
}
```

Résultat :

```
elem1;elem2;elem2;elem3;elem4
```

Cette solution utilise un bloc code ce qui la rend plus complexe et empêche l'utilisation d'une référence de méthode. Pour simplifier le code, il est possible d'utiliser la classe StringJoiner introduite dans Java 8.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;
```

```
import java.util.Arrays;
import java.util.List;
import java.util.StringJoiner;

public class TestCollectors {

  public static void main(String[] args) {
    List<String> elements = Arrays.asList("elem1", "elem2", "elem2", "elem3", "elem4");
    StringJoiner elmtJoiner =
      elements.stream()
              .collect(() -> new StringJoiner(";"),
                       (j, e) -> j.add(e),
                       (j1, j2) -> j1.merge(j2));
    System.out.println(elmtJoiner.toString());
  }
}
```

Résultat :

```
elem1;elem2;elem2;elem3;elem4
```

22.4.3. Les méthodes findFirst() et findAny(

L'interface Stream possède les méthodes findFirst() et findAny() pour obtenir un élément du Stream qui respecte le Predicate qui leur est fourni en paramètre.

Les méthodes findFirst() et findAny() sont des opérations terminales de type short-circuiting qui renvoient une instance de type Optional car il est possible que le Stream ne possède aucun élément et dans ce cas l'instance retournée encapsule l'absence de valeur.

La méthode findFirst() renvoie un objet de type Optional qui encapsule le premier élément dont le Predicate fourni en paramètre est validé.

Si aucun élément n'est trouvé, l'Optional retourné est vide.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

public class TestFindFirst {

  public static void main(String[] args) {
    List<Personne> personnes = new ArrayList<>(6);
    personnes.add(new Personne("p1", Genre.HOMME, 176));
    personnes.add(new Personne("p2", Genre.HOMME, 190));
    personnes.add(new Personne("p3", Genre.FEMME, 172));
    personnes.add(new Personne("p4", Genre.FEMME, 162));
    personnes.add(new Personne("p5", Genre.HOMME, 176));
    personnes.add(new Personne("p6", Genre.FEMME, 168));

    Optional<Personne> uneGrandePersonne =
        personnes.stream()
                 .filter(p -> p.getTaille() >= 250)
                 .findFirst();

    if (uneGrandePersonne.isPresent()) {
      System.out.println("Grande personne trouvee : " + uneGrandePersonne);
    } else {
      System.out.println("Aucune grande personne trouvee");
    }
  }
}
```

Résultat :

```
Aucune grande personne trouvee
```

L'opération findAny() renvoie n'importe quel élément du Stream dont le Predicate fourni en paramètre est validé.

L'utilisation de l'une ou l'autre de ces méthodes requière une attention particulière sur un Stream en parallèle.

La méthode findAny() est plus performante que la méthode findFirst() notamment lorsque le Stream est traité en mode parallèle.

22.4.4. Les méthodes xxxMatch()

L'API propose plusieurs méthodes xxxMatch() pour déterminer si aucun, au moins un ou tous les éléments respectent une certaine condition.

La méthode anyMatch(Predicate<? super T> predicate) renvoie un booléen qui précise si au moins un élément du Stream respecte le Predicate.

Exemple (code Java 8) :

```
    Stream<Integer> entiers = Stream.of(1, 2, 3, 4, 5);
    boolean auMoinsUnEgalATrois = entiers.anyMatch(e -> e == 3);
    System.out.println(auMoinsUnEgalATrois);
```

Résultat :

```
true
```

La méthode allMatch(Predicate<? super T> predicate) renvoie un booléen qui précise si tous les éléments du Stream respectent le Predicate.

Exemple (code Java 8) :

```
  Stream<Integer> entiers = Stream.of(1, 2, 3, 4, 5);
  boolean tousInferieursADix = entiers.allMatch(e -> e < 10);
  System.out.println(tousInferieursADix);
```

Résultat :

```
true
```

La méthode noneMatch(Predicate<? super T> predicate) renvoie un booléen qui précise si aucun élément du Stream respecte le Predicate.

Exemple (code Java 8) :

```
    Stream<Integer> entiers = Stream.of(1, 2, 3, 4, 5);
    boolean tousDifferentsDeDix = entiers.noneMatch(e -> e == 10);
    System.out.println(tousDifferentsDeDix);
```

Résultat :

```
true
```

22.4.5. La méthode count()

La méthode count() renvoie un entier long qui est le nombre d'éléments contenu dans le Stream.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.stream.Stream;

public class TestStream {

  public static void main(String[] args) {
    Stream<Integer> nombres = Stream.of(1, 2, 3, 4, 5);
    System.out.println("Nb elements=" + nombres.count());
  }
}
```

Résultat :

```
Nb elements=5
```

La méthode count() est une opération terminale donc elle effectue ces traitements sur le Stream issu de l'exécution des opérations intermédiaires.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.stream.Stream;

public class TestStream {

  public static void main(String[] args) {
    Stream<Integer> nombres = Stream.of(1, 2, 3, 4, 5).filter(e -> (e % 2) == 0);
    System.out.println("Nb elements=" + nombres.count());
  }
}
```

Résultat :

```
Nb elements=2
```

22.4.6. La méthode reduce

La réduction, aussi appelée folding en programmation fonctionnelle, permet de réaliser une opération d'agrégation sur un ensemble d'éléments afin de produire un résultat. Une opération de réduction applique de manière répétée une opération sur chacun des éléments pour les combiner afin de produire un unique résultat.

Par exemple, pour des éléments de type entier, si l'opérateur binaire fait une addition alors l'opération de réduction calcule la somme des éléments. Mais les traitements d'une réduction ne se limite pas à ce type de traitements. Une opération de réduction peut aussi être la recherche de la plus grande valeur en utilisant une expression lambda (x,y) -> Math.max(x,y) ou avec la référence de méthode équivalente Math::max.

Une réduction peut aussi s'appliquer sur tout type d'objet : dans ce cas, il est nécessaire de préciser le comportement des traitements de l'opération d'agrégation.

La méthode reduce() effectue une opération de type réduction. L'interface Stream<T> possède trois surcharges de la méthode reduce() :

Méthode	Rôle

Optional<T> reduce(BinaryOperator<T> accumulator)	Effectue la réduction des éléments du Stream en appliquant la fonction fournie en paramètre. Elle renvoie un Optional qui encapsule la valeur ou l'absence de valeur
T reduce(T identity, BinaryOperator<T> accumulator)	Effectue la réduction des éléments du Stream en appliquant la fonction fournie en paramètre à partir de la valeur fournie en premier paramètre
<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)	Effectue la réduction des éléments du Stream en appliquant la fonction d'accumulation et de combinaison fournies en paramètre à partir de la valeur fournie en premier paramètre

La surcharge qui attend en paramètre un BinaryOperator<T> renvoie comme résultat au plus un élément encapsulé dans un Optional. Elle attend en paramètre un opérateur binaire associatif qui effectue la réduction des éléments du Stream. Elle renvoie un Optional qui est vide si le Stream ne contient aucun élément. Si le Stream ne possède qu'un seul élément, alors c'est cet élément qui est retourné.

Exemple (code Java 8) :

```
    employes.stream()
            .reduce((p1, p2) -> p1.getTaille() > p2.getTaille() ? p1 : p2)
            .ifPresent(System.out::println);
```

L'exemple ci-dessus permet de déterminer la personne ayant la plus grande taille.

La fonction de type BinaryOperator<T> attend en paramètres deux objets de type T et renvoie un objet de T. Le premier paramètre est la valeur accumulée et le second paramètre est l'élément courant en cours de traitement dans le Stream.

Exemple (code Java 8) :

```
    String lettres = Stream.of("a", "b", "c", "d")
                           .reduce((accumulator, item) -> accumulator + ", " + item)
                           .orElse("");
    System.out.println(lettres);
```

Résultat :

```
a, b, c, d
```

L'opération de réduction peut utiliser un des BinaryOperator fournis par le JDK. L'exemple ci-dessous recherche la personne la plus grande.

Exemple (code Java 8) :

```
    Comparator<Employe> comparateurParTaille = Comparator.comparingInt(Employe::getTaille);
    BinaryOperator<Employe> lePlusGrand = BinaryOperator.maxBy(comparateurParTaille);
    Optional<Employe> plusGrandEmploye = employes.stream()
                                                 .reduce(lePlusGrand);
    System.out.println(plusGrandEmploye);
```

La seconde surcharge de la méthode reduce() attend en paramètre la valeur initiale et une fonction de type BinaryOperator qui sera invoquée pour effectuer la réduction.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.io.PrintStream;
import java.util.Arrays;
import java.util.List;
```

```
public class TestStream {

  public static void main(String[] args) {
    List<Integer> entiers = Arrays.asList(1, 2, 3, 4, 5, 6);
    Integer total = entiers.stream()
                           .reduce(0, (valeurAccumulee, valeur) -> valeurAccumulee + valeur);
    System.out.println("total=" + total);
  }
}
```

Résultat :

```
total=21
```

La fonction de type BinaryOperator peut être fournie sous la forme d'une référence de méthode. L'exemple ci-dessous détermine la valeur maximum d'éléments de type Integer.

Exemple (code Java 8) :

```
    Integer max = Stream.of(1, 2, 3, 4, 5)
                        .reduce(0, Integer::max);
    System.out.println(max);
```

Il faut prêter attention à la valeur initiale fournie.

Exemple (code Java 8) :

```
    String lettres = Stream.of("a", "b", "c", "d")
                           .reduce("", (accumulator, item) -> accumulator + ", " + item);
    System.out.println(lettres);
```

Résultat :

```
, a, b, c, d
```

Si le Stream ne contient aucun élément, alors c'est la valeur initiale qui est renvoyée.

Exemple (code Java 8) :

```
    String lettres = Stream.of("a", "b", "c", "d")
                           .limit(0)
                           .reduce("X", (accumulator, item) -> accumulator + ", " + item);
    System.out.println(lettres);
```

Résultat :

```
X
```

L'identity est la valeur initiale qui sera fournie en premier paramètre lors de la première invocation de l'opération d'accumulation.

Exemple (code Java 8) :

```
    int produit = Stream.of(1, 2, 3, 4, 5)
                        .reduce(0, Integer::max);
    System.out.println(produit);
```

Résultat :

```
5
```

Il est aussi possible de concaténer les éléments de type chaînes de caractères.

La valeur initiale est simplement retournée si le Stream ne contient aucun élément.

Il est important que l'agrégation de la valeur initiale avec un élément donne l'élément et vice versa que l'agrégation d'un élément avec la valeur initiale donne aussi l'élément.

Toutes les opérations n'ont pas de valeur initiale et lorsqu'elles en possèdent une elles ne permettent pas toujours d'avoir le résultat attendu. Par exemple, pour une opération de réduction qui détermine la plus grande valeur d'un ensemble d'entiers. Il peut sembler raisonnable d'utiliser la valeur Integer.MIN_VALUE comme identity. Cela fonctionne très bien si le Stream contient au moins un élément mais si le Stream ne contient aucun élément, la valeur retournée ne permet pas de déterminer si elle correspond au fait qu'il n'y a aucun élément ou si c'est vraiment la valeur du plus petit élément. C'est la raison pour laquelle l'interface Stream propose une surcharge qui renvoie un Optional et une autre surcharge qui attend en paramètre la valeur initiale.

Pour la concaténation de chaînes de caractères, la valeur initiale est une chaîne vide.

Exemple (code Java 8) :

```
    List<String> chaines = Arrays.asList("1", "2", "3", "4", "5");
    String chaine = chaines.stream().reduce("", String::concat);
    System.out.println(chaine);
```

Résultat :

```
12345
```

Pour la somme de valeurs entières, la valeur initiale est zéro.

Exemple (code Java 8) :

```
   List<Integer> entiers = Arrays.asList(1, 2, 3, 4, 5, 6);
   int somme = entiers.stream().reduce(0, (x, y) -> x + y);
   System.out.println(somme);
```

Résultat :

```
21
```

Remarque : il est préférable dans ce cas d'utiliser l'opération sum() de l'interface IntStream.

Parfois en fonction de l'opération d'agération, il faut prendre des précautions avec la valeur initiale.

Exemple (code Java 8) :

```
    int produit = Stream.of(1, 2, 3, 4, 5)
                        .reduce(0, (a, b) -> a * b);
    System.out.println(produit);
```

Résultat :

```
0
```

Ce n'est pas le résultat attendu car la valeur initiale est passée en premier paramètre de l'opération de réduction. Dans le cas où l'opération est une multiplication, il faut utiliser la valeur 1 comme valeur initiale.

Exemple (code Java 8) :

```
    int produit = Stream.of(1, 2, 3, 4, 5)
                        .reduce(1, (a, b) -> a * b);
    System.out.println(produit);
```

Résultat :

```
120
```

La troisième surcharge de la méthode reduce() attend en paramètre la valeur initiale, une BiFunction qui sera utilisée comme un accumulateur et une BinaryOperator qui sera utilisée comme combinateur.

Cette opération peut être définie explicitement de manière différente au moyen de deux opérations de type map() et reduce().

L'exemple ci-dessous calcule la taille des employés. Ce traitement aurait pu être réalisé différemment par le Stream mais il est utilisé pour illustrer le mode de fonctionnement de la méthode reduce().

Exemple (code Java 8) :

```
   Integer tailleTotale = employes.stream()
      .reduce(0, (somme, e) -> somme += e.getTaille(),
                 (somme1, somme2) -> somme1 + somme2);
  System.out.println("Taille totale = " + tailleTotale);
```

Résultat :

```
Taille totale = 1044
```

Le traitement exécuté est similaire à celui-ci-dessous.

Exemple (code Java 8) :

```
    BiFunction<Integer, Employe, Integer> accumulator = (somme, e) -> somme += e.getTaille();
    Integer resultat = 0;
    for (Employe e : employes) {
      resultat = accumulator.apply(resultat, e);
    }
  System.out.println("Taille totale = " + tailleTotale);
```

Résultat :

```
Taille totale = 1044
```

Pour comprendre le mode de fonction de la méthode reduce() et l'utilisation qu'elle fait des fonctions d'accumulation et de combinaison, il faut ajouter des traces dans leurs expressions Lambdas.

Exemple (code Java 8) :

```
   Integer tailleTotale = employes.stream().reduce(0, (somme, e) -> {
     afficher("accumulator", " somme=" + somme + " e=" + e.getTaille());
     return somme += e.getTaille();
   } , (somme1, somme2) -> {
     afficher("combiner", " somme1=" + somme1 + " somme2=" + somme2);
     return somme1 + somme2;
   });
   System.out.println("Taille totale = " + tailleTotale);
```

Résultat :

```
accumulator:
somme=0 e=176 [main]
accumulator:
somme=176 e=190 [main]
accumulator:
somme=366 e=172 [main]
accumulator:
```

```
somme=538 e=162 [main]
accumulator:
somme=700 e=176 [main]
accumulator:
somme=876 e=168 [main]
Taille totale = 1044
```

La fonction de combinaison n'est jamais invoquée lorsque le Stream est exécuté en séquentiel. Le comportement est différent lorsque le Stream est invoqué en parallèle.

Exemple (code Java 8) :

```
   Integer tailleTotale = employes.parallelStream().reduce(0, (somme, e) -> {
     afficher("accumulator", " somme=" + somme + " e=" + e.getTaille());
     return somme += e.getTaille();
   } , (somme1, somme2) -> {
     afficher("combiner", " somme1=" + somme1 + " somme2=" + somme2);
     return somme1 + somme2;
   });
   System.out.println("Taille totale = " + tailleTotale);
 }
```

Résultat :

```
accumulator:  somme=0 e=190
[ForkJoinPool.commonPool-worker-1]
accumulator:  somme=0 e=176
[ForkJoinPool.commonPool-worker-3]
accumulator:  somme=0 e=168
[ForkJoinPool.commonPool-worker-2]
accumulator:  somme=0 e=162 [main]
accumulator:  somme=0 e=176
[ForkJoinPool.commonPool-worker-3]
accumulator:  somme=0 e=172
[ForkJoinPool.commonPool-worker-1]
combiner:  somme1=176 somme2=168
[ForkJoinPool.commonPool-worker-3]
combiner:  somme1=190 somme2=172
[ForkJoinPool.commonPool-worker-1]
combiner:  somme1=162 somme2=344
[ForkJoinPool.commonPool-worker-3]
combiner:  somme1=176 somme2=362
[ForkJoinPool.commonPool-worker-1]
combiner:  somme1=538 somme2=506
[ForkJoinPool.commonPool-worker-1]
Taille totale = 1044
```

Le comportement des traitements est différent : comme l'accumulateur est invoqué en parallèle, ce n'est plus lui qui fait la somme mais la fonction de combinaison.

Une opération de réduction est particulièrement intéressante lors du traitement du Stream en parallèle. Lors de l'exécution des traitements en parallèle, il est nécessaire de partager une variable qui stocke la valeur cumulée et de gérer les accès concurrents à cette variable par les différents threads. Généralement, cette gestion des accès concurrents inhibe voire dégrade les performances gagnées par la parallélisation.

Pour limiter cet impact, l'opération reduce() d'un Stream, traite et cumule les éléments dans chaque threads : ainsi, il n'y a pas de variable partagée et donc aucune contention liée aux accès concurrents.

Les résultats de chaque threads sont ensuite combinés pour obtenir le résultat final.

Pour permettre à une opération de réduction d'être exécutée en parallèle, il est nécessaire que celle-ci soit associative.

Une opération # est associative si (a # b) # c = a # (b # c).

L'addition et la multiplication sont des opérations associatives. La soustraction n'est pas une opération associative :

(3 - 2) -1 = 0

3 - (2 -1) = 1

Exemple (code Java 8) :

```
     OptionalInt res = IntStream.range(1, 100)
                                .reduce((a, b) -> a - b);
     System.out.println(res.getAsInt());
```

Résultat :

```
-4948
```

Avec une opération associative, la réduction peut être effectuée dans n'importe quel ordre. Ceci est important pour une exécution en parallèle dans laquelle les éléments sont traités par lots, la réduction se faisant pour chacun des éléments des lots puis il y a une combinaison des résultats intermédiaires pour renvoyer le résultat de la réduction.

Si l'opération n'est pas associative, les résultats obtenus ne seront pas ceux escomptés.

Exemple (code Java 8) :

```
      OptionalInt res = IntStream.range(1, 100)
                                 .parallel()
                                 .reduce((a, b) -> a - b);
      System.out.println(res.getAsInt());
```

Résultat :

```
24
```

L'opération n'a pas besoin d'être commutative. Une opération # est commutative si a # b = b # a. L'addition et la multiplication sont des opérations commutatives. La soustraction n'est pas une opération commutative :

3 - 2 = 1

2 - 3 = -1

Un exemple d'opération de réduction qui est associative mais pas commutative est la concaténation de chaînes de caractères.

22.4.7. Les méthodes min() et max()

Les méthodes min() et max() permettent respectivement de retourner la valeur minimale et maximale issue des traitements du Stream.

Elles attendent en paramètre un Comparator qui permet de préciser l'ordre de comparaison des éléments.

Elles renvoient une instance de type Optional<T>

Le plus simple est d'utiliser la méthode comparing() de l'interface Comparator : elle renvoie un objet de type Comparator qui compare les clés extraites grâce à l'expression Lambda fournie en paramètre. Le paramètre de la méthode comparing() est une Function qui permet d'extraire la clé sur laquelle le Comparator retourné va faire la comparaison.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.Arrays;
import java.util.Comparator;
import java.util.List;
import java.util.Optional;

public class TestStream {

  public static void main(String[] args) {
    List<String> prenoms = Arrays.asList("andre", "benoit", "albert", "thierry", "alain",
    "jean");
    Optional<String> plusPetitPrenom =
      prenoms.stream().min(Comparator.comparing(element -> element.length()));
    System.out.println(plusPetitPrenom.orElseGet(() -> "aucun prenom trouve"));
  }
}
```

Résultat :

```
jean
```

22.4.8. La méthode toArray()

La méthode toArray() permet de renvoyer les éléments du Stream dans un tableau. Elle possède deux surcharges :

Méthode	Rôle
Object[] toArray()	Renvoyer un tableau contenant les éléments du Stream
<A> A[] toArray(IntFunction<A[]> generator)	Renvoyer un tableau contenant les éléments du Stream. Le tableau renvoyé est créé par la Function fournie en paramètre

Si le type du tableau n'est pas important, il est possible d'utiliser la méthode toArray() sans paramètre.

Exemple (code Java 8) :

```
    Stream<String> stream = Stream.of("a", "b", "c");
    Object[] strings = stream.toArray();
```

La Function attendue en paramètre de la surcharge de la méthode toArray() attend en paramètre un entier qui est la taille du tableau et renvoie un tableau dont la taille est celle passée en paramètre. Elle permet de préciser le type du tableau à utiliser et qui sera renvoyé par la méthode.

Exemple (code Java 8) :

```
Stream<String> stream = Stream.of("a", "b", "c");
String[] strings = stream.toArray(size -> new String[size]);
```

Il est aussi possible d'utiliser une référence de méthode, ce qui est plus simple et limite les possibilités d'erreurs.

Exemple (code Java 8) :

```
Stream<String> stream = Stream.of("a","b" "c");
String[] strings = stream.toArray(String[]::new);
```

Ce code correspond à celui-ci-dessous qui utilise une classe anonyme interne :

Exemple (code Java 8) :

```
Stream<String> stream = Stream.of("a", "b", "c");
String[] strings = stream.toArray(new IntFunction<String[]>() {
    @Override
```

```
        public String[] apply(int > size) {
       return new String[size];
    }
});
```

La méthode toArray() est utilisable sur n'importe quel Stream y compris sur les Streams primitifs.

Exemple (code Java 8) :

```
   Integer[] integerArray = Stream.of(1, 2, 3, 4, 5).toArray(Integer[]::new);
   int[] class=n>intArray = IntStream.of(1, 2, 3, 4, 5).toArray();
   long[] longArray = LongStream.of(1, 2, 3, 4, 5).toArray();
   double[] doubleArray = DoubleStream.of(1, 2, 3, 4, 5).toArray();
```

22.4.9. La méthode iterator

La méthode iterator() de l'interface BaseStream renvoie un Iterator<T> qui permet de parcourir dans une itération extérieure tous les éléments d'un Stream.

Exemple (code Java 8) :

```
    Stream<String> stream = Stream.of("a", "b", "c");
    Iterator<String> it = stream.iterator();
    while (it.hasNext()) {
      String s = it.next();
      System.out.println(s);
    }
```

Attention, comme un Stream ne stocke pas ses éléments, il n'est possible de parcourir les éléments avec l'Iterator qu'une seule fois.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.Iterator;
import java.util.stream.Stream;

public class TestStreamIterator {

  public static void main(String[] args) {
    Stream<String> stream = Stream.of("a", "b", "c");

    Iterator<String> it = stream.iterator();
    while (it.hasNext()) {
      String s = it.next();
      System.out.println(s);
    }

    it = stream.iterator();
    while (it.hasNext()) {
      String s = it.next();
      System.out.println(s);
    }
  }
}
```

Résultat :

```
a
b
c
Exception in thread "main"
java.lang.IllegalStateException: stream has already been operated upon or closed
      at java.util.stream.AbstractPipeline.spliterator(AbstractPipeline.java:343)
```

```
	at java.util.stream.ReferencePipeline.iterator(ReferencePipeline.java:139)
	at fr.jmdoudoux.dej.streams.TestStreamIterator.main(TestStreamIterator.java:16)
```

La méthode iterator() renvoie un Iterator. Cependant parfois c'est un Iterable qui est requis notamment pour une utilisation dans une instruction for. Il est alors possible de caster en Iterable une référence de méthode sur la méthode iterator() du Stream.

Exemple (code Java 8) :

```
    Stream<String> stream = Stream.of("a", "b", "c");
    for (String s : (Iterable<String>)stream::iterator) {
      System.out.println(s);
    }
```

Comme l'interface Iterable ne possède qu'une seule méthode abstraite Iterator<T> iterator(), c'est une interface fonctionnelle. Elle peut donc être implémenté sous la forme d'une expression Lambda et donc d'une référence de méthode. La signature de la méthode iterator() de l'interface Stream correspond à la signature de la méthode iterator() de l'interface Iterable : il est donc possible d'utiliser une référence de méthode sur la méthode iterator() de l'interface Stream pour implémenter l'interface Iterable. Il est cependant nécessaire d'effectuer un cast pour forcer le target type de la référence de méthode.

22.5. Les Collectors

Une des surcharges de la méthode collect() attend en paramètre un objet de type Collector. Les traitements à appliquer sont alors définis par l'interface Collector.

La classe java.util.stream.Collectors propose un ensemble de fabriques qui renvoient des implémentations de Collector pour des opérations de réduction communes.

22.5.1. L'interface Collector

Un Collector permet de réaliser une opération de réduction qui accumule les éléments d'un Stream dans un conteneur mutable. Il peut éventuellement appliquer une transformation pour permettre de fournir le résultat final dans un type différent de celui du conteneur dans lequel les éléments sont accumulés. Les traitements des opérations de réduction peuvent être exécutés de manière séquentielle ou parallèle.

Les traitements d'un Collector sont définis grâce à 4 fonctions qui sont utilisées pour agréger les éléments du Stream dans un conteneur mutable, avec éventuellement une transformation optionnelle pour produire le résultat final :

- Un supplier : permet de renvoyer une nouvelle instance du conteneur mutable
- Un accumulator : accumule un élément dans le conteneur
- Un combiner : combine une instance du type du conteneur pour en produire un seul
- Un finisher : transforme le conteneur pour renvoyer une instance du type du résultat renvoyé (cette fonction est optionnelle)

L'interface Collector<T,A,R> possède trois types génériques :

- T : le type des éléments utilisé par l'opération de réduction
- A : le type de l'objet mutable utilisé par l'opération de réduction
- R : le type du résultat de l'opération de réduction

Elle définit plusieurs méthodes :

Méthode	Rôle
BiConsumer<A,T> accumulator()	

	Renvoyer la fonction qui traite un élément de type T dans le conteneur de type A
Set<Collector.Characteristics> characteristics()	Renvoyer une collection contenant les caractéristiques du Collector
BinaryOperator<A> combiner()	Renvoyer la Function qui combine deux conteneurs de type A encapsulant chacun un résultat intermédiaire
Function<A,R> finisher()	Renvoyer la Function qui transforme un objet de type A contenant le résultat de l'accumulation des résultats en un objet de type R contenant le résultat final
static <T,A,R> Collector<T,A,R> of(Supplier<A> supplier, BiConsumer<A,T> accumulator, BinaryOperator<A> combiner, Function<A,R> finisher, Collector.Characteristics... characteristics)	Fabrique pour construire un Collector qui va utiliser les fonctions supplier, accumulator, combiner et finisher fournies en paramètre
static <T,R> Collector<T,R,R> of(Supplier<R> supplier, BiConsumer<R,T> accumulator, BinaryOperator<R> combiner, Collector.Characteristics... characteristics)	Fabrique pour construire un Collector qui va utiliser les fonctions supplier, accumulator, combiner et finisher ainsi que les caractéristiques fournies en paramètre
Supplier<A> supplier()	Renvoyer le Supplier qui permet de créer une nouvelle instance du conteneur mutable de type A

L'utilisation d'une implémentation d'un Collector dans des traitements séquentiels réalise basiquement plusieurs traitements :

- Utilise le Supplier pour créer une instance du conteneur
- Invoque la fonction accumulator pour chaque élément
- Invoque éventuellement la fonction finisher si nécessaire à la fin

L'utilisation d'une implémentation d'un Collector dans des traitements parallèles réalise basiquement plusieurs traitements :

- Utilise le Supplier pour créer une instance du conteneur pour chaque lot
- Invoque la fonction accumulator pour chaque élément du lot
- Invoque la fonction combiner pour fusionner les conteneurs de chaque lot
- Invoque éventuellement la fonction finisher si nécessaire à la fin

Un Collector peut avoir un ensemble de caractéristiques qui peuvent être utilisées pour optimiser les traitements et ainsi améliorer les performances. Ces caractéristiques sont stockées dans une collection de type Set immuable de valeurs définies dans l'énumération Collector.Characteristics.

Valeur	Rôle
CONCURRENT	Préciser que les traitements du Collector peuvent être exécutés en concurrence : dans ce cas, les fonctions accumulator des différents threads utilisent la même instance du conteneur
IDENTITY_FINISH	Préciser que la fonction finisher correspond simplement à la fonction identity et que son exécution peut donc être évitée
UNORDERED	Préciser que l'opération de réduction ne préserve pas l'ordre des éléments du Stream

Elles peuvent permettre la mise en oeuvre éventuelle d'optimisations dans le traitement de réduction du Collector.

22.5.2. La classe Collectors

La classe Collectors propose des fabriques pour obtenir des instances de Collector qui réalisent des agrégation communes telles que l'ajout des éléments dans une collection, la concaténation de chaînes de caractères, des réductions, des calculs numériques et statistiques, des groupements, ...

Elle propose des méthodes statiques qui sont des fabriques pour renvoyer des instances de type Collector fournies en standard dans l'API permettant de répondre aux principaux besoins courants :

Méthodes	Rôle
toList()	Renvoyer un Collector qui renvoie les éléments du Stream dans une collection de type List
toSet()	Renvoyer un Collector qui renvoie les éléments du Stream dans une collection de type Set
toCollection(Supplier<C>)	Renvoyer un Collector qui renvoie les éléments du Stream dans une collection dont l'implémentation est fournie par le Supplier
toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper)	Renvoyer un Collector qui renvoie les éléments du Stream dans une collection de type Map dont les clés et les valeurs sont déterminées en invoquant leurs Function respectives
toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction)	Renvoyer un Collector qui renvoie les éléments du Stream dans une collection de type Map en tenant compte des clés dupliquées grâce à la fonction de merge fournie sous la forme d'un BinaryOperator
collectingAndThen(Collector<T,A,R> downstream, Function<R,RR> finisher)	Renvoyer un Collector qui exécute une action supplémentaire sous la forme d'une Function après l'exécution du Collector fourni en paramètre
joining()	Renvoyer un Collector qui concatène les éléments d'un Steam<String>
joining(CharSequence delimiter)	Renvoyer un Collector qui concatène les éléments d'un Steam<String> avec le séparateur fourni en paramètre
joining(CharSequence delimiter, CharSequence prefix, CharSequence suffix)	Renvoyer un Collector qui concatène les éléments d'un Steam<String> avec le séparateur, le préfixe et le suffixe fournis en paramètres
counting()	Renvoyer un Collector qui compte le nombre d'éléments du Stream
<T> Collector<T,?,Integer> summingInt(ToIntFunction<T>) <T> Collector<T,?,Long> summingLong(ToLongFunction<T>) <T> Collector<T,?,Double> summingDouble(ToDoubleFunction<T>)	Renvoyer un Collector qui calcule la somme des valeurs retournées par la Function fournie en paramètre
<T> Collector<T,?,IntSummaryStatistics> summarizingInt(ToIntFunction<T>) <T> Collector<T,?,LongSummaryStatistics> summarizingLong(ToLongFunction<T>) <T> Collector<T,?,DoubleSummaryStatistics> summarizingDouble(ToDoubleFunction<T>)	Renvoyer un Collector qui calcule la somme, le minimum, le maximum, le nombre d'éléments et la moyenne des valeurs retournées par la Function fournie en paramètre

<T> Collector<T,?,Optional<T>> reducing(BinaryOperator<T> op)	Renvoyer un Collector qui applique l'opération de réduction définie par le BinaryOperator fourni en paramètre
<T> Collector<T,?,T> reducing(T identity, BinaryOperator<T> op)	Renvoyer un Collector qui applique l'opération de réduction définie par le BinaryOperator en utilisant la valeur initiale fournies en paramètres
static <T,U> Collector<T,?,U> reducing(U identity, Function<? super T,? extends U> mapper, BinaryOperator<U> op)	Renvoyer un Collector qui applique l'opération de réduction définie par le BinaryOperator sur le résultat de la transformation encapsulée dans la Function en utilisant la valeur initiale fournies en paramètres
static <T> Collector<T,?,Map<Boolean,List<T>>> partitioningBy(Predicate<? super T> predicate)	Renvoyer un Collector qui sépare les éléments de type T en deux groupes selon le résultat du Predicate fourni en paramètre. Le Collector renvoie une Map dont le type de la clé est Boolean et les valeurs sont de type List<T>
static <T,D,A> Collector<T,?,Map<Boolean,D>> partitioningBy(Predicate<? super T> predicate, Collector<? super T,A,D> downstream)	Renvoyer un Collector qui sépare les éléments en deux groupes selon le résultat du Predicate fourni en paramètre puis applique la réduction encapsulée par le downstream sur les éléments du groupe. Le Collector renvoie une Map dont le type de la clé est Boolean
static <T,U,A,R> Collector<T,?,R> mapping(Function<? super T,? extends U> mapper, Collector<? super U,A,R> downstream)	Renvoyer un Collector qui transforme les éléments en appliquant la Function fournie puis applique la réduction encapsulée par le downstream Collector
static <T> Collector<T,?,Optional<T>> maxBy(Comparator<? super T> comparator)	Renvoyer un Collector qui détermine le plus grand élément selon le Comparator fourni en paramètre et le renvoie dans un Optional
static <T> Collector<T,?,Optional<T>> minBy(Comparator<? super T> comparator)	Renvoyer un Collector qui détermine le plus petit élément selon le Comparator fourni en paramètre et le renvoie dans un Optional
static <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(Function<? super T,? extends K> classifier)	Renvoyer un Collector qui effectue une opération de groupement en renvoyant une Map dont la clé est obtenue grâce à la Function fournie et la valeur est une collection de type List des éléments associés à cette clé
static <T,K,A,D> Collector<T,?,Map<K,D>> groupingBy(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)	Renvoyer un Collector qui effectue une opération de groupement en renvoyant une Map dont la clé est obtenue grâce à la Function fournie et la valeur est le résultat de l'exécution de la réduction encapsulée dans le downstream fourni en paramètre sous la forme d'un Collector à tous les éléments associés à la clé
static <T,K,D,A,M extends Map<K,D>> Collector<T,?,M> groupingBy(Function<? super T,? extends K> classifier, Supplier<M> mapFactory, Collector<? super T,A,D> downstream)	Même comportement que la surcharge précédente mais l'instance de la Map renvoyée est obtenue en invoquant le Supplier passé en paramètre

Le plus simple pour les utiliser est de réaliser un import static

Exemple (code Java 8) :

```
import static java.util.stream.Collectors.*;
```

Dans l'exemple ci-dessous, la méthode collect() utilise le Collector renvoyé par la méthode toList() de la classe Collectors pour renvoyer une collection de type List contenant les éléments du Stream.

Exemple (code Java 8) :

```
    List<Employe> employesMasculins = employes
        .stream()
        .filter(e -> e.getGenre() == Genre.HOMME)
        .collect(Collectors.toList());
    System.out.println(employesMasculins);
```

De la même façon, pour obtenir un collection de type Set, il suffit d'utiliser le Collector retourné par la méthode toSet() de la classe Collectors.

Exemple (code Java 8) :

```
    Set<Employe> employesMasculins = employes
      .stream()
      .filter(e -> e.getGenre() == Genre.HOMME)
      .collect(Collectors.toSet());
    System.out.println(employesMasculins);
```

Il est possible d'utiliser la méthode toCollection() pour pouvoir fournir l'instance de la collection à utiliser pour l'agrégation des éléments. Par exemple, pour agréger les éléments dans une collection de type TreeSet sur des éléments qui nécessairement implémentent l'interface Comparable.

Exemple (code Java 8) :

```
    Set<Employe> employesMasculins = employes
        .stream()
        .filter(p -> p.getGenre() == Genre.HOMME)
       .collect(Collectors.toCollection(TreeSet::new));
    System.out.println(employesMasculins);
```

Si les éléments n'implémentent pas Comparable, il est possible d'utiliser une expression lambda pour implémenter le Supplier qui va instancier le TreeSet en passant en paramètre de son constructeur le Comparator à utiliser.

Exemple (code Java 8) :

```
    Set<Employe> employesMasculins = employes
       .stream()
       .filter(p -> p.getGenre() == Genre.HOMME)
       .collect(Collectors.toCollection(() ->
         new TreeSet<Employe>(Comparator.comparing(Employe::getNom))));
    System.out.println(employesMasculins);
```

Les Collector ne se contentent pas de retourner des collections, ils peuvent aussi réaliser des agrégations pour déterminer un résultat.

Dans l'exemple ci-dessous, la méthode averagingDouble() de la classe Collectors renvoie un Collector qui calcule la moyenne des données obtenues en invoquant la ToDoubleFunction() passée en paramètre sur chacun des éléments.

Exemple (code Java 8) :

```
  Double salaireMoyen = employes.stream()
                                .collect(Collectors.averagingDouble(Employe::getSalaire));
  System.out.println(salaireMoyen);
```

Certains Collector peuvent être plus complexes. Par exemple, la méthode groupingBy() de la classe Collectors renvoie un Collector qui va grouper dans une Map les éléments sur la base d'une clé précisée.

Exemple (code Java 8) :

```
    Map<Genre, List<Employe>> employesParGenre =
      employes.stream()
              .collect(Collectors.groupingBy(Employe::getGenre));
    employesParGenre.forEach((genre, listeEmployes) ->
      System.out.format("%s : %s\n", genre, listeEmployes));
```

Résultat :

```
FEMME : [Employe [nom=e3, genre=FEMME, taille=172, salaire=1850.0], Employe [nom=e4, genre=
FEMME, taille=162, salaire=3300.0], Employe [nom=e6, genre=FEMME, taille=168, salaire=2850.0]]
HOMME : [Employe [nom=e1, genre=HOMME, taille=176, salaire=1500.0], Employe [nom=e2, genre=
HOMME, taille=190, salaire=2700.0], Employe [nom=e5, genre=HOMME, taille=176, salaire=1280.0]]
```

22.5.2.1. Les fabriques pour des Collector vers des collections

La méthode toList() permet de renvoyer les éléments du Stream dans une collection de type List.

Remarque : il n'est pas possible avec cette méthode de pouvoir préciser le type de l'implémentation de la collection. Pour cela, il faut utiliser la méthode toCollection().

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import static java.util.stream.Collectors.toList;
import java.util.Arrays;
import java.util.List;

public class TestCollector {

  public static void main(String[] args) {
    List<String> elements = Arrays.asList("elem1", "elem2", "elem3", "elem4");
    List<String> resultats = elements.stream().collect(toList());
    System.out.println(resultats);
  }
}
```

La méthode toSet() permet de renvoyer les éléments du Stream dans une collection de type Set.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import static java.util.stream.Collectors.toSet;
import java.util.Arrays;
import java.util.List;
import java.util.Set;

public class TestCollector {

  public static void main(String[] args) {
    List<String> elements = Arrays.asList("elem1", "elem2", "elem2", "elem3", "elem4");
    Set<String> resultats = elements.stream().collect(toSet());
    System.out.println(resultats);
  }
}
```

Il n'y a aucune garantie sur l'instance de type Set retournée par le Collector de la méthode toSet().

Remarque : il n'est donc pas possible avec cette méthode de pouvoir préciser le type de l'implémentation de la collection. Pour cela, il faut utiliser la méthode toCollection().

La méthode toCollection() permet de renvoyer les éléments du Stream dans une collection dont l'implémentation est fournie par le Supplier.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import static java.util.stream.Collectors.toCollection;
import java.util.Arrays;
import java.util.List;
import java.util.Set;
import java.util.TreeSet;

public class TestCollectors {

  public static void main(String[] args) {
    List<String> elements = Arrays.asList("elem4", "elem2", "elem2", "elem1", "elem3");
    Set<String> resultats = elements.stream().collect(toCollection(TreeSet::new));
    System.out.println(resultats);
  }
}
```

Remarque : l'implémentation de type Collection fournie doit être mutable

L'utilisation permet d'avoir un contrôle précis sur l'instance utilisée. L'exemple ci-dessous est le même que le précédent mais l'instance de type Set utilisée est différente.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import static java.util.stream.Collectors.toCollection;
import java.util.Arrays;
import java.util.List;
import java.util.Set;
import java.util.TreeSet;

public class TestCollectors {

  public static void main(String[] args) {
    List<String> elements = Arrays.asList("elem4", "elem2", "elem2", "elem1", "elem3");
    Set<String> resultats = elements.stream().collect(toCollection(HashSet::new));
    System.out.println(resultats);
  }
}
```

Les surcharges de la méthode toMap() permettent de renvoyer les éléments du Stream dans une collection de type Map.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import static java.util.stream.Collectors.toMap;
import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.function.Function;

public class TestCollectors {

  public static void main(String[] args) {
    List<String> elements = Arrays.asList("elem1", "elem2", "elem3", "elem4");
    Map<String, Integer> resultats = elements.stream().collect(toMap(Function.identity(),
                                                                          String::length));
    System.out.println(resultats);
  }
}
```

Résultat :

```
{elem1=8, elem2=8, elem3=8, elem4=8}
```

Remarque : cette méthode ne permet pas d'avoir un support des clés dupliquées.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.function.Function;
import java.util.stream.Collectors;

public class TestCollectors {

  public static void main(String[] args) {
    List<String> elements = Arrays.asList("elem1", "elem2", "elem3", "elem4");
    Map<Integer, String> resultats =
      elements.stream()
              .collect(Collectors.toMap(String::length, Function.identity()));
    System.out.println(resultats);
  }
}
```

Résultat :

```
Exception in thread "main" java.lang.IllegalStateException: Duplicate key elem1
      at java.util.stream.Collectors.lambda$throwingMerger$114(Collectors.java:133)
      at java.util.HashMap.merge(HashMap.java:1245)
      at java.util.stream.Collectors.lambda$toMap$172(Collectors.java:1320)
      at java.util.stream.ReduceOps$3ReducingSink.accept(ReduceOps.java:169)
      at java.util.Spliterators$ArraySpliterator.forEachRemaining(Spliterators.java:948)
      at java.util.stream.AbstractPipeline.copyInto(AbstractPipeline.java:481)
      at java.util.stream.AbstractPipeline.wrapAndCopyInto(AbstractPipeline.java:471)
      at java.util.stream.ReduceOps$ReduceOp.evaluateSequential(ReduceOps.java:708)
      at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:234)
      at java.util.stream.ReferencePipeline.collect(ReferencePipeline.java:499)
      at fr.jmdoudoux.dej.streams.TestCollectors.main(TestCollectors.java:17)
```

La surcharge toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction) permet de renvoyer les éléments du Stream dans une collection de type Map en tenant compte des clés dupliquées grâce à la fonction de fusion fournie sous la forme d'un BinaryOperator (mergeFunction).

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import static java.util.stream.Collectors.toMap;
import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.function.Function;

public class TestCollectors {

  public static void main(String[] args) {
    List<String> elements = Arrays.asList("elem1", "elem2", "elem2", "elem3", "elem4");
    Map<Integer, String> resultats =
      elements.stream().collect(toMap(String::length, Function.identity(), (s1, s2) -> s1));
    System.out.println(resultats);
  }
}
```

Résultat :

```
{8=elem1}
```

Dans l'exemple ci-dessus, comme les clés sont identiques, la fonction de fusion des doublons renvoie simplement le premier des deux. Evidemment, la fonction peut être adaptée selon les besoins, par exemple pour cumuler les valeurs des différents éléments de chaque clé.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import static java.util.stream.Collectors.toMap;
import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.function.Function;

public class TestCollectors {

  public static void main(String[] args) {
    List<String> elements = Arrays.asList("elem1", "elem2", "elem2", "elem3", "elem4");
    Map<Integer, String> resultats =
      elements.stream()
              .collect(toMap(String::length,
                             Function.identity(),
                             (s1, s2) -> s1 + ";" + s2));
    System.out.println(resultats);
  }
}
```

Résultat :

```
{8=elem1;elem2;elem2;elem3;elem4}
```

22.5.2.2. La fabrique pour des Collector qui exécutent une action complémentaire

La méthode collectingAndThen() permet d'exécuter une action supplémentaire sous la forme d'une fonction après l'exécution du Collector fourni en paramètre.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import static java.util.stream.Collectors.collectingAndThen;
import static java.util.stream.Collectors.toList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class TestCollector {

  public static void main(String[] args) {
    List<String> elements = Arrays.asList("elem1", "elem2", "elem3", "elem4");
    List<String> resultats = elements.stream()
      .collect(collectingAndThen(toList(), Collections::unmodifiableList));
    System.out.println(resultats.getClass().getName());
  }
}
```

22.5.2.3. Les fabriques qui renvoient des Collector pour réaliser une agrégation

Les surcharges de la méthode joining() de la classe Collectors permettent de concaténer les éléments d'un Stream<String>.

La surcharge sans paramètre joining() renvoie un Collector qui permet de concaténer les éléments d'un Steam<String>

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

```

```
import static java.util.stream.Collectors.joining;
import java.util.Arrays;
import java.util.List;

public class TestCollector {

  public static void main(String[] args) {
    List<String> elements = Arrays.asList("elem1", "elem2", "elem3", "elem4");
    String resultat = elements.stream().collect(joining());
    System.out.println(resultat);
  }
}
```

La surcharge joining(CharSequence delimiter) renvoie un Collector qui permet de concaténer les éléments d'un Stream<String> avec le séparateur fourni en paramètre.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import static java.util.stream.Collectors.joining;
import java.util.Arrays;
import java.util.List;

public class TestCollector {

  public static void main(String[] args) {
    List<String> elements = Arrays.asList("elem1", "elem2", "elem3", "elem4");
    String resultat = elements.stream().collect(joining(","));
    System.out.println(resultat);
  }
}
```

La surcharge joining(CharSequence delimiter, CharSequence prefix, CharSequence suffix) renvoie un Collector qui permet de concaténer les éléments d'une Steam<String> avec le séparateur, le préfixe et le suffixe fournis en paramètre.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import static java.util.stream.Collectors.joining;
import java.util.Arrays;
import java.util.List;

public class TestCollector {

  public static void main(String[] args) {
    List<String> elements = Arrays.asList("elem1", "elem2", "elem3", "elem4");
    String resultat = elements.stream().collect(joining(", ", "[", "]"));
    System.out.println(resultat);
  }
}
```

Le préfixe et le suffixe peuvent être n'importe quelle chaîne de caractères.

Exemple (code Java 8) :

```
    String noms = employes
        .stream()
        .map(Employe::getNom)
        .collect(Collectors.joining(", ", "Les personnes ", " sont des employes."));
    System.out.println(noms);
```

Résultat :

```
Les personnes e1, e2, e3, e4, e5, e6 sont des employes.
```

22.5.2.4. Les fabriques qui renvoient des Collectors pour effectuer des opérations numériques

La méthode counting() renvoie un Collector qui permet de compter le nombre d'éléments du Stream.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import static java.util.stream.Collectors.counting;
import java.util.Arrays;
import java.util.List;

public class TestCollector {

  public static void main(String[] args) {
    List<String> elements = Arrays.asList("elem1", "elem2", "elem3", "elem4");
    Long resultat = elements.stream().collect(counting());
    System.out.println(resultat);
  }
}
```

Les méthodes summarizingInt(), summarizingLong() et summarizingDouble() de la classe Collectors renvoient des Collector qui calculent des informations statistiques basiques sur les données numériques extraites des éléments du Stream : le nombre d'éléments, les valeurs min et max, la moyenne et la somme.

Ces données sont respectivement encapsulées dans des objets de type IntSummaryStatistics, LongSummaryStatistics, et DoubleSummaryStatistics.

Exemple (code Java 8) :

```
    DoubleSummaryStatistics salaireSummary = employes
      .stream()
      .collect(Collectors.summarizingDouble(Employe::getSalaire));
    System.out.println(salaireSummary);
    System.out.println(salaireSummary.getCount());
    System.out.println(salaireSummary.getMin());
    System.out.println(salaireSummary.getMax());
    System.out.println(salaireSummary.getAverage());
    System.out.println(salaireSummary.getSum());
```

Résultat :

```
DoubleSummaryStatistics{count=6,sum=13480,000000, min=1280,000000, average=2246,666667,
 max=3300,000000}
6
1280.0
3300.0
2246.6666666666665
13480.0
```

Les méthodes averagingInt(), averagingLong() et averagingDouble() de la classe Collectors renvoient un Collector qui calcule la moyenne des données numériques extraites des éléments du Stream.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.stream.Collectors;
import java.util.stream.Stream;

public class TestCollectors {

  public static void main(String[] args) {
    Stream<String> chaines = Stream.of("aaa", "bb", "ccccc");
    Double moyenne = chaines.collect(Collectors.averagingDouble(String::length));
    System.out.println(moyenne);
  }
}
```

Résultat :

```
3.3333333333333335
```

Les méthodes summingInt(), summingLong() et summingDouble() de la classe Collectors renvoient un Collector qui calcule la somme des données numériques extraites des éléments du Stream.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.stream.Collectors;
import java.util.stream.Stream;

public class TestCollectors {

  public static void main(String[] args) {
    Stream<String> chaines = Stream.of("aaa", "bb", "ccccc");
    long somme = chaines.collect(Collectors.summingLong(String::length));
    System.out.println(somme);
  }
}
```

Résultat :

```
10
```

Les méthodes minBy() et maxBy() de la classe Collectors renvoient un Collector qui détermine respectivement le plus petit et le plus grand élément du Stream. Elles attendent en paramètre un Comparator qui sera utilisé pour déterminer l'élément concerné. Elles renvoient un Optional qui encapsule éventuellement cet élément s'il existe.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.Comparator;
import java.util.Optional;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class TestCollectors {

  public static void main(String[] args) {
    Stream<String> chaines = Stream.of("aaa", "bb", "ccccc");
    Optional<String> lePlusGrand = chaines.collect(
      Collectors.maxBy(Comparator.naturalOrder()));
    System.out.println(lePlusGrand.get());
  }
}
```

Résultat :

```
ccccc
```

Le Comparator utilisé peut par exemple être plus générique en utilisant la méthode static comparing() de l'interface Comparator.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.Comparator;
import java.util.Optional;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class TestCollectors {

  public static void main(String[] args) {
    Stream<String> chaines = Stream.of("aaa", "bb", "ccccc");
    Optional<String> lePlusLong = chaines.collect(
      Collectors.maxBy(Comparator.comparingInt(String::length)));
    System.out.println(lePlusLong.get());
  }
}
```

22.5.2.5. Les fabriques qui renvoient des Collectors pour effectuer des groupements

La méthode groupingBy() renvoie un Collector qui va regrouper les éléments du Stream dans une Map. Elle possède plusieurs surcharges :

Méthode	Rôle
static <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(Function<? super T,? extends K> classifier)	Renvoyer un Collector qui regroupe les éléments de type T selon la Function fournie dans une collection de type Map
static <T,K,A,D> Collector<T,?,Map<K,D>> groupingBy(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)	Renvoyer un Collector qui regroupe les éléments de type T selon la Function fournie dans une collection de type Map puis exécute le downstream Collector sur les valeurs de chaque clé
static <T,K,D,A,M extends Map<K,D>> Collector<T,?,M> groupingBy(Function<? super T,? extends K> classifier, Supplier<M> mapFactory, Collector<? super T,A,D> downstream)	Renvoyer un Collector qui regroupe les éléments de type T selon la Function fournie dans une collection de type Map fournie par le supplier puis exécute le downstream Collector sur les valeurs de chaque clé

La surcharge groupingBy(Function<? super T,? extends K> classifier) renvoie un Collector qui permet de grouper les éléments selon la clé obtenue par la fonction de classification fournie. Elle utilise implicitement le downstream Collector toList().

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class TestCollectors {

  public static void main(String[] args) {
    Stream<String> chaines = Stream.of("aaa", "bb", "ccccc");
    Map<Integer, List<String>> map = chaines.collect(
      Collectors.groupingBy(String::length, Collectors.toList()));
    for (Map.Entry entry : map.entrySet()) {
      System.out.println(entry.getKey() + ", " + entry.getValue());
    }
  }
}
```

Résultat :

```
2, [bb]
3, [aaa]
5, [ccccc]
```

La méthode partitioningBy() est une spécialisation de la méthode groupingBy(). Elle attend en paramètre un Predicate pour grouper les éléments selon la valeur booléenne retournée par le Predicate. La collection de type Map retournée possède donc forcément un booléen comme clé.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class TestCollectors {

  public static void main(String[] args) {
    Stream<String> chaines = Stream.of("aaa", "bb", "ccccc");
    Map<Boolean, List<String>> map = chaines.collect(
      Collectors.partitioningBy(s -> s.length() >= 3));
    for (Map.Entry entry : map.entrySet()) {
      System.out.println(entry.getKey() + ", " + entry.getValue());
    }
  }
}
```

Résultat :

```
false, [bb]
true, [aaa, ccccc]
```

22.5.2.6. Les fabriques qui renvoient des Collectors pour effectuer des transformations

La méthode mapping() renvoie un Collector qui va transformer les objets de type T en type U grâce à la Function avant d'appliquer le downstream Collector.

public static <T,U,A,R> Collector<T,?,R> mapping(Function<? super T,? extends U> mapper, Collector<? super U,A,R> downstream)

Exemple (code Java 8) :

```
   Map<String, Set<String>> nomsGroupesParVille
       = personnes.stream()
                  .collect(groupingBy(Personne::getVille,
                           mapping(Personne::getNom, toSet())));
```

La méthode reducing() renvoie un Collector qui permet de réaliser une opération de réduction appliquée de manière répétée sur tous les éléments du Stream pour produire un résultat.

Elle possède plusieurs surcharges :

Méthode	Rôle
static <T> Collector<T,?,Optional<T>> reducing(BinaryOperator<T> op)	Renvoyer un Collector qui effectue la réduction selon le BinaryOperator fourni

static <T> Collector<T,?,T> reducing(T identity, BinaryOperator<T> op)	Renvoyer un Collector qui effectue la réduction selon le BinaryOperator et la valeur initiale fournis
static <T,U> Collector<T,?,U> reducing(U identity, Function<? super T,? extends U> mapper, BinaryOperator<U> op)	Renvoyer un Collector qui effectue la réduction selon le BinaryOperator et la valeur initiale fournis en ayant appliqué au préalable la Function sur chaque élément

Ces surcharges attendent jusqu'à trois paramètres :

- Identity : la valeur initiale qui sera retournée si le Stream est vide
- Mapper : une fonction à appliquer sur chacun des éléments pour extraire la valeur à utiliser
- Op : une opération qui combine les valeurs

Exemple (code Java 8) :

```
       int tailleTotale = personnes.stream()
           .collect(reducing( 0, Personne::getTaille, Integer::sum));
```

On peut obtenir le même comportement pour certains de ces Collectors en utilisant des opérations de l'interface Stream notamment min(), max() ou reduce() qu'il est préférable d'utiliser. Cependant l'utilisation de ces Collectors est parfois requise notamment pour combiner des Collectors afin de réaliser des opérations plus complexes essentiellement en tant que downstream Collector.

22.5.3. La composition de Collectors

Il est possible de combiner des Collectors pour réaliser des réductions plus complexes : par exemple faire des groupements à plusieurs niveaux.

Ces combinaisons sont similaires à celles utilisables en SQL : il est possible de combiner un GROUP BY avec des opérations telles que COUNT. Il est donc possible d'utiliser un autre Collector pour déterminer la valeur comme par exemple pour compter le nombre d'éléments de chaque groupe.

Une surcharge de la méthode groupBy() qui attend en second paramètre un objet de type Collector permet d'appliquer le Collector pour réduire les éléments du groupe et ainsi obtenir la valeur associée à la clé.

Exemple (code Java 8) :

```
    Stream<String> mots = Stream.of("aa", "bb", "aa", "bb","cc", "bb");
    Map<String, Long> nbMots = mots.collect(
      Collectors.groupingBy(s -> s.toUpperCase(), Collectors.counting()));
    for (Map.Entry entry : nbMots.entrySet()) {
      System.out.println(entry.getKey() + ", " + entry.getValue());
    }
```

Résultat :

```
CC, 1
BB, 3
AA, 2
```

Comme la méthode groupingBy() renvoie un Collector, il est possible d'utiliser un groupingBy() comme downstream Collector et ainsi réaliser un groupement à deux niveaux. Dans l'exemple ci-dessous, on effectue un groupement par âge et par salaire des employés.

Exemple (code Java 8) :

```
import static java.util.stream.Collectors.groupingBy;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;

public class TestStream {

  public static void main(String[] args) {
    employes.add(new Employe("n1", 33, 29000));
    employes.add(new Employe("n2", 41, 41000));
    employes.add(new Employe("n3", 33, 29000));
    employes.add(new Employe("n4", 41, 37000));
    employes.add(new Employe("n5", 33, 33000));
    employes.add(new Employe("n6", 54, 54000));

    Map<Integer, Map<Double, List<Employe>>> employesParAgeEtParSalaire = employes
        .stream()
        .collect(groupingBy(Employe::getAge,
                 groupingBy(Employe::getSalaire)));
    System.out.println("resultat = "+ employesParAgeEtParSalaire);
  }
}
```

Résultat :

```
resultat = {33={33000.0=[Employe [nom=n5]],
29000.0=[Employe [nom=n1], Employe [nom=n3]]}, 54={54000.0=[Employe [nom=n6]]},
41={37000.0=[Employe [nom=n4]], 41000.0=[Employe [nom=n2]]}}
```

La combinaison de Collectors au travers des downstream Collectors permet d'exprimer des traitements complexes. Dans l'exemple ci-dessous, ils sont utilisés pour déterminer le salaire le plus élevé par tranches d'âges d'une collection d'employés.

Exemple (code Java 8) :

```
import static java.util.stream.Collectors.collectingAndThen;

import static java.util.stream.Collectors.groupingBy;
import static java.util.stream.Collectors.mapping;
import static java.util.stream.Collectors.maxBy;
import java.util.ArrayList;
import java.util.List;
importjava.util.Map;

public class TestStream {

  public static void main(String[] args) {
    List<Employe> employes = new ArrayList<>(6);
    employes.add(new Employe("n1", 33, 29000));
    employes.add(new Employe("n2", 41, 41000));
    employes.add(new Employe("n3", 33, 29000));
    employes.add(new Employe("n4", 41, 37000));
    employes.add(new Employe("n5", 33, 33000));
    employes.add(new Employe("n6", 54, 54000));

    Map<Integer, Double> salaireMaxParAges = employes.stream()
        .collect(groupingBy(Employe::getAge,
            mapping(Employe::getSalaire,
               collectingAndThen(maxBy(Double::compare),
                    d -> d.get()
                        .doubleValue()))));
    System.out.println(salaireMaxParAges);
  }
}
```

Résultat :

```
{33=33000.0, 54=54000.0, 41=41000.0}
```

22.5.4. L'implémentation d'un Collector

Le JDK propose en standard un ensemble d'implémentations de Collector pour des besoins courants. Il est possible de développer sa propre implémentation de l'interface Collector pour définir des opérations de réductions personnalisées si celles-ci ne sont pas fournies par le JDK.

L'interface Collector implémentée doit être typée avec 3 génériques :

public interface Collector<T, A, R> {...}

Les trois types génériques correspondent aux types utilisés par le Collector :

- T : le type des objets qui seront traités
- A : le type de l'objet utilisé comme accumulator
- R : le type de l'objet qui sera retourné comme résultat

Il est courant que les types A et R soient identiques : c'est par exemple le cas pour un type de l'API Collection. Mais ils peuvent être différents par exemple un accumulator de type StringBuilder et un résultat de type String.

L'implémentation d'un Collector peut se faire de deux manières :

- Utiliser une des surcharges de la fabrique of() de l'interface Collector : elle permet de créer des Collector simples
- Définir une classe qui implémente l'interface Collector : pour des cas plus complexes ou des besoins particuliers

22.5.5. Les fabriques of() pour créer des instances de Collector

Il est possible de créer une instance de type Collector en utilisant sa fabrique of() qui possède deux surcharges.

L'exemple ci-dessous créé une instance de type Collector pour concaténer les prénoms d'un Stream de d'objet de type Personne mis en majuscules. Le séparateur utilisé est une virgule.

Exemple (code Java 8) :

```
Collector<Personne, StringJoiner, String> prenomPersonneCollector =
    Collector.of(
     () -> new StringJoiner(","),                       // supplier
     (sj, p) -> sj.add(p.getPrenom().toUpperCase()),  // accumulator
     (sj1, sj2) -> sj1.merge(sj2),                    // combiner
     StringJoiner::toString);                        // finisher
String prenoms = personnes
 .stream()
 .collect(prenomPersonneCollector);
System.out.println(prenomss);
```

Le Collector utilise la classe StringJoiner de Java 8 pour concaténer les prénoms des personnes dans ses traitements :

- Le supplier fournit une instance de StringJoiner avec le délimiteur à utiliser
- L'accumulator utilise la méthode add() du StringJoiner pour concaténer chaque prénom mis en majuscules en les séparant par une virgule
- Le combiner utilise la méthode merge() pour fusionner les deux StringJoiner fournis en paramètre
- Le finisher utilise la méthode toString() du StringJoiner pour fournir le résultat final sous la forme d'une chaîne de caractères. Son utilisation est nécessaire car le type du Supplier et le type renvoyé du Collector sont différents

22.6. Les Streams pour des données primitives

L'interface Stream<T> utilise un type generic T et s'utilise donc avec des objets de type T. Il existe aussi plusieurs interfaces dédiées à la manipulation de types primitifs int, long et double respectivement IntStream, LongStream et DoubleStream.

Lorsqu'un Stream est utilisé sur des données primitives cela peut engendrer de nombreuses opérations de boxing/unboxing pour encapsuler une valeur primitive dans un objet de type wrapper et vice versa. Ces opérations peuvent être coûteuses lorsque le nombre d'éléments à traiter dans le Stream est important.

L'exemple ci-dessus est purement pédagogique pour réaliser des traitements sur un Stream qui effectue un ensemble d'opération sur des valeurs entières.

Exemple (code Java 8) :

```
public Long testStreamLong() {
  Long somme = Stream
      .iterate(0L, i -> i + 1L)
      .limit(20_000_000)
      .filter(i -> (i % 2) == 0)
      .map(i -> i + 1)
      .sorted()
      .reduce(0L, Long::sum);
  return somme;
}
```

Un benchmark est réalisé sur cette méthode avec JMH :

Résultat :

```
Benchmark                        Mode  Cnt   Score    Error     Units
StreamBenchmark.testStreamLong  avgt  10    4322,430 ± 258,794 ms/op
```

La méthode ci-dessous produit le même résultat mais elle utilise un Stream primitif

Exemple (code Java 8) :

```
public long testLongStream() {
  long somme = LongStream
      .iterate(0, i -> i + 1)
      .limit(20_000_000)
      .filter(i -> (i % 2) == 0)
      .map(i -> i + 1)
      .sorted()
      .sum();
  return somme;
}
```

Un benchmark est réalisé sur cette méthode pour permettre de comparer les performances des deux versions :

Résultat :

```
Benchmark                        Mode Cnt    Score   Error   Units
StreamBenchmark.testLongStream  avgt 10   233,061 ± 10,962  ms/op
StreamBenchmark.testStreamLong  avgt 10  4322,430 ± 258,794 ms/op
```

La différence de temps de traitement est sans appel en faveur de l'utilisation du Stream primitif. Lorsque les données à traiter par le Stream sont des données primitives de type int, long ou double, il est donc préférable d'utiliser les interfaces IntStream, LongStream et DoubleStream.

C'est par exemple un IntStream qui est retourné lorsque l'on utilise une opération de type map T vers int. Dans ce cas, ce sont des interfaces fonctionnelles renvoyant un type primitif qui sont utilisées comme IntSupplier, IntConsumer, ToIntFunction.

Ces interfaces fonctionnent comme l'interface Stream avec quelques différences :

- Les interfaces fonctionnelles utilisées sont celles spécifiques aux types primitifs : par exemple IntFunction à la place de Function, IntPredicate à la place de Predicate, ...
- Elles proposent quelques opérations terminales spécifiques aux types primitifs comme sum(), average(), ...

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.stream.IntStream;

public class TestStream {

  public static void main(String[] args) {
    int valeur = IntStream.of(1, 2, 3, 4, 5).map(n -> 2 * n).sum();
    System.out.println(valeur);
  }
}
```

Résultat :

```
30
```

Il est parfois utile voire même nécessaire de transformer un Stream primitif en Stream d'objets et vice versa.

Les méthodes mapToInt(), mapToLong() et mapToDouble() de l'interface Stream fonctionne comme la méthode map() sauf qu'au lieu de renvoyer un Stream<T>, elles renvoient respectivement un IntStream, un LongStream et un DoubleStream.

Elles permettent donc de transformer un Stream d'objets en Stream de primitifs de leur type respectif. Elles attendent respectivement en paramètre une interface fonctionnelle de type ToIntFunction, ToLongFunction et ToDoubleFunction.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.stream.Stream;

public class TestStream {

  public static void main(String[] args) {
    Stream.of("1", "2", "3")
          .mapToInt(Integer::parseInt)
          .max().ifPresent(System.out::println);
  }
}
```

Résultat :

```
3
```

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.stream.Stream;

public class TestStream {

  public static void main(String[] args) {
    int valeur = Stream.of(1.0, 2.0, 3.0).mapToInt(Double::intValue).sum();
    System.out.println(valeur);
  }
}
```

Résultat :

```
6
```

Les Streams primitifs peuvent être transformés en Stream d'objets en utilisant la méthode mapToObj(). Selon le type primitif, elle attend respectivement en paramètre une interface fonctionnelle de type IntFunction, LongFunction et DoubleFunction.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.stream.IntStream;

public class TestStream {

  public static void main(String[] args) {
    IntStream.range(1, 4).mapToObj(i -> "" + i).forEach(System.out::println);
  }
}
```

Résultat :

```
1
2
3
```

La méthode boxed() permet de transformer un Stream primitif en Stream<T> ou T est soit un Integer, un Long ou un Double.

22.6.1. Les interfaces IntStream, LongStream et DoubleStream

L'interface IntStream propose de nombreuses méthodes :

Méthode	Rôle
boolean allMatch(IntPredicate predicate)	Renvoyer un booléen qui précise si tous les éléments respectent le Predicate
boolean anyMatch(IntPredicate predicate)	Renvoyer un booléen qui précise si au moins un élément du Stream respecte le Predicate
DoubleStream asDoubleStream()	Renvoyer un DoubleStream contenant tous les éléments du Stream convertis en double
LongStream asLongStream()	Renvoyer un LongStream contenant tous les éléments du Stream convertis en long
OptionalDouble average()	Renvoyer un OptionalDouble qui encapsule la moyenne des éléments du Stream ou empty si le Stream est vide
Stream<Integer> boxed()	Renvoyer un Stream qui contient tous les éléments du Stream convertis en Integer
static IntStream.Builder builder()	Renvoyer un builder pour un IntStream
<R> R collect(Supplier<R> supplier, ObjIntConsumer<R> accumulator, BiConsumer<R,R>	Appliquer une opération de réduction sur chacun des éléments du Stream

combiner)	
static IntStream concat(IntStream a, IntStream b)	Renvoyer un Stream qui contient la concaténation des éléments du premier avec ceux du second Stream fournis en paramètre
long count()	Renvoyer le nombre d'éléments du Stream
IntStream distinct()	Renvoyer un Stream ne contenant que les éléments distincts
static IntStream empty()	Renvoyer un IntStream vide
IntStream filter(IntPredicate predicate)	Renvoyer un Stream qui contient les éléments qui respectent le Predicate
OptionalInt findAny()	Renvoyer un OptionalInt vide si le Stream ne contient aucun élément sinon un OptionalInt qui encapsule un des éléments du Stream
OptionalInt findFirst()	Renvoyer un OptionalInt qui encapsule le premier élément du Stream s'il contient au moins un élément sinon un OptionalInt vide
IntStream flatMap(IntFunction<? extends IntStream> mapper)	Renvoyer un Stream qui est le résultat de l'agrégation des IntStream produits par l'exécution de la fonction sur chacun des éléments
void forEach(IntConsumer action)	Exécuter l'action fournie en paramètre sur chacun des éléments
void forEachOrdered(IntConsumer action)	Exécuter l'action fournie en paramètre sur chacun des éléments en conservant leur ordre. Les traitements ne sont réalisés que dans un seul thread.
static IntStream generate(IntSupplier s)	Renvoyer un Steam infini dont les valeurs sont générées par l'IntSupplier fourni en paramètre
static IntStream iterate(int seed, IntUnaryOperator f)	Renvoyer un IntStream infini ordonné dont les valeurs sont générées en invoquant de manière itérative la fonction pour renvoyer seed, f(seed), f(f(seed)), ...
PrimitiveIterator.OfInt iterator()	Renvoyer un Iterator permettant le parcours des éléments du Stream
IntStream limit(long maxSize)	Renvoyer un Stream qui contient les maxSize premiers éléments du Stream
IntStream map(IntUnaryOperator mapper)	Renvoyer un IntStream contenant le résultat de l'application de la fonction sur chacun des éléments du Stream
DoubleStream mapToDouble(IntToDoubleFunction mapper)	Renvoyer un DoubleStream contenant le résultat de l'application de la fonction sur chacun des éléments du Stream
LongStream mapToLong(IntToLongFunction mapper)	Renvoyer un LongStream contenant le résultat de l'application de la fonction sur chacun des éléments du Stream
<U> Stream<U> mapToObj(IntFunction<? extends U> mapper)	Renvoyer un Stream d'éléments de type U contenant le résultat de l'application de la fonction sur chacun des éléments du Stream
OptionalInt max()	Renvoyer un OptionalInt qui contient la valeur maximale du Stream s'il possède au moins un élément sinon un OptionalInt vide si le Stream ne contient aucun élément
OptionalInt min()	Renvoyer un OptionalInt qui contient la valeur minimale du Stream s'il possède au moins un élément sinon un OptionalInt vide si le Stream ne contient aucun élément
boolean noneMatch(IntPredicate predicate)	Renvoyer un booléen qui indique si aucun élément ne respecte le Predicate fourni en paramètre

static IntStream of(int... values)	Renvoyer un IntStream qui contient les éléments fournis en paramètre
static IntStream of(int t)	Renvoyer un IntStream qui contient uniquement l'élément fourni en paramètre
IntStream parallel()	Renvoyer un Stream équivalent dont les traitements seront exécutés en parallèle
IntStream peek(IntConsumer action)	Renvoyer un Stream qui contient tous les éléments du Stream en ayant appliqué l'IntConsumer sur chacun d'eux
static IntStream range(int startInclusive, int endExclusive)	Renvoyer un IntStream ordonné qui contient les valeurs comprises entre le premier paramètre inclus et le second paramètre exclus avec une incrémentation de 1
static IntStream rangeClosed(int startInclusive, int endInclusive)	Renvoyer un IntStream ordonné qui contient les valeurs comprises entre le premier paramètre inclus et le second paramètre inclus avec une incrémentation de 1
OptionalInt reduce(IntBinaryOperator op)	Effectuer une opération de réduction sur chacun des éléments du Stream, en utilisant la fonction d'accumulation associative fournie en paramètre
int reduce(int identity, IntBinaryOperator op)	Effectuer une opération de réduction sur chacun des éléments du Stream, en utilisant la valeur initiale et la fonction d'accumulation associative fournie en paramètre
IntStream sequential()	Renvoyer un Stream équivalent dont les traitements seront exécutés en séquentiel
IntStream skip(long n)	Renvoyer un Stream qui contient les éléments du Stream en ayant ignoré les n premiers
IntStream sorted()	Renvoyer un Stream dont les éléments sont triés dans l'ordre naturel
Spliterator.OfInt spliterator()	Renvoyer un Spliterator pour les éléments du Stream
int sum()	Renvoyer la somme de tous les éléments
IntSummaryStatistics summaryStatistics()	Renvoyer un IntSummaryStatistics qui encapsule différentes valeurs statistiques calculées sur les valeurs du Stream
int[] toArray()	Renvoyer un tableau des éléments du Stream

Les méthodes range() et rangeClosed() permettent de créer un Stream dont les éléments seront les valeurs comprises entre les bornes fournies en paramètres :

- Les deux méthodes attendent en premier paramètre la valeur initiale de la plage de valeur
- Le second paramètre indique la valeur finale, exclue pour la méthode range() et incluse pour la méthode rangeClosed()

Exemple (code Java 8) :

```java
package fr.jmdoudoux.dej.streams;

import java.util.stream.IntStream;

public class TestStream {

  public static void main(String[] args) {
    IntStream.rangeClosed(1, 5)
             .forEach(n -> System.out.print(n));
    System.out.println();
    IntStream.range(1, 5)
             .forEach(n -> System.out.print(n));
  }
}
```

Résultat :

```
12345
1234
```

L'exemple ci-dessous détermine les nombres impairs compris entre 0 et 10 inclus.

Exemple (code Java 8) :

```
    IntStream nombresImpairs = IntStream.rangeClosed(0, 10)
                                    .filter(nombre -> (nombre % 2) == 1);
    nombresImpairs.forEach(System.out::println);
```

Résultat :

```
1
3
5
7
9
```

Il est possible de créer une collection d'objets en utilisant la méthode range() pour définir chaque élément.

Exemple (code Java 8) :

```
    List<Departement> departements = new ArrayList<>();
    IntStream.range(1, 3).forEach(i -> {
      Departement departement = new Departement("Departement" + i);
      departements.add(departement);
      IntStream.range(1, 4).forEach(
          j -> departement.getEtudiants().add(new Etudiant("nom" + j + "_"
            + departement.getNom(), 20 + j)));
    });
```

Remarque : l'utilisation de Streams dont les traitements modifient l'état d'objets externes n'est cependant pas recommandé.

La méthode sum() est une opération de réduction qui calcule la somme des éléments du Stream.

Exemple (code Java 8) :

```
   int tailleTotale = personnes.stream()
                               .mapToInt(Personne::getTaille)
                               .sum();
    System.out.println(tailleTotale);
```

La méthode average() est une opération de réduction qui calcule la moyenne des éléments du Stream. Elle renvoie un objet de type OptionalDouble. Elle renvoie un objet de type xxxOptional ou xxx est le type primitif du Stream.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.stream.IntStream;

public class TestStream {

  public static void main(String[] args) {
    IntStream.of(1, 2, 3, 4, 5)
             .map(n -> 2 * n)
             .average()
             .ifPresent(System.out::println);
  }
}
```

Résultat :

```
6.0
```

22.7. L'utilisation des Streams avec les opérations I/O

Certaines méthodes des classes I/O renvoient un Stream notamment à partir de la lecture d'un fichier texte ou le contenu des éléments d'un répertoire.

22.7.0.1. La création d'un Stream à partir d'un fichier texte

L'API NIO 2 de Java 7, propose une méthode pour lire l'intégralité des lignes d'un fichier texte.

List<String> lines = Files.readAllLines(somePath, someCharset);

L'utilisation d'un Stream permet de ne pas avoir besoin de stocker l'intégralité du contenu du fichier en mémoire : la lecture dans le fichier se fait au fur et à mesure de la consommation par le Stream. Ceci peut être intéressant notamment pour le traitement de gros fichiers.

La méthode lines() de la classe Files attend en paramètre le fichier sous la forme d'un objet de type Path. Elle permet de renvoyer un Stream<String> qui va lire de manière lazy les lignes d'un fichier pour alimenter le Stream au fur et à mesure de la consommation des lignes.

Contrairement à la méthode readAllLines() qui lit l'intégralité du fichier, la méthode lines() peut se contenter de ne lire que les lignes que lors de l'invocation de la méthode terminale et la lecture peut être interrompue par l'exécution d'une méthode de type short-circuiting.

Les caractères sont décodés en utilisant le ChartSet UTF-8 par défaut. La méthode lines() possède une seconde surcharge qui attend en paramètre deux objets de type Path et CharSet.

Lorsque la source de données du Stream effectue des opérations de type I/O, il est nécessaire de libérer les ressources allouées pour lire les données. L'interface BaseStream définit la méthode close() et la classe Stream implémente l'interface AutoCloseable.

La plupart des Streams n'ont pas besoin d'avoir leur méthode close() invoquée sauf dans certains cas notamment lorsque la source réalise des opérations de type I/O. La méthode close() doit être explicitement invoquée si nécessaire pour libérer les ressources ouvertes par la source.

Pour éviter de laisser le système libérer les ressources au bout d'un certain timeout, il faut invoquer la méthode close() de l'objet responsable de la lecture des données. Ceci peut éviter des fuites de ressources. Il est donc important d'invoquer la méthode close() du Stream pour qu'elle invoque la méthode close() de la source utilisée pour lire les lignes du fichier.

La méthode onClose() de l'interface BaseStream est une opération intermédiaire qui permet d'exécuter un traitement lorsque la méthode close() du Stream est invoquée. Il est possible d'invoquer plusieurs fois cette méthode : les traitements seront alors exécutés dans leur ordre d'ajout.

Exemple (code Java 8) :

```
   try {
     Stream<String> lignes = Files.lines(Paths.get("fichier.txt"), Charset.defaultCharset());
     long nbLignes = lignes.count();
     lignes.close();
     System.out.println("Nb lignes = " + nbLignes);
   } catch (IOException e) {
      e.printStackTrace();
   }
```

Il est aussi possible d'utiliser un try with resources sur un Stream qui va invoquer sa méthode close(). Cette dernière va se charger de libérer les éventuelles ressources ouvertes par la source de données.

Exemple (code Java 8) :

```
    try (Stream<String> lignes = Files.lines(Paths.get("fichier.txt"))) {
      long nbLignes = lignes.count();
      System.out.println("Nb lignes = " + nbLignes);
    } catch (IOException e) {
      e.printStackTrace();
    }
```

La méthode lines() de la classe BufferedReader renvoie un Stream dont les éléments sont les lignes du fichier lu.

Il est important que seul le Stream utilise le BufferedReader pour lire les données.

Si une exception de type IOException survient durant l'utilisation du BufferedReader, celle-ci est encapsulée dans une UncheckedIOException.

Exemple (code Java 8) :

```
    try {
      BufferedReader br = Files.newBufferedReader(Paths.get("fichier.txt"));
      Stream<String> lignes = br.lines();
      lignes.forEach(System.out::println);
      lignes.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
```

Il est aussi plus simple d'utiliser le Stream comme ressource d'un try with resources.

Exemple (code Java 8) :

```
    try (Stream<String> lignes = Files.newBufferedReader(Paths.get("fichier.txt"))
         .lines()
         .onClose(() -> System.out.println("Fermeture du fichier"))) {
      lignes.forEach(System.out::println);
    } catch (Exception e) {
      e.printStackTrace();
    }
```

22.7.0.2. La création d'un Stream à partir du contenu d'un répertoire

Plusieurs méthodes ont été ajoutées à la classe Files dans la version 8 de Java.

Méthode	Rôle
public static Stream<Path> list(Path dir)	Renvoyer un Stream dont la source est la liste des éléments contenus dans le répertoire fourni en paramètre sous la forme d'un objet de type Path. La liste ne contient que les éléments du répertoire et n'est donc pas construite de manière récursive. Cette liste ne contient pas non plus les liens vers le répertoire courant et le répertoire parent si le système de fichiers les gère. Le Stream est lazy : les éléments sont obtenus au fur et à mesure du parcours ce qui implique que des modifications dans le répertoire peuvent être ou non répercutées dans les éléments obtenus.

	Le Stream retourné est de type DirectoryStream.
public static Stream<Path> walk(Path start, int maxDepth, FileVisitOption... options)	Renvoyer un Stream dont la source est la liste des éléments obtenus lors du parcours de l'arborescence à partir de la racine précisée par le paramètre start. La liste contient des éléments de type Path correspond au chemin relatif à partir de la racine des éléments obtenus pendant le parcours. Le Stream est lazy : les éléments sont obtenus au fur et à mesure du parcours ce qui implique que des modifications dans le répertoire en cours de parcours peuvent être ou non répercutées dans les éléments obtenus. Par défaut, les liens symboliques ne sont pas suivis. Pour qu'ils soient suivis, il faut utiliser la valeur FOLLOW_LINKS dans le paramètre options. Les visites cycliques lors du parcours sont détectées et lève une exception de type FileSystemLoopException. Le paramètre maxDepth permet de préciser le niveau de répertoire maximum à visiter. La valeur 0 indique qu'il faut parcourir uniquement le répertoire de départ. La valeur MAX_VALUE indique qu'il n'y a pas de limite La paramètre start indique le répertoire de départ. Le Stream retourné est de type DirectoryStream.
public static Stream<Path> walk(Path start, FileVisitOption... options)	Cette méthode est équivalente à l'invocation de sa surcharge avec les options : walk(start, Integer.MAX_VALUE, options)
public static Stream<Path> find(Path start, int maxDepth, BiPredicate<Path,BasicFileAttributes> matcher, FileVisitOption... options)	Cette méthode fonctionne comme l'autre surcharge de la méthode find mais celle-ci permet de filter les éléments à inclure. Le paramètre matcher est un BiPredicate qui permet de filter les éléments qui sont inclus dans le Stream.

Il est important d'invoquer la méthode close() du Stream à la fin de son utilisation. Le plus simple est d'utiliser une instruction try with resources.

Exemple (code Java 8) :

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.stream.Stream;

public class TestFilesStream {

  public static void main(String[] args) {
    String folder = "c:/temp";
    try (Stream<Path> paths = Files.list(Paths.get(folder))) {
      paths.filter(p -> p.toString()
           .endsWith(".dat"))
           .forEach(System.out::println);
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }
  }
}
```

22.8. Le traitement des opérations en parallèle

Une des fonctionnalités les plus mises en avant concernant les Streams est la facilité déconcertante pour exécuter les traitements des opérations en parallèle.

L'avènement des processeurs multi-cour, omniprésents dans les appareils informatique (ordinateurs, tablettes, téléphones mobiles, ...) oblige à mettre en oeuvre des traitements en parallèle pour exploiter leur puissance.

Java 7 a introduit le framework Fork/Join pour faciliter la parallélisation de tâches. Ce framework est utilisé par l'implémentation de l'API Stream pour permettre l'exécution de ses traitements en parallèle.

Globalement aussi, le volume des données à traiter augmente. Généralement ces données sont stockées dans une collection. La conception de l'API Collection étant assez ancienne (Java 1.2), il n'est pas facile de proposer l'intégration de fonctionnalités en parallèle dans ces classes.

L'exécution de traitements sur des données d'une collection requiert de réaliser une itération sur chacun des éléments. Le traitement d'éléments dans une itération est par définition intrinsèquement séquentiel.

Exemple (code Java 5.0) :

```
List<String> elements = Arrays.asList("elem1", " elem2", " elem3","elem4", "elem5");
for (String element : elements)
   System.out.println(element);
```

La mise en oeuvre de ces traitements en parallèle est très compliquée même avec l'API Fork/Join. Pourtant elle peut être nécessaire si le volume des données est important. Pour ne pas réécrire intégralement l'API Collection et ainsi maintenir la rétrocompatibilité, Java SE 8 propose dans l'API Streams le traitement en parallèle de données.

Très facilement, l'API Stream permet de réaliser ses traitements en séquentiel ou en parallèle. Seules des méthodes par défaut pour l'obtention d'un Stream à partir d'une Collection ont été ajoutées à cette API.

Exemple (code Java 8) :

```
List<String> elements = Arrays.asList("elem1", " elem2", " elem3", "elem4", "elem5");
elements.stream()
        .forEach(System.out::println);
elements.parallelStream()
        .forEach(System.out::println);
```

22.8.1. La mise en oeuvre des Streams parallèles

La plupart des Streams créés dans l'API du JDK ont leurs opérations qui par défaut seront exécutées de manière séquentielle dans le thread courant. Il est alors nécessaire de préciser explicitement que l'on souhaite que les opérations du Stream soient exécutées de manière parallèle.

Puisque c'est l'API qui se charge d'itérer sur les différents éléments pour exécuter le pipeline d'opérations, il est facilement possible de demander l'exécution de ces traitements en parallèle. Le fonctionnement interne de l'API masque alors toute la complexité de l'exécution en parallèle des traitements.

Pour le développeur, cela consiste simplement :

- Soit à demander la création d'un Stream parallèle en utilisant la méthode parallelStream() des collections
- Soit à convertir un Stream séquentiel en Stream parallèle invoquant la méthode parallel() de l'interface BaseStream qui est une opération intermédiaire

Remarque : il est aussi possible de transformer un Stream parallèle en Stream séquentiel en invoquant la méthode sequential().

22.8.2. Le fonctionnement interne d'une Stream

Le découpage en sous-lots est réalisé grâce à un objet de type Spliterator. La suite d'opérations du pipeline pour chacun de ces sous-lots sera alors exécuté par un thread libre du pool du framework Fork/Join.

22.8.2.1. L'interface Spliterator pour l'obtention des données par la source

La liaison entre un Stream et sa source de données se fait avec un objet de type Spliterator : il permet l'obtention des données de la source par le Stream.

Un Spliterator est un objet pour traverser et partitionner les éléments d'une source. Il offre donc deux fonctionnalités :

- Un itérateur qui permet de parcourir séquentiellement les éléments d'une source et de les fournir au Stream
- La décomposition de l'ensemble des éléments en deux sous-ensembles pour leur traitement en parallèle par le Stream, idéalement contenant chacun une moitié, dans le Spliterator courant et dans un autre Spliterator. Ces deux Spliterator peuvent être eux-mêmes ensuite décomposés

L'interface Spliterator définit les méthodes pour un objet qui permet de parcourir et partitionner les éléments d'une source de données :

Méthode	Rôle
int characteristics()	Renvoyer les caractéristiques du Spliterator et de ses éléments
long estimateSize()	Renvoyer une estimation du nombre d'éléments qui pourrait être traité par l'invocation de la méthode forEachRemaining() ou renvoyer Long.MAX_VALUE si le nombre est infini, inconnu ou trop coûteux à traiter
default void forEachRemaining(Consumer<? super T> action)	Appliquer le Consumer sur les éléments restants de manière séquentielle dans le thread courant jusqu'à ce qu'il n'y ait plus d'élément ou que le Consumer lève une exception
default Comparator<? super T> getComparator()	Si la source du Spliterator est triée avec un Comparator, alors renvoie ce Comparator
default long getExactSizeIfKnown()	Si le Spliterator a la caractéristique SIZED, alors renvoie la valeur de la méthode estimateSize() sinon renvoie -1
default boolean hasCharacteristics(int characteristics)	Renvoyer un booléen qui indique si le Spliterator possède toutes les caractéristiques passées en paramètre
boolean tryAdvance(Consumer<? super T> action)	S'il reste un élément à traiter dans la source, alors applique le Consumer sur cet élément et renvoie true sinon renvoie false
Spliterator<T> trySplit()	Si le Spliterator peut être partitionné, renvoie un nouveau Spliterator qui prend en charge une partie des éléments (idéalement la moitié)

Le parcours des éléments peut se faire avec deux méthodes :

- La méthode tryAdvance() permet d'appliquer le Consumer aux éléments un par un (de manière similaire à l'invocation de la méthode next() de l'interface Iterator)
- La méthode forEachRemaining() permet d'appliquer le Consumer à tous les éléments restants

L'interface Spliterator propose le support de la création de lots d'éléments traités en parallèle dans plusieurs threads par l'API Stream. Un Spliterator permet de découper un ensemble plus petit des éléments d'entrée dans un nouveau Spliterator (idéalement, la moitié), et laisser le reste des éléments dans le Spliterator original. Les Spliterator peuvent ensuite être de nouveau décomposés au besoin. La méthode trySplit() renvoie une nouvelle instance de type Spliterator dont la responsabilité est de gérer un sous-ensemble des éléments du Spliterator. La méthode trySplit() peut renvoyer null si l'implémentation du Spliterator ne supporte pas la création de lots : attention dans ce cas, le Stream utilisant un tel

Spliterator ne pourra pas être exécuté en parallèle. L'implémentation du Spliterator utilisée peut affecter les performances d'un Stream exécuté en parallèle.

Un Spliterator contient des métadonnées comme par exemple le nombre d'éléments à traiter (s'il est connu) et un ensemble de quelques caractéristiques sous la forme d'un entier. Ces informations peuvent être utilisées pour optimiser les traitements réalisés par le Spliterator.

L'ensemble de caractéristiques (distinct, ordered, sorted, sized, nonnull, immutable, concurrent, subsized) dépend du type de source de données et est utilisé pour réaliser ses traitements de manière plus ou moins optimisée.

Caractéristique	Rôle
CONCURRENT	Les éléments de la source peuvent être modifiés de manière concurrente sans avoir à utiliser un mécanisme supplémentaire de synchronisation
DISTINCT	Aucun élément n'est égal à un autre de la source : tous les éléments sont distincts
IMMUTABLE	La source est immuable : il n'est pas possible d'ajouter, de remplacer ou de supprimer un élément
NONNULL	La source ne contient aucun élément qui soit null
ORDERED	Les éléments sont ordonnés
SIZED	La méthode estimateSize() renvoie une valeur fiable qui donne la taille des éléments à traiter sous réserve que la source ne soit pas modifiée
SORTED	Les éléments sont triés dans un certain ordre
SUBSIZED	L'invocation de la méthode trySplit() renvoie un Spliterator qui possède les caractéristiques SIZED et SUBSIZED

Pour des usages standards, l'implémentation de Spliterator utilisée est celle fournie par la méthode qui permet d'obtenir un Stream. Pour des besoins particuliers, il peut être nécessaire d'implémenter son propre Spliterator. Il suffit de définir une classe qui implémente l'interface Spliterator.

Plusieurs caractéristiques doivent être prise en compte lors du développement d'un Spliterator :

- Le Spliterator connait le nombre d'éléments de la source de données
- Le Spliterator est capable de partitionner les éléments de la source de données
- Le Spliterator peut partitionner les éléments en portion de taille similaire (caractéristique SUBSIZED)

La manière la plus simple d'obtenir un Spliterator mais avec des performances moyennes voire mauvaises est d'utiliser la méthode spliteratorUnknownSize() de la classe Spliterators en lui passant en paramètre un Iterator.

Pour obtenir de meilleure performance, il est possible d'utiliser la méthode spliterator() de la classe Spliterators en lui passant en paramètre un Iterator et le nombre d'éléments.

Il est enfin possible de définir son propre Spliterator en implémentant l'interface Spliterator.

Les classes de l'API Collection possèdent des implémentations de Spliterator : il est possible de s'en inspirer pour développer ses propres implémentations. Les interfaces Iterable et Collection proposent une implémentation basique et peu optimisée du Spliterator retournée par la méthode spliterator(). Les interfaces filles et leurs implémentations proposent des implémentations beaucoup plus performantes de leurs Spliterators notamment grâce à une exploitation de leurs métadonnées. Par exemple, l'implémentation proposée par ArrayList utilise le nombre d'éléments contenus dans la collection pour déterminer de manière efficace le nombre d'éléments de chaque lot retourné par la méthode split().

Les implémentations de Spliterator doivent faire un compromis entre simplicité d'implémentation et performance d'exécution.

Une implémentation basique mais peu performante est retournée par la méthode spliteratorUnkownSize(Iterator, int) de la classe Spliterators : celle-ci n'utilise pas le nombre d'éléments puisqu'il n'est pas connu par un Iterator. L'implémentation est donc obligée de parcourir les éléments de l'Iterator et utilise un algorithme de découpage rudimentaire.

Une implémentation plus performante pourra par exemple s'appuyer sur le nombre d'éléments de la source de données si celui-ci est connu. L'implémentation pourra alors utiliser cette information pour définir des lots de données équilibrés et de manière efficace surtout si par exemple, les données sont stockées dans un tableau. L'implémentation d'un Spliterator peut aussi utiliser d'autres caractéristiques pour optimiser ses traitements.

Plusieurs classes du JDK ont aussi été modifiées pour servir de source de données pour un Stream. Il est aussi possible d'adapter une classe qui encapsule des données pour qu'elle puisse être utilisée comme source pour un Stream. Pour créer un Stream à partir d'une classe qui sera sa source de données, il est nécessaire d'utiliser un Spliterator.

La classe StreamSupport est un utilitaire qui facilite la création de Streams. Elle propose plusieurs méthodes statiques pour créer un Stream qui attendent en paramètre un Spliterator et un booléen qui précise si le Stream est séquentiel (false) ou parallèle (true).

22.8.2.2. L'utilisation du framework Fork/Join

Le code utilisé pour paralléliser un traitement et le code de ce traitement exécuté de manière séquentielle est très différent : le premier nécessite de nombreuses lignes de code supplémentaires.

Le framework Fork/Join propose une API pour faciliter l'implémentation de traitements en parallèle sur un ensemble de données.

Le mode de fonctionnement de ce framework est composé de plusieurs étapes :

- Découper l'ensemble des données en sous-ensemble
- Traiter chaque sous-ensemble dans un des threads du pool pour produire un résultat partiel
- Agréger les résultats partiels pour produire un résultat final

L'API Stream propose d'utiliser le même code pour définir les traitements. Ceux-ci pourront très facilement être exécutés de manière séquentielle ou parallèle. Ceci rend très facile d'utiliser l'exécution des traitements d'un Stream en parallèle de manière fiable puisque c'est l'implémentation de l'API Stream qui se charge de tout en utilisant le framework Fork/Join.

Le framework Fork/Join utilise un pool de threads de type ForkJoinPool. A partir de Java 8, Fork/Join utilise un pool de threads commun. Il est possible d'obtenir l'instance de ce pool en invoquant la méthode statique commonPool() de la classe ForkFoinPool.

La méthode getParallelism() permet d'obtenir le nombre de threads souhaité dans le pool et la méthode getCommonParallelism() permet d'obtenir le nombre de threads souhaité dans le pool commun.

Par défaut, le nombre de threads souhaité dans le pool pour l'exécution des tâches par le framework Fork/Join est égale au nombre de coeurs moins un. Par exemple, sur une machine équipée d'un processeur Intel Core i7-5500U, la valeur de l'attribut parallelism est 3.

Exemple :

```
package fr.jmdoudoux.dej.streams;

import java.util.concurrent.ForkJoinPool;

public class TestParellelStream {

  public static void main(String[] args) {
    ForkJoinPool commonPool = ForkJoinPool.commonPool();
    System.out.println(commonPool.getParallelism());
  }
}
```

Résultat :

```
3
```

Il est possible de modifier cette valeur en utilisant la variable d'environnement de la JVM java.util.concurrent.ForkJoinPool.common.parallelism

Résultat :

```
-Djava.util.concurrent.ForkJoinPool.common.parallelism=6
```

Pour comprendre le fonctionnement de l'exécution d'un Stream parallèle, il est possible d'afficher des informations lors de l'exécution de chaque opération : la méthode invoquée, l'élément traité et le nom du thread courant.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.io.PrintStream;
import java.util.stream.IntStream;

public class TestParellelStream {

  public static void main(String[] args) {
    IntStream
        .rangeClosed(1, 5)
        .parallel()
        .mapToObj(v -> {
          afficher("mapToObj", "e" + v);
          return "e" + v;
        })
        .filter(s -> {
          afficher("filter ", s);
          return true;
        })
        .map(s -> {
          afficher("map ", s);
          return s.toUpperCase();
        })
        .forEach(s -> afficher("forEach ", s));
  }

  private static PrintStream afficher(String methode, String valeur) {
    return System.out.format("%s: %s [%s]\n", methode, valeur,
      Thread.currentThread().getName());
  }
}
```

Résultat :

```
mapToObj: e3 [main]
mapToObj: e1 [ForkJoinPool.commonPool-worker-3]
mapToObj: e5 [ForkJoinPool.commonPool-worker-2]
mapToObj: e2 [ForkJoinPool.commonPool-worker-1]
filter: e5 [ForkJoinPool.commonPool-worker-2]
filter: e1 [ForkJoinPool.commonPool-worker-3]
filter: e3 [main]
map: e3 [main]
map: e1 [ForkJoinPool.commonPool-worker-3]
forEach: E1 [ForkJoinPool.commonPool-worker-3]
map: e5 [ForkJoinPool.commonPool-worker-2]
filter: e2 [ForkJoinPool.commonPool-worker-1]
map: e2 [ForkJoinPool.commonPool-worker-1]
forEach: E5 [ForkJoinPool.commonPool-worker-2]
mapToObj:e4 [ForkJoinPool.commonPool-worker-3]
forEach: E3 [main]
filter: e4 [ForkJoinPool.commonPool-worker-3]
map: e4 [ForkJoinPool.commonPool-worker-3]
forEach: E2 [ForkJoinPool.commonPool-worker-1]
forEach: E4 [ForkJoinPool.commonPool-worker-3]
```

L'exécution des traitements en parallèle n'est pas prédictive. Plusieurs exécutions de ce code donneront un ordre de traces différent.

L'API Stream utilise le thread courant et les threads du pool commun de Fork/Join. Chaque thread exécute le pipeline d'opérations pour un sous-ensemble de données, restreint dans l'exemple ci-dessus car le nombre d'éléments est trop faible.

22.8.3. Le mode d'exécution des Streams

Lorsque l'opération terminale du Stream est invoquée, les opérations sont exécutées en séquentiel ou en parallèle selon la configuration du Stream.

Les opérations parallel() et sequential() de l'interface BaseStream sont des opérations intermédiaires : elles sont donc lazy.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.io.PrintStream;
import java.util.stream.IntStream;

public class TestParellelStream {

  public static void main(String[] args) {
    IntStream
        .rangeClosed(1, 5)
        .mapToObj(v -> {
          afficher("mapToObj", "e" + v);
          return "e" + v;
        })
        .filter(s -> {
          afficher("filter ", s);
          return true;
        })
        .map(s -> {
          afficher("map ", s);
          return s.toUpperCase();
        })
        .parallel()
        .forEach(s -> afficher("forEach ", s));
  }

  private static PrintStream afficher(String methode, String valeur) {
    return System.out.format("%s: %s [%s]\n", methode, valeur,
      Thread.currentThread().getName());
  }
}
```

Résultat :

```
mapToObj:e2 [ForkJoinPool.commonPool-worker-1]
mapToObj:e3 [main]
mapToObj:e1 [ForkJoinPool.commonPool-worker-3]
mapToObj:e5 [ForkJoinPool.commonPool-worker-2]
filter: e1 [ForkJoinPool.commonPool-worker-3]
filter: e3 [main]
map: e3 [main]
filter: e2 [ForkJoinPool.commonPool-worker-1]
map: e2 [ForkJoinPool.commonPool-worker-1]
forEach: E3 [main]
map: e1 [ForkJoinPool.commonPool-worker-3]
filter: e5 [ForkJoinPool.commonPool-worker-2]
forEach: E1 [ForkJoinPool.commonPool-worker-3]
mapToObj: e4 [main]
forEach: E2 [ForkJoinPool.commonPool-worker-1]
filter: e4 [main]
map: e4 [main]
map: e5 [ForkJoinPool.commonPool-worker-2]
```

```
forEach: E4 [main]
forEach: E5 [ForkJoinPool.commonPool-worker-2]
```

C'est la dernière de ces opérations qui sera utilisée pour déterminer le mode d'exécution du Stream.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.io.PrintStream;
import java.util.stream.IntStream;

public class TestParellelStream {

  public static void main(String[] args) {
    IntStream
        .rangeClosed(1, 5)
        .mapToObj(v -> {
          afficher("mapToObj", "e" + v);
          return "e" + v;
        })
        .parallel()
        .filter(s -> {
          afficher("filter ", s);
          return true;
        })
        .map(s -> {
          afficher("map ", s);
          return s.toUpperCase();
        })
        .sequential()
        .forEach(s -> afficher("forEach", s));
  }

  private static PrintStream afficher(String methode, String valeur) {
    return System.out.format("%s: %s [%s]\n", methode, valeur,
      Thread.currentThread().getName());
  }
}
```

Résultat :

```
mapToObj: e1 [main]
filter: e1 [main]
map: e1 [main]
forEach: E1 [main]
mapToObj:e2 [main]
filter: e2 [main]
map: e2 [main]
forEach: E2 [main]
mapToObj:e3 [main]
filter: e3 [main]
map: e3 [main]
forEach: E3 [main]
mapToObj: e4 [main]
filter: e4 [main]
map: e4 [main]
forEach: E4 [main]
mapToObj: e5 [main]
filter: e5 [main]
map: e5 [main]
forEach: E5 [main]
```

De la même manière, bien que l'opération parallel() ait été utilisée puis la méthode sequential(), le Stream est exécuté de manière séquentielle dans son intégralité. Aucune opération n'est exécutée en parallèle.

22.8.4. Le comportement de certaines opérations en parallèle

L'exécution en séquentiel ou en parallèle d'un Stream permet généralement d'obtenir le même résultat sauf si une opération non déterministe est utilisée comme findAny(), findFirst(), ... Dans ce cas, le résultat n'est pas forcement le même sauf si le Stream ne contient plus qu'un seul élément lors de l'exécution de l'opération terminale.

Le comportement de l'opération sorted() est particulier lors d'une exécution en parallèle.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.io.PrintStream;
import java.util.stream.IntStream;

public class TestParellelStream {

  public static void main(String[] args) {
    IntStream
        .rangeClosed(1, 5)
        .parallel()
        .mapToObj(v -> {
          afficher("mapToObj", "e" + v);
          return "e" + v;
        })
        .sorted((s1, s2) -> {
          afficher("sorted ", s1 + "/" + s2);
          return s1.compareTo(s2);
        })
        .forEach(s -> afficher("forEach ", s));
  }

  private static PrintStream afficher(String methode, String valeur) {
    return System.out.format("%s: %s [%s]\n", methode, valeur,
      Thread.currentThread().getName());
  }
}
```

Résultat :

```
mapToObj: e2 [ForkJoinPool.commonPool-worker-1]
mapToObj: e5 [main]
mapToObj: e1 [ForkJoinPool.commonPool-worker-3]
mapToObj: e3 [ForkJoinPool.commonPool-worker-2]
mapToObj: e4 [ForkJoinPool.commonPool-worker-1]
sorted: e2/e1 [main]
sorted: e3/e2 [main]
sorted: e4/e3 [main]
sorted: e5/e4 [main]
forEach: e3 [main]
forEach: e5 [ForkJoinPool.commonPool-worker-2]
forEach: e2 [ForkJoinPool.commonPool-worker-1]
forEach: e1 [ForkJoinPool.commonPool-worker-3]
forEach: e4 [main]
```

L'exécution de la méthode sorted() pour chaque élément se fait toujours dans le même thread ce qui n'est pas forcement le comportement attendu vu que le Stream est parallèle.

En fait, l'implémentation de la méthode sorted() utilise la méthode parallelSort() de la classe Arrays. Les traitements de cette méthode sont exécutés de manière séquentielle jusqu'à un certain nombre d'éléments et une fois ce nombre atteint les traitements sont effectués en parallèle.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.io.PrintStream;
import java.util.stream.IntStream;
```

```
public class TestParellelStream {
  public static void main(String[] args) {
    // ForkJoinPool commonPool = ForkJoinPool.commonPool();
    // System.out.println(commonPool.getParallelism());
    IntStream
        .rangeClosed(1, 10000)
        .parallel()
        .mapToObj(v -> {
          // afficher("mapToObj", "e" + v);
          return "e" + v;
        })
        .sorted((s1, s2) -> {
          afficher("sorted ", s1 + "/" + s2);
          return s1.compareTo(s2);
        })
        .forEach(s -> afficher("forEach ", s));
  }

  private static PrintStream afficher(String methode, String valeur) {
    return System.out.format("%s: %s [%s]\n", methode, valeur,
      Thread.currentThread().getName());
  }
}
```

Résultat :

```
...
sorted: e9906/e9905 [ForkJoinPool.commonPool-worker-2]
sorted: e9907/e9906 [ForkJoinPool.commonPool-worker-2]
sorted: e6206/e6205 [ForkJoinPool.commonPool-worker-1]
sorted: e6207/e6206 [ForkJoinPool.commonPool-worker-1]
sorted: e6208/e6207 [ForkJoinPool.commonPool-worker-1]
sorted: e9908/e9907 [ForkJoinPool.commonPool-worker-2]
sorted: e9909/e9908 [ForkJoinPool.commonPool-worker-2]
sorted: e9910/e9909 [ForkJoinPool.commonPool-worker-2]
sorted: e9911/e9910 [ForkJoinPool.commonPool-worker-2]
sorted: e9912/e9911 [ForkJoinPool.commonPool-worker-2]
...
```

Si le nombre d'éléments à traiter est important, alors l'exécution de la méthode sorted() ce fait bien en parallèle.

Certaines opérations, comme reduce() ou collect() ont besoin de traitements supplémentaires lors de l'exécution du Stream en parallèle par rapport à leur exécution en séquentiel notamment pour agréger les résultats issus de chacun des threads.

22.8.5. L'impact de l'état ordonné ou non des éléments

Certains Stream peuvent avoir un ordre précis dans les éléments qu'il doit traiter. Dans ce cas, le Stream possède un attribut Ordered dont l'origine peut provenir soit :

- De la source de données qui peut déjà contenir les éléments dans un certain ordre (par exemple les tableaux, les collections de type List, ...) ou pas (par exemple les collections de type HashSet)
- Des opérations intermédiaires peuvent ordonner les éléments du Stream (par exemple l'opération sorted()) ou au contraire retirer le caractère ordonné des éléments (par exemple l'opération unordered())

Certaines opérations intermédiaires peuvent imposer un ordre sur les éléments d'un Stream qui n'est pas ordonné, alors que d'autres opérations intermédiaires peuvent retourner un Stream non ordonné pour un Stream ordonné.

Certaines opérations terminales peuvent ne pas tenir compte de l'ordre des éléments ordonnés d'un Stream (par exemple l'opération forEach() dans le cas d'un traitement en parallèle).

Pour un Stream ordonné, la plupart des opérations sont contraints de respecter l'ordre des éléments.

Dans un Stream séquentiel, si les éléments d'un Stream sont ordonnés, généralement la plupart des opérations intermédiaires traitent les éléments dans leur ordre. Pour un Stream séquentiel, le fait que les éléments soient ordonnés où non n'affecte pas les performances. Seul le déterminisme du résultat est impacté : si les éléments sont ordonnés, plusieurs exécutions donneront toujours le même résultat. Si les éléments ne sont pas ordonnés alors les résultats peuvent être aléatoires.

Pour un Stream parallèle, le fait que les éléments ne soient pas ordonnés peut améliorer les performances. Certaines opérations exécutées en parallèle sont plus efficaces si les éléments ne sont pas ordonnés : c'est par exemple le cas de l'opération distinct().

Il est possible d'utiliser l'opération unordered() pour retirer la caractéristique ordonnée des éléments d'un Stream. Dans certain cas, cela peut améliorer les performances de traitement de certaines opérations en parallèle sous réserve que l'ordre des éléments ne soit pas nécessaire.

Certaines opérations qui sont intrinsèquement liées à l'ordre, telles que limit(), peuvent nécessiter une mise en mémoire tampon pour respecter l'ordre, compromettant potentiellement l'intérêt de l'exécution des traitements en parallèle.

22.8.6. Les performances des traitements en parallèle

Il est très fréquent que l'utilisation d'un Stream parallèle soit décevante voire ne permette pas d'obtenir les performances attendues. Ceci est due à plusieurs raisons :

- L'amélioration des performances en parallélisant les traitements dépend du type de traitements exécutés et de la stratégie de parallélisation. La meilleure stratégie dépend du type de traitements
- La vitesse d'exécution est dépendante de l'environnement. Certains facteurs sont à prendre en compte : nombre de coeurs disponibles sur la machine, type de processeur, exécution dans un serveur d'applications, ...
- Pour obtenir une amélioration des performances, il est nécessaire d'avoir une quantité significative de données à traiter. La parallélisation des traitements nécessite un surcoût d'opérations (découpage en lots (fork) et d'agrégation des résultats (join)). Ce surcoût ne sera compensé que par le volume des données à traiter en parallèle
- Certains traitements peuvent voir leur performance s'effondrer en s'exécutant en parallèle. C'est notamment le cas, si ceux-ci induisent une forte contention en les parallélisant

Quel que soit le type de traitements à effectuer par le Stream en parallèle, la stratégie utilisée reposant sur le framework Fork/Join est toujours la même. Cette stratégie requiert :

- Un pool de threads, partagé par tout le code qui exécute des tâches avec Fork/Join
- Découper les données à traiter en sous-lots, éventuellement de manière récursive
- Exécuter les traitements sur un sous-lot de données par un des threads libres du pool
- Agréger les résultats intermédiaires de chaque sous-lots pour créer le résultat final

Evidemment, hormis l'exécution des traitements, toutes ces étapes ajoutent un surcoût au temps de traitement par rapport à une exécution séquentielle.

Ce mode de fonctionnement peut aussi induire des effets de bord liés au framework Fork/Join. Toutes les exécutions de code utilisant le framework Fork/Join se font par les threads du pool. Le nombre de ces threads est défini et donc limité.

Si une ou plusieurs tâches en cours d'exécution par le framework Fork/Join sont très longues, elles bloquent autant de threads du pool et ainsi peuvent induire une dégradation des performances liées à l'attente par certaines tâches de leur exécution par un thread libre. Cela concerne des traitements du Stream parallèle, d'autres Streams parallèles ou d'autres fonctionnalités du JDK ou personnelles qui utilisent aussi le framework Fork/Join (Parallel Arrays par exemple)

Par défaut, la taille du pool est égale au nombre de coeurs disponibles sur la machine moins un avec la valeur un comme minimum.

Si les traitements effectués par le Stream requièrent beaucoup de CPU, le gain de performance est contraint entre autres par le nombre de cours et par l'activité des autres threads sur la machine.

Si les traitements ont beaucoup de temps d'attente, le gain de performance peut aussi être plus perceptible.

L'utilisation d'un Stream en parallèle dans un serveur ou conteneur Java EE n'est généralement pas plus performant qu'un traitement en séquentiel. Pour permettre une meilleure montée en charge, les serveurs d'applications exécutent les traitements dans différents threads, aussi bien dans le conteneur web que dans le conteneur d'EJB. Rajouter une couche de parallélisation dans des traitements déjà en parallèle n'apporte aucune amélioration et engendre généralement une baisse des performances.

Pour s'assurer du gain de performance lors de l'utilisation d'un Stream en parallèle, il est fortement recommandé de faire des mesures sous la forme de benchmarks.

En cas de mesures des performances, il est nécessaire de les faire dans des conditions les plus poches possibles de la cible d'exploitation sur différents critères :

- Caractéristiques hardwares : type et nombre de CPU, type de disques durs, type et quantité de mémoire, ...
- Charge de traitements sur l'application mais aussi sur la machine elle-même le plus proche possible

Sans prendre ceci en compte, il est tout à fait probable que les performances soient bonnes en environnement de test ou de développement et mauvaises voire particulièrement mauvaises en production.

22.8.7. Des recommandations pour les Streams parallèles

Tous les traitements en parallèles exécutés par des Streams utilisent par défaut le pool de threads communs du framework Fork/Join. Il est donc nécessaire de ne pas exécuter en parallèle de longs traitements bloquants.

Lors de l'exécution des traitements d'un Stream en parallèle, l'ordre de traitement des éléments n'est pas garanti.

Il n'est pas recommandé d'utiliser du code qui produise des effets de bord dans les traitements d'un Stream. Ce type de traitement devrait généralement être fait différemment ou exécuté dans une boucle.

22.8.7.1. L'utilisation de traitements stateless

En programmation fonctionnelle, il est fortement recommandé de travailler de manière stateless : il ne faut pas modifier l'état des éléments en cours de traitement. Même si rien ne l'empêche, il ne faut pas modifier les éléments en cours de traitement par un Stream. C'est encore plus important dans le cas de l'exécution en parallèle des traitements sous peine d'avoir des comportements imprédictibles et des résultats erronés.

Le modèle de programmation des Streams est intrinsèquement stateless. L'utilisation d'expressions Lambda stateful dans des Streams exécutés en mode séquentiel ne pose généralement pas de soucis. Par contre, cette utilisation dans un Stream exécuté en mode parallèle est une très mauvaise idée qui conduit généralement à des problèmes pouvant engendrer des résultats aléatoires.

Tenter de modifier l'état mutable d'un objet partager dans les traitements d'un Stream implique des problématiques de concurrence d'accès. Les accès concurrents doivent être gérés manuellement pour éviter des problématiques de race conditions. La meilleure approche est de conserver tous les traitements du Stream stateless autant que possible. Cela implique généralement de revoir l'organisation des traitements.

Il n'est pas possible de compter sur l'API ou sur le compilateur pour s'assurer que les traitements soient stateless ou que si les traitements sont stateful alors les accès concurrents sont correctement gérés. Cette gestion implique obligatoirement une dégradation, plus ou moins importante, des performances liées généralement à l'ajout de contention.

Les Streams permettent l'exécution de leur pipeline d'opérations en parallèle à partir de diverses sources données qui ne sont pas forcement thread-safe comme par exemple une collection de type ArrayList. Pour que cela fonctionne correctement, il est impératif qu'il n'y ait pas de modifications des éléments de la source de données.

D'une manière générale, il est important de s'assurer que la source de données ne soit pas modifiée, de quelque manière que ce soit durant l'exécution du pipeline d'opérations d'un Stream. Cela est cependant possible, mais pas recommandé, si la source de données prend en charge la gestion des accès concurrents (le Spliterator associé au Stream doit avoir la caractéristique CONCURRENT).

22.8.7.2. Les effets de bord

Il est préférable de ne pas produire d'effets de bord dans les traitements fournis aux opérations d'un Stream afin de maintenir les traitements sans état et surtout d'éviter d'avoir des comportements inattendus lors de l'exécution des traitements en parallèle.

L'utilisation de traitements qui induisent des effets de bord dans des traitements en parallèle d'un Stream peuvent engendrer différentes difficultés, notamment :

- Il n'y a de garantie que les effets de bord soient vus par les autres threads
- L'utilisation d'objets mutables qui ne prennent pas en charge la gestion de la concurrence peut avoir des résultats non désirés
- L'utilisation d'objets synchronized induit de la contention qui peut engendrer des problèmes de performances

Lors de l'utilisation de Stream, il est tentant d'utiliser des opérations comme peek() ou forEach() pour manipuler un ou plusieurs objets mutables. La tentation est grande car ce type de traitement est utilisé dans les itérations externes réalisées avant Java 8. Bien sûre si ces opérations sont utilisées pour afficher des données, l'impact des effets de bords induits est négligeable. Par contre, si ces traitements modifient un objet mutable, il y a de forts risques d'avoir des résultats indésirés ou ayant des performances dégradées.

Exemple (code Java 8) :

```
    ArrayList<Integer> resultats = new ArrayList<>();
    Stream.of(1, 2, 3, 4)
          .filter(i -> i % 2 == 0)
          .forEach(i -> resultats.add(i));
    System.out.println(resultats);
```

L'exemple ci-dessus produit un effet de bord en modifiant une collection de type List. Cette portion de code fonctionne correctement dans des traitements en séquentiel du Stream. Par contre, si les traitements sont exécutés en parallèle, et c'est très facile de modifier le code pour le faire, les résultats ne seront probablement pas ceux attendus car la classe ArrayList n'est pas thread-safe. Il est aussi possible dans ce cas d'utiliser une version synchronized de la collection mais cela va ajouter de la contention et donc dégrader les performances.

Il est préférable de ne pas utiliser de traitements qui effectuent des effets de bords. Dans l'exemple ci-dessus, c'est simple car il suffit d'utiliser un Collector pour aggéger les valeurs dans une collection.

Exemple (code Java 8) :

```
    List<Integer> resultats = Stream.of(1, 2, 3, 4)
        .filter(i -> i % 2 == 0)
        .collect(Collectors.toList());
    System.out.println(resultats);
```

L'avantage de ce code est qu'il s'exécute correctement en séquentiel et surtout en parallèle aussi.

22.9. Les optimisations réalisées par l'API Stream

La manière de traiter les éléments par un Stream est particulière. On pourrait imaginer que les traitements sont exécutés de manière horizontale : tous les éléments sont traités dans leur intégralité par chaque opération les uns après les autres. Mais en fait, les traitements d'un Stream sont verticaux : chaque élément est traité successivement par chaque opération.

Ce comportement à l'exécution des traitements du Stream permet d'optimiser le nombre d'opérations à réaliser.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;
```

```
import java.util.stream.Stream;

public class TestOperationsIntermediaires {

  public static void main(String[] args) {
    Stream.of("b2", "a1", "b1", "a2", "a3")
          .peek(e -> System.out.println("filter : " + e))
          .filter(e -> e.startsWith("a"))
          .peek(e -> System.out.println("anyMatch : " + e))
          .anyMatch(e -> e.contains("1"));
  }
}
```

Résultat :

```
filter: b2
filter: a1
anyMatch : a1
```

Dans l'exemple ci-dessus, l'opération filter n'est invoquée que sur deux éléments. Dès que le Predicat fourni à la méthode anyMatch() renvoie true pour un élément du Stream, les traitements de l'exécution du Stream sont interrompus. Cela permet d'optimiser le nombre d'opérations à exécuter pour obtenir le résultat.

22.9.1. Les optimisations liées aux opérations de type short circuiting

Les traitements réalisés en interne par l'API Steam sont conçus pour l'être de manière optimisée, en tout cas dans le cadre de traitements génériques.

Dans l'exemple ci-dessous, trois opérations intermédiaires sont appliquées. Celles-ci ne sont pas appliquées sur tous les éléments de la source.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class Test {

  public static void main(String[] args) {
    List<Personne> personnes = new ArrayList<>(6);
    personnes.add(new Personne("p1", Genre.HOMME, 176));
    personnes.add(new Personne("p2", Genre.HOMME, 190));
    personnes.add(new Personne("p3", Genre.FEMME, 182));
    personnes.add(new Personne("p4", Genre.FEMME, 162));
    personnes.add(new Personne("p5", Genre.HOMME, 186));
    personnes.add(new Personne("p6", Genre.FEMME, 168));

    List<String> nomsDeuxPlusGrands = personnes.stream().filter(p -> {
      System.out.println("filter - " + p);
      return p.getTaille() > 180;
    }).map(p -> {
      System.out.println("map - " + p);
      return p.getNom();
    }).limit(2).collect(Collectors.toList());

    System.out.println("Resultat : " + nomsDeuxPlusGrands);
  }
}
```

Résultat :

```
filter - Personne [nom=p1, genre=HOMME, taille=176]
filter - Personne [nom=p2, genre=HOMME, taille=190]
map - Personne [nom=p2, genre=HOMME, taille=190]
filter - Personne [nom=p3, genre=FEMME, taille=182]
map - Personne [nom=p3, genre=FEMME, taille=182]
Resultat : [p2, p3]
```

Le filtre n'est appliqué que sur 3 éléments et la transformation uniquement sur 2 éléments

Les autres éléments ne sont plus traités une fois que l'opération limit() de type short-circuiting est satisfaite.

Les opérations intermédiaires possèdent une caractéristique intéressante : elles sont lazy.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.stream.Stream;

public class TestOperationsIntermediaires {

  public static void main(String[] args) {
    Stream.of("b2", "a1", "b1", "a2", "a3")
          .filter(e -> {System.out.println("filter : " + e);
                        return e.startsWith("a");
                       });
  }
}
```

Lors de l'exécution de cet exemple, aucun message n'est affiché à la console.

L'exécution des traitements des opérations n'est déclenchée que lors de l'invocation de la méthode terminale.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.stream.Stream;

public class TestOperationsIntermediaires {

  public static void main(String[] args) {
    Stream.of("b2", "a1", "b1", "a2", "a3")
          .peek(e -> System.out.println("filter : " + e))
          .filter(e -> e.startsWith("a"))
          .forEach(e -> System.out.println("foreach : " + e));
  }
}
```

Résultat :

```
filter : b2
filter : a1
foreach : a1
filter : b1
filter : a2
foreach : a2
filter : a3
foreach : a3
```

22.9.2. L'ordre des opérations d'un Stream

L'ordre des opérations définies dans le pipeline peut avoir un impact non négligeable sur les performances d'exécution des traitements.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.stream.Stream;

public class TestOperationsIntermediaires {

  public static void main(String[] args) {
    Stream.of("b2", "a1", "b1", "a2", "a3")
          .peek(e -> System.out.println("map : " + e))
          .map(String::toUpperCase)
          .peek(e -> System.out.println("filter : " + e))
          .filter(e -> e.startsWith("B"))
          .forEach(System.out::println);
  }
}
```

Résultat :

```
map: b2
filter: B2
B2
map: a1
filter: A1
map: b1
filter: B1
B1
map: a2
filter: A2
map: a3
filter: A3
```

Dans l'exemple ci-dessus, les opérations sont invoquées sur tous les éléments du Stream.

Il est possible d'optimiser ces traitements simplement en intervertissant les opérations pour d'abord filter les éléments et ainsi réduire le nombre d'éléments sur laquelle la transformation est exécutée.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.stream.Stream;

public class TestOperationsIntermediaires {

  public static void main(String[] args) {
    Stream.of("b2", "a1", "b1", "a2", "a3")
          .peek(e -> System.out.println("filter : " + e))
          .filter(e -> e.startsWith("b"))
          .peek(e -> System.out.println("map : " + e))
          .map(String::toUpperCase)
          .forEach(System.out::println);
  }
}
```

Résultat :

```
filter : b2
map : b2
B2
filter : a1
filter : b1
map : b1
B1
filter : a2
filter : a3
```

C'est encore plus important lorsqu'une opération stateful est utilisée. L'exemple ci-dessous trie les éléments puis les filtre et les transforme pour les afficher.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.stream.Stream;

public class TestOperationsIntermediaires {

  public static void main(String[] args) {
    Stream.of("b2", "a1", "b1", "a2", "a3")
          .sorted((e1, e2) -> {
            System.out.printf("sort: %s; %s\n", e1, e2);
            return e1.compareTo(e2);
          })
          .peek(e -> System.out.println("filter : " + e))
          .filter(e -> e.startsWith("b"))
          .peek(e -> System.out.println("map : " + e))
          .map(String::toUpperCase)
          .forEach(System.out::println);
  }
}
```

Résultat :

```
sort: a1; b2
sort: b1; a1
sort: b1; b2
sort: b1; a1
sort: a2; b1
sort: a2; a1
sort: a3; b1
sort: a3; a2
filter: a1
filter: a2
filter: a3
filter: b1
map: b1
B1
filter: b2
map: b2
B2
```

L'opération de tri requiert la comparaison successive d'éléments deux à deux pour assurer ses traitements. Dans ce cas, il est possible d'optimiser les traitements en effectuant d'abord l'opération de filtre et ainsi réduire le nombre d'éléments à trier.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.stream.Stream;

public class TestOperationsIntermediaires {

  public static void main(String[] args) {
    Stream.of("b2", "a1", "b1", "a2", "a3")
          .peek(e -> System.out.println("filter : " + e))
          .filter(e -> e.startsWith("b"))
          .sorted((e1, e2) -> {
            System.out.printf("sort: %s; %s\n", e1, e2);
            return e1.compareTo(e2);
          })
          .peek(e -> System.out.println("map : " + e))
          .map(String::toUpperCase)
          .forEach(System.out::println);
  }
}
```

Résultat :

```
filter: b2
filter: a1
filter: b1
filter: a2
filter: a3
sort: b1; b2
map: b1
B1
map: b2
B2
```

22.10. Les Streams infinis

Un Stream infini fait référence à un Stream dont la source produit de manière continue des éléments à consommer. Sans condition d'arrêt, cette génération peut être infinie comme son l'indique.

La mise en oeuvre de Stream infini est possible car le principe de traitements des éléments d'un Stream est lazy. Les opérations intermédiaires définissent les traitements mais ceux-ci ne sont réellement exécutés que lors de l'invocation de l'opération terminale.

Exemple (code Java 8) :

```
Stream<Integer> stream = Stream.iterate(0, i -> i + 1);
```

Attention lors de l'utilisation de Stream infinis : il est nécessaire de fournir une restriction qui va limiter le nombre d'éléments générés sinon le Stream va exécuter sans arrêt la génération de nouveaux éléments.

Exemple (code Java 8) :

```
// attention : ce code effectue une boucle infinie
IntStream.iterate(0, i -> i + 1).forEach(System.out::println);
```

L'exemple ci-dessous induit une boucle infinie.

Il faut impérativement introduire une condition d'arrêt par exemple en utilisant l'opération limit() qui permet de limiter le nombre d'éléments contenus dans le Stream.

Exemple (code Java 8) :

```
IntStream.iterate(0, i -> i + 1).limit(5).forEach(System.out::println);
```

Parfois les traitements infinis sont plus subtils et bien qu'il existe une condition d'arrêt, il est nécessaire de s'assurer que celle-ci sera vérifiée tôt ou tard.

Exemple (code Java 8) :

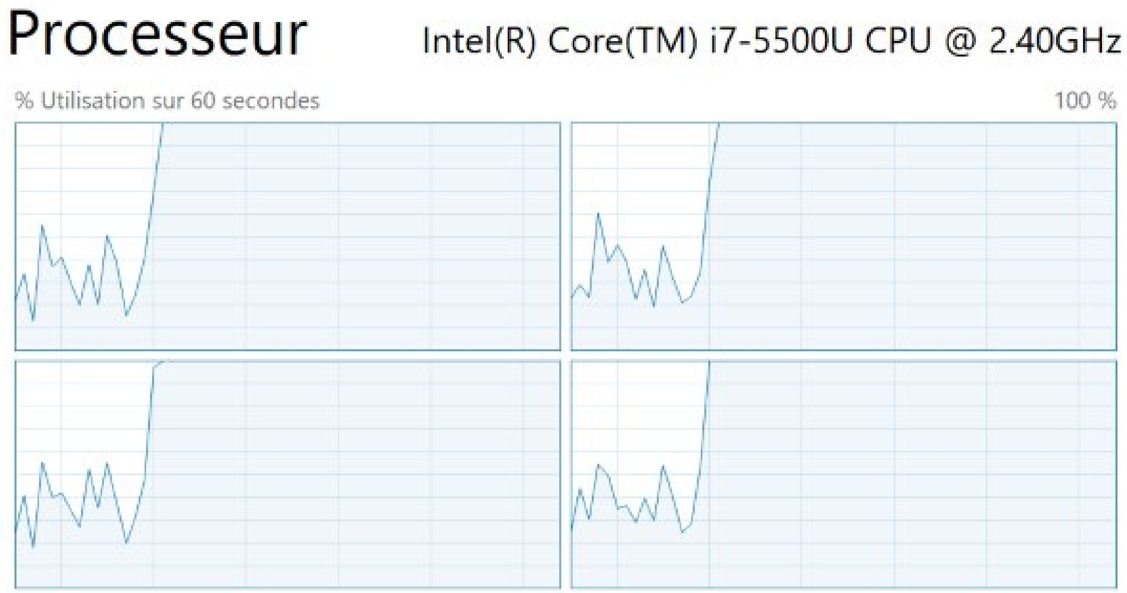
```
List<Double> valeur = Stream
    .generate(Math::random)
    .filter(v -> (v > 10) && (v < 20))
    .limit(10)
    .collect(Collectors.toList());
```

Dans l'exemple ci-dessus, les traitements du Stream ne s'arrêtent jamais bien qu'une opération limit(10) soit utilisée. Le problème vient du fait que la méthode random() de la classe Math renvoie une valeur comprise entre 0 et 1. Hors le filtre ne garde que les valeurs comprises entre 10 et 20 exclues. Aucune valeur générée n'est donc conservée par le filtre et la limite de 10 n'est jamais atteinte : ceci provoque une boucle infinie qui génère des nombres aléatoires.

Dans un Stream parallèle, c'est la catastrophe car tous les processeurs sont sollicités pour cette boucle infinie.

Exemple (code Java 8) :

```
List<Double> valeur = Stream
    .generate(Math::random)
    .parallel()
    .filter(v -> (v > 10) && (v < 20))
    .limit(10)
    .collect(Collectors.toList());
```


Processeur
Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz
% Utilisation sur 60 secondes
100 %

Même si ce n'est pas son but, il est possible d'utiliser un Stream infini pour réaliser l'équivalent d'une boucle. L'opération limit() permet de préciser le nombre d'itération qui sera réalisée.

Exemple (code Java 8) :

```
Stream.iterate(0, i -> i + 1)
      .limit(10)
      .forEach(System.out::println);
```

Cet exemple est équivalent à l'exemple ci-dessous

Exemple (code Java 8) :

```
i = 0;
while (i < 10) {
  System.out.println(i);
  i++;
}
```

Attention : dans des cas aussi basique que celui présenté ci-dessus, l'utilisation d'un Stream est moins performante et consomme plus de ressources.

L'opération generate() permet aussi de créer un Stream infini en lui passant en paramètre un Supplier qui aura la responsabilité de créer de nouveaux éléments.

Exemple (code Java 8) :

```
Stream<UUID> streamUUID = Stream.generate(UUID::randomUUID);
```

Comme avec la méthode iterate(), il est nécessaire de fournir une condition d'arrêt à la génération de nouveaux éléments par le Stream.

Exemple (code Java 8) :

```
List<UUID> valeurs = streamUUID
  .limit(5)
  .collect(Collectors.toList());
```

22.11. Le débogage d'un Stream

Le débogage d'un Stream n'est pas simple car une majorité des traitements est réalisée en interne par l'API.

Lorsqu'une anomalie survient dans les traitements d'un Stream, la stacktrace n'est généralement pas d'une grande aide.

Exemple (code Java 8) :

```
import java.util.stream.Stream;

public class DebugStream {

  public static void main(String[] args) {
    Double tailleMoyenne = Stream.of("texte", null, "grand texte")
       .mapToInt(String::length)
        .average()
        .getAsDouble();
    System.out.println("taille moyenne = " + tailleMoyenne);
  }
}
```

Résultat :

```
Exception in thread "main" java.lang.NullPointerException
      at java.util.stream.ReferencePipeline$4$1.accept(ReferencePipeline.java:210)
      at java.util.stream.ReferencePipeline$2$1.accept(ReferencePipeline.java:175)
      at java.util.Spliterators$ArraySpliterator.forEachRemaining(Spliterators.java:948)
      at java.util.stream.AbstractPipeline.copyInto(AbstractPipeline.java:481)
      at java.util.stream.AbstractPipeline.wrapAndCopyInto(AbstractPipeline.java:471)
      at java.util.stream.ReduceOps$ReduceOp.evaluateSequential(ReduceOps.java:708)
      at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:234)
      at java.util.stream.IntPipeline.collect(IntPipeline.java:472)
      at java.util.stream.IntPipeline.average(IntPipeline.java:434)
      at DebugStream.main(DebugStream.java:14)
```

Une exception de type NullPointerException est levée mais la lecture de la stacktrace ne fournit aucune information pour permettre de déterminer l'origine du problème.

Il est possible d'utiliser la méthode peek() pour afficher l'élément en cours de traitement dans le pipeline. C'est ailleurs normalement la seule raison d'utiliser la méthode peek() dans un pipeline d'opérations. Pour simplement afficher l'élément courant, il suffit de lui passer en paramètre la référence de méthode System.out::println. Si l'on doit utiliser la méthode peek() plusieurs fois dans le pipeline, il est préférable de construire une chaîne de caractères qui fournisse des informations complémentaires notamment l'étape courante dans le pipeline.

Exemple (code Java 8) :

```
import java.util.stream.Stream;

public class DebugStream {

  public static void main(String[] args) {
    Double tailleMoyenne = Stream.of("texte", null, "grand texte")
        .peek(e ->System.out.println("map : " + e))
        .mapToInt(String::length)
        .peek(e -> System.out.println("average : " + e))
        .average()
        .getAsDouble();
    System.out.println("taille moyenne = " + tailleMoyenne);
  }
}
```

Résultat :

```
map : texte
average : 5
map : null
Exception in thread "main" java.lang.NullPointerException
      at java.util.stream.ReferencePipeline$4$1.accept(ReferencePipeline.java:210)
      at java.util.stream.ReferencePipeline$11$1.accept(ReferencePipeline.java:373)
      at java.util.stream.ReferencePipeline$2$1.accept(ReferencePipeline.java:175)
      at java.util.stream.ReferencePipeline$11$1.accept(ReferencePipeline.java:373)
      at java.util.Spliterators$ArraySpliterator.forEachRemaining(Spliterators.java:948)
      at java.util.stream.AbstractPipeline.copyInto(AbstractPipeline.java:481)
      at java.util.stream.AbstractPipeline.wrapAndCopyInto(AbstractPipeline.java:471)
      at java.util.stream.ReduceOps$ReduceOp.evaluateSequential(ReduceOps.java:708)
      at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:234)
      at java.util.stream.IntPipeline.collect(IntPipeline.java:472)
      at java.util.stream.IntPipeline.average(IntPipeline.java:434)
      at DebugStream.main(DebugStream.java:14)
```

Si le problème est dans le code d'une expression lambda fournie en paramètre d'une opération, il est possible d'effectuer un refactoring pour inclure le code de l'expression dans une méthode et passer en paramètre de l'opération une référence de méthode permettant son invocation. Il suffit alors de mettre un point d'arrêt dans la méthode.

Les principaux IDE permettent de mettre des points d'arrêts pour faciliter le débogage du code fourni dans les expressions Lambda des opérations. Mais parfois cela ne suffit pas pour permettre de déboguer les traitements d'un Stream.

22.12. Les limitations de l'API Stream

L'API Stream permet la mise en oeuvre d'une approche fonctionnelle dans le langage Java. Java reste un langage orienté objet et n'est pas un langage fonctionnel. L'utilisation de l'API Stream n'est pas aussi riche ou poussée que dans d'autres langages qui sont fonctionnels.

L'API Stream possède aussi quelques limitations. Par exemple, il n'est pas possible de définir ses propres opérations : seules celles définies par l'API peuvent être utilisées.

22.12.1. Un Stream n'est pas réutilisable

Une fois qu'un Stream a été exécuté, il ne peut plus être réutilisé. Dès qu'une opération terminale est invoquée, celle-ci ferme le Stream.

Si le Stream est de nouveau invoqué, une exception de type IllegalStateException est levée.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.util.stream.Stream;

public class TestStream {

  public static void main(String[] args) {
    Stream<String> stream = Stream.of("a1", "a2", "a3")
                                  .filter(s -> s.startsWith("a"));
    stream.forEach(System.out::println);
    stream.forEach(System.out::println);
  }
}
```

Résultat :

```
a1
a2
a3
Exception in thread "main" java.lang.IllegalStateException: stream has already been
operated upon or closed
      at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:229)
      at java.util.stream.ReferencePipeline.forEach(ReferencePipeline.java:418)
      at fr.jmdoudoux.dej.streams.TestStream.main(TestStream.java:11)
```

Pour contourner cette limitation, il faut créer un nouveau Stream pour chaque utilisation. Si la configuration du Stream est complexe, il est possible de définir un Supplier dont le rôle sera de créer une instance à chaque invocation.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.streams;

import java.io.PrintStream;
import java.util.function.Supplier;
import java.util.stream.Stream;

public class TestStream {

  public static void main(String[] args) {
    Supplier<Stream<String>> supplier = () -> Stream.of("a1", "a2", "a3")
                    .filter(s -> s.startsWith("a"));
    supplier.get().forEach(System.out::println);
    supplier.get().forEach(System.out::println);
  }
}
```

Résultat :

```
a1
a2
a3
a1
a2
a3
```

22.13. Quelques recommandations sur l'utilisation de l'API Stream

L'API Stream permet de réaliser certains traitements sur un ensemble de données de manière déclarative, ce qui réduit la quantité de code à produire. La déclaration de certains traitements peut se faire de différentes manières pouvant avoir des impacts sur les performances.

Il ne faut pas utiliser systématiquement l'API Stream mais plutôt favoriser son utilisation lorsque celle-ci apporte une plus-value. Si l'API est utilisée, il est préférable de le faire de manière optimale.

22.13.1. L'utilisation à mauvais escient de l'API Stream

Parfois certaines utilisations de l'API Stream peuvent rendre le code moins lisible et compréhensible ou moins performant.

22.13.1.1. Le parcours des éléments

Il n'est pas utile d'avoir recours à un Stream pour appliquer un traitement sur les éléments d'une collection. C'est d'autant plus vrai si aucune opération intermédiaire n'est utilisée dans le Stream.

Exemple (code Java 8) :

```
import java.util.Arrays;
import java.util.List;

public class StreamBonnePratique {

  public static void main(String[] args) {
    String[] prenomsTab = { "alain", "anne", "sophie", "thierry", "antoine", "pierre" };
    List<String> prenoms = Arrays.asList(prenomsTab);
    prenoms.stream()
           .forEach(System.out::println);
  }
}
```

Il est préférable d'utiliser la méthode forEach() sur la collection sans utiliser un Stream.

Exemple (code Java 8) :

```
import java.util.Arrays;
import java.util.List;

public class StreamBonnePratique {

  public static void main(String[] args) {
    String[] prenomsTab = { "alain", "anne", "sophie", "thierry", "antoine", "pierre" };
    List<String> prenoms = Arrays.asList(prenomsTab);
    prenoms.forEach(System.out::println);
  }
}
```

L'utilisation de cette méthode garantie l'ordre des éléments lors du parcours si la collection est ordonnée.

22.13.1.2. Le remplacement des boucles for par un Stream

Il est courant de voir le remplacement des boucles for par un Stream de manière un peu systématique.

Exemple (code Java 8) :

```
import java.util.stream.IntStream;

public class StreamBonnePratique {

  public static void main(String[] args) {
    for (int i = 0; i < 5; i++) {
      System.out.println(i);
    }
    IntStream.range(0, 5)
             .forEach(System.out::println);
  }
}
```

Evidemment cet exemple est très (trop) simple. L'exemple ci-dessous qui imbrique deux itérations est probablement plus lisible et donc maintenable avec des boules for qu'avec leur équivalent utilisant des Streams

Exemple (code Java 8) :

```
import java.util.stream.IntStream;

public class StreamBonnePratique {

  public static void main(String[] args) {
    System.out.println("Tables de multiplications");
    for (int i = 0; i < 10; i++) {
      for (int j = 0; j < 10; j++)
        System.out.format("%2d ", i * j);
      System.out.println();
    }
    System.out.println("Tables de multiplications");
    IntStream.range(0, 10)
             .forEach(i -> {
                 IntStream.range(0, 10)
                          .forEach(j -> { System.out.format("%2d ", i * j); });
               System.out.println();
             });
  }
}
```

Si les traitements de la boucle for sont simples ou stateful ou qu'il n'est pas prévu de les paralléliser alors généralement leur remplacement par un Stream rend le code parfois moins maintenable et/ou moins performant.

Le remplacement d'une boucle for par un Stream n'est réellement intéressant que si l'approche fonctionnelle s'appuyant sur une itération interne est utilisée pour par exemple chaîner plusieurs opérations dans le but d'obtenir un résultat.

Ceci ne concerne pas forcément que l'API Stream : c'est généralement le cas lorsqu'on remplace du code de base par l'utilisation d'un framework. On gagne en fonctionnalité et en souplesse mais on perd en performance, d'autant plus si aucune attention particulière n'est prêtée lors de la mise en oeuvre du framework.

Une autre raison est la clarté des stacktraces si une exception est levée durant les traitements des itérations. Celles-ci sont très claires dans le cas d'une boucle for.

Exemple (code Java 8) :

```
public class StreamBonnePratique {

  public static void main(String[] args) {
    for (int i = 0; i < 10; i++) {
      System.out.println(i / (9 - i));
    }
  }
}
```

Résultat :

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
      atStreamBonnePratique.main(StreamBonnePratique.java:6)
```

Elles sont plus verbeuses dans le cas d'un Stream

Exemple (code Java 8) :

```
import java.util.stream.IntStream;

public class StreamBonnePratique {

  public static void main(String[] args) {
    IntStream.range(0, 10)
             .forEach(i -> {
                System.out.println(i / (9 - i));
             });
  }
}
```

Résultat :

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
      at StreamBonnePratique.lambda$0(StreamBonnePratique.java:9)
      at java.util.stream.Streams$RangeIntSpliterator.forEachRemaining(Streams.java:110)
      at java.util.stream.IntPipeline$Head.forEach(IntPipeline.java:557)
      at StreamBonnePratique.main(StreamBonnePratique.java:8)
```

22.13.1.3. La conversion d'une collection

Il n'est pas utile d'avoir recours à un Stream pour convertir les éléments d'un tableau en une collection de type List.

Exemple (code Java 8) :

```
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class StreamBonnePratique {

  public static void main(String[] args) {
    String[] prenomsTab = { "alain", "anne", "sophie", "thierry", "antoine", "pierre" };
    List<String> prenoms = Stream.of(prenomsTab)
                                 .collect(Collectors.toList());
  }
}
```

Il est préférable d'utiliser la méthode toList() de la classe Arrays.

Exemple (code Java 8) :

```
import java.util.Arrays;
import java.util.List;

public class StreamBonnePratique {

  public static void main(String[] args) {
    String[] prenomsTab = { "alain", "anne", "sophie", "thierry", "antoine", "pierre" };
    List<String> prenoms = Arrays.asList(prenomsTab);
  }
}
```

De la même manière, il n'est pas nécessaire d'utiliser un Stream pour convertir une collection vers une autre collection.

Exemple (code Java 8) :

```
import java.util.Arrays;
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;

public class StreamBonnePratique {

  public static void main(String[] args) {
    String[] prenomsTab = { "alain", "anne", "sophie", "thierry", "antoine", "pierre" };
    List<String> prenomsList = Arrays.asList(prenomsTab);
    Set<String> prenoms = prenomsList.stream()
                                     .collect(Collectors.toSet());
  }
}
```

Il est préférable d'utiliser la surcharge du constructeur des collections qui attend en paramètre une collection.

Exemple (code Java 8) :

```
import java.util.Arrays;
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;

public class StreamBonnePratique {

  public static void main(String[] args) {
    String[] prenomsTab = { "alain", "anne", "sophie", "thierry", "antoine",  "pierre" };
    List<String> prenomsList = Arrays.asList(prenomsTab);
    Set<String> prenoms = new HashSet<>(prenomsList);
  }
}
```

Enfin, il n'est pas nécessaire d'utiliser un Stream pour convertir une collection en un tableau.

Exemple (code Java 8) :

```
import java.util.Arrays;
import java.util.List;

public class StreamBonnePratique {

  public static void main(String[] args) {
    String[] prenomsTab = { "alain", "anne", "sophie", "thierry", "antoine", "pierre" };
    List<String> prenomsList = Arrays.asList(prenomsTab);
    String[] prenoms = prenomsList.stream()
                                  .toArray(String[]::new);
  }
}
```

Il est préférable d'utiliser la méthode toArray() de l'interface Collection.

Exemple (code Java 8) :

```
import java.util.Arrays;
import java.util.List;

public class StreamBonnePratique {

  public static void main(String[] args) {
    String[] prenomsTab = { "alain", "anne", "sophie", "thierry", "antoine", "pierre" };
    List<String> prenomsList = Arrays.asList(prenomsTab);
    String[] prenoms = prenomsList.toArray(new String[0]);
  }
}
```

22.13.1.4. La recherche du plus grand élément d'une collection

Il n'est pas nécessaire d'utiliser un Stream pour uniquement trouver le plus grand élément dans une collection.

Exemple (code Java 8) :

```
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;

public class StreamBonnePratique {

  public static void main(String[] args) {
    String[] prenomsTab = { "alain", "anne", "sophie", "thierry", "antoine", "pierre" };
    List<String> prenomsList = Arrays.asList(prenomsTab);
    String plusGrand = prenomsList.stream()
                                  .max(Comparator.naturalOrder())
                                  .orElse(null);
    System.out.println(plusGrand);
  }
}
```

Il est préférable d'utiliser la méthode max() de la classe Collections.

Exemple (code Java 8) :

```
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class StreamBonnePratique {

  public static void main(String[] args) {
    String[] prenomsTab = { "alain", "anne", "sophie", "thierry", "antoine", "pierre" };
    List<String> prenomsList = Arrays.asList(prenomsTab);
    String plusGrand = prenomsList.isEmpty() ? null
       : Collections.max(prenomsList, Comparator.naturalOrder());
    System.out.println(plusGrand);
  }
}
```

22.13.1.5. La détermination du nombre d'éléments d'une collection

Il n'est pas nécessaire d'utiliser un Stream pour uniquement déterminer le nombre d'éléments d'une collection.

Exemple (code Java 8) :

```
import java.util.Arrays;
import java.util.List;

public class StreamBonnePratique {

  public static void main(String[] args) {
    String[] prenomsTab = { "alain", "anne", "sophie", "thierry", "antoine", "pierre" };
    List<String> prenomsList = Arrays.asList(prenomsTab);
    long nbElements = prenomsList.stream()
                                 .count();
  }
}
```

Il est préférable d'utiliser la méthode size() de l'interface Collection.

Exemple (code Java 8) :

```
import java.util.Arrays;
import java.util.List;

public class StreamBonnePratique {

  public static void main(String[] args) {
    String[] prenomsTab = { "alain", "anne", "sophie", "thierry", "antoine", "pierre" };
    List<String> prenomsList = Arrays.asList(prenomsTab);
    long nbElements = prenomsList.size();
  }
}
```

22.13.1.6. La vérification de la présence d'un élément dans une collection

Il n'est pas nécessaire d'utiliser un Stream pour uniquement vérifier la présence d'un élément dans une collection.

Exemple (code Java 8) :

```
import java.util.Arrays;
import java.util.List;

public class StreamBonnePratique {

  public static void main(String[] args) {
    String[] prenomsTab = { "alain", "anne", "sophie", "thierry", "antoine", "pierre" };
    List<String> prenomsList = Arrays.asList(prenomsTab);
    boolean trouve = prenomsList.stream()
                                .anyMatch(s -> "pierre".equals(s));
  }
}
```

Il est préférable d'utiliser la méthode contains() de l'interface Collection.

Exemple (code Java 8) :

```
import java.util.Arrays;
import java.util.List;

public class StreamBonnePratique {

  public static void main(String[] args) {
    String[] prenomsTab = { "alain", "anne", "sophie", "thierry", "antoine", "pierre" };
    List<String> prenomsList = Arrays.asList(prenomsTab);
    boolean trouve = prenomsList.contains("pierre");
  }
}
```

Ceci est particulièrement vrai pour une collection de type Set.

22.13.2. L'utilisation de l'API Stream pour rechercher des éléments

Certaines recherches d'éléments peuvent parfois se faire de différentes manières. Certaines sont plus intéressantes d'un point de vue performance et/ou maintenabilité.

22.13.2.1. La recherche de la présence d'un élément

Il n'est pas utile de filtrer les éléments et de vérifier s'il y a un premier élément restant pour s'assurer qu'un élément est présent dans une collection.

Exemple (code Java 8) :

```
mport java.util.Arrays;
import java.util.List;

public class StreamBonnePratique {

  public static void main(String[] args) {
    String[] prenomsTab = { "alain", "anne", "sophie", "thierry", "antoine", "pierre" };
    List<String> prenomsList = Arrays.asList(prenomsTab);
    boolean trouve = prenomsList.stream()
                                .filter(s -> "pierre".equals(s))
                                .findFirst()
                                .isPresent();
  }
}
```

Il est préférable d'utiliser l'opération terminale anyMatch() du Stream.

Exemple (code Java 8) :

```
mport java.util.Arrays;
import java.util.List;

public class StreamBonnePratique {

  public static void main(String[] args) {
    String[] prenomsTab = { "alain", "anne", "sophie", "thierry", "antoine", "pierre" };
    List<String> prenomsList = Arrays.asList(prenomsTab);
    boolean trouve = prenomsList.stream()
                                .anyMatch(s-> "pierre".equals(s));
  }
}
```

22.13.2.2. La recherche du plus petit élément

Il n'est pas utile de trier les éléments dans un ordre croissant et de prendre le premier pour rechercher le plus petit élément.

Exemple (code Java 8) :

```
import java.util.Comparator;
import java.util.Optional;
import java.util.stream.Stream;

public class StreamBonnePratique {

  public static void main(String[] args) {
    Stream<String> prenoms = Stream.of("pierre", "anne", "sophie", "thierry", "antoine");
    Optional<String> premier = prenoms.sorted()
                                      .findFirst();
    System.out.println(premier.orElse("inconnu"));
  }
}
```

Il est préférable d'utiliser l'opération min() du Stream.

Exemple (code Java 8) :

```
import java.util.Optional;
import java.util.stream.Stream;

public class StreamBonnePratique {

  public static void main(String[] args) {
    Stream<String> prenoms = Stream.of("pierre", "anne", "sophie", "thierry", "antoine");
    Optional<String> premier = prenoms.min(Comparator.naturalOrder());
    System.out.println(premier.orElse("inconnu"));
  }
}
```

22.13.2.3. La recherche du plus grand élément

Il n'est pas utile de trier les éléments dans un ordre décroissant et de prendre le premier pour rechercher le plus grand élément.

Exemple (code Java 8) :

```
import java.util.Comparator;
import java.util.Optional;
import java.util.stream.Stream;

public class StreamBonnePratique {

  public static void main(String[] args) {
    Stream<String> prenoms = Stream.of("pierre", "anne", "sophie", "thierry", "antoine");
    Optional<String> premier = prenoms.sorted(Comparator.reverseOrder())
                                      .findFirst();
    System.out.println(premier.orElse("inconnu"));
  }
}
```

Il est préférable d'utiliser l'opération max() du Stream.

Exemple (code Java 8) :

```
import java.util.Comparator;
import java.util.Optional;
import java.util.stream.Stream;

public class StreamBonnePratique {

  public static void main(String[] args) {
    Stream<String> prenoms = Stream.of("pierre", "anne", "sophie", "thierry", "antoine");
    Optional<String> premier = prenoms.max(Comparator.naturalOrder());
    System.out.println(premier.orElse("inconnu"));
  }
}
```

22.13.3. La création de Streams

Il est préférable d'utiliser les fabriques dédiées de l'API Stream plutôt que de créer une collection à partir de laquelle on obtient un Stream pour traiter les éléments.

22.13.3.1. La création d'un Stream vide

Il n'est pas utile de créer une collection vide et de demander un Stream sur celle-ci pour obtenir un Stream vide.

Exemple (code Java 8) :

```
import java.util.Collections;
import java.util.stream.Stream;

public class StreamBonnePratique {

  public static void main(String[] args) {
    Stream<Object> streamVide = Collections.emptyList()
                                           .stream();
  }
}
```

Il est préférable d'utiliser la méthode empty() de l'interface Stream.

Exemple (code Java 8) :

```
import java.util.stream.Stream;

public class StreamBonnePratique {

  public static void main(String[] args) {
    Stream<Object> streamVide = Stream.empty();
  }
}
```

22.13.3.2. La création d'un Stream avec un seul élément

Il n'est pas utile de créer une collection avec un seul élément de type Set (avec la méthode singleton()) ou List (avec la méthode singletonList()) et de demander un Stream sur celle-ci pour obtenir un Stream.

Exemple (code Java 8) :

```
import java.util.Collections;
import java.util.stream.Stream;

public class StreamBonnePratique {

  public static void main(String[] args) {
    Stream<String> stream = Collections.singleton("test")
                                       .stream();
  }
}
```

Il est préférable d'utiliser la méthode of() de l'interface Stream.

Exemple (code Java 8) :

```
import java.util.stream.Stream;

public class StreamBonnePratique {

  public static void main(String[] args) {
    Stream<String> stream = Stream.of("test");
  }
}
```

22.13.3.3. La création d'un Stream à partir d'un tableau

Il n'est pas utile de créer une collection avec les éléments du tableau et demander un Stream sur celle-ci pour obtenir un Stream.

Exemple (code Java 8) :

```
import java.util.Collections;
import java.util.stream.Stream;

public class StreamBonnePratique {

  public static void main(String[] args) {
    String[] prenomsTab = { "alain", "anne", "sophie", "thierry", "antoine", "pierre" };
    Stream<String> prenoms = Arrays.asList(prenomsTab).stream();
  }
}
```

Il est préférable d'utiliser la méthode of() de l'interface Stream ou la méthode stream() de la classe Arrays.

Exemple (code Java 8) :

```
import java.util.stream.Stream;

public class StreamBonnePratique {

  public static void main(String[] args) {
    String[] prenomsTab = { "alain", "anne", "sophie", "thierry", "antoine", "pierre" };
    Stream<String> prenoms = Stream.of(prenomsTab);
    Stream<String> prenoms2 =Arrays.stream(prenomsTab);
  }
}
```

22.13.3.4. Le traitement d'une plage d'éléments d'un tableau

Il est pas utile d'utiliser la fabrique range() de l'interface IntStream puis de transformer chaque valeur en l'élément correspondant dans le tableau avec l'opération mapToObj().

Exemple (code Java 8) :

```
import java.util.stream.IntStream;

public class StreamBonnePratique {

  public static void main(String[] args) {
    String[] prenomsTab = { "alain", "anne", "sophie", "thierry", "antoine", "pierre" };
    long nb = IntStream.range(2, 5)
                       .mapToObj(idx -> prenomsTab[idx])
                       .count();
    System.out.println(nb);
  }
}
```

Il est préférable d'utiliser la surcharge de la méthode stream() de la classe Arrays qui attend en paramètre le tableau, l'index de début et l'index de fin de la plage des éléments à traiter.

Exemple (code Java 8) :

```
import java.util.Arrays;

public class StreamBonnePratique {

  public static void main(String[] args) {
    String[] prenomsTab = { "alain", "anne", "sophie", "thierry", "antoine", "pierre" };
    long nb = Arrays.stream(prenomsTab, 2, 5)
                    .count();
    System.out.println(nb);
  }
}
```

22.13.4. L'utilisation non requise d'un Collector

Certains Collector sont conçus pour être utilisés comme des downstreams Collector de l'opération groupingBy(). Il n'est pas nécessaire de les utiliser à la place des opérations de l'interface Stream équivalente.

Par exemple, pour compter simplement les éléments d'un Stream, il n'est pas utile d'utiliser le Collector obtenu par la fabrique counting().

Exemple (code Java 8) :

```
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class StreamBonnePratique {

  public static void main(String[] args) {
    Stream<String> prenoms = Stream.of("pierre", "anne", "sophie", "thierry", "antoine");
    long nb = prenoms.collect(Collectors.counting());
    System.out.println(nb);
  }
}
```

Il est préférable d'utiliser l'opération count() de l'interface Stream.

Exemple (code Java 8) :

```
import java.util.stream.Stream;
```

```
public class StreamBonnePratique {

  public static void main(String[] args) {
    Stream<String> prenoms = Stream.of("pierre", "anne", "sophie", "thierry", "antoine");
    long nb = prenoms.count();
    System.out.println(nb);
  }
}
```

De même, pour déterminer le plus petit ou le plus grand élément d'un Stream, il n'est pas utile d'utiliser le Collector obtenu par la fabrique minBy() ou maxBy().

Exemple (code Java 8) :

```
import java.util.Comparator;
import java.util.Optional;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class StreamBonnePratique {

  public static void main(String[] args) {
    Stream<String> prenoms = Stream.of("pierre", "anne", "sophie", "thierry", "antoine");
    Optional<String> plusGrand = prenoms.collect(Collectors.maxBy(Comparator.naturalOrder()));
    System.out.println(plusGrand.orElse("inconnu"));
  }
}
```

Il est préférable d'utiliser l'opération min() ou max() de l'interface Stream.

Exemple (code Java 8) :

```
import java.util.Comparator;
import java.util.Optional;
import java.util.stream.Stream;

public class StreamBonnePratique {

  public static void main(String[] args) {
    Stream<String> prenoms = Stream.of("pierre", "anne", "sophie", "thierry", "antoine");
    Optional<String> plusGrand = prenoms.max(Comparator.naturalOrder());
    System.out.println(plusGrand.orElse("inconnu"));
  }
}
```

De la même manière, il est préférable d'utiliser :

- L'opération reduce() plutôt que le Collector obtenu par la méthode reducing() de la classe Collectors
- L'opération map() plutôt que le Collector obtenu par la méthode mapping() de la classe Collectors
- L'opération filter() plutôt que le Collector obtenu par la méthode filtering() de la classe Collectors
- L'opération flatMap() plutôt que le Collector obtenu par la méthode flatMapping() de la classe Collectors

22.13.5. Le traitement de valeurs numériques

Pour éviter des opérations d'autoboxing et donc d'améliorer les performances, il n'est pas recommandé d'effectuer des calculs sur des wrappers de type primitif numérique.

Exemple (code Java 8) :

```
import java.util.stream.Stream;
```

```
public class StreamBonnePratique {

  public static void main(String[] args) {
    Personne p1 = new Personne("alain", 179);
    Personne p2 = new Personne("sopihe", 176);
    Personne p3 = new Personne("thierry", 172);

    int somme = Stream.of(p1, p2, p3)
                      .map(p -> p.getTaille())
                      .reduce(0, (a, b) -> a + b);
    System.out.println(somme);
  }
}
```

Il est préférable d'utiliser un IntStream, LongStream ou DoubleStream pour améliorer les performances, réduire le nombre d'objets créés et réduire la quantité de code nécessaire.

Exemple (code Java 8) :

```
import java.util.stream.Stream;

public class StreamBonnePratique {

  public static void main(String[] args) {
    Personne p1 = new Personne("alain", 179);
    Personne p2 = new Personne("sopihe", 176);
    Personne p3 = new Personne("thierry", 172);

    int somme = Stream.of(p1, p2, p3)
                      .mapToInt(p -> p.getTaille())
                      .sum();
    System.out.println(somme);
  }
}
```

22.13.6. Ne compter les éléments que si c'est nécessaire

Certains traitements exécutés en comptant le nombre d'éléments sont assez peu performants et il est parfois préférable d'utiliser une autre approche notamment si le nombre d'éléments est important.

22.13.6.1. La détermination du nombre d'éléments de sous-ensembles

Par exemple : des groupes contiennent des personnes et on souhaite calculer le nombre de personnes incluses dans un ensemble de groupes.

Une première approche consiste à mettre à plat chaque personne de chaque groupe en utilisant l'opération flatMap() et compter le nombre de personnes obtenues.

Exemple (code Java 8) :

```
import java.util.Arrays;
import java.util.stream.Stream;

public class StreamBonnePratique {

  public static void main(String[] args) {
    Personne p1 = new Personne("Alain", 179);
    Personne p2 = new Personne("Sophie", 176);
    Personne p3 = new Personne("Thierry", 172);
    Personne p4 = new Personne("Martine", 169);
    Personne p5 = new Personne("Paul", 174);
    Personne p6 = new Personne("Isabelle", 182);
    Personne p7 = new Personne("Jean", 172);
    Groupe g1 = new Groupe("Groupe1", Arrays.asList(p1, p2, p3));
    Groupe g2 = new Groupe("Groupe2", Arrays.asList(p4, p5));
    Groupe g3 = new Groupe("Groupe3", Arrays.asList(p6, p7));

    long somme = Stream.of(g1, g2, g3)
                       .flatMap(g -> g.getPersonnes()
                                      .stream())
                       .count();
    System.out.println(somme);
  }
}
```

Pour des raisons de performances, il est préférable de faire directement la somme de la taille des collections de personnes de chaque groupe. Cela évite le parcours de chaque collection et la création inutile d'objets.

Exemple (code Java 8) :

```
import java.util.Arrays;
import java.util.stream.Stream;

public class StreamBonnePratique {

  public static void main(String[] args) {
    Personne p1 = new Personne("Alain", 179);
    Personne p2 = new Personne("Sophie", 176);
    Personne p3 = new Personne("Thierry", 172);
    Personne p4 = new Personne("Martine", 169);
    Personne p5 = new Personne("Paul", 174);
    Personne p6 = new Personne("Isabelle", 182);
    Personne p7 = new Personne("Jean", 172);

    Groupe g1 = new Groupe("Groupe1", Arrays.asList(p1, p2, p3));
    Groupe g2 = new Groupe("Groupe2", Arrays.asList(p4, p5));
    Groupe g3 = new Groupe("Groupe3", Arrays.asList(p6, p7));

    long somme = Stream.of(g1, g2, g3)
                       .mapToLong(g -> g.getPersonnes()
                                        .size())
                     .sum();
    System.out.println(somme);
  }
}
```

22.13.6.2. La vérification qu'au moins un élément satisfasse une condition

Il n'est pas nécessaire de filtrer les éléments, de compter le nombre d'éléments et de vérifier si ce nombre est strictement supérieur à zéro.

Exemple (code Java 8) :

```
import java.util.stream.Stream;

public class StreamBonnePratique {

  public static void main(String[] args) {
    Personne p1 = new Personne("Alain", 189);
    Personne p2 = new Personne("Sophie", 176);
    Personne p3 = new Personne("Thierry", 172);
    Personne p4 = new Personne("Isabelle", 182);
    Personne p5 = new Personne("Jean", 172);

    long nbGrandePers = Stream.of(p1, p2, p3, p4, p5)
                              .filter(p -> p.getTaille() >= 180)
                              .count();
    boolean presenceGrandePers = nbGrandePers > 0;
    System.out.println(presenceGrandePers);
  }
}
```

Il est préférable d'utiliser l'opération terminale anyMatch() qui attend en paramètre la condition à vérifier. L'intérêt de cette opération est qu'elle est short-circuiting : dès qu'un élément répond à la condition, elle s'arrête et renvoie le résultat. Dans le cas précédent, le filtre est appliqué sur tous les éléments avant de compter leur nombre.

Exemple (code Java 8) :

```
import java.util.stream.Stream;

public class StreamBonnePratique {

  public static void main(String[] args) {
    Personne p1 = new Personne("Alain", 189);
    Personne p2 = new Personne("Sophie", 176);
    Personne p3 = new Personne("Thierry", 172);
    Personne p4 = new Personne("Isabelle", 182);
    Personne p5 = new Personne("Jean", 172);

    boolean presenceGrandePers = Stream.of(p1, p2, p3, p4, p5)
                                       .anyMatch(p -> p.getTaille() >= 180);
    System.out.println(presenceGrandePers);
  }
}
```

22.13.6.3. La vérification qu'aucun élément ne satisfasse une condition

Comme précédemment, il n'est pas nécessaire de filtrer les éléments, de compter le nombre d'éléments et de vérifier si ce nombre est égal à zéro.

Exemple (code Java 8) :

```
import java.util.stream.Stream;

public class StreamBonnePratique {

  public static void main(String[] args) {
    Personne p1 = new Personne("Alain", 189);
    Personne p2 = new Personne("Sophie", 176);
    Personne p3 = new Personne("Thierry", 172);
    Personne p4 = new Personne("Isabelle", 182);
    Personne p5 = new Personne("Jean", 172);

    long nbGrandePers = Stream.of(p1, p2, p3, p4, p5)
                              .filter(p -> p.getTaille() >= 180)
                              .count();
    boolean aucuneGrandePers = nbGrandePers == 0;
    System.out.println(aucuneGrandePers);
  }
}
```

Il est préférable d'utiliser l'opération terminale noneMatch() qui attend en paramètre la condition à vérifier. L'intérêt de cette opération est qu'elle est short-circuiting : dès qu'un élément ne répond pas à la condition, elle s'arrête et renvoie le résultat. Dans le cas précédent, le filtre est appliqué sur tous les éléments avant de compter leur nombre.

Exemple (code Java 8) :

```
import java.util.stream.Stream;

public class StreamBonnePratique {

  public static void main(String[] args) {
    Personne p1 = new Personne("Alain", 189);
    Personne p2 = new Personne("Sophie", 176);
    Personne p3 = new Personne("Thierry", 172);
    Personne p4 = new Personne("Isabelle", 182);
    Personne p5 = new Personne("Jean", 172);
    boolean aucuneGrandePers = Stream.of(p1, p2, p3, p4, p5)
                                     .noneMatch(p -> p.getTaille() >= 180);
    System.out.println(aucuneGrandePers);
  }
}
```

22.13.6.4. La vérification qu'au moins N éléments satisfassent une condition

Une première approche est de filtrer les éléments, de compter le nombre d'éléments et de vérifier si ce nombre est supérieur ou égal au nombre N.

Exemple (code Java 8) :

```
import java.util.stream.Stream;

public class StreamBonnePratique {

  public static void main(String[] args) {
    Personne p1 = new Personne("Alain", 189);
    Personne p2 = new Personne("Sophie", 176);
    Personne p3 = new Personne("Thierry", 172);
    Personne p4 = new Personne("Isabelle", 182);
    Personne p5 = new Personne("Jean", 191);
    Personne p6 = new Personne("Pierre", 165);

    long nbGrandePers = Stream.of(p1, p2, p3, p4, p5, p6)
                              .filter(p -> p.getTaille() >= 180)
                              .count();
    boolean auMoinsDeuxGrandePers = nbGrandePers >= 2;
    System.out.println(auMoinsDeuxGrandePers);
  }
}
```

Pour améliorer les performances, notamment si le nombre d'éléments à traiter est important, il est possible d'utiliser l'opération intermédiaire limit() qui est short-circuiting en lui passant en paramètre le nombre d'éléments qui doivent satisfaire la condition.

Exemple (code Java 8) :

```
import java.util.stream.Stream;

public class StreamBonnePratique {

  public static void main(String[] args) {
    Personne p1 = new Personne("Alain", 189);
    Personne p2 = new Personne("Sophie", 176);
    Personne p3 = new Personne("Thierry", 172);
    Personne p4 = new Personne("Isabelle", 182);
    Personne p5 = new Personne("Jean", 191);
    Personne p6 = new Personne("Pierre", 165);

    long nbGrandePers = Stream.of(p1, p2, p3, p4, p5, p6)
                              .filter(p -> p.getTaille() >= 180)
                              .limit(2)
                              .count();
    boolean auMoinsDeuxGrandePers = nbGrandePers >= 2;
    System.out.println(auMoinsDeuxGrandePers);
  }
}
```

23. Les expressions régulières

Chapitre 23

Niveau : 🟩🟩🟨⬜⬜⬜ Intermédiaire

Historiquement, les expressions régulières introduites en Java 1.4 sont la forme la plus ancienne de pattern matching proposée par Java.

Les expressions régulières, aussi appelées expressions rationnelles ou abrégées en regex, permettent de rechercher un motif avec une syntaxe particulière dans du texte.

Les cas d'usage sont nombreux notamment :

- rechercher un motif dans unr chaîne de caractères, un fichier, ...
- valider des contraintes sur des données saisies par l'utilisateur
- remplacer un ensemble de caractères par un autre

Une expression régulière est un motif sous la forme d'une chaîne de caractères qui décrit avec une syntaxe particulière un ensemble de caractères dont la correspondance exacte ou multiple pourra être recherchée dans une chaîne de caractères.

Le motif de recherche peut être un simple caractère, une chaîne fixe ou une expression complexe contenant des caractères spéciaux composant le motif.

Exemple : une expression régulière qui correspond à un ensemble d'un ou plusieurs chiffres

Exemple :

```
    String MOTIF_NOMBRE = "[0-9]+";
```

Les expressions régulières peuvent être plus ou moins complexes selon les besoins.

Exemple :

```
    String MOTIF_IPV4 = "^([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\\."
        + "([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\\."
        + "([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\\."
        + "([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])$";
```

Les expressions régulières sont supportées par de nombreux langages et outils malheureusement avec une syntaxe parfois différente ou des fonctionnalités dédiées. Dans le monde des expressions régulières, il existe différentes syntaxes proposées par différents langages ou outils, tels que Perl, Python, PHP, grep, awk, sed, etc . La syntaxe d'une expression régulière n'est donc pas forcément portable. La syntaxe des expressions régulières en Java est très similaire à celle que l'on trouve en Perl.

Le motif qui est défini par une expression régulière peut correspondre :

- à l'intégralité de la chaîne
- ou zéro ou plusieurs fois dans la chaîne
- ou ne pas correspondre du tout à la chaîne

Ce chapitre contient plusieurs sections :

- ♦ Le package java.util.regex
- ♦ La mise en oeuvre des expressions régulières
- ♦ Les remplacements de texte
- ♦ L'utilisation d'expressions régulières dans les méthodes de la classe String

23.1. Le package java.util.regex

Le support des expressions régulières est proposé dans les types du package java.util.regex qui contient :

- la classe Pattern : encapsule un motif
- la classe Matcher : applique un motif sur une chaîne de caractères
- l'exception PatternSyntaxException : levée sur si une erreur de syntaxe est détectée dans un motif
- l'interface MatchResult

La mise en oeuvre des expressions régulières combine l'utilisation des classes Pattern et Matcher :

- il faut obtenir une instance de type Pattern qui encapsule l'expression régulière
- cet objet Pattern permet de créer une instance de type Matcher pour une chaîne de caractères donnée
- cet objet Matcher permet d'effectuer des opérations en utilisant la regex sur une chaîne de caractères

23.1.1. La classe Pattern

La classe Pattern encapsule une représentation compilée d'une expression régulière.

Une expression régulière, fournie sous la forme d'une chaîne de caractères doit être compilée et encapsulée dans une instance de type Pattern.

La classe Pattern propose plusieurs méthodes :

Méthode	Rôle
Predicate<String> asPredicate()	Renvoyer un Predicate qui teste si le motif est trouvé dans une chaîne d'entrée donnée en utilisant la méthode find() d'une instance de Matcher (depuis Java 8)
Predicate<String> asMatchPredicate()	Renvoyer un Predicate qui teste si le motif correspond à la chaîne d'entrée donnée en utilisant la méthode matcher() d'une instance de Matcher (depuis Java 11)
static Pattern compile(String regex)	Compiler l'expression régulière fournie et retourner une instance de type Pattern qui l'encapsule
static Pattern compile(String regex, int flags)	Renvoyer une instance de type Pattern qui encapsule une version compilée de l'expression régulière et les options fournies en paramètres
int flags()	Renvoyer les options de correspondance utilisées
Matcher matcher(CharSequence input)	Renvoyer une instance de type Matcher qui applique l'expression sur la chaîne fournie en paramètre
static boolean matches(String regex, CharSequence input)	Compiler l'expression fournie et l'appliquer sur la chaîne fournie en paramètre pour renvoyer le résultat de la correspondance
String pattern()	Renvoyer l'expression régulière utilisée pour créer l'instance
static String quote(String s)	Renvoyer une chaîne littérale du motif pour la chaîne fournie en paramètre (depuis Java 1.5)
String[] split(CharSequence input)	

	Découper la chaîne de caractères en utilisant l'expression régulière comme séparateur
String[] split(CharSequence input, int limit)	Découper la chaîne de caractères en utilisant l'expression régulière comme séparateur avec un nombre maximum de sous-chaînes
Stream<String> splitAsStream(CharSequence input)	Obtenir un Stream des éléments résultant du découpage de la chaîne avec l'expression régulière comme séparateur (depuis Java 8)
String toString()	Renvoyer une représentation textuelle de l'instance sous la forme de chaîne de caractères du motif. Il s'agit de l'expression régulière à partir de laquelle le motif a été compilé.

La méthode static quote() renvoie un motif littéral pour la chaîne de caractères fournie en paramètre. Elle produit une chaîne qui peut être utilisée pour créer un motif qui correspondrait à la chaîne comme s'il s'agissait d'un motif littéral. Les métacaractères ou les séquences d'échappement dans la séquence d'entrée n'auront pas de signification particulière.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    System.out.println(Pattern.quote("[a-z]"));
  }
}
```

Résultat :

```
\Q[a-z]\E
```

23.1.1.1. L'obtention d'une instance de type Pattern

La classe Pattern ne propose aucun constructeur public.

Pour obtenir une instance de type Pattern, il faut invoquer une des surcharges de la fabrique compile(). Le premier paramètre de ces surcharges est le motif de l'expression régulière.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    Pattern pattern = Pattern.compile("[0-9]*");
  }
}
```

La classe Pattern est immuable et thread-safe.

Une des surcharges de la méthode compile attend en second paramètre un entier qui permet de préciser une ou plusieurs options lors de l'application de l'expression régulière.

Exemple :

```
    Pattern pattern = Pattern.compile("[a-z]*",Pattern.CASE_INSENSITIVE);
```

23.1.1.2. L'obtention d'une instance de type Matcher

La méthode matcher() permet d'obtenir une instance de type Matcher qui va utiliser l'expression régulière compilée.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    Pattern pattern = Pattern.compile("[a-z]*",Pattern.CASE_INSENSITIVE);
    Matcher matcher = pattern.matcher("Java");
  }
}
```

23.1.1.3. La méthode matches()

La méthode static matches() est proposée pour faciliter l'utilisation une seule fois d'une expression régulière. Elle compile l'expression et l'applique sur la chaîne fournie en paramètre en une seule invocation pour vérifier la correspondance du motif sur la chaîne

La méthode matches() renvoie un booléen qui renvoie le résultat de la correspondance avec l'application de l'expression régulière sur la chaîne de caractères passées en paramètres.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    boolean resultat = Pattern.matches("[0-9]*", "1234");
    System.out.println(resultat);
  }
}
```

Résultat :

```
true
```

L'exemple ci-dessus est équivalent à

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    Pattern pattern = Pattern.compile("[0-9]*");
    Matcher matcher = pattern.matcher("1234");
    boolean resultat = matcher.matches();
    System.out.println(resultat);
  }
}
```

Résultat :

```
true
```

Le résultat de la correspondance dépend du motif et la chaîne de caractères fournis.

Exemple :

```
    boolean trouve = Pattern.matches("bc", "abcd");
```

Exemple :

```
  Pattern pattern = Pattern.compile("bc");
  Matcher matcher = pattern.matcher("abcd");
  boolean trouve = matcher.matches();
```

Résultat :

```
false
```

La méthode matches() recompile l'expression à chaque invocation : elle n'est donc pas performante en cas d'invocations répétées. Dans ce cas, il est préférable de compiler l'expression et de réutiliser l'instance de type Pattern obtenue.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    boolean trouve = Pattern.matches("Java", "Java");
    System.out.println(trouve);
    trouve = Pattern.matches("Java", "Java utilise les regex");
    System.out.println(trouve);
  }
}
```

Résultat :

```
true
false
```

Dans ce cas, il est préférable de compiler l'expression et de réutilisé l'instance de type Pattern obtenue.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("Java");
    Matcher matcher = pattern.matcher("Java");
    boolean trouve = matcher.matches();
    System.out.println(trouve);

    matcher = pattern.matcher("Java utilise les regex");
    trouve = matcher.matches();
    System.out.println(trouve);
  }
}
```

Résultat :

```
true
false
```

23.1.1.4. Le découpage d'une chaîne selon un motif

La méthode split() permet de découper une chaîne en plusieurs sous-chaînes grâce à un délimiteur défini par l'expression régulière. Elle retourne un tableau de String qui contient les sous-chaînes.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern p = Pattern.compile(":");
    String[] elements = p.split("element1:element2:element3");
    for (String element : elements) {
      System.out.println(element);
    }
  }
}
```

Résultat :

```
element1
element2
element3
```

La surcharge de la méthode split() qui attend en second paramètre un entier limit permet de définir le nombre maximum de sous-chaînes obtenues.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern p = Pattern.compile(":");
    String[] elements = p.split("element1:element2:element3", 2);
    for (String element : elements) {
      System.out.println(element);
    }
  }
}
```

Résultat :

```
element1
element2:element3
```

23.1.2. La classe Matcher

La classe Matcher est un moteur d'application d'une expression régulière sur une chaîne de caractères et permet de restituer les informations issues de la correspondance trouvée.

La classe Matcher implémente l'interface MatchResult.

Elle ne possède aucun constructeur public. Pour obtenir une instance, il faut invoquer la méthode matcher() sur une instance de type Pattern en lui passant en paramètre la chaîne de caractères à traiter.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    String texte = "1234";
    Pattern pattern = Pattern.compile("[0-9]*");
    Matcher matcher = pattern.matcher(texte);
    if (matcher.matches()) {
      System.out.println("Correspondance trouvée");
    } else {
      System.out.println("Aucune correspondance");
    }
  }
}
```

Résultat :

```
Correspondance trouvée
```

Une instance de type Matcher encapsule l'état de la correspondance en cours : elle n'est donc pas thread-safe. Par contre, il est possible d'obtenir plusieurs instances de type Matcher à partir d'une même instance de type Pattern.

23.1.2.1. Les méthodes pour obtenir des index

Plusieurs méthodes permettent d'obtenir des index sur la précédente correspondance.

Méthode	Rôle
int start()	Renvoyer l'index de début de la précédente correspondance
int start(int group)	Renvoyer l'index de début du groupe capturé dont le numéro est fourni en paramètre de la précédente correspondance
int end()	Renvoyer l'index de fin de la précédente correspondance
public int end(int group)	Renvoyer l'index de fin du groupe capturé dont le numéro est fourni en paramètre de la précédente correspondance

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    String texte = "Java ou Java ou Java";
    Pattern pattern = Pattern.compile("Java");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc+" : "+texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : Java
2 : Java
3 : Java
Nb occurrences = 3
```

23.1.2.2. Les méthodes pour la recherche de correspondance

Les méthodes pour la recherche de correspondance analysent la chaîne de caractères d'entrée et renvoient un booléen indiquant si le motif est trouvé ou non :

Méthode	Rôle
boolean lookingAt()	Renvoyer un booléen qui indique si le motif est trouvé dans une partie de la chaîne de caractères
public boolean find()	Renvoyer un booléen si le motif est encore trouvé dans le reste du parcours de la chaîne de caractères
public boolean find(int start)	Renvoyer un booléen qui indique si le motif est trouvé à partir de l'index de début fourni en paramètre
public boolean matches()	Renvoyer un booléen qui indique si le motif est trouvé dans l'intégralité de la chaîne de caractères
int start()	Renvoyer l'index de début de la sous-chaîne capturée
int end()	Renvoyer l'index de fin de la sous-chaîne capturée

Les méthodes matches() et lookingAt() tentent toutes deux de faire correspondre l'expression régulière sur la chaîne de caractères. La différence réside dans le fait que matches() requiert la correspondance sur la chaîne de caractères dans son intégralité, alors que lookingAt() le fait partiellement.

La méthode matches() retourne true si l'intégralité d'une chaîne vérifie exactement le motif.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("Java");
    Matcher matcher = pattern.matcher("Java");
    boolean trouve = matcher.matches();
    System.out.println(trouve);
    matcher = pattern.matcher("Java utilise les regex");
    trouve = matcher.matches();
    System.out.println(trouve);
  }
}
```

Résultat :

```
true
false
```

Le comportement de la méthode lookAt() est différent car la correspondance ne se fait pas sur l'intégralité de la chaîne de caractères.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("Java");
    Matcher matcher = pattern.matcher("Java");
    boolean trouve = matcher.lookingAt();
    System.out.println(trouve);
    matcher = pattern.matcher("Java utilise les regex");
    trouve = matcher.lookingAt();
    System.out.println(trouve);
  }
}
```

Résultat :

```
true
true
```

La méthode find() tente de trouver la correspondance suivante du motif.

La recherche de correspondance commence au début de la région du Matcher ou, si une invocation précédente de la méthode a réussi et que le comparateur n'a pas été réinitialisé depuis, au premier caractère qui n'a pas été trouvé par la correspondance précédente.

Elle renvoie true si une correspondance est trouvée : dans ce cas, des informations supplémentaires peuvent être obtenues grâce aux méthodes start(), end() et group().

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    Pattern pattern = Pattern.compile("Java");
    Matcher matcher = pattern.matcher("Java Java Java");

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
Nb occurrences = 3
```

Une surcharge de le méthode find() qui attend en paramètre un entier de type int permet de préciser l'index de début de la recherche de correspondance.

Par convention, la méthode start() renvoie l'index de début et la méthode end() renvoie l'index de fin plus un qui correspond à l'index suivant.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("Java");
    Matcher matcher = pattern.matcher("Java");

    while (matcher.find()) {
      System.out.printf("Correspondance \"%s\" debut a l'" +
          "index %d, fin a l'index %d.%n",
          matcher.group(),
          matcher.start(),
          matcher.end());
    }
  }
}
```

Résultat :

```
Correspondance "Java" debut a l'index 0, fin a l'index 4
```

Dans l'exemple ci-dessus, bien que la chaîne de caractères contienne quatre caractères, l'index de début est à 0 et l'index de fin est à 4.

Lors des correspondances suivantes, il y a un chevauchement ; l'index de début de la correspondance suivante est le même que l'index de fin de la correspondance précédente.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("Java");
    Matcher matcher = pattern.matcher("JavaJavaJava");

    while (matcher.find()) {
      System.out.printf("Correspondance \"%s\" debut a l'" +
          "index %d, fin a l'index %d%n",
          matcher.group(),
          matcher.start(),
          matcher.end());
    }
  }
}
```

Résultat :

```
Correspondance "Java" debut a l'index 0, fin a l'index 4
Correspondance "Java" debut a l'index 4, fin a l'index 8
Correspondance "Java" debut a l'index 8, fin a l'index 12
```

23.1.2.3. Les méthodes concernant les groupes

Plusieurs méthodes permettent d'obtenir des informations relatives aux groupes capturés durant la correspondance d'un motif sur une chaîne de caractères.

Méthode	Rôle
String group()	Renvoyer la sous-chaîne correspondant au groupe capturé précédent lors de la correspondance
String group(int group)	Renvoyer la sous-chaîne correspondant au groupe capturé dont le numéro est fourni en paramètre
String group(String name)	Renvoyer la sous-chaîne correspondant au groupe capturé dont le nom est fourni en paramètre lors de la correspondance (depuis java 1.7)
int groupCount()	Renvoyer le nombre de groupes capturés lors de la correspondance
int start(int group)	Renvoyer l'index de début de la sous-chaîne capturée par le groupe dont le numéro est fourni en paramètre
int end(int group)	Renvoyer l'index de fin de la sous-chaîne capturée par le groupe dont le numéro est fourni en paramètre

23.1.2.4. Les méthodes de remplacement

Plusieurs méthodes permettent d'effectuer des remplacements.

Méthode	Rôle
Matcher appendReplacement(StringBuffer sb, String replacement)	Exécuter une étape intermédiaire d'ajout et remplacement
StringBuffer appendTail(StringBuffer sb)	Exécuter l'étape finale d'ajout et remplacement
String replaceAll(String replacement)	Remplacer toutes les sous-chaines de la séquence d'entrée qui correspond au motif par la chaîne de remplacement fournie
String replaceFirst(String replacement)	Remplacer la première sous-chaine de la séquence d'entrée qui correspond au motif par la chaîne de remplacement fournie
static String quoteReplacement(String s)	Renvoyer une chaîne littérale utilisable comme remplacement pour la chaîne spécifiée. Cette méthode produit une chaîne qui fonctionnera comme un remplacement littéral de la chaîne fournie en paramètre. La chaîne produite correspondra à la séquence de caractères fournie traitée comme une séquence littérale. Les caractères backslash « \ » et dollar « $ » n'ont pas de signification particulière.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    String resultat = "";
    Pattern pattern = Pattern.compile("euros");

    Matcher matcher = pattern.matcher("Le prix est de 100 euros TTC");

    try {
      resultat = matcher.replaceAll("$");
    } catch (Exception e) {
      System.out.println("Exception : " + e.getClass().getCanonicalName()
        + " "  + e.getMessage());
    }
    resultat = matcher.replaceAll(matcher.quoteReplacement("$"));
    System.out.println("Chaine de remplacement echappée : "+matcher.quoteReplacement("$"));
    System.out.println(resultat);
  }
}
```

Résultat :

```
Exception : java.lang.IllegalArgumentException Illegal group reference: group index is missing
Chaine de remplacement echappée : \$
Le prix est de 100 $ TTC
```

23.1.3. La classe PatternSyntaxException

La classe PatternSyntaxException est une exception unchecked qui indique qu'une expression régulière contient une erreur de syntaxe.

Elle possède plusieurs méthodes permettant d'obtenir des informations sur l'erreur de syntaxe :

Méthode	Rôle
String getDescription()	Renvoyer la description de l'erreur
int getIndex()	Renvoyer l'index de l'erreur
String getPattern()	Renvoyer l'expression régulière qui contient l'erreur de syntaxe
String getMessage()	Renvoyer un message multilignes qui décrit l'erreur de syntaxe

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Pattern;
import java.util.regex.PatternSyntaxException;

public class RegEx {

  public static void main(String[] args) {
    try {
    Pattern pattern = Pattern.compile("\\x");
    } catch (PatternSyntaxException pse) {
      pse.printStackTrace();
      System.out.println("\nDescription :");
      System.out.println(pse.getDescription());
      System.out.println("\nIndex :");
      System.out.println(pse.getIndex());
      System.out.println("\nPattern :");
      System.out.println(pse.getPattern());
      System.out.println("\nMessage :");
      System.out.println(pse.getMessage());
    }
  }
}
```

Résultat :

```
java.util.regex.PatternSyntaxException: Illegal hexadecimal escape sequence near index 2
\x
        at java.base/java.util.regex.Pattern.error(Pattern.java:2038)
        at java.base/java.util.regex.Pattern.x(Pattern.java:3378)
        at java.base/java.util.regex.Pattern.escape(Pattern.java:2608)
        at java.base/java.util.regex.Pattern.atom(Pattern.java:2296)
        at java.base/java.util.regex.Pattern.sequence(Pattern.java:2169)
        at java.base/java.util.regex.Pattern.expr(Pattern.java:2079)
        at java.base/java.util.regex.Pattern.compile(Pattern.java:1793)
        at java.base/java.util.regex.Pattern.<init>(Pattern.java:1440)
        at java.base/java.util.regex.Pattern.compile(Pattern.java:1079)
        at fr.jmdoudoux.dej.regex.RegEx.main(RegEx.java:10)

Description :
Illegal hexadecimal escape sequence

Index :
2

Pattern :
\x

Message :
Illegal hexadecimal escape sequence near index 2
\x
```

23.2. La mise en oeuvre des expressions régulières

Le premier paramètre de la méthode compile() de la classe Pattern est le motif qui décrit l'expression régulière.

La plus grande complexité lors de la mise en oeuvre des expressions régulières est de définir le motif qui correspond aux besoins.

La syntaxe des motifs des expressions régulières est proche de celle utilisée en Perl mais il y a quelques différences.

Les expressions régulières peuvent être testées et même analysées grâce à un débogueur en ligne sur le site https://regex101.com/. Il permet d'utiliser plusieurs syntaxes dont celle de l'API de Java.

La syntaxe permet d'utiliser différents concepts selon les besoins :

- une simple chaîne littérale
- des métacaractères (Meta Characters)
- des classes de caractères (Character Classes)
- des quantificateurs (Quantifiers)
- des groupes de capture (Capturing Groups)
- des limites de correspondance (Boundary Matchers)

Les expressions régulières peuvent contenir des métacaractères qui ont des rôles particuliers dans l'expression du motif.

Syntaxe	Rôle
`^`	Correspond à un début de la chaîne
`$`	Correspond à une fin de la chaîne
`.`	Correspond à n'importe quel caractère sauf par défaut une terminaison de ligne
`[abc]`	Correspond à un des caractères entre la paire de crochets

[^abc]	Correspond à n'importe quel caractère sauf ceux entre la paire de crochets
\A	Correspond au début de la chaîne dans son intégralité
\z	Correspond à la fin de la chaîne dans son intégralité
\Z	Correspond à la fin de la chaîne dans son intégralité à l'exception de la terminaison de ligne finale. Si une terminaison de ligne est présente, la correspondance se fait sur le caractère précédent.
re*	Correspond à une répétition 0 à n fois de l'excodession précédente
re+	Correspond à une répétition 1 à n fois de l'excodession précédente
re?	Correspond à une répétition 0 à 1 fois de l'expression précédente
re{ n}	Correspond à une répétition exactement n fois de l'expression précédente
re{ n,}	Correspond à une répétition au moins n fois de l'expression précédente
re{ n, m}	Correspond à une répétition d'au moins n fois et au plus m fois de l'expression précédente
a|b	Correspond à l'une ou l'autre des expressions à gauche et à droite. Ici les caractères a ou b
(re)	Définir un groupe de capture grâce à l'expression et permettre d'obtenir le texte correspondant
\w	Correspond à des caractères dans des mots
\W	Correspond à des caractères qui ne sont pas dans des mots
\s	Correspond à des caractères d'espacement. Equivalent à [\t\n\r\f]
\S	Correspond à des caractères qui ne sont pas des caractères d'espacement
\d	Correspond à des chiffres. Equivalent à [0-9]
\D	Correspond à n'importe quoi sauf des chiffres
\G	Correspond à l'endroit où la dernière correspondance s'est terminée
\b	Correspond aux caractères qui sont un début ou une fin de mot
\B	Correspond aux caractères qui ne sont pas un début ou une fin de mot
\n	Correspond à une nouvelle ligne
\t	Correspond à une tabulation
\Q	Débuter la considération des caractères comme des littéraux suivants jusqu'à \E
\E	Finir la considération des caractères comme des littéraux initiée par \Q

La regex est appliquée sur la chaîne de caractère de gauche à droite à la recherche de correspondance. Une fois qu'un caractère de la chaîne a été utilisé dans une correspondance, il ne peut pas être réutilisé pour une autre.

Exemple :

```java
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "xyxyxyxyx";
    Pattern pattern = Pattern.compile("xyx");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(),
        matcher.end())+", debut="+matcher.start());
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : xyx, debut=0
2 : xyx, debut=4
Nb occurrences = 2
```

23.2.1. L'échappement de caractères

Les caractères backslash dans les chaînes de caractères littérales dans le code source Java sont interprétées comme des échappements de caractères. Il est donc nécessaire de doubler les backslash dans les chaînes de caractères littérales qui contiennent des expressions régulières afin de les protéger de l'interprétation par le compilateur Java.

Les expressions régulières en Java utilise dans leur syntaxe un certain nombre de caractères considérés comme spéciaux car ils ont une signification particulière :

```
\.[]{}()*+?^$|
```

Le caractère backslash « \ » est le caractère d'échappement dans les chaînes de caractères en Java. Il doit lui-même être échappé par un caractère backslash qui le précède.

Les métacaractères qui débutent par un backslash doivent donc être exprimés dans une chaîne de caractères en Java par deux backslashs.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    System.out.println(Pattern.matches("\\d", "0"));
  }
}
```

Résultat :

```
true
```

Le caractère backslash est aussi le caractère d'échappement dans une expression régulière. La chaîne littérale "\b", par exemple, correspond à un simple caractère de retour arrière lorsqu'il est interprété comme une expression régulière, tandis que "\\b" correspond à une limite de correspondance de mot. La chaîne de caractères littérale "\(java\)" est illégale et provoque une erreur de compilation. Pour indiquer la chaîne "(java)", il faut utiliser la chaîne de caractères littérale "\\(java\\)".

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    System.out.println(Pattern.matches("\\{\\}", "{}"));
  }
}
```

Résultat :

```
true
```

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    System.out.println(Pattern.matches("\\(10\\)", "(10)"));
  }
}
```

Résultat :

```
true
```

Pour exprimer un backslash littéral dans une expression régulière, il faut donc utiliser quatre backslashs qui se suivent.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    System.out.println(Pattern.matches("\\\\", "\\"));
  }
}
```

Résultat :

```
true
```

Une exception de type PatternSyntaxException est levée lors de l'utilisation d'un backslash avant tout caractère alphabétique qui ne désigne pas une construction échappée. Ces échappements sont réservés pour de futures extensions de la syntaxe des expressions régulières.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    System.out.println(Pattern.matches("\\j", "10"));
  }
}
```

Résultat :

```
Exception in thread "main" java.util.regex.PatternSyntaxException: Illegal/unsupported
escape sequence near index 1
\j
 ^
        at java.base/java.util.regex.Pattern.error(Pattern.java:2038)
        at java.base/java.util.regex.Pattern.escape(Pattern.java:2618)
        at java.base/java.util.regex.Pattern.atom(Pattern.java:2296)
        at java.base/java.util.regex.Pattern.sequence(Pattern.java:2169)
        at java.base/java.util.regex.Pattern.expr(Pattern.java:2079)
        at java.base/java.util.regex.Pattern.compile(Pattern.java:1793)
        at java.base/java.util.regex.Pattern.<init>(Pattern.java:1440)
        at java.base/java.util.regex.Pattern.compile(Pattern.java:1079)
        at java.base/java.util.regex.Pattern.matches(Pattern.java:1184)
        at fr.jmdoudoux.dej.regex.RegEx.main(RegEx.java:9)
```

23.2.2. Les terminaisons de ligne (Line terminators)

Une terminaison de ligne est une séquence d'un ou deux caractères qui désigne la fin d'une ligne dans un ensemble de caractères. Plusieurs terminaisons de ligne sont reconnues par défaut dans les expressions régulières Java :

- le caractère newline (line feed) ('\n'),
- le caractère retour chariot ('\r'),
- le caractère retour chariot suivi du caractère new line ("\r\n"),
- le caractère next-line ('\u0085'),
- le caractère line-separator ('\u2028'),
- le caractère paragraph-separator ('\u2029)

La séquence \R représente toute séquence de saut de ligne Unicode : elle est équivalente à \u000D\u000A|[\u000A\u000B\u000C\u000D\u0085\u2028\u2029]

Par défaut, l'expression « . » correspond à un caractère quelconque excepté une terminaison de ligne. Pour modifier ce comportement, il est possible d'utiliser le mode DOTALL.

Par défaut, les expressions ^ and $ ignore les terminaisons de ligne et correspondent uniquement au début et à la fin de l'ensemble de la chaîne de caractères.

Si le mode UNIX_LINES est activé, la seule terminaison de ligne reconnue est le caractère newline.

Si le mode MULTILINE est activé, ^ correspond au début de la chaîne d'entrée et après tout terminaison de ligne, sauf à la fin de la chaîne d'entrée. En mode MULTILINE, $ est utilisé juste avant une fin de ligne ou la fin de la chaîne d'entrée.

23.2.3. Une simple chaîne littérale

L'expression régulière la plus simple consiste simplement en une simple chaîne de caractères littérale.

Les caractères peuvent être exprimés dans le motif de l'expression régulière de différentes manières :

Séquence	Rôle
`x`	Le caractère x
`\\`	Le caractère backslash
`\0n`	Le caractère correspond correspondant à la valeur octale 0n (0 <= n <= 7)
`\0nn`	Le caractère correspond correspondant à la valeur octale 0nn (0 <= n <= 7)
`\0mnn`	Le caractère correspond correspondant à la valeur octale 0mnn (0 <= m <= 3, 0 <= n <= 7)
`\xhh`	Le caractère correspond correspondant à la valeur hexadécimale 0xhh
`\uhhhh`	Le caractère correspond correspondant à la valeur hexadécimale 0xhhhh

`\x{h...h}`	Le caractère correspond correspondant à la valeur hexadécimale 0xh...h (Character.MIN_CODE_POINT <= 0xh...h <= Character.MAX_CODE_POINT)
`\t`	Le caractère tabulation ('\u0009')
`\n`	Le caractère newline (line feed) ('\u000A')
`\r`	Le caractère retour chariot (carriage-return) ('\u000D')
`\f`	Le caractère form-feed ('\u000C')
`\a`	Le caractère alert (bell) ('\u0007')
`\e`	Le caractère escape ('\u001B')

Lorsqu'une expression régulière est une simple chaîne de caractères, elle peut correspondre à zéro ou plusieurs fois selon le contenu de la chaîne de caractère sur laquelle elle est appliquée.

Exemple :

```java
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("Java");
    Matcher matcher = pattern.matcher("Java");
    System.out.println(matcher.find());
  }
}
```

Résultat :

```
true
```

Il est possible d'itérer sur l'invocation de la méthode find() pour trouver plusieurs occurrences.

Exemple :

```java
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("Java");
    Matcher matcher = pattern.matcher("JavaJava JavaJava");
    int nbOcc = 0;
    while (matcher.find()) {
        nbOcc++;
    }
    System.out.println("Nb occurences = " + nbOcc);
  }
}
```

Résultat :

```
Nb occurences = 4
```

23.2.4. Les métacaractères (Meta Characters)

Pour exprimer des besoins plus complexes dans le motif, il est possible d'utiliser des métacaractères. Ce sont des caractères qui ont une signification particulière et peuvent être combinés.

Les métacaractères modifient la manière dont un motif est utilisé pour la correspondance.

Plusieurs métacaractères sont utilisables dans les motifs :

```
<([{\^-=$!|]})?*+.>
```

Métacaractère	Rôle
`.`	Un seul caractère, n'importe lequel sauf une terminaison de ligne
`*`	0, 1 ou plusieurs caractères
`?`	0 ou une fois le motif qui précède
`()`	Groupe de capture
`[]`	Ensemble de caractères
`{}`	Quantificateur
`\`	Déspécialise le métacaractère qu'il précède
`^`	Début de ligne ou négation
`$`	Fin de ligne
`|`	Ou logique entre deux motifs
`+`	Une ou plusieurs fois le motif qui précède

Il y a deux manières de déspécialiser un métacaractère :

- faire précéder la métacaractère par un backslash
- déspécialiser un ensemble de caractères en les entourant avec \Q pour marquer le début et \E pour marquer la fin

23.2.5. Les classes de caractères (Character Classes)

Une classe de caractères définit un ensemble de caractères.

Les classes de caractères peuvent être incluses à l'intérieur d'autres classes de caractères.

Elles peuvent aussi être composées grâce à plusieurs opérateurs qui peuvent être combinés :

- l'opérateur d'union (implicite) désigne une classe qui contient tous les caractères figurant dans au moins une de ses classes d'opérandes
- l'opérateur d'intersection (&&) désigne une classe qui contient tous les caractères présents dans ses deux classes d'opérandes
- l'opérateur de négation (^) désigne une classe qui contient tous les caractères non présents dans la classe

23.2.5.1. Les ensembles de caractères

Un ensemble de caractères est une classe de caractères qui se définit en utilisant une paire de crochets.

Une syntaxe particulière peut être utilisée à l'intérieure de la paire de crochets pour définir l'ensemble de caractères :

Expression	Rôle
`[abc]`	Définir un ensemble de caractères : ceux fournis dans la paire de crochets
`[^abc]`	Définir un ensemble de caractères : ceux non fournis dans la paire de crochets
`[0-9]`	Définir un ensemble de caractères : ceux définis par la plage dont les bornes sont fournies à gauche et à droite du caractère moins
`[a-zA-Z]`	Définir un ensemble de caractères : l'union des deux plages

Attention : les métacaractères «^» et «$» ont un rôle différent entre une paire de crochets lors de la définition de classes de caractères.

Il est possible de définir un ensemble de caractères définis par ceux fournis.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "java";
    Pattern pattern = Pattern.compile("[ajv]");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : j
2 : a
3 : v
4 : a
Nb occurrences = 4
```

Il est possible de définir un ensemble de caractères définis par l'inverse de ceux fournis.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "java";
    Pattern pattern = Pattern.compile("[^perl]");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : j
2 : a
3 : v
4 : a
Nb occurrences = 4
```

Il est possible de définir une plage de caractères.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "Java";
    Pattern pattern = Pattern.compile("[A-Z]");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : J
Nb occurrences = 1
```

Une plage de caractères peut inclure des lettres majuscules, des chiffres et tous les caractères Unicode entre "0" et "Z".

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "Java : 1 <= 2";
    Pattern pattern = Pattern.compile("[0-Z]");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : J
2 : :
3 : 1
4 : <
5 : =
6 : 2
Nb occurrences = 6
```

23.2.5.2. L'opérateur OR

C'est l'opérateur par défaut utilisé entre les caractères précisés dans la paire de crochets.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "test";
    Pattern pattern = Pattern.compile("[est]");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : t
2 : e
3 : s
4 : t
Nb occurrences = 4
```

Un ensemble de caractères peut être combinés avec d'autres expressions pour former le motif.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "java Java";
    Pattern pattern = Pattern.compile("[jJ]ava");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : java
2 : Java
Nb occurrences = 2
```

23.2.5.3. L'opérateur NOR

Cet opérateur permet de définir l'ensemble inverse des caractères précisés entre la paire de crochet. Il utilise le caractère ^ qui doit précéder l'ensemble de caractères.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "Java";
    Pattern pattern = Pattern.compile("[^Jav]");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
Nb occurrences = 0
```

23.2.5.4. Une plage

Il est possible de définir une plage de caractères en séparant les deux bornes avec un caractère « - ».

Exemple avec des lettres minuscules

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "Java";
    Pattern pattern = Pattern.compile("[a-z]");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : a
2 : v
3 : a
Nb occurrences = 3
```

Exemple avec des lettres majuscules

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "JaVa";
    Pattern pattern = Pattern.compile("[A-Z]");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : J
2 : V
Nb occurrences = 2
```

Il est possible de combiner des plages. Par exemple, pour définir un ensemble de lettres minuscules et majuscules :

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "JaVa";
    Pattern pattern = Pattern.compile("[a-zA-Z]");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : J
2 : a
3 : V
4 : a
Nb occurrences = 4
```

Exemple : une plage de chiffres

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "Java 19";
    Pattern pattern = Pattern.compile("[0-9]");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : 1
2 : 9
Nb occurrences = 2
```

23.2.5.5. L'opérateur union

L'opérateur union permet de combiner 2 ou plusieurs classes de caractères.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "Java";
    Pattern pattern = Pattern.compile("[a-c[I-L]]");
    Matcher matcher = pattern.matcher(texte);
    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : J
2 : a
3 : a
Nb occurrences = 3
```

23.2.5.6. L'opérateur intersection

L'opérateur intersection permet de définir l'intersection 2 ou plusieurs classes de caractères : cela revient à prendre les caractères en commun dans les ensembles. Il utilise les caractères && qui doivent être utilisés entre deux ensembles de caractères.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "java";
    Pattern pattern = Pattern.compile("[a-z&&[aeiou]]");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : a
2 : a
Nb occurrences = 2
```

23.2.5.7. La soustraction de classe de caractères (Subtraction Class)

Il est possible d'utiliser une soustraction pour exclure une classe de caractères. Elle se fait en utilisant une combinaison des opérateurs intersection et NOR.

Par exemple, l'exemple ci-dessous fait une correspondance avec les nombres impairs :

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;
```

```
public class RegEx {

  public static void main(String[] args) {
    String texte = "123456789";
    Pattern pattern = Pattern.compile("[0-9&&[^02468]]");
    Matcher matcher = pattern.matcher(texte);
    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = "+nbOcc);
  }
}
```

Résultat :

```
1
3
5
7
9
Nb occurrences = 5
```

23.2.5.8. Les classes de caractères prédéfinies (Predefined Character Classes)

L'API pour les expressions régulières de Java propose un ensemble de classes de caractères prédéfinies. Ces classes de caractères prédéfinies sont des raccourcis pour des besoins courants.

Ces métacaractères ont la même première lettre que leur représentation en anglais, par exemple, d pour digit (chiffre), s pour space (caractère d'espacement), w pour word (mot), ... Les classes de caractères en majuscules définissent le contraire.

Plusieurs classes de caractères prédéfinies sont proposées par l'API :

Classe	Description
`\d`	Un chiffre. Equivalent à : [0-9]
`\D`	Tout sauf un chiffre. Equivalent à : [^0-9]
`\h`	Un caractère d'espacement horizontal. Equivalent à : `[ \t\xA0\u1680\u180e\u2000-\u200a\u202f\u205f\u3000]`
`\H`	Tout sauf un caractère d'espacement horizontal. Equivalent à : [^\h]
`\s`	Un caractère d'espacement. Equivalent à : [\t\n\x0B\f\r]
`\S`	Tout sauf un caractère d'espacement. Equivalent à : [^\s]
`\v`	Un caractère d'espacement vertical. Equivalent à : `[\n\x0B\f\r\x85\u2028\u2029]`
`\V`	Tout sauf un caractère d'espacement vertical. Equivalent à : [^\v]
`\w`	Un caractère alphanumérique. Equivalent à : [a-zA-Z_0-9]
`\W`	Tout sauf un caractère alphanumérique. Equivalent à : [^\w]
`.`	N'importe quel caractère sauf par défaut une terminaison de ligne

La plupart de ces classes de caractères débutent par un caractère « \ » qui doit être échappé avec un autre caractère « \ » comme dans toute chaîne de caractères en Java.

Le métacaractère « . » permet de désigner n'importe quel caractère excepté une terminaison de ligne.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern pattern = Pattern.compile(".a.a");
    Matcher matcher = pattern.matcher("Java");
    System.out.println(matcher.find());
  }
}
```

Résultat :

```
true
```

La recherche de la correspondance peut être effectuée plusieurs fois.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern pattern = Pattern.compile(".");
    Matcher matcher = pattern.matcher("Java");
    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
    }
    System.out.println("Nb occurrences = "+nbOcc);
  }
}
```

Résultat :

```
Nb occurrences = 4
```

La classe \d correspond à un chiffre.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "Java 19";
    Pattern pattern = Pattern.compile("\\d");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : 1
2 : 9
Nb occurrences = 2
```

La classe \D correspond à tout sauf un chiffre.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "Java 19";
    Pattern pattern = Pattern.compile("\\D");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : J
2 : a
3 : v
4 : a
5 :  
Nb occurrences = 5
```

La classe \s correspond à un caractère d'espacement.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "Java 19";
    Pattern pattern = Pattern.compile("\\s");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 :
Nb occurrences = 1
```

La classe \S correspond à tout sauf un caractère d'espacement.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "Java 19";
    Pattern pattern = Pattern.compile("\\S");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : J
2 : a
3 : v
4 : a
5 : 1
6 : 9
Nb occurrences = 6
```

La classe \w correspond un caractère alphanumérique.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "Java 19";
    Pattern pattern = Pattern.compile("\\w");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : J
2 : a
3 : v
4 : a
5 : 1
6 : 9
Nb occurrences = 6
```

La classe \W correspond à tout sauf un caractère alphanumérique.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "Java 19 n'est pas LTS";
    Pattern pattern = Pattern.compile("\\W");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 :  
2 :  
3 : '
4 :  
5 :  
Nb occurrences = 5
```

23.2.5.9. Les classes de caractères POSIX

Des classes de caractères POSIX (uniquement US-ASCII) sont proposées :

Classes	Correspondance
\p{ASCII}	Tous les caractères ASCII : [\x00-\x7F]
\p{Alnum}	Un caractère alphanumérique : [\p{Alpha}\p{Digit}]
\p{Alpha}	Un caractère alphabétique : [\p{Lower}\p{Upper}]
\p{Blank}	Un espace ou une tabulation : [\t]
\p{Cntrl}	Un caractère de contrôle : [\x00-\x1F\x7F]
\p{Digit}	Un chiffre : [0-9]
\p{Graph}	Un caractère visible : [\p{Alnum}\p{Punct}]
\p{Lower}	Un caractère alphabétique minuscule : [a-z]
\p{Print}	Un caractère affichable : [\p{Graph}\x20]

\p{Punct}	Un caractère de ponctuation : !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
\p{Space}	Un caractère d'espacement : [\t\n\x0B\f\r]
\p{Upper}	Un caractère alphabétique minuscule : [A-Z]
\p{XDigit}	Un chiffre hexadécimal : [0-9a-fA-F]

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    String texte = "Remise de 10% !";
    Pattern pattern = Pattern.compile("\\p{Punct}");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : %
2 : !
Nb occurrences = 2
```

23.2.5.10. Les classes de java.lang.Character

Plusieurs classes correspondantes à l'invocation de méthodes de la classe java.lang.Character sont proposées :

Classe	Correspondance
\p{javaLowerCase}	Equivalent à java.lang.Character.isLowerCase()
\p{javaUpperCase}	Equivalent à java.lang.Character.isUpperCase()
\p{javaWhitespace}	Equivalent à java.lang.Character.isWhitespace()
\p{javaMirrored}	Equivalent à java.lang.Character.isMirrored()

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    String texte = "(123)[abc]";
    Pattern pattern = Pattern.compile("\\p{javaMirrored}");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : (
2 : )
3 : [
4 : ]
Nb occurrences = 4
```

23.2.5.11. Les classes pour Unicode

Des classes sont proposées pour les scripts, les blocs, les catégories et les propriétés binaires Unicode.

Les scripts, blocs, catégories et propriétés binaires Unicode sont utilisés avec \p et \P. \p{prop} correspond à la propriété prop, tandis que \P{prop} ne correspond pas à la propriété.

Les scripts peuvent être précisés de plusieurs manières :

- avec le préfixe « Is ». Exemple : IsGreek
- en utilisant le mot clé « script ». Exemple : script=Greek
- ou sa forme courte « sc ». Exemple : sc=Greek

Les noms de scripts utilisables sont ceux définis et valides pour la méthode forName() de l'énumération UnicodeScript.

Les blocs peuvent être précisés de plusieurs manières :

- avec le préfixe « Is ». Exemple : InCyrillic
- en utilisant le mot clé « block ». Exemple : block=Cyrillic
- ou sa forme courte « blk ». Exemple : blk=Cyrillic

Les noms de scripts utilisables sont ceux définis et valides pour la méthode forName() de l'énumération UnicodeBlock.

Les catégories peuvent être précisées de plusieurs manières :

- sans préfixe. Exemple : \p{Lu}
- avec le préfixe « Is ». Exemple : \p{IsLu}
- en utilisant le mot clé « general_category ». Exemple : general_category=Lu
- ou sa forme courte « gc ». Exemple : gc=Lu

Les catégories utilisables sont celles définies dans le standard Unicode dont la version est supportée par la classe Character.

Un caractère Unicode peut également être représenté dans une expression régulière en utilisant sa notation hexadécimale (valeur du code point hexadécimal) en utilisant la séquence \x{...}.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    String texte = "aBcD";
    Pattern pattern = Pattern.compile("\\p{Lu}");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);

  }
}
```

Résultat :

```
1 : B
2 : D
Nb occurrences = 2
```

Les propriétés binaires (Binary properties) sont spécifiées avec le préfixe « Is », comme par exemple IsAlphabetic. Les propriétés binaires prises en charge par la classe Pattern sont :

- Alphabetic
- Ideographic
- Letter
- Lowercase
- Uppercase
- Titlecase
- Punctuation
- Control
- White_Space
- Digit
- Hex_Digit
- Join_Control
- Noncharacter_Code_Point
- Assigned

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    String texte = "aBcD";
    Pattern pattern = Pattern.compile("\\p{IsUpperCase}");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);

  }
}
```

Résultat :

```
1 : B
2 : D
Nb occurrences = 2
```

23.2.6. Les quantificateurs (Quantifiers)

Un quantificateur permet de préciser le nombre d'occurrences d'un motif d'une expression régulière lors de la correspondance.

Les quantificateurs sont utilisés dans une expression régulière en utilisant les métacaractères ?, *, + et {}.

Motif	Rôle
`*`	Répétition zéro ou plusieurs fois, raccourci pour {0,} Exemple : A* : zéro ou plusieurs lettres A .* : zéro ou plusieurs lettres quelconque
`+`	Répétition une ou plusieurs fois, raccourci pour {1,} Exemple : A+ : une ou plusieurs lettres A
`?`	Répétition une ou plusieurs fois, raccourci pour {0,1} Exemple : A ? : zéro ou une fois la lettre A
`{X}`	Répétition exactement X fois Exemple : \d{3} : trois chiffres .{5} : 5 caractères quelconques
`{X,Y}`	Répétition entre X et Y fois Exemple : \d{1,3} : entre un et trois chiffres

Les quantificateurs peuvent fonctionner selon trois modes avec des comportements différents :

- les avides (greedy) : les quantificateurs avides sont considérés comme "avides" parce qu'ils obligent le Matcher à lire la totalité de la chaîne d'entrée avant de tenter la première correspondance. Si la première tentative de correspondance (la chaîne d'entrée entière) échoue, le Matcher recule dans la chaîne d'entrée d'un caractère et essaie à nouveau, répétant le processus jusqu'à ce qu'une correspondance soit trouvée ou qu'il n'y ait plus de caractères à reculer. En fonction du quantificateur utilisé dans l'expression, la dernière chose qu'il essaiera de comparer sera 1 ou 0 caractère
- les réticents (reluctant) : les quantificateurs réticents mettent en ouvre une approche inverse : ils commencent au début de la chaîne d'entrée, puis consomme un caractère après l'autre à la recherche d'une correspondance. La dernière chose qu'ils essaient est la chaîne d'entrée entière
- les possessifs (possessive) : les quantificateurs possessifs consomment toujours la totalité de la chaîne d'entrée, en essayant une et une seule fois de trouver une correspondance. Contrairement aux quantificateurs avides, les quantificateurs possessifs ne reculent jamais, même si cela permettrait à la correspondance globale de réussir

Quantificateurs			Rôle
Avide (Greedy)	Réticent (Reluctant)	Possessif (Possessive)	

X?	X??	X?+	Zéro ou une fois
X*	X*?	X*+	Zéro ou plusieurs fois
X+	X+?	X++	Une ou plusieurs fois
X{n}	X{n}?	X{n}+	Exactement n fois
X{n,}	X{n,}?	X{n,}+	Au moins n fois
X{n, m}	X{n, m}?	X{n, m}+	Au moins n fois et jusqu'à m fois

Pour faire correspondre un motif zéro ou une fois, il faut utiliser le quantificateur « ? »

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("\\w?");
    rechercherCorrespondance(pattern, "");
    rechercherCorrespondance(pattern, "a");
    rechercherCorrespondance(pattern, "aa");
  }

  private static void rechercherCorrespondance(Pattern pattern, String texte) {
    Matcher matcher = pattern.matcher(texte);
    if (matcher.matches()) {
      System.out.println(texte + " : correspondance trouvee");
    } else {
      System.out.println(texte + " : aucune correspondance");
    }
  }
}
```

Résultat :

```
: correspondance trouvee
a : correspondance trouvee
aa : aucune correspondance
```

Alternativement, il est possible d'utiliser un quantificateur avec une paire d'accolades.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("\\w{0,1}");
    rechercherCorrespondance(pattern, "");
    rechercherCorrespondance(pattern, "a");
    rechercherCorrespondance(pattern, "aa");
  }

  private static void rechercherCorrespondance(Pattern pattern, String texte) {
    Matcher matcher = pattern.matcher(texte);
    if (matcher.matches()) {
      System.out.println(texte + " : correspondance trouvee");
    } else {
      System.out.println(texte + " : aucune correspondance");
    }
  }
}
```

Résultat :

```
: correspondance trouvee
a : correspondance trouvee
aa : aucune correspondance
```

Pour faire correspondre un motif zéro ou plusieurs fois, il faut utiliser le quantificateur « * »

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("\\w*");
    rechercherCorrespondance(pattern, "");
    rechercherCorrespondance(pattern, "a");
    rechercherCorrespondance(pattern, "ab");
    rechercherCorrespondance(pattern, "abc");
    rechercherCorrespondance(pattern, " ");
  }

  private static void rechercherCorrespondance(Pattern pattern, String texte) {
    Matcher matcher = pattern.matcher(texte);
    if (matcher.matches()) {
      System.out.println(texte + " : correspondance trouvee");
    } else {
      System.out.println(texte + " : aucune correspondance");
    }
  }
}
```

Résultat :

```
: correspondance trouvee
a : correspondance trouvee
ab : correspondance trouvee
abc : correspondance trouvee
  : aucune correspondance
```

Alternativement, il est possible d'utiliser un quantificateur avec une paire d'accolades.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("\\w{0,}");
    rechercherCorrespondance(pattern, "");
    rechercherCorrespondance(pattern, "a");
    rechercherCorrespondance(pattern, "ab");
    rechercherCorrespondance(pattern, "abc");
    rechercherCorrespondance(pattern, " ");
  }
  private static void rechercherCorrespondance(Pattern pattern, String texte) {
    Matcher matcher = pattern.matcher(texte);
    if (matcher.matches()) {
      System.out.println(texte + " : correspondance trouvee");
    } else {
      System.out.println(texte + " : aucune correspondance");
    }
  }
}
```

Résultat :

```
: correspondance trouvee
a : correspondance trouvee
ab : correspondance trouvee
abc : correspondance trouvee
  : aucune correspondance
```

Remarque : ce quantificateur peut trouver une correspondance avec un motif sur une chaîne de caractères vide en entrée puisque le motif peut être trouvé zéro ou plusieurs fois.

Pour faire correspondre un motif une ou plusieurs fois, il faut utiliser le quantificateur « + »

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("\\w+");
    rechercherCorrespondance(pattern, "");
    rechercherCorrespondance(pattern, "a");
    rechercherCorrespondance(pattern, "ab");
    rechercherCorrespondance(pattern, "abc");
    rechercherCorrespondance(pattern, " ");
  }

  private static void rechercherCorrespondance(Pattern pattern, String texte) {
    Matcher matcher = pattern.matcher(texte);
    if (matcher.matches()) {
      System.out.println(texte + " : correspondance trouvee");
    } else {
      System.out.println(texte + " : aucune correspondance");
    }
  }
}
```

Résultat :

```
: aucune correspondance
a : correspondance trouvee
ab : correspondance trouvee
abc : correspondance trouvee
  : aucune correspondance
```

Alternativement, il est possible d'utiliser un quantificateur avec une paire d'accolades.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("\\w{1,}");
    rechercherCorrespondance(pattern, "");
    rechercherCorrespondance(pattern, "a");
    rechercherCorrespondance(pattern, "ab");
    rechercherCorrespondance(pattern, "abc");
    rechercherCorrespondance(pattern, " ");
  }

  private static void rechercherCorrespondance(Pattern pattern, String texte) {
    Matcher matcher = pattern.matcher(texte);
    if (matcher.matches()) {
      System.out.println(texte + " : correspondance trouvee");
    } else {
      System.out.println(texte + " : aucune correspondance");
    }
  }
}
```

Résultat :

```
: aucune correspondance
a : correspondance trouvee
ab : correspondance trouvee
abc : correspondance trouvee
  : aucune correspondance
```

Pour faire correspondre un motif un nombre exact de fois, il faut utiliser un quantificateur avec une paire d'accolades.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("\\w{3}");
    rechercherCorrespondance(pattern, "ab");
    rechercherCorrespondance(pattern, "abc");
    rechercherCorrespondance(pattern, "abcd");
  }

  private static void rechercherCorrespondance(Pattern pattern, String texte) {
    Matcher matcher = pattern.matcher(texte);
    if (matcher.matches()) {
      System.out.println(texte + " : correspondance trouvee");
    } else {
      System.out.println(texte + " : aucune correspondance");
    }
  }
}
```

Résultat :

```
ab : aucune correspondance
abc : correspondance trouvee
abcd : aucune correspondance
```

Pour faire correspondre un motif un nombre de fois précisé dans une plage de valeur, il faut utiliser un quantificateur avec une paire d'accolades.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("\\w{2,3}");
    rechercherCorrespondance(pattern, "a");
    rechercherCorrespondance(pattern, "ab");
    rechercherCorrespondance(pattern, "abc");
    rechercherCorrespondance(pattern, "abcd");
  }

  private static void rechercherCorrespondance(Pattern pattern, String texte) {
    Matcher matcher = pattern.matcher(texte);
    if (matcher.matches()) {
      System.out.println(texte + " : correspondance trouvee");
    } else {
      System.out.println(texte + " : aucune correspondance");
    }
  }
}
```

Résultat :

```
a : aucune correspondance
ab : correspondance trouvee
abc : correspondance trouvee
abcd : aucune correspondance
```

L'utilisation d'un des trois mode (avide, réticent, possessif) a des conséquences sur la recherche de la correspondance. Par exemple avec motif ".*Java" qui désigne zéro ou plusieurs caractères suivi de « Java ».

Le premier exemple utilise un quantificateur avide.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "xJavaxxxxxxJava";
    Pattern pattern = Pattern.compile(".*Java");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : xJavaxxxxxxJava
Nb occurrences = 1
```

Comme le quantificateur est avide, le motif .* consomme l'intégralité de la chaîne de caractères en entrée. À ce moment, la correspondance échoue car les quatre derniers caractères « Java » sont consommés. Le comparateur du Matcher recule caractère par caractère jusqu'à ce qu'il trouve « Java », ce qui fait trouver la correspondance et arrêter la recherche. Ainsi une seule correspondance est trouvée.

Le second exemple utilise un quantificateur réticent.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "xJavaxxxxxxJava";
    Pattern pattern = Pattern.compile(".*?Java");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : xJava
2 : xxxxxxJava
Nb occurrences = 2
```

Le quantificateur réticent commence par vérifier si la chaîne commence par « Java ». Comme ce n'est pas le cas, il consomme les caractères pour trouver une correspondance et répète l'opération jusqu'à la fin de la chaîne de caractères. Ainsi deux correspondances sont trouvées.

Le troisième exemple utilise un quantificateur possessif.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "xJavaxxxxxxJava";
    Pattern pattern = Pattern.compile(".*+Java");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
Nb occurrences = 0
```

Avec le quantificateur possessif, toute la chaîne correspond à .*+ et il n'y a donc pas de correspondance avec « Java ». Aucune correspondance n'est donc trouvée.

Un quantificateur possessif est intéressant pour trouver une correspondance sans jamais reculer, ce qui le rend dans ce cas plus performant qu'un quantificateur avide.

23.2.7. Les groupes de capture (capturing groups)

Il est possible de regrouper des parties d'une expression régulière. Dans le motif, un groupe est défini entourant les éléments qui composent le groupe d'une paire de parenthèses.

Les groupes de captures permettent de traiter plusieurs caractères comme une seule unité sous la forme d'un groupe. Par exemple, (java) crée un groupe contenant les caractères "j", "a", "v" et "a".

Les groupes de capture sont ainsi nommés parce que, pendant une correspondance, chaque sous-séquence de la séquence d'entrée qui correspond à un tel groupe est conservée en mémoire. La sous-séquence capturée peut être utilisée ultérieurement dans l'expression, par le biais d'une référence arrière (back reference), et peut également être récupérée dans le Matcher une fois l'opération de correspondance terminée.

L'entrée capturée associée à un groupe est toujours la sous-séquence que le groupe a fait correspondre le plus récemment. Si un groupe est évalué une seconde fois en raison de la quantification, sa valeur précédemment capturée, le cas échéant, sera conservée si la seconde évaluation échoue. Par exemple, si la chaîne de caractères "xyx" est comparée à l'expression (x(y) ?)+, la valeur "y" sera attribuée au groupe deux. Toutes les entrées capturées sont rejetées au début de chaque correspondance.

23.2.7.1. La syntaxe de définition de groupes

Un groupe de capture est défini en utilisant une paire de parenthèses.

Exemple :

```
    Pattern pattern = Pattern.compile("(Java)");
```

Un groupe peut être imbriqué dans un autre groupe.

Exemple :

```
    Pattern pattern = Pattern.compile("(J((av)(a)))");
```

Il est possible d'assigner une quantification à un groupe.

Exemple :

```
    Pattern pattern = Pattern.compile("(Java){2}");
```

23.2.7.2. L'utilisation des méthodes de la classe Matcher pour les groupes

Une fois l'application de l'expression régulière, la classe Matcher propose plusieurs méthodes qui permettent d'obtenir des informations sur les groupes capturés :

Méthode	Rôle

int groupCount()	Renvoyer le nombre de groupes capturés
String group()	Renvoyer la valeur du groupe capturé par la précédente correspondance
String group(int)	Renvoyer la valeur du groupe capturé dont le numéro est fourni en paramètre
int start(int)	Renvoyer l'index du début du groupe dont le numéro est fourni en paramètre
int end(int)	Renvoyer l'index de la fin du groupe dont le numéro est fourni en paramètre

La surcharge de la méthode group() qui attend en paramètre entier renvoie la sous-chaîne capturée lors la précédente recherche de correspondance pour le groupe dont le numéro est fourni en paramètre.

Pour un Matcher m, une chaîne d'entrée c, et un indice de groupe g, les expressions m.group(g) et c.substring(m.start(g), m.end(g)) sont équivalentes.

Si la correspondance réussie mais que le groupe spécifié ne correspond à aucune partie de la chaîne d'entrée, null est renvoyé.

La surcharge de la méthode group() qui attend en paramètre une chaîne de caractères renvoie la sous-chaîne capturée lors la précédente recherche de correspondance pour le groupe dont le nom est fourni en paramètre.

Si la correspondance est réussie mais que le groupe spécifié ne correspond à aucune partie de la séquence d'entrée, null est renvoyé.

Elle peut lever deux exceptions :

- IllegalStateException si aucune recherche de correspondance n'a été faite ou si la précédente recherche a échouée
- IllegalArgumentException si le nom de groupe fourni en paramètre n'est pas défini dans le motif de l'expression régulière

Certains groupes, par exemple (a*), peuvent correspondre à une chaîne vide. Ces deux surcharges renvoient une chaîne vide lorsqu'un tel groupe correspond à la chaîne vide de l'entrée.

La méthode groupCount() de la classe Matcher renvoie le nombre total de groupe contenu dans une expression.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    String MOTIF_IPV4 = "^([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\\."
        + "([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\\."
        + "([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\\."
        + "([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])$";
    Pattern pattern = Pattern.compile(MOTIF_IPV4);
    Matcher matcher = pattern.matcher("127.0.0.1");
    if (matcher.matches()) {
      for (int i = 1; i <= matcher.groupCount(); i++) {
        System.out.println("Groupe " + i + " :" + matcher.group(i));
      }
    } else {
      System.out.println("Correspondance non trouvée");
    }
  }
}
```

Résultat :

```
Groupe 1 :127
Groupe 2 :0
Groupe 3 :0
Groupe 4 :1
```

23.2.7.3. La numérotation des groupes

Un nombre est associé à chaque groupe trouvé : il est possible d'utiliser ce nombre pour désigner le groupe correspondant.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    String texte = "Java";
    Pattern pattern = Pattern.compile("(Java)");
    Matcher matcher = pattern.matcher(texte);

    System.out.println("Nb groupes = " + matcher.groupCount());
    if (matcher.matches()) {
      for (int i = 0; i <= matcher.groupCount(); ++i) {
        System.out.println("groupe " + i + " :" + matcher.group(i));
      }
    }
  }
}
```

Résultat :

```
Nb groupes = 1
groupe 0 : Java
groupe 1 : Java
```

Le groupe 0 correspond toujours à l'intégralité de l'expression régulière.

Les groupes de capture sont numérotés en comptant leurs parenthèses ouvrantes de gauche à droite.

Par exemple, dans l'expression ((X)(Y(Z)) il y a quatre groupes :

1. ((X)(Y(Z)))
2. (X)
3. (Y(Z))
4. (Z)

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    String texte = "Java";
    Pattern pattern = Pattern.compile("(J((av)(a)))");
    Matcher matcher = pattern.matcher(texte);

    System.out.println("Nb groupes = " + matcher.groupCount());
    if (matcher.matches()) {
      for (int i = 0; i <= matcher.groupCount(); ++i) {
        System.out.println("groupe " + i + " : " + matcher.group(i));
      }
    } else {
      System.out.println("Aucune correspondance");
    }
  }
}
```

Résultat :

```
Nb groupes = 4
groupe 0 : Java
groupe 1 : Java
groupe 2 : ava
groupe 3 : av
groupe 4 : a
```

L'exemple ci-dessus définit 4 groupes. Ils sont numérotés de gauche à droite selon l'ordre de leur parenthèse ouvrante.

23.2.7.4. Le nommage d'un groupe

Depuis Java 7, il est possible d'affecter un nom à un groupe pour permettre d'y faire référence à travers de ce nom.

Le nom d'un groupe est composé d'un ensemble de caractères qui peuvent être :

- des lettres minuscules de « a » à « z »
- des lettres majuscules de « a » à « Z »
- des chiffres de « 0 » à « 9 »

La première lettre doit obligatoirement être une lettre.

Pour nommer un groupe, il faut utiliser la syntaxe :

```
(?<nom>sous-motif)
```

Un groupe qui possède un nom possède aussi un numéro attribué automatiquement.

Exemple (code Java 7) :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String chaine = "Le prix est de 315 euros";
    Pattern pattern = Pattern.compile("(?<libelle>\\D*)(?<montant>\\d*)(?<monnaie>\\D*)");

    Matcher matcher = pattern.matcher(chaine);
    if (matcher.matches()) {
      for (int i = 0; i <= matcher.groupCount(); i++) {
        System.out.println("Groupe " + i + " : " + matcher.group(i));
      }
    } else {
      System.out.println("Correspondance non trouvée");
    }
  }
}
```

Résultat :

```
Groupe 0 : Le prix est de 315 euros
Groupe 1 : Le prix est de
Groupe 2 : 315
Groupe 3 :  euros
```

La surcharge de la méthode group() de la classe Matcher qui attend en paramètre une chaîne de caractères permet d'obtenir la valeur capturée du groupe dont le nom est fourni en paramètre.

Exemple (code Java 7) :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String chaine = "Le prix est de 315 euros";
    Pattern pattern = Pattern.compile("(?<libelle>\\D*)(?<montant>\\d*)(?<monnaie>\\D*)");

    Matcher matcher = pattern.matcher(chaine);
    if (matcher.matches()) {
      for (int i = 0; i <= matcher.groupCount(); i++) {
        System.out.println("Groupe " + i + " : " + matcher.group(i));
      }
    } else {
      System.out.println("Correspondance non trouvée");
    }
  }
}
```

Résultat :

```
Groupe 0 : Le prix est de 315 euros
Groupe 1 : Le prix est de
Groupe 2 : 315
Groupe 3 :  euros
```

Une exception de type IllegalArgumentException est levée si le nom du groupe fourni en paramètre n'existe pas dans l'expression régulière.

Exemple (code Java 7) :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String chaine = "Le prix est de 315 euros";
    Pattern pattern = Pattern.compile("(?<libelle>\\D*)(?<montant>\\d*)(?<monnaie>\\D*)");

    Matcher matcher = pattern.matcher(chaine);
    if (matcher.matches()) {
      System.out.println("Groupe libelle : " + matcher.group("libelle"));
      System.out.println("Groupe montant : " + matcher.group("montant"));
      System.out.println("Groupe monnaie : " + matcher.group("devise"));

    } else {
      System.out.println("Correspondance non trouvée");
    }
  }
}
```

Résultat :

```
Groupe libelle : Le prix est de
Groupe montant : 315
Exception in thread "main" java.lang.IllegalArgumentException: No group with name <devise>
        at java.base/java.util.regex.Matcher.getMatchedGroupIndex(Matcher.java:1802)
        at java.base/java.util.regex.Matcher.group(Matcher.java:680)
        at fr.jmdoudoux.dej.regex.RegEx.main(RegEx.java:17)
```

Il n'est pas possible s'associer le même nom à deux groupes dans une même expression régulière. Dans ce cas, une exception de type PatternSyntaxException est levée à la compilation de l'expression.

Exemple (code Java 7) :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String chaine = "Le prix est de 315 euros";
    Pattern pattern = Pattern.compile("(?<libelle>\\D*)(?<montant>\\d*)(?<libelle>\\D*)");

    Matcher matcher = pattern.matcher(chaine);
    if (matcher.matches()) {
      System.out.println("Groupe libelle : " + matcher.group("libelle"));
      System.out.println("Groupe montant : " + matcher.group("montant"));
      System.out.println("Groupe monnaie : " + matcher.group("monnaie"));

    } else {
      System.out.println("Correspondance non trouvée");
    }
  }
}
```

Résultat :

```
Exception in thread "main" java.util.regex.PatternSyntaxException: Named capturing group
<libelle> is already defined near index 40
(?<libelle>\D*)(?<montant>\d*)(?<libelle>\D*)
                                        ^
        at java.base/java.util.regex.Pattern.error(Pattern.java:2038)
        at java.base/java.util.regex.Pattern.group0(Pattern.java:3002)
        at java.base/java.util.regex.Pattern.sequence(Pattern.java:2134)
        at java.base/java.util.regex.Pattern.expr(Pattern.java:2079)
        at java.base/java.util.regex.Pattern.compile(Pattern.java:1793)
        at java.base/java.util.regex.Pattern.<init>(Pattern.java:1440)
        at java.base/java.util.regex.Pattern.compile(Pattern.java:1079)
        at fr.jmdoudoux.dej.regex.RegEx.main(RegEx.java:11)
```

23.2.7.5. Les références arrières (back reference)

Il est possible de faire référence à un groupe dans le motif en utilisant son numéro ou éventuellement son nom si le groupe en possède un.

Il est possible de faire référence au contenu dans un groupe en utilisant une référence arrière en utilisant le caractère \ suivi du numéro du groupe concerné.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    String texte = "JavaJava";
    Pattern pattern = Pattern.compile("(Java)\\1");
    Matcher matcher = pattern.matcher(texte);

    System.out.println("Nb groupes = " + matcher.groupCount());
    if (matcher.matches()) {
      for (int i = 0; i <= matcher.groupCount(); ++i) {
        System.out.println("groupe " + i + " : " + matcher.group(i));
      }
    } else {
      System.out.println("Aucune correspondance");
    }
  }
}
```

Résultat :

```
Nb groupes = 1
groupe 0 : JavaJava
groupe 1 : Java
```

La correspondance se fait sur la même valeur que celle du groupe capturé précisé.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    String texte = "JavaFX";
    Pattern pattern = Pattern.compile("(Java)\\1");
    Matcher matcher = pattern.matcher(texte);

    System.out.println("Nb groupes = " + matcher.groupCount());
    if (matcher.matches()) {
      for (int i = 0; i <= matcher.groupCount(); ++i) {
        System.out.println("groupe " + i + " : " + matcher.group(i));
      }
    } else {
      System.out.println("Aucune correspondance");
    }
  }
}
```

Résultat :

```
Nb groupes = 1
Aucune correspondance
```

Il est possible de définir un autre groupe identique à un autre en utilisant une référence arrière (back reference) en utilisant son numéro comme référence.

Exemple :

```
package fr.jmdoudoux.dej.regex;

```

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    String texte = "JavaJava";
    Pattern pattern = Pattern.compile("(Java)(\\1)");
    Matcher matcher = pattern.matcher(texte);

    System.out.println("Nb groupes = " + matcher.groupCount());
    if (matcher.matches()) {
      for (int i = 0; i <= matcher.groupCount(); ++i) {
        System.out.println("groupe " + i + " : " + matcher.group(i));
      }
    }
  }
}
```

Résultat :

```
Nb groupes = 2
groupe 0 : JavaJava
groupe 1 : Java
groupe 2 : Java
```

Il est aussi possible d'utiliser une référence sur le nom du groupe en utilisant la syntaxe :

```
\k<nom>
```

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    String texte = "JavaJava";
    Pattern pattern = Pattern.compile("(?<cafe>Java)\\k<cafe>");
    Matcher matcher = pattern.matcher(texte);

    System.out.println("Nb groupes = " + matcher.groupCount());
    if (matcher.matches()) {
      for (int i = 0; i <= matcher.groupCount(); ++i) {
        System.out.println("groupe " + i + " : " + matcher.group(i));
      }
    } else {
      System.out.println("Aucune correspondance");
    }
  }
}
```

Résultat :

```
Nb groupes = 1
groupe 0 : JavaJava
groupe 1 : Java
```

23.2.7.6. Les groupes non capturants

Il est parfois utile de créer un groupe dont on n'a pas besoin de la capture de la correspondance.

La syntaxe d'un groupe non capturant est de la forme :

```
(?:motif)
```

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    traiter("Ja(va)?", "Java");
    traiter("Ja(va)?", "Ja");
    traiter("Ja(?:va)?", "Java");
    traiter("Ja(?:va)?", "Ja");
  }

  public static void traiter(String regex, String chaine) {
    System.out.printf("%nRegex : %s, chaine : %s%n", regex, chaine);
    Matcher matcher = Pattern.compile(regex)
        .matcher(chaine);
    while (matcher.find()) {
      System.out.printf("Groupe 0, debut=%s, fin=%s, correspondance='%s'%n",
        matcher.start(), matcher.end(), matcher.group());
      for (int i = 1; i <= matcher.groupCount(); i++) {
        System.out.printf("Groupe %s, debut=%s, fin=%s, correspondance='%s'%n",
          i, matcher.start(i), matcher.end(i), matcher.group(i));
      }
    }
  }
}
```

Résultat :

```
Regex : Ja(va)?, chaine : Java
Groupe 0, debut=0, fin=4, correspondance='Java'
Groupe 1, debut=2, fin=4, correspondance='va'

Regex : Ja(va)?, chaine : Ja
Groupe 0, debut=0, fin=2, correspondance='Ja'
Groupe 1, debut=-1, fin=-1, correspondance='null'

Regex : Ja(?:va)?, chaine : Java
Groupe 0, debut=0, fin=4, correspondance='Java'

Regex : Ja(?:va)?, chaine : Ja
Groupe 0, debut=0, fin=2, correspondance='Ja'
```

Dans l'exemple ci-dessus, on veut que la séquence « va » de « Java » soit facultative en utilisant le quantificateur « ? ». Il est nécessaire de regrouper « va » à l'aide de parenthèses pour définir un groupe qui sera facultatif. Si on utilise « va ? », seule la lettre « a » sera facultative, pas la séquence « va ». Dans ce cas, il n'est pas utile de capturer le groupe car il ne sera pas réutilisé dans le motif. C'est la raison pour laquelle un groupe non capturant est utilisé : (?:va)?

23.2.7.7. Les groupes atomiques

Les groupes atomiques ne font pas marche arrière (backtrack) une fois qu'une correspondance est réussie. Les groupes atomiques rejettent/oublient les parties suivantes du groupe une fois qu'une correspondance a été trouvée.

La syntaxe des groupes atomiques est de la forme :

```
<?>motif>
```

Les groupes atomiques ne sont pas capturant mais peuvent avoir des groupes de capture imbriqués.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    traiter("p(at|a)te", "patte");
    traiter("p(at|a)te", "pate");
    traiter("p(ot|at)e", "pate");

    traiter("p(?>at|a)te", "patte");
    traiter("p(?>at|a)te", "pate");
    traiter("p(?>ot|at)e", "pate");
  }

  public static void traiter(String regex, String chaine) {
    System.out.printf("%nRegex : %s, chaine : %s%n", regex, chaine);
    Matcher matcher = Pattern.compile(regex)
        .matcher(chaine);
    if (matcher.find()) {
      do {
        System.out.printf("Groupe 0, debut=%s, fin=%s, correspondance='%s'%n",
          matcher.start(), matcher.end(), matcher.group());
        for (int i = 1; i <= matcher.groupCount(); i++) {
          System.out.printf("Groupe %s, debut=%s, fin=%s, correspondance='%s'%n", i,
            matcher.start(i), matcher.end(i), matcher.group(i));
        }
      } while (matcher.find());
    } else {
      System.out.println("Aucune correspondance");
    }
  }
}
```

Résultat :

```
Regex : p(at|a)te, chaine : patte
Groupe 0, debut=0, fin=5, correspondance='patte'
Groupe 1, debut=1, fin=3, correspondance='at'

Regex : p(at|a)te, chaine : pate
Groupe 0, debut=0, fin=4, correspondance='pate'
Groupe 1, debut=1, fin=2, correspondance='a'

Regex : p(ot|at)e, chaine : pate
Groupe 0, debut=0, fin=4, correspondance='pate'
Groupe 1, debut=1, fin=3, correspondance='at'

Regex : p(?>at|a)te, chaine : patte
Groupe 0, debut=0, fin=5, correspondance='patte'

Regex : p(?>at|a)te, chaine : pate
Aucune correspondance

Regex : p(?>ot|at)e, chaine : pate
Groupe 0, debut=0, fin=4, correspondance='pate'
```

Lors de l'utilisation d'un groupe atomique s'il n'y a pas de correspondance c'est parce que le moteur ne revient pas en arrière pour les groupes atomiques. Le premier « p » de la chaîne d'entrée correspond à la partie littérale « p » de l'expression. Ensuite, « at » de la chaîne d'entrée correspond à la partie « at » dans le groupe. Le reste de la chaîne d'entrée « e » ne correspond pas au reste de l'expression régulière « te ». Le moteur ne revient pas en arrière parce que le groupe est atomique pour essayer une autre combinaison. Cette situation n'est valable que si le moteur a une correspondance avec la combinaison actuelle mais n'a pas de correspondance globale. En d'autres termes, le groupe atomique ne revient pas en arrière après la première correspondance de combinaison, même s'il existe une correspondance globale.

Si la première combinaison ne correspond pas parce que « ot » de la chaîne d'entrée ne satisfait pas la première combinaison du groupe, le moteur fait marche arrière pour vérifier la seconde combinaison, même si le groupe est atomique.

23.2.8. Les assertions lookaround

Parfois, il est difficile de faire correspondre une chaîne de caractères avec une expression régulière. Par exemple, il se peut que nous ne sachions pas exactement ce que nous voulons faire correspondre, mais nous savons qui vient directement avant ou ce qui manque après. Dans ces cas, il est possible d'utiliser les assertions lookaround. Ces expressions sont appelées assertions car elles indiquent seulement si quelque chose est une correspondance ou non, mais ne sont pas incluses dans le résultat.

Lookahead et lookbehind, collectivement appelés "lookaround", sont des assertions de longueur zéro, tout comme les métacaractères de début et de fin de ligne ou de début et de fin de mot. Les assertions ne consomment pas de caractères de la chaîne, mais affirment seulement si une correspondance est possible ou non.

Les fonctionnalités Lookaround permettent de créer des expressions régulières qu'il serait impossible de créer sans elles, ou qui deviendraient très fastidieuses sans elles.

Il existe quatre types d'assertions utilisables dans les expressions régulières.

Syntaxe	Nom	Rôle
(?=foo)	Lookahead	Affirme que ce qui suit immédiatement la position actuelle dans la chaîne est foo
(?!foo)	Negative Lookahead	Affirme que ce qui suit immédiatement la position actuelle dans la chaîne n'est pas foo
(?<=foo)	Lookbehind	Affirme que ce qui précède immédiatement la position actuelle dans la chaîne est foo
(?<!foo)	Negative Lookbehind	Affirme que ce qui précède immédiatement la position actuelle dans la chaîne n'est pas foo

Les assertions lookahead et lookbehind ne consomment aucun caractère de la chaîne d'entrée. Cela signifie qu'après la parenthèse de fermeture du lookahead ou du lookbehind, le moteur de regex reste à l'endroit même de la chaîne où il a commencé à chercher. À partir de cette position, le moteur peut recommencer à faire correspondre des caractères.

Le fait que le lookaround soit de longueur nulle le rend automatiquement atomique. Dès que la condition de la zone de recherche est satisfaite, le moteur de regex oublie tout ce qui se trouve à l'intérieur du lookaround.

Il est toutefois possible d'utiliser des groupes de capture à l'intérieur du lookaround.

Exemple :

```java
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    String texte = "5x5 55x55 155x55 551x55";
    Pattern pattern = Pattern.compile("(?=(\\d+))\\w+\\1");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : 5x5
2 : 55x55
3 : 55x55
Nb occurrences = 3
```

23.2.8.1. Les assertions Lookahead

Les assertions Lookaheads sont des assertions de longueur zéro, ce qui signifie qu'elles ne sont pas incluses dans la correspondance. Elles affirment seulement si la portion immédiate qui précède la portion courante d'une chaîne d'entrée donnée est appropriée pour une correspondance ou non.

La syntaxe générale d'une assertion lookahead est de la forme : elle commence par une parenthèse ouvrante suivi d'un point d'interrogation « (? » suivie d'un métacaractère « = » ou « ! » suivi d'une regex d'affirmation (toutes les expressions regex sont autorisées) suivie de parenthèses fermantes «) ».

Il existe deux types d'assertions lookahead :

- lookahead positive
- lookahead negative

Chacune d'entre elles a deux syntaxes :

- assertion après la correspondance
- assertion avant la correspondance

Les deux appliquent la condition d'assertion en amont de leur position.

Tous les motifs valides peuvent être utilisés dans un lookahead.

23.2.8.1.1. Positive Lookahead

Une assertion positive lookahead est utile pour faire correspondre quelque chose suivi d'autre chose.

La syntaxe d'un positive lookahead utilise une paire de parenthèses, contenant un point d'interrogation et un signe égal suivi du motif.

Exemple :

```
o(?=u)
```

La correspondance se fait sur «o» suivi de «u», avec «u» qui ne fait pas partie de la correspondance.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String chaine = "oui";
    Pattern pattern = Pattern.compile("o(?=u)");

    Matcher matcher = pattern.matcher(chaine);
    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + chaine.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : o
Nb occurrences = 1
```

Dans l'exemple ci-dessus, le moteur applique l'expression régulière «o(?=u)» à la chaîne «oui». Le premier élément de la regex est le littéral «o». Le moteur parcourt la chaîne jusqu'à ce que le caractère «o» soit trouvé dans la chaîne. L'élément suivant dans la regex est le lookahead. Le moteur prend note qu'il est à l'intérieur d'une construction lookahead maintenant, et commence à faire correspondre la regex à l'intérieur du lookahead. L'élément suivant de la regex est le u à l'intérieur du lookahead. Le caractère suivant est un «u». Il y a correspondance. Le moteur avance au caractère suivant : «i». Cependant, il le fait avec la regex à l'intérieur du lookahead. Le moteur note le succès, et rejette la correspondance de l'expression régulière. Cela amène le moteur à reculer dans la chaîne jusqu'à u. La correspondance se fait uniquement sur le caractère «o» qui est bien suivi d'un caractère «u».

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String chaine = "oui";
    Pattern pattern = Pattern.compile("o(?=u)i");

    Matcher matcher = pattern.matcher(chaine);
    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + chaine.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
Nb occurrences = 0
```

Dans l'exemple ci-dessus, le moteur applique l'expression régulière «o(?=u)i» à la chaîne «oui».

Le positive lookahead est suivi d'un autre élément dans la regex. Encore une fois, «o» correspond à «o» suivi d'un «u» qui correspond à «u». Encore une fois, la correspondance du lookahead est écartée, donc le moteur recule de «i» dans la chaîne à «u». Le lookahead a réussi, donc le moteur continue avec «i». Mais «i» ne correspond pas à «u». Donc cette tentative de correspondance échoue.

La regex o(?=u)i ne peut jamais correspondre à quoi que ce soit. Elle essaie de faire correspondre «u» et «i» à la même position. S'il y a un «u» immédiatement après le «o», le lookahead réussit mais «i» ne correspond pas à «u». S'il y a autre chose qu'un «u» immédiatement après le «o», le lookahead échoue.

Pour modifier ce comportement, il faut ajouter le métacaractère «.» après le lookahead pour que le moteur reconsomme le caractère «u».

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String chaine = "oui";
    Pattern pattern = Pattern.compile("o(?=u).i");

    Matcher matcher = pattern.matcher(chaine);
    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + chaine.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : oui
Nb occurrences = 1
```

Dans l'exemple ci-dessus, la correspondance réussie car «oui» est bien composé du caractère «o» suivi du caractère «u» et du caractère «i». La correspondance obtenue est «oui».

Si l'assertion contient des groupes de capture, ces groupes seront capturés normalement et des back references vers eux sont utilisables même en dehors du lookahead.

Une assertion lookahead elle-même n'est pas un groupe de capture. Elle n'est pas incluse dans le calcul de la numérotation des back references. Si l'on veut stocker la correspondance du motif à l'intérieur d'un lookahead, il faut mettre des parenthèses de capture autour du motif à l'intérieur du loohahead (?=(motif)).

Positive lookahead après la correspondance

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String chaines[] = { "10 pour 315 euros", "20 pour 263 dollars" };
    Pattern pattern = Pattern.compile("\\d+(?= euros)");

    for (int i = 0; i < chaines.length; i++) {
      Matcher matcher = pattern.matcher(chaines[i]);
      System.out.println(chaines[i]);
      int nbOcc = 0;
      while (matcher.find()) {
        nbOcc++;
        System.out.println(nbOcc + " : " + chaines[i].substring(matcher.start(),
          matcher.end()));
      }
      System.out.println("Nb occurrences = " + nbOcc);
    }
  }
}
```

Résultat :

```
10 pour 315 euros
1 : 315
Nb occurrences = 1
20 pour 263 dollars
Nb occurrences = 0
```

L'expression régulière "\\d+(?= euros)" correspond à des chiffres suivis immédiatement de la chaîne de caractères " euros ".

Positive lookahead avant la correspondance

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String chaines[] = { "10 pour 315 euros", "20 pour 263 dollars" };
    Pattern pattern = Pattern.compile("(?=\\d+ euros)\\d+");

    for (int i = 0; i < chaines.length; i++) {
      Matcher matcher = pattern.matcher(chaines[i]);
      System.out.println(chaines[i]);
      int nbOcc = 0;
      while (matcher.find()) {
        nbOcc++;
        System.out.println(nbOcc + " : " + chaines[i].substring(matcher.start(),
          matcher.end()));
      }
      System.out.println("Nb occurrences = " + nbOcc);
    }
  }
}
```

Résultat :

```
10 pour 315 euros
1 : 315
Nb occurrences = 1
20 pour 263 dollars
Nb occurrences = 0
```

Le positive lookahead (?=\d+ euros) affirme qu'à la position actuelle dans la chaîne, ce qui suit est des chiffres puis les caractères " euros". Si l'assertion réussit, le moteur fait correspondre les chiffres avec \d+.

Avec une assertion avant la correspondance, il est nécessaire de répéter le motif de la correspondance recherchée. Dans l'exemple ci-dessus, dans la regex «?=\d+ euros)\d+», le motif \d+ est répété deux fois, une fois à l'intérieur du lookahead et une fois à l'extérieur. C'est nécessaire quel que soit l'assertion fournie au moteur, il faut aussi lui ajouter la correspondance recherchée. Il ne faut répéter que la partie pour laquelle la correspondance est recherchée. Si ce n'est pas le cas alors il n'y aura simplement pas de correspondance trouvée car l'assertion échouera.

Pour simplifier la répétition, il est possible de définir un groupe capture dans l'assertion et d'utiliser une back reference.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {
  public static void main(String[] args) {

    String chaines[] = { "10 pour 315 euros", "20 pour 263 dollars" };
    Pattern pattern = Pattern.compile("(?=(\\d+) euros)\\1");

    for (int i = 0; i < chaines.length; i++) {
      Matcher matcher = pattern.matcher(chaines[i]);
      System.out.println(chaines[i]);
      int nbOcc = 0;
      while (matcher.find()) {
        nbOcc++;
        System.out.println(nbOcc + " : " + chaines[i].substring(matcher.start(),
          matcher.end()));
      }
      System.out.println("Nb occurrences = " + nbOcc);
    }
  }
}
```

Résultat :

```
10 pour 315 euros
1 : 315
Nb occurrences = 1
20 pour 263 dollars
Nb occurrences = 0
```

Le motif utilisé permet d'obtenir le même résultat que \d+(?= euros) utilisé précédemment, mais il est moins efficace car \d+ est évalué deux fois. Une meilleure utilisation de l'assertion avant la correspondance est de valider plusieurs conditions.

23.2.8.1.2. Negative Lookahead

Une assertion negative lookahead est à utiliser lorsque l'on veut faire correspondre quelque chose qui n'est pas suivi par autre chose. Ce comportement ne peut pas être mis en ouvre avec des classes de caractères. Un negative lookahead fournit la solution.

La syntaxe d'un negative lookahead utilise une paire de parenthèses, contenant un point d'interrogation et d'un point d'exclamation suivi du motif.

Exemple :

```
a(?!z)
```

Negative Lookahead après la correspondance

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String chaines[] = { "10 pour 315 euros", "20 pour 263 dollars" };
    Pattern pattern = Pattern.compile("\\d+(?!\\d+| euros)");

    for (int i = 0; i < chaines.length; i++) {
      Matcher matcher = pattern.matcher(chaines[i]);
      System.out.println(chaines[i]);
      int nbOcc = 0;
      while (matcher.find()) {
        nbOcc++;
        System.out.println(nbOcc + " : " + chaines[i].substring(matcher.start(),
          matcher.end()));
      }
      System.out.println("Nb occurrences = " + nbOcc);
    }
  }
}
```

Résultat :

```
10 pour 315 euros
1 : 10
Nb occurrences = 1
20 pour 263 dollars
1 : 20
2 : 263
Nb occurrences = 2
```

Le lookahead (?!\d+| euros) affirme qu'à la position actuelle dans la chaîne, ce qui suit immédiatement n'est ni un chiffre ni les caractères " euros". Si l'assertion réussit, le moteur fait correspondre les chiffres avec \d+.

Negative Lookahead avant la correspondance

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String chaines[] = { "10 pour 315 euros", "20 pour 263 dollars" };
    Pattern pattern = Pattern.compile("(?!\\d+ euros)\\d+");

    for (int i = 0; i < chaines.length; i++) {
      Matcher matcher = pattern.matcher(chaines[i]);
      System.out.println(chaines[i]);
      int nbOcc = 0;
      while (matcher.find()) {
        nbOcc++;
        System.out.println(nbOcc + " : " + chaines[i].substring(matcher.start(),
          matcher.end()));
      }
      System.out.println("Nb occurrences = " + nbOcc);
    }
  }
}
```

Résultat :

```
10 pour 315 euros
1 : 10
Nb occurrences = 1
20 pour 263 dollars
1 : 20
2 : 263
Nb occurrences = 2
```

Le lookahead négatif (?!\d+ euros) affirme qu'à la position actuelle dans la chaîne, ce qui suit n'est pas des chiffres mais les caractères " euros". Si l'assertion réussit, le moteur fait correspondre les chiffres avec \d+.

Le motif utilisé permet d'obtenir le même résultat que \d+(?!\d+| euros) utilisé précédemment, mais il est moins efficace car \d+ est évalué deux fois. Une meilleure utilisation de l'assertion avant la correspondance est de valider plusieurs conditions.

23.2.8.2. Les assertions lookbehind

Une assertion lookbehind a le même effet que lookahead, mais fonctionne à l'envers.

Il indique au moteur regex de reculer temporairement dans la chaîne de caractères, pour vérifier si le texte contenu dans le lookbehind peut être trouvé à cet endroit.

Lookbehind est une autre assertion de longueur nulle, tout comme les assertions Lookahead. Leur assertion est validée seulement si la portion immédiate derrière la portion courante d'une chaîne d'entrée donnée est appropriée pour une correspondance ou non.

Il y a deux types d'assertions lookbehind :

- positive lookbehind
- negative lookbehind

Chacun possède deux syntaxes :

- assertion avant la correspondance
- assertion après la correspondance

Les deux appliquent la condition d'assertion derrière leur position.

23.2.8.2.1. Positive lookBehind

Une assertion positive lookbehind est utile pour faire correspondre quelque chose précédé d'autre chose.

La syntaxe d'un positive lookbehind utilise une paire de parenthèses, contenant un point d'interrogation, un inférieur et un signe égal suivi du motif.

Exemple :

```
(?<=X)
```

Positive lookbehind avant la correspondance

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String chaines[] = { "Code : 1234", "PIN : 5678" };
    Pattern pattern = Pattern.compile("(?<=Code : )\\d{4}");

    for (int i = 0; i < chaines.length; i++) {
      Matcher matcher = pattern.matcher(chaines[i]);
      System.out.println(chaines[i]);
      int nbOcc = 0;
      while (matcher.find()) {
        nbOcc++;
        System.out.println(nbOcc + " : " + chaines[i].substring(matcher.start(),
          matcher.end()));
      }
      System.out.println("Nb occurrences = " + nbOcc);
    }
  }
}
```

Résultat :

```
Code : 1234
1 : 1234
Nb occurrences = 1
PIN : 5678
Nb occurrences = 0
```

Le lookbehind positif (?<=Code :) affirme qu'à la position actuelle dans la chaîne, ce qui précède est composé des caractères "Code : ". Si l'assertion réussit, le moteur fait correspondre les 4 chiffres avec \d{4}.

Positive lookbehind après la correspondance

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String chaines[] = { "Code : 1234", "PIN : 5678" };
    Pattern pattern = Pattern.compile("\\d{4}(?<=Code : \\d{4})");

    for (int i = 0; i < chaines.length; i++) {
      Matcher matcher = pattern.matcher(chaines[i]);
      System.out.println(chaines[i]);
      int nbOcc = 0;
      while (matcher.find()) {
        nbOcc++;
        System.out.println(nbOcc + " : " + chaines[i].substring(matcher.start(),
          matcher.end()));
      }
      System.out.println("Nb occurrences = " + nbOcc);
    }
  }
}
```

Résultat :

```
Code : 1234
1 : 1234
Nb occurrences = 1
PIN : 5678
Nb occurrences = 0
```

"\d{4}" correspond à 4 chiffres. Le lookbehind "(?<=Code : \d{4})" affirme qu'à cette position dans la chaîne, ce qui précède immédiatement est la séquence "Code : " puis 4 chiffres.

Le motif utilisé permet d'obtenir le même résultat que "(?<=Code :)\d{4}" utilisé précédemment, mais il est moins efficace car "\d{4}" est évalué deux fois.

23.2.8.2.2. Negative Lookbehind

Une assertion negative lookbehind est utile pour faire correspondre quelque chose qui ne soit pas précédé d'autre chose.

Le moteur de regex traite un negative lookbehind de la même manière qu'un positive lookbehind avant la correspondance sauf qu'il applique une négation sur l'assertion.

La syntaxe d'un negative lookbehind utilise une paire de parenthèses, contenant un point d'interrogation, un inférieur et un point d'exclamation suivi du motif.

Exemple :

```
(?<!X)
```

Negative lookbehind avant la correspondance

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String chaines[] = { "Code : 1234", "PIN : 5678" };
    Pattern pattern = Pattern.compile("(?<!Code : )\\d{4}");

    for (int i = 0; i < chaines.length; i++) {
      Matcher matcher = pattern.matcher(chaines[i]);
      System.out.println(chaines[i]);
      int nbOcc = 0;
      while (matcher.find()) {
        nbOcc++;
        System.out.println(nbOcc + " : " + chaines[i].substring(matcher.start(),
          matcher.end()));
      }
      System.out.println("Nb occurrences = " + nbOcc);
    }
  }
}
```

Résultat :

```
Code : 1234
Nb occurrences = 0
PIN : 5678
1 : 5678
Nb occurrences = 1
```

Le lookbehind négatif "(?<!Code :)" affirme qu'à la position actuelle dans la chaîne, ce qui précède n'est pas composé des caractères "Code : ". Si l'assertion réussit, le moteur fait correspondre quatre chiffres avec "\d{4}".

Negative lookbehind après la correspondance

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String chaines[] = { "Code : 1234", "PIN : 5678" };
    Pattern pattern = Pattern.compile("\\d{4}(?<!Code : \\d{4})");

    for (int i = 0; i < chaines.length; i++) {
      Matcher matcher = pattern.matcher(chaines[i]);
      System.out.println(chaines[i]);
      int nbOcc = 0;
      while (matcher.find()) {
        nbOcc++;
        System.out.println(nbOcc + " : " + chaines[i].substring(matcher.start(),
          matcher.end()));
      }
      System.out.println("Nb occurrences = " + nbOcc);
    }
  }
}
```

Résultat :

```
Code : 1234
Nb occurrences = 0
PIN : 5678
1 : 5678
Nb occurrences = 1
```

"\d{4}" correspond à 4 chiffes. Le lookbehind négatif "(?<!Code : \d{4})" affirme qu'à cette position dans la chaîne, ce qui précède immédiatement n'est pas les caractères " Code : " puis 4 chiffres.

Le motif utilisé permet d'obtenir le même résultat que "\d{4}(?<!Code :)" utilisé précédemment, mais il est moins efficace car "\d{4}" est évalué deux fois.

23.2.8.2.3. Les limitations des assertions Lookbehind en Java

La plupart des moteurs ne supportent pas toutes les expressions dans un Lookbehind. La raison est que le moteur doit être capable de déterminer le nombre de caractères à reculer avant de vérifier l'expression fournie dans le lookbehind .

Lors de l'évaluation du lookbehind, le moteur d'expression régulière détermine la longueur de l'expression régulière contenue dans le lookbehind, recule d'autant de caractères dans la chaîne du sujet, puis applique l'expression régulière contenue dans le lookbehind de gauche à droite, comme il le ferait avec une expression régulière normale.

Java autorise tout sauf les quantificateurs et les métacaractères '+' et '*' dans une assertion lookbehind qui présentent des limitations dans certains cas.

Dans des cas comme "[0-9]*", ces quantificateurs fonctionnent, mais ils ne fonctionnent pas dans des cas comme "x[0-9]*" (lorsque l'expression est bornée à gauche).

Exemple :

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    valider("a2 456b 456789c 45d ef", "(?<=[0-9]+)[a-z]");
  }

  private static void valider(String chaine, String regex) {
    Pattern pattern = Pattern.compile(regex);

    Matcher matcher = pattern.matcher(chaine);
    System.out.println(chaine);
    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + chaine.substring(matcher.start(),
        matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
C:\java>java RegEx
a2 456b 456789c 45d ef
1 : b
2 : c
3 : d
Nb occurrences = 3
```

Le moteur de regex de Java permet l'utilisation de répétition finie dans un lookbehind. Il est possible d'utiliser le métacaractère «?» et une quantification avec le paramètre max spécifié. Java détermine les longueurs minimales et maximales possibles du lookbehind.

Le moteur va tenter successivement tenter d'appliquer l'expression du lookbehind de gauche à droite en reculant de la longueur minimale vers la longueur maximale tant que la correspondance n'est pas trouvée. Cette recherche peut nuire aux performances notamment si la longueur maximale est grande.

Certains bugs présents en Java 4 et 5 dans l'exploitation d'un lookbehind sont corrigés en Java 6.

Une des limitations concernent une quantification supérieure sans limite sur un motif qui est lui-même précédé par au moins un autre motif.

Exemple :

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    valider("a2 456b 456789c 45d ef", "(?<=4[0-9]{3,})[a-z]");
  }

  // ...
}
```

Résultat :

```
C:\java>java -version
openjdk version "1.7.0_75"
OpenJDK Runtime Environment (build 1.7.0_75-b13)
OpenJDK Client VM (build 24.75-b04, mixed mode)

C:\java>javac RegEx.java

C:\java>java RegEx
Exception in thread "main" java.util.regex.PatternSyntaxException: Look-behind group
 does not have an obvious maximum length near index 13
(?<=4[0-9]{3,})[a-z]
             ^
        at java.util.regex.Pattern.error(Pattern.java:1924)
        at java.util.regex.Pattern.group0(Pattern.java:2812)
        at java.util.regex.Pattern.sequence(Pattern.java:2018)
        at java.util.regex.Pattern.expr(Pattern.java:1964)
        at java.util.regex.Pattern.compile(Pattern.java:1665)
        at java.util.regex.Pattern.<init>(Pattern.java:1337)
        at java.util.regex.Pattern.compile(Pattern.java:1022)
        at RegEx.valider(RegEx.java:12)
        at RegEx.main(RegEx.java:8)
```

Cela a été corrigé en Java 13.

Résultat :

```
C:\java>java -version
openjdk version "13" 2019-09-17
OpenJDK Runtime Environment (build 13+33)
OpenJDK 64-Bit Server VM (build 13+33, mixed mode, sharing)

C:\java>java RegEx
a2 456b 456789c 45d ef
1 : c
Nb occurrences = 1
```

Une autre limitation concerne l'utilisation des métacaractères « + » et « * » pour la répétition d'un motif lui-même précédé d'au moins un autre motif.

Exemple :

```
public class RegEx {

  public static void main(String[] args) {

    valider("a2 456b 456789c 45d ef", "(?<=4[0-9]+)[a-z]");
  }

  // ...
}
```

Résultat :

```
C:\java>java -version
openjdk version "1.7.0_75"
OpenJDK Runtime Environment (build 1.7.0_75-b13)
OpenJDK Client VM (build 24.75-b04, mixed mode)

C:\java>java RegEx
Exception in thread "main" java.util.regex.PatternSyntaxException: Look-behind group
 does not have an obvious maximum length near index 10
(?<=4[0-9]*)[a-z]
          ^
        at java.util.regex.Pattern.error(Pattern.java:1924)
        at java.util.regex.Pattern.group0(Pattern.java:2812)
        at java.util.regex.Pattern.sequence(Pattern.java:2018)
        at java.util.regex.Pattern.expr(Pattern.java:1964)
        at java.util.regex.Pattern.compile(Pattern.java:1665)
        at java.util.regex.Pattern.<init>(Pattern.java:1337)
        at java.util.regex.Pattern.compile(Pattern.java:1022)
        at RegEx.valider(RegEx.java:12)
        at RegEx.main(RegEx.java:8)
```

Une possibilité pour contourner cette limitation est d'utiliser un quantificateur avec une limite supérieure à définir pour ne pas être trop élevée afin de ne pas pénaliser les performances.

Exemple :

```
public class RegEx {

  public static void main(String[] args) {
    valider("a2 456b 456789c 45d ef", "(?<=4[0-9]{1,10})[a-z]");
  }

  // ...
}
```

Résultat :

```
C:\java>java -version
openjdk version "1.7.0_75"
OpenJDK Runtime Environment (build 1.7.0_75-b13)
OpenJDK Client VM (build 24.75-b04, mixed mode)

C:\java>java RegEx
a2 456b 456789c 45d ef
1 : b
2 : c
3 : d
Nb occurrences = 3
```

Cette limitation a été réglée en Java 9.

Exemple :

```
public class RegEx {

  public static void main(String[] args) {
    valider("a2 456b 456789c 45d ef", "(?<=4[0-9]*)[a-z]");
  }

  // ...
}
```

Résultat :

```
C:\java>java -version
java version "9.0.1"
Java(TM) SE Runtime Environment (build 9.0.1+11)
Java HotSpot(TM) 64-Bit Server VM (build 9.0.1+11, mixed mode)

C:\java>java RegEx
a2 456b 456789c 45d ef
1 : b
2 : c
3 : d
Nb occurrences = 3
```

Pour des raisons de performance, il est tout de même conseillé d'utiliser à la place un quantificateur avec une limite supérieure à définir pour ne pas être trop élevée afin de ne pas pénaliser les performances.

23.2.9. Les limites de correspondance (Boundary Matchers)

L'API RegEx de Java prend également en charge les limites de correspondance. Cela permet de définir où la correspondance doit être trouvée dans la chaîne en entrée car les limites de correspondance permettent de préciser où débute et où fini la recherche d'un motif.

Séquence	Rôle
^	Début de ligne
$	Fin de ligne
\b	Extrémité de mot
\B	Extrémité de non-mot
\A	Début de la séquence en entrée
\G	Fin de l'occurrence précédente
\Z	Fin de la séquence, sauf le caractère final
\z	Fin de la séquence en entrée

Le métacaractère ^ indique le début du texte.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "Java supporte les regex";
    Pattern pattern = Pattern.compile("^Java");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : Java
Nb occurrences = 1
```

Une correspondance est trouvée car la chaîne débute par «Java».

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "Le langage Java supporte les regex";
    Pattern pattern = Pattern.compile("^Java");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
Nb occurrences = 0
```

Aucune correspondance n'est trouvée car la chaîne ne débute pas par «Java».

Le métacaractère $ indique la fin du texte.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "Les regex sont supportées par Java";
    Pattern pattern = Pattern.compile("Java$");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : Java
Nb occurrences = 1
```

Une correspondance est trouvée car la chaîne se termine par «Java».

Le métacaractère \b indique une limite de mot. Un espace est considéré comme une limite de mot.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "Le langage Java supporte les regex";
    Pattern pattern = Pattern.compile("\\bJava\\b");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : Java
Nb occurrences = 1
```

Un début ou une fin de ligne sont considérés comme une limite de mot.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "Java supporte les regex";
    Pattern pattern = Pattern.compile("\\bJava\\b");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : Java
Nb occurrences = 1
```

Aucune correspondance n'est trouvée si le motif n'est pas suivi d'une limite de mot.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "JavaFX est un framework pour interface graphique";
    Pattern pattern = Pattern.compile("\\bJava\\b");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
Nb occurrences = 0
```

Une correspondance est trouvée en remplaçant la limite de fin par le métacaractère \B qui indique une limite qui ne soit pas un mot.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    String texte = "JavaFX est un framework pour interface graphique";
    Pattern pattern = Pattern.compile("\\bJava\\B");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : Java
Nb occurrences = 1
```

Le métacaractère \G indique la fin de l'occurrence précédente.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "JavaJava";
    Pattern pattern = Pattern.compile("\\GJava");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : Java
2 : Java
Nb occurrences = 2
```

Les occurrences doivent se suivrent pour obtenir une correspondance.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "Java Java";
    Pattern pattern = Pattern.compile("\\GJava");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);
  }
}
```

Résultat :

```
1 : Java
Nb occurrences = 1
```

Une seule correspondance est trouvée, la première, car la deuxième n'apparaît pas après la première, puisque séparée avec un caractère espace.

23.2.10. Les modes utilisables

Différents modes sont utilisables pour influencer la manière dont le motif est interprété.

Pour préciser un ou plusieurs modes, il est possible d'utiliser :

- une surcharge de la méthode compile() attend en second paramètre un entier qui permet de préciser des modes à utiliser lors la recherche de correspondance
- la plupart des modes propose une option embarquée qu'il faut utiliser obligatoirement en début de l'expression régulière

Ces modes sont définies sous la forme de constante de type int dans la classe Pattern :

Constante	Option embarquée	Rôle
CANON_EQ		Activer l'équivalence canonique (canonical equivalence). Deux caractères seront considérés comme correspondant si leurs décompositions canoniques complètes correspondent. Exemple : l'expression "a\u030A" correspondra à la chaîne "\u00E5". Ce mode ne possède pas d'option embarquée. L'utilisation de ce mode peut dégrader les performances.
CASE_INSENSITIVE	(?i)	Activer la correspondance sans tenir compte de la casse. Par défaut, cette correspondance insensible à la casse suppose que seuls les caractères du jeu de caractères US-ASCII sont mis en correspondance. La correspondance insensible à la casse et sensible à l'Unicode peut être activé en utilisant le mode UNICODE_CASE en conjonction avec ce mode. L'utilisation de ce mode peut dégrader les performances.
COMMENTS	(?x)	Ignorer les caractères d'espacement et les commentaires dans la regex. Les commentaires commencent par # et sont ignorés jusqu'à la fin d'une ligne
DOTALL	(?s)	Activer le mode dotall : la correspondance de "." inclut aussi les caractères de terminaison de lignes.

LITERAL		Le motif est traité comme une séquence de caractères littéraux : les métacaractères contenus dans le motif n'ont pas de signification particulière et sont traités comme des caractères ordinaires lors de la recherche de correspondance. Les modes CASE_INSENSITIVE et UNICODE_CASE conservent leur impact sur la correspondance lorsqu'ils sont utilisés conjointement avec ce mode. Les autres modes sont superflus. Ce mode ne possède pas d'option embarquée. (depuis Java 1.5)
MULTILINE	(?m)	Activer le mode multilignes. Dans ce mode, les métacaractères ^ et $ correspondent juste après ou juste avant respectivement à une fin d'une ligne ou à la fin de la séquence d'entrée.
UNICODE_CASE	(?u)	Activer la gestion des caractères Unicode. Le mode CASE_INSENSITIVE est utilisable conjointement avec ce mode pour ne pas tenir compte de la casse. L'utilisation de ce mode peut dégrader les performances.
UNICODE_CHARACTER_CLASS	(?U)	Activer la version Unicode des classes de caractères prédéfinies (US-ASCII) et des classes de caractères POSIX pour être conforme avec Unicode Technical Standard #18: Unicode Regular Expression (annexe C). Ce mode implique le mode UNICODE_CASE. L'utilisation de ce mode peut dégrader les performances. (depuis Java 1.7)
UNIX_LINES	(?d)	Activer le mode dans lequel seul la terminaison de ligne Unix \n est utilisé dans le comportement des métacaractère « . », « ^ » et « $ »

Il est possible de combiner plusieurs constantes pour activer plusieurs modes en utilisant l'opérateur | :

Exemple :

```
    Pattern pattern = Pattern.compile(exp, Pattern.DOTALL | Pattern.UNIX_LINES);
```

Il est aussi possible de combiner des options embarquées simplement en les concaténant.

Exemple :

```
    Pattern pattern = Pattern.compile("(?sd).*");
```

Pattern.CANON_EQ

Ce mode permet d'activer l'équivalence canonique (canonical equivalence). Lorsqu'elle est utilisée, deux caractères seront considérés comme correspondant si leurs décompositions canoniques correspondent.

Par exemple, le caractère accentué Unicode « é », dont le point de code composite est u00E9. Unicode possède également un point de code distinct pour les caractères composants « e » (u0065) et l'accent aigu (u0301). Dans ce cas, le caractère composite u00E9 est équivalent à la séquence des deux caractères u0065 u0301.

Par défaut, la correspondance ne tient pas compte de l'équivalence canonique.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "\u0065\u0301";
    Pattern pattern = Pattern.compile("\u00E9");
    Matcher matcher = pattern.matcher(texte);

    if (matcher.matches()) {
      System.out.println("Correspondance trouvée");
    } else {
      System.out.println("Aucune correspondance");
    }
  }
}
```

Résultat :

```
Aucune correspondance
```

Le mode CANON_EQ active la prise en compte de l'équivalence canonique lors de la recherche de correspondance.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String texte = "\u0065\u0301";
    Pattern pattern = Pattern.compile("\u00E9", Pattern.CANON_EQ);
    Matcher matcher = pattern.matcher(texte);

    if (matcher.matches()) {
      System.out.println("Correspondance trouvée");
    } else {
      System.out.println("Aucune correspondance");
    }
  }
}
```

Résultat :

```
Correspondance trouvée
```

Ce mode ne peut pas être exprimé sous la forme d'une option embarquée.

Pattern.CASE_INSENSITIVE

Par défaut, la correspondance est sensible à la casse.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("java");
    Matcher matcher = pattern.matcher("Java");

    if (matcher.matches()) {
      System.out.println("Correspondance trouvée");
    } else {
      System.out.println("Aucune correspondance");
    }
  }
}
```

Résultat :

```
Aucune correspondance
```

Le mode CASE_INSENSITIVE active la correspondance sans tenir compte de la casse.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("java", Pattern.CASE_INSENSITIVE);
    Matcher matcher = pattern.matcher("Java");

    if (matcher.matches()) {
      System.out.println("Correspondance trouvée");
    } else {
      System.out.println("Aucune correspondance");
    }
  }
}
```

Résultat :

```
Correspondance trouvée
```

Il est aussi possible d'utiliser l'option équivalente en début de l'expression régulière.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("(?i)java");
    Matcher matcher = pattern.matcher("Java");

    if (matcher.matches()) {
      System.out.println("Correspondance trouvée");
    } else {
      System.out.println("Aucune correspondance");
    }
  }
}
```

Résultat :

```
Correspondance trouvée
```

Pattern.COMMENTS

En Java, il est possible d'utiliser des commentaires dans une expression régulière pour la documenter.

Un commentaire débute par un caractère # dans l'expression régulière.

Par défaut, les commentaires sont interprétés comme faisant partie du motif.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("^Java.* # Commence par Java");
    Matcher matcher = pattern.matcher("Java supporte les regex");

    if (matcher.matches()) {
      System.out.println("Correspondance trouvée");
    } else {
      System.out.println("Aucune correspondance");
    }
  }
}
```

Résultat :

```
Aucune correspondance
```

Le mode COMMENTS ignore les caractères d'espacement et les commentaires à la fin dans la regex.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("^Java.* # Commence par Java", Pattern.COMMENTS);
    Matcher matcher = pattern.matcher("Java supporte les regex");

    if (matcher.matches()) {
      System.out.println("Correspondance trouvée");
    } else {
      System.out.println("Aucune correspondance");
    }
  }
}
```

Résultat :

```
Correspondance trouvée
```

Il est aussi possible d'utiliser l'option équivalente en début de l'expression régulière.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("(?x)^Java.* # Commence par Java");
    Matcher matcher = pattern.matcher("Java supporte les regex");

    if (matcher.matches()) {
      System.out.println("Correspondance trouvée");
    } else {
      System.out.println("Aucune correspondance");
    }
  }
}
```

Résultat :

```
Correspondance trouvée
```

Pattern.DOTALL

Par défaut, lors de l'utilisation du caractère point "." dans une regex, la correspondance se fait sur un caractère quelconque jusqu'à ce qu'un caractère de nouvelle ligne soit rencontré.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern pattern = Pattern.compile(".*");
    Matcher matcher = pattern.matcher("Ligne 1\nLigne2\nLigne3");

    if (matcher.matches()) {
      System.out.println("Correspondance trouvée");
    } else {
      System.out.println("Aucune correspondance");
    }
  }
}
```

Résultat :

```
Aucune correspondance
```

L'option DOTALL permet de demander que la correspondance de "." inclut aussi les caractères de nouvelle ligne.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern pattern = Pattern.compile(".*", Pattern.DOTALL);
    Matcher matcher = pattern.matcher("Ligne 1\nLigne2\nLigne3");

    if (matcher.matches()) {
      System.out.println("Correspondance trouvée");
    } else {
      System.out.println("Aucune correspondance");
    }
  }
}
```

Résultat :

```
Correspondance trouvée
```

Il est aussi possible d'utiliser l'option équivalente en début de l'expression régulière.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("(?s).*");
    Matcher matcher = pattern.matcher("Ligne 1\nLigne2\nLigne3");

    if (matcher.matches()) {
      System.out.println("Correspondance trouvée");
    } else {
      System.out.println("Aucune correspondance");
    }
  }
}
```

Résultat :

```
Correspondance trouvée
```

Pattern.LITERAL

Par défaut, certains métacaractères ont une signification particulière lors de la recherche de la correspondance.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("(.*)");
    Matcher matcher = pattern.matcher("Java");
    if (matcher.matches()) {
      System.out.println("Correspondance trouvée");
    } else {
      System.out.println("Aucune correspondance");
    }
  }
}
```

Résultat :

```
Correspondance trouvée
```

Le mode LITERAL désactive l'interprétation de tous les métacaractères.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("(.*)", Pattern.LITERAL);
    Matcher matcher = pattern.matcher("Java");

    if (matcher.matches()) {
      System.out.println("Correspondance trouvée");
    } else {
      System.out.println("Aucune correspondance");
    }
  }
}
```

Résultat :

```
Aucune correspondance
```

Pattern.MULTILINE

Par défaut, les métacaractères « ^ » et « $ » correspondent respectivement au début et à la fin de la chaîne de caractères d'entrée entière. La correspondance ne tient pas compte des terminaisons de ligne intermédiaires.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    String texte = "Java\nJava";
    Pattern pattern = Pattern.compile("Java$");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);

  }
}
```

Résultat :

```
1 : Java
Nb occurrences = 1
```

Le mode MULTILINE permet de modifier le comportement par défaut des métacaractères ^ et $ pour qu'ils prennent en compte chaque ligne.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    String texte = "Java\nJava";
    Pattern pattern = Pattern.compile("Java$", Pattern.MULTILINE);
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);

  }
}
```

Résultat :

```
1 : Java
2 : Java
Nb occurrences = 2
```

Il est aussi possible d'utiliser l'option équivalente en début de l'expression régulière.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    String texte = "Java\nJava";
    Pattern pattern = Pattern.compile("(?m)Java$");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);

  }
}
```

Résultat :

```
1 : Java
2 : Java
Nb occurrences = 2
```

23.2.11. Le support d'Unicode

Les séquences d'échappement Unicode telles que \u20AC sont supportées dans les motifs.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    String texte = "200?";
    Pattern pattern = Pattern.compile("\\u20AC");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);

  }
}
```

Résultat :

```
1 : ?
Nb occurrences = 1
```

Un caractère Unicode peut également être représenté dans une expression régulière en utilisant sa notation hexadécimale (valeur du code point hexadécimal) en utilisant la séquence \x{...}.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    String texte = "200?";
    Pattern pattern = Pattern.compile("\\x{20AC}");
    Matcher matcher = pattern.matcher(texte);

    int nbOcc = 0;
    while (matcher.find()) {
      nbOcc++;
      System.out.println(nbOcc + " : " + texte.substring(matcher.start(), matcher.end()));
    }
    System.out.println("Nb occurrences = " + nbOcc);

  }
}
```

Résultat :

```
1 : ?
Nb occurrences = 1
```

Lorsque l'indicateur UNICODE_CHARACTER_CLASS est utilisé alors les classes de caractères prédéfinies et les classes de caractères POSIX ci-dessous sont conformes à la recommandation de l'annexe C : Propriétés de compatibilité des expressions régulières Unicode.

Classes	Matches
`\d`	Un chiffre : `\p{IsDigit}`
`\D`	Tout sauf un chiffre : `[^\d]`
`\s`	Un caractère d'espacement : `\p{IsWhite_Space}`
`\S`	Tout sauf un caractère d'espacement : `[^\s]`
`\w`	Un caractère dans des mots : `[\p{Alpha}\p{gc=Mn}\p{gc=Me}\p{gc=Mc}\p{Digit}\p{gc=Pc}\p{IsJoin_Control}]`
`\W`	Un caractère qui n'est pas dans un mot : `[^\w]`

23.3. Les remplacements de texte

Les expressions régulières peuvent être utilisées pour effectuer des remplacements des occurrences trouvées dans la chaîne de caractères à traiter.

23.3.1. Les remplacements avec la classe Matcher

La classe Matcher propose plusieurs méthodes pour effectuer des remplacements des correspondances trouvées par une autre chaîne.

23.3.1.1. Les méthodes replaceFirst() et replaceAll()

Plusieurs méthodes permettent d'effectuer des remplacements : les méthodes replaceFirst() et replaceAll() permettent de faire un ou plusieurs remplacements du motif s'il est trouvé.

La méthode replaceFirst() remplace uniquement la première occurrence du motif de la chaîne de caractères par celle fournie en paramètre.

Exemple :

```java
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    String texteJava = "Java est orienté objet, Java supporte les regex";
    Pattern pattern = Pattern.compile("Java");
    Matcher matcher = pattern.matcher(texteJava);
    String textePerl = matcher.replaceFirst("Perl");
    System.out.println(textePerl);
  }
}
```

Résultat :

```
Perl est orienté objet, Java supporte les regex
```

La méthode replaceAll() remplace toutes les occurrences du motif de la chaîne de caractères par celle fournie en paramètre.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    String texteJava = "Java est orienté objet, Java supporte les regex";
    Pattern pattern = Pattern.compile("Java");
    Matcher matcher = pattern.matcher(texteJava);
    String textePerl = matcher.replaceAll("Perl");
    System.out.println(textePerl);
  }
}
```

Résultat :

```
Perl est orienté objet, Perl supporte les regex
```

L'expression régulière peut être plus complexe.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    String chaine = "Le langage <b>Java</b>";
    Pattern pattern = Pattern.compile("<b>([^<]+)</b>");
    Matcher matcher = pattern.matcher(chaine);
    System.out.println(matcher.replaceAll(""));
  }
}
```

Résultat :

```
Le langage
```

Il est possible de faire référence à la valeur d'un groupe comme valeur de remplacement en utilisant la syntaxe $n, où n correspond au numéro du groupe de capture, dans la chaîne de remplacement.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {
  public static void main(String[] args) {

    String chaine = "Le langage <b>Java</b>";
    Pattern pattern = Pattern.compile("<b>([^<]+)</b>");
    Matcher matcher = pattern.matcher(chaine);
    System.out.println(matcher.replaceAll("$1"));
  }
}
```

Résultat :

```
Le langage Java
```

23.3.1.2. Les méthodes appendReplacement() et appendTail()

La classe Matcher fournit également les méthodes appendReplacement() et appendTail() pour le remplacement du texte.

Elles s'utilisent conjointement pour le remplacement d'occurrences dans un objet de type StringBuffer.

La méthode appendReplacement() exécute une étape d'ajout et remplacement : elle remplace la précédente correspondance par la chaîne fournie en paramètre et ajoute ensuite le résultat au StringBuilder.

La méthode appendTail() doit être invoquée après les invocations de la méthode appendReplacement() pour copier le reste de la chaîne de caractères. Si cette méthode n'est pas invoquée après des remplacements, alors la chaîne de caractères résultantes ne sera pas complète.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {

    Pattern pattern = Pattern.compile("Perl");
    Matcher matcher = pattern.matcher("Perl est orienté objet, Perl supporte les regex");
    StringBuffer sb = new StringBuffer();
    while (matcher.find()) {
      matcher.appendReplacement(sb, "Java");
    }
    matcher.appendTail(sb);
    System.out.println(sb.toString());
  }
}
```

Résultat :

```
Java est orienté objet, Java supporte les regex
```

L'utilisation combinée de ces deux méthodes peut avoir le même effet que l'utilisation de la méthode replaceAll() mais elle peut aussi permettre d'avoir un contrôle très fin sur chacun des remplacement effectué.

Exemple :

```
package fr.jmdoudoux.dej.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegEx {

  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("Perl");
    Matcher matcher = pattern.matcher("Perl est orienté objet, Perl supporte les regex");
    StringBuffer sb = new StringBuffer();
    int nbOcc = 1;
    while (matcher.find()) {
      matcher.appendReplacement(sb, "Java("+nbOcc+")");
      nbOcc++;
    }
    matcher.appendTail(sb);
    System.out.println(sb.toString());
  }
}
```

Résultat :

```
Java(1) est orienté objet, Java(2) supporte les regex
```

23.4. L'utilisation d'expressions régulières dans les méthodes de la classe String

Plusieurs méthodes de la classe String attendent en paramètre une expression régulières :

Plusieurs méthodes proposent des équivalents à celles de la classe Pattern ou de la classe Matcher :

- matches() : vérifier la correspondance intégrale vis-à-vis d'une expression régulière
- replaceFirst() : remplacer la première occurrence trouvée d'une expression régulière
- replaceAll() : remplacer toutes les occurrences trouvées d'une expression régulière
- split() : découper la chaîne selon une expression régulière

Remarque : la méthode replace() n'utilise pas d'expression régulière.

Ces méthodes sont pratiques mais elles ne sont pas optimisées d'un point de vue performance.

La méthode matches() de la classe String permet de vérifier la correspondance intégrale de la chaîne avec l'expression régulière fournie. Elle renvoie un booléen qui indique si cette correspondance est vérifiée ou non.

Une invocation de cette méthode de la forme str.matches(regex) est équivalente à Pattern.matches(regex, str).

Exemple :

```
package fr.jmdoudoux.dej.regex;

public class RegEx {

  public static void main(String[] args) {

    String chaine = "Java 11";
    System.out.println(chaine.matches("Java [0-9]*"));
  }
}
```

Résultat :

```
true
```

La méthode replaceFirst() de la classe String remplace la première occurrence trouvée correspondant à l'expression régulière fournie par la valeur fournie.

Une invocation de cette méthode de la forme str.replaceFirst(regex, rempl) est équivalente à Pattern.compile(refex).matcher(str).replaceFirst(rempl).

Exemple :

```
package fr.jmdoudoux.dej.regex;

public class RegEx {

  public static void main(String[] args) {
    String texteJava = "Perl est orienté objet, Perl supporte les regex";
    String textePerl = texteJava.replaceFirst("Perl", "Java");
    System.out.println(textePerl);
  }
}
```

Résultat :

```
Java est orienté objet, Perl supporte les regex
```

La méthode replaceAll() de la classe String remplace toutes les occurrences trouvées correspondant à l'expression régulière fournie par la valeur fournie.

Une invocation de cette méthode de la forme str.replaceAll(regex, rempl) est équivalente à Pattern.compile(refex).matcher(str).replaceAll(rempl).

Exemple :

```
package fr.jmdoudoux.dej.regex;

public class RegEx {

  public static void main(String[] args) {
    String texteJava = "Perl est orienté objet, Perl supporte les regex";
    String textePerl = texteJava.replaceAll("Perl", "Java");
    System.out.println(textePerl);
  }
}
```

Résultat :

```
Java est orienté objet, Java supporte les regex
```

Remarque : elle ne remplace que les caractères ASCII. Si ce comportement ne correspond pas au besoin, il faut utiliser un Matcher avec le mode adéquat.

La méthode split() de la classe String découpe la chaîne de caractères en fonction des correspondances de l'expression régulière fournie.

Elle possède deux surcharges :

Méthode	Rôle
String[] split(String regex, int limit)	Découper la chaîne de caractères selon les correspondances de l'expression régulière fournie. Une invocation de cette méthode sous la forme str.split(regex, n) est équivalente à Pattern.compile(regex).split(str, n). Le paramètre limit contrôle le nombre de fois que le motif est appliqué et affecte donc la longueur du tableau retourné. Les correspondances ne sont pas incluses dans le tableau retourné.
String[] split(String regex)	Découper la chaîne selon les correspondances de l'expression régulière fournie. Cette méthode fonctionne de la même manière que l'invocation la méthode split() à deux arguments avec l'expression régulière fournie et un argument limite de zéro. Les chaînes vides à la fin ne sont pas incluses dans le tableau retourné.

Exemple :

```
package fr.jmdoudoux.dej.regex;

public class RegEx {

  public static void main(String[] args) {

    String chaine = "element1:element2:element3";
    String[] elements = chaine.split(":");
    for (String element : elements) {
      System.out.println(element);
    }
  }
}
```

Résultat :

```
element1
element2
element3
```

Partie 3 : Les API avancées

Le JDK fournit un certain nombres d'API avancées.

Cette partie contient les chapitres suivants :

- La gestion dynamique des objets et l'introspection : ces mécanismes permettent dynamiquement de connaître le contenu d'une classe et de l'utiliser
- L'appel de méthodes distantes : RMI : étudie la mise en oeuvre de la technologie RMI pour permettre l'appel de méthodes distantes
- La sécurité : partie intégrante de Java, elle revêt de nombreux aspects dans les spécifications, la gestion des droits d'exécution et plusieurs API dédiées
- JCA (Java Cryptography Architecture) : détaille l'utilisation de l'API proposant des fonctionnalités cryptographiques de base
- JCE (Java Cryptography Extension) : détaille l'API pour l'encryptage et le décryptage, la génération de clés et l'authentification de messages avec des algorithmes de type MAC
- JNI (Java Native Interface) : technologie qui permet d'utiliser du code natif dans une classe Java et vice versa
- JNDI (Java Naming and Directory Interface) : introduit l'API qui permet d'accéder aux services de nommage et d'annuaires
- Le scripting : L'utilisation d'outils de scripting avec Java a longtemps été possible au travers de produits open source. Depuis la version 6.0 de Java, une API standard est proposée.
- JMX (Java Management Extensions) : ce chapitre détaille l'utilisation de JMX. C'est une spécification qui définit une architecture, une API et des services pour permettre de surveiller et de gérer des ressources en Java
- L'API Service Provider (SPI) : ce chapitre détaille la mise en oeuvre de services en utilisant l'API ServiceLoader

24. La gestion dynamique des objets et l'introspection

Chapitre 24

Niveau : 🟩🟩🟨🟨🟥⬜ Confirmé

Depuis la version 1.1 de Java, il est possible de créer et de gérer dynamiquement des objets.

L'introspection est un mécanisme qui permet de connaître le contenu d'une classe dynamiquement. Il permet notamment de savoir ce que contient une classe sans en avoir les sources. Ces mécanismes sont largement utilisés dans des outils de type IDE (Integrated Development Environnement : environnement de développement intégré).

Pour illustrer ces différents mécanismes, ce chapitre va construire une classe utilitaire qui proposera un ensemble de méthodes fournissant des informations sur une classe donnée.

Les différentes classes utiles pour l'introspection sont rassemblées dans le package java.lang.reflect.

Voici le début de cette classe qui attend dans son constructeur une chaîne de caractères précisant la classe sur laquelle elle va travailler.

Exemple (code Java 1.1) :

```
import java.util.*;
import java.lang.reflect.*;

public class ClasseInspecteur {
  private Class classe;
  private String nomClasse;

  public ClasseInspecteur(String nomClasse) {
    this.nomClasse = nomClasse;
    try {
      classe = Class.forName(nomClasse);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Ce chapitre contient plusieurs sections :

♦ La classe Class
♦ La recherche des informations sur une classe
♦ La définition dynamique d'objets
♦ L'invocation dynamique d'une méthode
♦ L'API Reflection et le SecurityManager
♦ L'utilisation de l'API Reflection sur les annotations

24.1. La classe Class

Les instances de la classe Class sont des objets représentant les classes du langage. Il y aura une instance représentant chaque classe utilisée : par exemple la classe String, la classe Frame, la classe Class, etc ... Ces instances sont crées automatiquement par la machine virtuelle lors du chargement de la classe. Il est ainsi possible de connaître les caractéristiques d'une classe de façon dynamique en utilisant les méthodes de la classe Class. Les applications telles que les débogueurs, les inspecteurs d'objets et les environnements de développement doivent faire une analyse des objets qu'ils manipulent en utilisant ces mécanismes.

La classe Class est définie dans le package java.lang.

La classe Class permet :

- de décrire une classe ou une interface par introspection : obtenir son nom, sa classe mère, la liste de ses méthodes, de ses variables de classe, de ses constructeurs et variables d'instances, etc ...
- d'agir sur une classe en envoyant des messages à un objet Class comme à tout autre objet. Par exemple, créer dynamiquement à partir d'un objet Class une nouvelle instance de la classe représentée

24.1.1. L'obtention d'un objet de type Class

La classe Class ne possède pas de constructeur public mais il existe plusieurs façons d'obtenir un objet de la classe Class.

24.1.1.1. La détermination de la classe d'un objet

La méthode getClass() définit dans la classe Object renvoie une instance de la classe Class. Par héritage, tout objet Java dispose de cette méthode.

Exemple (code Java 1.1) :

```
package introspection;

public class TestGetClass {

  public static void main(java.lang.String[] args) {
    String chaine = "test";
    Class classe = chaine.getClass();
    System.out.println("classe de l'objet chaine = "+classe.getName());
  }
}
```

Résultat :

```
classe de l'objet chaine = java.lang.String
```

24.1.1.2. L'obtention d'un objet Class à partir d'un nom de classe

La classe Class possède une méthode statique forName() qui permet à partir d'une chaîne de caractères désignant une classe d'instancier un objet de cette classe et de renvoyer un objet de la classe Class pour cette classe.

Cette méthode peut lever l'exception ClassNotFoundException.

Exemple (code Java 1.1) :

```
public class TestForName {

  public static void main(java.lang.String[] args) {
    try {
      Class classe = Class.forName("java.lang.String");
      System.out.println("classe de l'objet chaine = "+classe.getName());
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
classe de l'objet chaîne = java.lang.String
```

24.1.1.3. Une troisième façon d'obtenir un objet Class

Il est possible d'avoir un objet de la classe Class en écrivant type.class où type est le nom d'une classe.

Exemple (code Java 1.1) :

```
package introspection;

public class TestClass {

  public static void main(java.lang.String[] args) {
    Class c = Object.class;
    System.out.println("classe de Object  = "+c.getName());
  }
}
```

Résultat :

```
classe de Object  = java.lang.Object
```

24.1.2. Les méthodes de la classe Class

La classe Class fournit de nombreuses méthodes pour obtenir des informations sur la classe qu'elle représente. Voici les principales méthodes :

Méthodes	Rôle
static Class forName(String)	Instancier un objet de la classe dont le nom est fourni en paramètre et renvoie un objet Class la représentant
Class[] getClasses()	Renvoyer les classes et interfaces publiques qui sont membres de la classe
Constructor[] getConstructors()	Renvoyer les constructeurs publics de la classe
Class[] getDeclaredClasses()	Renvoyer un tableau des classes définies comme membres dans la classe
Constructor[] getDeclaredConstructors()	Renvoyer tous les constructeurs de la classe
Field[] getDeclaredFields()	Renvoyer un tableau de tous les attributs définis dans la classe
Method[] getDeclaredMethods()	Renvoyer un tableau de toutes les méthodes
Field[] getFields()	Renvoyer un tableau des attributs publics
Class[] getInterfaces()	Renvoyer un tableau des interfaces implémentées par la classe
Method[] getMethod()	Renvoyer un tableau des méthodes publiques de la classe incluant celles héritées
int getModifiers()	Renvoyer un entier qu'il faut décoder pour connaître les modificateurs de la classe
Package getPackage()	Renvoyer le package de la classe
Class<?>[] getPermittedSubclasses()	Renvoyer un tableau des type autorisés à hériter du type scellé (Java 17)
Classe getSuperClass()	Renvoyer la classe mère de la classe

boolean isArray()	Indiquer si la classe est un tableau
boolean IsInterface()	Indiquer si la classe est une interface
boolean isSealed()	Indiquer si le type est scellé (Java 17)
Object newInstance()	Créer une nouvelle instance de la classe

24.2. La recherche des informations sur une classe

En utilisant les méthodes de la classe Class, il est possible d'obtenir quasiment toutes les informations sur une classe.

24.2.1. La recherche de la classe mère d'une classe

La classe Class possède une méthode getSuperClass() qui retourne un objet de la classe Class représentant la classe mère si elle existe sinon elle retourne null.

Pour obtenir toute la hiérarchie d'une classe il suffit d'appeler successivement cette méthode sur l'objet qu'elle a retourné.

Exemple (code Java 1.1) : méthode qui retourne un vecteur contenant les classes mères

```
  public Vector getClassesParentes() {

    Vector cp = new Vector();

    Class sousClasse = classe;
    Class superClasse;

    cp.add(sousClasse.getName());
    superClasse = sousClasse.getSuperclass();
    while (superClasse != null) {
      cp.add(0,superClasse.getName());
      sousClasse = superClasse;
      superClasse = sousClasse.getSuperclass();
    }
    return cp;
  }
```

24.2.2. La recherche des modificateurs d'une classe

La classe Class possède une méthode getModifiers() qui retourne un entier représentant les modificateurs de la classe. Pour décoder cette valeur, la classe Modifier possède plusieurs méthodes qui attendent cet entier en paramètre et qui retournent un booléen selon leur fonction : isPublic(), isAbstract(), isFinal(), ...

La classe Modifier ne contient que des constantes et des méthodes statiques qui permettent de déterminer les modificateurs d'accès :

Méthode	Rôle
boolean isAbstract(int)	Renvoyer true si le paramètre contient le modificateur abstract
boolean isFinal(int)	Renvoyer true si le paramètre contient le modificateur final
boolean isInterface(int)	Renvoyer true si le paramètre contient le modificateur interface
boolean isNative(int)	Renvoyer true si le paramètre contient le modificateur native
boolean isPrivate(int)	Renvoyer true si le paramètre contient le modificateur private
boolean isProtected(int)	Renvoyer true si le paramètre contient le modificateur protected
boolean isPublic(int)	Renvoyer true si le paramètre contient le modificateur public
boolean isStatic(int)	Renvoyer true si le paramètre contient le modificateur static
boolean isSynchronized(int)	Renvoyer true si le paramètre contient le modificateur synchronized

boolean isTransient(int)	Renvoyer true si le paramètre contient le modificateur transient
boolean isVolatile(int)	Renvoyer true si le paramètre contient le modificateur volatile

Ces méthodes étant static il est inutile d'instancier un objet de type Modifier pour les utiliser.

Exemple (code Java 1.1) :

```
public Vector getModificateurs() {
  Vector cp = new Vector();

  int m = classe.getModifiers();
  if (Modifier.isPublic(m))
    cp.add("public");
  if (Modifier.isAbstract(m))
    cp.add("abstract");
  if (Modifier.isFinal(m))
    cp.add("final");
  return cp;
}
```

24.2.3. La recherche des interfaces implémentées par une classe

La classe Class possède une méthode getInterfaces() qui retourne un tableau d'objets de type Class contenant les interfaces implémentées par la classe.

Exemple (code Java 1.1) :

```
public Vector getInterfaces() {
  Vector cp = new Vector();

  Class[] interfaces = classe.getInterfaces();
  for (int i = 0; i < interfaces.length; i++) {
    cp.add(interfaces[i].getName());
  }
  return cp;
}
```

24.2.4. La recherche des champs publics

La classe Class possède une méthode getFields() qui retourne les attributs public de la classe. Cette méthode retourne un tableau d'objets de type Field.

La classe Class possède aussi une méthode getField() qui attend en paramètre un nom d'attribut et retourne un objet de type Field si celui-ci est défini dans la classe ou dans une de ses classes mères. Si la classe ne contient pas d'attribut dont le nom correspond au paramètre fourni, la méthode getField() lève une exception de la classe NoSuchFieldException.

La classe Field représente un attribut d'une classe ou d'une interface et permet d'obtenir des informations sur cet attribut. Elle possède plusieurs méthodes :

Méthode	Rôle
String getName()	Retourner le nom de l'attribut
Class getType()	Retourner un objet de type Class qui représente le type de l'attribut
Class getDeclaringClass()	Retourner un objet de type Class qui représente la classe qui définit l'attribut
int getModifiers()	Retourner un entier qui décrit les modificateurs d'accès. Pour les connaître précisément il faut utiliser les méthodes static de la classe Modifier.
Object get(Object)	Retourner la valeur de l'attribut pour l'instance de l'objet fournie en paramètre. Il existe aussi plusieurs méthodes getXXX() où XXX représente un type primitif et qui renvoient

	la valeur dans ce type.

Exemple (code Java 1.1) :

```
public Vector getChampsPublics() {
  Vector cp = new Vector();

  Field[] champs = classe.getFields();
  for (int i = 0; i < champs.length; i++)
    cp.add(champs[i].getType().getName()+" "+champs[i].getName());
  return cp;
}
```

24.2.5. La recherche des paramètres d'une méthode ou d'un constructeur

L'exemple ci-dessous présente une méthode qui permet de formater sous forme de chaîne de caractères les paramètres d'une méthode fournis sous la forme d'un tableau d'objets de type Class.

Exemple (code Java 1.1) :

```
private String rechercheParametres(Class[] classes) {
  StringBuffer param = new StringBuffer("(");

  for (int i = 0; i < classes.length; i ++) {
    param.append(formatParametre(classes[i].getName()));
    if (i < classes.length - 1)
      param.append(", ");
  }
  param.append(")");

  return param.toString();
}
```

La méthode getName() de la classe Class renvoie une chaîne de caractères formatée qui précise le type de la classe. Ce type est représenté par une chaîne de caractères qu'il faut décoder pour l'extraire.

Si le type de la classe est un tableau alors la chaîne commence par un nombre de caractères '[' correspondant à la dimension du tableau.

Ensuite la chaîne contient un caractère qui précise un type primitif ou un objet. Dans le cas d'un objet, le nom de la classe de l'objet avec son package complet est contenu dans la chaîne suivie d'un caractère ';'.

Caractère	Type
B	byte
C	char
D	double
F	float
I	int
J	long
L*classname;*	*classe ou interface*
S	short
Z	boolean

Exemple :

La méthode getName() de la classe Class représentant un objet de type float[10][5] renvoie « [[F »

Pour simplifier les traitements, la méthode formatParametre() ci-dessous retourne une chaîne de caractères qui décode le contenu de la chaîne retournée par la méthode getName() de la classe Class.

Exemple :

```
  private String formatParametre(String s) {

    if (s.charAt(0) == '[') {

      StringBuffer param = new StringBuffer("");
      int dimension = 0;
      while (s.charAt(dimension) == '[') dimension++;

      switch(s.charAt(dimension)) {
        case 'B' : param.append("byte");break;
        case 'C' : param.append("char");break;
        case 'D' : param.append("double");break;
        case 'F' : param.append("float");break;
        case 'I' : param.append("int");break;
        case 'J' : param.append("long");break;
        case 'S' : param.append("short");break;
        case 'Z' : param.append("boolean");break;
        case 'L' : param.append(s.substring(dimension+1,s.indexOf(";")));
      }

      for (int i =0; i < dimension; i++)
        param.append("[]");

      return param.toString();
    }
    else return s;

  }
```

24.2.6. La recherche des constructeurs de la classe

La classe Class possède une méthode getConstructors() qui retourne un tableau d'objets de type Constructor contenant les constructeurs de la classe.

La classe Constructor représente un constructeur d'une classe et possède plusieurs méthodes :

Méthode	Rôle
String getName()	Retourner le nom du constructeur
Class[] getExceptionTypes()	Retourner un tableau de type Class qui représente les exceptions qui peuvent être propagées par le constructeur
Class[] getParametersType()	Retourner un tableau de type Class qui représente les paramètres du constructeur
int getModifiers()	Retourner un entier qui décrit les modificateurs d'accès. Pour les connaître précisément il faut utiliser les méthodes static de la classe Modifier.
Object newInstance(Object[])	Instancier un objet en utilisant le constructeur avec les paramètres fournis à la méthode

Exemple (code Java 1.1) :

```
  public Vector getConstructeurs() {

    Vector cp = new Vector();
    Constructor[] constructeurs = classe.getConstructors();
    for (int i = 0; i < constructeurs.length; i++) {
      cp.add(rechercheParametres(constructeurs[i].getParameterTypes()));
    }

    return cp;
  }
```

L'exemple ci-dessus utilise la méthode rechercherParamètres() définie précédemment pour simplifier les traitements.

24.2.7. La recherche des méthodes publiques

Pour consulter les méthodes d'un objet, il faut obtenir sa classe et lui envoyer le message getMethod(), qui renvoie les méthodes publiques qui sont déclarées dans la classe ou qui sont héritées des classes mères.

Elle renvoie un tableau d'instances de la classe Method du package java.lang.reflect.

Une méthode est caractérisée par un nom, une valeur de retour, une liste de paramètres, une liste d'exceptions et une classe d'appartenance.

La classe Method contient plusieurs méthodes :

Méthode	Rôle
Class[] getParameterTypes	Renvoyer un tableau de classes représentant les paramètres.
Class getReturnType	Renvoyer le type de la valeur de retour de la méthode.
String getName()	Renvoyer le nom de la méthode
int getModifiers()	Renvoyer un entier qui représente les modificateurs d'accès
Class[] getExceptionTypes	Renvoyer un tableau de classes contenant les exceptions propagées par la méthode
Class getDeclaringClass[]	Renvoyer la classe qui définit la méthode

Exemple (code Java 1.1) :

```
public Vector getMethodesPubliques() {
  Vector cp = new Vector();

  Method[] methodes = classe.getMethods();
  for (int i = 0; i < methodes.length; i++) {
    StringBuffer methode = new StringBuffer();

    methode.append(formatParametre(methodes[i].getReturnType().getName()));
    methode.append(" ");
    methode.append(methodes[i].getName());
    methode.append(rechercheParametres(methodes[i].getParameterTypes()));

    cp.add(methode.toString());
  }
  return cp;
}
```

L'exemple ci-dessus utilise les méthodes formatParametre() et rechercherParametres() définies précédemment pour simplifier les traitements.

24.2.8. La recherche de toutes les méthodes

Pour consulter toutes les méthodes d'un objet, il faut obtenir sa classe et lui envoyer le message getDeclaredMethods(), qui renvoie toutes les méthodes qui sont déclarées dans la classe ou qui sont héritées des classes mères quelque soit leur accessibilité.

Elle renvoie un tableau d'instances de la classe Method du package java.lang.reflect.

Exemple :

```
public List getSignatureMethodes() {
    List cp = new ArrayList();
    Method[] methodes = classe.getDeclaredMethods();
    for (int i = 0; i < methodes.length; i++) {
      StringBuffer methode = new StringBuffer();

      methode.append(formatParametre(methodes[i].getReturnType().getName()));
      methode.append(" ");
      methode.append(methodes[i].getName());
      methode.append(rechercheParametres(methodes[i].getParameterTypes()));

      cp.add(methode.toString());
    }
    return cp;
  }
```

L'exemple ci-dessus utilise les méthodes formatParametre() et rechercherParametres() définies précédemment pour simplifier les traitements.

24.2.9. La recherche des getters et des setters

Par convention, la valeur d'une propriété est gérée grâce à deux méthodes public :

- un getter : c'est une méthode qui permet d'obtenir la valeur de la propriété. Son nom commence par « get » suivi du nom de la propriété. Elle ne doit pas avoir de paramètre et renvoie la valeur de la propriété
- un setter : c'est une méthode pour modifier la valeur d'une propriété. Son nom commence par « set » suivi du nom de la propriété. Elle n'attend qu'un seul paramètre qui est la nouvelle valeur de la propriété et ne renvoie rien (void)

Même si elle ne le propose pas directement, il est possible d'utiliser l'API Reflection pour obtenir les getters et les setters d'une classe.

Exemple :

```
package fr.jmdoudoux.dej.reflection;

import java.beans.BeanInfo;
import java.lang.reflect.Method;

public class TestGetterSetters {

  public static void main(String args[]) {
    afficherGettersSetters(MaClasse.class);
  }

  public static void afficherGettersSetters(Class aClass) {
    Method[] methods = aClass.getMethods();
    System.out.println("classe : " + aClass.getName());
    for (Method method : methods) {
      if (isGetter(method)) {
        System.out.println("  getter : " + method);
      }
      if (isSetter(method)) {
        System.out.println("  setter : " + method);
      }
    }
  }

  public static boolean isGetter(Method method) {
    boolean result = method.getName().startsWith("get")
            && (method.getParameterTypes().length == 0)
            && (!Void.class.equals(method.getReturnType()));
    return result;
  }

  public static boolean isSetter(Method method) {
    boolean result = (method.getName().startsWith("set"))
```

```
            && (method.getParameterTypes().length == 1);
    return result;
  }
}
```

24.2.10. Le support des types scellés

En Java 17, la classe Class est enrichie de deux nouvelles méthodes pour obtenir des informations relatives aux classes scellées :

- Class<?>[] getPermittedSubclasses()
- boolean isSealed()

La méthode getPermittedSubclasses() renvoie un tableau de type java.lang.Class qui contient toutes les sous-classes directes autorisées d'une classe scellée. L'ordre des classes n'est pas déterminé. Elle renvoie un tableau vide si la classe n'est pas scellée. Elle renvoie null si la classe du type est un tableau ou un type primitif.

La méthode isSealed() renvoie un booléen qui indique si le type (la classe ou l'interface) est scellé.

Exemple (code Java 17) :

```
package fr.jmdoudoux.dej.typescelles;

public sealed class Forme permits Triangle, Cercle {

  public static void main(String[] args) {
    System.out.println(Forme.class.isSealed());
    System.out.println(Triangle.class.isSealed());

    Class<?>[] permittedSubclasses = Forme.class.getPermittedSubclasses();

    System.out.println("Classes filles autorisees :");
    for (Class<?> clazz : permittedSubclasses) {
      System.out.println("  "+clazz);
    }
  }
}

non-sealed class Triangle extends Forme { }

final class TriangleRectangle extends Triangle { }

final class Cercle extends Forme { }
```

Résultat :

```
true
false
Classes filles autorisees :
  class fr.jmdoudoux.dej.typescelles.Triangle
  class fr.jmdoudoux.dej.typescelles.Cercle
```

24.3. La définition dynamique d'objets

L'API Reflection permet de créer dynamiquement des instances d'un type.

24.3.1. La création d'objets grâce à la classe Class

La méthode statique forName() de la classe Class permet de charger dynamiquement une classe dont le nom pleinement qualifié est fourni en paramètre. Elle renvoie une instance de la classe Class qui encapsule la classe chargée.

La méthode newInstance() de la classe Class permet de créer une instance de la classe et d'invoquer son constructeur par défaut.

Exemple (code Java 1.4) :

```
import java.util.logging.Level;
import java.util.logging.Logger;

import fr.jmdoudoux.dej.introspection.MaClasse;

public class TestNewInstance {
  public static Logger LOGGER     = Logger.getLogger("TestNewInstance");
  public static String NOM_CLASSE = "fr.jmdoudoux.dej.introspection.MaClasse";

  public static void main(String[] args) {
    try {
      Class classe = Class.forName(NOM_CLASSE);
      MaClasse instance = (MaClasse) classe.newInstance();
      instance.afficher();
    } catch (ClassNotFoundException cnfe) {
      if (LOGGER.isLoggable(Level.SEVERE))
        LOGGER.log(Level.SEVERE, "La classe " + NOM_CLASSE + " n'existe pas",
            cnfe);
    } catch (InstantiationException ie) {
      if (LOGGER.isLoggable(Level.SEVERE))
        LOGGER.log(Level.SEVERE, "La classe " + NOM_CLASSE
            + " n'est pas instanciable", ie);
    } catch (IllegalAccessException iae) {
      if (LOGGER.isLoggable(Level.SEVERE))
        LOGGER.log(Level.SEVERE, "La classe " + NOM_CLASSE
            + " n'est pas accessible", iae);
    }
  }
}
```

A partir de Java 5, la classe Class est générique.

Exemple (code Java 5.0) :

```
import java.util.logging.Level;
import java.util.logging.Logger;

import fr.jmdoudoux.dej.introspection.MaClasse;

public class TestNewInstance {
  public static Logger LOGGER     = Logger.getLogger("TestNewInstance");
  public static String NOM_CLASSE = "fr.jmdoudoux.dej.introspection.MaClasse";

  public static void main(String[] args) {
    try {
      Class<MaClasse> classe = (Class<MaClasse>) Class.forName(NOM_CLASSE);
      MaClasse instance = classe.newInstance();
      instance.afficher();
    } catch (ClassNotFoundException cnfe) {
      if (LOGGER.isLoggable(Level.SEVERE))
        LOGGER.log(Level.SEVERE, "La classe " + NOM_CLASSE + " n'existe pas", cnfe);
    } catch (InstantiationException ie) {
      if (LOGGER.isLoggable(Level.SEVERE))
        LOGGER.log(Level.SEVERE, "La classe " + NOM_CLASSE
            + " n'est pas instanciable", ie);
    } catch (IllegalAccessException iae) {
      if (LOGGER.isLoggable(Level.SEVERE))
        LOGGER.log(Level.SEVERE, "La classe " + NOM_CLASSE
            + " n'est pas accessible", iae);
    }
  }
}
```

La méthode newInstance() de la classe Class présente plusieurs contraintes :

- seul le constructeur sans paramètre peut être invoqué
- ce constructeur doit être public
- toutes les exceptions checked et unchecked levées lors de l'invocation du constructeur sont propagées

24.3.2. La création d'objets grâce à la classe Constructor

A partir de la version 1.1, le package java.lang.reflect propose la classe Constructor pour créer des instances en invoquant un constructeur quelconque d'une classe.

La méthode getDeclaredConstructor() de la classe Class permet d'obtenir une instance de la classe Constructor qui encapsule le constructeur dont les types des paramètres ont été fournis à la méthode getDeclaredConstructor().

La méthode getDeclaredMethod() attend en paramètre un tableau d'objets de type Class qui doit contenir les types de chaque paramètre dans l'ordre de leur définition dans la signature du constructeur souhaité.

La classe Constructor propose la méthode newInstance() qui attend en paramètre un tableau de type Object devant contenir les valeurs qui seront fournies lors de l'invocation du constructeur.

Exemple :

```
import java.lang.reflect.Constructor;
import java.util.logging.Level;
import java.util.logging.Logger;

import fr.jmdoudoux.dej.introspection.MaClasse;

public class TestGetConstrutor {

  public static Logger LOGGER     = Logger.getLogger("TestGetConstrutor");
  public static String NOM_CLASSE = "fr.jmdoudoux.dej.introspection.MaClasse";

  public static void main(String[] args) {
    try {
      Class classe = Class.forName(NOM_CLASSE);
      Constructor constructeur = classe.getConstructor(new Class[] {
          boolean.class, Class.forName("java.lang.String") });
      MaClasse instance = (MaClasse) constructeur.newInstance(new Object[] {
          Boolean.FALSE, "nom instance" });
      instance.afficher();
    } catch (ClassNotFoundException cnfe) {
      if (LOGGER.isLoggable(Level.SEVERE))
        LOGGER.log(Level.SEVERE, "La classe " + NOM_CLASSE + " n'existe pas",
            cnfe);
    } catch (NoSuchMethodException nme) {
      if (LOGGER.isLoggable(Level.SEVERE))
        LOGGER.log(Level.SEVERE, "Le constructeur de la classe " + NOM_CLASSE
            + " n'existe pas", nme);
    } catch (InstantiationException ie) {
      if (LOGGER.isLoggable(Level.SEVERE))
        LOGGER.log(Level.SEVERE, "La classe " + NOM_CLASSE
            + " n'est pas instanciable", ie);
    } catch (IllegalAccessException iae) {
      if (LOGGER.isLoggable(Level.SEVERE))
        LOGGER.log(Level.SEVERE, "La classe " + NOM_CLASSE
            + " n'est pas accessible", iae);
    } catch (java.lang.reflect.InvocationTargetException ite) {
      if (LOGGER.isLoggable(Level.SEVERE))
        LOGGER.log(Level.SEVERE, "Le constructueur de la classe " + NOM_CLASSE
            + " a leve une exception", ite);
    } catch (IllegalArgumentException iae) {
      if (LOGGER.isLoggable(Level.SEVERE))
        LOGGER.log(Level.SEVERE, "Un parametre du constructueur de la classe "
            + NOM_CLASSE + " n'est pas du bon type", iae);
    }
  }
}
```

A partir de Java 5, les classes Class et Constructor sont génériques.

Exemple (code Java 5.0) :

```
import java.lang.reflect.Constructor;
import java.util.logging.Level;
import java.util.logging.Logger;
import fr.jmdoudoux.dej.introspection.MaClasse;

public class TestGetConstrutor {

  public static Logger LOGGER     = Logger.getLogger("TestGetConstrutor");
  public static String NOM_CLASSE = "fr.jmdoudoux.dej.introspection.MaClasse";

  public static void main(String[] args) {
    try {
      Class<MaClasse> classe = (Class<MaClasse>) Class.forName(NOM_CLASSE);
      Constructor<MaClasse> constructeur = classe.getConstructor(new Class[] {
          boolean.class, Class.forName("java.lang.String") });
      MaClasse instance = constructeur.newInstance(new Object[] {
          Boolean.FALSE, "nom instance" });
      instance.afficher();
    } catch (ClassNotFoundException cnfe) {
      if (LOGGER.isLoggable(Level.SEVERE))
        LOGGER.log(Level.SEVERE, "La classe " + NOM_CLASSE + " n'existe pas",
            cnfe);
    } catch (NoSuchMethodException nme) {
      if (LOGGER.isLoggable(Level.SEVERE))
        LOGGER.log(Level.SEVERE, "Le constructeur de la classe " + NOM_CLASSE
            + " n'existe pas", nme);
    } catch (InstantiationException ie) {
      if (LOGGER.isLoggable(Level.SEVERE))
        LOGGER.log(Level.SEVERE, "La classe " + NOM_CLASSE
            + " n'est pas instanciable", ie);
    } catch (IllegalAccessException iae) {
      if (LOGGER.isLoggable(Level.SEVERE))
        LOGGER.log(Level.SEVERE, "La classe " + NOM_CLASSE
            + " n'est pas accessible", iae);
    } catch (java.lang.reflect.InvocationTargetException ite) {
      if (LOGGER.isLoggable(Level.SEVERE))
        LOGGER.log(Level.SEVERE, "Le constructueur de la classe " + NOM_CLASSE
            + " a leve une exception", ite);
    } catch (IllegalArgumentException iae) {
      if (LOGGER.isLoggable(Level.SEVERE))
        LOGGER.log(Level.SEVERE, "Un parametre du constructueur de la classe "
            + NOM_CLASSE + " n'est pas du bon type", iae);
    }
  }
}
```

Si une exception est levée lors de l'invocation du constructeur, celle-ci est chaînée dans une exception checked de type TargetInvocationException.

24.4. L'invocation dynamique d'une méthode

L'API Reflection permet d'invoquer dynamiquement une méthode d'un objet.

Pour invoquer dynamiquement une méthode d'une instance, il faut utiliser la méthode invoke(Object obj, Object[] args) de la classe java.lang.Method qui possède plusieurs paramètres :

- le premier paramètre est l'instance sur laquelle la méthode doit être invoquée
- les paramètres suivants sont les valeurs qui seront passées en paramètres lors de l'invocation : un nombre arbitraire de paramètres peuvent être passés. Les valeurs des paramètres fournies doivent respecter le type et l'ordre de la signature de la méthode.

Exemple :

```
package fr.jmdoudoux.dej.reflection;

public class MaClasse {

  public void maMethode() {
    System.out.println("maMethode sans param");
  }

  public String maMethode(String param1) {
    System.out.println("maMethode avec String:"+param1);
    return param1;
  }

  public String maMethode(String param1, int param2) {
    String resultat = param1+param2;
    System.out.println("maMethode avec String:"+param1+", int:"+param2);
    return resultat;
  }

  public String maMethode(String param1, Integer param2) {
    String resultat = param1+param2;
    System.out.println("maMethode avec String:"+param1+", int:"+param2);
    return resultat;
  }

  public void maMethode(int param1) {
    System.out.println("maMethode avec int:"+param1);
  }

  private void maMethodePrivee() {
    System.out.println("maMethodePrivee sans param");
  }

  public static void maMethodeStatic() {
    System.out.println("maMethodeStatic sans param");
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.reflection;

import java.lang.reflect.Method;
import java.lang.reflect.Modifier;

public class TestExecuterMethode {

  public static void main(String[] args) {
    MaClasse maClasse = new MaClasse();
    try {
      Object retour = executerMethode(maClasse, "maMethode", null);
      System.out.println("Valeur de retour = " + retour);
      retour = executerMethode(maClasse, "maMethode", new
        Object[]{"chaine1"});
      System.out.println("Valeur de retour = " + retour);
      retour = executerMethode(maClasse, "maMethode", new
        Object[]{"chaine", 99});
      System.out.println("Valeur de retour = " + retour);
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  public static Object executerMethode(Object objet, String nomMethode,
    Object[] parametres) throws Exception {
    Object retour;
    Class[] typeParametres = null;

    if (parametres != null) {
      typeParametres = new Class[parametres.length];
      for (int i = 0; i < parametres.length; ++i) {
        typeParametres[i] = parametres[i].getClass();
      }
    }
    Method m = objet.getClass().getMethod(nomMethode, typeParametres);
    if (Modifier.isStatic(m.getModifiers())) {
      retour = m.invoke(null, parametres);
    } else {
      retour = m.invoke(objet, parametres);
    }
    return retour;
  }
}
```

Résultat :

```
maMethode sans param
Valeur de retour = null
maMethode avec String:chaine1
Valeur de retour = chaine1
maMethode avec String:chaine, int:99
Valeur de retour = chaine99
```

24.4.1. La passage de paramètre à la méthode invoquée

Une exception de type IllegalArgumentException est levée si aucune méthode dont la signature correspond aux types passés en paramètre n'est trouvée.

Exemple :

```
package fr.jmdoudoux.dej.reflection;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class TestExecuterMethode {

  public static void main(String[] args) {
    MaClasse maClasse = new MaClasse();
    try {
      Class<?> c = maClasse.getClass();
      Method m = c.getMethod("maMethode");
      System.out.format("Methode : %s%n", m.toGenericString());
      m.invoke(maClasse, "test");
    } catch (NoSuchMethodEx ception x){
      x.printStackTrace();
    } catch (IllegalArgumentException ex) {
      ex.printStackTrace();
    } catch (IllegalAccessException ex) {
      ex.printStackTrace();
    } catch (InvocationTargetException ex) {
      ex.printStackTrace();
    }
  }
}
```

Résultat :

```
Methode : public void fr.jmdoudoux.dej.reflection.MaClasse.maMethode()
java.lang.IllegalArgumentException: wrong number of arguments
       at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
       at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
       at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
       at java.lang.reflect.Method.invoke(Method.java:601)
       at fr.jmdoudoux.dej.reflection.TestExecuterMethode.main(TestExecuterMethode.java:14)
```

Comme le second paramètre de la méthode invoke() est un varargs, il est possible de passer un tableau de type Object de taille 0 pour indiquer qu'il n'y a pas de paramètre à la méthode.

Exemple :

```
package fr.jmdoudoux.dej.reflection;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class TestExecuterMethode {

  public static void main(String[] args) {
    MaClasse maClasse = new MaClasse();
    try {
      Class<?> c = maClasse.getClass();
      Method m = c.getMethod("maMethode");
      System.out.format("Methode : %s%n", m.toGenericString());
      m.invoke(maClasse, new Object[0]);
    } catch (NoSuchMethodException x) {
      x.printStackTrace();
    } catch (IllegalArgumentException ex) {
      ex.printStackTrace();
    } catch (IllegalAccessException ex) {
      ex.printStackTrace();
    } catch (InvocationTargetException ex) {
      ex.printStackTrace();
    }
  }
}
```

Résultat :

```
Methode : public void fr.jmdoudoux.dej.reflection.MaClasse.maMethode()
maMethode sans param
```

Si la valeur null est passée comme paramètre de la méthode invoke() pour invoquer une méthode sans paramètre alors le compilateur émet un warning.

Exemple :

```
package fr.jmdoudoux.dej.reflection;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class TestExecuterMethode {

  public static void main(String[] args) {
    MaClasse maClasse = new MaClasse();
    try {
      Class<?> c = maClasse.getClass();
      Method m = c.getMethod("maMethode");
      System.out.format("Methode : %s%n", m.toGenericString());
        m.invoke(maClasse, null);
    } catch (NoSuchMethodException x) {
      x.printStackTrace();
    } catch (IllegalArgumentException ex) {
      ex.printStackTrace();
    } catch (IllegalAccessException ex) {
      ex.printStackTrace();
    } catch (InvocationTargetException ex) {
      ex.printStackTrace();
    }
  }
}
```

Résultat :

```
C:\java\src>javac com/jmdoudoux/test/reflection/TestExecuterMethode.java
com\jmdoudoux\test\reflection\TestExecuterMethode.java:14: warning:
non-varargs call of varargs method with inexact argument type for last parameter;
   m.invoke(maClasse, null);
```

```
             ^
  cast to Object for a varargs call
  cast to Object[] for a non-varargs call and to suppress this warning
1 warning
```

Le compilateur signale par son warning qu'il n'est pas en mesure de déterminer si la valeur null concerne la valeur du premier élément du varargs ou un tableau d'objets null.

Le résultat à l'exécution est tout de même celui attendu.

Résultat :

```
Methode : public void fr.jmdoudoux.dej.reflection.MaClasse.maMethode()
maMethode sans param
```

24.4.2. La gestion d'une exception levée par la méthode invoquée

Lors de l'invocation dynamique d'une méthode en utilisant la méthode invoke(), si une exception est levée par la méthode invoquée alors celle-ci est chainée dans une exception de type java.lang.reflect.InvocationTargetException.

Exemple :

```
package fr.jmdoudoux.dej.reflection;

public class MaClasse {

  public void maMethode(int param1) {
    System.out.println("maMethode avec int:"+param1);
    if (param1 == 10) {
      throw new IllegalStateException("La valeur 10 n'est pas permise.");
    }
  }
}
```

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.reflection;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class TestExecuterMethode {

  public static void main(String[] args) {
    MaClasse maClasse = new MaClasse();
    try {
      Class<?> c = maClasse.getClass();
      Method m = c.getMethod("maMethode", Integer.TYPE);
      System.out.format("Methode : %s%n", m.toGenericString());
      m.invoke(maClasse, Integer.valueOf(10));
    } catch (NoSuchMethodException x) {
      x.printStackTrace();
    } catch (IllegalArgumentException ex) {
      ex.printStackTrace();
    } catch (IllegalAccessException ex) {
      ex.printStackTrace();
    } catch (InvocationTargetException ex) {
      ex.printStackTrace();
      Throwable cause = ex.getCause();
      System.out.println("Cause : "+cause.getMessage());
    }
  }
}
```

Résultat :

```
Methode : public void fr.jmdoudoux.dej.reflection.MaClasse.maMethode(int)
maMethode avec int:10
java.lang.reflect.InvocationTargetException
      at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
      at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
      at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
      at java.lang.reflect.Method.invoke(Method.java:601)
      at fr.jmdoudoux.dej.reflection.TestExecuterMethode.main(TestExecuterMethode.java:14)
Caused by: java.lang.IllegalStateException: La valeur 10 n'est pas permise.
      at fr.jmdoudoux.dej.reflection.MaClasse.maMethode(MaClasse.java:34)
      ... 5 more
Cause : La valeur 10 n'est pas permise.
```

Pour obtenir l'exception levée par la méthode exécutée, il faut utiliser la méthode getCause() de l'exception InvocationTargetException.

24.4.3. L'invocation d'une méthode statique

Si la méthode à invoquer est static alors il faut passer null comme valeur du premier paramètre qui correspond à l'instance à invoquer.

Exemple :

```
package fr.jmdoudoux.dej.reflection;

import java.lang.reflect.Method;

public class TestExecuterMethode {

  public static void main(String[] args) {
    MaClasse maClasse = new MaClasse();
    try {
      Method method = maClasse.getClass().getDeclaredMethod("maMethodeStatic", null);
      Object retour = method.invoke(null);
      System.out.println("Valeur de retour = " + retour);
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

Résultat :

```
maMethodeStatic sans param
Valeur de retour = null
```

24.4.4. L'accès aux méthodes privées

Les méthodes getMethod(String name, Class[] parameterTypes) et getMethods() ne permettent de renvoyer que des méthodes publiques. Pour obtenir des méthodes privées, il faut utiliser les méthodes getDeclaredMethod() et getDeclaredMethods().

Par défaut, l'invocation dynamique d'une méthode inaccessible, par exemple déclarée private, lève une exception de type IllegalAccessException.

Exemple :

```
package fr.jmdoudoux.dej.reflection;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class TestExecuterMethode {
  public static void main(String[] args) {
    MaClasse maClasse = new MaClasse();
    try {
      Method method = maClasse.getClass().getDeclaredMethod("maMethodePrivee", null);
      Object retour = method.invoke(maClasse);
      System.out.println("Valeur de retour = " + retour);
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

Résultat :

```
java.lang.IllegalAccessException:
Class fr.jmdoudoux.dej.reflection.TestExecuterMethode can not access a member
of class fr.jmdoudoux.dej.reflection.MaClasse with modifiers "private"
       at sun.reflect.Reflection.ensureMemberAccess(Reflection.java:95)
       at java.lang.reflect.AccessibleObject.slowCheckMemberAccess(AccessibleObject.java:261)
       at java.lang.reflect.AccessibleObject.checkAccess(AccessibleObject.java:253)
       at java.lang.reflect.Method.invoke(Method.java:594)
       at fr.jmdoudoux.dej.reflection.TestExecuterMethode.main(TestExecuterMethode.java:13)
```

La méthode getDeclaredMethod() ne peut qu'accéder aux méthodes qui sont déclarées dans la classe elle-même : elle ne permet pas d'accéder aux méthodes des super-classes.

Par défaut, les restrictions d'accès à une méthode s'appliquent aussi lors de l'utilisation de l'API Reflection.

La classe Method hérite de la classe AccessibleObject qui possèdent la méthode setAccessible(). Elle attend en paramètre un booléen : elle permet avec la valeur true de retirer les vérifications d'accessibilité qui seront faites pour permettre un accès par introspection à la méthode encapsulée. Cela permet de contourner les vérifications d'accès et ainsi d'accéder à une méthode déclarée privée, protected ou package-private uniquement en utilisant l'API Reflection.

Exemple :

```
package fr.jmdoudoux.dej.reflection;

import java.lang.reflect.Method;
import java.util.logging.Level;
import java.util.logging.Logger;

public class TestPrivateMethodInvoke {

  public static void main(String[] args) {
    MaClasse maClasse = new MaClasse();
    try {
      Method method = maClasse.getClass().getDeclaredMethod("maMethodePrivee", null);
      method.setAccessible(true);
      Object retour = method.invoke(maClasse);
      Logger.getLogger(TestPrivateMethodInvoke.class.getName())
        .log(Level.INFO, "Valeur de retour = " + retour);
    } catch (Exception ex) {
      Logger.getLogger(TestPrivateMethodInvoke.class.getName())
        .log(Level.SEVERE, null, ex);
    }
  }
}
```

24.4.5. L'invocation dynamique d'une méthode avec type generic

Il est nécessaire de tenir compte de plusieurs points lors de l'utilisation de l'introspection pour invoquer une méthode dont le type d'un paramètre est un type générique.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.reflection;

public class MaClasseGenerique<T> {

  public void maMethode(T t) {
    System.out.println("maMethode "+t);
  }
}
```

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.reflection;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class TestExecuterMethode {
  public static void main(String[] args) {
    MaClasseGenerique<Integer> maClasse = new MaClasseGenerique<Integer>();
    try {
      Class<?> c = maClasse.getClass();
      Method m = c.getMethod("maMethode", Integer.class);
      System.out.format("Methode : %s%n", m.toGenericString());
    } catch (NoSuchMethodException x) {
      x.printStackTrace();
    } catch (IllegalArgumentException ex) {
      ex.printStackTrace();
    }
  }
}
```

Résultat :

```
java.lang.NoSuchMethodException:
fr.jmdoudoux.dej.reflection.MaClasseGenerique.maMethode(java.lang.Integer)
       at java.lang.Class.getMethod(Class.java:1622)
       at fr.jmdoudoux.dej.reflection.TestExecuterMethode.main(TestExecuterMethode.java:12)
```

Bien que le type générique de la classe soit Integer, la méthode n'est pas trouvée par introspection en précisant le type Integer comme paramètre.

A cause de l'implémentation des generics qui utilise le type erasure, le type generic original est perdu à la compilation pour laisser le type Object. Lorsque le type de la méthode est un type généric, il faut le remplacer par le type Object lorsque l'API Reflection est utilisée pour invoquer la méthode.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.reflection;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class TestExecuterMethode {

  public static void main(String[] args) {
    MaClasseGenerique<Integer> maClasse = new MaClasseGenerique<Integer>();
    try {
      Class<?> c = maClasse.getClass();
      Method m = c.getMethod("maMethode", Object.class);
      System.out.format("Methode : %s%n", m.toGenericString());
      m.invoke(maClasse, Integer.valueOf(100));
    } catch (NoSuchMethodException x) {
      x.printStackTrace();
    } catch (IllegalAccessException ex) {
      ex.printStackTrace();
    } catch (IllegalArgumentException ex) {
      ex.printStackTrace();
    } catch (InvocationTargetException ex) {
      ex.printStackTrace();
    }
  }
}
```

Résultat :

```
Methode : public void fr.jmdoudoux.dej.reflection.MaClasseGenerique.maMethode(T)
maMethode 100
```

24.5. L'API Reflection et le SecurityManager

L'API Reflection permet la mise en oeuvre de puissantes fonctionnalités : c'est une des raisons qui fait qu'elle est fréquemment utilisée par de nombreux frameworks parmi lesquels Spring ou Hibernate.

Cependant certaines fonctionnalités peuvent aussi être utilisées à des fins malveillantes qui peuvent nuire à la sécurité d'une application (invocation de méthodes, modifications de la valeur de champs, ... même si les modificateurs de ces membres ne permettent normalement pas leur accès, ...).

Les accès à un objet en utilisant l'API Reflection se font en utilisant une implémentation de l'interface AccessibleObject. Pour contourner les vérifications de l'accessibilité aux éléments d'un objet, il faut mettre à true la propriété access en utilisant la méthode setAccessible(). Par contre cela ne désactive pas les vérifications faites par le SecurityManager, s'il y en a un d'activé.

Par défaut, aucun SecurityManager n'est activé dans une JVM. Pour en activer un, il faut soit :

- utiliser l'option -Djava.security.manager au lancement de la JVM
- créer une nouvelle instance de type SecurityManager() et la passer en paramètre de la méthode setSecurityManager de la classe System

Lorsqu'un SecurityManager est activé sur une JVM, il est nécessaire d'autoriser la permission de type ReflectPermission dont le nom est "suppressAccessChecks" pour pouvoir utiliser des fonctionnalités de l'API Reflection. Si cette permission n'est pas donnée, alors une exception est levée par la méthode checkPermission() lors de l'utilisation de ces fonctionnalités.

Exemple :

```
package fr.jmdoudoux.dej.reflection;

import java.lang.reflect.Method;

public class TestExecuterMethode {

  public static void main(String[] args) {
    System.setSecurityManager(new SecurityManager());
    MaClasse maClasse = new MaClasse();
    try {
      Method method = maClasse.getClass().getDeclaredMethod("maMethodePrivee", null);
      method.setAccessible(true);
      Object retour = method.invoke(maClasse);
      System.out.println("Valeur de retour = " + retour);
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

Résultat :

```
java.security.AccessControlException:
access denied ("java.lang.reflect.ReflectPermission" "suppressAccessChecks")
       at java.security.AccessControlContext.checkPermission(AccessControlContext.java:366)
       at java.security.AccessController.checkPermission(AccessController.java:555)
       at java.lang.SecurityManager.checkPermission(SecurityManager.java:549)
       at java.lang.reflect.AccessibleObject.setAccessible(AccessibleObject.java:128)
       at fr.jmdoudoux.dej.reflection.TestExecuterMethode.main(TestExecuterMethode.java:14)
```

Il est nécessaire de définir ou de modifier la politique de sécurité pour accorder la permission "suppressAccessChecks" à la classe java.lang.reflect.ReflectPermission.

Il est possible de définir son propre fichier qui contient la définition de la politique de sécurité à appliquer.

Le fichier TestExecuterMethode.policy

```
grant {
  permission java.lang.reflect.ReflectPermission "suppressAccessChecks";
};
```

Il faut préciser le fichier comme valeur de la propriété java.security.policy de la JVM.

Exemple :

```
package fr.jmdoudoux.dej.reflection;

import java.lang.reflect.Method;

public class TestExecuterMethode {

  public static void main(String[] args) {
    System.setProperty("java.security.policy",
      "file:/C:/java/TestReflection/src/TestExecuterMethode.policy");
    System.setSecurityManager(new SecurityManager());

    MaClasse maClasse = new MaClasse();
    try {
      Method method = maClasse.getClass().getDeclaredMethod("maMethodePrivee", null);
      method.setAccessible(true);
      Object retour = method.invoke(maClasse);
      System.out.println("Valeur de retour = " + retour);
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

Il est important de modifier la valeur de la propriété avant l'activation du SecurityManager sinon il faut ajouter une permission autorisant la modification des propriétés système.

Attention : il faut autoriser la permission "suppressAccessChecks" avec précaution en limitant son effet uniquement sur les classes connues pour en avoir besoin. Typiquement, dans l'exemple ci-dessus, cette permission est donnée à toutes les classes dans la JVM ce qui peut être à l'origine de problèmes de sécurité.

Une exception de type SecurityException est levée si la méthode setAccessible() est invoquée sur une instance de type Constructor pour la classe Class.

Exemple :

```
package fr.jmdoudoux.dej.reflection;

import java.lang.reflect.Constructor;

public class TestConstructeurClass {

  public static void main(String[] args) {
    Class classe = Class.class;
    try {
      Constructor constructeur = classe.getDeclaredConstructor();
      constructeur.setAccessible(true);
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

Résultat :

```
java.lang.SecurityException: Can not make a java.lang.Class constructor accessible
        at java.lang.reflect.AccessibleObject.setAccessible0(AccessibleObject.java:139)
        at java.lang.reflect.AccessibleObject.setAccessible(AccessibleObject.java:129)
        at fr.jmdoudoux.dej.reflection.TestPrivateConstructeurInvoke.main(
TestPrivateConstructeurInvoke.java:11)
```

24.6. L'utilisation de l'API Reflection sur les annotations

Les annotations permettent d'ajouter des métadonnées dans le code source Java. Ces métadonnées peuvent être exploitées dans le code source, à la compilation ou à l'exécution en utilisant l'API Reflection.

Elle permet d'accéder aux annotations définies sur un type, une méthode, un champ ou un paramètre de manière dynamique à l'exécution.

Pour pouvoir utiliser l'API Reflection sur une annotation à l'exécution, il est nécessaire que la définition de l'annotation soit faite avec l'annotation @Retention à laquelle la valeur RetentionPolicy.RUNTIME est utilisée en paramètre.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.reflection;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.TYPE, ElementType.PARAMETER, ElementType.FIELD})
public @interface MonAnnotation {
    public String name();
    public String value();
}
```

24.6.1. Les annotations sur une classe

Telle que définie, l'annotation peut s'utiliser sur un type (une classe ou une interface).

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.reflection;

@MonAnnotation(nom="nom1", valeur="valeur1")
public class MaClasse {
}
```

Il est possible d'utiliser l'API Reflection pour accéder dynamiquement aux annotations utilisées sur une classe.

La méthode getAnnotations() de la classe Class permet d'obtenir un tableau de type Annotation qui contient toutes les annotations définies sur la classe.

Exemple :

```
package fr.jmdoudoux.dej.reflection;

import java.lang.annotation.Annotation;

public class TestGetAnnotations {

  public static void main(String args[]) {
    Class classeClass = MaClasse.class;
    Annotation[] annotations = classeClass.getAnnotations();
    for (Annotation annotation : annotations) {
      if (annotation instanceof MonAnnotation) {
        MonAnnotation monAnnotation = (MonAnnotation) annotation;
        System.out.println("nom    : " + monAnnotation.nom());
        System.out.println("valeur : " + monAnnotation.valeur());
      }
    }
  }
}
```

Résultat :

```
non    : nom1
valeur : valeur1
```

La méthode getAnnotation() de la classe Class permet d'obtenir une instance de type Annotation encapsulant l'annotation utilisée sur la classe dont le type correspond à celui passé en paramètre.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.reflection;

import java.lang.annotation.Annotation;

public class TestGetAnnotations {

  public static void main(String args[]) {
    Class classeClass = MaClasse.class;
    Annotation annotation = classeClass.getAnnotation(MonAnnotation.class);
    if (annotation instanceof MonAnnotation) {
      MonAnnotation monAnnotation = (MonAnnotation) annotation;
      System.out.println("nom : " + monAnnotation.nom());
      System.out.println("valeur : " + monAnnotation.valeur());
    }
  }
}
```

24.6.2. Les annotations sur une méthode

Telle que définie, l'annotation peut s'utiliser sur une méthode.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.reflection;

public class MaClasse {

  @MonAnnotation(nom="nom2", valeur="valeur2")
  public void maMethode() {
  }
}
```

Il est possible d'utiliser l'API Reflection pour accéder dynamiquement aux annotations utilisées sur une méthode.

La méthode getDeclaredAnnotations() de la classe Method permet d'obtenir un tableau de type Annotation qui contient toutes les annotations définies sur la méthode.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.reflection;

import java.lang.annotation.Annotation;
import java.lang.reflect.Method;
```

```
public class TestGetAnnotations {

  public static void main(String args[]) {
    Class classeClass = MaClasse.class;
    try {
      Method maMethode = classeClass.getMethod("maMethode");
      Annotation[] annotations = maMethode.getDeclaredAnnotations();
      for (Annotation annotation : annotations) {
        if (annotation instanceof MonAnnotation) {
          MonAnnotation monAnnotation = (MonAnnotation) annotation;
          System.out.println("nom    : " + monAnnotation.nom());
          System.out.println("valeur : " + monAnnotation.valeur());
        }
      }
    } catch (NoSuchMethodException ex) {
      ex.printStackTrace();
    } catch (SecurityException ex) {
      ex.printStackTrace();
    }
  }
}
```

Résultat :

```
nom    : nom2
valeur : valeur2
```

La méthode getAnnotation() de la classe Method permet d'obtenir une instance de type Annotation encapsulant l'annotation utilisée sur la méthode dont le type est passé en paramètre.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.reflection;

import java.lang.annotation.Annotation;
import java.lang.reflect.Method;

public class TestGetAnnotations {

  public static void main(String args[]) {
    Class classeClass = MaClasse.class;
    try {
      Method maMethode = classeClass.getMethod("maMethode");
      Annotation annotation = maMethode.getAnnotation(MonAnnotation.class);
      if (annotation instanceof MonAnnotation) {
        MonAnnotation monAnnotation = (MonAnnotation) annotation;
        System.out.println("non    : " + monAnnotation.nom());
        System.out.println("valeur : " + monAnnotation.valeur());
      }
    } catch (NoSuchMethodException ex) {
      ex.printStackTrace();
    } catch (SecurityException ex) {
      ex.printStackTrace();
    }
  }
}
```

Résultat :

```
non    : nom2
valeur : valeur2
```

24.6.3. Les annotations sur un paramètre d'une méthode

Telle que définie, l'annotation peut s'utiliser sur un paramètre d'une méthode.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.reflection;

public class MaClasse {

  public void maMethode() {
    System.out.println("maMethode sans param");
  }

  public String maMethode(@MonAnnotation(nom="nom3", valeur="valeur3") String param1) {
    System.out.println("maMethode avec String:"+param1);
    return param1;
  }
}
```

Il est possible d'utiliser l'API Reflection pour accéder dynamiquement aux annotations utilisées sur les paramètres d'une méthode.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.reflection;

import java.lang.annotation.Annotation;
import java.lang.reflect.Method;

public class TestGetAnnotations {

  public static void main(String args[]) {
    Class classeClass = MaClasse.class;
    try {
      Method maMethode = classeClass.getMethod("maMethode", String.class);
      Annotation[][] parameterAnnotations = maMethode.getParameterAnnotations();
      Class[] parameterTypes = maMethode.getParameterTypes();
      int i = 0;
      for (Annotation[] annotations : parameterAnnotations) {
        Class parameterType = parameterTypes[i++];
        System.out.println("type du paramètre "+i+" "+parameterType);
        for (Annotation annotation : annotations) {
          if (annotation instanceof MonAnnotation) {
            MonAnnotation monAnnotation = (MonAnnotation) annotation;
            System.out.println("nom    : " + monAnnotation.nom());
            System.out.println("valeur : " + monAnnotation.valeur());
          }
        }
      }
    } catch (NoSuchMethodException ex) {
      ex.printStackTrace();
    } catch (SecurityException ex) {
      ex.printStackTrace();
    }
  }
}
```

Résultat :

```
type du paramètre 1 class java.lang.String
nom    : nom3
valeur : valeur3
```

La méthode getParameterAnnotations() renvoie un tableau à deux dimensions de type Annotation qui contient pour chaque paramètre, les annotations qui lui sont associées.

24.6.4. Les annotations sur un champ

Telle que définie, l'annotation peut s'utiliser sur un champ d'une classe.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.reflection;

public class MaClasse {

  @MonAnnotation(nom="nom4", valeur="valeur4")
  private String monChamp;
}
```

Il est possible d'utiliser l'API Reflection pour accéder dynamiquement aux annotations utilisées sur un champ.

La méthode getDeclaredAnnotations() de la classe Field permet d'obtenir un tableau de type Annotation qui contient toutes les annotations définies sur le champ.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.reflection;

import java.lang.annotation.Annotation;
import java.lang.reflect.Field;
import java.util.logging.Level;
import java.util.logging.Logger;

public class TestGetAnnotations {

  public static void main(String args[]) {
    Class classeClass = MaClasse.class;
    try {
      Field monChamp = classeClass.getDeclaredField("monChamp");
      if (monChamp != null) {
        Annotation[] annotations = monChamp.getDeclaredAnnotations();
        for (Annotation annotation : annotations) {
          if (annotation instanceof MonAnnotation) {
            MonAnnotation monAnnotation = (MonAnnotation) annotation;
            System.out.println("non    : " + monAnnotation.nom());
            System.out.println("valeur : " + monAnnotation.valeur());
          }
        }
      }
    } catch (NoSuchFieldException ex) {
      ex.printStackTrace();
    } catch (SecurityException ex) {
      ex.printStackTrace();
    }
  }
}
```

Résultat :

```
non    : nom4
valeur : valeur4
```

La méthode getAnnotation() de la classe Field permet d'obtenir une instance de type Annotation encapsulant l'annotation utilisée sur le champ dont le type est passé en paramètre.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.reflection;

import java.lang.annotation.Annotation;
import java.lang.reflect.Field;
import java.util.logging.Level;
import java.util.logging.Logger;

public class TestGetAnnotations {

  public static void main(String args[]) {
    Class classeClass = MaClasse.class;
    try {
      Field monChamp = classeClass.getDeclaredField("monChamp");
      if (monChamp != null) {
        Annotation annotation = monChamp.getAnnotation(MonAnnotation.class);
        if (annotation != null && annotation instanceof MonAnnotation) {
          MonAnnotation monAnnotation = (MonAnnotation) annotation;
          System.out.println("nom    : " + monAnnotation.nom());
          System.out.println("valeur : " + monAnnotation.valeur());
        }
      }
    } catch (NoSuchFieldException ex) {
      ex.printStackTrace();
    } catch (SecurityException ex) {
      ex.printStackTrace();
    }
  }
}
```

Résultat :

```
non    : nom4
valeur : valeur4
```

25. L'appel de méthodes distantes : RMI

Chapitre 25

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

RMI (Remote Method Invocation) est une technologie fournie à partir du JDK 1.1 pour permettre de mettre en oeuvre facilement des objets distribués.

Ce chapitre contient plusieurs sections :

- ♦ La présentation et l'architecture de RMI
- ♦ Les différentes étapes pour créer un objet distant et l'appeler avec RMI
- ♦ Le développement coté serveur
- ♦ Le développement coté client
- ♦ La génération de la classe stub
- ♦ La mise en oeuvre des objets RMI

25.1. La présentation et l'architecture de RMI

Le but de RMI est de permettre l'appel, l'exécution et le renvoi du résultat d'une méthode exécutée dans une machine virtuelle différente de celle de l'objet l'appelant. Cette machine virtuelle peut être sur une machine différente pourvu qu'elle soit accessible par le réseau.

La machine sur laquelle s'exécute la méthode distante est appelée serveur.

L'appel coté client d'une telle méthode est un peu plus compliqué que l'appel d'une méthode d'un objet local mais il reste simple. Il consiste à obtenir une référence sur l'objet distant puis à simplement appeler la méthode à partir de cette référence.

La technologie RMI se charge de rendre transparente la localisation de l'objet distant, son appel et le renvoi du résultat.

En fait, elle utilise deux classes particulières, le stub et le skeleton, qui doivent être générées avec l'outil rmic fourni avec le JDK.

Le stub est une classe qui se situe côté client et le skeleton est son homologue coté serveur. Ces deux classes se chargent d'assurer tous les mécanismes d'appel, de communication, d'exécution, de renvoi et de réception du résultat.

25.2. Les différentes étapes pour créer un objet distant et l'appeler avec RMI

Le développement coté serveur se compose de :

- La définition d'une interface qui contient les méthodes qui peuvent être appelées à distance
- L'écriture d'une classe qui implémente cette interface
- L'écriture d'une classe qui instanciera l'objet et l'enregistrera en lui affectant un nom dans le registre de noms RMI (RMI Registry)

Le développement côté client se compose de :

- L'obtention d'une référence sur l'objet distant à partir de son nom
- L'appel à la méthode à partir de cette référence

Enfin, il faut générer les classes stub et skeleton en exécutant le programme rmic avec le fichier source de l'objet distant.

25.3. Le développement coté serveur

Côté serveur, l'objet distant est décrit par une interface. Une instance de l'objet doit être créée et enregistrée dans le registre RMI.

25.3.1. La définition d'une interface qui contient les méthodes de l'objet distant

L'interface à définir doit hériter de l'interface java.rmi.Remote. Cette interface ne contient aucune méthode mais indique simplement que l'interface peut être appelée à distance.

L'interface doit contenir toutes les méthodes qui seront susceptibles d'être appelées à distance.

La communication entre le client et le serveur lors de l'invocation de la méthode distante peut échouer pour diverses raisons telles qu'un crash du serveur, une rupture de la liaison, etc ...

Ainsi chaque méthode appelée à distance doit déclarer qu'elle est en mesure de lever l'exception java.rmi.RemoteException.

Exemple (code Java 1.1) :

```
package fr.jmdoudoux.dej.rmi;

import java.rmi.*;

public interface Information extends Remote {

   public String getInformation() throws RemoteException;

}
```

25.3.2. L'écriture d'une classe qui implémente cette interface

Cette classe correspond à l'objet distant. Elle doit donc implémenter l'interface définie et contenir le code nécessaire.

Cette classe doit obligatoirement hériter de la classe UnicastRemoteObject qui contient les différents traitements élémentaires pour un objet distant dont l'appel par le stub du client est unique. Le stub ne peut obtenir qu'une seule référence sur un objet distant héritant de la classe UnicastRemoteObject. On peut supposer qu'une future version de RMI sera capable de faire du MultiCast, permettant à RMI de choisir parmi plusieurs objets distants identiques la référence à fournir au client.

La hiérarchie de la classe UnicastRemoteObject est :

java.lang.Object

    java.rmi.Server.RemoteObject

        java.rmi.Server.RemoteServer

            java.rmi.Server.UnicastRemoteObject

Comme indiqué dans l'interface, toutes les méthodes distantes, mais aussi le constructeur de la classe, doivent indiquer qu'elles peuvent lever l'exception RemoteException. Ainsi, même si le constructeur ne contient pas de code il doit être redéfini pour inhiber la génération du constructeur par défaut qui ne lève pas cette exception.

Exemple (code Java 1.1) :

```
package fr.jmdoudoux.dej.rmi;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class InformationImpl extends UnicastRemoteObject implements Information {

  private static final long serialVersionUID = 2674880711467464646L;

  protected InformationImpl() throws RemoteException {
    super();
  }

  public String getInformation() throws RemoteException {
    System.out.println("Invocation de la méthode getInformation()");
    return "bonjour";
  }
}
```

25.3.3. L'écriture d'une classe pour instancier l'objet et l'enregistrer dans le registre

Ces opérations peuvent être effectuées dans la méthode main d'une classe dédiée ou dans la méthode main de la classe de l'objet distant. L'intérêt d'une classe dédiée et qu'elle permet de regrouper toutes ces opérations pour un ensemble d'objets distants.

La marche à suivre contient trois étapes :

- la mise en place d'un security manager dédié qui est facultative
- l'instanciation d'un objet de la classe distante
- l'enregistrement de la classe dans le registre de noms RMI

25.3.3.1. La mise en place d'un security manager

Cette opération n'est pas obligatoire mais elle est recommandée en particulier si le serveur doit charger des classes récupérées sur des machines distantes. Sans security manager, il faut obligatoirement mettre à la disposition du serveur toutes les classes dont il aura besoin (Elles doivent être dans le CLASSPATH du serveur). Avec un security manager, le serveur peut charger dynamiquement certaines classes.

Cependant, le chargement dynamique de ces classes peut poser des problèmes de sécurité car le serveur va exécuter du code d'une autre machine. Cet aspect peut conduire à ne pas utiliser de security manager.

Exemple (code Java 1.1) :

```
public static void main(String[] args) {
  try {
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
    }
  } catch (Exception e) {
     e.printStrackTrace();
  }
}
```

Il est aussi possible d'activer un security manager en utilisant simplement l'option -Djava.security.manager de la JVM.

25.3.3.2. L'instanciation d'un objet de la classe distante

Cette opération est très simple puisqu'elle consiste simplement en la création d'un objet de la classe de l'objet distant

Exemple (code Java 1.1) :

```
public static void main(String[] args) {
  try {
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
    }

    InformationImpl informationImpl = new InformationImpl();
  } catch (Exception e) {
     e.printStrackTrace();
  }
}
```

25.3.3.3. L'enregistrement dans le registre de noms RMI

La dernière opération consiste à enregistrer l'objet créé dans le registre de noms en lui affectant un nom. Ce nom est fourni au registre sous forme d'une URL constituée du préfix rmi://, du nom du serveur (hostname) et du nom associé à l'objet précédé d'un slash.

Le nom du serveur peut être fourni « en dur » sous forme d'une constante chaîne de caractères ou peut être dynamiquement obtenu en utilisant la classe InetAddress pour une utilisation en locale.

C'est ce nom qui sera utilisé dans une URL par le client pour obtenir une référence sur l'objet distant.

L'enregistrement se fait en utilisant la méthode rebind de la classe Naming. Elle attend en paramètre l'URL du nom de l'objet et l'objet lui-même.

Exemple (code Java 1.1) :

```
public static void main(String[] args) {
  try {
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
    }

    InformationImpl informationImpl = new InformationImpl();
    String url = "rmi://" + InetAddress.getLocalHost().getHostAddress() + "/TestRMI";
    System.out.println("Enregistrement de l'objet avec l'url : " + url);
    Naming.rebind(url, informationImpl);

    System.out.println("Serveur lancé");
  } catch (RemoteException e) {
    e.printStackTrace();
  } catch (MalformedURLException e) {
    e.printStackTrace();
  } catch (UnknownHostException e) {
    e.printStackTrace();
  }
}
```

25.3.3.4. Le lancement dynamique du registre de noms RMI

Sur le serveur, le registre de noms RMI doit s'exécuter avant de pouvoir enregistrer un objet ou obtenir une référence.

Ce registre peut être lancé en tant qu'application fournie dans le JDK (rmiregistry) comme indiqué dans un chapitre suivant ou être lancé dynamiquement dans la classe qui enregistre l'objet. Ce lancement ne doit avoir lieu qu'une seule et unique fois. Il peut être intéressant d'utiliser le code ci-dessous si l'on crée une classe dédiée à l'enregistrement des objets distants.

Le code pour exécuter le registre est la méthode createRegistry() de la classe java.rmi.registry.LocateRegistry. Cette méthode attend en paramètre un numéro de port.

Exemple (code Java 1.1) :

```
package fr.jmdoudoux.dej.rmi;

import java.net.InetAddress;
import java.net.MalformedURLException;
import java.net.UnknownHostException;
import java.rmi.Naming;
import java.rmi.RMISecurityManager;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;

public class LanceServeur {

  public static void main(String[] args) {
    try {
      LocateRegistry.createRegistry(1099);

      System.out.println("Mise en place du Security Manager ...");
      if (System.getSecurityManager() == null) {
        System.setSecurityManager(new RMISecurityManager());
      }

      InformationImpl informationImpl = new InformationImpl();

      String url = "rmi://" + InetAddress.getLocalHost().getHostAddress() + "/TestRMI";
      System.out.println("Enregistrement de l'objet avec l'url : " + url);
      Naming.rebind(url, informationImpl);

      System.out.println("Serveur lancé");
    } catch (RemoteException e) {
      e.printStackTrace();
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (UnknownHostException e) {
      e.printStackTrace();
    }
  }
}
```

25.4. Le développement coté client

L'appel d'une méthode distante peut se faire dans une application ou dans une applet.

25.4.1. La mise en place d'un security manager

Comme pour le coté serveur, cette opération est facultative.

Le choix de la mise en place d'un sécurity manager côté client suit des règles identiques à celles appliquées côté serveur. Sans son utilisation, il est nécessaire de mettre dans le CLASSPATH du client toutes les classes nécessaires dont la classe stub.

Exemple (code Java 1.1) :

```
public static void main(String[] args) {
  if (System.getSecurityManager() == null) {
    System.setSecurityManager(new RMISecurityManager());
  }
}
```

25.4.2. L'obtention d'une référence sur l'objet distant à partir de son nom

Pour obtenir une référence sur l'objet distant à partir de son nom, il faut utiliser la méthode statique lookup() de la classe Naming.

Cette méthode attend en paramètre une URL indiquant le nom qui référence l'objet distant. Cette URL est composée de plusieurs éléments : le préfix rmi://, le nom du serveur (hostname) et le nom de l'objet tel qu'il a été enregistré dans le registre précédé d'un slash.

Il est préférable de prévoir le nom du serveur sous forme de paramètres de l'application ou de l'applet pour plus de souplesse.

La méthode lookup() va rechercher l'objet dans le registre du serveur et retourner un objet stub. L'objet retourné est de la classe Remote (cette classe est la classe mère de tous les objets distants).

Si le nom fourni dans l'URL n'est pas référencé dans le registre, la méthode lève l'exception NotBoundException.

Exemple (code Java 1.1) :

```
public static void main(String[] args) {

  if (System.getSecurityManager() == null) {
    System.setSecurityManager(new RMISecurityManager());
  }

  try {
    Remote r = Naming.lookup("rmi://10.0.0.13/TestRMI");
  } catch (Exception e) {
    e.printStrackTrace();
  }

}
```

25.4.3. L'appel de la méthode à partir de la référence sur l'objet distant

L'objet retourné étant de type Remote, il faut réaliser un cast vers l'interface qui définit les méthodes de l'objet distant. Pour plus de sécurité, on vérifie que l'objet retourné est bien une instance de cette interface.

Un fois le cast réalisé, il suffit simplement d'appeler la méthode.

Exemple (code Java 1.1) :

```
package fr.jmdoudoux.dej.rmi;

import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RMISecurityManager;
import java.rmi.Remote;
import java.rmi.RemoteException;

public class LanceClient {

  public static void main(String[] args) {
    System.out.println("Lancement du client");
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
    }
    try {
      Remote r = Naming.lookup("rmi://10.0.0.13/TestRMI");
      System.out.println(r);
      if (r instanceof Information) {
        String s = ((Information) r).getInformation();
        System.out.println("chaine renvoyee = " + s);
      }
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (RemoteException e) {
      e.printStackTrace();
    } catch (NotBoundException e) {
      e.printStackTrace();
    }
    System.out.println("Fin du client");
  }
}
```

25.4.4. L'appel d'une méthode distante dans une applet

L'appel d'une méthode distante est le même dans une application et dans une applet.

Seule la mise en place d'un security manager dédié dans les applets est inutile car elles utilisent déjà un sécurity manager (AppletSecurityManager) qui autorise le chargement de classes distantes.

Exemple (code Java 1.1) :

```
package fr.jmdoudoux.dej.rmi;

import java.applet.*;
import java.awt.*;
import java.rmi.*;

public class AppletTestRMI extends Applet {

  private String s;

  public void init() {

    try {
      Remote r = Naming.lookup("rmi://10.0.0.13/TestRMI");

      if (r instanceof Information) {
       s = ((Information) r).getInformation();
      }
    } catch (Exception e) {
      e.printStrackTrace();
    }
  }

  public void paint(Graphics g) {
    super.paint(g);
    g.drawString("chaine retournée = "+s,20,20);
  }
}
```

25.5. La génération de la classe stub

Pour générer la classe stub, il suffit d'utiliser l'outil rmic fourni avec le JDK en lui donnant en paramètre le nom pleinement qualifié de la classe.

Attention la classe doit avoir été compilée : rmic a besoin du fichier .class.

Exemple (code Java 1.1) :

```
rmic fr.jmdoudoux.dej.rmi.InformationImpl
```

rmic va générer et compiler la classe stub sous le nom InformationImpl_Stub.class. Cette classe sera utilisée par la partie cliente pour invoquer l'objet distant correspondant.

25.6. La mise en oeuvre des objets RMI

La mise en oeuvre et l'utilisation d'objets distants avec RMI nécessite plusieurs étapes :

1. Démarrer le registre RMI sur le serveur soit en utilisant le programme rmiregistry livré avec le JDK soit en exécutant une classe qui effectue le lancement.
2. Exécuter la classe qui instancie l'objet distant et l'enregistre dans le serveur de noms RMI
3. Lancer l'application ou l'applet pour tester.

25.6.1. Le lancement du registre RMI

La commande rmiregistry est fournie avec le JDK. Il faut la lancer en tâche de fond :

Sous Unix : rmiregistry&

Sous Windows : start rmiregistry

Ce registre permet de faire correspondre un objet à un nom et inversement. C'est lui qui est sollicité lors d'un appel aux méthodes Naming.bind() et Naming.lookup()

25.6.2. L'instanciation et l'enregistrement de l'objet distant

Il faut exécuter la classe qui va instancier l'objet distant et l'enregistrer sous son nom dans le registre précédemment lancé.

Pour ne pas avoir de problème, il faut s'assurer que toutes les classes utiles (la classe de l'objet distant, l'interface qui définit les méthodes) sont présentes dans un répertoire défini dans le classpath.

Si un gestionnaire de sécurité est mis en place, il faut définir un fichier qui va contenir la politique de sécurité qu'il doit mettre en oeuvre.

Exemple (code Java 1.1) : le fichier ma_policy_serveur

```
grant{
permission java.net.SocketPermission "localhost:1099", "connect, resolve";
permission java.net.SocketPermission "*:1024-", "connect, resolve";
permission java.net.SocketPermission "*:1024-", "accept, resolve";
};
```

Les permissions définies concernent les permissions de connexions par socket au serveur.

Lors du lancement du serveur, l'option java.security.policy permet de préciser le fichier qui sera utilisé par le gestionnaire de sécurité.

Exemple (code Java 1.1) : le fichier ma_policy_serveur

```
C:\Users\Jean Michel\workspace\TestRmiServer>java -cp bin -Djava.security.policy
=ma_policy_serveur fr.jmdoudoux.dej.rmi.LanceServeur
Mise en place du Security Manager ...
Enregistrement de l'objet avec l'url : rmi://10.0.0.13/TestRMI
Serveur lancé
```

25.6.3. Le lancement de l'application cliente

L'archive de la partie cliente doit contenir le client, l'interface de l'objet distant et le stub qui a été généré par rmic.

Exemple :

```
C:\temp>jar -tf TestRMIClient.jar
META-INF/MANIFEST.MF
com/jmdoudoux/test/rmi/Information.class
com/jmdoudoux/test/rmi/InformationImpl_Stub.class
com/jmdoudoux/test/rmi/LanceClient.class
```

Le client qui invoque l'objet distant est lancé de manière classique.

Exemple :

```
C:\temp>java -jar TestRMIClient.jar
Lancement du client
InformationImpl_Stub[UnicastRef [liveRef: [endpoint:[10.0.0.13:62802](remote),ob
jID:[7b7739e4:135b4a87a5e:-7fff, -3323459310870193038]]]]
chaine renvoyee = bonjour
Fin du client
```

Si le serveur n'est pas démarré, une exception est levée

Exemple :

```
C:\temp>java -jar TestRMIClient.jar
Lancement du client
java.rmi.ConnectException: Connection refused to host: 10.0.0.13; nested excepti
on is:
        java.net.ConnectException: Connection timed out: connect
        at sun.rmi.transport.tcp.TCPEndpoint.newSocket(Unknown Source)
        at sun.rmi.transport.tcp.TCPChannel.createConnection(Unknown Source)
        at sun.rmi.transport.tcp.TCPChannel.newConnection(Unknown Source)
        at sun.rmi.server.UnicastRef.newCall(Unknown Source)
        at sun.rmi.registry.RegistryImpl_Stub.lookup(Unknown Source)
        at java.rmi.Naming.lookup(Unknown Source)
        at fr.jmdoudoux.dej.rmi.LanceClient.main(LanceClient.java:17)
Caused by: java.net.ConnectException: Connection timed out: connect
        at java.net.TwoStacksPlainSocketImpl.socketConnect(Native Method)
        at java.net.AbstractPlainSocketImpl.doConnect(Unknown Source)
        at java.net.AbstractPlainSocketImpl.connectToAddress(Unknown Source)
        at java.net.AbstractPlainSocketImpl.connect(Unknown Source)
        at java.net.PlainSocketImpl.connect(Unknown Source)
        at java.net.SocksSocketImpl.connect(Unknown Source)
        at java.net.Socket.connect(Unknown Source)
        at java.net.Socket.connect(Unknown Source)
        at java.net.Socket.<init>(Unknown Source)
        at java.net.Socket.<init>(Unknown Source)
        at sun.rmi.transport.proxy.RMIDirectSocketFactory.createSocket(Unknown S
ource)
        at sun.rmi.transport.proxy.RMIMasterSocketFactory.createSocket(Unknown S
ource)
        ... 7 more
Fin du client
```

La partie client peut être lancée avec un gestionnaire et une politique de sécurité associée.

Exemple (code Java 1.1) :

```
C:\temp>java -jar -Djava.security.policy=ma_policy_client TestRMIClient.jar
Lancement du client
InformationImpl_Stub[UnicastRef [liveRef: [endpoint:[10.0.0.13:62802](remote),ob
jID:[7b7739e4:135b4a87a5e:-7fff, -3323459310870193038]]]]
chaine renvoyee = bonjour
Fin du client
```

Si le gestionnaire est activé sans politique de sécurité associée alors la connexion au serveur est impossible.

Exemple (code Java 1.1) :

```
C:\temp>java -jar -Djava.security.manager TestRMIClient.jar
Lancement du client
Exception in thread "main" java.security.AccessControlException: access denied (
"java.net.SocketPermission" "10.0.0.13:1099" "connect,resolve")
        at java.security.AccessControlContext.checkPermission(Unknown Source)
        at java.security.AccessController.checkPermission(Unknown Source)
        at java.lang.SecurityManager.checkPermission(Unknown Source)
        at java.lang.SecurityManager.checkConnect(Unknown Source)
        at java.net.Socket.connect(Unknown Source)
        at java.net.Socket.connect(Unknown Source)
        at java.net.Socket.<init>(Unknown Source)
        at java.net.Socket.<init>(Unknown Source)
        at sun.rmi.transport.proxy.RMIDirectSocketFactory.createSocket(Unknown Source)
        at sun.rmi.transport.proxy.RMIMasterSocketFactory.createSocket(Unknown Source)
        at sun.rmi.transport.tcp.TCPEndpoint.newSocket(Unknown Source)
        at sun.rmi.transport.tcp.TCPChannel.createConnection(Unknown Source)
        at sun.rmi.transport.tcp.TCPChannel.newConnection(Unknown Source)
        at sun.rmi.server.UnicastRef.newCall(Unknown Source)
        at sun.rmi.registry.RegistryImpl_Stub.lookup(Unknown Source)
        at java.rmi.Naming.lookup(Unknown Source)
        at fr.jmdoudoux.dej.rmi.LanceClient.main(LanceClient.java:17)
```

26. La sécurité

Chapitre 26

Niveau : 🟩🟩🟨🟨🟥⬜ Confirmé

Depuis sa conception, la sécurité dans le langage Java a toujours été une grande préoccupation pour Sun.

Avec Java, la sécurité revêt de nombreux aspects :

- les spécifications du langage disposent de fonctionnalités pour renforcer la sécurité du code
- la plate-forme définit un modèle pour gérer les droits d'une application
- les API JCA/JCE permettent d'utiliser des technologies de cryptographie
- l'API JSSE permet d'utiliser le réseau au travers des protocoles sécurisés SSL ou TLS
- l'API JAAS propose un service pour gérer l'authentification et les autorisations d'un utilisateur

Ces deux premiers aspects ont été intégrés à Java dès sa première version.

Ce chapitre contient plusieurs sections :

- ♦ La sécurité dans les spécifications du langage
- ♦ Le contrôle des droits d'une application
- ♦ La cryptographie
- ♦ JCA (Java Cryptography Architecture) et JCE (Java Cryptography Extension)
- ♦ JSSE (Java Secure Sockets Extension)
- ♦ JAAS (Java Authentication and Authorization Service)

Ce chapitre est très incomplet : la suite de ce chapitre sera développée dans une version future de ce document

26.1. La sécurité dans les spécifications du langage

Les spécifications du langage apportent de nombreuses fonctionnalités pour renforcer la sécurité du code aussi bien lors de la phase de compilation que lors de la phase d'exécution :

- typage fort (toutes les variables doivent posséder un type)
- initialisation des variables d'instances avec des valeurs par défaut
- modificateur d'accès pour gérer l'encapsulation et donc l'accessibilité aux membres d'un objet
- les membres final
- ...

26.1.1. Les contrôles lors de la compilation

26.1.2. Les contrôles lors de l'exécution

La JVM exécute un certain nombre de contrôles au moment de l'exécution :

- vérification des accès en dehors des limites des tableaux
- contrôle de l'utilisation des casts
- vérification par le classloader de l'intégrité des classes utilisées
- ...

26.2. Le contrôle des droits d'une application

Un système de contrôle des droits des applications a été intégré à Java dès sa première version notamment pour permettre de sécuriser l'exécution des applets. Ces applications téléchargées sur le réseau et exécutées sur le poste client doivent impérativement assurer aux personnes qui les utilisent qu'elles ne risquent pas de réaliser des actions malveillantes sur le système dans lequel elles s'exécutent.

Le modèle de sécurité relatif aux droits des applications développées en Java a évolué au fur et à mesure des différentes versions de Java.

26.2.1. Le modèle de sécurité de Java 1.0

Le modèle proposé par Java 1.0 était très sommaire puisqu'il ne distinguait que deux catégories d'applications :

- les applications locales
- les applications téléchargées sur le réseau

Le modèle est basé sur le "tout ou rien". Les applications locales ont tous les droits et les applications téléchargées ont des droits très limités. Les restrictions de ces dernières sont nombreuses :

- impossibilité d'écrire sur le disque local
- impossibilité d'obtenir des informations sur le système local
- impossibilité de se connecter à un autre serveur que celui d'où l'application a été téléchargée
- ...

La mise en oeuvre de ce modèle est assurée par le "bac à sable" (sandbox en anglais) dans lequel s'exécutent les applications téléchargées.

26.2.2. Le modèle de sécurité de Java 1.1

Le modèle proposé par la version 1.0 était très efficace mais beaucoup trop restrictif surtout dans le cadre d'une utilisation personnelle telle que celle des applications pour un intranet par exemple.

Le modèle de la version 1.1 propose la possibilité de signer les applications packagées dans un fichier .jar. Une application ainsi signée possède les mêmes droits qu'une application locale.

26.2.3. Le modèle Java 1.2

Le modèle proposé par la version 1.1 a apporté des débuts de solution pour attribuer des droits à certaines applications. Mais ce modèle manque cruellement de souplesse puisqu'il s'appuie toujours sur le modèle "tout au rien".

Le modèle de la version 1.2 apporte enfin une solution très souple mais plus compliquée à mettre en oeuvre.

Les droits accordés à une application sont rassemblés dans un fichier externe au code qui se nomme politique de sécurité. Pour les différentes applications, l'ensemble des fichiers se situe dans le répertoire lib/security du répertoire où est installé le JRE. Par convention, ces fichiers ont pour extension .policy.

26.3. La cryptographie

Le mot cryptographie est dérivé des mots grecs kryptos (caché) et graphie (écriture). La cryptologie est la science qui étudie la transformation d'un texte en clair (plain text) en un texte chiffré difficile à comprendre (ciphered text).

La cryptographie permet de stocker ou d'échanger des données de façon plus ou moins sécurisée.

La cryptographie utilise deux opérations :

- le chiffrement : cette opération consiste à transformer des données en clair en des données chiffrées qui soient difficiles voire impossible à exploiter par un tiers ne pouvant effectuer l'opération de déchiffrement
- le déchiffrement : c'est l'opération inverse qui permet le retour des données en clair à partir de données chiffrées

La cryptographie est l'art de sécuriser un ensemble de données : elle est principalement utilisée pour créer une valeur de hachage d'un message ou pour chiffrer/déchiffrer un message. Dans les deux cas, des algorithmes mathématiques complexes voire très complexes sont utilisés.

La cryptographie est un élément très important de la sécurité des échanges notamment au travers des réseaux : elle est par exemple utilisée pour mettre en oeuvre les signatures digitales, les certificats, l'authentification de messages, le chiffrement/déchiffrement des messages, ...

La cryptographie est utilisée pour mettre en oeuvre plusieurs fonctionnalités notamment :

- confidentialité : les données peuvent être chiffrées pour garantir leur caractère privé
- intégrité des données : un algorithme mathématique permet de calculer une valeur de hachage et ainsi de vérifier l'intégrité d'un message
- authentification : les signatures numériques permettent de garantir que les données proviennent du bon émetteur

Une clé est un ensemble de données utilisée lors du chiffrement ou du déchiffrement de données afin de rendre ces opérations plus résistantes.

La cryptographie est assez ancienne : elle est utilisée depuis très longtemps avec des algorithmes initialement basiques. Par exemple, à l'époque gallo-romaine, l'empereur Jules César utilisait un système de chiffrement symétrique reposant sur un décalage alphabétique : ces principes de substitutions étaient utilisés pour crypter des messages.

Texte en clair	B	O	N	J	O	U	R
Décalage	+3	+3	+3	+3	+3	+3	+3
Texte chiffré	E	R	Q	M	R	X	U

Cette substitution est assez facile à casser, notamment par ce que chaque lettre en clair correspond à une même lettre chiffrée.

Il existe des versions plus complexes qui font varier la valeur de décalage pour chaque lettre.

Texte en clair	B	O	N	J	O	U	R

Décalage	+1	+2	+3	+1	+2	+3	+1
Texte chiffré	C	R	Q	K	Q	X	S

Dans l'exemple ci-dessus, la lettre O peut être cryptée en R ou Q selon sa position dans le texte.

Les algorithmes de substitution sont facilement cassables avec des machines.

Il est possible d'utiliser une clé de substitution qui associe à chaque caractère un autre caractère. L'inconvénient est que la clé doit être connue lors du chiffrement et du déchiffrement, ce qui nécessite d'une manière ou d'une autre de réaliser un échange de cette clé.

Caractère	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Clé	L	O	G	P	R	T	C	M	A	Q	S	U	Z	H	D	I	W	B	V	J	Y	N	E	K	X	F

En utilisant la clé, il est possible de chiffrer le message.

Texte en clair	B	O	N	J	O	U	R
Texte chiffré	O	D	H	Q	D	Y	B

Plusieurs types de modèles sont utilisés en cryptographie :

- Symétrique : utilise une clé unique qui est utilisée par l'émetteur et le récepteur. Ils doivent donc la partager entre-eux
- Asymétrique : utilise une clé publique et une clé privée
- Hybride : il utilise les modèles cryptographiques symétrique et asymétrique en combinant leurs avantages

Il existe deux grandes familles d'algorithmes de chiffrement :

- le chiffrement symétrique : une seule clé secrète est utilisée pour le chiffrement et le déchiffrement. L'émetteur et le récepteur doivent disposer de la même clé pour réaliser le chiffrement et le déchiffrement. La difficulté est de conserver secrète la clé qui doit être échangée.
- le chiffrement asymétrique : une clé publique et une clé privée sont utilisées pour le chiffrement et le déchiffrement. Les données sont chiffrées avec une clé et déchiffrées avec l'autre.

Le chiffrement symétrique utilise la même clé pour chiffrer et déchiffrer les données.

Ce procédé présente plusieurs inconvénients :

- chaque émetteur doit utiliser une clé différente avec chacun de ses correspondants
- l'échange de la clé doit se faire de manière sécurisée
- toute personne qui possède la clé de l'émetteur peut se faire passer pour lui : la non répudiation n'est pas garantie

Le chiffrement asymétrique utilise deux clés distinctes qui sont liées entre elles : la clé utilisée pour chiffrer peut être utilisée pour déchiffrer ce qui a été chiffré avec l'autre clé et vice versa. Une des deux clés conservée par l'émetteur est nommée clé privée, l'autre clé diffusée est nommée clé publique.

	Avantages	Inconvénients
Chiffrement symétrique	Rapide	Echange de la clé Ne peut pas être utilisé pour les signatures électroniques
Chiffrement asymétrique	Utilise deux clés	Lent

	Utilisable pour les signatures électroniques	

Avec un chiffrement asymétrique, les deux clés sont liées mais il n'est pas possible de retrouver une clé à partir de l'autre.

Les algorithmes de chiffrements asymétriques sont généralement plus lents que les symétriques et nécessitent de nombreuses ressources lors de leurs calculs.

La cryptanalyse est la science qui étudie comment casser un algorithme de chiffrement : son but est donc de décrypter les données encodées avec un algorithme sans avoir les informations utiles pour réaliser l'opération. Pour cela, elle emploie de nombreuses techniques notamment la force brute, la factorisation, la cryptanalyse linéaire, la cryptanalyse différentielle, ...

Certains algorithmes ne possédant pourtant pas de failles connues ne sont plus utilisés car la puissance de calcul des machines actuelles permet de trouver leur clé (exemple : l'algorithme DES)

Il ne faut considérer aucun algorithme comme fiable et ils sont tous potentiellement vulnérables, si ce n'est maintenant, ce sera plus tard. La vulnérabilité d'un algorithme dépend du temps et de la puissance du hardware. Par exemple, les algorithmes DES et MD5 fréquemment utilisés sont maintenant considérés comme non sûrs et ne doivent donc plus être utilisés pour des besoins critiques.

26.3.1. Le modèle symétrique

Avec les algorithmes à clés symétriques, la même clé est utilisée pour les opérations de chiffrement et de déchiffrement. Ce type d'algorithme porte aussi le nom d'algorithme à clé secrète. Ce type de chiffrement est historiquement le plus ancien puisqu'il était déjà utilisé par les égyptiens et les romains plusieurs milliers d'années avant J.C.

La taille de la clé influe sur la robustesse de l'algorithme.

L'émetteur et le récepteur doivent partager la même clé. La clé ne doit être connue que de l'émetteur et du récepteur. Il faut donc garantir que l'échange de cette clé se soit fait de manière sécurisée.

Son principal défaut concerne donc l'échange de la clé entre l'émetteur et le récepteur. Hormis par un moyen physique, il n'est pas possible d'échanger électroniquement la clé de façon sécurisée sans la chiffrer.

Chaque paire d'acteurs (émetteur/receveur) doit avoir une clé différente sinon tous les acteurs qui partagent la même clé seront capables de déchiffrer les données. Pour des échanges sécurisés avec plusieurs acteurs indépendants, il faut une clé distincte ce qui peut rendre le nombre de clés à gérer important.

Son grand intérêt est d'être rapide lors de l'exécution des opérations car les algorithmes mathématiques sont relativement simples.

L'algorithme peut s'appliquer selon deux approches :

- Bloc (Block cipher)
- Flux (Stream cipher)

Il existe de nombreux algorithmes de chiffrements symétriques :

Algorithme	Description
DES	DES est l'acronyme de Data Encryption Standard. Cet algorithme a été inventé en 1977 par IBM et adopté en 1978 par le NBS (National Bureau of Standards). Il a été utilisé par les administrations fédérales américaines pour chiffrer les données sensibles. Les données sont chiffrées par blocs de 64 bits (block cipher). Il utilise une clé sur 56 bits ce qui le rend vulnérable aux attaques par force brute. Cet algorithme n'est donc plus considéré comme sûr.

Triple DES (3DES)	Pour palier la faiblesse de DES, l'algorithme Triple DES applique trois fois l'algorithme DES avec deux ou trois clés différentes. Ceci rend cet algorithme plus résistant aux tentatives de déchiffrement par force brute.
Blowfish	Blowfish utilise une clé de longueur variable de 32 à 448 bits et est plus rapide et plus sûr que DES (block cipher)
AES / Rijndael	(Advanced Encryption Standard) : 128/192/256 bits, plus rapide et plus sûr que le Triple DES, finalisé en 2000 (block cipher) AES est l'acronyme de Advanced Encryption Standard. Cet algorithme a été inventé en 2000 par Vincent Rijmen et Joan Daemen pour remplacer le DES. Il est utilisé par les administrations américaines pour chiffrer les données sensibles. AES peut utiliser plusieurs longueurs de clés notamment 128 bits, 192 bits et 256 bits selon le degré de sécurité souhaité.
IDEA	(International Data Encryption Algorithm) : 128 bits IDEA est l'acronyme d'International Data Encryption Algorithm. Cet algorithme a été inventé en 1991 par J. Massey et X. Lai. Son principe de fonctionnement est similaire à DES mais il utilise une clé plus longue sur 128 bits.
RC2, RC4, RC5, RC6 (Rivets Cipher 4)	128bits, taille variable (stream cipher) RC2, RC4 et RC5 sont des algorithmes créés par Ronald Rivest. RC est l'acronyme de "Ron's Code" ou "Rivest's Cipher". Ces algorithmes permettent de choisir la longueur de la clé (jusqu'à 1024 bits).

Le modèle symétrique possède plusieurs limitations :

- Le partage de la clé peut être une vulnérabilité
- La gestion de la clé est problématique : diffusion, révocation, multiplication des clés requises, ...

26.3.2. Le modèle asymétrique

Ce modèle repose sur des algorithmes mathématiques complexes qui utilisent deux clés distinctes, une dite publique et une autre dite privée :

- une clé publique : cette clé peut être diffusée à tout le monde
- une clé privée : cette clé ne doit être connue que du destinataire

Un grand nombre aléatoire est utilisé pour générer les deux clés. Les deux clés utilisées sont différentes et il est impossible de déduire une clé à partir de l'autre. Par contre, l'usage des deux clés est réversible : l'une peut être indifféremment la clé publique ou la clé privée et vice versa.

Chaque clé joue un rôle particulier : une clé est utilisée pour chiffrer les données et l'autre clé est pour déchiffrer.

La clé publique est utilisée pour chiffrer les données qui ne seront alors déchiffrable qu'avec la clé privée. Inversement, un message chiffré avec la clé privée ne pourra être déchiffré qu'avec la clé publique. La clé publique peut être diffusée à tout le monde mais la clé privée doit être gardée secrète.

L'utilisation de ce modèle nécessite plusieurs étapes :

1. La génération des deux clés
2. L'envoie à l'expéditeur de l'une des deux clés comme clé publique
3. L'expéditeur chiffre les données avec la clé publique et l'envoie au destinataire

4. Le destinataire déchiffre les données avec la clé privée

A et B possèdent chacun leur clé privée. Ils se sont échangé leurs clés publiques respectives.

A utilise sa clé privée pour envoyer un message chiffré à B qui utilise la clé publique correspondante pour déchiffrer le message.

L'échange de clés publiques est facile puisque cette clé est inutilisable sans la clé privée correspondante qui elle n'est pas échangée.

Ce type d'algorithme porte aussi le nom d'algorithme à clé publique. Le développement de ces algorithmes est assez récent puisqu'ils ont débuté dans les années 1970. Ils permettent d'utiliser la cryptographie pour assurer des fonctionnalités de type confidentialité et authentification.

Il existe plusieurs algorithmes qui mettent en oeuvre ce modèle dont :

Algorithme	Description
RSA	RSA est l'acronyme du nom de ses trois inventeurs (Ronald Rivest, Adi Shamir et Leonard Adleman). Le brevet relatif à RSA a expiré en septembre 2000. jusqu'à 2048 bits, publié en 1978 La clé publique est le résultat de la multiplication de deux très grands nombres premiers. La clé privée dépend de ces deux valeurs mais il est extrêmement difficile de les recalculer à partir de la clé privée.
DSA	DSA est l'acronyme de Digital Signature Algorithm
Diffie-Hellman	Diffie-Hellman est un algorithme de chiffrement asymétrique créé par Whitfeld Diffie et Martin Hellman. L'algorithme a fait l'objet d'un dépôt de brevet en 1977 qui a expiré en 1997. C'est un protocole pour l'échange de clés qui est vulnérable
ElGamal	L'algorithme ElGamal est une variante de Diffie-Hellman inventée par Taher Elgamal.
Elliptic Curve Cryptography (ECC)	La vitesse de l'AES est utilisée avec les clés RSA : le chiffrage est efficace car plus rapide avec une taille de clé réduite Améliore grandement la vitesse de traitement par rapport à l'encryptage standard avec des clés publiques Plusieurs implémentations open source (BouncyCastle, OpenSSL, ...) Implémentations de plusieurs algorithmes dans Java 7

Ce type de modèle est utilisé dans la signature digitale :

- l'émetteur chiffre son message avec sa clé privée
- le receveur déchiffre le message chiffré avec sa clé publique

Ceci permet d'authentifier l'émetteur car c'est le seul qui possède la clé correspondant à la clé publique mais ne permet pas la confidentialité des données car la clé publique est diffusée.

Les algorithmes pour le chiffrement symétrique présentent plusieurs inconvénients :

- ils reposent sur des algorithmes mathématiques complexes qui nécessitent donc des ressources notamment en CPU lors des opérations de chiffrement/déchiffrement.
- la clé doit être échangée : durant cet échange, la clé peut être obtenue par un tiers qui pourra alors déchiffrer les messages
- il faut utiliser une clé différente entre un émetteur et ses différents récepteurs. Le nombre de clé à gérer peut alors devenir important

Le chiffrement asymétrique ne garantit pas l'intégrité du message reçu. A chiffre un message avec sa clé publique et envoie le message chiffré à B. C intercepte le message et le remplace par un autre message chiffré avec la même clé publique que A. B reçoit le message et peut le déchiffrer avec la clé privée.

Pour garantir l'intégrité du message, il est nécessaire de le signer. La signature électronique utilise la clé privée de A pour générer une valeur de hachage du message qui sera alors envoyée chiffrée.

Il est aussi possible de signer un message sans le chiffrer.

26.4. JCA (Java Cryptography Architecture) et JCE (Java Cryptography Extension)

Deux API fournies depuis Java 1.4 permettent la mise en oeuvre de la cryptographie :

- JCA (Java Cryptography Achitecture) qui définit l'architecture générale du framework et les fonctionnalités cryptographiques de base (fonctions de hachage, signatures numériques, clés, certificats, ...)
- JCE (Java Cryptography Extension) qui fournit des fonctionnalités cryptographiques de haut niveau (chiffrement/déchiffrement avec algorithmes symétriques/asymétriques, authentification de messages (HMAC), ...)

Historiquement, seul l'API JCA était fournie dans le JDK car le JCE était soumis à des restrictions liées à la législation américaine. Depuis l'assouplissement de ces restrictions, les deux API sont fournies dans le JDK.

Historiquement, les API de cryptographie de Java étaient séparées en deux parties afin de permettre une restriction de diffusion :

- le package java.security contient des classes qui ne possèdent pas de restriction de diffusion
- le package javax.crypto contient des classes qui ont été pendant un moment diffusées sous restriction

JCE est une API qui propose de standardiser l'utilisation de la cryptographie en restant indépendant des algorithmes utilisés. Elle prend en compte le cryptage/décryptage de données, la génération de clés et l'utilisation de la technologie MAC (Message Authentication Code) pour garantir l'intégrité d'un message.

JCE a été intégrée au JDK 1.4. Auparavant, cette API était disponible en tant qu'extension pour les JDK 1.2 et 1.3.

Pour pouvoir utiliser cette API, il faut obligatoirement utiliser une implémentation développée par un fournisseur (provider). Avec le JDK 1.4, Sun fournit une implémentation de référence nommée SunJCE.

L'API JCA est contenue dans le package java.security et l'API JCE est contenue dans le package javax.crypto.

JCA	JCE
java.security	javax.crypto
java.security.acl	javax.crypto.interfaces
java.security.cert	javax.crypto.spec
java.security.interfaces	
java.security.spec	

Ces API fournissent différentes fonctionnalités nommées services. Ces services peuvent offrir la même typologie de fonctionnalités mais leurs implémentations utilisent des algorithmes différents.
L'API utilise des fabriques pour permettre d'obtenir une instance de ces services pour un algorithme et éventuellement un fournisseur donné.

La cryptographie est généralement limitée par les législations de nombreux pays. De ce fait, les algorithmes fournis par Java sont répartis en deux groupes :

- strong : inclus dans tous les JRE
- limited : téléchargeables séparement selon les restrictions des lois américaines

La configuration utilisable dans le monde entier est fournie avec les JRE dans le fichier default_local.policy contenu dans le fichier lib/security/local_policy.jar

Exemple :

```
// Some countries have import limits on crypto strength. This policy file
// is worldwide importable.

grant {
    permission javax.crypto.CryptoPermission "DES", 64;
    permission javax.crypto.CryptoPermission "DESede", *;
    permission javax.crypto.CryptoPermission "RC2", 128,
          "javax.crypto.spec.RC2ParameterSpec", 128;
    permission javax.crypto.CryptoPermission "RC4", 128;
    permission javax.crypto.CryptoPermission "RC5", 128,
          "javax.crypto.spec.RC5ParameterSpec", *, 12, *;
    permission javax.crypto.CryptoPermission "RSA", *;
    permission javax.crypto.CryptoPermission *, 128;
};
```

Le chapitre «JCA (Java Cryptography Architecture)» détaille l'utilisation de cette API.

Le chapitre «JCE (Java Cryptography Extension)» détaille l'utilisation de cette API.

26.5. JSSE (Java Secure Sockets Extension)

JSSE permet de mettre en oeuvre TLS (Transport Layer Security). TLS est le successeur de SSL (Secure Sockets Layer). SSL et TLS permettent de sécuriser les échanges sur le réseau en utilisant la cryptographie asymétique. SSL et TLS sont indépendants du protocole utilisé.

Plusieurs versions de TLS ont été publiées :

- v 1.0 définie dans la RFC 2246 en 1999
- v 1.1 définie dans la RFC en 2006
- v1.2 définie dans la RFC 5246 en 2008

Par abus de langage, SSL est souvent utilisé comme synonyme de TLS.

Les classes et interfaces de cette API sont regroupées dans les packages javax.net et javax.net.ssl.

26.6. JAAS (Java Authentication and Authorization Service)

Les classes et interfaces de cette API sont regroupées dans le package javax.security.auth

Cette API a été intégrée au JDK 1.4.

27. JCA (Java Cryptography Architecture)

Chapitre 27

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Le but de l'API JCA (Java Cryptography Architecture) est de fournir des fonctionnalités cryptographiques de base à la plate-forme Java.

JCA a été conçu pour suivre plusieurs objectifs :

- laisser l'implémentation à des fournisseurs
- être extensible : différentes implémentations de fonctions cryptographiques utilisant différents algorithmes peuvent être proposées par différents fournisseurs
- assurer l'indépendance et garantir l'interopérabilité entre les différentes implémentations

L'API JCA propose d'utiliser des services cryptographiques sans se préoccuper de l'implémentation des algorithmes.

JCA est un framework qui permet d'utiliser des fonctionnalités de cryptographie comprenant plusieurs services :

- algorithmes pour Message digest
- algorithmes pour signature digitale
- cryptographie symétrique par lot ou par flux
- cryptographie asymétrique
- Password-based encryption (PBE)
- Elliptic Curve Cryptography (ECC)
- Algorithmes pour Key agreement
- Générateur de clés (Key generator)
- Message Authentication Codes (MAC)
- Générateur de nombres pseudo-aléatoires
- Stockage et gestion de clés et certificats

L'API JCA est incluse à partir de Java 2.

Plusieurs exemples de cette section utilisent une méthode statique pour convertir un tableau d'octets en hexadecimal.

Exemple :

```
package fr.jmdoudoux.dej.securite;

public class ConvertionHelper {

  public static String bytesToHex(byte[] b) {
    char hexDigit[] = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A',
      'B', 'C', 'D', 'E', 'F' };
    StringBuffer buf = new StringBuffer();
    for (int j = 0; j < b.length; j++) {
      buf.append(hexDigit[(b[j] >> 4) & 0x0f]);
      buf.append(hexDigit[b[j] & 0x0f]);
    }
    return buf.toString();
  }
}
```

Ce chapitre contient plusieurs sections :

- L'architecture de JCA
- Les classes et interfaces de JCA
- Les fournisseurs d'implémentations
- La classe java.security.Provider
- La classe java.security.Security
- La classe java.security.MessageDigest
- Les classes DigestInputStream et DigestOuputStream
- La classe java.security.Signature
- La classe java.security.KeyStore
- Les interfaces de type java.security.Key
- La classe java.security.KeyPair
- La classe java.security.KeyPairGenerator
- La classe java.security.KeyFactory
- La classe java.security.SecureRandom
- La classe java.security.AlgorithmParameters
- La classe java.security.AlgorithmParameterGenerator
- La classe java.security.cert.CertificateFactory
- L'interface java.security.spec.KeySpec et ses implémentations
- La classe java.security.spec.EncodedKeySpec et ses sous-classes
- L'interface java.security.spec.AlgorithmParameterSpec

27.1. L'architecture de JCA

JCA propose une architecture offrant l'interopérabilité qui permet d'être indépendant des implémentations des algorithmes et d'être extensible en utilisant la notion de services.

Chaque type de service est encapsulé de manière abstraite dans une classe de type cryptographic engine qui permet :

- d'obtenir une instance pour une implémentation particulière
- de proposer des fonctionnalités spécifiques au service

Les implémentations respectent une interface de type SPI (Service Provider Interface). A chaque classe de type cryptographic engine correspond une classe abstraite qui définit les méthodes SPI. Les fournisseurs doivent hériter de ces classes pour implémenter leurs services.

Chaque classe de type SPI possède le même nom que la classe en se terminant par spi. Chaque instance d'une classe de l'API possède une instance de sa classe de type spi correspondante. Toutes les méthodes sont final et invoquent la méthode correspondante de l'instance de type SPI.

Toutes les classes SPI sont abstraites : chaque implémentation d'un service doit fournir une sous-classe pour chaque algorithme.

27.2. Les classes et interfaces de JCA

Les classes et interfaces de L'API JCA sont contenues dans le package java.security et ses sous-packages.

JCA contient des engine classes qui proposent une abstraction de certains services de cryptographie sans implémentation concrète. Chaque service permet l'obtention d'ne implémentation pour un type de fonctionnalité et son utilisation. L'API JCA propose plusieurs classes pour ses services dont les principales sont :

Classe	Rôle
AlgorithmParameterGenerator	Créer des classes qui encapsulent les paramètres pour certains algorithmes
AlgorithmParameters	Encapsuler les paramètres d'une fonction cryptographique de manière opaque

AllPermission	Une permission qui regroupe toutes les autres
DigestInputStream	Calculer une valeur de hachage pour des octets lus dans un flux
DigestOutputStream	Calculer une valeur de hachage pour des octets écrits dans un flux
GuardedObject	Contrôler l'accès à d'autres objets
Identity	Deprecated
IdentityScope	Deprecated
KeyFactory	Convertir des clés transparentes en clés opaques et vice versa.
KeyPair	Encapsuler une paire de clés publique et privée
KeyPairGenerator	Générer une paire de clés publiques/privées utilisable pour un algorithme
KeyStore	Stocker, gérer et récupérer les éléments contenus dans un dépôt de clés et de certificats.
MessageDigest	Obtenir et utiliser une implémentation d'un service de type message digest permettant de calculer une valeur de hachage pour un ensemble de données
Permission	La classe abstraite qui encapsule un accès à une ressource
Provider	Servir de point d'entrée pour une implémentation de services cryptographiques proposée par un fournisseur
SecureRandom	Générer des nombres pseudo-aléatoires forts requis pour les besoins de la cryptographie
Security	Gérer les fournisseurs de services en enregistrant ou en retirant une implémentation
Signature	Obtenir et utiliser une implémentation d'un service de type signature digitale permettant de créer une signature pour des données et vérifier une signature numérique

Chaque engine class possède une méthode statique getInstance() qui est une fabrique qui permet d'obtenir une instance particulière du service dont l'algorithme est fourni en paramètre. La fabrique recherche l'instance pour le fournisseur demandé ou recherche une implémentation parmi celles enregistrées pour les différents fournisseurs.

L'API définit plusieurs interfaces dont les principales sont :

Interface	Rôle
Key	Décrire les fonctionnalités communes à toutes les clés opaques
PrivateKey	Une clé privée
PublicKey	Une clé publique

27.3. Les fournisseurs d'implémentations

L'architecture de JCA permet à des fournisseurs d'enrichir la plate-forme avec leurs propres implémentations. Un fournisseur de service propose un accès à différents algorithmes de cryptographie.

JCA propose une architecture qui permet d'utiliser des implémentations fournies par des tiers. Des fournisseurs de services cryptographiques (CSP : Cryptographic Service Provider) fournissent des implémentations particulières de services reposant sur l'API JCA.

Chaque JDK est fourni avec une ou plusieurs implémentations par défaut notamment deux nommées Sun et SunJCE pour le JDK de Sun/Oracle. Il est possible d'ajouter d'autres implémentations de fournisseurs et de préciser un ordre de préférence d'utilisation.

Sun propose avec Java 1.2 plusieurs implémentations des principaux algorithmes dans le package sun.security.provider. Avec Java 1.3, Sun propose plusieurs implémentations des algorithmes RSA dans le package com.sun.rsajca.

Le JDK de Sun propose en standard une implémentation nommée SUN qui permet d'utiliser :

- Digital Signature Algorithm (DSA)
- MD5
- SHA-1
- une fabrique pour les certificats X.509
- une implémentation d'un keystore nommée jks

Le JDK 5.0 contient un fournisseur nommé SunPKCS11

Le JDK sous Windows contient un fournisseur nommé MSCAPI qui permet d'utiliser les services natifs CryptoAPI de Microsoft.

La plate-forme Java fournit plusieurs implémentations d'algorithmes cryptographiques couramment utilisés. Ces implémentations restent cependant « basiques » : il est possible d'utiliser des implémentations plus complexes fournies par des tiers.

L'API propose un mécanisme pour permettre l'ajout de nouvelles implémentations par des fournisseurs tiers.

Pour enregistrer de manière statique un fournisseur, il faut le déclarer dans le fichier <java_home>/lib/security/java.security :

security.provider.n=nom_pleinement_qualifie_de_la_classe_principale

Pour enregistrer de manière dynamique un fournisseur, il faut utiliser la méthode addProvider() ou la méthode insertProviderAt() de la classe java.security.Security. L'utilisation de cette classe requiert des privilèges particuliers.

L'API JCA permet à des tiers de proposer des implémentations de services relatifs à la cryptographie (Cryptographic Service Provider). Différents fournisseurs peuvent fournir leurs propres implémentations de différents algorithmes pour un ou plusieurs services.

JCA permet d'obtenir la liste des fournisseurs installés et des services qu'ils proposent.

27.4. La classe java.security.Provider

La classe java.security.Provider sert de point d'entrée pour une implémentation de services cryptographiques proposée par un fournisseur (désignée par l'API JCA sous le nom provider). Chaque implémentation doit fournir une classe fille de la classe Provider.

La classe Provider propose plusieurs méthodes pour obtenir des informations sur l'implémentation.

Méthode	Rôle
String getName()	Obtenir le nom sensible à la casse
double getVersion()	Obtenir le numéro de version
String getInfo()	Obtenir une description

La classe Provider.Service encapsule un service de l'implémentation : la classe Provider propose plusieurs méthodes pour gérer ces services.

Méthode	Rôle
Provider.Service getService(String type, String algorithm)	Obtenir le service correspondant au type et à l'algorithme fournis en paramètres
Set<Provider.Service> getServices()	Obtenir une collection immuable des services
protected void removeService(Provider.Service s)	Retirer un service

protected void putService(Provider.Service s)	Ajouter un service

Dans chaque JVM, les fournisseurs sont enregistrés dans un ordre particulier qui permet de rechercher une implémentation particulière si aucun fournisseur n'est pas explicitement précisé.

Pour pouvoir utiliser une implémentation, il faut l'installer et la configurer.

1. Il faut l'ajouter l'implémentation dans le classpath.
2. Il faut ajouter le fournisseur à la liste de ceux utilisables dans le fichier java.security contenu dans le sous-répertoire lib/security du JRE.

L'enregistrement se fait dans le fichier sous la forme :

security.provider.*n*=NomDeLaClassePrincipale

Le n permet de préciser l'ordre dans lequel le fournisseur sera utilisé lors d'une recherche d'un algorithme. Le fournisseur le plus prioritaire possède l'ordre numéro 1.

La valeur NomDeLaClassePrincipale est le nom de la classe précisée par le fournisseur dans sa documentation. Cette classe hérite de la classe Provider.

Il est possible d'enregistrer dynamiquement un fournisseur en invoquant les méthodes addProvider() ou insertProviderAt() de la classe Security.

27.5. La classe java.security.Security

La classe java.security.Security permet de gérer les fournisseurs de services en enregistrant ou en retirant une implémentation. Elle permet aussi d'obtenir la liste des fournisseurs enregistrés.

Cette classe ne contient que des méthodes statiques : il n'est donc pas nécessaire de l'instancier pour l'utiliser.

Méthode	Rôle
Provider[] getProviders()	Renvoyer un tableau des fournisseurs installés dans leur ordre de préférence d'utilisations
Provider getProvider (String providerName)	Renvoyer un fournisseur à partir de son nom. Renvoie null si le nom n'est pas trouvé
int addProvider(Provider provider)	Ajouter un fournisseur à la fin de la liste. Renvoie la position d'insertion ou -1 si l'ajout est impossible car le fournisseur est déjà installé
int insertProviderAt (Provider provider, int position)	Insérer un fournisseur à la position précisée dans la liste. Renvoyer la position d'insertion ou -1 si l'ajout est impossible car le fournisseur est déjà installé
void removeProvider(String name)	Retirer le fournisseur de la liste dont le nom est fourni en paramètre. Si le fournisseur est retiré, la position des autres fournisseurs est décalée en conséquence
String getProperty(String key)	Obtenir la valeur d'une propriété relative à la sécurité
void setProperty(String key, String data)	Modifier la valeur d'une propriété relative à la sécurité

Pour modifier la position d'un fournisseur, il faut le retirer puis l'ajouter à nouveau.

L'exemple ci-dessous affiche la liste de tous les fournisseurs enregistrés et utilisables dans la JVM.

Exemple :

```
package fr.jmdoudoux.dej.securite;

import java.security.Provider;
import java.security.Security;

public class TestProviders {

  public static void main(String[] args) {
    Provider[] providers = Security.getProviders();

    for (Provider provider : providers) {
      System.out.println("Provider : " + provider.getName() + " v"
          + provider.getVersion());
    }
  }
}
```

L'exemple ci-dessous, affiche tous les services du fournisseur SunJCE fourni en standard avec le JDK de Sun.

Exemple :

```
package fr.jmdoudoux.dej.securite;

import java.security.Provider;
import java.security.Provider.Service;
import java.security.Security;

public class TestProvider {

  public static void main(String[] args) {
    Provider provider = Security.getProvider("SunJCE");

    System.out.println("Services du provider " + provider.getName());
    for (Service service : provider.getServices()) {
      System.out.println("\t" + service.getType() + " "
          + service.getAlgorithm());
    }
  }
}
```

Résultat :

```
Services du provider SunJCE
        Cipher RSA
        Cipher DES
        Cipher DESede
        Cipher DESedeWrap
        Cipher PBEWithMD5AndDES
        Cipher PBEWithMD5AndTripleDES
        Cipher PBEWithSHA1AndRC2_40
        Cipher PBEWithSHA1AndDESede
        Cipher Blowfish
        Cipher AES
        Cipher AESWrap
        Cipher RC2
        Cipher ARCFOUR
        KeyGenerator DES
        KeyGenerator DESede
...
```

La méthode getAlgorithms() attend en paramètre le nom d'un type de service (Signature, MessageDigest, Cipher, Mac, KeyStore, ...) et renvoie un ensemble de noms d'algorithmes disponibles auprès des différents providers enregistrés.

Exemple :

```
package fr.jmdoudoux.dej.securite;

import java.security.Security;
```

```
public class TestProvider {

  public static void main(String[] args) {
    for (String algo : Security.getAlgorithms("Cipher")) {
      System.out.println(algo);
    }
  }
}
```

Résultat :

```
BLOWFISH
ARCFOUR
PBEWITHMD5ANDDES
RC2
RSA
PBEWITHMD5ANDTRIPLEDES
PBEWITHSHA1ANDDESEDE
DESEDE
AESWRAP
AES
DES
DESEDEWRAP
RSA/ECB/PKCS1PADDING
PBEWITHSHA1ANDRC2_40
```

La classe Security encapsule une liste de propriétés relatives à la sécurité globale de la JVM. Les valeurs de ces propriétés peuvent être obtenues en utilisant la méthode getProperty() et modifiées avec la méthode setProperty().

La classe Security ne peut être utilisée que dans certaines circonstances :

- une application locale qui ne s'exécute pas avec un Security Manager
- une applet ou une application qui a la permission

27.6. La classe java.security.MessageDigest

La classe MessageDigest permet d'obtenir et d'utiliser une implémentation d'un service de type message digest tel que MD5 ou SHA-1 qui calcule une valeur de hachage de taille fixe à partir d'une quantité de données fournies sous la forme d'un tableau d'octets.

La valeur de hachage produite permet d'identifier de manière quasi unique les données avec lesquelles elle a été calculée.

La méthode statique getInstance() permet de demander l'implémentation d'un algorithme particulier : plusieurs algorithmes sont utilisables (MD5 128bits, SHA-1 160bits, ...). Il existe plusieurs surcharges de la méthode getInstance(). La plus simple attend uniquement le nom de l'algorithme : ce nom n'est pas sensible à la casse.

Exemple :

```
MessageDigest.getInstance("sha-1");
MessageDigest.getInstance("SHA-1");
```

Une autre surcharge permet de préciser le nom de l'algorithme et le nom du fournisseur. Une troisième surcharge permet de préciser le nom de l'algorithme et le fournisseur sous la forme d'une instance de type Provider.

- static MessageDigest getInstance(String algorithm, String provider)
- static MessageDigest getInstance(String algorithm, Provider provider)

Les surcharges de la méthode update() permettent d'ajouter des données à traiter en une seule ou plusieurs fois.

- void update(byte input)

- void update(byte[] input)
- void update(byte[] input, int offset, int len)

Une fois que toutes les données ont été ajoutées, il faut invoquer une des surcharges de la méthode digest() pour calculer la valeur de hachage pour les données fournies.

- byte[] digest()
- byte[] digest(byte[] input)
- int digest(byte[] buf, int offset, int len)

Les deux premières surcharges renvoient la valeur de hachage. La seconde surcharge invoque la méthode update() avec les données fournies en paramètre avant de calculer la valeur de hachage. La troisième surcharge copie la valeur de hachage dans le tampon fourni en paramètre et renvoie la taille des données copiées dans le tampon. Si le tampon est trop petit alors une exception de type DigestException est levée.

Exemple :

```
  public static byte[] calculerValeurDeHachage(String algorithme,
      String monMessage) {
    byte[] digest = null;
    try {
      MessageDigest sha = MessageDigest.getInstance(algorithme);
      sha.update(monMessage.getBytes());
      digest = sha.digest();
      System.out.println("algorithme : " + algorithme);
      System.out.println(bytesToHex(digest));
    } catch (NoSuchAlgorithmException e) {
      e.printStackTrace();
    }
    return digest;
  }
```

Après le calcul de la valeur de hachage, l'instance de type MessageDigest est automatiquement réinitialisée pour être de nouveau capable de calculer la valeur de hachage d'un nouveau message.

La méthode reset() permet de réinitialiser l'objet pour calculer une valeur de hachage pour d'autres données.

Exemple :

```
package fr.jmdoudoux.dej.securite;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class TestMessageDigest {
  public static void main(String[] args) {
    String monMessage = "Mon message";

    calculerValeurDeHachage("MD2", monMessage);
    calculerValeurDeHachage("MD5", monMessage);
    calculerValeurDeHachage("SHA-1", monMessage);
    calculerValeurDeHachage("SHA-256", monMessage);
    calculerValeurDeHachage("SHA-384", monMessage);
    calculerValeurDeHachage("SHA-512", monMessage);

  }

  public static byte[] calculerValeurDeHachage(String algorithme,
      String monMessage) {
    byte[] digest = null;
    try {
      MessageDigest sha = MessageDigest.getInstance(algorithme);
      digest = sha.digest(monMessage.getBytes());
      System.out.println("algorithme : " + algorithme);
      System.out.println(bytesToHex(digest));
    } catch (NoSuchAlgorithmException e) {
      e.printStackTrace();
    }
    return digest;
  }

  public static String bytesToHex(byte[] b) {
    char hexDigits[] = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A',
        'B', 'C', 'D', 'E', 'F' };
    StringBuffer buffer = new StringBuffer();
    for (int j = 0; j < b.length; j++) {
      buffer.append(hexDigits[(b[j] >> 4) & 0x0f]);
      buffer.append(hexDigits[b[j] & 0x0f]);
    }
    return buffer.toString();
  }

}
```

Résultat :

```
algorithme : MD2
78B9CEFD91776D6518008B8A9A92AC6F
algorithme : MD5
F547EA699E14B43ABE0C43FA7B809705
algorithme : SHA-1
36BF135AA507ABC0613117EAFFE5280250A13BBF
algorithme : SHA-256
E648CBA96738CF7F6DD84FBB92223063D7DFEE2B5678AE18BA9D006E82337648
algorithme : SHA-384
555A7AFAA68B5A9EF632F69287725ABAF58A3AA944A270E977F52D9DCFAC9EA1
CD4A3C5DFDB09AF3A8B6BF75883E0EAB
algorithme : SHA-512
D279F5BD5AA43D74080D88C92621315489A3071A7D423307E33D6AFE6BBF2CAD
9B34400ED822DE346269AE3410FAB055D16E89CB4785D93997EAF2B8100CC02B
```

27.7. Les classes DigestInputStream et DigestOuputStream

Les classes java.security.DigestInputStream et java.security.DigestOutputStream permettent de calculer une valeur de hachage pour des octets contenus dans un flux.

Ces deux classes héritent de la classe FilterOuputStream : les octets lus du flux ou écrits dans le flux sont ajoutés aux données à traiter par un objet de type MessageDigest associé à l'instance.

La classe DigestInputStream possède un seul constructeur qui attend en paramètre une instance de type InputStream et une instance de type MessageDigest.

La lecture de données du flux doit se faire en utilisant une des surcharges de la méthode read() de la classe DigestInputStream.

Exemple :

```
package fr.jmdoudoux.dej.securite;

import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.security.DigestInputStream;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class TestDigestInputStream {

  public static void main(String[] args) {
    InputStream is;
    DigestInputStream dis = null;
    try {
      is = new BufferedInputStream(new FileInputStream("monfichier.txt"));
      MessageDigest md = MessageDigest.getInstance("SHA-1");
      dis = new DigestInputStream(is, md);

      byte[] buffer = new byte[64];
      while (dis.read(buffer) != -1)
        ;

    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (NoSuchAlgorithmException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      if (dis != null) {
        try {
          dis.close();
          byte[] hash = dis.getMessageDigest().digest();
          System.out.println(ConvertionHelper.bytesToHex(hash));
        } catch (IOException e) {
          e.printStackTrace();
        }
      }
    }
  }
}
```

La classe DigestOutputStream possède un seul constructeur qui attend en paramètre une instance de type OutputStream et une instance de type MessageDigest.

L'écriture des données dans le flux doit se faire en utilisant une des surcharges de la méthode write() de la classe DigestOutputStream.

Exemple :

```
package fr.jmdoudoux.dej.securite;

import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.security.DigestOutputStream;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class TestDigestOutputStream {

  public static void main(String[] args) {

    byte[] donnees = "Hello World".getBytes();

    FileOutputStream fop = null;
    File file;
    DigestOutputStream dos = null;

    try {
      MessageDigest md = MessageDigest.getInstance("SHA-512");

      file = new File("fichier.txt");
      fop = new FileOutputStream(file);

      dos = new DigestOutputStream(fop, md);
      dos.write(donnees);
    } catch (IOException e) {
      e.printStackTrace();
    } catch (NoSuchAlgorithmException e) {
      e.printStackTrace();
    } finally {
      try {
        if (dos != null) {
          dos.close();
          byte[] hash = dos.getMessageDigest().digest();
          System.out.println(ConvertionHelper.bytesToHex(hash));
        }
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
}
```

27.8. La classe java.security.Signature

La classe java.security.Signature permet d'obtenir et d'utiliser une implémentation d'un service de type signature digitale tel que DSA ou RSA avec SHA1 ou MD5.

Elle permet de signer des données et de vérifier des données signées.

Un objet de type Signature possède un état qui peut prendre trois valeurs :

- UNINITIALIZED
- SIGN
- VERIFY

Pour créer une signature digitale, il faut instancier un objet de type Signature et l'initialiser avec une clé privée. Les données sont fournies à l'objet Signature pour créer la signature.

Pour vérifier la signature, il faut créer un objet de type Signature et l'initialiser avec la clé publique. Les données et la signature sont fournies à l'objet Signature pour vérification.

Pour obtenir une instance de type Signature avec un algorithme spécifique, il faut invoquer la méthode statique getInstance() en lui précisant le nom de l'algorithme qui n'est pas sensible à la casse. Deux surcharges de la méthode getInstance() permettent aussi de préciser explicitement le fournisseur par son nom ou par une instance de type Provider :

- public static Signature getInstance(String algorithm)
- public static Signature getInstance(String algorithm, String provider)
- public static Signature getInstance(String algorithm, Provider provider)

Chaque implémentation de la plate-forme Java doit fournir au minimum l'implémentation de trois algorithmes : SHA1withDSA, SHA1withRSA et SHA256withRSA.

A sa création, une instance de type Signature est initialisée à l'état UNINITIALIZED.

Pour utiliser un objet de type Signature, il faut l'initialiser dans un mode de fonctionnement.

La classe Signature possède deux méthodes initSign() et initVerify() qui permettent respectivement de passer l'état à SIGN ou VERIFY selon l'utilisation qui doit être faite.

La méthode initSign() attend en paramètre un objet de type PrivateKey qui encapsule la clé privée pour signer des données.

- void initSign(PrivateKey privateKey)

La méthode initVerify() possède deux surcharges qui attendent respectivement en paramètre :

- un objet de type PublicKey qui encapsule la clé publique : void initVerify(PublicKey publicKey)
- un objet de type Certificate qui encapsule le certificat : void initVerify(Certificate certificate)

Pour signer des données, il faut utiliser une instance de type Signature dans l'état SIGN. Les données doivent être fournies en utilisant une des surcharges de la méthode update() :

- final void update(byte b)
- final void update(byte[] data)
- final void update(byte[] data, int off, int len)

La méthode update() doit être invoquée autant de fois que nécessaire pour fournir toutes les données qui doivent être signées.

Pour générer la signature des données avec la clé, il faut invoquer une des surcharges de la méthode sign() :

- final byte[] sign() : renvoie la signature sous la forme d'un tableau d'octets
- final int sign(byte[] outbuf, int offset, int len) : stocke la signature dans le tableau d'octets fourni en paramètre. Elle renvoie le nombre d'octets insérés dans le tableau à partir de l'offset précisé

A la fin de l'invocation de la méthode sign(), l'instance de type Signature est réinitialisée à son état suite à l'invocation de la méthode initSign(). Elle peut alors être réutilisée pour signer une autre quantité de données avec la clé. Pour préciser une autre clé, il faut de nouveau invoquer la méthode initSign().

Pour vérifier des données signées, il faut utiliser une instance de type Signature dans l'état VERIFY. Les données doivent être fournies en utilisant une des surcharges de la méthode update() :

- final void update(byte b)
- final void update(byte[] data)
- final void update(byte[] data, int off, int len)

La méthode update() doit être invoquées autant de fois que nécessaire pour fournir toutes les données qui doivent être signées.

Pour vérifier les données avec la signature, il faut invoquer une des surcharge de la méthode verify() qui renvoie un booléen indiquant si oui ou non la signature encodée est la signature authentique des données fournies grâce à la méthode update() :

- final boolean verify(byte[] signature) : elle attend en paramètre un tableau d'octets qui contient la signature
- final boolean verify(byte[] signature, int offset, int length)

A la fin de l'invocation de la méthode verify(), l'instance de type Signature est réinitialisée à son état suite à l'invocation de la méthode initVerify(). Elle peut alors être réutilisée pour vérifier une autre signature pour d'autres données. Pour préciser une autre clé, il faut de nouveau invoquer la méthode initVerify().

Exemple :

```
package fr.jmdoudoux.dej.securite;

import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.SecureRandom;
import java.security.Signature;

public class TestSignature {

  public static void main(String[] args) throws Exception {
    byte[] message = "Hello world".getBytes();

    KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("RSA");
    keyPairGen.initialize(1024, new SecureRandom());
    KeyPair keyPair = keyPairGen.generateKeyPair();

    Signature signature = Signature.getInstance("SHA1withRSA");

    signature.initSign(keyPair.getPrivate(), new SecureRandom());
    signature.update(message);
    byte[] signatureBytes = signature.sign();
    System.out.println(ConvertionHelper.bytesToHex(signatureBytes));

    signature.initVerify(keyPair.getPublic());
    signature.update(message);
    System.out.println(signature.verify(signatureBytes));
  }
}
```

Résultat :

```
63E72D3E8271401DC4E2A0930EA6B3E14AB1FF7BB88DD61EAF57BDDDFB7B64C0
33AB5F67CE780B724CC0B52D153BA32663B456E49B980BA4B9BE521423D9CAED
3EC3C2C00600A685FEB0C9C30FCFF2C27A0A3C102D85E3F5CC758497763290E9
4C741BDAED3576A529698F7F05A8119C54256931B5F9A8658FFE9619574F112F
true
```

27.9. La classe java.security.KeyStore

La classe KeyStore permet de stocker, gérer et récupérer les éléments contenus dans un dépôt de clés.

La classe KeyStore permet d'accéder et de modifier les deux types d'éléments que peut contenir un dépôt : des clés et des certificats. Chaque élément contenu dans un keyStore est identifié par un alias unique.

L'implémentation est libre de choisir la solution pour rendre persistantes les données qu'elle contient et pour sécuriser l'accès à ces données.

Pour obtenir une instance de type KeyStore, il faut invoquer la méthode getInstance() qui est une fabrique attendant une chaîne de caractères qui précise le type de KeyStore à obtenir. Deux autres surcharges attendent en paramètre un objet de type Provider ou une chaîne de caractères qui permet de préciser le fournisseur de l'implémentation.

- static KeyStore getInstance(String type)
- static KeyStore getInstance(String type, String provider)
- static KeyStore getInstance(String type, Provider provider)

Le type de dépôt n'est pas sensible à la casse.

La classe KeyStore possède plusieurs méthodes :

Méthode	Rôle
final void load(InputStream stream, char[] password)	Pour charger les données du keystore, il faut invoquer la méthode load() Le second paramètre est un mot de passe qui, s'il est fourni, permet de vérifier l'intégrité des données. Pour créer un keystore vide, il faut invoquer la méthode load() avec la valeur null comme premier paramètre.
Enumeration aliases()	Obtenir une énumération de tous les alias associés à un élément du keystore
boolean isKeyEntry(String alias)	Renvoyer un booléen qui précise si l'élément associé à l'alias fourni en paramètre est une clé
boolean isCertificateEntry(String alias)	Renvoyer un booléen qui précise si l'élément associé à l'alias fourni en paramètre est un certificat
void setCertificateEntry(String alias, Certificate cert)	Ajouter un certificat dans le keystore et lui associer un alias si l'entrée n'existe pas dans le keystore ou modifier le certificat s'il existe
final void setKeyEntry(String alias, Key key, char[] password, Certificate[] chain) final void setKeyEntry(String alias, byte[] key, Certificate[] chain)	Ajouter une clé dans le keystore et lui associer un alias si l'entrée n'existe pas dans le keystore ou modifier le certificat s'il existe
final void deleteEntry(String alias)	Supprimer l'entrée dans le dépôt de clés pour l'alias en fourni en paramètre
final Key getKey(String alias, char[] password)	Obtenir la clé correspondant à l'alias fourni en paramètre. Le mot de passe fourni en paramètre doit être celui associé à la protection de la clé.

final Certificate getCertificate(String alias) final Certificate[] getCertificateChain(String alias)	Obtenir le certificat ou les certificats correspondant à l'alias fourni en paramètre
final String getCertificateAlias(Certificate cert)	Obtenir l'alias du premier élément du dépôt associé au certificat fourni en paramètre
final void store(OutputStream stream, char[] password)	Sauvegarder le keystore. Le mot de passe est utilisé pour calculer un checksum des données et vérifier ultérieurement son intégrité
final static String getDefaultType()	Obtenir le type par défaut de dépôt de clés à utiliser : cette valeur est définie par la propriété keystore.type dans le fichier de sécurité. Si cette valeur n'est pas définie explicitement, elle est par défaut à «JKS»

Pour obtenir une instance de type KeyStore, il faut utiliser la méthode statique getInstance() qui est une fabrique.

Il faut ensuite charger en mémoire le contenu du dépôt de clés en utilisant la méthode load(). Le paramètre optionnel password est utilisé pour vérifier l'intégrité des données du dépôt : si le paramètre n'est pas fourni alors la vérification n'est pas effectuée.

Exemple : afficher un certificat du dépôt

Exemple :

```
package fr.jmdoudoux.dej.securite;

import java.io.FileInputStream;
import java.security.KeyStore;
import java.security.cert.Certificate;

public class TestKeyStoreExportCert {

  public static void main(String[] args) {

    char[] mdp = { '1', '2', '3', '4', '5', '6' };

    FileInputStream is;
    try {
      is = new FileInputStream("c:/java/jmPrivateKey.store");
      KeyStore keystore = KeyStore.getInstance(KeyStore.getDefaultType());
      keystore.load(is, mdp);

      String alias = "jmTrustKey";
      Certificate cert = keystore.getCertificate(alias);

      System.out.println(cert);

    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Exemple : obtenir une paire de clés du dépôt

Exemple :

```
package fr.jmdoudoux.dej.securite;

import java.io.FileInputStream;
import java.security.Key;
import java.security.KeyPair;
import java.security.KeyStore;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.UnrecoverableKeyException;
import java.security.cert.Certificate;

public class TestKeyStoreObtenirKeyPair {

  public static void main(String[] args) {

    char[] mdpDepot = { '1', '2', '3', '4', '5', '6' };
    char[] mdpCle = { 'a', 'b', 'c', 'd', 'e', 'f' };

    FileInputStream is;
    try {
      is = new FileInputStream("c:/java/jmPrivateKey.store");
      KeyStore keystore = KeyStore.getInstance(KeyStore.getDefaultType());
      keystore.load(is, mdpDepot);

      KeyPair cles = getPrivateKey(keystore, "jmTrustKey", mdpCle);

    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public static KeyPair getPrivateKey(KeyStore keystore, String alias, char[] password)
      throws UnrecoverableKeyException, KeyStoreException, NoSuchAlgorithmException {
    KeyPair resultat = null;

    Key clePrivee = keystore.getKey(alias, password);

    if (clePrivee instanceof PrivateKey) {
      Certificate cert = keystore.getCertificate(alias);
      PublicKey clePublique = cert.getPublicKey();
      resultat =  new KeyPair(clePublique, (PrivateKey) clePrivee);
    }
    return resultat;
  }
}
```

Plusieurs outils du JDK utilisent la classe KeyStore : keytool, jarsigner et Policy Tool.

L'implémentation de la classe KeyStore est dépendante du fournisseur : le JDK de Sun/Oracle propose une implémentation nommée SUN qui stocke le dépôt dans un fichier. Le type de ce dépôt est JKS.

L'implémentation nommée SunJCE propose le type de dépôt JCEKS qui met en oeuvre un chiffrement des mots de passes utilisant un algorithme de type Triple DES.

27.10. Les interfaces de type java.security.Key

L'interface Key décrit les fonctionnalités communes à toutes les clés opaques. Une clé opaque ne permet pas un accès direct à sa valeur et possède plusieurs caractéristiques :

- un algorithme (par exemple : DSA, RSA, MD5withRSA, SHA1withRSA, ...)
- une forme encodée qui permet une utilisation hors de la JVM par exemple pour transmettre la clé à un tiers
- le format qui a encodé la clé

Elle définit plusieurs méthodes pour les obtenir :

Méthode	Rôle
String getAlgorithm()	Retourner le nom de l'algorithme pour lequel la clé est adaptée
byte[] getEncoded()	Retourner la forme encodée de la clé
String getFormat()	Retourner le format utilisé pour encoder la clé

De nombreuses interfaces héritent de l'interface Key.

JCA	JCE
java.security.PrivateKey java.security.PublicKey java.security.interfaces.DSAPrivateKey java.security.interfaces.DSAPublicKey java.security.interfaces.ECPrivateKey java.security.interfaces.ECPublicKey java.security.interfaces.RSAMultiPrimePrivateCrtKey java.security.interfaces.RSAPrivateCrtKey java.security.interfaces.RSAPrivateKey java.security.interfaces.RSAPublicKey	javax.crypto.interfaces.DHPrivateKey javax.crypto.interfaces.DHPublicKey javax.crypto.interfaces.PBEKey javax.crypto.SecretKey

Les interfaces java.security.PublicKey et java.security.PrivateKey héritent de l'interface Key. Elles sont uniquement des marqueurs : elles ne définissent aucune méthode supplémentaire. Les interfaces pour les clés publiques ou privées de certains algorithmes héritent de ces interfaces.

27.11. La classe java.security.KeyPair

La classe KeyPair encapsule une paire de clés : une clé publique et une clé privée.

Elle possède deux méthodes pour obtenir les clés :

- PrivateKey getPrivate() : obtenir la clé privée
- PublicKey getPublic() : obtenir la clé publique

Les objets de type Key (clé opaque) et KeySpec (clé transparente) sont deux représentations des données d'une clé. Les algorithmes de cryptage utilisent une clé opaque mais les clés doivent être transformées dans un format plus portable pour être transmises ou stockées.

Il est possible d'accéder à chacun des éléments d'une clé transparente en utilisant des getters : ces propriétés dépendent de l'implémentation de la clé. Par exemple, classe DSAPrivateKeySpec permet un accès aux paramètres utilisés pour le calcul de la clé : nombre premier p, nombre premier q, la base g et à la clé privée x.

La représentation opaque d'une clé est définit par l'interface Key qui ne contient que trois méthodes : getAlgorithm(), getFormat() et getEncoded().

27.12. La classe java.security.KeyPairGenerator

La classe KeyPairGenerator permet de générer une paire de clés : une publique et une privée.

La génération d'une paire de clés peut se faire de deux manières :

- de manière indépendante de tout algorithme
- de manière spécifique à un algorithme : dans ce cas, la seule différence est la nécessité d'initialiser l'objet.

Pour obtenir une instance de type KeyPairGenerator, il faut invoquer la méthode getInstance() qui est une fabrique qui attend en paramètre le nom de l'algorithme. Deux autres surcharges, permettent aussi de préciser quel est le fournisseur de l'algorithme à utiliser.

- static KeyPairGenerator getInstance(String algorithm)
- static KeyPairGenerator getInstance(String algorithm, String provider)

- static KeyPairGenerator getInstance(String algorithm, Provider provider)

La classe KeyPairGenerator permet de créer une paire de clés publique/privée utilisable pour un algorithme particulier.

Un objet de type KeyPairGenerator doit être initialisé avant de pouvoir générer des clés. Cette initialisation concerne la taille de la clé et une solution pour générer des nombres aléatoires.

- void initialize(int keysize, SecureRandom random)

Une surcharge de la méthode initialize() attend uniquement en paramètre la taille de la clé et utilise un objet de type SecureRandom par défaut.

Pour certains algorithmes, cette initialisation peut en plus être particulière. Deux surcharges de la méthode initialize() attendent en paramètre un objet de type AlgorithmParameterSpec

- void initialize(AlgorithmParameterSpec params, SecureRandom random)
- void initialize(AlgorithmParameterSpec params)

La méthode generateKeyPair() renvoie une instance de type KeyPair.

Plusieurs appels à la méthode generateKeyPair() permet d'obtenir plusieurs instances.

Exemple :

```
package fr.jmdoudoux.dej.securite;

import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;

public class TestKeyPairGenerator {
  public static void main(String[] argv) throws Exception {
    KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA");
    keyGen.initialize(1024);
    KeyPair keypair = keyGen.genKeyPair();
    PrivateKey privateKey = keypair.getPrivate();
    System.out.println(privateKey);
    PublicKey publicKey = keypair.getPublic();
    System.out.println(publicKey);
  }
}
```

Résultat :

```
Sun DSA Private Key
parameters:
    p:
    fd7f5381 1d751229 52df4a9c 2eece4e7 f611b752 3cef4400 c31e3f80 b6512669
    455d4022 51fb593d 8d58fabf c5f5ba30 f6cb9b55 6cd7813b 801d346f f26660b7
    6b9950a5 a49f9fe8 047b1022 c24fbba9 d7feb7c6 1bf83b57 e7c6a8a6 150f04fb
    83f6d3c5 1ec30235 54135a16 9132f675 f3ae2b61 d72aeff2 2203199d d14801c7
    q:
    9760508f 15230bcc b292b982 a2eb840b f0581cf5
    g:
    f7e1a085 d69b3dde cbbcab5c 36b857b9 7994afbb fa3aea82 f9574c0b 3d078267
    5159578e bad4594f e6710710 8180b449 167123e8 4c281613 b7cf0932 8cc8a6e1
    3c167a8b 547c8d28 e0a3ae1e 2bb3a675 916ea37f 0bfa2135 62f1fb62 7a01243b
    cca4f1be a8519089 a883dfe1 5ae59f06 928b665e 807b5525 64014c3b fecf492a

x:     4e775468 567ff544 85411a5e ba8b839b ec56beb4

Sun DSA Public Key
    Parameters:
    p:
    fd7f5381 1d751229 52df4a9c 2eece4e7 f611b752 3cef4400 c31e3f80 b6512669
    455d4022 51fb593d 8d58fabf c5f5ba30 f6cb9b55 6cd7813b 801d346f f26660b7
    6b9950a5 a49f9fe8 047b1022 c24fbba9 d7feb7c6 1bf83b57 e7c6a8a6 150f04fb
    83f6d3c5 1ec30235 54135a16 9132f675 f3ae2b61 d72aeff2 2203199d d14801c7
    q:
    9760508f 15230bcc b292b982 a2eb840b f0581cf5
    g:
    f7e1a085 d69b3dde cbbcab5c 36b857b9 7994afbb fa3aea82 f9574c0b 3d078267
    5159578e bad4594f e6710710 8180b449 167123e8 4c281613 b7cf0932 8cc8a6e1
    3c167a8b 547c8d28 e0a3ae1e 2bb3a675 916ea37f 0bfa2135 62f1fb62 7a01243b
    cca4f1be a8519089 a883dfe1 5ae59f06 928b665e 807b5525 64014c3b fecf492a

  y:
    2e5efc6a 8a8e046e c06cfb5f dc3d1acb 83ff9209 1b189208 0378eaef ac919241
    96242890 c285e26d 91a0386d 1b30e856 aaf6e3bc 90184f0e 76ea8551 d6333faa
    349a480e 13baa599 0c6e571b f649251f 099726dc a9b1412c ebf57990 5ccee057
    9fa2b515 0585afa1 19365c58 dd741347 889c0e25 0f112476 d49dc7af 70b61867
```

Exemple :

```
package fr.jmdoudoux.dej.securite;

import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.SecureRandom;

public class TestKeyPairGenerator2 {

  public static void main(String[] args) throws Exception {
    KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA");

    byte[] userSeed = new byte[256];

    SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");
    random.setSeed(userSeed);
    keyGen.initialize(1024, random);
    KeyPair keypair = keyGen.genKeyPair();
    PrivateKey privateKey = keypair.getPrivate();
    System.out.println(privateKey);
    PublicKey publicKey = keypair.getPublic();
    System.out.println(publicKey);
  }

}
```

27.13. La classe java.security.KeyFactory

La classe KeyFactory permet de convertir des clés transparentes en clés opaques et vice versa.

Pour obtenir une instance de type KeyFactory, il faut invoquer sa méthode getInstance() en lui fournissant le nom de l'algorithme à utiliser. Ce nom n'est pas sensible à la casse.

La méthode getInstance() possède deux autres surcharges qui permettent de préciser le nom du fournisseur sous la forme d'une chaîne de caractères ou d'une instance de type Provider.

La méthode generatePublic() attend en paramètre un objet de type KeySpec et renvoie un objet de type PublicKey.

- PublicKey generatePublic(KeySpec keySpec)

Elle permet de transformer une clé publique transparente en une clé publique opaque.

Exemple :

```
package fr.jmdoudoux.dej.securite;

import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;
import java.security.PublicKey;
import java.security.spec.EncodedKeySpec;
import java.security.spec.InvalidKeySpecException;
import java.security.spec.X509EncodedKeySpec;

public class TestKeyFactory {

  public static void main(String[] args) {
    try {

      // Generation d'une paire de cles opaques pour RSA
      KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
      keyGen.initialize(1024);
      KeyPair keypair = keyGen.genKeyPair();
      PublicKey publicKey = keypair.getPublic();
      System.out.println(ConvertionHelper.bytesToHex(publicKey.getEncoded()));

      // Convertion de la cle opaque en cle transparente
      KeyFactory keyFactory = KeyFactory.getInstance("RSA");
      byte[] publicKeyBytes = publicKey.getEncoded();
      EncodedKeySpec publicKeySpec = new X509EncodedKeySpec(publicKeyBytes);

      // Reconvertion de la cle transparente en cle opaque
      PublicKey publicKey2 = keyFactory.generatePublic(publicKeySpec);
      System.out.println(ConvertionHelper.bytesToHex(publicKey2.getEncoded()));
    } catch (NoSuchAlgorithmException e) {
      e.printStackTrace();
    } catch (InvalidKeySpecException e) {
      e.printStackTrace();
    }
  }
}
```

La méthode generatePrivate() attend en paramètre un objet de type KeySpec et renvoie un objet de type PrivateKey.

- PrivateKey generatePrivate(KeySpec keySpec)

Elle permet de transformer une clé privée transparente en une clé privée opaque.

Exemple :

```
package fr.jmdoudoux.dej.securite;

import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.spec.EncodedKeySpec;
import java.security.spec.InvalidKeySpecException;
import java.security.spec.PKCS8EncodedKeySpec;

public class TestKeyFactory {

  public static void main(String[] args) {
    try {
      // Generation d'une paire de cles opaques pour RSA
      KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
      keyGen.initialize(1024);
      KeyPair keypair = keyGen.genKeyPair();
      PrivateKey privateKey = keypair.getPrivate();
      System.out.println(ConvertionHelper.bytesToHex(privateKey.getEncoded()));

      // Convertion de la cle opaque en cle transparente
      KeyFactory keyFactory = KeyFactory.getInstance("RSA");
      byte[] privateKeyBytes = privateKey.getEncoded();
      EncodedKeySpec privateKeySpec = new PKCS8EncodedKeySpec(privateKeyBytes);

      // Reconvertion de la cle transparente en cle opaque
      PrivateKey privateKey2 = keyFactory.generatePrivate(privateKeySpec);
      System.out.println(ConvertionHelper.bytesToHex(privateKey2.getEncoded()));

    } catch (NoSuchAlgorithmException e) {
      e.printStackTrace();
    } catch (InvalidKeySpecException e) {
      e.printStackTrace();
    }
  }
}
```

La méthode getKeySpec() attend en paramètres deux objets de type Key pour l'un et KeySpec pour l'autre. Elle renvoie un objet de type KeySpec.

- KeySpec getKeySpec(Key key, Class keySpec)

Elle permet de transformer une clé opaque en une clé transparente.

Exemple :

```
package fr.jmdoudoux.dej.securite;

import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;
import java.security.PublicKey;
import java.security.spec.InvalidKeySpecException;
import java.security.spec.RSAPublicKeySpec;

public class TestKeyFactory {

  public static void main(String[] args) {

    try {
      // Generation d'une paire de cles opaques pour RSA
      KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
      keyGen.initialize(1024);
      KeyPair keypair = keyGen.generateKeyPair();
      PublicKey publicKey = keypair.getPublic();

      // Convertion de la cle opaque en cle transparente
      KeyFactory kfactory = KeyFactory.getInstance("RSA");
      RSAPublicKeySpec keySpec = kfactory.getKeySpec(publicKey,
          RSAPublicKeySpec.class);
    } catch (NoSuchAlgorithmException e) {
      e.printStackTrace();
    } catch (InvalidKeySpecException e) {
      e.printStackTrace();
    }
  }
}
```

27.14. La classe java.security.SecureRandom

La classe de base du JDK pour générer un nombre aléatoire est la classe java.util.Random qui utilise une graine (seed en anglais) de 48 bits. Si le seed n'est pas fourni explicitement, alors une valeur basée sur l'heure courante est utilisée pour l'initialiser.

La génération de nombres aléatoires est très importante dans la cryptographie moderne : elle est fréquemment utilisée pour générer les clés ou les paramètres de certains algorithmes.

La classe java.security.SecureRandom est une implémentation d'un PRNG (Pseudo Random Number Generator) fort requis pour les besoins de la cryptographie. Elle diffère de la classe Random en produisant des nombres aléatoires utilisables avec des fonctions cryptographiques. Les implémentations de la classe SecureRandom tentent de rendre le plus aléatoire possible l'état interne du générateur.

Comme pour toutes les classes de type engine de JCA, pour obtenir une instance de type SecureRandom, il faut invoquer la méthode statique getInstance()

Pour créer une instance de type SecuredRandom, il est préférable d'utiliser une des surcharges de la méthode getInstance() qui est une fabrique :

- public static SecureRandom getInstance(String algorithm)
- public static SecureRandom getInstance(String algorithm, String provider)
- public static SecureRandom getInstance(String algorithm, Provider provider)

Il est aussi possible de préciser le fournisseur de l'implémentation à utiliser soit par son nom sous la forme d'une chaîne de caractères ou sous la forme d'une instance de type Provider.

L'implémentation du JCA proposée par Sun/Oracle propose deux implémentations d'algorithmes pour la génération de nombres aléatoires :

- SHA1PRNG par le provider SUN
- Windows-PRNG par le provider SunMSCAPI

Pour des raisons de compatibilité, il est possible d'utiliser un des constructeurs de la classe SecureRandom pour obtenir une instance :

- public SecureRandom()
- public SecureRandom(byte[] seed)

La méthode setSeed() permet de fournir une valeur qui sera utilisée pour renforcer le caractère aléatoire des valeurs générées en assurant la dispersion de ces valeurs.

- synchronized public void setSeed(byte[] seed)
- public void setSeed(long seed)

La méthode setSeed() permet de préciser l'état initial du générateur.

Il est aussi possible de ré-invoquer la méthode setSeed() sur une instance de SecureRandom qui est utilisée plusieurs fois : dans ce cas, les données fournies sont ajoutées aux données existantes au lieu de les remplacer pour ne pas risquer de diminuer le caractère aléatoire des valeurs générées.

Pour générer un nombre pseudo-aléatoire, il faut invoquer l'une des méthodes :

- int nextInt() : pour obtenir un entier
- int next(int n) : pour obtenir un entier entre 0 et la valeur n exclue
- double nextDouble() : pour obtenir un double
- float nextFloat() : pour obtenir un nombre flottant
- long nextLong() : pour obtenir un entier long
- void nextBytes(byte[] bytes): pour obtenir un tableau d'octets contenant des valeurs aléatoires

La méthode generateSeed() permet de générer un tableau d'octets de valeurs aléatoires dont la taille est fournie en paramètre.

- byte[] generateSeed(int numBytes)

Cette valeur peut être passée en paramètre de la méthode setSeed() pour réinitialiser la valeur seed d'une instance de type SecureRandom.

Exemple :

```
package fr.jmdoudoux.dej.securite;

import java.security.NoSuchAlgorithmException;
import java.security.NoSuchProviderException;
import java.security.SecureRandom;

public class TestSecureRandom {

  public static void main(String[] args) {
    SecureRandom secureRandom;
    try {
      secureRandom = SecureRandom.getInstance("SHA1PRNG");
      int randomValue = secureRandom.nextInt();
      System.out.println("valeur=" + randomValue);
    } catch (NoSuchAlgorithmException e) {
      e.printStackTrace();
    }

    try {
      secureRandom = SecureRandom.getInstance("SHA1PRNG");
      int seedByteCount = 128;
      byte[] seed = secureRandom.generateSeed(seedByteCount);
      secureRandom.setSeed(seed);
      System.out.println("valeur=" + secureRandom.nextDouble());
    } catch (NoSuchAlgorithmException e) {
      e.printStackTrace();
    }

    try {
      secureRandom = SecureRandom.getInstance("Windows-PRNG", "SunMSCAPI");
      int randomValue = secureRandom.nextInt();
      System.out.println("valeur=" + randomValue);
    } catch (NoSuchAlgorithmException e) {
      e.printStackTrace();
    } catch (NoSuchProviderException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
valeur=-813765588
valeur=0.027554423801555594
valeur=125142869
```

27.15. La classe java.security.AlgorithmParameters

La classe AlgorithmParameters encapsule les paramètres d'une fonction cryptographique de manière opaque.

La méthode statique getInstance() est une fabrique pour créer une instance de type AlgorithmParameters qui possède plusieurs surcharges :

- public static AlgorithmParameters getInstance(String algorithm)
- public static AlgorithmParameters getInstance(String algorithm, String provider)
- public static AlgorithmParameters getInstance(String algorithm, Provider provider)

L'instance doit être initialisée en utilisant une des surcharges de la méthode init()

- void init(AlgorithmParameterSpec paramSpec)
- void init(byte[] params)
- void init(byte[] params, String format)

Le tableau d'octets contient les paramètres encodés et la chaîne de caractères précise le format à utiliser pour décoder les données (ASN.1 par défaut).

Une instance de type AlgorithmParameters ne peut être initialisée qu'une seule fois.

La méthode getEncoded() renvoie un tableau d'octets qui contient l'AlgorithmParameters de manière encodée, par défaut avec le format ASN.1.

- byte[] getEncoded()

Une surcharge de la méthode getEncoded() permet de préciser le format d'encodage. Si la valeur null est passée en paramètre alors c'est le format par défaut qui est utilisé.

- byte[] getEncoded(String format)

La méthode getParameterSpec(Class paramSpec) permet de renvoyer une instance de type AlgorithmParameterSpec encapsulant les paramètres de manière transparente. Le paramètre de type Class permet de préciser le type de l'instance retournée.

Exemple :

```
package fr.jmdoudoux.dej.securite;

import java.math.BigInteger;
import java.security.AlgorithmParameters;
import java.security.spec.DSAParameterSpec;

public class TestAlgorithmParameters {

  public static void main(String[] args) {
    BigInteger p = new BigInteger("D24700960FFA32D3F1557344E5871"
        + "01237532CC641646ED7A7C104743377F6D46251698B665CE2A6"
        + "CBAB6714C2569A7D2CA22C0CF03FA40AC930201090202020", 16);
    BigInteger q = new BigInteger("09", 16);
    BigInteger g = new BigInteger("512");

    try {
      DSAParameterSpec dsaParamsSpec = new DSAParameterSpec(p, q, g);

      AlgorithmParameters algParams = AlgorithmParameters.getInstance("DSA");
      algParams.init(dsaParamsSpec);

      System.out.println(ConvertionHelper.bytesToHex(algParams.getEncoded()));
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

27.16. La classe java.security.AlgorithmParameterGenerator

La classe java.security.AlgorithmParameterGenerator permet de créer des classes qui encapsulent des paramètres pour certains algorithmes.

Pour créer une instance de type AlgorithmParameterGenerator, il faut invoquer la méthode static getInstance() qui est une fabrique.

- public static AlgorithmParameterGenerator getInstance(String algorithm)
- public static AlgorithmParameterGenerator getInstance(String algorithm, String provider)
- public static AlgorithmParameterGenerator getInstance(String algorithm, Provider provider)

L'instance obtenue doit être initialisée de manière dépendante ou indépendante de l'algorithme.

L'initialisation de l'instance de manière indépendante repose sur deux paramètres partagés par tous les algorithmes : une taille et une source pour obtenir des valeurs aléatoires. Deux surcharges de la méthode init() permettent de fournir ces informations :

- void init(int size, SecureRandom random);
- void init(int size)

La taille peut avoir une signification et une utilité particulière selon l'algorithme.

Deux autres surcharges de la méthode init() permettent d'initialiser l'instance en utilisant des paramètres spécifiques à l'algorithme encapsulés dans une instance de type AlgorithmParameterSpec.

- void init(AlgorithmParameterSpec genParamSpec, SecureRandom random)
- void init(AlgorithmParameterSpec genParamSpec)

Exemple :

```
package fr.jmdoudoux.dej.securite;

import java.math.BigInteger;
import java.security.AlgorithmParameterGenerator;
import java.security.AlgorithmParameters;
import java.security.NoSuchAlgorithmException;
import java.security.spec.DSAParameterSpec;
import java.security.spec.InvalidParameterSpecException;

public class TestAlgorithmParameterGenerator {

  public static void main(String[] args) {
    BigInteger q = null;
    BigInteger p = null;
    BigInteger g = null;

    AlgorithmParameterGenerator apgDSA;
    try {
      apgDSA = AlgorithmParameterGenerator.getInstance("DSA");
      AlgorithmParameters apDSA = apgDSA.generateParameters();
      DSAParameterSpec dsaparameterspec = apDSA
          .getParameterSpec(DSAParameterSpec.class);
      p = dsaparameterspec.getP();
      q = dsaparameterspec.getQ();
      g = dsaparameterspec.getG();
      System.out.println(p);
      System.out.println(q);
      System.out.println(g);
    } catch (NoSuchAlgorithmException e) {
      e.printStackTrace();
    } catch (InvalidParameterSpecException e) {
      e.printStackTrace();
    }
  }
}
```

27.17. La classe java.security.cert.CertificateFactory

La classe CertificateFactory est une fabrique pour générer des certificats et des CRL (Certificate Revocation List).

La classe permet d'obtenir une instance de type java.security.cert.X509Certificate pour un certificat X.509.

La classe permet d'obtenir une instance de type java.security.cert.X509CRL pour un CRL.

La méthode getInstance() permet d'obtenir une instance de type CertificateFactory : elle attend en paramètre le nom du type de certificat qui sera généré par la fabrique. Ce nom n'est pas sensible à la casse. Deux autres surcharges de la méthode getInstance() permettent de préciser le fournisseur.

- static CertificateFactory getInstance(String type)
- static CertificateFactory getInstance(String type, String provider)
- static CertificateFactory getInstance(String type, Provider provider)

Pour générer une instance de type Certificate, il faut invoquer la méthode generateCertificate(). Elle attend en paramètre un objet de type InputStream qui sera utilisé pour lire le contenu du certificat.

- final Certificate generateCertificate(InputStream inStream)

La méthode generateCertificates() retourne une collection des certificats lus à partir de l'instance de type InputStream fournie en paramètre.

- final Collection generateCertificates(InputStream inStream)

La méthode generateCRL() permet de créer une instance de type CLR qui encapsule une liste de certificats révoqués (Certificate Revocation List). Elle attend en paramètre un objet de type InputStream qui permet de lire les données.

- final CRL generateCRL(InputStream inStream)

La méthode generateCLRs() permet de créer une collection d'objets de type CLR.

- final Collection generateCRLs(InputStream inStream)

Exemple :

```
package fr.jmdoudoux.dej.securite;

import java.io.FileInputStream;
import java.io.IOException;
import java.security.cert.Certificate;
import java.security.cert.CertificateFactory;

public class TestCertificateFactory {

  public static void main(String[] args) {
    FileInputStream in = null;

    try {
      CertificateFactory cf = CertificateFactory.getInstance("X.509");
      in = new FileInputStream("C:/java/moncertificat.cer");
      Certificate cert = cf.generateCertificate(in);
      System.out.println(cert);
    } catch (Exception e) {
      e.printStackTrace();
    } finally {
      try {
        in.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
}
```

Résultat :

```
[
[
  Version: V3
  Subject: CN=Doudoux Jean-Michel, OU=Developpement, O=jmdoudoux.fr, L=Pont a Mousson,
  ST=Lorraine, C=FR
  Signature Algorithm: SHA1withDSA, OID = 1.2.840.10040.4.3

  Key:  Sun DSA Public Key
    Parameters:DSA
        p:     fd7f5381 1d751229 52df4a9c 2eece4e7 f611b752 3cef4400 c31e3f80 b6512669
    455d4022 51fb593d 8d58fabf c5f5ba30 f6cb9b55 6cd7813b 801d346f f26660b7
    6b9950a5 a49f9fe8 047b1022 c24fbba9 d7feb7c6 1bf83b57 e7c6a8a6 150f04fb
    83f6d3c5 1ec30235 54135a16 9132f675 f3ae2b61 d72aeff2 2203199d d14801c7
        q:     9760508f 15230bcc b292b982 a2eb840b f0581cf5
        g:     f7e1a085 d69b3dde cbbcab5c 36b857b9 7994afbb fa3aea82 f9574c0b 3d078267
    5159578e bad4594f e6710710 8180b449 167123e8 4c281613 b7cf0932 8cc8a6e1
    3c167a8b 547c8d28 e0a3ae1e 2bb3a675 916ea37f 0bfa2135 62f1fb62 7a01243b
    cca4f1be a8519089 a883dfe1 5ae59f06 928b665e 807b5525 64014c3b fecf492a

  y:
    0af345a2 4b0238b3 4e499986 6d9d9f1b 21b51090 a4cb28dc f98b587a 91bb98a1
    5e6f2d59 be186ee6 dd6bebbb a7800a5e 22c3f999 ab53ec9b 467522c5 9ef03ac7
    a034c4e4 56e3a116 93bc4737 4c0898dc a450df3b 71299b75 12c4a038 66a7a5ae
    3e57fde2 bdfc122d 6351b2c6 d6eae42c 890110d2 c263a8b4 d67d0743 a8c6eaa3

  Validity: [From: Sun Sep 01 22:53:57 CEST 2013,
               To: Sat Nov 30 21:53:57 CET 2013]
  Issuer: CN=Doudoux Jean-Michel, OU=Developpement, O=jmdoudoux.fr, L=Pont a Mousson,
  ST=Lorraine, C=FR
  SerialNumber: [    51bd99b8]
```

La méthode generateCertPath() permet de créer et initialiser une instance de type CertPath. Elle possède plusieurs surcharges :

- final CertPath generateCertPath(InputStream inStream)
- final CertPath generateCertPath(InputStream inStream, String encoding)
- final CertPath generateCertPath(List certificates)

Exemple :

```
package fr.jmdoudoux.dej.securite;

import java.io.FileInputStream;
import java.io.IOException;
import java.security.cert.CertPath;
import java.security.cert.Certificate;
import java.security.cert.CertificateFactory;
import java.util.ArrayList;
import java.util.List;

public class TestCertificateFactory {

  public static void main(String[] args) {
    FileInputStream in = null;
    FileInputStream in2 = null;

    try {
      CertificateFactory cf = CertificateFactory.getInstance("X.509");
      in = new FileInputStream("C:/java/moncertificat.cer");
      Certificate cert = cf.generateCertificate(in);
      // System.out.println(cert);

      List<Certificate> listeCert = new ArrayList<Certificate>();
      listeCert.add(cert);

      in2 = new FileInputStream("C:/java/montrustcert.cer");
      Certificate cert2 = cf.generateCertificate(in2);
      listeCert.add(cert2);

      CertPath cp = cf.generateCertPath(listeCert);
      System.out.println(cp);
    } catch (Exception e) {
      e.printStackTrace();
    } finally {
      try {
        in.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
      try {
        in2.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
}
```

La méthode getCertPathEncodings() renvoie les encodages supportés par la fabrique de certificats. Le premier encodage est l'encodage par défaut.

final Iterator getCertPathEncodings()

27.18. L'interface java.security.spec.KeySpec et ses implémentations

L'interface java.security.spec.KeySpec est un marqueur pour toutes les implémentations de clés transparentes.

La classe java.security.spec.DSAPrivateKeySpec encapsule une clé privée DSA.

Plusieurs méthodes permettent d'obtenir la clé privée (x) et les paramètres utilisés pour déterminer la clé (un nombre premier (p), un second nombre premier (q) et la base (g)) :

BigInteger getX()
BigInteger getP()
BigInteger getQ()
BigInteger getG()

La classe java.security.spec.DSAPublicKeySpec encapsule une clé publique DSA.

Plusieurs méthodes permettent d'obtenir la clé publique (y) et les paramètres utilisés pour déterminer la clé (un nombre premier (p), un second nombre premier (q) et la base (g)) :

BigInteger getY()
BigInteger getP()
BigInteger getQ()
BigInteger getG()

La classe java.security.spec.RSAPrivateKeySpec encapsule une clé privée RSA.

Deux méthodes permettent d'obtenir le modulo (n) et l'exposant (d) qui permettent de construire la clé :

BigInteger getModulus()
BigInteger getPrivateExponent()

La classe java.security.spec.RSAPrivateCrtKeySpec encapsule une clé privée RSA telle que précisé dans le standard PKCS #1. Elle hérite de la classe RSAPrivateKeySpec.

Plusieurs méthodes permettent d'obtenir l'exponent public (e) et les entiers qui composent le CRT (Chinese Remainder Theorem : le prime factor (p) du modulus (n), le prime factor (q) de n, l'exponent d mod (p-1), l'exponent d mod (q-1), et le coefficient du CRT (inverse de q) mod p) qui sont utilisés pour construire la clé :

BigInteger getPublicExponent()
BigInteger getPrimeP()
BigInteger getPrimeQ()
BigInteger getPrimeExponentP()
BigInteger getPrimeExponentQ()
BigInteger getCrtCoefficient()

La classe java.security.spec.RSAPublicKeySpec encapsule une clé publique RSA.

Deux méthodes fournissent le modulo (n) et l'exposant (d) qui permettent de construire la clé :

BigInteger getModulus()
BigInteger getPrivateExponent()

27.19. La classe java.security.spec.EncodedKeySpec et ses sous-classes

La classe abstraite java.security.spec.EncodedKeySpec encapsule une clé publique ou privée de manière encodée.

La méthode getEncoded() permet d'obtenir la clé encodée :

- abstract byte[] getEncoded();

La méthode getFormat() renvoie le format d'encodage :

- abstract String getFormat();

Cette classe possède deux classes filles : PKCS8EncodedKeySpec et X509EncodedKeySpec.

La classe java.security.spec.X509EncodedKeySpec encapsule une clé publique encodée selon le standard X.509.

La méthode getFormat() renvoie "X.509".

27.19.1. La classe java.security.spec.PKCS8EncodedKeySpec

La classe java.security.spec.PKCS8EncodedKeySpec encapsule une clé publique encodée selon le standard PKCS #8.

La méthode getFormat() renvoie "PKCS#8".

Il est possible d'utiliser OpenSSL pour générer une clé au format PKCS8. OpenSSL (Win32OpenSSL-1_0_1e.exe) peut être téléchargé sur le site www.openssl.org/related/binaries.html

Il faut générer une clé RSA

Résultat :

```
C:\OpenSSL-Win32\bin>openssl genrsa -out c:\java\key.pem 1024
WARNING: can't open config file: /usr/local/ssl/openssl.cnf
Loading 'screen' into random state - done
Generating RSA private key, 1024 bit long modulus
.................++++++
...........++++++
unable to write 'random state'
e is 65537 (0x10001)
```

L'exemple ci-dessus génère une clé RSA de 1024 bits.

Il faut ensuite convertir la clé RSA au format DER avec PKCS#8.

Résultat :

```
C:\OpenSSL-Win32\bin>openssl pkcs8 -topk8 -in "c:\java\key.pem"  -outform DER
 -out c:\java\key.pkcs8
WARNING: can't open config file: /usr/local/ssl/openssl.cnf
Enter Encryption Password:
Verifying - Enter Encryption Password:
Loading 'screen' into random state - done
unable to write 'random state'
```

L'exemple ci-dessous va lire le contenu de la clé pkcs8 et l'utiliser pour signer des données.

Exemple :

```
package fr.jmdoudoux.dej.securite;

import java.io.ByteArrayOutputStream;
import java.io.FileInputStream;
import java.security.KeyFactory;
import java.security.PrivateKey;
import java.security.Signature;
import java.security.spec.EncodedKeySpec;
import java.security.spec.PKCS8EncodedKeySpec;

public class TestPKCS8EncodedKeySpec {

  public static void main(String[] args) {
    PrivateKey privateKey = null;
    FileInputStream privateKeyIs = null;
    try {
      privateKeyIs = new FileInputStream("C:/java/key.pkcs8");

      ByteArrayOutputStream baos = new ByteArrayOutputStream();
      byte[] buffer = new byte[1024];
      int len;
      while ((len = privateKeyIs.read(buffer)) >= 0)
        baos.write(buffer, 0, len);
      privateKeyIs.close();
      baos.close();
      byte[] privateKeyByte = baos.toByteArray();

      KeyFactory keyFactory = KeyFactory.getInstance("RSA");
      EncodedKeySpec privateKeySpec = new PKCS8EncodedKeySpec(privateKeyByte);
      privateKey = keyFactory.generatePrivate(privateKeySpec);

      byte[] donneesSignees = null;
      byte[] donnees = "mes donnees".getBytes();
      Signature signature = Signature.getInstance("SHA1withRSA");
      signature.initSign(privateKey);
      signature.update(donnees);
      donneesSignees = signature.sign();
      System.out.println(ConvertionHelper.bytesToHex(donneesSignees));
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

27.20. L'interface java.security.spec.AlgorithmParameterSpec

L'interface java.security.spec.AlgorithmParameterSpec est un marqueur pour des classes qui encapsulent les paramètres d'algorithmes cryptographiques.

De nombreuses classes du JDK implémentent l'interface AlgorithmParameterSpec : DSAParameterSpec, DSAPrivateKeySpec, DSAPublicKeySpec, ECFieldF2m, ECFieldFp, ECGenParameterSpec, ECParameterSpec, ECPoint, ECPrivateKeySpec, ECPublicKeySpec, EllipticCurve, EncodedKeySpec, MGF1ParameterSpec, PKCS8EncodedKeySpec, PSSParameterSpec, RSAKeyGenParameterSpec, RSAMultiPrimePrivateCrtKeySpec, RSAOtherPrimeInfo, RSAPrivateCrtKeySpec, RSAPrivateKeySpec, RSAPublicKeySpec, X509EncodedKeySpec, ...

27.20.1. La classe java.security.spec.DSAParameterSpec

La classe java.security.spec.DSAParameterSpec encapsule les paramètres pour un algorithme de type DSA.

Elle possède trois méthodes :

Méthode	Rôle
BigInteger getG()	Renvoyer la base G
BigInteger getP()	Renvoyer le nombre premier P
BigInteger getQ()	Renvoyer le nombre premier Q

Elle possède un constructeur qui attend en paramètres les trois valeurs de type BaseInteger.

28. JCE (Java Cryptography Extension)

Chapitre 28

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

L'API JCE (Java Cryptography Extension) est une extension de JCA qui lui ajoute des API pour l'encryptage et le décryptage, la génération de clés et l'authentification de messages avec des algorithmes de type MAC.

A l'origine, JCE a été développée comme une extension du JDK 1.2. Avant Java 1.4, JCE était diffusée de manière séparée. JCE est intégré à Java SE à partir de la version 1.4.

Depuis que le JCE est fourni avec le JDK, les deux API peuvent sembler moins distinctes d'autant que JCE s'appuie sur JCA. JCE repose sur un principe de conception senblable à celui de JCA : sa mise en oeuvre requiert l'utilisation d'une implémentation d'un fournisseur.

L'implémentation de Sun/Oracle fournie par défaut avec Java est nommée «SunJCE».

Le framework JCE (Java Cryptography Extension) sert de base pour l'implémentation de certaines fonctionnalités notamment :

- des algorithmes de chiffrement/déchiffrement symétrique et asymétrique travaillant par flux ou par blocs
- la génération de clés,
- des algorithmes de type MAC (Message Authentification Code).
- la gestion de dépôts de clés
- les objets scellés
- les signatures digitales
- Password Based Encryption (PBE)

Les classes de l'API JCE sont contenues dans le package javax.crypto qui contient plusieurs classes et interfaces :

Classe/Interface	Rôle
SecretKey	Interface qui définit les fonctionnalités d'une clé secrète
Cipher	Proposer des fonctionnalités de chiffrement/déchiffrement
CipherInputStream	Implémenter le concept de flux sécurisé qui combine un objet de type InputStream et un objet de type Cipher pour décrypter des données en les lisant
CipherOutputStream	Implémenter le concept de flux sécurisé qui combine un objet de type OutputStream et un objet de type Cipher pour crypter des données en les écrivant
EncryptedPrivateKeyInfo	Implémenter le type correspondant définit dans PKCS #8.
KeyAgreement	Implémenter des fonctionnalités d'échanges de clés (key agreement)
KeyGenerator	Générer des clés secrètes pour algorithmes symétriques
Mac	Proposer des fonctionnalités de type "Message Authentication Code" (MAC)
SealedObject	Encapsuler de manière cryptée le résultat de la sérialisation d'un objet
SecretKeyFactory	C'est une fabrique qui permet de convertir des clés opaques (instance de type java.security.Key) en clés transparentes (de type KeySpec) et vice versa

L'API JCE doit être implémentée par des fournisseurs (Cryptographic Service Providers). Chaque fournisseur doit implémenter le SPI (Service Provider Interface) qui définit les fonctionnalités à proposer. L'architecture de JCE permet d'enregistrer des implémentations en vu de leur utilisation.

Le JDK 7.0 est fourni avec plusieurs implémentations de différents fournisseurs essentiellement pour des raisons historiques et par type de services proposés :

- Sun fournie à partir de Java 1.1
- SunRsaSign fournie à partir de Java 1.3
- SunJSSE fournie à partir de Java 1.4
- SunJCE fournie à partir Java 5
- SunPKCS11 fournie à partir Java 5
- SunMSCAPI fournie à partir de Java 6
- SunPCSC fournie à partir de Java 6
- SunEC fournie à partir de Java 7
- SunSASL

En raison de la législation de certains pays, quelques algorithmes ne disposent que d'une implémentation limitée.

La première implémentation nommée « Sun » fournie avec le JDK propose différentes fonctionnalités :

Fonctionnalités	Implémentation
Fabrique de certificats	X.509
Dépôt de clés	JKS
Message Digest	MD2, MD5, SHA-1, SHA-256, SHA-384, SHA-512
(SecureRandom)	SHA1PRNG
Signature	NONEwithDSA, SHA1withDSA

L'implémentation « SunJCE » fournie avec le JDK propose différentes fonctionnalités :

Fonctionnalités	Implémentation
Algorithme de chiffrement symétrique	DES (56 bits), AES, RC2, RC4 and RC5, IDEA, Triple DES (112 bits), Blowfish (56 bits), PBEWithMD5AndDES, PBEWithHmacSHA1AndDESede, DES ede
Mode d'encryption	Electronic Code Book (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), et Propagating Cipher Block Chaining (PCBC)
Algorithme de chiffrement asymétrique	RSA
Algorithme d'échange de clés	Diffie-Hellman (1024 bits)
Mac	HmacMD5, HmacSHA1, HmacSHA256, HmacSHA384, HmacSHA512
Dépôt de clés (KeyStore)	JCEKS

Ce chapitre contient plusieurs sections :

♦ La classe javax.crypto.KeyGenerator
♦ La classe javax.crypto.SecretKeyFactory
♦ La classe javax.crypto.Cipher
♦ Les classes javax.crypto.CipherInputStream et javax.crypto.CipherOutputStream
♦ La classe javax.crypto.SealedObject
♦ La classe javax.crypto.Mac

28.1. La classe javax.crypto.KeyGenerator

La classe javax.crypto.KeyGenerator, fournie à partir de Java 1.4, permet de générer des clés utilisables par des algorithmes symétriques.

Pour obtenir une instance de type KeyGenerator, il faut invoquer la méthode getInstance() qui est une fabrique attendant en paramètre le nom de l'algorithme. Deux autres surcharges, permettent aussi de préciser le fournisseur de l'algorithme à utiliser.

- static KeyGenerator getInstance(String algorithm)
- static KeyGenerator getInstance(String algorithm, String provider)
- static KeyGenerator getInstance(String algorithm, Provider provider)

La clé générée est utilisable par l'algorithme dont le nom est fourni en paramètre. Les noms des algorithmes fournis en standard peuvent être : AES, Blowfish, DES, DESede, HmacMD5 et HmacSHA1

Avant d'obtenir une clé, l'instance de type KeyGenerator doit être initialisée en invoquant la méthode init().

La clé peut être générée de deux manières dont la principale différence est l'initialisation de l'objet :

- de manière indépendante de l'algorithme
- de manière dépendante et spécifique à l'algorithme

Quelque soit l'algorithme cible, ils ont toujours au moins deux paramètres :

- la taille de la clé (keysize)
- une source pour la génération de nombres aléatoires sous la forme d'un objet de type SecureRandom

Plusieurs surcharges de la méthode init() permettent de fournir l'un, l'autre ou ces deux paramètres.

- public void init(int keysize);
- public void init(SecureRandom random);
- public void init(int keysize, SecureRandom random);

Si l'algorithme nécessite d'autres paramètres, il faut utiliser deux autres surcharges qui attendent un paramètre de type AlgorithmParameterSpec

- public void init(AlgorithmParameterSpec params);
- public void init(AlgorithmParameterSpec params, SecureRandom random);

Si l'instance de type KeyGenerator n'est pas initialisée, chaque implémentation des fournisseurs doit obligatoirement offrir une initialisation par défaut.

Une fois initialisée, il est possible d'invoquer la méthode generateKey() qui permet de demander la génération d'une clé en utilisant l'algorithme configuré :

- public SecretKey generateKey();

La première surcharge permet de définir la taille de la clé : cette taille est dépendante de l'algorithme. Chaque implémentation de la plate-forme Java doit obligatoirement fournir une implémentation pour les algorithmes suivants :

- AES taille de la clé 128
- DES taille de la clé 56
- DESede taille de la clé 168
- HmacSHA1
- HmacSHA256

Exemple (code Java 1.4) :

```
package fr.jmdoudoux.dej.securite;
```

```
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;

public class TestKeyGeneratorDESede {

  public static void main(String[] args) {

    KeyGenerator keyGen;
    try {
      keyGen = KeyGenerator.getInstance("DESede");
      keyGen.init(168);
      SecretKey cle = keyGen.generateKey();
      System.out.println("cle (" + cle.getAlgorithm() + "," + cle.getFormat()
          + ") : " + new String(cle.getEncoded()));
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Si la taille fournie n'est pas supportée par l'implémentation de l'algorithme alors une exception de type InvalidParameterException est levée.

Résultat :

```
java.security.InvalidParameterException: Wrong keysize: must be equal to 56
        at com.sun.crypto.provider.DESKeyGenerator.engineInit(DESKeyGenerator.java:90)
        at javax.crypto.KeyGenerator.init(KeyGenerator.java:501)
        at fr.jmdoudoux.dej.securite.TestKeyGeneratorDES.main(TestKeyGeneratorDES.java:15)
```

La seconde surcharge attend une instance de type SecureRandom qui permet de fournir la source pour la génération de nombres aléatoires.

Exemple (code Java 1.4) :

```
package fr.jmdoudoux.dej.securite;

import java.security.SecureRandom;

import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;

public class TestKeyGeneratorDES {

  public static void main(String[] args) {

    KeyGenerator keyGen;
    try {
      keyGen = KeyGenerator.getInstance("DES");
      keyGen.init(new SecureRandom());
      SecretKey cle = keyGen.generateKey();
      System.out.println("cle (" + cle.getAlgorithm() + "," + cle.getFormat()
          + ") : " + new String(cle.getEncoded()));
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

La troisième surcharge attend en paramètre la taille de la clé et une instance de type SecureRandom.

Exemple (code Java 1.4) :

```
package fr.jmdoudoux.dej.securite;

import java.security.SecureRandom;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;

public class TestKeyGeneratorDES {

  public static void main(String[] args) {

    KeyGenerator keyGen;
    try {
      keyGen = KeyGenerator.getInstance("DES");
      keyGen.init(56, new SecureRandom());
      SecretKey cle = keyGen.generateKey();
      System.out.println("cle (" + cle.getAlgorithm() + "," + cle.getFormat()
          + ") : " + new String(cle.getEncoded()));
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

L'initialisation du générateur de manière indépendante de l'algorithme utilise une des deux surcharges de la méthode init() qui attend en paramètre un objet de type AlgorithmParameterSpec.

Si le KeyGenerator n'est pas initialisé, le fournisseur de l'algorithme utilisé doit proposer des valeurs par défaut pour chaque paramètre.

Exemple (code Java 1.4) :

```
package fr.jmdoudoux.dej.securite;

import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;

public class TestKeyGeneratorDES {

  public static void main(String[] args) {

    KeyGenerator keyGen;
    try {
      keyGen = KeyGenerator.getInstance("DES");
      SecretKey cle = keyGen.generateKey();
      System.out.println("cle (" + cle.getAlgorithm() + "," + cle.getFormat()
          + ") : " + new String(cle.getEncoded()));
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

La classe KeyGenerator est utilisable pour les différents algorithmes proposés par les implémentations du ou des fournisseurs enregistrées.

Exemple (code Java 1.4) :

```
package fr.jmdoudoux.dej.securite;

import java.security.NoSuchAlgorithmException;

import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;

public class TestKeyGenerator {

  public static void main(String[] args) {
    KeyGenerator keyGen;
    try {
      System.out.println("Generation d'une cle pour DES");
      keyGen = KeyGenerator.getInstance("DES");
      SecretKey key = keyGen.generateKey();
      System.out
          .println("cle=" + ConvertionHelper.bytesToHex(key.getEncoded()));

      System.out.println("Generation d'une cle pour Blowfish");
      keyGen = KeyGenerator.getInstance("Blowfish");
      key = keyGen.generateKey();
      System.out
          .println("cle=" + ConvertionHelper.bytesToHex(key.getEncoded()));

      System.out.println("Generation d'une cle pour Triple DES");
      keyGen = KeyGenerator.getInstance("DESede");
      key = keyGen.generateKey();
      System.out
          .println("cle=" + ConvertionHelper.bytesToHex(key.getEncoded()));

      System.out.println("Generation d'une cle pour AES");
      keyGen = KeyGenerator.getInstance("AES");
      key = keyGen.generateKey();
      System.out
          .println("cle=" + ConvertionHelper.bytesToHex(key.getEncoded()));

    } catch (NoSuchAlgorithmException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
Generation d'une cle pour DES
cle=FE3731A82052A27A
Generation d'une cle pour Blowfish
cle=83F75B5BBB336632C1A27E9A7211966D
Generation d'une cle pour Triple DES
cle=E6169E9249B0F746801AADE9D6AD85BFCBE05E4AC19EC716
Generation d'une cle pour AES
cle=B71822FA122A3EA57281B2EB9517D518
```

28.2. La classe javax.crypto.SecretKeyFactory

La classe javax.crypto.SecretKeyFactory, ajoutée à Java 1.4, est une fabrique qui permet de convertir des clés opaques (instance de type java.security.Key) en clés transparentes (de type KeySpec) et vice versa.

Une clé opaque (java.security.Key et ses classes filles java.security.PublicKey, java.security.PrivateKey et javax.crypto.SecretKey) ne permet pas de connaitre son implémentation.

A la différence de la classe KeyFactory qui est utilisable avec des paires de clés publiques et privées, la classe SecretFactory n'est utilisable qu'avec des clés privées pour algorithmes symétriques.

Il est nécessaire de consulter la documentation du fournisseur de l'implémentation pour connaître les clés transparentes supportées par les méthodes generateSecret() et getKeySpec(). Par exemple, l'implémentation SunJCE propose un support de la classe DESKeySpec pour les clés de l'algorithme DES et DESedeKeySpec pour les clés de l'algorithme Triple DES.

Chaque implémentation du JDK doit obligatoirement fournir un support des algorithmes DES et DESede par la SecretKeyFactory.

La méthode static getInstance() permet d'obtenir une instance de type SecretFactory.

La méthode generateSecret() permet d'obtenir une clé opaque correspondant à la clé transparente fournie en paramètre :

- SecretKey generateSecret(KeySpec keySpec)

La méthode getKeySpec() permet d'obtenir une clé transparente à partir de la clé opaque fournie en paramètre.

- KeySpec getKeySpec(Key key, Class keySpec)

Le paramètre keySpec permet d'indiquer le type de la classe de retour, par exemple la classe DESKeySpec.

Exemple :

```
package fr.jmdoudoux.dej.securite;

import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import java.security.spec.InvalidKeySpecException;

import javax.crypto.SecretKey;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.DESKeySpec;

public class TestSecretKeyFactory {

  public static void main(String[] args) {
    byte[] desKeyData = { (byte) 0x04, (byte) 0x01, (byte) 0x07, (byte) 0x04,
        (byte) 0x02, (byte) 0x08, (byte) 0x02, (byte) 0x01 };
    DESKeySpec desKeySpec;
    try {
      desKeySpec = new DESKeySpec(desKeyData);
      SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("DES");
      SecretKey secretKey = keyFactory.generateSecret(desKeySpec);
      System.out.println("cle secrete : "
          + ConvertionHelper.bytesToHex(secretKey.getEncoded()));
      System.out.println("algorithme  : " + secretKey.getAlgorithm());
      System.out.println("format      : " + secretKey.getFormat());
    } catch (InvalidKeyException e) {
      e.printStackTrace();
    } catch (NoSuchAlgorithmException e) {
      e.printStackTrace();
    } catch (InvalidKeySpecException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
cle secrete : 0401070402080201
algorithme  : DES
format      : RAW
```

La classe SecretFactory renvoie des implémentations dépendantes du fournisseur utilisé.

28.3. La classe javax.crypto.Cipher

La classe Cipher permet d'utiliser des fonctionnalités de chiffrement et de déchiffrement de données selon un algorithme donné parmi ceux proposés par les fournisseurs. Le chiffrement utilise une clé pour transformer des données en clair en données chiffrées. Le déchiffrement est l'opération inverse.

Elle possède plusieurs méthodes :

Méthode	Rôle
getInstance()	Renvoyer une instance de l'objet pour un algorithme particulier dont l'implémentation est celle fournie par le fournisseur précisé
init()	Initialiser la classe pour le mode de fonctionnement précisé (Cipher.ENCRYPT_MODE et Cipher.DECRYPT_MODE)
update()	Ajouter des données partielles à traiter par la classe

doFinal()	Ajouter la dernière partie des données à traiter et générer le résultat
getBlockSize()	Retourner la taille du bloc utilisé par la classe pour ses traitements
getAlgorithm()	Retourner l'algorithme utilisé par la classe pour ses traitements
getProvider()	Retourner le fournisseur de l'algorithme utilisé par la classe pour ses traitements

28.3.1. La création d'une instance de type Cipher

Pour créer une instance de la classe javax.crypto.Cipher, il faut invoquer sa méthode getInstance() qui est une fabrique qui possède trois surcharges : les trois attendent en paramètre le nom de la transformation à utiliser. Les deux dernières attendent aussi le fournisseur. La première va tenter de trouver une implémentation dans l'ordre de préférence d'enregistrement.

- public static Cipher getInstance(String transformation);
- public static Cipher getInstance(String transformation, String provider);
- public static Cipher getInstance(String transformation, Provider provider);

Quelque soit la version surchargée de la méthode getInstance() invoquée, il faut lui passer en paramètre la transformation à utiliser. La transformation est une chaîne de caractères qui décrit une ou plusieurs opérations à exécuter pour chiffrer ou déchiffrer les données. Le nom de la transformation peut prendre plusieurs formes :

- Algorithme_de_chiffrement
- Algorithme_de_chiffrement/mode/padding_scheme

Exemple :

"DES"

"DES/CBC/PKCS5Padding"

Exemple (code Java 1.4) :

```
Cipher cipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
```

L'exemple ci-dessus demande l'utilisation de l'algorithme DES, en utilisant le mode ECB (Electronic CodeBook) et le style de padding PKCS-5.

Plusieurs algorithmes sont utilisables avec les implémentations fournies dans le JDK 7 de Sun/Oracle : AES, AESWrap, ARCFOUR, Blowfish, CCM, DES, DESede, DESedeWrap, ECIES, GCM, PBEWith<digest>And<encryption>, RC2, RC4, RC5, RSA.

Si la transformation contient simplement un nom, JCE va tenter de trouver une implémentation correspondant au nom et prendre celle qui est préférée dans la configuration.

Le mode permet de préciser comment les données vont être chiffrées. Il existe différents modes :

- CBC : Cipher Block Chaining (défini dans le FIPS PUB 81)
- CFB, CFBn : Cipher FeedBack (défini dans le FIPS PUB 81). n est la taille optionnelle en bits du bloc
- CTR : version simplifiée du mode OFB
- CTS : Cipher Text Stealing
- ECB : Electronic CookBook (défini dans le FIPS PUB 81)
- NONE : aucun mode
- OFB, OFBn : Output FeedBack (défini dans FIPS PUB 81). n est la taille optionnelle en bits du bloc
- PCBC : Propagating Cipher Block Chaining (défini dans Kerberos V4)

Lorsque la transformation implique une utilisation par bloc (par exemple avec le mode CFB ou OFB), il est possible de préciser la taille du bloc en accolant le nombre de bits au mode.

Exemple :

DES/CFB8/NoPadding

DES/OFB32/PKCS5Padding

Si la taille du bloc n'est pas précisée alors la taille par défaut définie par le fournisseur est utilisée (par exemple avec SunJCE, la taille par défaut est de 64 bits).

Le padding permet de préciser comment sera rempli le dernier bloc de données à chiffrer. Plusieurs styles de padding existent :

- NoPadding : pas de padding
- ISO10126Padding : défini par le W3C dans le document "XML Encryption Syntax and Processing"
- OAEPPadding
- OAEPWith<digest>And<mgf>Padding : Optimal Asymmetric Encryption Padding avec digest qui est le nom de l'algorithme de type message digest
- PKCS1Padding : PKCS#1
- PKCS5Padding : PKCS#5 défini par les laboratoires RSA en 1993
- SSL3Padding : défini par le protocole SSL version 3.0

Exemple (code Java 1.4) :

```
Cipher cipher = Cipher.getInstance("DES");
Cipher cipher = Cipher.getInstance("DES/CBC/PKCS5Padding");
Cipher cipher = Cipher.getInstance("AES/CTR/NoPadding");
```

Si aucun mode ni padding ne sont précisés dans la transformation, alors ce sont les valeurs par défaut du fournisseur qui sont utilisées. Par exemple, le fournisseur SunJCE utilise le mode ECB par défaut et le padding PKCS5Padding par défaut pour les algorithmes DES, DESede et Blowfish. Ainsi avec le fournisseur SunJCE, les deux exemples ci-dessous sont identiques :

Exemple (code Java 1.4) :

```
Cipher cipher1 = Cipher.getInstance("DES/ECB/PKCS5Padding");
Cipher cipher2 = Cipher.getInstance("DES");
```

28.3.2. L'initialisation de l'instance de type Cipher

L'instance retournée par la fabrique n'est pas directement utilisable sans être initialisée en invoquant la méthode init(). Cette méthode possède de nombreuses surcharges utilisables selon les besoins de l'algorithme et permettant de fournir les informations pour initialiser l'instance :

Plusieurs surcharges de la méthode init() existent :

- public void init(int opmode, Key key)
- public void init(int opmode, Certificate certificate)
- public void init(int opmode, Key key, SecureRandom random)
- public void init(int opmode, Certificate certificate, SecureRandom random)
- public void init(int opmode, Key key, AlgorithmParameterSpec params)
- public void init(int opmode, Key key, AlgorithmParameterSpec params, SecureRandom random);
- public void init(int opmode, Key key, AlgorithmParameters params)
- public void init(int opmode, Key key, AlgorithmParameters params, SecureRandom random)

Le premier paramètre (opmode) est toujours une valeur entière qui précise le mode d'utilisation de l'instance de la classe Cipher. Cette dernière définit quatre constantes pour spécifier le mode d'utilisation :

- ENCRYPT_MODE : chiffrer des données
- DECRYPT_MODE : déchiffrer des données
- WRAP_MODE : chiffrer une clé pour permettre son échange de manière sécurisée
- UNWRAP_MODE : déchiffrer une clé reçue pour obtenir une instance de type java.security.Key

Les autres paramètres (Key, Certificate, AlgorithmParameters, AlgorithmParametersSpec et SecureRandom) permettent de fournir des informations pour initialiser l'instance.

Les implémentations de la classe Cipher doivent proposer des valeurs par défaut pour certains paramètres d'initialisation s'ils ne sont pas explicitement fournis.

Si un objet de type Cipher initialisé pour le chiffrement a besoin de paramètres et qu'aucun d'eux n'est fournis par la méthode init() invoquée alors l'implémentation de la classe Cipher doit fournir des valeurs par défaut libres ou générées aléatoirement.

Si un objet de type Cipher initialisé pour le déchiffrement a besoin de paramètres et qu'aucun d'eux n'est fournis par la méthode init() invoquée alors une exception de type InvalidKeyException ou InvalidAlgorithmParameterException est levée selon l'implémentation.

Certaines valeurs d'initialisation doivent être obligatoirement fournies et cohérentes sous peine qu'une exception de type InvalidKeyException ou InvalidAlgorithmParameterException soit levée.

Les mêmes paramètres d'initialisation que ceux utilisés pour le chiffrement doivent être fournis à l'objet Cipher pour décrypter des données.

L'initialisation d'une instance de type Cipher réinitialise toute son éventuelle configuration.

28.3.3. Le chiffrement et le déchiffrement de données

L'encryptage et le décryptage de données peuvent se faire en une seule opération ou plusieurs opérations : les données peuvent être traitées par une instance de type Cipher en une ou plusieurs fois. Il est par exemple utile de fractionner le traitement des données si l'on ne connaît pas le volume à traiter ou que ce volume est trop important pour tenir en mémoire.

Pour chiffrer ou déchiffrer des données en une seule fois, il faut utiliser une des surcharges de la méthode doFinal() :

- public byte[] doFinal(byte[] input);
- public byte[] doFinal(byte[] input, int inputOffset, int inputLen);
- public int doFinal(byte[] input, int inputOffset, int inputLen, byte[] output);
- public int doFinal(byte[] input, int inputOffset, int inputLen, byte[] output, int outputOffset)

Pour chiffrer ou déchiffrer des données en plusieurs fois, il faut utiliser une des surcharges de la méthode update() pour fournir les données :

- public byte[] update(byte[] input);
- public byte[] update(byte[] input, int inputOffset, int inputLen);
- public int update(byte[] input, int inputOffset, int inputLen, byte[] output);
- public int update(byte[] input, int inputOffset, int inputLen, byte[] output, int outputOffset)

Une fois toutes les données traitées par des appels multiples à la méthode update(), il faut utiliser une des surcharges de la méthode doFinal() pour finaliser le traitement et obtenir le résultat de l'opération :

- public byte[] doFinal();
- public int doFinal(byte[] output, int outputOffset);

La seconde surcharge permet de fournir les dernières données si nécessaire.

La méthode doFinal() prend en compte le padding si nécessaire.

Une invocation de la méthode doFinal() réinitialise l'état de l'instance de type Cipher comme il l'était après l'invocation de la méthode init(). L'instance de type Cipher peut alors être de nouveau utilisée pour chiffrer ou déchiffrer des données avec la même configuration.

Exemple (code Java 1.4) :

```
package fr.jmdoudoux.dej.securite;

import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;

import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.KeyGenerator;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.SecretKey;

public class TestCipherDESede {

  public static void main(String[] args) {

    final String message = "Mon message a traiter";

    KeyGenerator keyGen;
    try {
      keyGen = KeyGenerator.getInstance("DESede");
      keyGen.init(168);
      SecretKey cle = keyGen.generateKey();
      System.out.println("cle : " + new String(cle.getEncoded()));

      byte[] enc = encrypter(message, cle);
      System.out.println("texte encrypte : " + new String(enc));

      String dec = decrypter(enc, cle);
      System.out.println("texte decrypte : " + dec);

    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public static byte[] encrypter(final String message, SecretKey cle)
      throws NoSuchAlgorithmException, NoSuchPaddingException,
      InvalidKeyException, IllegalBlockSizeException, BadPaddingException {
    Cipher cipher = Cipher.getInstance("DESede");
    cipher.init(Cipher.ENCRYPT_MODE, cle);
    byte[] donnees = message.getBytes();

    return cipher.doFinal(donnees);
  }

  public static String decrypter(final byte[] donnees, SecretKey cle)
      throws NoSuchAlgorithmException, NoSuchPaddingException,
      InvalidKeyException, IllegalBlockSizeException, BadPaddingException {
    Cipher cipher = Cipher.getInstance("DESede");
    cipher.init(Cipher.DECRYPT_MODE, cle);

    return new String(cipher.doFinal(donnees));
  }

}
```

Résultat :

```
cle : ›_QþÐ)_ãÖ_Úh†...Jý@ÇÈkµûÚJ
texte encrypte : texte encrypte : ³Ù_ó"ú‰f0Z&õ¯?¤_yö×_Ù¿ÛÂ
texte decrypte : Mon message a traiter
```

La méthode getOutputSize(int taille) retourne la taille des données encryptées quand on lui fournit en paramètre celle des données non chiffrées. L'utilisation de cette méthode est pratique pour permettre de définir la taille du buffer qui contiendra les données chiffrées ou déchiffrées.

28.3.4. Les modes wrap et unwrap

La classe Cipher permet d'envelopper une clé pour permettre son transfert ou son stockage de manière sécurisée.

Le mode wrap permet de chiffrer le format encodé d'une clé : la clé ainsi chiffrée peut être stockée en étant à l'abri des regards indiscrets. Le mode unwrap permet le déchiffrement de données préalablement obtenues en utilisant le mode wrap. Ces modes permettent de sécuriser les échanges de clés.

Pour encrypter une clé, il faut initialiser l'objet Cipher avec le mode WRAP_MODE et invoquer la méthode wrap() en lui passant en paramètre un objet de type Key qui est la clé à traiter.

- public final byte[] wrap(Key key)

Pour permettre d'extraire une clé d'une enveloppe, il est nécessaire d'avoir le nom de l'algorithme utilisé et le type de clé enveloppée.

Il faut initialiser l'instance de type Cipher avec le mode UNWRAP_MODE et invoquer la méthode unwrap().

- public final Key unwrap(byte[] wrappedKey, String wrappedKeyAlgorithm, int wrappedKeyType))

Le paramètre wrappedKey correspond à l'enveloppe de la clé (créée en invoquant la méthode wrap()).

Le paramètre wrappedKeyAlgorithm est le nom de l'algorithme.

Le paramètre wrappedKeytype est le type de l'enveloppe (Cipher.SECRET_KEY, Cipher.PRIVATE_KEY ou Cipher.PUBLIC_KEY).

Exemple :

```
package fr.jmdoudoux.dej.securite;

import java.security.Key;

import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;

public class TestCipherWrap {

  public static void main(String[] args) {

    try {
      // génération de la clé à envelopper
      KeyGenerator generator = KeyGenerator.getInstance("AES");
      generator.init(128);
      SecretKey cleAEnvelopper = generator.generateKey();
      System.out.println("cle          : "
          + ConvertionHelper.bytesToHex(cleAEnvelopper.getEncoded()));

      // wrap de la clé
      Cipher cipher = Cipher.getInstance("AES/ECB/NoPadding");
      KeyGenerator KeyGen = KeyGenerator.getInstance("AES");
      KeyGen.init(128);
      Key clePourChiffrer = KeyGen.generateKey();
      cipher.init(Cipher.WRAP_MODE, clePourChiffrer);
      byte[] cleEnveloppee = cipher.wrap(cleAEnvelopper);
      System.out.println("cle wrapped  : "
          + ConvertionHelper.bytesToHex(cleEnveloppee));

      // unwrap de la clé
      cipher.init(Cipher.UNWRAP_MODE, clePourChiffrer);
      Key key = cipher.unwrap(cleEnveloppee, "AES", Cipher.SECRET_KEY);
      System.out.println("cle unwrapped: "
          + ConvertionHelper.bytesToHex(key.getEncoded()));

    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
cle          : 88694775945182C05A7BAA7ED90C0708
cle wrapped  : 3CF655ABEE9135F4D766CDDE0A80622C
cle unwrapped: 88694775945182C05A7BAA7ED90C0708
```

Dans l'exemple ci-dessus, il est possible de ne pas utiliser de padding car la taille de la clé est un multiple de la taille du bloc.

28.3.5. Les paramètres des algorithmes de chiffrement

Le chiffrement peut se faire en utilisant deux modes :

- block : le chiffrement se fait par bloc. Chaque bloc est traité dans son intégralité : s'il n'y a pas assez de données pour remplir le dernier bloc, les octets manquants doivent être complétés. Cette opération est nommée padding : plusieurs types de padding existent
- stream : le chiffrement se fait octet par octet. Les données peuvent être de taille arbitraire sans avoir à réaliser un padding

Lors d'un chiffrement simple, deux blocs de données identiques donneront les mêmes données une fois chiffrés. Ceci facilite le travail des crypto-analystes lorsque des blocs de données se répètent. Pour l'éviter et améliorer la complexité d'un déchiffrement forcé, des modes ont été définis pour utiliser les données du bloc précédent pour modifier le bloc en cours de traitement avant son chiffrement. Le premier bloc a alors besoin d'une valeur initiale qui est appelée Initialisation Vector (IV). Les données de l'IV sont simplement utilisées pour modifier les données du premier bloc : cette valeur peut être obtenue aléatoirement et ne doit pas nécessairement être gardée de manière secrète.

Certains algorithmes comme RSA ou AES peuvent utiliser des clés de tailles différentes et d'autres algorithmes comme DES ou Triple DES utilisent des clés de tailles fixes.

Généralement, plus la taille de la clé est importante, plus la résistance aux tentatives de déchiffrement forcé sera grande. Mais plus la taille est importante, plus le temps nécessaire aux algorithmes est important.

La plupart des algorithmes utilisent des clés binaires qui sont difficiles à retenir par des humains : il est plus facile de retenir des mots de passes composés de caractères alphanumériques. C'est la raison pour laquelle le protocole Password Based Encryption (PBE) a été développé pour générer une clé binaire forte à partir d'un mot de passe et de plusieurs paramètres dépendants de l'implémentation (nombre aléatoire, nombre d'itérations, salt, ...). L'utilisation de ces différents paramètres par l'algorithme de l'implémentation permet d'améliorer la génération aléatoire de la clé binaire.

L'API propose plusieurs classes pour encapsuler les différentes valeurs pour les paramètres.

La classe javax.crypto.spec.PBEParameterSpec permet d'encapsuler les paramètres (salt et nombre d'itérations) pour une transformation de type PBE, par exemple PBEWithMD5AndDES.

La classe javax.crypto.spec.IVParameterSpec encapsule un Initialisation Vector (IV) pour les transformations qui en ont besoin. C'est par exemple le cas des algorithmes DES, DESede et Blowfish qui utilisent le mode CBC, CFB, OFB ou PCBC. La méthode getIV() permet d'obtenir le vecteur d'initialisation (initialization vector ou IV).

La méthode getParameters() permet d'obtenir les paramètres utilisés par l'implémentation de l'algorithme utilisé par la classe Cipher. Elle renvoie un objet de type AlgorithmParameters. Elle renvoie null si aucun paramètre n'est utilisé. Par exemple, une transformation de type PBE requiert plusieurs paramètres dont un salt et un nombre d'itérations.

La méthode getEncoded() permet d'obtenir ces paramètres pour les réutiliser pour le chiffrement. Il suffit de fournir les paramètres encodés à la méthode init() de la classe AlgorithmParameters.

Exemple :

```
package fr.jmdoudoux.dej.securite;
```

```
import java.security.AlgorithmParameters;

import javax.crypto.Cipher;
import javax.crypto.SecretKey;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.PBEKeySpec;
import javax.crypto.spec.PBEParameterSpec;

public class TestCipherGetParameters {

  private static final String TRANSFORMATION = "PBEWithMD5AndDES";

  static char[] motDePasse = { 'M', 'o', 't', 'D', 'e', 'P',
      'a', 's', 's', 'e' };

  static byte[] salt = new byte[] { 0x7e, (byte) 0xe0,
      0x41, (byte) 0xf9, 0x4e, (byte) 0xa0, 0x60, 0x02 };

  public static void main(String[] args) {

    try {
      SecretKeyFactory kf = SecretKeyFactory.getInstance(TRANSFORMATION);
      PBEKeySpec keySpec = new PBEKeySpec(motDePasse);
      SecretKey key = kf.generateSecret(keySpec);
      PBEParameterSpec params = new PBEParameterSpec(salt, 1000);

      Cipher cipherEnc = Cipher.getInstance(TRANSFORMATION);
      cipherEnc.init(Cipher.ENCRYPT_MODE, key, params);

      byte[] texteChiffre = cipherEnc.doFinal("mon message".getBytes());
      System.out.println("texte chiffre="
          + ConvertionHelper.bytesToHex(texteChiffre));

      AlgorithmParameters algParams = cipherEnc.getParameters();
      byte[] encodedAlgParams = algParams.getEncoded();

      String texteClair = dechiffrer(key, encodedAlgParams, texteChiffre);
      System.out.println(texteClair);

    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public static String dechiffrer(SecretKey key, byte[] encodedAlgParams,
      byte[] texteChiffre) throws Exception {
    AlgorithmParameters algParamsDec;
    algParamsDec = AlgorithmParameters.getInstance(TRANSFORMATION);
    algParamsDec.init(encodedAlgParams);

    Cipher cipherDec = Cipher.getInstance(TRANSFORMATION);

    cipherDec.init(Cipher.DECRYPT_MODE, key, algParamsDec);
    byte[] texteClair = cipherDec.doFinal(texteChiffre);

    return new String(texteClair);
  }
}
```

Résultat :

```
texte chiffre=9709B1F1310C655F0D75FE20148CECAF
mon message
```

28.3.6. Des exemples d'utilisation d'algorithmes de chiffrement

L'exemple ci-dessous met en oeuvre l'algorithme DES.

Exemple :

```
package fr.jmdoudoux.dej.securite;

import java.security.InvalidAlgorithmParameterException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;

import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.KeyGenerator;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.SecretKey;

public class TestCipherDES {

  public static void main(String[] args) {

    try {
      KeyGenerator keyGen = KeyGenerator.getInstance("DES");
      SecretKey secretKey = keyGen.generateKey();
      String message = "Mon message à chiffer";

      utiliserCipher(secretKey, "DES", message);
      utiliserCipher(secretKey, "DES/ECB/PKCS5Padding", message);
      utiliserCipher(secretKey, "DES/CBC/PKCS5Padding", message);
      utiliserCipher(secretKey, "DES/PCBC/PKCS5Padding", message);
      utiliserCipher(secretKey, "DES/CFB/PKCS5Padding", message);
      utiliserCipher(secretKey, "DES/OFB/PKCS5Padding", message);

    } catch (Exception e) {
      System.out.println("Erreur " + e);
    }
  }

  public static void utiliserCipher(SecretKey secretKey, String transformation,
      String message) throws NoSuchAlgorithmException, NoSuchPaddingException,
      InvalidKeyException, IllegalBlockSizeException, BadPaddingException,
      InvalidAlgorithmParameterException {
    Cipher desCipher = Cipher.getInstance(transformation);
    desCipher.init(Cipher.ENCRYPT_MODE, secretKey);
    byte[] byteCipherText = desCipher.doFinal(message.getBytes());
    System.out.println(ConvertionHelper.bytesToHex(byteCipherText));

    desCipher.init(Cipher.DECRYPT_MODE, secretKey, desCipher.getParameters());
    byte[] byteDecryptedText = desCipher.doFinal(byteCipherText);
    System.out.println(new String(byteDecryptedText));
  }
}
```

Résultat :

```
2A856E495FCC686DB21FCD677CA7F7081BD39FCBD8053913
Mon message à chiffer
2A856E495FCC686DB21FCD677CA7F7081BD39FCBD8053913
Mon message à chiffer
B05B556E44401B60BA7CB43B6BC3307FD33B99D782FFBB4A
Mon message à chiffer
9864AE319B96C2EAF7149CC2E5D196268FD17321D20D93E6
Mon message à chiffer
C3F682D4577CE3EAAA7F2532983CCBDBEEF1A7BC1A3EC6F1
Mon message à chiffer
49375D703EE141E3DD30747531C312D583E14551B00A180D
Mon message à chiffer
```

L'exemple ci-dessous met en oeuvre l'algorithme AES.

Exemple :

```
package fr.jmdoudoux.dej.securite;

import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;

public class TestCipherAES {

  public static void main(String[] args) {

    try {
      KeyGenerator keyGen = KeyGenerator.getInstance("AES");
      keyGen.init(128);
      SecretKey secretKey = keyGen.generateKey();

      Cipher aesCipher = Cipher.getInstance("AES");
      aesCipher.init(Cipher.ENCRYPT_MODE, secretKey);
      byte[] byteCipherText = aesCipher.doFinal("Mon message".getBytes());
      System.out.println(ConvertionHelper.bytesToHex(byteCipherText));

      aesCipher.init(Cipher.DECRYPT_MODE, secretKey, aesCipher.getParameters());
      byte[] byteDecryptedText = aesCipher.doFinal(byteCipherText);
      System.out.println(new String(byteDecryptedText));
    } catch (Exception e) {
      System.out.println("Erreur " + e);
    }
  }
}
```

Résultat :

```
BF1FFEECF2C58AB9CB295D7DB95A18E7
Mon message
```

28.4. Les classes javax.crypto.CipherInputStream et javax.crypto.CipherOutputStream

Le JCE propose le concept de flux sécurisé qui combine :

- un objet de type InputStream ou OutputStream
- un objet de type Cipher

Ces flux sont encapsulés dans les classes CipherInputStream et CipherOutputStream.

La classe CipherInputStream est un FilterInputStream qui est associé à un objet de type Cipher pour permettre, selon sa configuration, de chiffrer ou déchiffrer les données lues.

Selon sa configuration choisie, les méthodes de lecture renvoient les données obtenues par la lecture du flux et traitées par l'instance de type Cipher. Si l'instance de type Cipher est configurée pour décrypter des données, les données lues du flux seront décryptées avant d'être renvoyées.

L'instance de type Cipher doit être complètement initialisée avant d'être utilisée par un CipherInputStream pour commencer la lecture des données.

Les méthodes de lecture de la classe CipherInputStream vont attendre les données retournées par l'instance de type Cipher : c'est notamment le cas si l'instance de type Cipher travaille sur des blocs, il est nécessaire d'obtenir toutes les données d'un bloc de l'InputStream.

Exemple (code Java 1.4) :

```
package fr.jmdoudoux.dej.securite;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;

import javax.crypto.Cipher;
import javax.crypto.CipherInputStream;
import javax.crypto.KeyGenerator;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.SecretKey;

public class TestCipherInputStream {

  public static void main(String[] args) {

    try {
      KeyGenerator keyGen;
      keyGen = KeyGenerator.getInstance("DESede");
      keyGen.init(168);
      SecretKey cle = keyGen.generateKey();
      System.out.println("cle : " + new String(cle.getEncoded()));

      encrypterFichier(cle, "C:/java/test/donnees.txt",
          "C:/java/test/donnees_enc.txt");

      decrypterFichier(cle, "C:/java/test/donnees_enc.txt",
          "C:/java/test/donnees_dec.txt");

    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public static void encrypterFichier(SecretKey cle, String source, String cible)
      throws NoSuchAlgorithmException, NoSuchPaddingException,
      InvalidKeyException {
    encrypterDecrypterFichier(Cipher.ENCRYPT_MODE, cle, source, cible);
  }

  public static void decrypterFichier(SecretKey cle, String source, String cible)
      throws NoSuchAlgorithmException, NoSuchPaddingException,
      InvalidKeyException {
    encrypterDecrypterFichier(Cipher.DECRYPT_MODE, cle, source, cible);
  }

  public static void encrypterDecrypterFichier(int mode, SecretKey cle,
      String source, String cible) throws NoSuchAlgorithmException,
      NoSuchPaddingException, InvalidKeyException {
    Cipher cipher = Cipher.getInstance("DESede");
    cipher.init(mode, cle);

    FileInputStream fis = null;
    FileOutputStream fos = null;
    CipherInputStream cis = null;

    try {
      fis = new FileInputStream(source);
      cis = new CipherInputStream(fis, cipher);
      fos = new FileOutputStream(cible);
      byte[] b = new byte[8];
      int i = cis.read(b);
      while (i != -1) {
        fos.write(b, 0, i);
        i = cis.read(b);
      }
    } catch (IOException ioe) {
      if (fis != null) {
        try {
          fis.close();
        } catch (IOException e) {
          e.printStackTrace();
        }
      }
      if (fos != null) {
        try {
          fos.close();
        } catch (IOException e) {
          e.printStackTrace();
        }
      }
    }
  }
}
```

La classe CipherOutputStream est un FilterOutputStream qui peut encoder ou décoder les données qu'on lui envoie.

Elle encapsule un objet de type OutpuStream et un objet de type Cipher. Les données transmises à la méthode write() sont préalablement traitées par l'instance de type Cipher. Cet objet de type Cipher doit donc être obligatoirement configuré et initialisé avant de traiter le premier octet des données.

La classe CipherOutputStream hérite de la classe FilterOutputStream : elle permet d'encrypter ou de décrypter les données qui transitent par le flux. Elle encapsule une instance de type OutputStream ou une de ses sous-classes et une instance de type Cipher.

Il est important d'invoquer les méthodes flush() et close() de la classe CipherOutputStream pour permettre le traitement de l'intégralité des données. Si l'instance de type Cipher est configurée pour utiliser un algorithme avec bloc, il est nécessaire de lui envoyer toutes les données d'un bloc avant que celles-ci ne soient appliquées à l'OutputStream. Dans ce cas, il est important d'invoquer la méthode flush() pour s'assurer que les dernières données transmises au Cipher soient également appliquées à l'instance de type OutputStream. La méthode close() invoque la méthode doFinal() du Cipher et les méthodes flush() et close() de l'instance de type OutputStream.

28.5. La classe javax.crypto.SealedObject

La classe SealedObject encapsule de manière cryptée le résultat de la sérialisation d'un objet. L'objet encapsulé doit donc implémenter l'interface java.io.Serializable.

Pour créer une instance de type SealedObject, il suffit d'invoquer le constructeur de la classe SealObject en lui passant en paramètre l'objet et une instance de type Cipher correctement initialisée.

Pour décrypter l'objet encapsulé, il suffit d'invoquer la méthode getObject() en lui passant la clé.

Exemple (code Java 1.4) :

```
package fr.jmdoudoux.dej.securite;

import java.io.IOException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;

import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.KeyGenerator;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.SealedObject;
import javax.crypto.SecretKey;

public class TestSealedObject {

  public static void main(String[] args) {

    final Personne personne = new Personne(1, "nom1", "prenom1");

    KeyGenerator keyGen;
    try {
      keyGen = KeyGenerator.getInstance("DESede");
      keyGen.init(168);
      SecretKey cle = keyGen.generateKey();
      System.out.println("cle : " + new String(cle.getEncoded()));

      SealedObject so = encrypter(personne, cle);

      Personne personneDec = decrypter(so, cle);
      System.out.println(personne.equals(personneDec));

    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public static SealedObject encrypter(final Personne personne, SecretKey cle)
      throws NoSuchAlgorithmException, NoSuchPaddingException,
      InvalidKeyException, IllegalBlockSizeException, BadPaddingException,
      IOException {
    Cipher cipher = Cipher.getInstance("DESede");
    cipher.init(Cipher.ENCRYPT_MODE, cle);

    return new SealedObject(personne, cipher);
  }

  public static Personne decrypter(SealedObject so, SecretKey cle)
      throws NoSuchAlgorithmException, NoSuchPaddingException,
      InvalidKeyException, IllegalBlockSizeException, BadPaddingException,
      ClassNotFoundException, IOException {
    Cipher cipher = Cipher.getInstance("DESede");
    cipher.init(Cipher.DECRYPT_MODE, cle);

    Personne resultat = (Personne) so.getObject(cle);
    return resultat;
  }

}
```

Il est aussi possible de fournir en paramètre de la méthode getObject(), une instance de type Cipher correctement initialisée.

Exemple (code Java 1.4) :

```
// ...
  public static Personne decrypter(SealedObject so, SecretKey cle)
    throws NoSuchAlgorithmException, NoSuchPaddingException,
    InvalidKeyException, IllegalBlockSizeException, BadPaddingException,
    ClassNotFoundException, IOException {
    Cipher cipher = Cipher.getInstance("DESede");
    cipher.init(Cipher.DECRYPT_MODE, cle);

    Personne resultat = (Personne) so.getObject(cle);
    return resultat;
  }

  // ...
```

La classe SealedObject permet de créer une version sécurisée par cryptage d'un objet sous une forme sérialisée. L'objet doit implémenter l'interface Serializable.

28.6. La classe javax.crypto.Mac

Une fonction de type Message Authentication Code (MAC) est similaire à un MessageDigest : elle permet de vérifier l'intégrité de données transmises par un moyen non fiable.

Une fonction de type MAC propose un moyen de vérifier l'intégrité de données transmises en calculant une empreinte grâce à des algorithmes mathématiques et une clé secrète. Ces algorithmes reposent sur l'utilisation de fonctions de hachage nommées HMAC qui combinent une fonction de hachage et une clé secrète. Les fonctionnalités de type HMAC sont décrites dans la RFC 2014. Une fonction HMAC est utilisée en cryptographie pour calculer une valeur de hachage grâce à un algorithme de hachage (MD5, SHA-1, ...) et une clé partagée. Il est donc nécessaire de connaitre la clé pour pouvoir utiliser la fonction MAC et ainsi pouvoir vérifier l'intégrité des données reçues.

La classe Mac, ajoutée à Java 1.4, permet de mettre en oeuvre des algorithmes de type MAC. Un cas typique d'utilisation est la vérification de données transmises entre deux parties qui partagent une même clé secrète.

L'émetteur et le récepteur doivent partager la clé à utiliser lors de l'utilisation de la fonction MAC. L'émetteur calcul une valeur de hachage avec un algorithme de type MAC en utilisant la clé partagée. Le récepteur calcule lui aussi la valeur de hachage des données reçues avec le même algorithme de type MAC en utilisant lui aussi la clé partagée. Le récepteur peut alors comparer les deux valeurs de hachage pour vérifier l'intégrité des données reçues et garantir ainsi que ce sont bien celles qui ont été envoyées par l'émetteur.

La méthode static getInstance() est une fabrique qui permet de créer une instance de type Mac mettant en oeuvre l'algorithme dont le nom est fourni en paramètre. Deux autres surcharges permettent de préciser le fournisseur de l'implémentation de l'algorithme à utiliser.

Méthode	Rôle
Mac getInstance(String algorithm)	Fabrique qui renvoie une instance de type Mac pour l'algorithme précisé
Mac getInstance(String algorithm, String Provider)	Fabrique qui renvoie une instance de type Mac pour l'algorithme du provider précisé
Mac getInstance(String algorithm, Provider provider)	Fabrique qui renvoie une instance de type Mac pour l'algorithme du provider précisé

Chaque implémentation de la plate-forme Java a l'obligation de fournir un support de plusieurs algorithmes de type MAC : HmacMD5, HmacSHA1 et HmacSHA256.

D'autres algorithmes peuvent être utilisés selon les implémentations du fournisseur, par exemple :

Nom	Algorithme	Taille de l'empreinte
HmacMD5	HMAC avec la fonction de hachage MD5	128 bits
HmacSHA[1|256|384|512]	HMAC avec la fonction de hachage SHA[1|256|384|512]	160, 256, 384, 512 bits
PBEWith<*mac*>	HMAC initialisé avec PBE <*mac*> est à remplacer par le nom de la fonction de hachage à utiliser (MD5, SHA[1|256|384|512])	

L'instance de type Mac doit être initialisée en utilisant une des deux surcharges de la méthode init() pour fournir la clé secrète sous la forme d'une instance de type Key.

Méthode	Rôle
public void init(Key key);	Fournir la clé
public void init(Key key, AlgorithmParameterSpec params);	Fournir la clé et un objet de type AlgorithParameterSec pour transmettre d'autres paramètres à l'implémentation de l'algorithme utilisé

La clé doit implémenter l'interface SecretKey : elle peut donc être le résultat de l'invocation de la méthode KeyGenerator.generateKey() ou KeyAgreement.generateSecret().

Une valeur MAC peut être obtenue en une seule opération ou en plusieurs étapes, ce qui est pratique si on ne connait pas à l'avance la taille des données à traiter ou si la taille des données ne peut pas être stockée en une seule fois en mémoire.

Les données peuvent être fournies en un ou plusieurs blocs grâce aux méthodes suivantes :

Méthode	Rôle
byte[] doFinal(byte[] input)	Finaliser le calcul de la valeur MAC avec les données fournies en paramètre et retourner l'empreinte
void update(byte input) void update(byte[] input) void update(byte[] input, int inputOffset, int inputLen)	Fournir une partie des données
byte[] doFinal()	Finaliser le calcul de la valeur MAC pour les données déjà fournies et retourner l'empreinte
void doFinal(byte[] output, int outOffset)	Finaliser le calcul de la valeur MAC pour les données déjà fournies et mettre l'empreinte dans le tableau à l'offset précisé en paramètre

La méthode doFinal(byte[]) permet de calculer la valeur MAC en une seule opération.

Exemple :

```
package fr.jmdoudoux.dej.securite;

import java.io.UnsupportedEncodingException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;

import javax.crypto.Mac;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;

public class TestMac {

  public static void main(String[] args) {
    try {
      String resultat = calculerMAC("Mon message", "maCle", "HmacSHA256");
      System.out.println("HmacSHA256 digest : " + resultat);

      resultat = calculerMAC("Mon message", "maCle", "HmacMD5");
      System.out.println("HmacMD5 digest : " + resultat);
    } catch (NoSuchAlgorithmException e) {
      e.printStackTrace();
    } catch (UnsupportedEncodingException e) {
      e.printStackTrace();
    } catch (InvalidKeyException e) {
      e.printStackTrace();
    }
  }

  public static String calculerMAC(String message, String cle, String algorithme)
      throws UnsupportedEncodingException, NoSuchAlgorithmException,
      InvalidKeyException {
    String resultat;

    SecretKey secretKey = new SecretKeySpec(cle.getBytes("UTF-8"), algorithme);
    System.out.println("cle : " + bytesToHex(secretKey.getEncoded()));

    Mac mac = Mac.getInstance(secretKey.getAlgorithm());
    mac.init(secretKey);

    byte[] b = message.getBytes("UTF-8");
    byte[] digest = mac.doFinal(b);

    resultat = bytesToHex(digest);
    return resultat;
  }

  public static String bytesToHex(byte[] b) {
    char hexDigit[] = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A',
        'B', 'C', 'D', 'E', 'F' };
    StringBuffer buf = new StringBuffer();
    for (int j = 0; j < b.length; j++) {
      buf.append(hexDigit[(b[j] >> 4) & 0x0f]);
      buf.append(hexDigit[b[j] & 0x0f]);
    }
    return buf.toString();
  }

}
```

Résultat :

```
cle : 6D61436C65
HmacSHA256 digest : AF9DAA10208DDC5DCD3618C422B4E97AD23A8A3C0D648EB5648D424420FD9B02
cle : 6D61436C65
HmacMD5 digest : DDB1B8B6F8F29578E5D143A5834285DB
```

Plusieurs surcharges de la méthode update() permettent de calculer la valeur MAC en plusieurs invocations. Pour obtenir la valeur MAC, il faut alors invoquer une des deux surcharge de la méthode doFinal() : byte[] doFinal() ou byte[] doFinal(byte[], int).

Exemple :

```
package fr.jmdoudoux.dej.securite;

import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;

import javax.crypto.Mac;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;

public class TestMac {

  public static void main(String[] args) {
    try {
      String resultat = calculerMAC("monfichier.txt", "maCle", "HmacSHA256");
      System.out.println("HmacSHA256 digest : " + resultat);

      resultat = calculerMAC("monfichier.txt", "maCle", "HmacMD5");
      System.out.println("HmacMD5 digest : " + resultat);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public static String calculerMAC(String nomFichier, String cle,
      String algorithme) throws NoSuchAlgorithmException, InvalidKeyException,
      IllegalStateException, IOException {
    String resultat;

    SecretKey secretKey = new SecretKeySpec(cle.getBytes("UTF-8"), algorithme);
    System.out.println("cle : "
        + ConvertionHelper.bytesToHex(secretKey.getEncoded()));

    Mac mac = Mac.getInstance(secretKey.getAlgorithm());
    mac.init(secretKey);

    byte[] buffer = new byte[1024];
    BufferedInputStream in = new BufferedInputStream(new FileInputStream(
        nomFichier));
    int nbLus = 0;
    try {
      while ((nbLus = in.read()) != -1) {
        mac.update(buffer, 0, nbLus);
      }
      byte[] digest = mac.doFinal();
      resultat = ConvertionHelper.bytesToHex(digest);
    } finally {
      try {
        in.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }

    return resultat;
  }

}
```

Résultat :

```
cle : 6D61436C65
HmacSHA256 digest : 7785E591C04ADA3A1628AEA8D429B2368F1BDE3D92288D922522ECAAE62ECB21
cle : 6D61436C65
HmacMD5 digest : 13F3FEEB29B335D988757C81D89713B5
```

A la fin de l'invocation de la méthode doFinal(), l'instance est réinitialisée pour permettre de calculer à nouveau les empreintes de nouvelles données en utilisant l'algorithme et la clé associés.

Plusieurs méthodes permettent d'obtenir des informations :

Méthode	Rôle
String getAlgorithm()	Renvoyer le nom de l'algorithme
int getMacLength()	Retourner la taille de l'empreinte calculée avec l'algorithme
Provider getProvider()	Retourner le fournisseur de l'implémentation de l'algorithme

La méthode reset() permet de réinitialiser l'objet.

29. JNI (Java Native Interface)

Chapitre 29

Niveau : 🟩🟩🟨🟨🟥⬜ Confirmé

JNI est l'acronyme de Java Native Interface. C'est une technologie qui permet d'utiliser du code natif, notamment C, dans une classe Java.

L'inconvénient majeur de cette technologie est d'annuler la portabilité du code Java. En contre-partie cette technologie peut être très utile dans plusieurs cas :

- pour des raisons de performance
- utilisation de composants éprouvés déjà existants

La mise en oeuvre de JNI nécessite plusieurs étapes :

- la déclaration et l'utilisation de la ou des méthodes natives dans la classe Java
- la compilation de la classe Java
- la génération du fichier d'en-tête avec l'outil javah
- l'écriture du code natif en utilisant entre autres les fichiers d'en-tête fournis par le JDK et celui généré précédemment
- la compilation du code natif sous la forme d'une bibliothèque

La bibliothèque est donc dépendante du système d'exploitation pour lequel elle est développée : .dll pour les systèmes de type Windows, .so pour les systèmes de type Unix, ...

Ce chapitre contient plusieurs sections :

- La déclaration et l'utilisation d'une méthode native
- La génération du fichier d'en-tête
- L'écriture du code natif en C
- Le passage de paramètres et le renvoi d'une valeur (type primitif)
- Le passage de paramètres et le renvoi d'une valeur (type objet)

29.1. La déclaration et l'utilisation d'une méthode native

La déclaration dans le code source Java est très facile puisqu'il suffit de déclarer la signature de la méthode avec le modificateur native. Le modificateur permet au compilateur de savoir que cette méthode est contenue dans une bibliothèque native.

Il ne doit pas y avoir d'implémentation même pas un corps vide pour une méthode déclarée native.

Exemple :

```
class TestJNI1 {
  public native void afficherBonjour();

  static {
    System.loadLibrary("mabibjni");
  }

  public static void main(String[] args) {
    new TestJNI1().afficherBonjour();
  }
}
```

Pour pouvoir utiliser une méthode native, il faut tout d'abord charger la bibliothèque. Pour réaliser ce chargement, il faut utiliser la méthode statique loadLibrary() de la classe system et obligatoirement s'assurer que la bibliothèque est chargée avant le premier appel de la méthode native.

Le plus simple pour assurer ce chargement est de le demander dans un morceau de code d'initialisation statique de la classe.

Exemple :

```
class TestJNI1 {
  public native void afficherBonjour();
  static {
    System.loadLibrary("mabibjni");
  }
}
```

Le nom de la bibliothèque fournie en paramètre doit être indépendant de la plate-forme utilisée : il faut préciser le nom de la bibliothèque sans son extension. Le nom sera automatiquement adapté selon le système d'exploitation sur lequel le code Java est exécuté.

L'utilisation de la méthode native dans le code Java se fait de la même façon qu'une méthode classique.

Exemple :

```
class TestJNI1 {
  public native void afficherBonjour();
  static {
    System.loadLibrary("mabibjni");
  }

  public static void main(String[] args) {
    new TestJNI1().afficherBonjour();
  }
}
```

29.2. La génération du fichier d'en-tête

Jusqu'à la version 8 de Java, l'outil javah fourni avec le JDK permet de générer un fichier d'en-tête qui va contenir la définition dans le langage C des fonctions correspondant aux méthodes déclarées natives dans le code source Java.

Javah utilise le bytecode pour générer le fichier .h. Il faut donc que la classe Java soit préalablement compilée.

La syntaxe est donc : javah -jni nom_fichier_sans_extension

Exemple :

```
D:\java\test\jni>dir
03/12/2003  14:39       <DIR>          .
03/12/2003  14:39       <DIR>          ..
03/12/2003  14:39                  230 TestJNI1.java
            2 fichier(s)              230 octets
            2 Rép(s)   2 200 772 608 octets libres
D:\java\test\jni>javac TestJNI1.java
D:\java\test\jni>javah -jni TestJNI1
D:\java\test\jni>dir
Répertoire de D:\java\test\jni
```

```
03/12/2003  14:39       <DIR> .
03/12/2003  14:39       <DIR> ..
03/12/2003  14:39                  459 TestJNI1.class
03/12/2003  14:39                  399 TestJNI1.h
03/12/2003  14:39                  230 TestJNI1.java
            3 fichier(s) 1 088 octets
            2 Rép(s) 2 198 208 512 octets libres
D:\java\test\jni>
```

Le fichier TestJNI1.h généré est le suivant :

Exemple :

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class TestJNI1 */

#ifndef _Included_TestJNI1
#define _Included_TestJNI1
#ifdef __cplusplus
extern " C " {
#endif
/*
 *      Class:     TestJNI1
 *      Method:    afficherBonjour
 *      Signature: ()V
 */
JNIEXPORT void JNICALL Java_TestJNI1_afficherBonjour(JNIEnv *, jobject);
#ifdef __cplusplus
}
#endif
#endif
```

Le nom de chaque fonction native respecte le format suivant :
Java_nomPleinementQualifieDeLaClasse_NomDeLaMethode

Ce fichier doit être utilisé dans l'implémentation du code de la fonction.

Même si la méthode native est déclarée sans paramètre, il y a toujours deux paramètres passés à la fonction native :

- un pointeur vers une structure JniEnv : cette structure permet d'invoquer certaines fonctionnalités natives de JNI grâce à un tableau de pointeurs de fonctions initialisé par la JVM
- jobject qui est l'objet lui-même : c'est l'équivalent du mot clé this dans le code Java

A partir de Java 9, l'outil javah n'est plus disponible. Il faut utiliser l'option -h du compilateur javac en lui précisant le chemin d'un répertoire dans lequel le compilateur va lui-même le fichier d'en-tête.

Exemple :

```
D:\java\test\jni>dir
23/01/2021  20:39       <DIR>          .
23/01/2021  20:39       <DIR>          ..
23/01/2021  20:39                  230 TestJNI1.java
            2 fichier(s)              230 octets
            2 Rép(s)   2 200 772 608 octets libres
D:\java\test\jni>javac -h . TestJNI1.java
D:\java\test\jni>dir
Répertoire de D:\java\test\jni
23/01/2021  20:40       <DIR> .
23/01/2021  20:40       <DIR> ..
23/01/2021  20:40                  459 TestJNI1.class
23/01/2021  20:40                  408 TestJNI1.h
23/01/2021  20:39                  230 TestJNI1.java
            3 fichier(s) 1 088 octets
            2 Rép(s) 2 198 208 503 octets libres
D:\java\test\jni>
```

29.3. L'écriture du code natif en C

La bibliothèque contenant la ou les fonctions qui seront appelées doit être écrite dans un langage (c ou c++) et compilée.

Pour l'écriture en C, facilitée par la génération du fichier.h, il est nécessaire en plus des includes liées au code des fonctions d'inclure deux fichiers d'en-tête :

- jni.h qui est fourni avec le JDK
- le fichier .h généré par la commande javah

Exemple : TestJNI.c

```
#include <jni.h>
#include <stdio.h>
#include "TestJNI1.h"

JNIEXPORT void JNICALL
Java_TestJNI1_afficherBonjour(JNIEnv *env, jobject obj)
{
  printf(" Bonjour\n ");
  return;
}
```

Il faut compiler ce fichier source sous la forme d'un fichier objet .o

Exemple : avec MinGW sous Windows

```
D:\java\test\jni>gcc -c -I"C:\j2sdk1.4.2_02\include" -I"C:\j2sdk1.4.2_02\include
\win32" -o TestJNI.o TestJNI.c
```

Il faut ensuite définir un fichier .def qui contient la définition des fonctions exportées par la bibliothèque

Exemple : TestJNI.def

```
EXPORTS
Java_TestJNI1_afficherBonjour
```

Il ne reste plus qu'à générer la dll.

Exemple : TestJNI.def

```
D:\java\test\jni>gcc -shared -o mabibjni.dll TestJNI.o TestJNI.def
Warning: resolving _Java_TestJNI1_afficherBonjour by linking to _Java_TestJNI1_a
fficherBonjour@8
Use-enable-stdcall-fixup to disable these warnings
Use-disable-stdcall-fixup to disable these fixups

D:\java\test\jni>dir
Répertoire de D:\java\test\jni
03/12/2003  16:22       <DIR>          .
03/12/2003  16:22       <DIR>          ..
03/12/2003  16:22               12 017 mabibjni.dll
03/12/2003  15:58                  193 TestJNI.c
03/12/2003  16:20                   40 TestJNI.def
03/12/2003  16:04                  543 TestJNI.o
03/12/2003  14:39                  459 TestJNI1.class
03/12/2003  14:39                  399 TestJNI1.h
03/12/2003  14:39                  230 TestJNI1.java
               9 fichier(s)           14 074 octets
               2 Rép(s)   2 198 392 832 octets libres

D:\java\test\jni>
```

Il ne reste plus qu'à exécuter le code Java dans une machine virtuelle.

Exemple :

```
D:\java\test\jni>java TestJNI1
Bonjour
D:\java\test\jni>
```

Il est intéressant de noter que tant que la signature de la méthode native ne change pas, il est inutile de recompiler la classe Java si la fonction dans la bibliothèque est modifiée et recompilée.

29.4. Le passage de paramètres et le renvoi d'une valeur (type primitif)

Une méthode a quasiment toujours besoin de paramètres et souvent besoin de retourner une valeur.

Cette section va définir et utiliser une méthode native qui ajoute deux entiers et renvoie le résultat de l'addition.

Exemple : le code Java

```
class TestJNI1 {
  public native int ajouter(int a, int b);

  static {
    System.loadLibrary("mabibjni");
  }

  public static void main(String[] args) {
    TestJNI1 maclasse = new TestJNI1();
    System.out.println("2 + 3 = " + maclasse.ajouter(2,3));
  }
}
```

La déclaration de la méthode n'a rien de particulier hormis le modificateur native.

La signature de la fonction dans le fichier .h tient compte des paramètres.

Exemple :

```
JNIEXPORT jint JNICALL Java_TestJNI1_ajouter
(JNIEnv *, jobject, jint, jint);
```

Les deux paramètres sont ajoutés dans la signature de la fonction avec un type particulier jint, défini avec un typedef dans le fichier jni.h. Il y a d'ailleurs des définitions pour toutes les primitives.

Primitive Java	Type natif
boolean	jboolean
byte	jbyte
char	jchar
double	jdouble
int	jint
float	jfloat
long	jlong

short	jshort
void	void

Il suffit ensuite d'écrire l'implémentation du code natif.

Exemple :

```
#include <jni.h>
#include <stdio.h>
#include "TestJNI2.h"

JNIEXPORT jint JNICALL Java_TestJNI2_ajouter
(JNIEnv *env, jobject obj, jint a, jint b)
{
  return a + b;
}
```

Il faut ensuite compiler le code :

Exemple :

```
D:\java\test\jni>gcc -c -I"C:\j2sdk1.4.2_02\include" -I"C:\j2sdk1.4.2_02\include
\win32" -o TestJNI2.o TestJNI2.c
```

Il faut définir le fichier .def : l'exemple ci-dessous construit une bibliothèque contenant les fonctions natives des deux classes Java précédemment définies.

Exemple :

```
EXPORTS
Java_TestJNI1_afficherBonjour
Java_TestJNI2_ajouter
```

Il suffit de générer la bibliothèque.

Exemple :

```
D:\java\test\jni>gcc -shared -o mabibjni.dll TestJNI.c TestJNI2.c TestJNI.def
Warning: resolving _Java_TestJNI1_afficherBonjour by linking to _Java_TestJNI1_a
fficherBonjour@8
Use-enable-stdcall-fixup to disable these warnings
Use-disable-stdcall-fixup to disable these fixups
Warning: resolving _Java_TestJNI2_ajouter by linking to _Java_TestJNI2_ajouter@1
6
```

Il ne reste plus qu'à exécuter le code Java

Exemple :

```
D:\java\test\jni>java TestJNI2
2 + 3 = 5
```

29.5. Le passage de paramètres et le renvoi d'une valeur (type objet)

Les objets sont passés par référence en utilisant une variable de type jobject. Plusieurs autres types sont prédéfinis par JNI pour des objets fréquemment utilisés :

Objet C	Objet Java
jobject	java.lang.object
jstring	java.lang.String
jclass	java.lang.Class
jthrowable	java.lang.Throwable
jarray	type de base pour les tableaux
jintArray	int[]
jlongArray	long[]
jfloatArray	float[]
jdoubleArray	double[]
jobjectArray	Object[]
jbooleanArray	boolean[]
jbyteArray	byte[]
jcharArray	char[]
jshortArray	short[]

Exemple : concaténation de deux chaînes de caractères

```
class TestJNI3 {
  public native String concat(String a, String b);

  static {
    System.loadLibrary("mabibjni");
  }

  public static void main(String[] args) {
    TestJNI3 maclasse = new TestJNI3();
    System.out.println("abc + cde = " + maclasse.concat("abc","cde"));
  }
}
```

La déclaration de la fonction native dans le fichier TestJNI3.h est la suivante :

Exemple :

```
/*
 * Class:     TestJNI3
 * Method:    concat
 * Signature: (Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_TestJNI3_concat
  (JNIEnv *, jobject, jstring, jstring);
```

Pour utiliser les paramètres de type jstring dans le code natif, il faut les transformer en utilisant des fonctions proposées par l'interface JNIEnv car le type String de Java n'est pas directement compatible avec les chaînes de caractères C (char *). Il existe des fonctions pour transformer des chaînes codées en UTF-8 ou en Unicode.

Les méthodes pour traiter les chaînes au format UTF-8 sont :

- la méthode GetStringUTFChars() permet de convertir une chaîne de caractères Java en une chaîne de caractères de type C.
- la méthode NewStringUTF() permet de demander la création d'une nouvelle chaîne de caractères.
- la méthode GetStringUTFLength() permet de connaître la taille de la chaîne de caractères.

- la méthode ReleaseStringUTFChars() permet de demander la libération des ressources allouées pour la chaîne de caractères dès que celle-ci n'est plus utilisée. Son utilisation permet d'éviter des fuites mémoire.

Les méthodes équivalentes pour les chaines de caractères au format Unicode sont : GetStringChars(), NewString(), GetStringUTFLength() et ReleaseStringChars()

Exemple : TestJNI3.c

```
#include <jni.h>
#include <stdio.h>
#include "TestJNI3.h"
JNIEXPORT jstring JNICALL Java_TestJNI3_concat
  (JNIEnv *env, jobject obj, jstring chaine1, jstring chaine2){
    char resultat[256];
    const char *str1 = (*env)->GetStringUTFChars(env, chaine1, 0);
    const char *str2 = (*env)->GetStringUTFChars(env, chaine2, 0);
    sprintf(resultat,"%s%s", str1, str2);
    (*env)->ReleaseStringUTFChars(env, chaine1, str1);
    (*env)->ReleaseStringUTFChars(env, chaine2, str2);
    return (*env)->NewStringUTF(env, resultat);
}
```

Attention : ce code est très simpliste car il ne vérifie pas un éventuel débordement du tableau nommé resultat.

Après la compilation des différents éléments, l'exécution affiche le résultat escompté.

Exemple :

```
D:\java\test\jni>java TestJNI3
abc + cde = abccde
```

30. JNDI (Java Naming and Directory Interface)

Chapitre 30

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

JNDI est l'acronyme de Java Naming and Directory Interface. Cette API fournit une interface unique pour utiliser différents services de nommages ou d'annuaires et définit une API standard pour permettre l'accès à ces services.

Il existe plusieurs types de services de nommage parmi lesquels :

- DNS (Domain Name System) : service de nommage utilisé sur internet pour permettre la correspondance entre un nom de domaine et une adresse IP
- LDAP(Lightweight Directory Access Protocol) : annuaire
- NIS (Network Information System) : service de nommage réseau développé par Sun Microsystems
- COS Naming (Common Object Services) : service de nommage utilisé par Corba pour stocker et obtenir des références sur des objets Corba
- etc, ...

Un service de nommage permet d'associer un nom unique à un objet et de faciliter ainsi l'obtention de cet objet.

Un annuaire est un service de nommage qui possède en plus une représentation hiérarchique des objets qu'il contient et un mécanisme de recherche.

JNDI propose donc une abstraction pour permettre l'accès à ces différents services de manière standard. Ceci est possible grâce à l'implémentation de pilotes qui mettent en oeuvre la partie SPI (Service Provider Interface) de l'API JNDI. Cette implémentation se charge d'assurer le dialogue entre l'API et le service utilisé.

JNDI possède un rôle particulier dans les architectures applicatives développées en Java car elle est utilisée dans les spécifications de plusieurs API majeures : JDBC, EJB, JMS, ...

De plus, la centralisation de données dans une source unique pour une ou plusieurs applications facilite l'administration de ces données et leur accès.

Oracle propose un tutorial sur JNDI à l'url : https://docs.oracle.com/javase/jndi/tutorial/ .

Pour utiliser JNDI, il faut un service de nommage correctement installé et configuré et un pilote dédié à ce service.

Ce chapitre contient plusieurs sections :

- ♦ La présentation de JNDI
- ♦ La mise en oeuvre de l'API JNDI
- ♦ L'utilisation d'un service de nommage
- ♦ L'utilisation avec un DNS
- ♦ L'utilisation du File System Context Provider
- ♦ LDAP
- ♦ L'utilisation avec un annuaire LDAP
- ♦ JNDI et J2EE/Java EE

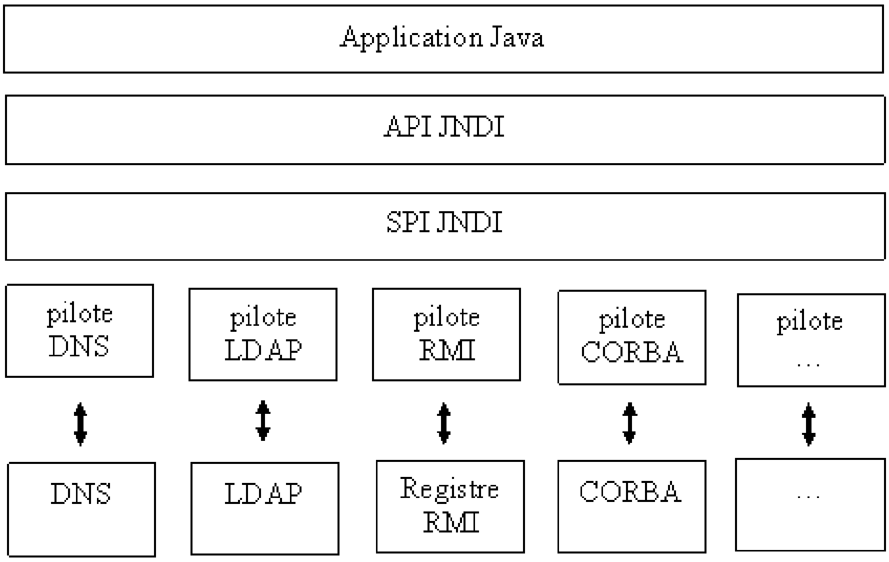
30.1. La présentation de JNDI

JNDI est composée de deux parties

- Une API utilisée pour le développement des applications
- Une SPI utilisée par les fournisseurs d'une implémentation d'un pilote

Un pilote est un ensemble de classes qui implémentent les interfaces de JNDI pour permettre les interactions avec un service particulier. Ce mode de fonctionnement est identique à celui proposé par l'API JDBC.

Il est donc nécessaire de disposer d'un pilote pour assurer le dialogue entre l'application via l'API et le service de nommage ou l'annuaire. La partie API est incluse dans le JDK et Sun propose une implémentation des pilotes pour LDAP, DNS et Corba. Pour d'autres services ou implémentations, il faut utiliser des implémentations des pilotes fournis par des fournisseurs tiers.



Pour définir une connexion, JNDI à besoin d'au moins deux éléments :

- La fabrique du contexte racine : c'est cet objet qui assure le dialogue avec le service en utilisant le protocole adéquat
- L'url du service à utiliser

JNDI n'est pas utilisable uniquement pour des applications J2EE. Une application standalone peut par exemple réaliser une authentification à partir d'un annuaire grâce au protocole LDAP.

Ainsi JNDI est inclus dans J2SE depuis la version 1.3. Pour les versions antérieures (J2SE 1.1 et 1.2), il est nécessaire de télécharger JNDI en tant qu'extension standard et de l'installer.

Pilote	J2SE 1.3	J2SE 1.4
LDAP	Oui	Oui
Corba COS	Oui	Oui
Registre RMI	Oui	Oui
DNS	Non	Oui

Il est aussi possible d'utiliser d'autres pilotes fournis séparément par Sun ou par d'autres fournisseurs.

30.1.1. Les services de nommage

Il existe de nombreux services de nommage : les plus connus sont sûrement les systèmes de fichiers (File system), les DNS, les annuaires LDAP, ...

Un service de nommage permet d'associer un nom à un objet ou à une référence sur un objet. L'objet associé dépend du service : un fichier dans un système de fichiers, une adresse I.P. dans un DNS, ...

Le nom associé à un objet respecte une convention de nommage particulière à chaque type de service.

- Avec un système de fichiers de type Unix, le nom est composé d'éléments séparés par des caractères "/"
- Avec un système de fichiers de type Windows, le nom est composé d'éléments séparés par des caractères "\"
- Avec un service de type DNS, le nom est composé d'éléments séparés par des caractères "." (exemple : www.test.fr).
- Avec un service de type LDAP, le nom désigné par le terme Distinguished Name est composé d'éléments séparés par des caractères ",". Un élément est de la forme clé=valeur.

Pour permettre une abstraction des différents formats de noms utilisés par les différents services, JNDI utilise la classe Name.

30.1.2. Les annuaires

Un annuaire est un outil qui permet de stocker et de consulter des informations selon un protocole particulier. Un annuaire est plus particulièrement dédié à la recherche et la lecture d'informations : il est optimisé pour ce type d'activité mais il doit aussi être capable d'ajouter et de modifier des informations.

Les annuaires sont des extensions des services de nommage en ajoutant en plus la possibilité d'associer d'éventuels attributs à chaque objet.

Caractéristiques	Annuaire	Bases de données
Accès aux données	Lecture privilégiée	Lecture et modification
Représentation des données	Hiérarchique	Ensembliste

Les annuaires les plus connus dans le monde réel sont les pages jaunes et les pages blanches du principal opérateur téléphonique. Même si le but de ces deux annuaires est identique (obtenir un numéro de téléphone), la structure des données est différentes :

- Pages blanches : regroupement par département, ville, nom/prenom
- Pages jaunes : regroupement par activités, ville, nom

Les systèmes de fichiers sont aussi des annuaires : ils associent un nom à un fichier mais stockent aussi des attributs liés à ces fichiers (droits d'accès, dates de création et de modification, ...)

30.1.3. Le contexte

Un service de nommage permet d'associer un nom à un objet. Cette association est nommée binding. Un ensemble d'associations nom/objet est nommé un contexte.

Ce contexte est utilisé lors de l'accès à un élément contenu dans le service.

Il existe deux types de contexte :

- Contexte racine
- Sous contexte

Un sous-contexte est un contexte relatif à un contexte racine.

Par exemple, c:\ est un contexte racine dans un système de fichiers de type Windows. Le répertoire windows (C:\windows) est un sous-contexte du contexte racine qui est dans ce cas nommé sous-répertoire.

Dans DNS, com est un contexte racine et test est un sous contexte (test.com)

30.2. La mise en oeuvre de l'API JNDI

L'API JNDI est contenue dans cinq packages :

Packages	Rôle
javax.naming	Classes et interfaces pour utiliser un service de nommage
javax.naming.directory	Etend les fonctionnalités du package javax.naming pour l'utilisation des services de type annuaire
javax.naming.event	Classes et interfaces pour la gestion des événements lors d'un accès à un service
javax.naming.ldap	Etend les fonctionnalités du package javax.naming.directory pour l'utilisation de la version 3 de LDAP
javax.naming.spi	Classes et interfaces dédiées aux Service Provider pour le développement de pilotes

30.2.1. L'interface Name

Cette interface encapsule un nom en permettant de faire abstraction des conventions de nommage utilisées par le service.

Deux classes implémentent cette interface :

- CompositeName : chaque élément qui compose le CompositeName est séparé par un caractère /
- CompoundName : chaque élément issu de la hiérarchie compose le nom selon certaines règles dépendantes de l'implémentation

30.2.2. L'interface Context et la classe ItitialContext

L'interface javax.Naming.Context représente un ensemble de correspondances nom/objet d'un service de nommage. Elle propose des méthodes pour interroger et mettre à jour ces correspondances.

Méthode	Rôle
void bind(String, Object)	Ajouter une nouvelle correspondance entre le nom et l'objet passé en paramètre
void rebind(String, Object)	Redéfinir l'association nom - objet en écrasant la précédente correspondance si elle existe
Object lookup(String)	Renvoyer un objet à partir de son nom
void unbind(String)	Supprimer la correspondance désignée par le nom fourni en paramètre
void rename(String, String)	Modifier le nom d'une correspondance
NamingEnumeration listBindings(String)	Obtenir une énumération des noms et de leurs objets associés pour le contexte passé en paramètre
NamingEnumeration list(String)	Obtenir une énumération des noms et des classes des objets associés pour le contexte passé en paramètre

Toutes ces méthodes possèdent une version surchargée qui attend le nom de la correspondance sous la forme d'un objet de type Name.

La classe javax.Naming.InitialContext qui implémente l'interface Context encapsule le contexte racine : c'est le noeud qui sert de point d'entrée lors de la connexion avec le service.

Toutes les opérations réalisées avec JNDI sont relatives à ce contexte racine.

Pour obtenir une instance de la classe InitialContext et ainsi réaliser la connexion au service, plusieurs paramètres sont nécessaires :

- java.naming.factory.initial permet de préciser le nom de la fabrique proposée par le fournisseur. Cette fabrique est en charge de l'instanciation d'un objet de type InitialContext
- java.naming.provider.url : URL du context racine

Plusieurs fabriques sont fournies en standard dans J2SE 1.4 :

Service	Fabrique
CORBA	com.sun.jndi.cosnaming.CNCtxFactory
DNS	com.sun.jndi.dns.DnsContextFactory
LDAP	com.sun.jndi.ldap.LdapCtxFactory
RMI	com.sun.jndi.rmi.registry.RegistryContextFactory

Ces deux paramètres sont obligatoires mais d'autres peuvent être nécessaires notamment ceux concernant la sécurité pour l'accès au service.

L'interface Context définit des constantes pour le nom de ces paramètres. Il y a plusieurs moyens pour les définir :

- les définir sous la forme de variables d'environnement passées à la JVM en utilisant l'option -D
- les définir sous la forme d'une collection de type Hashtable passée en paramètre au constructeur de la classe InitialContext
- les définir dans un fichier nommé jndi.properties accessible dans le classpath

Exemple :

```
      Hashtable hashtableEnvironment = new Hashtable();
      hashtableEnvironment.put(Context.INITIAL_CONTEXT_FACTORY,
         "com.sun.jndi.fscontext.RefFSContextFactory");
      hashtableEnvironment.put(Context.PROVIDER_URL, "file:c:/");
      Context context = new InitialContext(hashtableEnvironment);
```

Il est possible de réaliser des opérations particulières à partir du Context. Attention toutefois, toutes ces opérations ne sont pas utilisables avec tous les pilotes. Par exemple, l'accès à un service de type DNS n'est possible qu'en consultation.

30.3. L'utilisation d'un service de nommage

Pour pouvoir utiliser un service de nommage, il faut tout d'abord obtenir un contexte racine qui va encapsuler la connexion au service.

A partir de ce contexte, il est possible de réaliser plusieurs opérations :

- bind : associer un objet avec un nom
- rebind : modifier une association
- unbind : supprimer une association
- lookup : obtenir un objet à partir de son nom
- list : obtenir une liste des associations

Toutes les opérations possèdent deux versions surchargées attendant respectivement :

- Un objet de type Name : cet objet encapsule une séquence ordonnée de un ou plusieurs éléments (l'intérêt de cette classe est de permettre la manipulation individuelle de chaque élément).
- Une chaîne de caractères qui contient une séquence d'éléments

30.3.1. L'obtention d'un objet

Pour obtenir un objet du service de nommage, utiliser la méthode lookup() du contexte.

Exemple :

```
import javax.naming.*;
...
  public String getValeur() throws NamingException {
    Context context = new InitialContext();
    return (String) context.lookup("/config/monApplication");
  }
```

Ceci peut permettre de facilement stocker des options de configuration d'une application, plutôt que de les stocker dans un fichier de configuration. C'est encore plus intéressant si le service qui stocke ces données est accessible par le réseau car cela permet de centraliser ces options de configuration.

Il peut permettre aussi de stocker des données "sensibles" comme des noms d'utilisateurs et des mots de passe pour accéder à une ressource et ainsi empêcher leur accès en clair dans un fichier de configuration.

30.3.2. Le stockage d'un objet

Généralement les objets à stocker doivent être d'un type particulier, dépendant du pilote utilisé : il est fréquent que de tels objets doivent implémenter une interface (java.io.Serializable, java.rmi.Remote, etc ...)

La méthode bind() permet d'associer un objet à un nom.

Exemple :

```
import javax.naming.*;
...
  public void createName() throws NamingException {
    Context context = new InitialContext();
    context.bind("/config/monApplication", "valeur");
  }
```

30.4. L'utilisation avec un DNS

A partir de J2SE 1.4, Sun propose en standard une implémentation permettant d'accéder à un DNS par JNDI.

Exemple :

```
import javax.naming.*;
import javax.naming.directory.*;
import java.util.*;

public class TestDNS2 {

     public static void main(String[] args) {
        try {
           Hashtable env = new Hashtable();
           env.put("java.naming.factory.initial",
                 "com.sun.jndi.dns.DnsContextFactory");
```

```
          env.put("java.naming.provider.url", "dns://80.10.246.2/");

          DirContext ctx = new InitialDirContext(env);
          Attributes attrs = ctx.getAttributes("java.sun.com",
                new String[] { "A" });

          for (NamingEnumeration ae = attrs.getAll(); ae.hasMoreElements();) {
             Attribute attr = (Attribute) ae.next();
             String attrId = attr.getID();
             for (Enumeration vals = attr.getAll();
                            vals.hasMoreElements();
                            System.out.println(attrId + ": " + vals.nextElement())
                         );
          }
          ctx.close();
       } catch (Exception e) {
          System.err.println("Probleme lors de l'interrogation du DNS: " + e);
          e.printStackTrace();
       }
    }
}
```

Pour permettre une exécution correcte de ce programme, il est nécessaire de mettre l'adresse IP du serveur DNS utilisé.

Lors de l'exécution, il faut fournir en paramètre le nom d'un domaine et d'un serveur.

30.5. L'utilisation du File System Context Provider

C'est une implémentation de référence proposée par Sun qui permet un accès à un système de fichiers par JNDI.

Cela peut paraître étonnant mais un système de fichiers peut être vu comme un service de nommage qui associe un nom (par exemple c:\temp\test.txt) à un fichier ou un répertoire

Cette implémentation n'est pas fournie en standard avec le JDK mais elle peut être téléchargée (fscontext-x.x.x.jar)

La version utilisée dans cette section est la 1_2 beta3. Il suffit de décompresser le fichier fscontext-1_2-beta3.zip dans un répertoire du système et d'ajouter les fichiers fscontext.jar et providerutil.jar du sous-répertoire lib décompressé dans le classpath de l'application.

Exemple : obtenir la liste de tous les fichiers et répertoires à la racine du disque C:

```
import java.util.Hashtable;
import javax.naming.Binding;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingEnumeration;
import javax.naming.NamingException;

public class TestJNDI {

  public static void main(String[] args) {

    try {
      Hashtable hashtableEnvironment = new Hashtable();
      hashtableEnvironment.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.fscontext.RefFSContextFactory");
      hashtableEnvironment.put(Context.PROVIDER_URL, "file:c:/");

      Context context = new InitialContext(hashtableEnvironment);
      NamingEnumeration namingEnumeration = context.listBindings("");

      while (namingEnumeration.hasMore()) {
        Binding binding = (Binding) namingEnumeration.next();
        System.out.println(binding.getName());
      }
      context.close();
    } catch (NamingException namingexception) {
      namingexception.printStackTrace();
    }
  }
}
```

Il est aussi possible de rechercher un fichier dans un répertoire. Dans ce cas, le contexte initial précisé est le répertoire dans lequel le fichier doit être recherché. La méthode lookup() recherche uniquement dans ce répertoire

Exemple :

```
import java.io.File;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class TestJNDI2 {

  public static void main(String argv[]) {
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY,
          "com.sun.jndi.fscontext.RefFSContextFactory");
    env.put(Context.PROVIDER_URL, "file:c:/");

    try {
      Context ctx = new InitialContext(env);
      File fichier = (File) ctx.lookup("boot.ini");
      System.out.println("objet trouve = " + fichier);
    } catch (NamingException e) {
      e.printStackTrace();
    }
  }
}
```

Attention, le cast effectué sur l'objet retourné par la méthode lookup() doit être pertinent en fonction du contexte.

30.6. LDAP

LDAP, acronyme de Lightweight Directory Access Protocol, est un protocole de communication vers un annuaire en utilisant TCP/IP. Il est une simplification du protocole X 500 (d'où le L de Lightweight).

Le but principal est de retrouver des données insérées dans l'annuaire. Ce protocole est donc optimisé pour la lecture et la recherche d'informations.

LDAP est un protocole largement supporté par l'industrie informatique : il existe de nombreuses implémentations libres et commerciales : Microsoft Active Directory, OpenLDAP, Netscape Directory Server, Sun NIS, Novell NDS, ..

Ce protocole ne précise pas comment ces données sont stockées sur le serveur. Ainsi un serveur de type LDAP peut stocker n'importe quel type de données : ce sont souvent des ressources (personnes, matériels réseaux, ...).

La version actuelle de LDAP est la v3 définie par les RFC 2252 et RFR 2256 de l'IETF.

Dans un annuaire LDAP, les noeuds sont organisés sous une forme arborescente hiérarchique nommée le DIT (Direct Information Tree). Chaque noeud de cette arborescence représente une entrée dans l'annuaire. Chaque entrée contient un objet qui possède un ou plusieurs attributs dont les valeurs permettent d'obtenir des informations sur l'objet. Un objet appartient à une classe au sens LDAP.

La première entrée dans l'arborescence est nommée racine et est unique.

Chaque objet possède un Relative Distinguish Name (RDN) qui correspond à une paire clé/valeur d'un attribut obligatoire. Un objet est identifié de façon unique grâce à sa référence unique dans le DIT : son Distinguish Name (DN) qui est composé de l'ensemble des RDN de chaque objet père dans l'arborescence lue de droite à gauche et son RDN (ceci correspond donc au DN de l'entrée père et de son RDN). Cette référence représente donc le chemin d'accès depuis la racine de l'arborescence. Le DN se lit de droite à gauche puisque la racine est à droite.

La convention de nommage utilisée pour le DN, utilise la virgule comme séparateur et se lit de droite à gauche.

Exemple :

```
uid=jm,ou=utilisateur,o=test.com
```

Le premier élément du DN, nommé Relative Distinguished Name (RDN), est composé d'une paire clé/valeur. Comme valeur de clé, LDAP utilise généralement un mnémonique :

Mnnémonique	Libellé	Description
dn	Distinguished name	Nom unique dans l'arborescence
uid	Userid	Identifiant unique pour l'utilisateur
cn	Common name	Nom et prénom d'un utilisateur
givenname	First name	Prénom d'un utilisateur
sn	Surname	Nom de l'utilisateur
l	Location	Ville de l'utilisateur
o	Organization	Généralement la racine de l'annuaire (exemple : le nom de l'entreprise)
ou	Organizational unit	Généralement une branche de l'arbre (exemple : une division, un département ou un service)
st	State	Etat du pays de l'utilisateur
c	Country	pays de l'utilisateur
Mail	Email	Email de l'utilisateur

Un élément qui compose une entrée dans l'annuaire est nommé objet. Chaque objet peut contenir des attributs obligatoires ou facultatifs. Un attribut correspond à une propriété d'un objet, par exemple un email ou un numéro de téléphone pour une personne. Un attribut se présente sous la forme d'une paire clé/valeur(s).

Les classes caractérisent les objets en définissant les attributs optionnels et obligatoires qui les composent. Il existe des attributs standard communément utilisés mais il est aussi possible d'en définir d'autres.

L'ensemble des règles qui définissent l'arborescence et les attributs utilisables est stocké dans un schéma. : ce dernier permet donc de définir les classes et les objets pouvant être stockés dans l'annuaire. Un annuaire peut supporter plusieurs schémas.

Une fonctionnalité intéressante est la possibilité de pouvoir stocker des objets Java directement dans l'annuaire et de pouvoir les retrouver en utilisant le protocole LDAP. Ces objets peuvent avoir des fonctionnalités diverses telles qu'une connexion à une source de données, un objet contenant des options de paramétrage de l'application, etc ...

Un serveur LDAP propose les fonctionnalités de base suivantes :

- Connexion/déconnexion au serveur
- Gestion de la sécurité lors d'accès aux objets
- Ajout, modification, suppression d'objets
- Gestion d'attributs sur les objets
- Recherche d'objets

30.6.1. L'outil OpenLDAP

Il faut télécharger OpenLDAP sur le site https://www.openldap.org/. et l'installer. La version utilisée dans cette section, est la 2.2.29.

Il faut sélectionner la langue d'installation entre anglais et allemand.

Un assistant guide l'utilisateur dans les différentes étapes de l'installation :

- sur la page d'accueil : cliquez sur le bouton « Next »
- sur la page « Licence Agreement » : lisez la licence et si vous l'acceptez cliquez sur le bouton radio « I accept the agreement » et cliquez sur le bouton « Next »
- sur la page « Select Destination Location », sélectionnez le répertoire de destination et cliquez sur le bouton « Next ». Il est préférable de choisir un répertoire sans espace (exemple : C:\OpenLDAP) plutôt que le répertoire C:\Program Files\OpenLDAP proposé par défaut
- sur la page « Select Components » : laissez la sélection par défaut et cliquez sur le bouton « Next »
- sur la page « Select Start Menu Folder », cliquez sur le bouton « Next »
- sur la page « Select Additional tasks », cliquez sur le bouton « Next »
- sur la page « Ready to Install », cliquez sur le bouton « Install »
- Les fichiers sont copiés sur le système d'exploitation
- sur la page « Completing the OpenLDAP Setup Wizard », cliquez sur le bouton « Finish »

OpenLDAP propose en standard plusieurs schémas prédéfinis stockés dans le sous-répertoire schema.

Le fichier slapd.conf contient les principaux paramètres. Il est installé pré-paramétré dans le répertoire d'installation d'OpenLDAP (c:\openldap dans cette section).

Au début du fichier, si l'on utilise OpenLDAP avec JNDI pour stocker des objets Java, il faut ajouter le schéma Java.

Exemple :

```
#
ucdata-path      ./ucdata
include          ./schema/core.schema
include>/b<>b<          ./schema/java.schema>/b<
...
```

Il faut ensuite configurer la base de données, le suffixe qui est la racine du serveur et le compte de l'administrateur du serveur (root).

Exemple :

```
#######################################################################
# BDB database definitions
#######################################################################
database              bdb
suffix                "dc=my-domain,dc=com"
rootdn                "cn=Manager,dc=my-domain,dc=com"
# Cleartext passwords, especially for the rootdn, should
# be avoid.  See slappasswd(8) and slapd.conf(5) for details.
# Use of strong authentication encouraged.
rootpw                secret
# The database directory MUST exist prior to running slapd AND
# should only be accessible by the slapd and slap tools.
# Mode 700 recommended.
directory             ./data
# Indices to maintain
index    objectClass      eq
```

Il faut remplacer la valeur des clés suffix et rootdn par les valeurs appropriées au contexte.

Exemple :

```
suffix                "dc=test-ldap,dc=net"
rootdn              "cn=ldap-admin,dc=test-ldap,dc=net"
```

Pour insérer le mot de passe dans le fichier slapd.conf, il faut le crypter grâce à la commande slappasswd

Exemple :

```
C:\openldap>slappasswd -s ldap-admin
{SSHA}ZUPUkq7mt21rEmrFgFc0cgk9izpwL7oY
```

Il suffit alors de remplacer dans le fichier slapd.conf la ligne

rootpw secret

par la ligne ci-dessous qui contient le mot de passe crypté

rootpw {SSHA}ZUPUkq7mt21rEmrFgFc0cgk9izpwL7oY

Pour lancer le serveur LDAP, il suffit de double cliquer sur le fichier slapd.exe

Il ne faut pas fermer cette fenêtre dans laquelle le serveur s'exécute. Pour éviter d'avoir une fenêtre DOS ouverte, il faut utiliser le serveur en tant que service en exécutant la commande net start OpenLDAP-slapd.

Exemple :

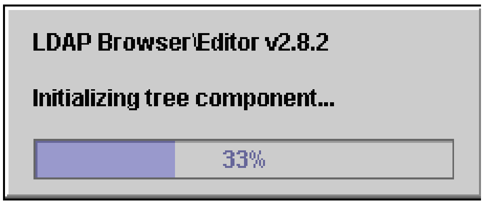
```
C:\OpenLDAP>net start OpenLDAP-slapd
Le service OpenLDAP Directory Service démarre..
Le service OpenLDAP Directory Service a démarré.
```

Par défaut, les serveurs de type LDAP utilise le port 389 : c'est le cas pour OpenLDAP.

30.6.2. LDAPBrowser

Téléchargez le fichier Browser282b2.zip et le décompresser dans un répertoire du système.

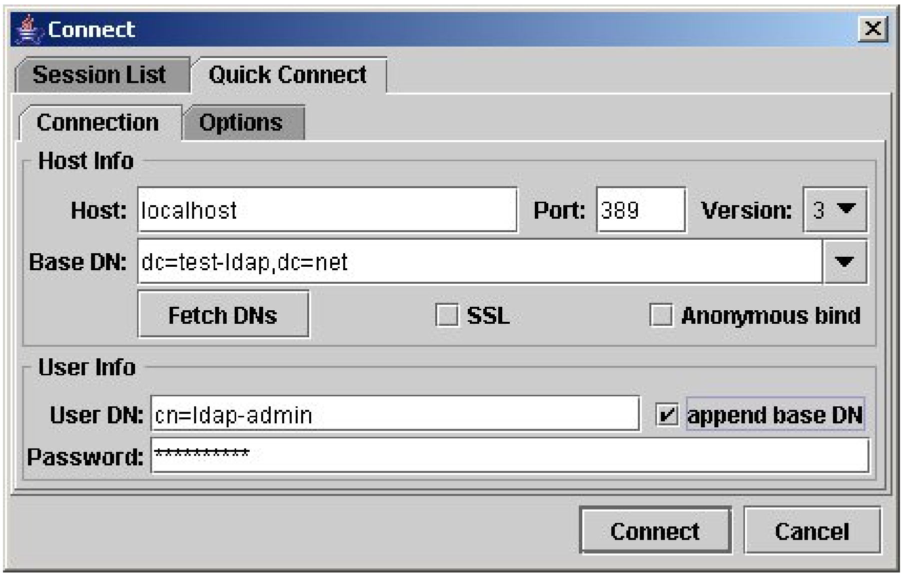
Pour lancer l'application, il suffit de double cliquer sur le fichier lbe.bat.



LDAP Browser\Editor v2.8.2

Initializing tree component...

Dans la boîte de dialogue « Connect », sélectionnez l'onglet « Quick Connect » et saisissez les informations nécessaires à la connexion.



Host: localhost Port: 389 Version: 3

Base DN: dc=test-ldap,dc=net

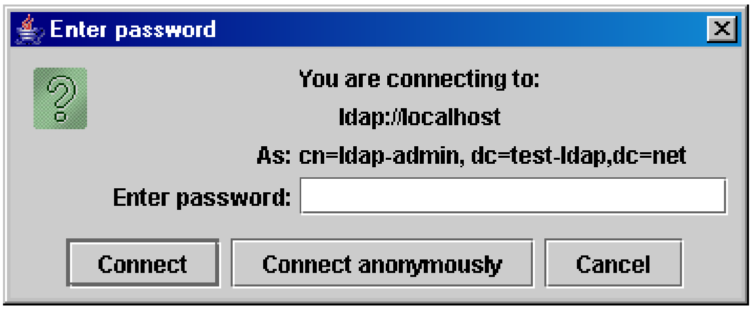
□ SSL □ Anonymous bind

User DN: cn=ldap-admin ☑ append base DN

Password: **********

Cliquez sur le bouton « Connect ».

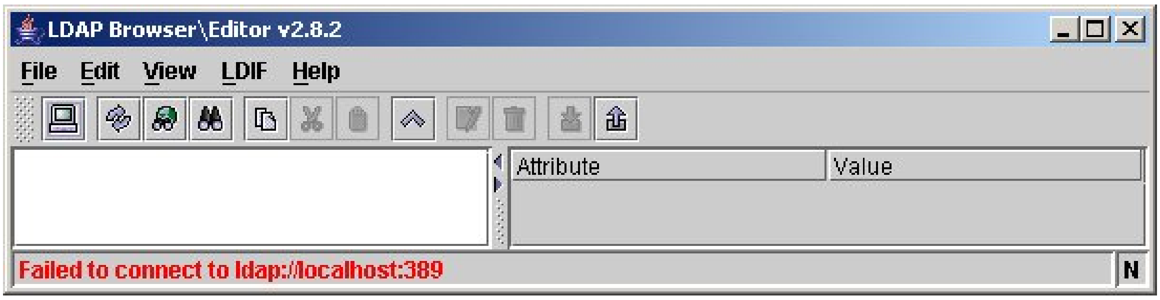
Si le mot de passe n'est pas saisi, une boîte de dialogue permet de le fournir.



You are connecting to:
ldap://localhost
As: cn=ldap-admin, dc=test-ldap,dc=net
Enter password:

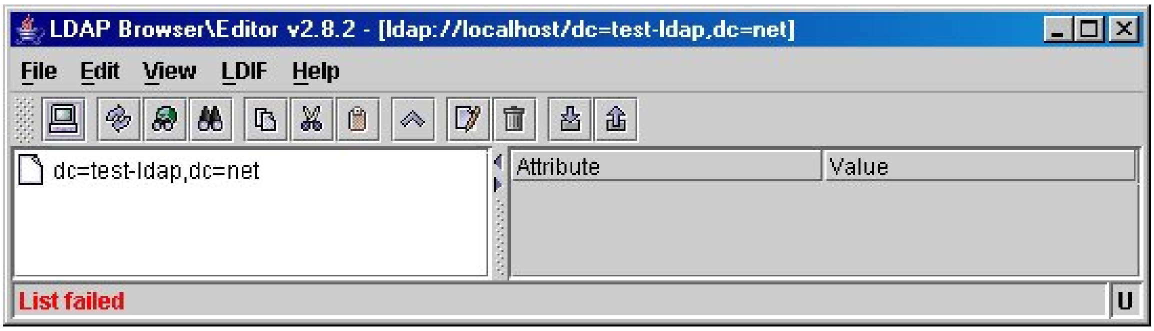
Il suffit alors de saisir le mot de passe défini dans le fichier slapd.conf et de cliquer sur le bouton « Connect ».

Si les informations saisies ne permettent pas de réussir la connexion, alors le message « Failed to connect » est affiché.



Failed to connect to ldap://localhost:389

Si l'annuaire est vide, alors le message « List failed » est affiché


LDAP Browser\Editor v2.8.2 - [ldap://localhost/dc=test-ldap,dc=net]
File
Edit
View
LDIF
Help
dc=test-ldap,dc=net
Attribute
Value
U

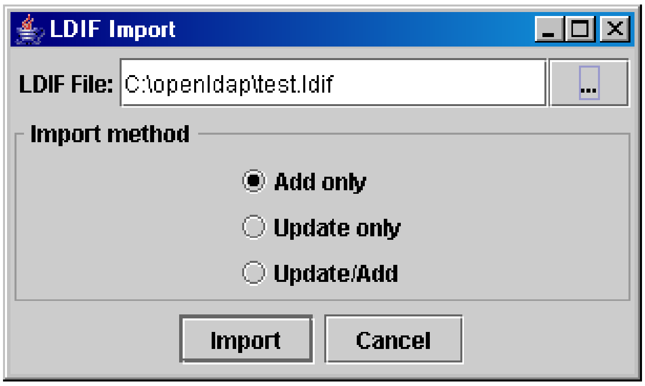
List failed

Pour initialiser l'annuaire, le plus facile est d'écrire un fichier au format LDIF (Lightweight Data Interchange Format). Ce format permet d'importer ou d'exporter des données de l'annuaire. Il permet aussi de modifier des données dans l'annuaire. Il est détaillé dans la section suivante.

Exemple : le fichier test.ldif

```
dn: dc=test-ldap,dc=net
objectClass: dcObject
objectClass: organization
dc: test-ldap
o: Entreprise Test
description: Entreprise de tests

dn: cn=Durand,dc=test-ldap,dc=net
objectClass: organizationalRole
cn: Durand
description: Président directeur général
```

Pour insérer les données du fichier test.ldif, il faut sélectionner la racine et utiliser l'option Import du menu LDIF.

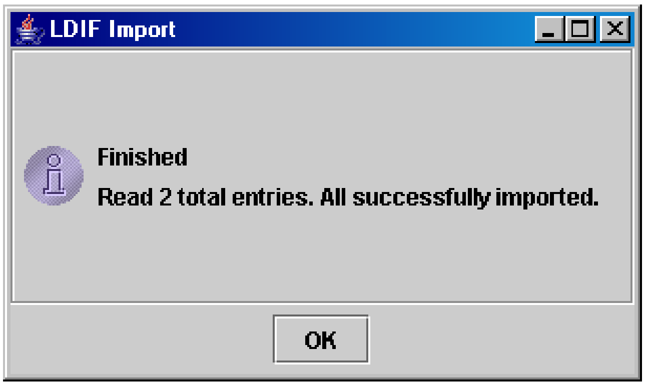


LDIF File: C:\openldap\test.ldif

Import method

● Add only
○ Update only
○ Update/Add

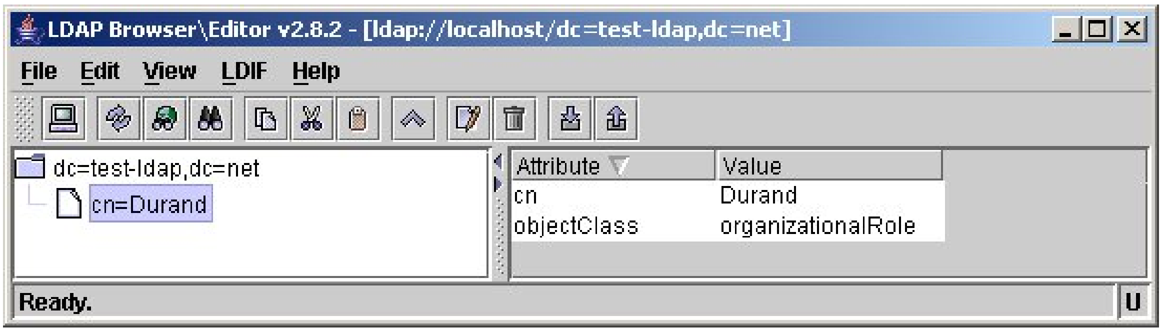
Sélectionner le fichier .ldif et cliquez sur le bouton « Import ».



Finished

Read 2 total entries. All successfully imported.

Les deux entrées sont affichées dans l'arborescence du serveur.



LDAP Browser\Editor v2.8.2 - [ldap://localhost/dc=test-ldap,dc=net]

dc=test-ldap,dc=net
- cn=Durand

Attribute	Value
cn	Durand
objectClass	organizationalRole

Ready.

30.6.3. LDIF

Le format LDIF permet de réaliser des opérations d'import/export de données d'un annuaire.

La structure générale de ce format est la suivante :

Exemple :

```
[<id>]
dn: <distinguished name>
objectclass: <objectclass>
objectclass: <objectclass>
...
<attribut> : <valeur>
<attribut> : <valeur>
...
```

Chaque entrée est séparée dans le fichier par une ligne vide.

<id> est un entier positif facultatif qui représente un identifiant des données au niveau du serveur.

Chaque élément définit dans le fichier est séparé par une ligne vide. Il commence par son DN

Chaque attribut est définit sur sa propose ligne. La définition peut se poursuivre sur la ligne suivante si celle-ci commence par une espace ou une tabulation.

Pour fournir plusieurs valeurs à un attribut, il suffit de répéter la clé de cet attribut à raison d'une ligne pour chaque valeur.

Si la valeur d'un attribut contient des caractères non imprimables (des données binaires comme une image par exemple) alors la clé de l'attribut est suivie de :: et la valeur est encodée en base 64.

Le format LDIF permet également d'effectuer des modifications de données grâce à des opérations : add (ajouter une entrée), delete (supprimer une entrée), modrdn (modifier le rdn)

30.7. L'utilisation avec un annuaire LDAP

L'API JNDI permet un accès à un annuaire LDAP.

30.7.1. L'interface DirContext

L'interface DirContext est une classe fille de l'interface Context. Elle propose des fonctionnalités pour utiliser un service de nommage et propose en plus des fonctionnalités dédiées aux annuaires telles que la gestion des attributs et la recherche d'éléments.

Méthode	Rôle
void bind(String, Object, Attributes)	Associer un objet avec des attributs à un nom
void rebind(String, Object , Attributes)	Redéfinir l'association d'un nom avec un objet et ses attributs
Attributes getAttributes(String)	Obtenir tous les attributs de l'objet associé au nom fourni en paramètre
Attributes getAttributes(String, String [])	Obtenir les valeurs des attributs listés dans le tableau en paramètre pour l'objet dont le nom est fourni
void modifyAttributes(String, int, Attributes)	Modifier les attributs de l'objet en paramètre.

	L'entier permet de préciser le type de mise à jour à effectuer : ADD_ATTRIBUTE, REPLACE_ATTRIBUTE et REMOVE_ATTRIBUTE
void modifyAttributes(String, ModificationItem [])	Mettre à jour des attributs dans l'ordre des éléments du tableau fourni en paramètre
NamingEnumeration search()	Rechercher des entrées dans l'annuaire selon des critères fournis sous la forme d'un filtre. Il existe plusieurs surcharges de cette méthode
DirContext getSchema(String)	Retourner le schéma associé à un nom

Pour pouvoir accéder à un annuaire, les étapes sont similaires à celles d'un accès à un service de nommage. Il faut obtenir une instance de type DirContext en instanciant un objet de type InitialDirContext(). Cet objet a besoin de paramètres généralement fournis sous la forme d'une collection de type Hashtable.

Ces paramètres sont les mêmes que pour un accès à un service de nommage.

30.7.2. La classe InitialDirContext

L'instanciation d'un objet de type InitialDirContext permet de se connecter à l'annuaire et de se positionner à un endroit précis de l'arborescence de l'annuaire nommé contexte initial.

Toutes les opérations réalisées dans l'annuaire le seront relativement à ce contexte initial.

Pour se connecter à un serveur LDAP, il faut obtenir un objet qui implémente l'interface DirContext : c'est généralement un objet de type InitialDirContext qui est obtenu en utilisant une collection de type Hashtable contenant les paramètres de connexion fournis à une fabrique dédiée.

Afin de réaliser la connexion, il est nécessaire de fournir des paramètres pour configurer son environnement. Ces paramètres sont fournis au constructeur de la classe InitialDirContext sous la forme d'un objet de type Hashtable : ces paramètres concernent plusieurs types d'informations :

- Le fournisseur de l'implémentation
- La localisation de l'annuaire
- La sécurité d'accès

Deux paramètres sont obligatoires :

Context.INITIAL_CONTEXT_FACTORY	permet de préciser la classe fournie par le fournisseur
Context.PROVIDER_URL	permet de préciser une url pour localiser l'annuaire. Le format de cette url dépend du fournisseur

Exemple :

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL,"ldap://localhost:389");
DirContext dircontext = new InitialDirContext(env);
```

Si l'accès au serveur est sécurisé, il faut fournir des paramètres supplémentaires pour permettre cette authentification : le type de sécurité utilisé, le DN d'un utilisateur et son mot de passe :

Context.SECURITY_AUTHENTICATION	Permet de préciser le type de sécurité utilisé. Les valeurs possibles sont : simple, SSL, SASL
Context.SECURITY_PRINCIPAL	Permet de préciser le Distinguished Name de l'utilisateur
Context.SECURITY_CREDENTIALS	Le mot de passe de l'utilisateur

LDAP supporte trois modes de sécurité :

- Simple : pas de cryptage du DN de l'utilisateur ni de son mot de passe
- SSL : utilisation du cryptage SSL à travers le réseau si le serveur LDAP le supporte
- SASL : utilisation des algorithmes MD5/Kerberos

Exemple :

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;

public class TestLDAP {

  public static void main(String[] args) {
    Hashtable env = new Hashtable();
    env
        .put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "ldap://localhost:389");
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    env.put(Context.SECURITY_PRINCIPAL, "cn=ldap-admin,dc=test-ldap,dc=net");
    env.put(Context.SECURITY_CREDENTIALS, "inconnu");

    DirContext dirContext;

    try {
      dirContext = new InitialDirContext(env);
      dirContext.close();
    } catch (NamingException e) {
      System.err.println("Erreur lors de l'acces au serveur LDAP" + e);
      e.printStackTrace();
    }
  }
}
```

Comme l'objet InitialDircontext encapsule la connexion vers l'annuaire, il est nécessaire de fermer cette connexion dès que celle-ci n'est plus utilisée en faisant appel à la méthode close().

La plupart des méthodes de la classe InitialDirContext peuvent lever une exception de type NamingException.

Si les informations de connexion au serveur sont erronées alors une exception de type javax.naming.CommunicationException est levée.

Si les informations fournies pour l'authentification sont erronées alors une exception de type javax.naming.AuthenticationException est levée avec le message «[LDAP: error code 49 - Invalid Credentials]»

A partir d'une instance de DirContext, il est possible d'accéder et de réaliser des opérations dans l'annuaire.

30.7.3. Les attributs

Pour manipuler les attributs d'un objet, deux interfaces existent :

- Attributes : qui encapsule les différents attributs d'un objet
- Attribut qui encapsule la valeur d'un attribut

Exemple :

```
      dirContext = new InitialDirContext(env);
      Attributes attributs = dirContext.getAttributes("cn=Dupont,dc=test-ldap,dc=net");
      Attributs attribut = (Attribut) attributs.get("description") ;
      System.out.println("Description : " + attribut.get());
```

Deux classes implémentent respectivement ces deux interfaces : BasicAttributes et BasicAttribut

Il est possible d'instancier une liste d'attributs par exemple pour les associer à un nouvel objet ajouté dans l'annuaire.

Exemple :

```
      Attributes attributes = new BasicAttributes(true);
      Attribute attribut = new BasicAttribute("telephoneNumber");
      attribut.add("99.99.99.99.99");
      attributes.put(attribut);
```

30.7.4. L'utilisation d'objets Java

La possibilité de stocker des objets Java dans un annuaire LDAP offre plusieurs intérêts :

- Stocker des objets accessibles par plusieurs applications
- Stocker des objets entre plusieurs exécutions d'une même application
- Stocker des objets pour échanger des données entre plusieurs applications

A partir d'un objet de type contexte, il suffit de faire appel à la méthode bind() qui attend en paramètre un nom d'objet et un objet. Cette méthode va ajouter une entrée dans l'annuaire qui va associer le nom de l'objet à l'objet fourni en paramètre.

La méthode lookup() d'un objet de type Context permet d'obtenir un objet Java stocké dans l'annuaire à partir de son nom.

Ces deux méthodes peuvent lever une exception de type NamingException lors de leur exécution.

30.7.5. Le stockage d'objets Java

La plupart des annuaires permettent le stockage d'objets Java, sous réserve que l'annuaire le propose et que le schéma adéquat soit utilisé dans la configuration du serveur, ce qui n'est généralement pas le cas par défaut.

Le stockage se fait en utilisant la méthode bind() du contexte

Exemple :

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;

public class TestLDAP2 {

  public static void main(String[] args) {
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "ldap://localhost:389");
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    env.put(Context.SECURITY_PRINCIPAL, "cn=ldap-admin,dc=test-ldap,dc=net");
    env.put(Context.SECURITY_CREDENTIALS, "ldap-admin");
    DirContext dirContext;

    try {

      dirContext = new InitialDirContext(env);
      MonObjet objet = new MonObjet("valeur1","valeur2");
      dirContext.bind("cn=monobject,dc=test-ldap,dc=net", objet);
      dirContext.close();

    } catch (NamingException e) {
      System.err.println("Erreur lors de l'acces au serveur LDAP" + e);
      e.printStackTrace();
    }
    System.out.println("fin des traitements");
  }
}
```

Les objets Java peuvent être stockés de différentes manières selon le serveur :

- Stockage des objets eux-mêmes sous la forme sérialisée
- Stockage d'une référence mémoire vers l'objet Java : cette référence est encapsulée dans un objet de type java.naming.Reference
- Stockage des champs de l'objet sous la forme d'attributs : l'objet ainsi stocké doit obligatoirement implémenter l'interface DirContext.

L'implémentation de toutes ces méthodes est laissée libre mais le serveur doit au moins en proposer une.

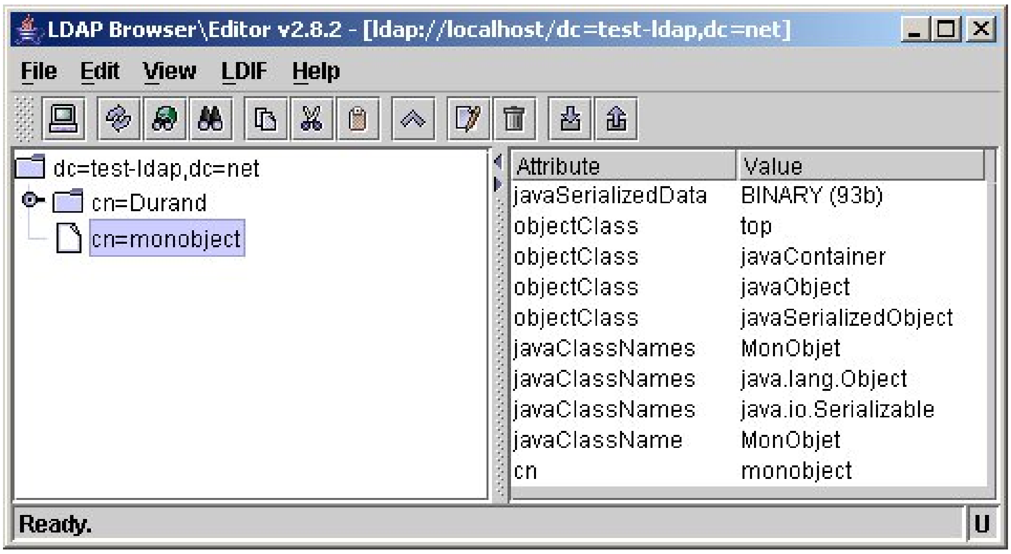
Pour le stockage sous la forme sérialisée, il est nécessaire que l'objet stocké implémente l'interface java.io.Serilizable. C'est la solution la plus facile à mettre en oeuvre

Exemple :

```
import java.io.Serializable;

public class MonObjet implements Serializable {

  private static final long serialVersionUID = 3309572647822157460L;
  private String champ1;
  private String champ2;

  public MonObjet() {
    super();
  }

  public MonObjet(String champ1, String champ2) {
    super();
    this.champ1 = champ1;
    this.champ2 = champ2;
  }

  public String getChamp1() {
    return champ1;
  }

  public void setChamp1(String champ1) {
    this.champ1 = champ1;
  }

  public String getChamp2() {
    return champ2;
  }

  public void setChamp2(String champ2) {
    this.champ2 = champ2;
  }
}
```

Une exception de type java.lang.IllegalArgumentException est levée si l'objet ne respecte pas les règles permettant son ajout dans l'annuaire. Avec OpenLDAP, cette exception est levée avec le message « can only bind Referenceable, Serializable, DirContext ».

Si tout se passe bien, l'objet est ajouté dans l'annuaire sous sa forme sérialisée.


LDAP Browser\Editor v2.8.2 - [ldap://localhost/dc=test-ldap,dc=net]
File
Edit
View
LDIF
Help
dc=test-ldap,dc=net
cn=Durand
cn=monobject
Ready.
U

Attribute	Value
javaSerializedData	BINARY (93b)
objectClass	top
objectClass	javaContainer
objectClass	javaObject
objectClass	javaSerializedObject
javaClassNames	MonObjet
javaClassNames	java.lang.Object
javaClassNames	java.io.Serializable
javaClassName	MonObjet
cn	monobject

30.7.6. L'obtention d'un objet Java

Pour obtenir un objet stocké, il faut utiliser la méthode lookup()

Exemple :

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;

public class TestLDAP3 {

  public static void main(String[] args) {
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "ldap://localhost:389");
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    env.put(Context.SECURITY_PRINCIPAL, "cn=ldap-admin,dc=test-ldap,dc=net");
    env.put(Context.SECURITY_CREDENTIALS, "ldap-admin");
    DirContext dirContext;

    try {

      dirContext = new InitialDirContext(env);
      MonObjet objet = (MonObjet) dirContext.lookup("cn=monobject,dc=test-ldap,dc=net");

      System.out.println("champ1="+objet.getChamp1());
      System.out.println("champ2="+objet.getChamp2());

      dirContext.close();
    } catch (NamingException e) {
      System.err.println("Erreur lors de l'acces au serveur LDAP" + e);
      e.printStackTrace();
    }
    System.out.println("fin des traitements");
  }
}
```

Résultat :

```
champ1=valeur1
champ2=valeur2
fin des traitements
```

Si le DN fourni en paramètre de la méthode lookup ne correspond pas à celui d'un objet stocké dans l'annuaire, une exception de type javax.naming.NameNotFoundException avec le message « [LDAP: error code 32 - No Such Object] » est levée.

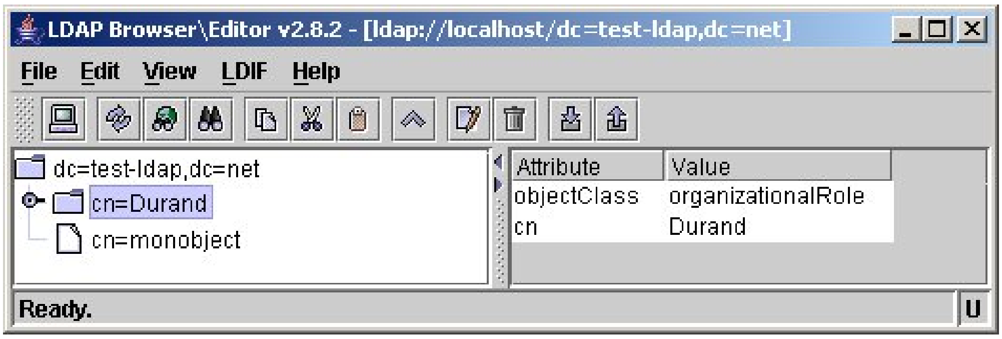
30.7.7. La modification d'un objet

La méthode modifyAttributes() de la classe DirContext permet de modifier les attributs d'un objet stocké dans l'annuaire. La méthode modifyAttributes() possède plusieurs surcharges.

Différentes opérations sont réalisables avec cette méthode en utilisant des constantes prédéfinies pour chaque type :

- ADD_ATTRIBUTE : ajout d'un attribut
- REMOVE_ATTRIBUTE : suppression d'un attribut
- REPLACE_ATTRIBUTE : modification d'un attribut

Ces modifications sont soumises aux restrictions mises en place sur le serveur au niveau du schéma.

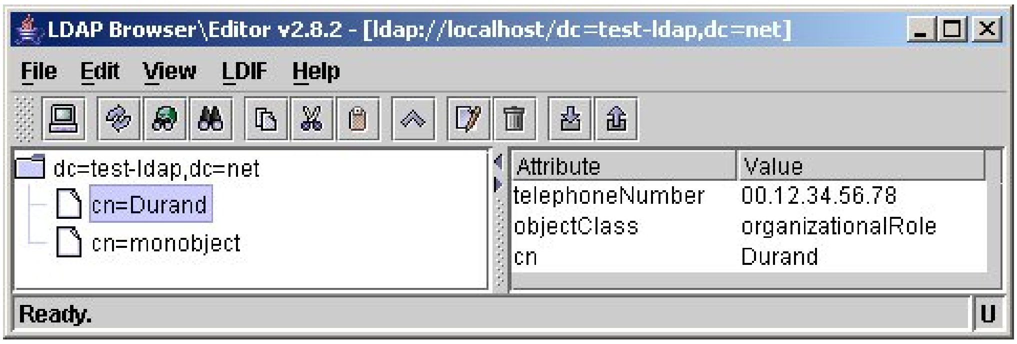

LDAP Browser\Editor v2.8.2 - [ldap://localhost/dc=test-ldap,dc=net]
File
Edit
View
LDIF
Help
dc=test-ldap,dc=net
cn=Durand
cn=monobject
Ready.
U

Attribute	Value
objectClass	organizationalRole
cn	Durand

Exemple :

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.naming.directory.Attribute;
import javax.naming.directory.Attributes;
import javax.naming.directory.BasicAttribute;
import javax.naming.directory.BasicAttributes;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;

public class TestLDAP4 {

  public static void main(String[] args) {
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY,
          "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "ldap://localhost:389");
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    env.put(Context.SECURITY_PRINCIPAL, "cn=ldap-admin,dc=test-ldap,dc=net");
    env.put(Context.SECURITY_CREDENTIALS, "ldap-admin");
    DirContext dirContext;

    try {

      dirContext = new InitialDirContext(env);

      Attributes attributes = new BasicAttributes(true);
      Attribute attribut = new BasicAttribute("telephoneNumber");
      attribut.add("99.99.99.99.99");
      attributes.put(attribut);

      dirContext.modifyAttributes("cn=Durand,dc=test-ldap,dc=net",
            DirContext.ADD_ATTRIBUTE,attributes);
      dirContext.close();
    } catch (NamingException e) {
      System.err.println("Erreur lors de l'acces au serveur LDAP" + e);
      e.printStackTrace();
    }
    System.out.println("fin des traitements");
  }
}
```

Suite à l'exécution de ce programme, l'attribut est ajouté.



Attribute	Value
telephoneNumber	00.12.34.56.78
objectClass	organizationalRole
cn	Durand

Si l'attribut modifié n'est pas défini dans le schéma alors une exception de type javax.naming.directory.SchemaViolationException avec le message « [LDAP: error code 65 - attribute 'xxx' not allowed] » est levée.

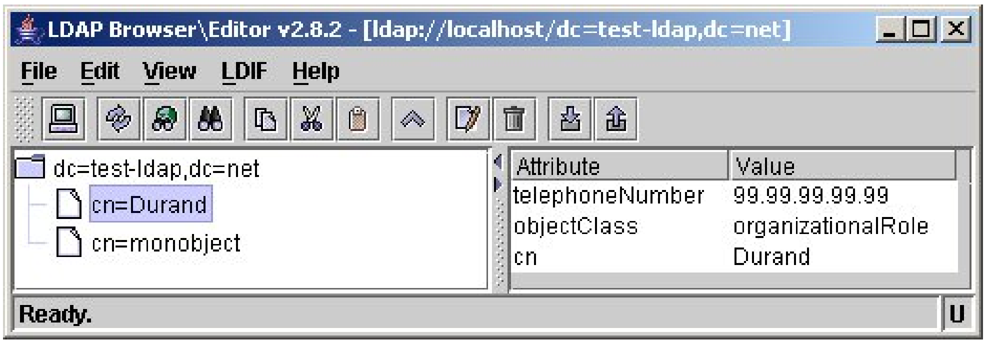
Si l'attribut est ajouté alors qu'il existe déjà, une exception de type javax.naming.directory.AttributeInUseException avec le message « [LDAP: error code 20 - modify/add: xxx: value #0 already exists] » est levée.

La modification d'un attribut est similaire en utilisant le type d'opération REPLACE_ATTRIBUTE

Exemple :

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.naming.directory.Attribute;
import javax.naming.directory.Attributes;
import javax.naming.directory.BasicAttribute;
import javax.naming.directory.BasicAttributes;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;

public class TestLDAP4 {

  public static void main(String[] args) {
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "ldap://localhost:389");
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    env.put(Context.SECURITY_PRINCIPAL, "cn=ldap-admin,dc=test-ldap,dc=net");
    env.put(Context.SECURITY_CREDENTIALS, "ldap-admin");
    DirContext dirContext;

    try {

      dirContext = new InitialDirContext(env);

      Attributes attributes = new BasicAttributes(true);
      Attribute attribut = new BasicAttribute("telephoneNumber");
      attribut.add("99.99.99.99.99");
      attributes.put(attribut);

      dirContext.modifyAttributes("cn=Durand,dc=test-ldap,dc=net",
         DirContext.REPLACE_ATTRIBUTE,attributes);
      dirContext.close();
    } catch (NamingException e) {
      System.err.println("Erreur lors de l'acces au serveur LDAP" + e);
      e.printStackTrace();
    }
    System.out.println("fin des traitements");
  }
}
```

Suite à l'exécution de ce programme, l'attribut est modifié.


LDAP Browser\Editor v2.8.2 - [ldap://localhost/dc=test-ldap,dc=net]
File
Edit
View
LDIF
Help
Ready.
U

dc=test-ldap,dc=net
cn=Durand
cn=monobject

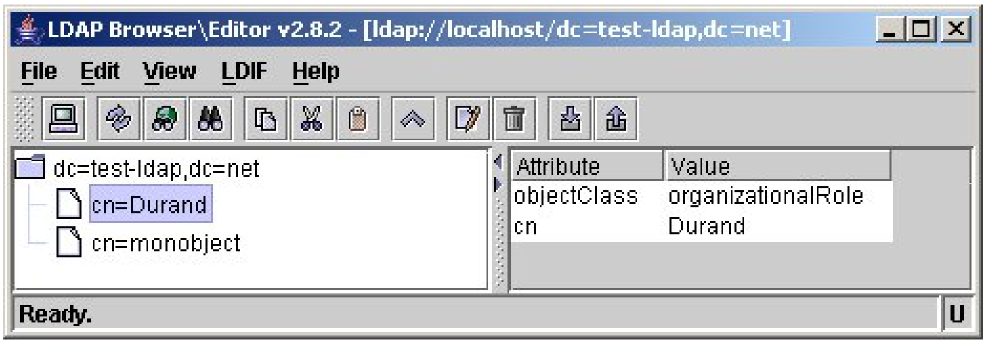
Attribute	Value
telephoneNumber	99.99.99.99.99
objectClass	organizationalRole
cn	Durand

La modification d'un attribut est similaire en utilisant le type d'opération REPLACE_ATTRIBUTE

Exemple :

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.naming.directory.Attribute;
import javax.naming.directory.Attributes;
import javax.naming.directory.BasicAttribute;
import javax.naming.directory.BasicAttributes;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;

public class TestLDAP4 {

  public static void main(String[] args) {
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "ldap://localhost:389");
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    env.put(Context.SECURITY_PRINCIPAL, "cn=ldap-admin,dc=test-ldap,dc=net");
    env.put(Context.SECURITY_CREDENTIALS, "ldap-admin");

    DirContext dirContext;

    try {
      dirContext = new InitialDirContext(env);

      Attributes attributes = new BasicAttributes(true);
      Attribute attribut = new BasicAttribute("telephoneNumber");
      attributes.put(attribut);

      dirContext.modifyAttributes("cn=Durand,dc=test-ldap,dc=net",
         DirContext.REMOVE_ATTRIBUTE,attributes);
      dirContext.close();
    } catch (NamingException e) {
      System.err.println("Erreur lors de l'acces au serveur LDAP" + e);
      e.printStackTrace();
    }
    System.out.println("fin des traitements");
  }
}
```

Suite à l'exécution de ce programme, l'attribut est supprimé.



LDAP Browser\Editor v2.8.2 - [ldap://localhost/dc=test-ldap,dc=net]
File
Edit
View
LDIF
Help
dc=test-ldap,dc=net
cn=Durand
cn=monobject
Ready.
U

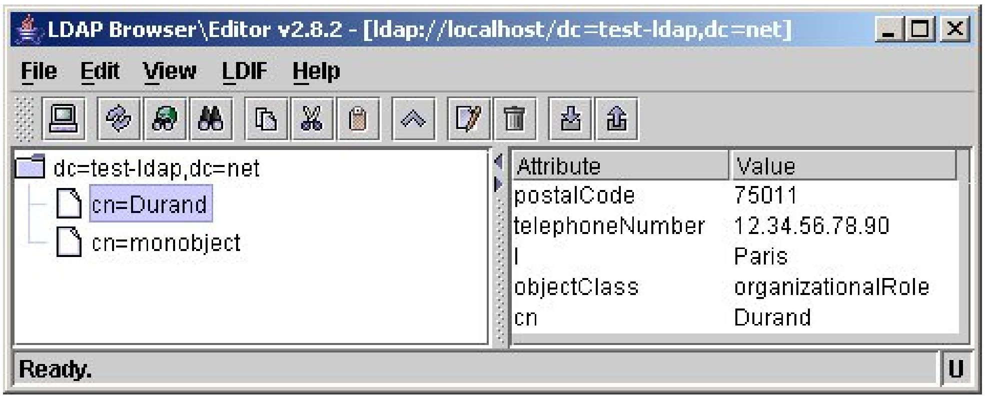
Attribute	Value
objectClass	organizationalRole
cn	Durand

Pour réaliser plusieurs opérations, il est nécessaire d'utiliser un tableau d'objets de type ModificationItem passé en paramètre d'une version surchargée de la méthode modifyAttributes(). Dans ce cas, toutes les modifications sont effectuées ou aucune ne l'est.

Exemple :

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.naming.directory.Attribute;
import javax.naming.directory.BasicAttribute;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;
import javax.naming.directory.ModificationItem;

public class TestLDAP5 {
  public static void main(String[] args) {
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "ldap://localhost:389");
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    env.put(Context.SECURITY_PRINCIPAL, "cn=ldap-admin,dc=test-ldap,dc=net");
    env.put(Context.SECURITY_CREDENTIALS, "ldap-admin");

    DirContext dirContext;
    try {
      dirContext = new InitialDirContext(env);

      ModificationItem[] modifItems = new ModificationItem[3];

      Attribute mod0 = new BasicAttribute("telephonenumber","12.34.56.78.90");
      Attribute mod1 = new BasicAttribute("l", "Paris");
      Attribute mod2 = new BasicAttribute("postalCode", "75011");

      modifItems[0] = new ModificationItem(DirContext.ADD_ATTRIBUTE,mod0);
      modifItems[1] = new ModificationItem(DirContext.ADD_ATTRIBUTE,mod1);
      modifItems[2] = new ModificationItem(DirContext.ADD_ATTRIBUTE,mod2);

      dirContext.modifyAttributes("cn=Durand,dc=test-ldap,dc=net", modifItems);

      dirContext.close();
    } catch (NamingException e) {
      System.err.println("Erreur lors de l'acces au serveur LDAP" + e);
      e.printStackTrace();
    }
    System.out.println("fin des traitements");
  }
}
```

Suite à l'exécution de ce programme, les attributs sont ajoutés.


LDAP Browser\Editor v2.8.2 - [ldap://localhost/dc=test-ldap,dc=net]
File
Edit
View
LDIF
Help
Ready.
U

dc=test-ldap,dc=net
cn=Durand
cn=monobject

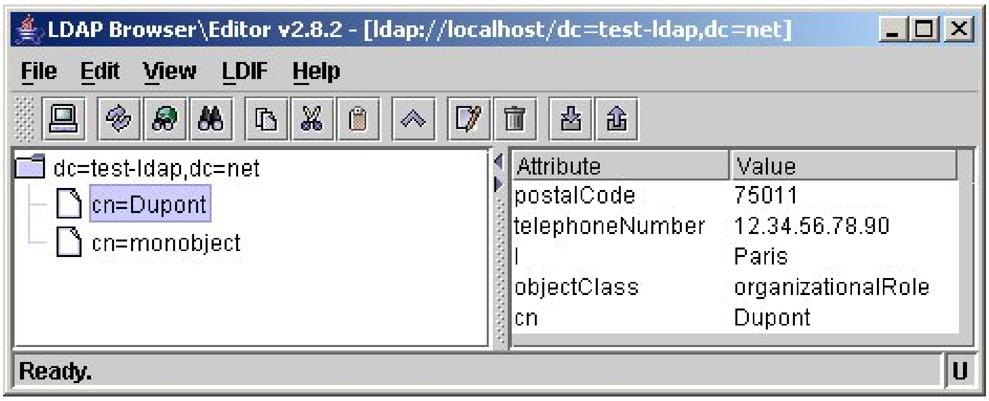
Attribute	Value
postalCode	75011
telephoneNumber	12.34.56.78.90
l	Paris
objectClass	organizationalRole
cn	Durand

La méthode rename() permet de modifier le DN d'une entrée de l'annuaire.

Exemple :

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;

public class TestLDAP6 {

  public static void main(String[] args) {
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "ldap://localhost:389");
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    env.put(Context.SECURITY_PRINCIPAL, "cn=ldap-admin,dc=test-ldap,dc=net");
    env.put(Context.SECURITY_CREDENTIALS, "ldap-admin");
    DirContext dirContext;

    try {
      dirContext = new InitialDirContext(env);
      dirContext.rename("cn=Durand,dc=test-ldap,dc=net",
          "cn=Dupont,dc=test-ldap,dc=net");
      dirContext.close();
    } catch (NamingException e) {
      System.err.println("Erreur lors de l'acces au serveur LDAP" + e);
      e.printStackTrace();
    }
    System.out.println("fin des traitements");
  }
}
```

Suite à l'exécution de ce programme, le DN est modifié.


LDAP Browser\Editor v2.8.2 - [ldap://localhost/dc=test-ldap,dc=net]
File
Edit
View
LDIF
Help
Ready.
U

dc=test-ldap,dc=net
cn=Dupont
cn=monobject

Attribute	Value
postalCode	75011
telephoneNumber	12.34.56.78.90
l	Paris
objectClass	organizationalRole
cn	Dupont

Si le DN à modifier fourni en paramètre n'est pas trouvé dans l'annuaire, une exception de type javax.naming.NameNotFoundException avec le message « [LDAP: error code 32 - No Such Object] » est levée.

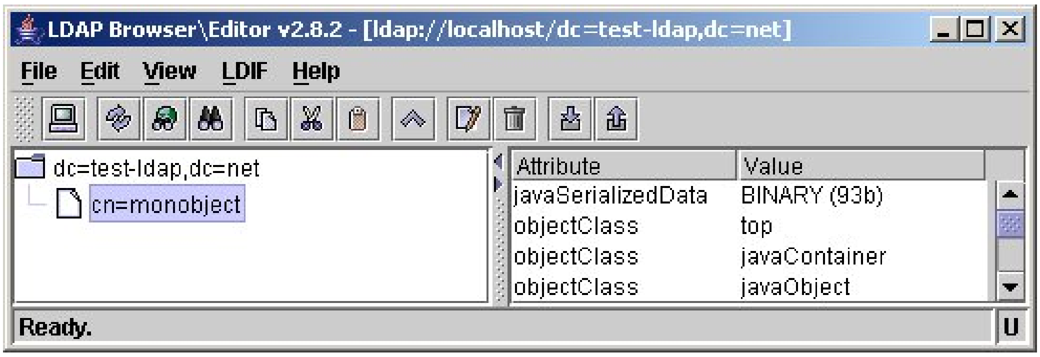
30.7.8. La suppression d'un objet

La méthode unbind() de la classe Context permet de supprimer une association entre un nom et un objet.

Exemple :

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;

public class TestLDAP7 {

  public static void main(String[] args) {
    Hashtable env = new Hashtable();
    env
        .put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "ldap://localhost:389");
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    env.put(Context.SECURITY_PRINCIPAL, "cn=ldap-admin,dc=test-ldap,dc=net");
    env.put(Context.SECURITY_CREDENTIALS, "ldap-admin");
    DirContext dirContext;

    try {
      dirContext = new InitialDirContext(env);
      dirContext.unbind("cn=Dupont,dc=test-ldap,dc=net");
      dirContext.close();
    } catch (NamingException e) {
      System.err.println("Erreur lors de l'acces au serveur LDAP" + e);
      e.printStackTrace();
    }
    System.out.println("fin des traitements");
  }
}
```

Suite à l'exécution de ce programme, l'entrée dans l'annuaire est supprimée.


LDAP Browser\Editor v2.8.2 - [ldap://localhost/dc=test-ldap,dc=net]
File
Edit
View
LDIF
Help
dc=test-ldap,dc=net
cn=monobject
Ready.
U

Attribute	Value
javaSerializedData	BINARY (93b)
objectClass	top
objectClass	javaContainer
objectClass	javaObject

La suppression d'un contexte n'est pas autorisée s'il existe encore un seul sous-contexte. Une demande de suppression portant sur un contexte ayant encore une descendance lèvera une exception de type ContextNotEmptyException avec le message « [LDAP: error code 66 - subtree delete not supported] ».

Exemple :

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;

public class TestLDAP8 {

  public static void main(String[] args) {
    Hashtable env = new Hashtable();
    env
        .put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "ldap://localhost:389");
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    env.put(Context.SECURITY_PRINCIPAL, "cn=ldap-admin,dc=test-ldap,dc=net");
    env.put(Context.SECURITY_CREDENTIALS, "ldap-admin");
    DirContext dirContext;

    try {
      dirContext = new InitialDirContext(env);
      dirContext.destroySubcontext("dc=test-ldap,dc=net");
      dirContext.close();
    } catch (NamingException e) {
      System.err.println("Erreur lors de l'acces au serveur LDAP" + e);
      e.printStackTrace();
    }
    System.out.println("fin des traitements");
  }
}
```

30.7.9. La recherche d'associations

La méthode listBindings() permet d'obtenir une liste des associations nom/objet.

Elle renvoie un objet de type NamingEnumeration qui encapsule des objets de type Binding.

Exemple :

```
import java.util.Hashtable;
import javax.naming.Binding;
import javax.naming.Context;
import javax.naming.NamingEnumeration;
import javax.naming.NamingException;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;

public class TestLDAP13 {

  public static void main(String[] args) {
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "ldap://localhost:389");
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    env.put(Context.SECURITY_PRINCIPAL, "cn=ldap-admin,dc=test-ldap,dc=net");
    env.put(Context.SECURITY_CREDENTIALS, "ldap-admin");
    DirContext dirContext;

    try {

      dirContext = new InitialDirContext(env);
      NamingEnumeration e = dirContext.listBindings("dc=test-ldap,dc=net");

      while (e.hasMore()) {
        Binding b = (Binding) e.next();
        System.out.println("nom    : " + b.getName());
        System.out.println("objet  : " + b.getObject());
        System.out.println("classe : " + b.getObject().getClass().getName());
      }

      dirContext.close();
    } catch (NamingException e) {
      System.err.println("Erreur lors de l'acces au serveur LDAP" + e);
      e.printStackTrace();
    }
    System.out.println("fin des traitements");
  }
}
```

Exemple :

```
nom    : cn=monobject
objet  : MonObjet@1764be1
classe : MonObjet
nom    : cn=Durand
objet  : com.sun.jndi.ldap.LdapCtx@16fd0b7
classe : com.sun.jndi.ldap.LdapCtx
nom    : cn=Pierre
objet  : com.sun.jndi.ldap.LdapCtx@1ef9f1d
classe : com.sun.jndi.ldap.LdapCtx
nom    : cn=Martin
objet  : com.sun.jndi.ldap.LdapCtx@b753f8
classe : com.sun.jndi.ldap.LdapCtx
nom    : cn=Dupont
objet  : com.sun.jndi.ldap.LdapCtx@1e9cb75
classe : com.sun.jndi.ldap.LdapCtx
```

30.7.10. La recherche dans un annuaire LDAP

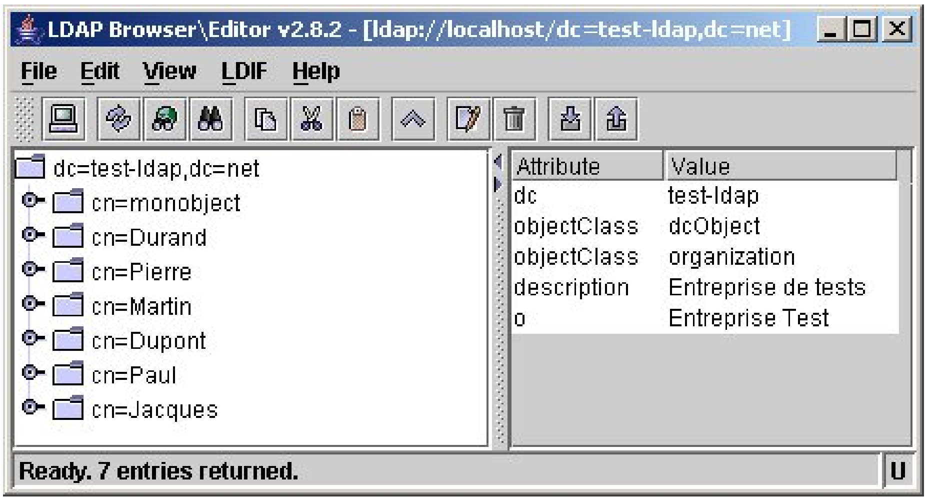
La recherche d'objets et d'informations contenues dans un objet est une des principales actions réalisées sur un annuaire.

La recherche dans un annuaire peut se faire à partir du DN d'un objet mais aussi à partir d'un ou plusieurs attributs. Cette recherche s'effectue grâce à une requête de type filtre qui possède une syntaxe particulière.

La classe DirContext propose deux fonctionnalités pour effectuer des recherches :

- Une recherche à partir du DN
- Une recherche à partir d'un filtre qui permet une recherche avancée (éventuellement sur plusieurs critères)

Les exemples de cette section utilisent le jeu d'essais suivant :



Attribute	Value
dc	test-ldap
objectClass	dcObject
objectClass	organization
description	Entreprise de tests
o	Entreprise Test

Ready. 7 entries returned.

La méthode getAttributes() permet d'obtenir tous les attributs d'un objet à partir de son DN.

Exemple :

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.naming.directory.Attributes;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;

public class TestLDAP10 {
  public static void main(String[] args) {
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "ldap://localhost:389");
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    env.put(Context.SECURITY_PRINCIPAL, "cn=ldap-admin,dc=test-ldap,dc=net");
    env.put(Context.SECURITY_CREDENTIALS, "ldap-admin");
    DirContext dirContext;

    try {
      dirContext = new InitialDirContext(env);

      Attributes attrs = dirContext.getAttributes("cn=Dupont,dc=test-ldap,dc=net");
      System.out.println("Description : " + attrs.get("description").get());

      dirContext.close();
    } catch (NamingException e) {
      System.err.println("Erreur lors de l'acces au serveur LDAP" + e);
      e.printStackTrace();
    }
    System.out.println("fin des traitements");
  }
}
```

Résultat :

```
  Description : Directeur
  fin des traitements
```

Ceci impose de connaître le DN de l'objet. JNDI propose la possibilité de rechercher un ou plusieurs objets en utilisant un filtre.

Il est possible de faire une recherche sur un ou plusieurs attributs. Cette recherche se fait en utilisant la méthode search().

Deux surcharges de la méthode search permettent la recherche à partir d'attributs :

- NamingEnumeration search(String stringName, Attributes attributesToMatch)
- NamingEnumeration search(String stringName, Attributes attributesToMatch, String [] rgstringAttributesToReturn)

Les deux méthodes permettent de retrouver un objet dont le nom est fourni en paramètre et qui possède en plus les attributs précisés.

La seconde méthode permet aussi de préciser un tableau des attributs renvoyés dans les résultats de la recherche.

Exemple :

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.NamingEnumeration;
import javax.naming.NamingException;
import javax.naming.directory.Attributes;
import javax.naming.directory.BasicAttribute;
import javax.naming.directory.BasicAttributes;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;
import javax.naming.directory.SearchResult;

public class TestLDAP11 {
  public static void main(String[] args) {
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "ldap://localhost:389");
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    env.put(Context.SECURITY_PRINCIPAL, "cn=ldap-admin,dc=test-ldap,dc=net");
    env.put(Context.SECURITY_CREDENTIALS, "ldap-admin");
    DirContext dirContext;

    try {
      dirContext = new InitialDirContext(env);
      Attributes matchattribs = new BasicAttributes(true);
      matchattribs.put(new BasicAttribute("description", "Employe"));
      NamingEnumeration resultat = dirContext.search("dc=test-ldap,dc=net", matchattribs);

      while (resultat.hasMore()) {
        SearchResult sr = (SearchResult)resultat.next();
        System.out.println("Description : " + sr.getAttributes().get("cn").get());
      }

      dirContext.close();
    } catch (NamingException e) {
      System.err.println("Erreur lors de l'acces au serveur LDAP" + e);
      e.printStackTrace();
    }
    System.out.println("fin des traitements");
  }
}
```

Résultat :

```
  Description : Pierre
  Description : Paul
  Description : Jacques
  fin des traitements
```

La recherche peut se faire à partir d'un filtre dont les spécifications sont définies dans la RFC 2254.

Le filtre est une expression logique qui précise les critères de recherche. La syntaxe de ce filtre est composée de conditions utilisées avec des opérateurs logiques. Un opérateur doit être précisé avant la ou les conditions sur lesquelles il agit. La syntaxe est donc de la forme :

```
(operateur(condition)(condition)...))
```

Opérateur	**Condition**	**Exemple**	**Description**
=	Egalité	`(sn=test)`	tous les objets dont l'attribut sn vaut test
>	Plus grand que	`(sn>test)`	tous les objets dont l'attribut sn est alphabétiquement plus grand que test
>=	Plus grand ou égal à	`(sn>=test)`	tous les objets dont l'attribut sn est alphabétiquement plus grand ou égal à test
<	Plus petit que	`(sn<test)`	tous les objets dont l'attribut sn est alphabétiquement plus petit que test
<=	Plus petit ou égal à	`(sn<=test)`	tous les objets dont l'attribut sn est alphabétiquement plus petit ou égal à test
=*	Est présent	`(sn=*)`	tous les objets possédant un attribut sn

*	Aucun ou plusieurs caractères quelconques	`(sn=test*),`
		`(sn=*test*),`
		`(sn=*test)`
			respectivement tous les objets dont l'attribut sn commence par test, contient test ou termine par test

&	ET	`(&(sn=test) (cn=test))`	tous les objets dont l'attribut sn et cn valent test
|	OU	`(|(sn=test) (cn=test))`	tous les objets dont l'attribut sn ou cn valent test
!	NON	`(!(sn=test))`	tous les objets dont l'attribut sn est différent de test

Quatre autres surcharges de la méthode search() permettent de faire une recherche à partir d'un filtre.

- NamingEnumeration search(Name name, String filterExpr, Object[] filterArgs, SearchControls cons)
- NamingEnumeration search(String name, String filterExpr, Object[] filterArgs, SearchControls cons)

- NamingEnumeration search(Name name, String filter, SearchControls cons)
- NamingEnumeration search(String name, String filter, SearchControls cons)

La classe SearchControls encapsule des informations de contrôle sur la recherche à effectuer notamment :

- searchScope : la portée de la recherche (OBJECT_SCOPE, ONELEVEL_SCOPE, SUBTREE_SCOPE)
- countLimit : le nombre maximum d'occurrences renvoyées par la recherche
- timeLimit : durée maximale en millisecondes de la recherche
- returningAttributes : tableau des attributs retourné par la recherche
- returningObjFlag : précise si les objets correspondant à la recherche sont retournés dans les résultats

Le résultat de la recherche est encapsulé dans un objet de type NamingEnumeration : cet objet est une énumération d'objets de type SearchResult.

Exemple :

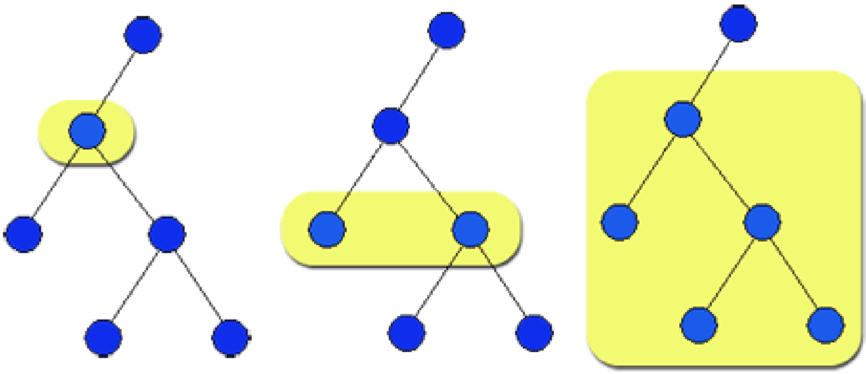
```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.NamingEnumeration;
import javax.naming.NamingException;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;
import javax.naming.directory.SearchControls;
import javax.naming.directory.SearchResult;

public class TestLDAP12 {

  public static void main(String[] args) {
    Hashtable env = new Hashtable();

    env.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "ldap://localhost:389");
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    env.put(Context.SECURITY_PRINCIPAL, "cn=ldap-admin,dc=test-ldap,dc=net");
    env.put(Context.SECURITY_CREDENTIALS, "ldap-admin");
    DirContext dirContext;

    try {
      dirContext = new InitialDirContext(env);
      SearchControls searchControls = new SearchControls();
      searchControls.setSearchScope(SearchControls.SUBTREE_SCOPE);
      NamingEnumeration resultat = dirContext.search("dc=test-ldap,dc=net",
            "(cn=Martin)", searchControls);

      while (resultat.hasMore()) {
        SearchResult sr = (SearchResult)resultat.next();
        System.out.println("Description : " + sr.getAttributes().get("cn").get()
           +", "+sr.getAttributes().get("description").get());
      }

      dirContext.close();
    } catch (NamingException e) {
      System.err.println("Erreur lors de l'acces au serveur LDAP" + e);
      e.printStackTrace();
    }
    System.out.println("fin des traitements");
  }
}
```

Résultat :

```
  Description : Martin, Chef d'equipe
  fin des traitements
```

Lors d'une recherche, il faut préciser le noeud de départ (base object) de la recherche et la portée de cette recherche (scope). La portée permet de définir les noeuds concernés par la recherche. Trois portées de recherche sont définies :

Portée	Définition
OBJECT_SCOPE	Cette portée ne concerne que le noeud de départ lui-même. Cette portée est utile pour rechercher des attributs sur un objet
ONELEVEL_SCOPE	Cette portée concerne tous les noeuds d'un même niveau
SUBTREE_SCOPE	C'est la portée la plus grande puisqu'elle inclut le noeud de départ et tous ses noeuds fils



OBJECT_SCOPE ONELEVEL_SCOPE SUBTREE_SCOPE

Exemple :

```
    SearchControls ctls = new SearchControls();
    ctls.setSearchScope(SearchControls.SUBTREE_SCOPE);
```

Par défaut, tous les attributs des objets trouvés sont retournés. Il est possible de limiter les attributs retournés en créant un tableau des clés des attributs concernés. Il suffit alors de passer ce paramètre à la méthode setReturningAttributes() de l'instance de la classe SearchControls.

Exemple :

```
      String[] attributIDs = {"cn", "description"};
      searchControls.setReturningAttributes(attributIDs);
```

Il est possible de limiter le nombre d'objets retournés dans le résultat de la recherche. Il suffit de fournir en paramètre de la méthode setCountLimit() de l'instance de la classe SearchControls le nombre maximum d'objets retournés.

Exemple :

```
      searchControls.setCountLimit(1);
```

Attention : une exception de type javax.naming.SizeLimitExceededException avec le message « [LDAP: error code 4 - Sizelimit Exceeded] » est levée si la limite est dépassée par le nombre d'objets trouvés.

30.8. JNDI et J2EE/Java EE

J2EE utilise énormément JNDI de façon implicite ou explicite notamment pour proposer des références vers des ressources nécessaires aux applications.

Chaque conteneur J2EE utilise en interne un service accessible par JNDI pour stocker des informations sur les applications et les composants. Généralement l'utilisation de JNDI dans une application J2EE se fait en utilisant ce service du conteneur.

Ces informations sont essentiellement des données de configuration : interface Home des EJB, DataSource pour accès à des bases de données, ... Ceci permet de rendre dynamique la recherche de composants de l'application.

Plusieurs technologies mises en oeuvre dans J2EE font un usage de JNDI : par exemple JDBC, EJB, JMS, ...

JDBC utilise JNDI pour stocker des objets de type DataSource qui encapsulent les informations utiles à la connexion à la source de données. Cette utilisation a été proposée à partir du package optionnel JDBC 2.0. Son utilisation n'est pas obligatoire mais elle est fortement recommandée.

Comme JDBC, JMS recommande de stocker les informations concernant les files (queues) et les sujets (topics) dans un annuaire et de les rechercher grâce à JNDI.

Les EJB stockent aussi leur référence vers leur interface home dans l'annuaire du serveur d'applications pour permette à un client d'obtenir une référence sur l'EJB.

Pour permettre de standardiser les pratiques, J2EE propose dans ses spécifications des règles de nommage pour certains objets ou composants J2EE dans l'annuaire.

31. Le scripting

Chapitre 31

Niveau : ■■■■□□ Supérieur

Le scripting est utilisé depuis longtemps, dans un premier temps, pour automatiser certaines tâches sur des systèmes d'exploitation (exemple le shell sous Linux) puis sous la forme de langages de développement (exemple Perl, Python, ...)

Ces langages n'ont pas pour but de remplacer le langage Java mais ils peuvent avoir une place de choix pour remplir certaines tâches et permettre de bénéficier des points forts de Java et du scripting.

Les langages de scripting possèdent plusieurs caractéristiques qui peuvent être intéressantes :

- ils sont généralement typés dynamiquement : il n'est pas nécessaire de fournir le type lors de la déclaration d'une variable et la valeur contenue peut changer de type
- certains peuvent être compilés mais la plupart sont interprétés
- ils permettent de personnaliser certaines parties d'une application comme la configuration ou les règles métiers

La plate-forme Java permet depuis longtemps d'utiliser des langages de scripts notamment avec des solutions open source comme BeanShell.

Java 6 intègre une API standard, indépendante du langage de scripting utilisé du moment qu'il est compatible avec l'API.

Certains langages de scripting ont été spécifiquement développés pour la JVM : c'est notamment le cas de Groovy.

Ce chapitre contient une section :

♦ L'API Scripting

31.1. L'API Scripting

Java SE 6.0 intègre la possibilité d'utiliser des moteurs de scripting suite à l'intégration des spécifications de la JSR 223.

La JSR 223 a pour but d'intégrer des possibilités de scripting dans les applications Java en permettant :

- L'intégration de moteurs de scripting
- La possibilité pour ces moteurs d'accéder à la plate-forme Java
- L'ajout d'une console permettant l'exécution de scripts en mode ligne de commande (jrunscript)

Les classes et interfaces de cette fonctionnalité sont regroupées dans le package javax.script.

L'API propose un support pour tous les moteurs de scripting compatibles avec elle.

Java SE 6.0 intègre en standard le moteur de scripting Rhino version 1.6 R2 qui propose un support pour le langage Javascript.

La gestion des moteurs utilisables se fait grâce à la classe ScriptEngineManager : elle permet d'obtenir la liste des objets de type ScriptEngineFactory de chaque moteur de scripting installé. Ces méthodes ne sont pas statiques, il est donc nécessaire d'instancier un objet de type ScriptEngineManager pour les utiliser.

31.1.1. La mise en oeuvre de l'API

Des fabriques permettent l'instanciation d'un objet de type ScriptEngine qui encapsule le moteur de scripting.

Exemple :

```
package fr.jmdoudoux.dej.java6;

import java.util.List;
import javax.script.ScriptEngineFactory;
import javax.script.ScriptEngineManager;

public class ListerScriptEngine {

  public static void main(String args[]) {
    ScriptEngineManager manager = new ScriptEngineManager();
    List<ScriptEngineFactory> factories = manager.getEngineFactories();

    for (ScriptEngineFactory factory : factories) {
      System.out.println("Name : " + factory.getEngineName());
      System.out.println("Version : " + factory.getEngineVersion());
      System.out.println("Language name : " + factory.getLanguageName());
      System.out.println("Language version : " + factory.getLanguageVersion());
      System.out.println("Extensions : " + factory.getExtensions());
      System.out.println("Mime types : " + factory.getMimeTypes());
      System.out.println("Names : " + factory.getNames());
    }
  }
}
```

Résultat :

```
Name : Mozilla Rhino
Version : 1.6 release 2
Language name : ECMAScript
Language version : 1.6
Extensions : [js]
Mime types:[application/javascript, application/ecmascript, text/javascript, text/ecmascript]
Names : [js, rhino, JavaScript, javascript, ECMAScript, ecmascript]
```

Les propriétés Extensions, MimeType et Names sont importantes car elles sont utilisées pour obtenir une instance de la classe ScriptEngine.

Le ScriptEngineManager permet d'obtenir directement une instance du moteur de scripting à partir d'un nom, d'une extension et d'un type mime particulier respectivement grâce aux méthodes getEngineByName(), getEngineByExtension() et getEngineByMimeType().

Exemple :

```
package fr.jmdoudoux.dej.java6;

import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;

public class TestScriptEngine {
  public static void main(String args[]) {
    ScriptEngineManager manager = new ScriptEngineManager();
    ScriptEngine moteur1 = manager.getEngineByName("rhino");
    ScriptEngine moteur2 = manager.getEngineByExtension("js");
    ScriptEngine moteur3 = manager.getEngineByName("test");
    if (moteur3== null) {
      System.out.println("Impossible de trouver le moteur test ");
    }
  }
}
```

Si aucune fabrique ne correspond au paramètre fourni alors l'instance de type ScriptEngine retournée est null.

31.1.2. Ajouter d'autres moteurs de scripting

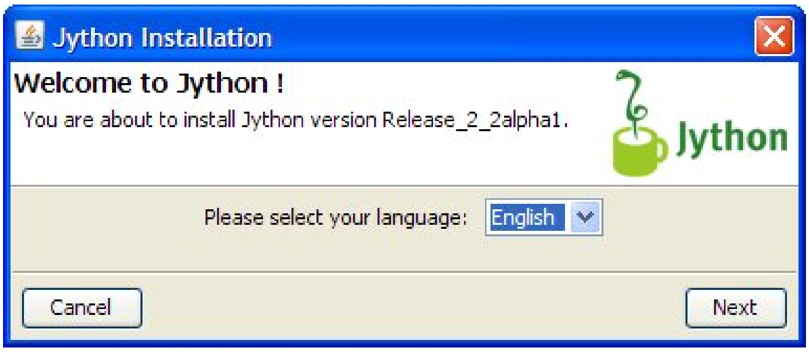
Il est possible d'ajouter d'autres moteurs de scripting. Le projet scripting hébergé par java.net propose l'encapsulation de nombreux moteurs de scripting pour l'utilisation avec l'API Scripting. https://scripting.dev.java.net/

Il faut télécharger le fichier jsr223-engines.zip. Cette archive contient un répertoire pour chaque moteur. Il faut ajouter le fichier build/xxx-engine.jar au classpath où xxx est le nom du moteur.

Résultat de l'exécution :

```
Name : Mozilla Rhino
Version : 1.6 release 2
Language name : ECMAScript
Language version : 1.6
Extensions : [js]
Mime types:[application/javascript, application/ecmascript, text/javascript, text/ecmascript]
Names : [js, rhino, JavaScript, javascript, ECMAScript, ecmascript]
Name : jython
Version : 2.1
Language name : python
Language version : 2.1
Extensions : [jy, py]
Mime types : []
Names : [jython, python]
```

Il est nécessaire pour instancier le moteur que celui-ci soit présent dans le classpath. Dans le cas de jython, il faut ajouter le fichier jython.jar dans le classpath. Pour cela, il faut télécharger le fichier jython_Release_2_2alpha1.jar et l'exécuter en double cliquant dessus. Le programme d'installation utilise un assistant :

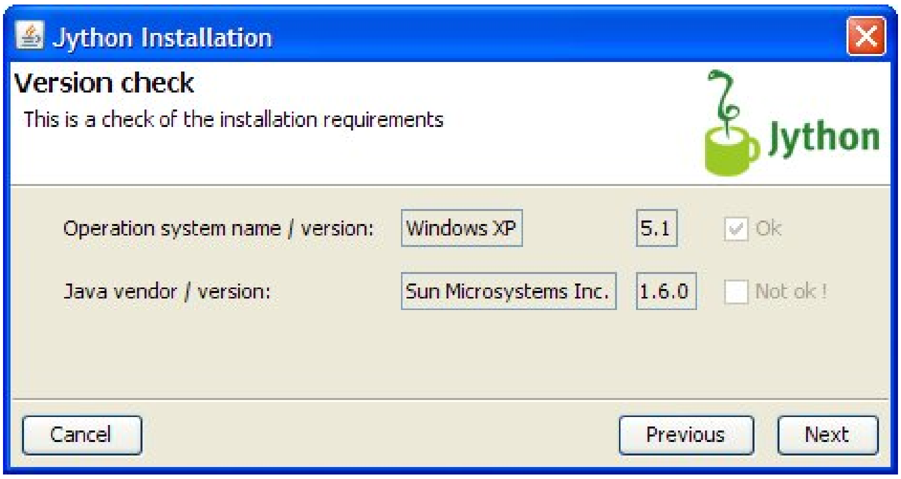


Welcome to Jython !

You are about to install Jython version Release_2_2alpha1.

Please select your language: English

- Sur la page « Welcome to Jython », cliquez sur le bouton « Next »
- Sur la page « Installation type », laissez All sélectionné et cliquez sur Next



Operation system name / version: Windows XP 5.1 ☑ Ok

Java vendor / version: Sun Microsystems Inc. 1.6.0 □ Not ok !

- Sur la page « Version check », cliquez sur Next
- Sur la page « License agreement », lisez la licence et si vous l'acceptez cliquez sur « I accept » et sur le bouton « Next »
- Sur la page « Target directory » modifiez le répertoire d'installation au besoin et cliquez sur Next. Si le répertoire n'existe pas, cliquez sur OK puis de nouveau sur le bouton Next
- Sur la page « Overview (summary of options) », cliquez sur Next pour démarrer l'installation
- Sur la page « Read me », cliquez sur Next
- Cliquez sur Finish pour terminer l'installation.

Ajoutez le fichier jython.jar contenu dans le répertoire d'installation au classpath de l'application. Il est alors possible de créer une instance du moteur de script Jython.

Exemple :

```
package fr.jmdoudoux.dej.java6;

import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;

public class TestJython {
  public static void main(String args[]) {
    try {
      ScriptEngineManager manager = new ScriptEngineManager();
      ScriptEngine moteur = manager.getEngineByName("jython");
      if (moteur== null) {
        System.out.println("Impossible de trouver le moteur jython ");
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Si le fichier jython.jar n'est pas présent dans le classpath une exception est levée.

Exemple :

```
Exception in thread "main" java.lang.NoClassDefFoundError: org/python/core/PyObject
at com.sun.script.jython.JythonScriptEngineFactory.getScriptEngine(
JythonScriptEngineFactory.java:132)
at javax.script.ScriptEngineManager.getEngineByName(ScriptEngineManager.java:225)
at fr.jmdoudoux.dej.java6.TestJython.main(TestJython.java:11)
```

31.1.3. L'évaluation d'un script

La classe ScriptEngine propose plusieurs surcharges de la méthode eval() pour exécuter un script. Ces surcharges attendent en paramètre le script sous la forme d'une chaîne de caractères ou d'un flux de type Reader.

Exemple :

```
package fr.jmdoudoux.dej.java6;

import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;

public class TestJython {
  public static void main(String args[]) {
    try {
      ScriptEngineManager manager = new ScriptEngineManager();
      ScriptEngine moteur = manager.getEngineByName("jython");
      if (moteur == null) {
        System.out.println("Impossible de trouver le moteur jython ");
      } else {
        moteur.eval("print \"test\"");
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

La méthode eval() peut lever une exception de type javax.script.ScriptExceptionsi le moteur détecte une erreur dans le script.

Exemple :

```
package fr.jmdoudoux.dej.java6;

import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class TestRhino {
  public static void main(String args[]) {
    ScriptEngineManager manager = new ScriptEngineManager();
    ScriptEngine moteur = manager.getEngineByName("rhino");
    try {
      moteur.eval("alert('test');");
    } catch (ScriptException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
javax.script.ScriptException: sun.org.mozilla.javascript.internal.EcmaError: ReferenceError:
"Alert" n'est pas défini (<Unknown source>#1) in <Unknown source> at line number 1
at com.sun.script.javascript.RhinoScriptEngine.eval(RhinoScriptEngine.java:110)
at com.sun.script.javascript.RhinoScriptEngine.eval(RhinoScriptEngine.java:124)
at javax.script.AbstractScriptEngine.eval(AbstractScriptEngine.java:247)
at fr.jmdoudoux.dej.java6.TestRhino.main(TestRhino.java:13)
```

Deux surcharges de la méthode eval() attendent en paramètre un objet de type Bindings. C'est une paire clé/valeur qui permet de passer des objets Java au script.

Exemple :

```
package fr.jmdoudoux.dej.java6;

import javax.script.Bindings;
import javax.script.ScriptContext;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;
```

```
public class TestBindings {

  public static void main(String args[]) {
    ScriptEngineManager manager = new ScriptEngineManager();
    ScriptEngine moteur = manager.getEngineByName("rhino");
    try {
      Bindings bindings = moteur.getBindings(ScriptContext.ENGINE_SCOPE);
      bindings.clear();
      bindings.put("entree","valeur");
      moteur.eval("var sortie = '';"+
        " sortie = entree + ' modifiee '", bindings);
      String resultat = (String)bindings.get("sortie");
      System.out.println("resultat = "+resultat);
    } catch (ScriptException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
resultat = valeur modifiee
```

La classe ScriptEngine possède deux méthodes pour faciliter l'utilisation des Bindings : les méthodes put() et get() pour respectivement passer un objet au script et obtenir un objet du script.

Exemple :

```
package fr.jmdoudoux.dej.java6;

import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class TestBindings2 {
  public static void main(String args[]) {
    ScriptEngineManager manager = new ScriptEngineManager();
    ScriptEngine moteur = manager.getEngineByName("rhino");
    try {
      moteur.put("entree","valeur");
      moteur.eval("var sortie = '';"+
        " sortie = entree + ' modifiee '");
      String resultat = (String)moteur.get("sortie");
      System.out.println("resultat = "+resultat);
    } catch (ScriptException e) {
      e.printStackTrace();
    }
  }
}
```

Deux surcharges de la méthode eval() attendent en paramètre un objet de type ScriptContext qui permet de préciser la portée des Bindings.

Il existe deux portées prédéfinies :

- ScriptContext.GLOBAL_SCOPE : portée pour tous les moteurs
- ScriptContext.ENGINE_SCOPE : portée pour le moteur courant uniquement

Il est possible de préciser le contexte par défaut du moteur en utilisant la méthode SetContext() de la classe ScriptEngine.

La méthode getBindings() permet d'obtenir les bindings pour la portée fournie en paramètre.

31.1.4. L'interface Compilable

Les scripts sont généralement interprétés : ils doivent donc être lus, validés et évalués avant d'être exécutés. Ces opérations peuvent être coûteuses en ressources et en temps.

L'interface Compilable propose de compiler ces scripts afin de rendre leurs exécutions plus rapides. L'implémentation de cette interface par un moteur de scripting est optionnelle : il faut donc vérifier que l'instance du moteur de scripting implémente cette interface et la caster vers le type Compilable avant d'utiliser cette fonctionnalité.

La méthode compile() réalise une compilation du script et retourne un objet de type CompiledScript en cas de succès.

Le script compilé est exécuté avec la méthode eval() de la classe CompiledScript.

Exemple :

```
package fr.jmdoudoux.dej.java6;

import javax.script.Bindings;
import javax.script.Compilable;
import javax.script.CompiledScript;
import javax.script.ScriptContext;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class TestCompilable {

  public static void main(String args[]) {
    ScriptEngineManager manager = new ScriptEngineManager();
    ScriptEngine moteur = manager.getEngineByName("rhino");
    try {
      Bindings bindings = moteur.getBindings(ScriptContext.ENGINE_SCOPE);
      bindings.clear();
      bindings.put("compteur", 1);

      if (moteur instanceof Compilable) {
        Compilable moteurCompilable = (Compilable) moteur;
        CompiledScript scriptCompile = moteurCompilable
          .compile("var sortie = '';"
          + "sortie = 'chaine' + compteur;"
          + "compteur++;");

        for (int i = 1; i < 11; i++) {
          scriptCompile.eval(bindings);
          String resultat = (String) bindings.get("sortie");
          System.out.println("valeur " + i + " = " + resultat);
        }
      } else {
      System.err
        .println("Le moteur n'implemente pas l'interface Compilable");
      }
    } catch (ScriptException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
valeur 1 = chaine1
valeur 2 = chaine2
valeur 3 = chaine3
valeur 4 = chaine4
valeur 5 = chaine5
valeur 6 = chaine6
valeur 7 = chaine7
valeur 8 = chaine8
valeur 9 = chaine9
valeur 10 = chaine10
```

L'utilisation de cette fonctionnalité est particulièrement intéressante pour des exécutions répétées du script.

31.1.5. L'interface Invocable

Cette interface permet d'invoquer une fonction définie dans le code source du script.

Dès qu'une fonction a été évaluée par le moteur de scripting, elle peut être invoquée grâce à la méthode invoke() de l'interface Invocable. L'implémentation de cette interface par un moteur de scripting est optionnelle : il faut donc vérifier avant d'utiliser cette fonctionnalité si le moteur implémente cette interface.

Il est possible de fournir des paramètres à la fonction invoquée.

Exemple :

```
package fr.jmdoudoux.dej.java6;

import javax.script.Bindings;
import javax.script.Invocable;
import javax.script.ScriptContext;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class TestInvocable {

  public static void main(String args[]) {
    ScriptEngineManager manager = new ScriptEngineManager();
    ScriptEngine moteur = manager.getEngineByName("rhino");
    try {

      moteur.eval("function afficher(valeur) {"
        + "var sortie = '';"
        + "sortie = 'chaine' + valeur;"
        + "return sortie;"
        + "}");

      if (moteur instanceof Invocable) {
        Invocable moteurInvocable = (Invocable) moteur;
        Object resultat = moteurInvocable.invokeFunction("afficher",
          new Integer(10));
        System.out.println("resultat = " + resultat);
      } else {
        System.err.println("Le moteur n'implemente pas l'interface Invocable");
      }

    } catch (ScriptException e) {
      e.printStackTrace();
    } catch (NoSuchMethodException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
resultat = chaine10
```

La méthode getInterface() de l'interface Invocable permet d'obtenir dynamiquement un objet dont la ou les méthodes sont codées dans le script.

L'exemple ci-dessous va définir une fonction run() qui sera invoquée dans un thread en utilisant la méthode getInterface() avec en paramètre un objet de type Class qui encapsule la classe Runnable

Exemple :

```
package fr.jmdoudoux.dej.java6;

import javax.script.Invocable;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class TestInvocable2 {

  public static void main(String args[]) {
    ScriptEngineManager manager = new ScriptEngineManager();
    ScriptEngine moteur = manager.getEngineByName("rhino");
    try {

      moteur.eval("function run(){"
        + "for (i = 0 ; i < 1000 ; i ++) {"
        + "print('run'+i);"
        + "}"
        + "}");

      if (moteur instanceof Invocable) {
        Invocable moteurInvocable = (Invocable) moteur;
        Runnable runnable = moteurInvocable.getInterface(Runnable.class);
        Thread thread = new Thread(runnable);
        thread.start();
        for (int i = 0; i < 1000; i++) {
          System.out.println("main" + i);
        }
        thread.join();
      } else {
        System.err.println("Le moteur n'implemente pas l'interface Invocable");
      }

    } catch (ScriptException e) {
      e.printStackTrace();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

Extrait du résultat :

```
main988
run174run175main989
main990
main991
main992
main993
main994
main995
main996
main997
main998
main999
run176run177run178run179run180
```

L'itération du programme s'exécute beaucoup plus rapidement que le thread : l'appel à la méthode join() du thread permet d'attendre la fin de son exécution avant de terminer l'application.

31.1.6. La commande jrunscript

La commande jrunscript est un outil du JDK utilisable en ligne de commande qui permet d'exécuter des scripts.

Remarque : pour pouvoir utiliser cette commande, il est nécessaire d'ajouter dans le path le chemin du répertoire bin du JDK.

Les options -help et -? permettent d'obtenir la liste des options de la commande.
L'option -q permet de connaître la liste des moteurs de scripting utilisables.
Les options -cp et -classpath permettent de préciser le classpath qui sera utilisé.

Si seul le moteur de scripting par défaut est installé alors il n'est pas utile de préciser le moteur à utiliser. Dans le cas contraire, il est nécessaire de préciser le moteur à utiliser grâce à l'option -l

Exemple :

```
C:\>jrunscript -q
Language ECMAScript 1.6 implemention "Mozilla Rhino" 1.6 release 2
```

Sans autre argument, la commande jrunscript affiche un prompt qui permet de saisir le script à évaluer.

Exemple :

```
C:\>jrunscript
js> var i = 10* 10;
js> i;
100.0
js> i
100.0
js> i++;
100.0
js> i
101.0
js> print i;
script error: sun.org.mozilla.javascript.internal.EvaluatorException: il manque
';' avant une instruction (<STDIN>#1) in <STDIN> at line number 1
js>
```

32. JMX (Java Management Extensions)

Chapitre 32

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

JMX est l'acronyme de Java Management Extensions. Historiquement, cette API se nommait JMAPI (Java Management API). La version 5.0 de Java a ajouté l'API JMX 1.2 dans la bibliothèque de classes standard.

JMX est une spécification qui définit une architecture, une API et des services pour permettre de surveiller et de gérer des ressources en Java. JMX permet de mettre en place, en utilisant un standard, un système de surveillance et de gestion d'une application, d'un service ou d'une ressource sans avoir à fournir beaucoup d'efforts.

JMX permet de construire et de mettre en oeuvre une solution de gestion de ressources sous une forme modulaire grâce à des composants. Son but est de proposer un standard pour faciliter le développement de systèmes de contrôle, d'administration et de supervision des applications et des ressources.

JMX peut permettre de configurer, gérer et maintenir une application durant son exécution en fonction des fonctionnalités développées. Il peut aussi favoriser l'anticipation de certains problèmes par une information sur les événements critiques de l'application ou du système.

Java SE version 5.0 intègre JMX 1.2 et JMX Remote API 1.0. Il existe aussi une implémentation téléchargeable indépendamment pour J2SE 1.4

La plupart des principaux serveurs d'applications Java EE utilisent JMX pour la surveillance et la gestion de leurs composants.

Ce chapitre contient plusieurs sections :

- La présentation de JMX
- L'architecture de JMX
- Un premier exemple
- La couche instrumentation : les MBeans
- Les MBeans standard
- La couche agent
- Les services d'un agent JMX
- La couche services distribués
- Les notifications
- Les Dynamic MBeans
- Les Model MBeans
- Les Open MBeans
- Les MXBeans
- L'interface PersistentMBean
- Le monitoring d'une JVM
- Des recommandations pour l'utilisation de JMX
- Des ressources

32.1. La présentation de JMX

JMX est une spécification : pour la mettre oeuvre, il faut obligatoirement utiliser une implémentation : Sun propose une implémentation de référence, mais il est aussi possible d'utiliser JBossMX ou MX4J par exemple.

JMX propose une API standard qui permet de gérer, de contrôler et de surveiller des applications, des composants, des services et même la JVM ou des périphériques dans la plate-forme Java.

Ainsi les utilisations possibles de JMX sont nombreuses, par exemple :

- consulter et modifier les paramètres de la configuration
- calculer et diffuser des statistiques d'utilisation
- émettre des événements lors de changements d'état ou d'erreurs
- ...

JMX est architecturé en couches ce qui permet de séparer les responsabilités des différents composants utilisés. Ceci permet aussi d'assurer une meilleure extensibilité.

L'architecture de JMX est composée de trois niveaux :

- Instrumentation : les ressources sont instrumentées grâce à des objets de type MBean
- Agent : les MBeans enregistrés sont gérés par le MBeanServer d'un agent JMX
- Distributed services : une IHM, par exemple sous la forme d'une application tierce de gestion des ressources, interagit avec les MBeans grâce à l'agent JMX

L'architecture de JMX permet de faire de l'administration en locale ou à distance. JMX offre un accès distant pour permettre à une application de gestion d'interagir avec l'application instrumentée grâce à l'agent et aux MBeans.

JMX est spécifiée dans plusieurs JSR :

- JSR 003 : Java Management Extensions Instrumentation and Agent Specification
- JSR 160 : Java Management Extensions Remote API
- JSR 174 : Monitoring and management Specification for the JVM
- JSR 255 : version 2.0 de JMX
- JSR 262 : Web services connector for JMX agents
- JSR 146 : WBEM services : JMX provider protocol adapter
- JSR 070 : IIOP protocol adaptor for JMX

La version 1.1 des spécifications de JMX ne détaille que la partie Instrumentation et Agent.

Tout programme peut bénéficier de l'instrumentation par JMX, simplement en ajoutant du code aux classes existantes. Cette instrumentation concerne des propriétés, des opérations et des événements qui peuvent être exposés aux agents.

JMX est largement utilisée notamment dans les serveurs d'applications par exemple.

La plupart des applications et notamment celles exécutées côté serveur ont besoin d'être administrées pour :

- obtenir des informations sur l'activité de l'application (audit)
- modifier des paramètres de configuration sans redémarrer l'application (configuration)
- anticiper de futurs problèmes selon la valeur de certains indicateurs (surveillance/monitoring)

L'avantage de JMX est de normaliser l'API de management et de proposer une exploitation de cette API grâce à des agents qui utilisent différents protocoles pour dialoguer avec les serveurs de MBeans.

Depuis l'intégration de JMX dans la version 1.5 de Java, l'utilisation de JMX par n'importe quelle application est possible en standard. Les fonctionnalités de base de JMX étant faciles à mettre oeuvre, il devient aisé de surveiller et administrer une application depuis un client JMX distant.

JMX est un framework à usage général pour la surveillance et l'administration de ressources ou d'applications : il ne propose par exemple aucune structure de données spécifique au monitoring ou à la gestion.

L'API JMX est regroupée dans le package javax.management et ses sous-packages :

JSR	Packages
JSR 003	javax.management javax.management.loading javax.management.modelmbean javax.management.monitor javax.management.openmbean javax.management.relation javax.management.timer
JSR 160	javax.management.remote javax.management.remote.rmi
JSR 174	java.lang.management

La page web officielle relative à JMX est à l'url https://www.oracle.com/java/technologies/javase/javamanagement.html

32.2. L'architecture de JMX

L'architecture de JMX se compose de plusieurs niveaux :

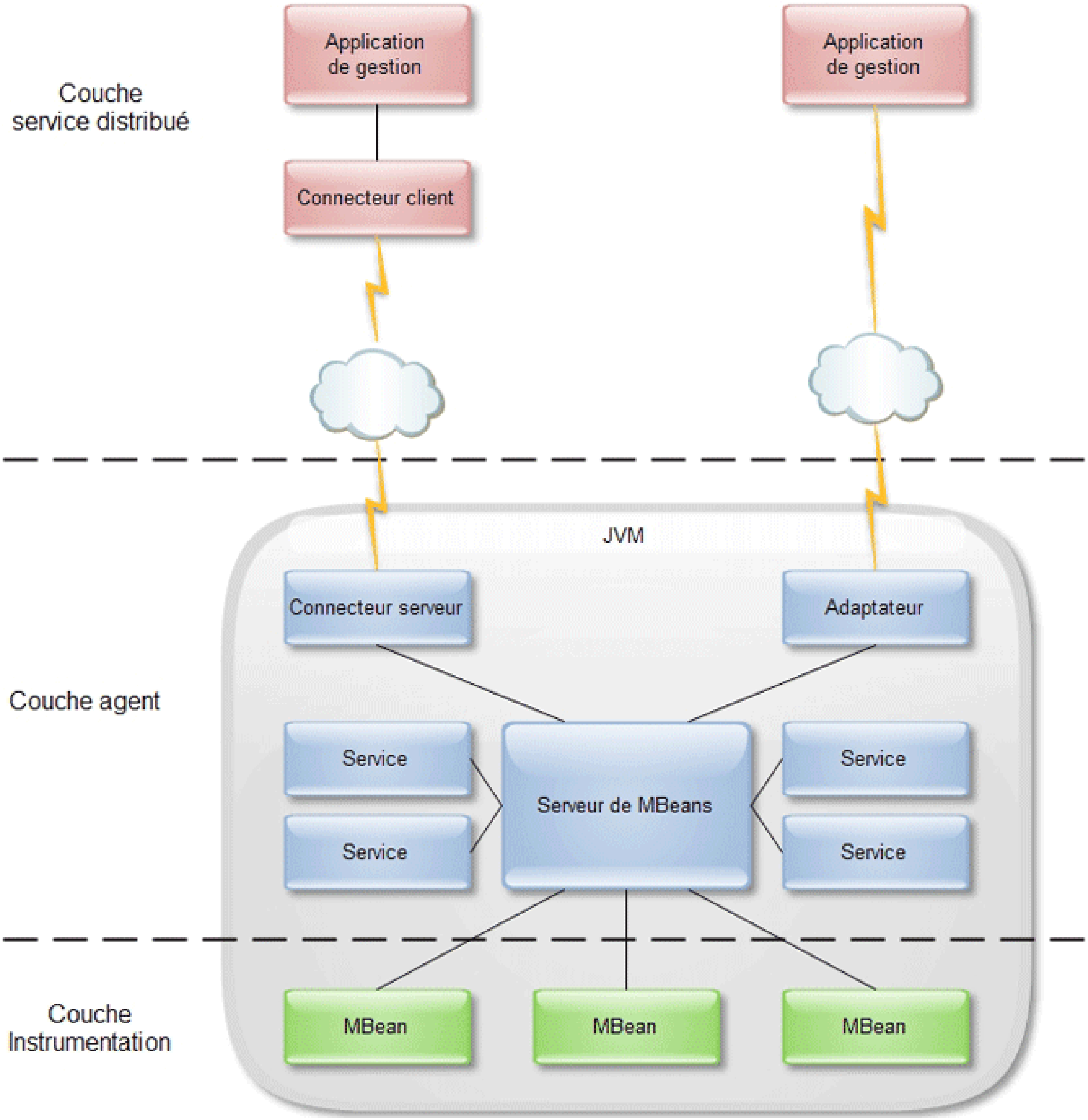
- Services distribués : cette couche définit la partie IHM. C'est généralement une application qui permet de consulter les données relatives à l'application et d'interagir avec elles. Cette couche utilise des connecteurs et des adaptateurs de protocoles pour permettre à des outils de gestion de se connecter à un agent.
- Agent : cette couche définit un serveur de MBeans qui gère les MBeans. Elle propose des fonctionnalités sous la forme d'un agent JMX et assure la communication avec la couche services distribués grâce à des Connectors et des Adapters
- Instrumentation : cette couche définit des MBeans qui permettent l'instrumentation d'une ressource (application, service, composant, objet, appareil, ...) grâce à des attributs, des opérations et des événements. La ressource peut être écrite en Java ou la ressource peut être encapsulée par une classe qui va communiquer avec la ressource (wrapper). Une ressource peut être instrumentée par un ou plusieurs MBeans. Un dynamic MBean implémente une interface particulière qui permet plus de flexibilité à l'exécution. Les MBeans n'ont pas besoin de référence sur l'agent qui va les gérer.
- Ressources gérées (composants de l'application, services, périphériques, ...) : cette couche n'est pas directement concernée par l'API JMX

L'architecture de l'API JMX est composée des trois premières parties. Les couches instrumentation et agent sont spécifiées par la JSR 003.

La couche Remote management ou distributed services est spécifiée par la JSR 160.

La technologie JMX propose une architecture en trois couches pour permettre à des applications de gérer des ressources :

- instrumentation : des ressources (applications, services, composants, appareils, ...) sont instrumentées avec des objets Java de type MBean. Un MBean a pour rôle de fournir une interface qui permet d'obtenir des informations sur la ressource et éventuellement de la gérer au travers de méthodes dédiées.
- agent : le composant principal de ce niveau est un agent qui est un serveur de MBeans dans lequel les MBeans se sont enregistrés. L'agent permet un accès aux MBeans grâce à des connecteurs ou des adapteurs de protocoles
- service distribué : fournit une IHM pour permettre d'interagir sur les ressources grâce à l'agent.



L'élément principal du niveau agent est un objet de type « MBean Server » : son rôle est de gérer et de mettre en oeuvre les MBeans qui se sont enregistrés auprès de lui.

Le découpage de l'architecture de l'API en trois couches permet une meilleure répartition des rôles et réduit la complexité des fonctionnalités des différentes couches.

Chacune des trois couches propose des objets avec des interfaces bien définies.

L'élément principal du niveau instrumentation est un objet de type MBean.

La mise en oeuvre d'un MBean standard implique plusieurs étapes :

- définition des fonctionnalités du MBean dans une interface
- créer le MBean qui doit implémenter cette interface
- instancier le MBean
- enregistrer le MBean dans un serveur de MBeans
- utiliser une application de gestion qui va dialoguer avec le serveur de MBeans au moyen d'un connecteur et interagir avec le MBean

Cette architecture permet de rendre la façon dont une ressource est instrumentée indépendante de l'infrastructure de gestion utilisée. Cette indépendance est assurée par des connecteurs (connectors) qui permettent à une application de gestion de dialoguer avec un agent JMX. Ce connecteur peut utiliser différents protocoles.

Ceci permet d'intégrer de façon standard la surveillance et la gestion de ressources en Java avec des applications de monitoring et de gestion existantes pour peu que ces applications possèdent un connecteur respectant les spécifications JMX ou qu'il existe un adaptateur pour le protocole utilisé.

32.3. Un premier exemple

Ce premier exemple va utiliser Java SE 5.0 pour créer un MBean de type standard, instancier un serveur de MBean, enregistrer le MBean dans le serveur et interagir avec le MBean grâce à l'outil Jconsole du JDK.

32.3.1. La définition de l'interface et des classes du MBean

Il faut définir l'interface du MBean : son nom doit obligatoirement être composé du nom de sa classe suivi de MBean

Exemple :

```
package fr.jmdoudoux.dej.jmx;

public interface PremierMBean {

  public String getNom();

  public int getValeur();
  public void setValeur(int valeur);

  public void rafraichir();
}
```

Il faut définir le MBean qui doit implémenter l'interface définie

Exemple :

```
package fr.jmdoudoux.dej.jmx;

public class Premier implements PremierMBean {

  private static String nom = "PremierMBean";
  private int valeur = 100;

  public String getNom() {
    return nom;
  }

  public int getValeur() {
    return valeur;
  }

  public synchronized void setValeur(int valeur) {
     this.valeur = valeur;
  }

  public void rafraichir() {
    System.out.println("Rafraichir les donnees");
  }

  public Premier() {
  }
}
```

Il faut définir une application qui va créer un serveur de MBeans, instancier le MBean et l'enregistrer dans le serveur.

Exemple :

```
package fr.jmdoudoux.dej.jmx;

import java.lang.management.ManagementFactory;

import javax.management.InstanceAlreadyExistsException;
import javax.management.MBeanRegistrationException;
import javax.management.MBeanServer;
import javax.management.MalformedObjectNameException;
import javax.management.NotCompliantMBeanException;
import javax.management.ObjectName;

public class LancerAgent {

  public static void main(String[] args) {
    MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();

    ObjectName name = null;
    try {
      name = new ObjectName("fr.jmdoudoux.dej.jmx:type=PremierMBean");

      Premier mbean = new Premier();

      mbs.registerMBean(mbean, name);

      System.out.println("Lancement ...");
      while (true) {

        Thread.sleep(1000);
        mbean.setValeur(mbean.getValeur() + 1);
      }
    } catch (MalformedObjectNameException e) {
      e.printStackTrace();
    } catch (NullPointerException e) {
      e.printStackTrace();
    } catch (InstanceAlreadyExistsException e) {
      e.printStackTrace();
    } catch (MBeanRegistrationException e) {
      e.printStackTrace();
    } catch (NotCompliantMBeanException e) {
      e.printStackTrace();
    } catch (InterruptedException e) {
    }
  }
}
```

32.3.2. L'exécution de l'application

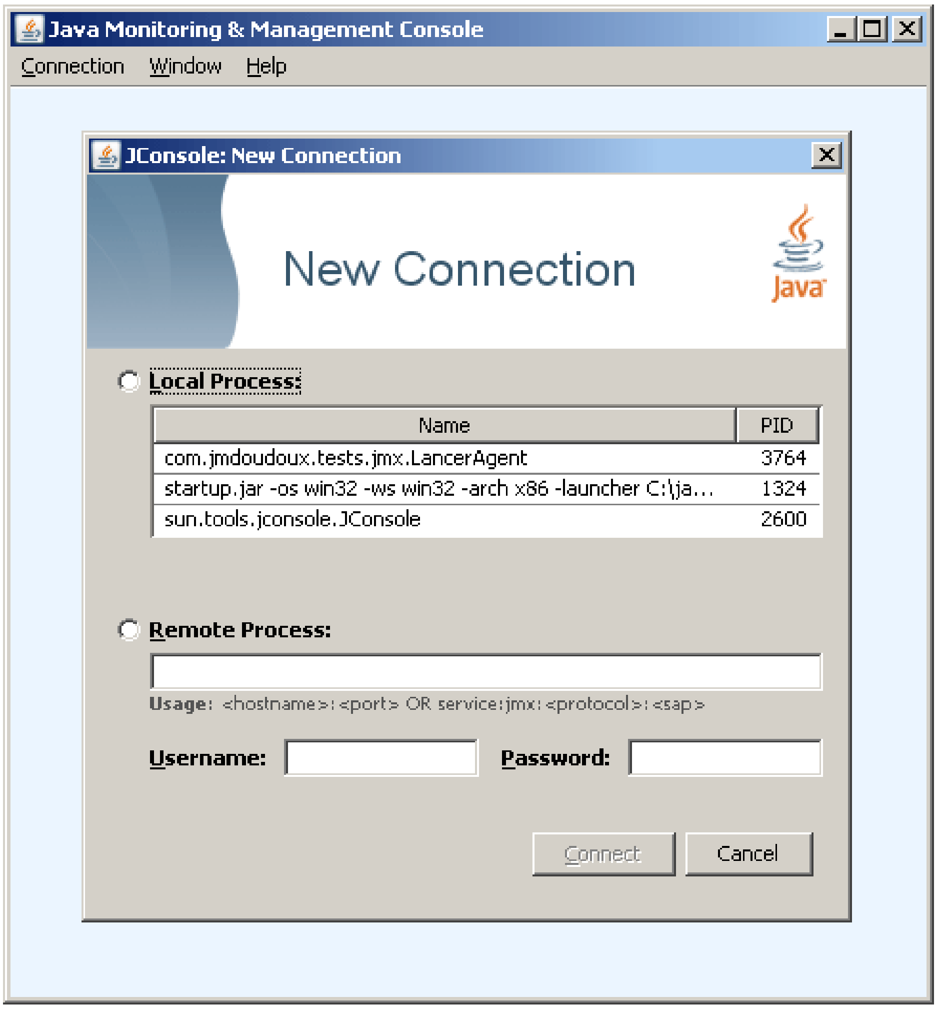
Il faut compiler la classe et l'exécuter en demandant l'activation de l'accès distant aux fonctionnalités de JMX

Exemple :

```
java -Dcom.sun.management.jmxremote fr.jmdoudoux.dej.jmx.LancerAgent
```

Il est alors possible d'accéder à l'agent en utilisant par exemple l'outil JConsole fourni avec le JDK à partir de sa version 5.0 ou Visual VM fourni avec le JDK à partir de sa version 6.0

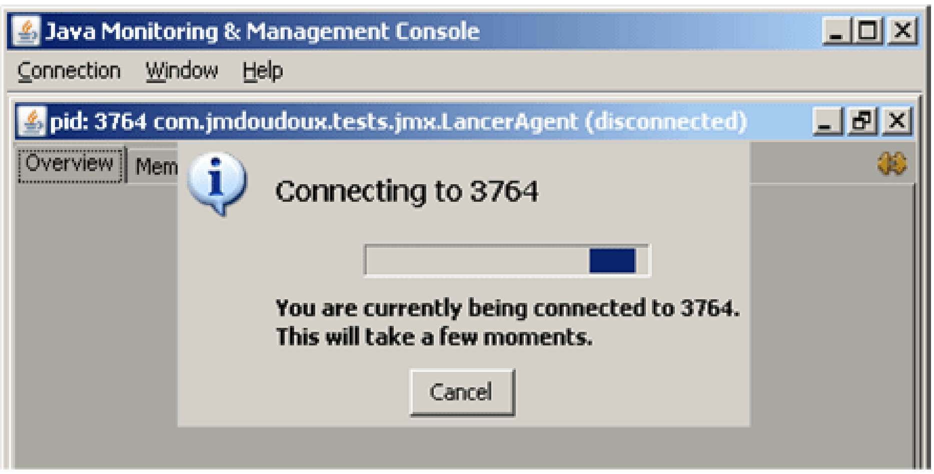
Sous Windows, il faut ouvrir une nouvelle boîte de commandes et lancer la commande jconsole du JDK.



Name	PID
com.jmdoudoux.tests.jmx.LancerAgent	3764
startup.jar -os win32 -ws win32 -arch x86 -launcher C:\ja...	1324
sun.tools.jconsole.JConsole	2600

Usage: <hostname>:<port> OR service:jmx:<protocol>:<sap>

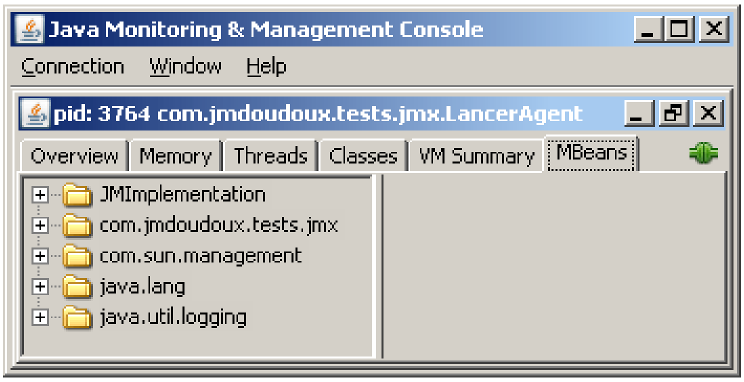
Il faut sélectionner « Local Process » puis la JVM dans laquelle le serveur MBean est en cours d'exécution et cliquez sur le bouton « Connect »


Java Monitoring & Management Console
Connection Window Help
pid: 3764 com.jmdoudoux.tests.jmx.LancerAgent (disconnected)
Overview
Mem
Connecting to 3764

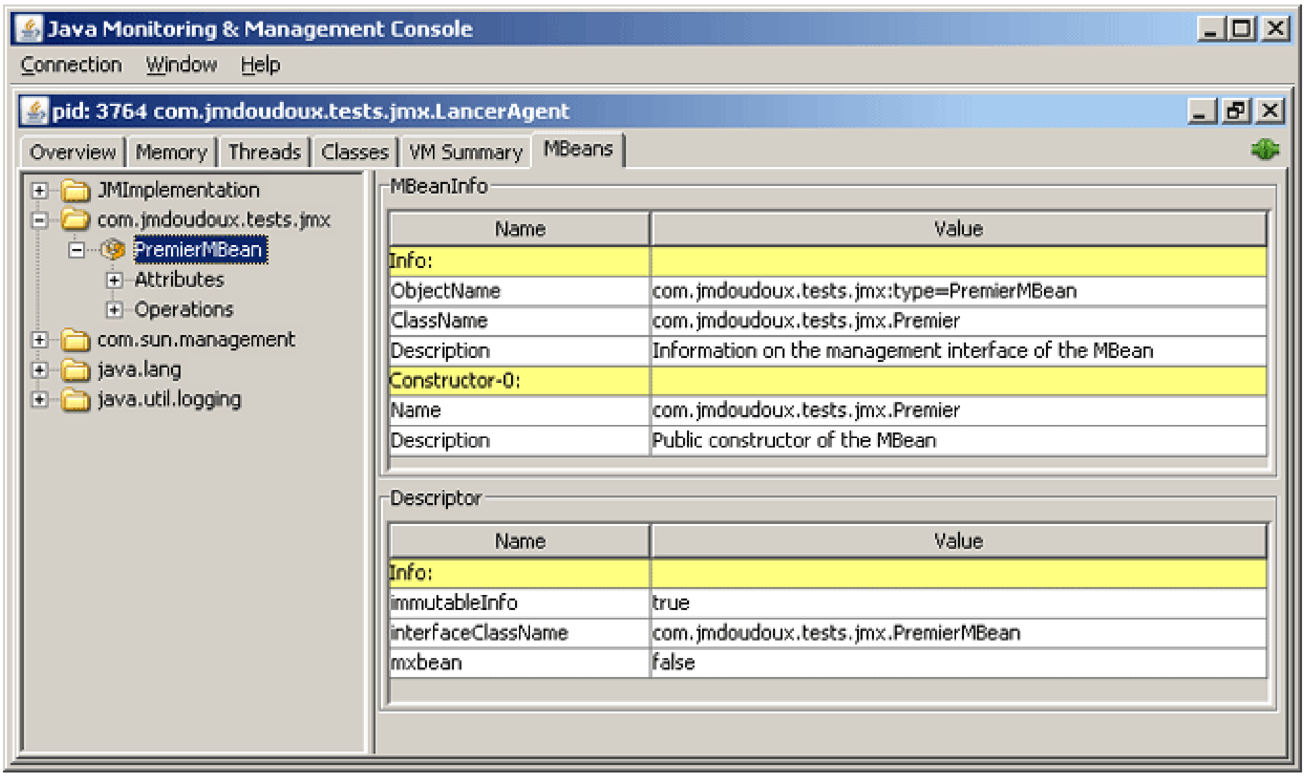
You are currently being connected to 3764.
This will take a few moments.

JConsole tente de se connecter au processus de la JVM.

JConsole permet d'obtenir des informations sur la JVM et de gérer les MBeans.



Il faut sélectionner dans l'arborescence le MBean concerné.



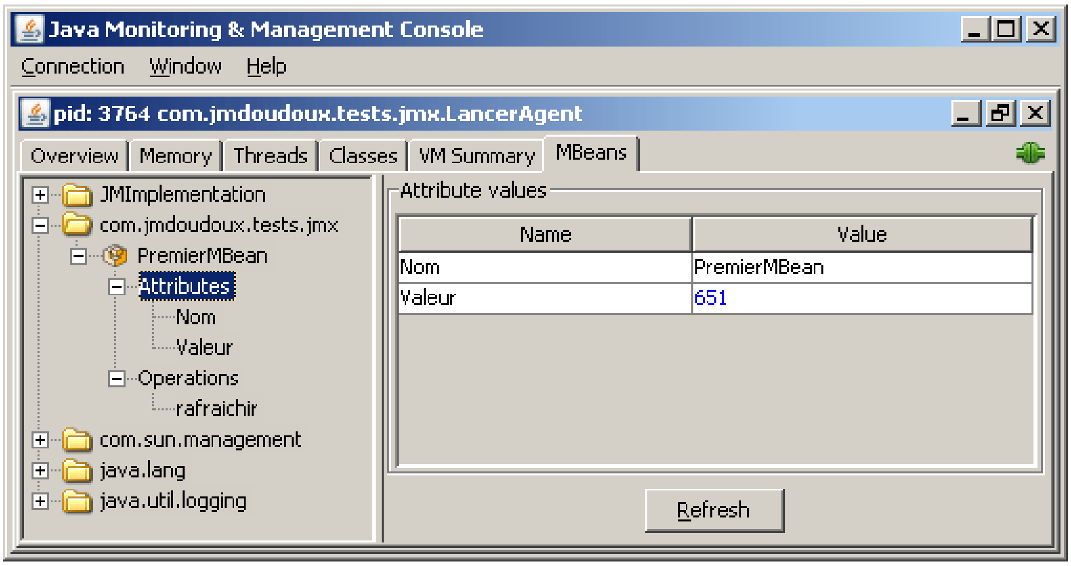
MBeanInfo

Name	Value
Info:	
ObjectName	com.jmdoudoux.tests.jmx:type=PremierMBean
ClassName	com.jmdoudoux.tests.jmx.Premier
Description	Information on the management interface of the MBean
Constructor-0:	
Name	com.jmdoudoux.tests.jmx.Premier
Description	Public constructor of the MBean

Descriptor

Name	Value
Info:	
immutableInfo	true
interfaceClassName	com.jmdoudoux.tests.jmx.PremierMBean
mxbean	false

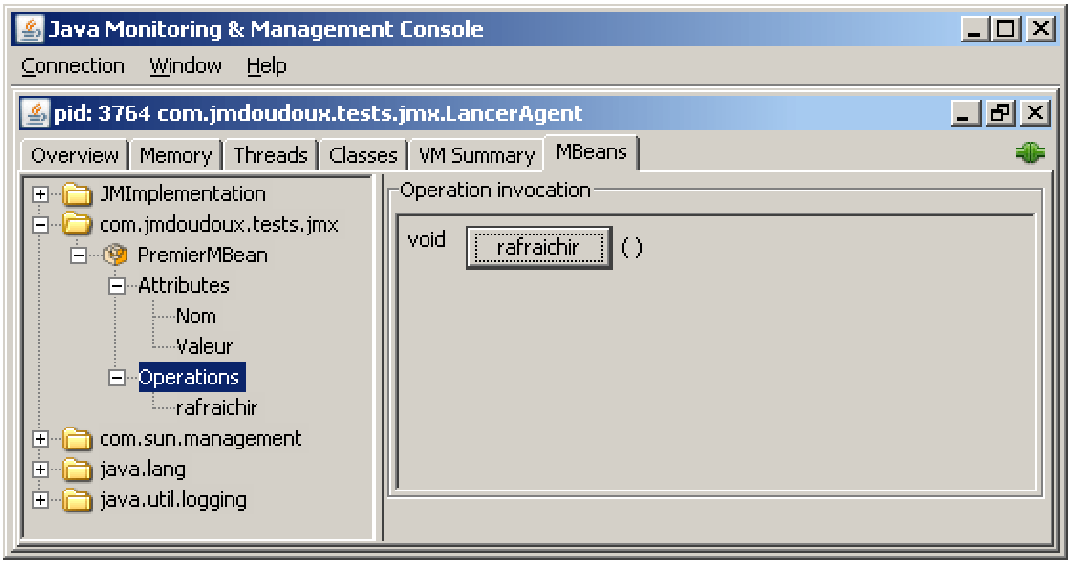
La branche Attributes permet de voir les différents attributs.



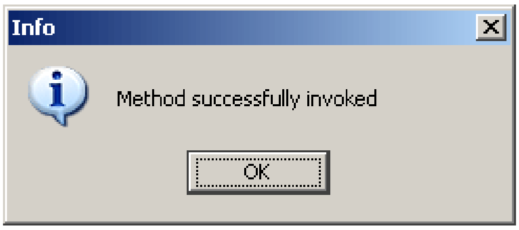
Attribute values

Name	Value
Nom	PremierMBean
Valeur	651

La branche « Operations » permet d'invoquer les méthodes du MBean.



Il faut sélectionner la méthode et cliquer sur le bouton.



Method successfully invoked

32.4. La couche instrumentation : les MBeans

La couche instrumentation assure l'instrumentation de ressources grâce aux MBeans.

L'instrumentation consiste à écrire des MBeans qui vont permettre d'instrumenter des ressources.

Pour être instrumentée, une ressource doit être une classe Java ou être encapsulée dans une classe Java. C'est notamment le cas si la ressource est un appareil. L'instrumentation se fait au moyen de la définition d'une interface qui décrit les fonctionnalités d'instrumentation proposées et une classe de type MBean qui implémente cette interface.

Rien n'est imposé quant à la granularité de la ressource à instrumenter : celle-ci peut aller d'une simple classe jusqu'à une application dans son intégralité.

L'instrumentation d'une ressource se fait grâce à un Managed Bean ou MBean. Il existe plusieurs types de MBean.

- MBean standard : c'est une classe Java qui implémente une interface dédiée et respecte les spécifications de JMX.
- MBean dynamique : classe Java encapsulant un MBean qui offre plus de possibilités au runtime pour exposer dynamiquement ses fonctionnalités.

32.4.1. Les MBeans

Un MBean est l'élément de la spécification JMX le plus bas : son rôle est d'assurer la communication avec la ressource à gérer. MBean est l'abréviation de Managed Bean.

Un MBean a pour rôle de permettre la gestion et le dialogue avec une ressource. La nature d'une telle ressource peut être variée : application, composant, service, dispositif, appareil électronique, etc ...

Pour gérer des ressources grâce à JMX, il faut tout d'abord instrumenter la ressource en développant une classe qui sera le MBean.

Un MBean est une classe Java respectant les spécifications JMX qui implémente une interface particulière. Un MBean possède les caractéristiques suivantes :

- doit être une classe public concrète
- doit avoir au moins un constructeur public
- doit implémenter sa propre interface, dont le nom est celui de la classe du MBean suffixé par MBean, ou implémenter l'interface DynamicMBean

Un MBean est accessible par son interface qui peut proposer :

- l'appel des constructeurs du MBean
- l'obtention et/ou la modification de la valeur de propriétés (attributs en lecture et/ou écriture)
- l'invocation des méthodes
- l'émission de notifications lorsque certains événements surviennent
- une description pour les fonctionnalités proposées

Remarque : il n'est pas recommandé de surcharger des méthodes exposées par un MBean.

Un MBean est plus qu'une interface puisqu'il doit contenir le code permettant les interactions avec la classe ou l'appareil qu'il doit instrumenter et/ou surveiller.

En plus de l'instrumentation, un MBean peut émettre des notifications en réponse à des événements. Le modèle de notifications proposé par JMX pour les MBeans repose sur le modèle des événements Java. Ces notifications permettent aux MBeans et à l'agent qui les gère de notifier certains événements à un ou plusieurs abonnés.

32.4.2. Les différents types de MBeans

Les MBeans sont répartis en deux grandes familles : standard MBeans et dynamic MBeans.

Il existe quatre types de MBeans plus ou moins complexes à développer :

- Standard MBean : ce sont des Java beans qui implémentent une interface définie de façon statique par le développeur : leurs fonctionnalités sont décrites dans cette interface implémentée par le MBean. Dans un tel objet, le nombre de propriétés exposées par le MBean est fixe puisque défini par l'interface. C'est le type le plus utilisé car c'est le plus simple à implémenter.
- Dynamic MBean : ils exposent les informations concernant leurs fonctionnalités au travers de métadonnées. Ils implémentent l'interface DynamicMBean qui leur permet d'exposer leurs fonctionnalités dynamiquement à l'exécution : le nombre de propriétés exposées peut donc être variable. Chaque attribut, opération et notification doit être découvert à l'exécution. Leur écriture est relativement complexe.
- Open MBean : ce sont des Dynamic MBeans qui respectent des conventions ce qui les rend un peu plus complexes mais ils sont en contrepartie plus portables. Ces conventions imposent notamment de n'utiliser que des types de base de Java et certaines classes définies dans les spécifications JMX. Il est alors inutile de rajouter des classes au classpath de l'agent et des clients.
- Model MBean : ce sont des MBeans dynamic génériques qui peuvent être entièrement configurés. Ils sont fournis par l'implémentation de JMX utilisée et leur mise en oeuvre dépend de cette implémentation.

Les MBeans standard et dynamic publient tous les deux dynamiquement leur interface :

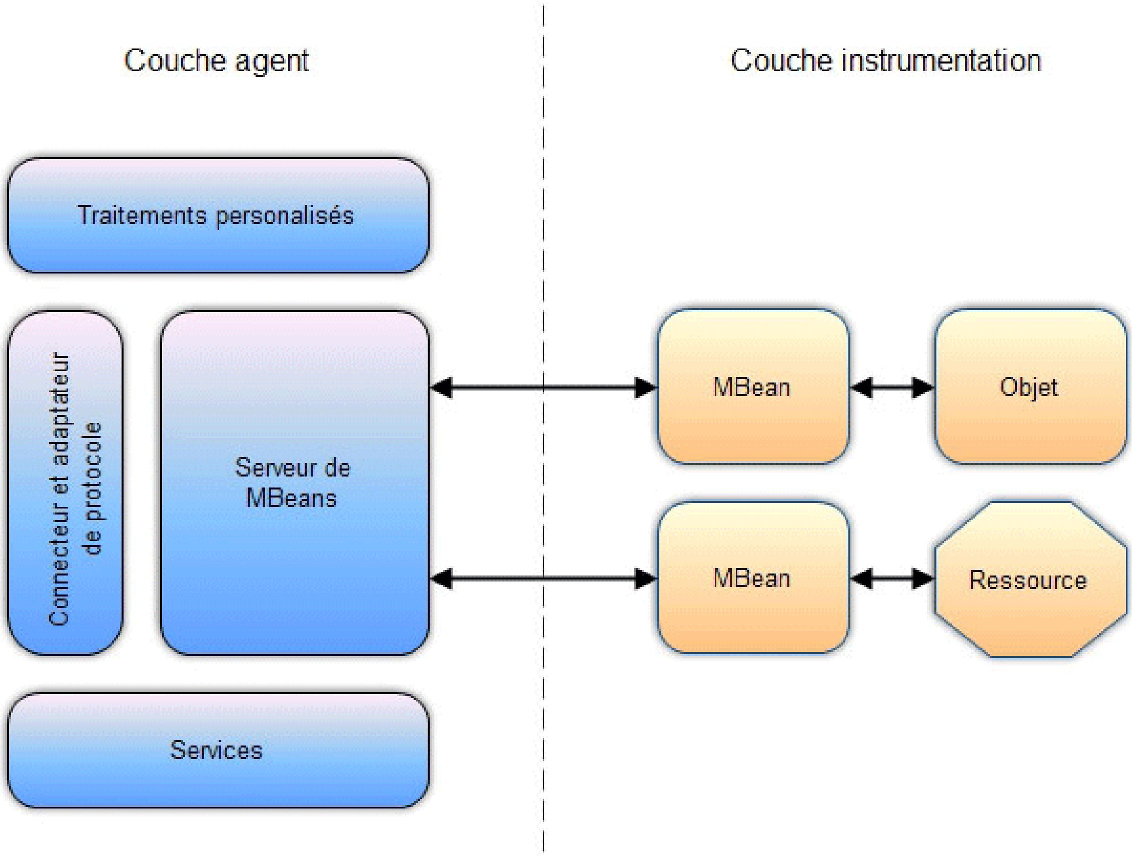
- avec les Dynamic MBeans, le développeur doit coder les méthodes pour publier son interface
- avec les Standard MBeans, JMX utilise l'introspection pour publier l'interface du MBean

Les Dynamic MBeans publient les méthodes de l'interface du MBean en fournissant une description de chacune des méthodes exposées aux clients JMX.

Les Dynamic MBeans utilisent des classes qui encapsulent les métadonnées d'un MBean. Ces métadonnées permettent de décrire la structure et les fonctionnalités du MBean (constructeurs, attributs, opérations et notifications). Elles comprennent un nom, une description et des caractéristiques.

32.4.3. Les MBeans dans l'architecture JMX

Les composants MBeans sont gérés par un serveur de MBeans.



Chaque MBean enregistré dans le serveur de MBeans d'un agent JMX expose son interface aux applications de gestion. Un MBean est accédé de l'extérieure grâce à l'agent dans lequel il est enregistré.

Lorsque l'on crée un MBean, il n'est pas utile de se soucier du type d'agent ou d'application de gestion qui va solliciter le MBean : un MBean n'a pas besoin d'avoir de référence sur le serveur qui gère son cycle de vie.

32.4.4. Le nom des MBeans

Chaque MBean possède un ObjectName qui doit être choisi judicieusement car il permet de l'identifier. Un ObjectName est un objet de type javax.management.ObjectName.

Un ObjectName encapsule le nom d'un MBean ou un motif de recherche de noms de MBean.

Un ObjectName est composé de deux parties :

- un nom de domaine qui peut être le domaine par défaut ou un domaine personnalisé
- un ensemble de propriétés sous la forme de paires clé = valeur séparées par des virgules. Au moins une propriété doit être définie

La syntaxe d'un ObjectName est de la forme [nomDomain]:propriete=valeur[,propriete=valeur]*

Le contenu d'un ObjectName devrait permettre de facilement déterminer le rôle du MBean.

Exemple :

fr.jmdoudoux.dej.jmx:type=MaRessourceBean,name=maRessource

Le domaine est une chaîne de caractères arbitraire facultative. Si le domaine est vide alors cela implique l'utilisation du domaine par défaut du serveur de MBeans dans lequel l'ObjectName est utilisé.

Il est recommandé de préfixer le nom de domaine par le nom du package Java pour éviter les collisions de noms entre MBeans.

Le nom de domaine ne doit pas contenir de caractères "/" qui est réservé pour les hiérarchies de serveurs de MBean. Le domaine ne peut pas contenir de caractère ":" puisqu'il est utilisé comme séparateur entre le domaine et les propriétés.

Si le domaine contient au moins un caractère "*" ou "?" alors l'ObjectName est un motif pour la recherche de MBeans. Ces deux caractères ne peuvent pas être utilisés dans le nom d'un MBean.

La liste de propriétés doit obligatoirement contenir au moins une propriété sous la forme clé=valeur. Chaque propriété de la liste est séparée par un caractère virgule. Le nom de la propriété doit respecter les conventions de nommage des entités Java.

Chaque ObjectName d'un même type devrait avoir le même ensemble de propriétés. Les valeurs de ces propriétés vont permettre de différencier plusieurs instances d'un même type. Parmi ces propriétés, il est fréquent de trouver la propriété name.

Hormis la clé type, toutes les autres clés peuvent être librement utilisées. Les clés sont généralement utilisées par les clients JMX pour afficher une représentation graphique hiérarchique du MBean.

La JSR 77 définit une propriété j2eeType qui possède un rôle similaire. Les propriétés type et j2eeType peuvent être utilisées simultanément.

Attention : les espaces contenus dans l'ObjectName sont tous significatifs et les ObjectNames sont sensibles à la casse.

L'ordre des propriétés n'est pas significatif.

La valeur d'une propriété peut être entourée de double quotes surtout si elle contient des caractères spéciaux comme le caractère virgule par exemple.

Exemple :

fr.jmdoudoux.dej.jmx:type=MaRessourceBean,name=maRessource, taille="200"

fr.jmdoudoux.dej.jmx:type=MaRessourceBean,name=maRessource, taille="200,250"

Le caractère * peut aussi être utilisé dans la liste de propriétés sur l'ObjectName : dans ce cas il concerne un motif de recherche.

Exemple :

fr.jmdoudoux.dej.jmx:type=MaRessourceBean,*

32.4.5. Les types de données dans les MBeans

Les MBeans peuvent être amenés à manipuler des types de données plus ou moins complexes pour différents besoins :

- pour typer un attribut qui est alors utilisé dans les getter et setter
- pour des paramètres des méthodes et leur valeur de retour
- pour lever des exceptions dans les méthodes
- pour les notifications et les données qu'elles utilisent

32.4.5.1. Les types de données complexes

Il est fréquent d'avoir besoin des types de données complexes qui encapsulent les informations d'une entité.

Il est parfois possible si ces données sont des attributs du MBeans de les séparer en différents attributs de types communs. Cependant ceci n'est pas possible pour différentes autres situations :

- le type complexe est lui-même composé de types complexes
- le type complexe comporte de nombreux attributs et doit être utilisé en paramètre d'une méthode
- le type complexe doit être utilisé comme valeur de retour d'une méthode

L'utilisation de ces types complexes va inévitablement poser des problèmes avec certains clients génériques comme l'outil JConsole ou l'adaptateur de protocole HTML car naturellement ces types de données ne sont pas connus.

JMX propose également au travers des spécifications des Open MBeans une solution pour utiliser des types complexes de façon portable.

32.5. Les MBeans standard

Ce sont les plus simples des MBeans. Il suffit de définir une interface dont le nom est composé du nom de la classe du MBean et du suffixe MBean.

Les fonctionnalités de base de JMX sont faciles à mettre en oeuvre dans un MBean standard puisqu'il suffit d'écrire une interface qui décrit les fonctionnalités du MBean et de fournir une implémentation dans une classe qui respecte les conventions Java Bean.

Dans un MBean standard, l'interface du MBean ne peut pas être modifiée à l'exécution : JMX propose d'autres types de MBeans pour répondre à ce besoin.

32.5.1. La définition de l'interface d'un MBean standard

Un MBean peut contenir des accesseurs (getters et/ou setters) pour des attributs et des méthodes qui pourront être invoquées pour réaliser des actions.

Ainsi un MBean peut proposer :

- des attributs en lecture et/ou écriture
- des opérations qui pourront être invoquées
- des notifications qui pourront être émises par le MBean et envoyées à des abonnés

Dans un MBean standard, ces différents éléments sont définis de façon statique dans une interface. Cette interface peut donc contenir :

- la définition des getters/setters sur les attributs du MBean
- la définition des opérations proposées par le MBean

Exemple :

```
package fr.jmdoudoux.dej.jmx;

public interface PremierMBean {

  public String getNom();

  public int getValeur();
  public void setValeur(int valeur);

  public void rafraichir();
}
```

Dans un MBean standard, tous les constructeurs publics sont exposés automatiquement même le constructeur par défaut si celui-ci est créé par le compilateur. Ainsi un client JMX peut instancier un MBean en utilisant un des constructeurs publics.

Les méthodes d'un MBean peuvent être des getters et setters sur des attributs ou des opérations qui permettront de réaliser certains traitements. Les méthodes de types getter et setter doivent respecter les conventions de nommage des Java beans. Il n'est pas possible de surcharger un getter ou un setter.

Si les conventions de nommage des Java beans ne sont pas respectées ou si une incohérence est détectée entre le type utilisé pour le getter et le setter, une exception sera levée lors de l'utilisation du MBean.

Exemple :

```
package fr.jmdoudoux.dej.jmx;

import java.io.IOException;

public interface SecondMBean {

  public int getValeur() throws IOException;
  public void setValeur(String valeur) throws IOException;

}
```

Lors d'une tentative d'enregistrement d'un MBean implémentant cette interface, une exception de type NotCompliantException est levée.

Exemple :

```
javax.management.NotCompliantMBeanException: Getter and setter for Valeur have inconsistent
types
```

Chaque méthode publique qui n'est pas identifiée comme étant un getter ou un setter d'une propriété est considérée comme une opération exposée par le MBean. Une opération peut avoir un nombre quelconque de paramètres et éventuellement avoir une valeur de retour.

Les spécifications JMX préconisent de ne pas surcharger les méthodes exposées des MBeans.

Une méthode d'un MBean peut lever des exceptions qui devront être gérées par le client JMX. Il est recommandé pour un MBean de ne lever que des exceptions fournies en standard par la plate-forme Java SE. Si un MBean lève une exception non standard, l'application qui utiliserait ce MBean lèvera une exception de type ClassNotFoundException.

C'est une best practice que chaque méthode déclarée dans l'interface d'un bean standard indique qu'elle peut lever l'exception java.io.IOException. Ceci impose la prise en compte de cette exception notamment lors de l'utilisation d'un proxy sinon une erreur de communication lèvera une exception de type UndeclaredThrowableException qui encapsulera l'exception de type IOException.

Exemple :

```
package fr.jmdoudoux.dej.jmx;

import java.io.IOException;

public interface PremierMBean {

  public String getNom() throws IOException;

  public int getValeur() throws IOException;
  public void setValeur(int valeur) throws IOException;

  public void rafraichir() throws IOException;
}
```

Si le MBean n'est accédé que dans la JVM dans laquelle il s'exécute, il n'est pas utile de déclarer qu'une méthode peut lever une exception de type IOException. Mais généralement, l'accès aux MBeans se fait de façon distante. Pour simplifier l'utilisation locale, il est possible de définir une interface fille qui redéfinit les méthodes mais sans déclarer la levée de l'exception.

Exemple :

```
package fr.jmdoudoux.dej.jmx;

public interface PremierLocalMBean {

  public String getNom();

  public int getValeur();
  public void setValeur(int valeur);

  public void rafraichir();
}
```

32.5.2. L'implémentation du MBean Standard

La classe du MBean doit implémenter l'interface du MBean. Le nom de la classe doit obligatoirement être identique à celui de l'interface sans le suffixe MBean. Si ce n'est pas le cas, une exception de type javax.management.NotCompliantMBeanException est levée.

Seules les propriétés et opérations définies dans l'interface seront utilisables par l'agent JMX. L'agent JMX va utiliser l'introspection pour déterminer la liste des propriétés et des opérations supportées par le MBean en recherchant l'interface du MBean.

Pour transformer une classe Xyz en MBean standard, il faut :

- créer une interface XyzMBean qui contient les méthodes et les getter et setter des attributs à exposer
- que la classe Xyz implémente l'interface XyzMBean

Afin d'assurer une séparation des rôles, il est préférable de placer dans deux classes distinctes la ressource gérée ou l'encapsulation de cette ressource et la classe du MBean qui va gérer la ressource. Il suffit pour cela que le MBean possède une référence sur la classe de la ressource, généralement passée dans le constructeur du MBean.

32.5.3. L'utilisation d'un MBean

Chaque MBean doit être enregistré dans un serveur de MBeans avec un identifiant unique sous la forme d'un nom d'objet (ObjectName).

Un ObjectName est composé d'un nom de domaine et d'attributs sous la forme de paires clé/valeur. La combinaison du nom de domaine et des attributs doit obligatoirement être unique dans un serveur de MBeans.

Les méthodes déclarées dans l'interface du MBean seront accessibles aux clients JMX.

Les constructeurs publics sont toujours accessibles aux clients JMX par introspection.

Les propriétés accessibles aux clients JMX sont exposées grâce aux getters et setters définis dans l'interface du MBean.

Il n'y a rien qui puisse empêcher l'utilisation des MBeans directement sans passer par JMX puisque ce sont de simples Java beans.

32.6. La couche agent

Si une ressource est instrumentée par un MBean, alors la gestion du MBean est assurée par un agent JMX : il assure donc la gestion et l'exploitation de différents MBeans.

Le niveau agent est essentiellement composé d'un agent JMX qui possède plusieurs responsabilités :

- instancier et gérer les MBeans dans un serveur de MBeans
- charger et initialiser le ou les connecteurs et adaptateurs de protocoles pour dialoguer avec des clients
- fournir des services définis dans les spécifications JMX

Le composant principal d'un agent JMX est un serveur de MBeans. Un serveur de MBeans enregistre et gère des MBeans. Généralement un agent JMX s'exécute dans la JVM où s'exécutent les MBeans qu'il gère mais ce n'est pas une obligation. L'agent permet à une application d'interagir avec les MBeans par son intermédiaire en utilisant des connecteurs ou des adaptateurs de protocoles.

Un serveur de MBeans est un registre pour MBeans : il gère le cycle de vie des MBeans qui s'enregistrent auprès de lui. Le serveur de MBeans donne à des applications tierces un accès aux MBeans en exposant leurs interfaces.

Les MBeans peuvent être instanciés et enregistrés dans le serveur de MBeans par un autre MBean ou par l'agent. Chaque MBean s'enregistre avec un identifiant unique de type ObjectName. Cet identifiant est fourni par le développeur, c'est donc lui qui doit être le garant de son unicité dans un même serveur de MBeans.

Un agent JMX permet donc à une application de gestion d'invoquer les fonctionnalités des MBeans : il assure la communication entre les MBeans et les interfaces de gestions grâce à plusieurs entités : un serveur de MBeans et un ou plusieurs adaptateurs de protocoles ou connecteurs.

Par défaut un agent ne possède pas de possibilités de communications. Les connecteurs et adaptateurs de protocoles permettent à un agent de communiquer en utilisant un protocole tel que HTTP, SNMP, ...

L'implémentation par défaut propose un adaptateur de protocole pour HTML, qui permet de disposer, pour n'importe quel agent JMX, d'une console d'administration accessible depuis un navigateur internet. C'est sur ce principe que fonctionne la console JMX fournie en standard avec JBoss par exemple.

Un agent JMX peut se voir ajouter des fonctionnalités dynamiquement sous la forme de services. Plusieurs services sont fournis en standard et il est possible de développer ses propres services. Généralement, ils sont fournis sous la forme de MBeans et sont donc administrables par JMX et le serveur de MBeans qui les gère.

32.6.1. Le rôle d'un agent JMX

Un agent JMX sert d'intermédiaire entre les MBeans et un client JMX (généralement une application de gestion) : il assure l'indépendance entre la ressource gérée et l'application de gestion distante.

Pour gérer le MBean et permettre son accès, il faut l'enregistrer dans un agent JMX. C'est le serveur de MBeans qui est le composant principal de l'agent et assure la gestion du cycle de vie des MBeans qui se sont enregistrés auprès de lui.

L'agent JMX va utiliser l'introspection sur l'interface pour déterminer les fonctionnalités offertes par un MBean standard.

La communication entre ce serveur et les applications clientes se fait grâce à des adaptateurs de protocoles ou des connecteurs qui assurent la communication de façon indépendante du MBean.

L'agent JMX propose aussi plusieurs services offrant différentes fonctionnalités.

Un agent JMX s'exécute généralement dans la JVM où sont exécutés les MBeans mais ce n'est pas une obligation.

32.6.2. Le serveur de MBeans (MBean Server)

Le MBean Server compose le coeur de l'agent : il gère les MBeans qui se sont enregistrés auprès de lui grâce à un identifiant unique (Object Name). Le serveur de MBeans est alors en charge de la gestion de ces MBeans.

Ce serveur permet de gérer le cycle de vie des MBeans (ajout, modification ou suppression) et de permettre leur utilisation de manière locale ou distante. C'est le coeur du système de gestion proposé par JMX.

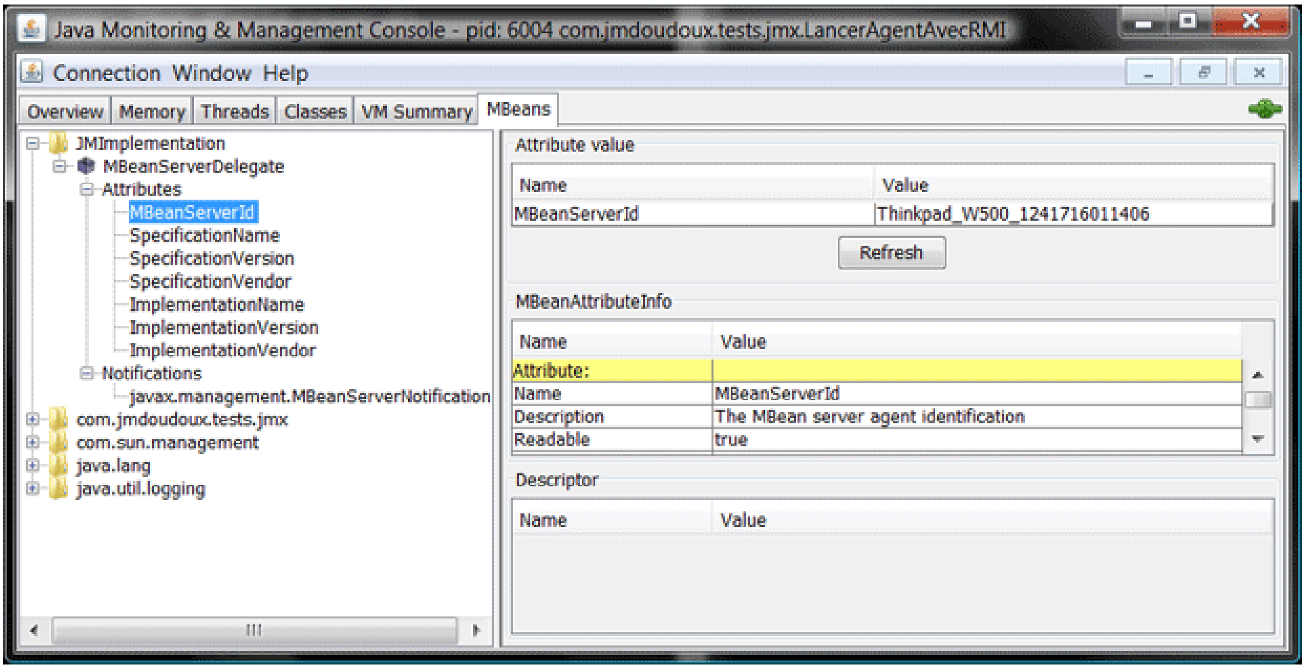
Un serveur de MBeans possède plusieurs fonctionnalités notamment :

- agir comme un registre pour les MBeans qui se sont enregistrés auprès de lui
- gérer le cycle de vie des MBeans
- découvrir les fonctionnalités des interfaces des MBeans et les exposer aux applications de gestion
- lire et modifier les valeurs des propriétés d'un MBean
- invoquer les méthodes du MBean
- obtenir les notifications émises par le MBean

32.6.3. Le Mbean de type MBeanServerDelegate

Lorsqu'un serveur de Mbeans est instancié, un MBean de type MBeanServerDelegate est automatiquement instancié et enregistré dans le serveur avec un ObjectName qui vaut « JMImplementation:type=MBeanServerDelegate ».

Cet MBean permet d'obtenir des informations sur le serveur MBean sous la forme de plusieurs attributs en lecture seule : MBeanServerId, SpecificationName, SpecificationVersion, SpecificationVendor, ImplementationName, ImplementationVersion et ImplementationVendor.



Name	Value
MBeanServerId	Thinkpad_W500_1241716011406

Name	Value
Attribute:	
Name	MBeanServerId
Description	The MBean server agent identification
Readable	true

Name	Value

Le plus utile de ces attributs est MBeanServerId puisqu'il fournit l'identifiant du serveur de MBeans.

C'est aussi lui qui est responsable des notifications de type jmx.mbean.created et jmx.mbean.deleted émises par le serveur de MBeans notamment lors de l'enregistrement ou la suppression de MBeans du serveur.

32.6.3.1. L'enregistrement d'un MBean dans le serveur de MBeans

Chaque MBean doit être associé à une instance d'un objet de type ObjectName lors de l'enregistrement dans le serveur de MBeans. Un objet de type ObjectName sert d'identifiant unique et doit respecter les spécifications JMX :

- il doit avoir un domaine (généralement le package qui contient la classe du MBean) ou à défaut le domaine par défaut du serveur
- un ensemble de propriétés (exemple le type d'objet)

Pour utiliser un MBean, il faut donc l'enregistrer dans le serveur de MBeans d'un agent JMX. Le plus simple pour réaliser cette opération est de suivre deux étapes :

- obtenir le serveur de MBeans de la JVM : le plus facile est d'utiliser la méthode getPlatformMBeanServer() de la classe ManagementFactory qui permet d'obtenir un serveur de MBeans en cours d'exécution ou une nouvelle instance d'un serveur de MBeans si aucun n'est déjà en cours d'exécution dans la JVM.
- enregistrer le MBean auprès de l'instance du serveur obtenue : le MBean est enregistré dans le serveur de MBeans en utilisant la méthode registerMBean() qui attend en paramètre une instance du MBean et l'objet de type ObjectName associé au MBean.

32.6.3.2. L'interface MBeanRegistration

L'interface javax.management.MBeanRegistration peut être implémentée par un MBean pour définir des callbacks qui seront invoqués par le serveur de MBeans durant le cycle de vie du MBean. Ces callbacks fournissent un mécanisme de contrôle sur le processus d'enregistrement et de désenregistrement du MBean.

Elle définit quatre méthodes :

Méthode	Rôle
ObjectName preRegister(MBeanServer mbs, ObjectName name)	Invoquée juste avant que le MBean ne soit enregistré dans le serveur de MBeans
void postRegister()	Invoquée juste après que le MBean soit correctement enregistré dans le serveur de MBeans
void preDeRegister()	Invoquée juste avant que le MBean ne soit désenregistré du serveur de MBeans
void postDeRegister()	Invoquée juste après que le MBean soit correctement désenregistré du serveur de MBeans

32.6.3.3. La suppression d'un MBean du serveur de MBeans

La méthode unregisterMBean() permet de supprimer un MBean du serveur de MBeans : elle attend en paramètre l'identifiant du Mbean sous la forme de son ObjectName

Une fois cette méthode invoquée, le serveur ne possède plus de référence sur l'instance du MBean.

Remarque : la destruction d'un serveur de MBeans entraîne la destruction des MBeans qu'il contenait.

32.6.4. La communication avec la couche agent

Les agents ne communiquent pas directement avec la couche services distribués : cette communication est assurée par des connecteurs et/ou des adaptateurs de protocoles. Ceci permet d'utiliser plusieurs protocoles comme HTTP ou SNMP.

Les adaptateurs de protocoles permettent d'accéder à un agent JMX en utilisant un protocole donné : ils adaptent les échanges avec l'agent en utilisant le protocole pour lequel ils ont été écrits.

Généralement les adaptateurs de protocoles ont uniquement une partie serveur qui se charge d'adapter les échanges au format du protocole utilisé.

Un connecteur implique obligatoirement une partie côté client et une partie sur l'agent : il permet un échange entre un client JMX et un agent JMX en utilisant un protocole particulier. Ces échanges sont assurés par une partie du connecteur côté client et une autre côté serveur.

L'implémentation de référence propose un adaptateur de protocole pour HTML qui permet d'avoir un accès à un agent JMX avec un simple navigateur (attention : ce connecteur n'est pas fournie avec le JDK).

La mise en oeuvre des connecteurs et des adaptateurs de protocoles est détaillée dans une des sections suivantes.

32.6.5. Le développement d'un agent JMX

Le développement d'un agent comporte plusieurs étapes :

- Instancier un serveur de MBeans
- Démarrer le serveur
- Instancier et enregistrer le ou les MBeans dans le serveur
- Instancier un connecteur ou un adaptateur de protocole
- Démarrer le connecteur ou l'adaptateur de protocole
- Eventuellement instancier et enregistrer les services de l'agent

Remarque : généralement les services, les connecteurs et les adaptateurs de protocoles sont implémentés sous la forme de MBeans qu'il est possible d'enregistrer dans le serveur de MBeans pour permettre leur administration.

32.6.5.1. L'instanciation d'un serveur de MBeans

Pour instancier un serveur de MBeans, il faut utiliser directement ou indirectement une fabrique de type MBeanServerFactory. Cette fabrique ne possède aucune instance car toutes les méthodes qu'elle propose sont statiques.

La fabrique peut conserver en interne des références sur les instances de type MBeanServer qu'elle crée.

Elle propose plusieurs méthodes pour obtenir ou manipuler une instance de type MBeanServer notamment :

Méthode	Rôle
MBeanServer createMBeanServer()	Renvoyer une nouvelle instance d'un objet qui implémente l'interface MBeanServer associé au nom de domaine par défaut (DefaultDomain).
MBeanServer createMBeanServer(String domain)	Renvoyer une nouvelle instance d'un objet qui implémente l'interface MBeanServer associé au nom de domaine fourni en paramètre.
ArrayList<MBeanServer> findMBeanServer(String agentId)	Retourner une collection des instances de MBeanServer créées avec la méthode createMBeanServer() et toujours présentes dans la fabrique. Toutes les instances sont retournées avec null en paramètres.
MBeanServer newMBeanServer()	Renvoyer une nouvelle instance d'un objet qui implémente l'interface MBeanServer associé au nom de domaine par défaut (DefaultDomain) sans conserver de référence sur l'instance
MBeanServer newMBeanServer(String domain)	Renvoyer une nouvelle instance d'un objet qui implémente l'interface MBeanServer associé au nom de domaine fourni en paramètre sans conserver de référence sur l'instance
void releaseMBeanServer(MBeanServer mbeanServer)	Supprimer les références au MBeanServer dans la fabrique pour permettre au ramasse-miettes de libérer l'espace mémoire de l'instance

Le paramètre domain attendu par certaines de ces méthodes permet de préciser le domaine utilisé par le serveur de MBeans. Le domaine par défaut est DefaultDomain.

Le plus simple pour obtenir une instance d'un serveur de MBeans est d'invoquer la méthode createMBeanServer() de la classe MBeanServerFactory.

Exemple :

```
...
    MBeanServer mbs = MBeanServerFactory.createMBeanServer();
...
```

La fabrique fait appel à un objet de type MBeanServerBuilder dont une implémentation par défaut est fournie.

Depuis la version 1.2 de JMX, il est possible de remplacer l'implémentation par défaut de la classe MBeanServer en définissant une classe qui héritent de la classe MBeanServerBuilder et qui possède un constructeur par défaut. Il suffit alors de passer le nom pleinement qualifié de cette classe comme valeur de la propriété javax.management.builder.initial de la JVM. Cette fonctionnalité est cependant à réserver pour des besoins très particuliers.

Il est aussi possible d'obtenir une instance de type MBeanServer en utilisant la méthode getPlatformMBeanServer() de la fabrique java.lang.management.ManagementFactory. Cette fabrique permet d'obtenir des instances des MBeans de la JVM et une instance du MBeanServer par défaut de la JVM.

Exemple :

```
...
MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
...
```

La classe MBeanServerFactory possède la méthode findMBeanServer(String) qui permet de rechercher une ou plusieurs instances de serveurs de MBeans. Elle retourne une collection qui contient les instances de serveurs de MBeans qui correspondent à l'identifiant fourni en paramètre ou à tous les serveurs de MBeans si le paramètre fourni est null.

32.6.5.2. L'instanciation et l'enregistrement d'un MBean dans le serveur

L'interface MBeanServer propose deux méthodes pour enregistrer un MBean :

- ObjectInstance registerMBean(Object, ObjectName) : enregistrer une instance d'un MBean
- ObjectInstance createMBean(String, ObjectName)

Pour utiliser la méthode registerMbean(), il faut créer une instance de type ObjectName qui va encapsuler le nom unique du MBean dans le serveur.

Il faut ensuite créer une instance du MBean et enregistrer le MBean dans le serveur en utilisant la méthode registrerMBean() de l'interface MBeanServer qui attend en paramètre l'instance du MBean et son ObjectName.

Exemple :

```
...
    ObjectName name = null;
    try {
      name = new ObjectName("fr.jmdoudoux.dej.jmx:type=PremierMBean");
      Premier mbean = new Premier();
      mbs.registerMBean(mbean, name);

    } catch (MalformedObjectNameException e) {
      e.printStackTrace();
    } catch (NullPointerException e) {
      e.printStackTrace();
    } catch (InstanceAlreadyExistsException e) {
      e.printStackTrace();
    } catch (MBeanRegistrationException e) {
      e.printStackTrace();
    } catch (NotCompliantMBeanException e) {
      e.printStackTrace();
    }

...
```

La méthode createMBean() possède plusieurs surcharges : elles permettent toutes avec différents paramètres d'instancier dynamiquement un MBean en utilisant l'introspection. Ces paramètres contiennent toujours le nom de la classe de type String et l'ObjectName du MBean. Les autres paramètres permettent de préciser le classloader à utiliser et les paramètres à fournir lors de l'invocation du constructeur du MBean.

Il est possible d'avoir plusieurs instances d'un même MBean dans un serveur de Mbeans en leur affectant à chacune un ObjectName distinct.

32.6.5.3. L'ajout d'un connecteur ou d'un adaptateur de protocoles

Pour pouvoir être utilisé par un client JMX, l'agent doit avoir au moins un connecteur ou un adaptateur de protocoles.

Généralement l'un ou l'autre sont fournis sous la forme d'un MBean qu'il convient d'instancier et d'enregistrer auprès du serveur de MBeans.

Il faut enfin les démarrer pour qu'ils commencent à écouter les appels des clients, le plus souvent en invoquant leur méthode start().

Le détail de la mise en oeuvre d'un connecteur et d'un adaptateur de protocole est proposé dans une prochaine section.

32.6.5.4. L'utilisation d'un service de l'agent

Le détail de la mise en oeuvre des services offerts par un agent est proposé dans la prochaine section.

32.7. Les services d'un agent JMX

Un agent JMX propose plusieurs services définis dans la version 1.1 des spécifications JMX visant à rendre la solution de gestion plus riche en fonctionnalités avancées :

- management applet (m-let) : permet le chargement et l'instanciation dynamique de classes en utilisant une url dédiée qui pointe sur un fichier utilisant des tags particuliers pour décrire les MBeans à traiter
- moniteur : permet d'observer les modifications de valeurs de propriétés numériques ou chaîne de caractères d'un MBean et de notifier ces changements à des abonnés
- timer : permet l'envoi de notifications répétitives ou programmées selon une valeur temporelle à des abonnés en vue de l'exécution de traitements
- relations entre MBeans : permet de définir et de maintenir des associations entre MBeans et d'assurer l'intégrité de ces relations

Ces services peuvent être implémentés sous la forme de MBeans ce qui leur permet d'être utilisés par les autres MBeans et d'être administrables.

32.7.1. Le service de type M-Let

M-Let est l'abréviation de management applet. Le service de type M-Let permet de charger un MBean local ou distant, de l'instancier et de l'enregistrer dans le serveur de MBeans. La description des MBeans à traiter est contenue dans un fichier texte possédant une syntaxe dédiée. Le fichier est fourni au service grâce à une url pointant sur un fichier local ou distant qui contient la définition des MBeans.

Le fichier doit être un fichier texte dans lequel chaque MBean doit être défini avec un tag <MLET>. Ce tag possède plusieurs attributs qui permettent de fournir les informations concernant le MBean.

Le service M-Let est enregistré en tant que MBean dans le serveur de MBeans. Le service M-Let lit le fichier de description précisé par une url. Chaque MBean décrit dans le fichier est traité par le service :

- Chargement de la classe du MBean
- Création d'une instance du MBean
- Enregistrement du MBean dans le serveur de MBeans de l'agent du service

Ce service permet de créer dynamiquement des agents extensibles.

32.7.1.1. Le format du fichier de définitions

Chaque MBean devant être traité par le service M-Let doit avoir une définition dans le fichier sous la forme d'un tag <MLET>

Le format du tag MLET est le suivant :

```
<MLET

    CODE = class | OBJECT = serfile
    ARCHIVE = "archiveList"
    [CODEBASE = codebaseURL]
    [NAME = mbeanname]
    [VERSION = version]
    >
    [arglist]
</MLET>
```

Les attributs du tags MLET sont :

Attribut	Rôle
CODE	Préciser le nom pleinement qualifié de la classe du MBean. La classe doit être présente dans un des fichiers jar indiqués par l'attribut ARCHIVE
OBJECT	Préciser un fichier .ser qui contient le résultat de la sérialisation de l'instance du MBean. Ce fichier doit être présent dans un des fichiers jar indiqués par l'attribut ARCHIVE
ARCHIVE	Préciser un ou plusieurs fichiers jar contenant le ou les MBeans et leurs dépendances. Si plusieurs sont précisés, la valeur de l'attribut doit être entourée de double quotes et chaque jar doit être séparés avec un caractère virgule. Tous les jars utilisés doivent être stockés dans le répertoire précisé par l'attribut CODEBASE . (obligatoire)
CODEBASE	Préciser le répertoire dans lequel les jars sont stockées. L'utilisation de cet attribut n'est obligatoire que si les jars ne sont pas stockés dans le même répertoire que le fichier de description
NAME	Préciser l'ObjectName du MBean lors de son enregistrement dans le serveur de MBeans. Si la valeur de l'attribut commence par un caractère « : » alors l'ObjectName sera préfixé par le nom de domaine par défaut du serveur de MBeans
VERSION	Préciser le numéro de version du MBean et du jar qui le contient

Deux attributs sont obligatoires :

- CODE ou OBJECT : pour préciser le nom de classe ou le fichier .ser qui contient le résultat de la sérialisation du MBean. CODE et OBJECT sont mutuellement exclusifs.
- ARCHIVE pour préciser le ou les jars contenant les classes requises

Le tag MLET possède le tag fils <ARG> qui permet de préciser des arguments qui seront passés au constructeur du MBean lors de son instanciation.

Le tag <ARG> possède deux attributs :

Attribut	Rôle
TYPE	Préciser le type de l'argument. Seuls quelques types possédant une représentation sous la forme d'une chaîne de caractères peuvent être utilisés : java.lang.Boolean, java.lang.Byte, java.lang.Short, java.lang.Long, java.lang.Integer, java.lang.Float, java.lang.Double, java.lang.String
VALUE	Préciser la valeur de l'argument sous la forme d'une chaîne de caractères

Lors de l'instanciation du MBean, le service M-Let va rechercher un constructeur du MBean dont la signature corresponde aux arguments précisés par le tag <ARG>.

Le fichier peut contenir plusieurs tags <MLET>, un pour chaque MBean qui devra être instancié et enregistré dans le serveur de MBeans.

32.7.1.2. L'instanciation et l'utilisation d'un service M-Let dans un agent

La classe javax.management.loading.MLet est une implémentation du service M-Let fournie en standard. L'implémentation d'un service M-Let doit implémenter l'interface javax.management.loading.MLetMBean.

La classe MLet hérite de la classe URLClassLoader ce qui lui permet de télécharger des classes à travers le réseau.

C'est un MBean qui doit être instancié et enregistré dans le serveur de MBean : il est ainsi possible d'utiliser ce MBean à distance en passant par l'agent JMX.

La classe MLet propose la méthode getMbeansFromURL() qui attend en paramètre l'url du fichier de description et qui permet de lire le fichier et de traiter les MBeans qu'il contient. Deux surcharges permettent de préciser l'url sous la forme d'une chaîne de caractères ou d'un objet de type java.net.URL.

32.7.1.3. Un exemple de mise en oeuvre du service M-Let

Il faut développer un agent qui va instancier et enregistrer un objet de type MLet.

L'instance de cet objet va lire un fichier de description qui va permettre d'instancier et d'enregistrer un MBean dans le serveur de MBeans.

Exemple :

```
package fr.jmdoudoux.dej.jmx;

import java.lang.management.ManagementFactory;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.Arrays;
import java.util.Set;

import javax.management.Attribute;
import javax.management.AttributeNotFoundException;
import javax.management.InstanceAlreadyExistsException;
import javax.management.InstanceNotFoundException;
import javax.management.InvalidAttributeValueException;
import javax.management.MBeanException;
import javax.management.MBeanRegistrationException;
import javax.management.MBeanServer;
import javax.management.MalformedObjectNameException;
import javax.management.NotCompliantMBeanException;
import javax.management.ObjectName;
import javax.management.ReflectionException;
import javax.management.ServiceNotFoundException;
import javax.management.loading.MLet;

public class LancerAgentAvecMLet {
  public static void main(String[] args) {
    MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();

    ObjectName name = null;
    try {
      name = new ObjectName("fr.jmdoudoux.dej.jmx:type=PremierMBean");

      // Instanciation et enregistrement du Service MLet
      System.out.println("Instanciation et enregistrement du Service MLet");
      MLet mlet = new MLet();
      mlet.setLibraryDirectory("c:/temp");
      mbs.registerMBean(mlet, new ObjectName("Services:type=MLet"));

      // Lecture du fichier de configuration pour instanciation et
      // enregistrement du MBean
      System.out.println("\nLecture du fichier de configuration");
      Set<Object> mbeans = mlet.getMBeansFromURL(new URL(
          "http://localhost:8080/jmx/mlet.txt"));
      for (Object obj : mbeans) {
        System.out.println("Object = " + obj);
      }

      System.out.println("\nClasspath du service MLet : "
          + Arrays.asList(mlet.getURLs()));

      System.out.println("\nRecherche du mbean enregistré");
      Set<ObjectName> names = mbs.queryNames(name, null);
      for (ObjectName objName : names) {
        System.out.println("ObjectName=" + objName);
      }

      System.out.println("\nExecution de l'agent ...");
      while (true) {

        try {
          Thread.sleep(1000);
        } catch (InterruptedException e) {
        }

        int valeur = Integer.valueOf(mbs.getAttribute(name, "Valeur")
            .toString());
        Attribute attr = new Attribute("Valeur", valeur + 1);
        mbs.setAttribute(name, attr);
      }
    } catch (MalformedObjectNameException e) {
      e.printStackTrace();
    } catch (NullPointerException e) {
      e.printStackTrace();
    } catch (InstanceAlreadyExistsException e) {
      e.printStackTrace();
    } catch (MBeanRegistrationException e) {
      e.printStackTrace();
    } catch (NotCompliantMBeanException e) {
      e.printStackTrace();
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (NumberFormatException e) {
      e.printStackTrace();
    } catch (AttributeNotFoundException e) {
      e.printStackTrace();
    } catch (InstanceNotFoundException e) {
      e.printStackTrace();
    } catch (MBeanException e) {
      e.printStackTrace();
    } catch (ReflectionException e) {
      e.printStackTrace();
    } catch (InvalidAttributeValueException e) {
      e.printStackTrace();
    } catch (ServiceNotFoundException e) {
      e.printStackTrace();
    }
  }
}
```

Il faut rédiger le fichier de description.

Exemple :

```
<MLET CODE=fr.jmdoudoux.dej.jmx.Premier
 ARCHIVE="TestJMX.jar"
 CODEBASE=http://localhost:8080/jmx/
 NAME=fr.jmdoudoux.dej.jmx:type=PremierMBean></MLET>
```

Il faut un serveur web sur lequel on place :

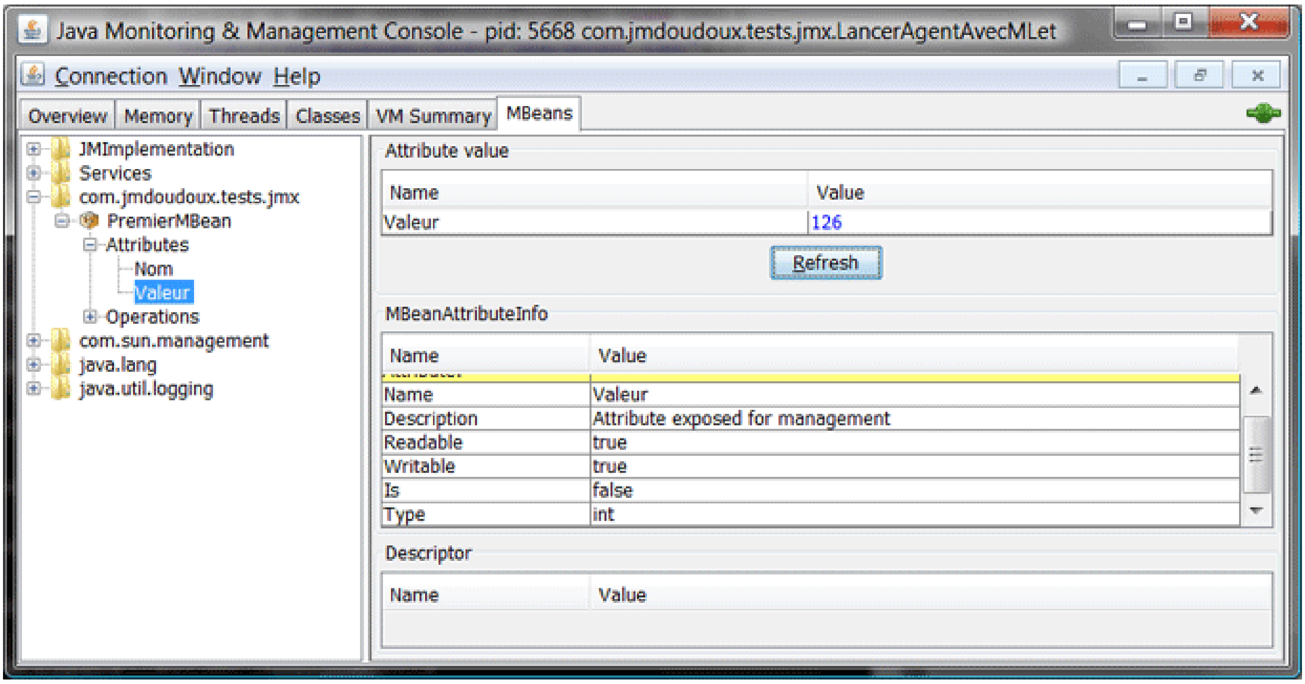
- le fichier mlet.txt
- un fichier jar qui contient la classe du MBean (TestJMX.jar dans cet exemple)

Dans l'exemple de cette section, c'est une simple webapp déployée dans un serveur Tomcat qui contient à sa racine les deux fichiers. Il faut exécuter l'agent.

Résultat :

```
Instanciation et enregistrement du Service MLet

Lecture du fichier de configuration
Object = fr.jmdoudoux.dej.jmx.Premier[fr.jmdoudoux.dej.jmx:type=PremierMBean]

Classpath du service MLet : [http://localhost:8080/jmx/TestJMX.jar]

Recherche du mbean enregistré
ObjectName=fr.jmdoudoux.dej.jmx:type=PremierMBean

Execution de l'agent ...
```

Il est possible de consulter le MBean enregistré dans l'agent par exemple avec JConsole.



Attribute value

Name	Value
Valeur	126

MBeanAttributeInfo

Name	Value
Name	Valeur
Description	Attribute exposed for management
Readable	true
Writable	true
Is	false
Type	int

Descriptor

Name	Value

32.7.2. Le service de type Timer

Le service Timer permet d'envoyer des notifications prédéfinies à un moment donné ou périodiquement. Ces notifications sont envoyées à tous les objets qui se sont abonnés pour recevoir les notifications émises par le service.

Les caractéristiques de l'émission d'une notification sont assez souples : elle démarre à une certaine date/heure et est répétée à intervalles de temps réguliers durant une période ou pour un certain nombre d'occurrences.

Le service Timer est implémenté sous la forme d'un MBean ce qui permet de l'administrer au travers de JMX lui-même.

Les notifications émises par le service Timer sont de type TimerNotification.

L'implémentation du service Timer est encapsulée dans la classe javax.management.timer.Timer

32.7.2.1. Les fonctionnalités du service Timer

Le service Timer est implémenté sous la forme d'un MBean. Pour l'utiliser, il faut l'instancier et l'enregistrer dans le serveur de MBeans.

Pour activer le service, il faut utiliser sa méthode start(). Pour le désactiver, il faut utiliser la méthode stop(). Avant l'appel à la méthode start() et après la méthode stop() aucune notification n'est émise même si les conditions d'une émission sont remplies. Si le service est redémarré et que la méthode setSendPastNotifications() a été invoquée avec le paramètre true alors les notifications ratées seront émises.

La méthode isActive() permet de savoir si le service est actif ou non.

Pour s'abonner aux notifications, un client ou une classe doivent s'enregistrer en tant que listener sur le MBean du service Timer. Chaque fois qu'une condition est remplie, une notification est envoyée à tous les abonnés.

Dès que les conditions d'une notification ne peuvent plus être remplies (par exemple si le nombre d'occurrences est atteint), la définition de la notification est supprimée automatiquement de la liste maintenue dans le service.

Il est possible de supprimer la définition d'une notification en utilisant la méthode removeNotification() qui attend l'identifiant de cette définition. Il est aussi possible de supprimer plusieurs définitions en utilisant la méthode removeNotifications() qui attend en paramètre leur type : dans ce cas, elle va supprimer toutes les définitions de notifications qui ont le type fourni en paramètre.

La méthode removeAllNotifications() permet de supprimer toutes les notifications contenues dans le service.

32.7.2.2. L'ajout d'une définition de notifications

Le service Timer maintient une liste des définitions de notifications qu'il aura à traiter. La méthode addNotification() permet d'ajouter une définition de notifications. Cette méthode possède plusieurs surcharges qui ont toutes quatre paramètres en commun :

- type : chaîne de caractères qui précise le type de la notification
- message : chaîne de caractères qui contient le message de la notification
- userData : un objet qui encapsule des données dédiées à la notification
- date : date/heure de début d'émission des notifications

Plusieurs surcharges attendent en paramètres un ou plusieurs des paramètres ci-dessous :

- period : intervalle de temps en millisecondes entre l'émission de deux notifications. La valeur 0 inhibe toute répétition.
- nbOccurences : nombre total d'émissions de notifications à réaliser. Avec la valeur 0 les émissions sont infinies.
- fireImmediate : booléen qui précise si l'émission de la première notification doit avoir lieu dès l'ajout de la définition de notifications au service. La valeur true émet une notification immédiatement, la valeur false n'émet la première notification qu'un fois que les conditions de la définition sont remplies.

La méthode addNotification() peut lever une exception de type IllegalArgumentException si une ou plusieurs valeurs de ses paramètres est invalide, par exemple :

- date est null (Timer notification date cannot be null)

- period ou nbOccurences a une valeur négative (Negative values for the periodicity)

La méthode addNotification() renvoie un identifiant de la définition des notifications qui peut être utile notamment pour supprimer la définition.

Remarque : il n'est pas possible de modifier les paramètres d'une définition de notifications.

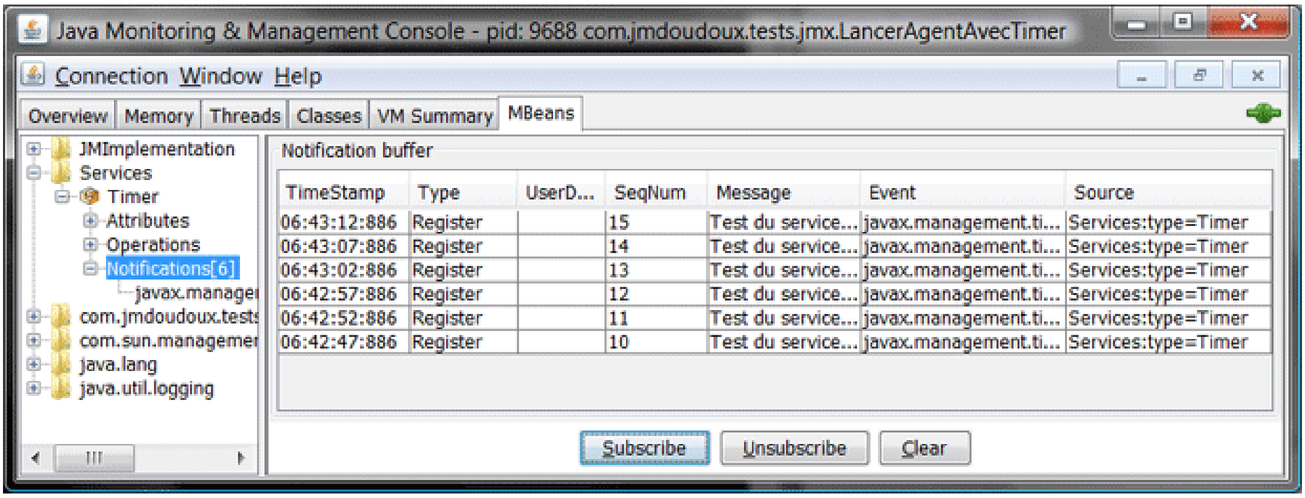
32.7.2.3. Un exemple de mise en oeuvre du service Timer

Dans l'exemple de cette section, une notification sera émise toutes les 5 secondes pour une durée infinie.

Exemple :

```
package fr.jmdoudoux.dej.jmx;

import java.io.IOException;
import java.lang.management.ManagementFactory;
import java.net.MalformedURLException;
import java.util.Date;

import javax.management.InstanceAlreadyExistsException;
import javax.management.MBeanRegistrationException;
import javax.management.MBeanServer;
import javax.management.MalformedObjectNameException;
import javax.management.NotCompliantMBeanException;
import javax.management.ObjectName;
import javax.management.remote.JMXConnectorServer;
import javax.management.remote.JMXConnectorServerFactory;
import javax.management.remote.JMXServiceURL;

public class LancerAgentAvecTimer {

  public static void main(String[] args) {
    MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();

    try {

      // instanciation et enregistrement du service Timer
      javax.management.timer.Timer timer = new javax.management.timer.Timer();
      mbs.registerMBean(timer, new ObjectName("Services:type=Timer"));

      // ajout de la définition des notifications
      timer.addNotification("Register", "Test du service timer", new String(),
          new Date(), 5000, 0);
      timer.setSendPastNotifications(false);
      timer.start();

      // Creation et démarrage du connecteur RMI
      JMXServiceURL url = new JMXServiceURL(
          "service:jmx:rmi:///jndi/rmi://localhost:9000/server");
      JMXConnectorServer cs = JMXConnectorServerFactory.newJMXConnectorServer(
          url, null, mbs);
      cs.start();
      System.out.println("Lancement connecteur RMI " + url);

      while (true) {
        Thread.sleep(1000);
      }

    } catch (MalformedObjectNameException e) {
      e.printStackTrace();
    } catch (NullPointerException e) {
      e.printStackTrace();
    } catch (InstanceAlreadyExistsException e) {
      e.printStackTrace();
    } catch (MBeanRegistrationException e) {
      e.printStackTrace();
    } catch (NotCompliantMBeanException e) {
      e.printStackTrace();
    } catch (InterruptedException e) {
      e.printStackTrace();
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Pour voir les notifications dans JConsole, il faut sélectionner « Notitifications » pour le MBean du service Timer et cliquer sur le bouton « Subscribe ».



TimeStamp	Type	UserD...	SeqNum	Message	Event	Source
06:43:12:886	Register		15	Test du service...	javax.management.ti...	Services:type=Timer
06:43:07:886	Register		14	Test du service...	javax.management.ti...	Services:type=Timer
06:43:02:886	Register		13	Test du service...	javax.management.ti...	Services:type=Timer
06:42:57:886	Register		12	Test du service...	javax.management.ti...	Services:type=Timer
06:42:52:886	Register		11	Test du service...	javax.management.ti...	Services:type=Timer
06:42:47:886	Register		10	Test du service...	javax.management.ti...	Services:type=Timer

32.7.3. Le service de type Monitor

La suite de cette section sera développée dans une version future de ce document

32.7.4. Le service de type Relation

La suite de cette section sera développée dans une version future de ce document

32.8. La couche services distribués

Il existe de nombreuses applications de gestion et de monitoring qui utilisent des protocoles standard comme SNMP (Simple Network Management Protocol) ou propriétaires.

Une application de gestion permet à des utilisateurs d'interagir avec les MBeans en communiquant avec le serveur de MBeans ou un service de l'agent JMX. Une application web utilisant un adaptateur de protocole pour HTML ou une application utilisant un adaptateur de protocole pour SNMP sont des exemples d'applications de gestion. Ces applications peuvent ne pas implémenter l'API JMX et dans ce cas elles communiquent avec l'agent JMX grâce à un protocole dédié.

La couche services distribuées fournit des interfaces et des composants pour permettre à des outils distants de communiquer avec l'agent JMX.

La couche services distribuées contient une application de gestion qui va interagir avec l'agent JMX. Les clients JMX distants s'exécutent dans une JVM différente de celle du serveur de MBeans et utilisent un protocole pour communiquer avec ce dernier.

Un client JMX distant ne peut pas utiliser le serveur de MBeans directement. Les composants de cette couche permettent d'accéder à un serveur de MBeans au travers de différents protocoles.

La manière dont un agent JMX est accédé au travers du réseaux est spécifié par la JSR 160 (JMX Remoting) qui est une fonctionnalité de la version 1.2 de JMX.

32.8.1. L'interface MBeanServerConnection

Un client JMX distant manipulent des MBeans en utilisant un objet qui implémente l'interface MBeanServerConnection. Une instance de l'interface MBeanServerConnection permet de se connecter à un serveur de MBeans local ou distant et d'interagir avec lui.

Pour utiliser un MBean local, il est possible d'utiliser directement le serveur de MBeans. Pour accéder à un MBean distant, il faut obligatoirement utiliser une instance de l'interface MBeanServerConnection.

Le client utilise les méthodes de l'interface MBeanServerConnection pour accéder aux fonctionnalités exposées par un MBean :

- createMBean() : pour instancier et enregister un MBean dans le serveur de MBeans
- getAttribute() : pour obtenir la valeur d'un attribut d'un MBean
- setAttribute() : pour mettre à jour la valeur d'un attribut d'un MBean
- invoke() : pour invoquer un constructeur ou une méthode d'un MBean
- isRegistred() : pour déterminer si un MBean est déjà enregistré dans le serveur de MBeans
- queryMBeans() : pour obtenir une collection des MBeans enregistrés dans le serveur de MBeans
- queryNames() : pour obtenir une collection des noms des MBeans enregistrés dans le serveur de MBeans

32.8.2. Les connecteurs et les adaptateurs de protocoles

Pour permettre la communication entre un agent et un client JMX, JMX propose des adaptateurs de protocoles ou des connecteurs qui se chargent de la communication entre l'application de gestion et l'agent JMX avec un protocole particulier.

La communication entre un agent JMX et une application de gestion peut donc être assurée par deux mécanismes :

- les connecteurs (connectors) : ils assurent la communication entre l'application de gestion et l'agent JMX. Le protocole utilisé par défaut est RMI. JMX définit aussi un protocole reposant sur des sockets TCP dont le support est optionnel et nommé JMX Messaging Protocol (JMXMP).
- les adaptateurs de protocoles (protocol adaptors) : ils permettent à une application de manipuler les MBeans enregistrés dans un serveur de MBeans en utilisant un certain protocole (par exemple SNMP ou l'adaptateur de protocole pour HTML qui permet de manipuler les MBeans d'un agent JMX au moyen d'un simple navigateur web). Les interactions avec les MBeans se font au travers de ce protocole : les actions à réaliser sont converties de et vers le protocole utilisé.

Les connecteurs et les adaptateurs de protocoles permettent l'accès, par une application distante, aux MBeans enregistrés dans un agent JMX. Ils permettent un accès aux services de l'agent et aux MBeans enregistrés dans celui-ci.

Ainsi pour être utilisable, un agent JMX doit fournir ou moins un connecteur ou un adaptateur de protocole. La plate-forme Java SE fournit en standard un connecteur reposant sur RMI.

Un agent peut être accédé par plusieurs connecteurs ou adaptateurs de protocoles simultanément.

32.8.2.1. Les connecteurs

Un connecteur permet le dialogue entre l'agent et l'application de gestion distante grâce à un protocole dédié. Un connecteur est composé d'une partie cliente liée à l'application de gestion et d'une partie serveur liée à l'agent JMX. La partie serveur du connecteur attend les connexions de la partie cliente : c'est donc la partie cliente qui est responsable de l'initialisation de la connexion.

Un connecteur permet donc à un client JMX distant de communiquer avec un agent JMX. L'agent JMX est chargé d'initialiser et de configurer le connecteur côté serveur. Le client JMX est chargé d'initialiser et de configurer le connecteur côté client.

Un connecteur permet d'obtenir une instance de l'interface MBeanServerConnection.

Les spécifications de l'API JMX Remote définissent trois catégories de connecteurs :

- connecteur RMI : ce type de connecteur utilise la technologie RMI de Java. L'implémentation de ce type de connecteur est obligatoire.
- connecteur générique : ce type de connecteur utilise des sockets TCP et le protocole JMXMP (JMX Messaging Protocol). L'implémentation de ce type de connecteur est optionnelle.
- connecteur utilisant un protocole spécifique : ce type de connecteur utilise un protocole qui n'est pas défini par JMX

Le connecteur RMI peut utiliser les deux standards de transport de RMI :

- Java Remote Method Protocol (JRMP)
- Internet Inter-ORB Protocol (IIOP)

32.8.2.2. Les adaptateurs de protocoles

Les adaptateurs de protocoles permettent un accès à un agent JMX au travers d'un protocole particulier. Celui-ci peut par exemple être un protocole propriétaire utilisé par une application de gestion commerciale.

Un adaptateur de protocole permet à une application qui n'utilise pas JMX de communiquer avec un agent JMX sans que la partie cliente n'utilise aucune API de JMX.

Par exemple, l'adaptateur HTML fourni avec l'implémentation de référence de JMX permet d'interagir avec les MBeans d'un agent JMX utilisant cet adaptateur avec un simple navigateur web.

32.8.3. L'utilisation du connecteur RMI

Pour utiliser le connecteur RMI, l'agent JMX doit créer un connecteur RMI et le lancer. Le connecteur est accessible par une url encapsulée dans la classe JMXServiceURL.

La partie serveur d'un connecteur est encapsulée dans la classe JMXConnectorServer. Une instance de la classe JMXConnectorServer est obtenue en utilisant la méthode newJMXConnectorServer() de la fabrique JMXConnectorServerFactory.

Exemple :

```
package fr.jmdoudoux.dej.jmx;

import java.io.IOException;
import java.lang.management.ManagementFactory;
import java.net.MalformedURLException;

import javax.management.InstanceAlreadyExistsException;
import javax.management.MBeanRegistrationException;
import javax.management.MBeanServer;
import javax.management.MalformedObjectNameException;
import javax.management.NotCompliantMBeanException;
import javax.management.ObjectName;
import javax.management.remote.JMXConnectorServer;
import javax.management.remote.JMXConnectorServerFactory;
import javax.management.remote.JMXServiceURL;

public class LancerAgentAvecRMI {

  public static void main(String[] args) {

    System.out.println("Lancement de l'agent JMX");

    MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();

    ObjectName name = null;
    try {
      System.out.println("Instanciation et enregistrement du MBean");

      name = new ObjectName("fr.jmdoudoux.dej.jmx:type=PremierMBean");

      Premier mbean = new Premier();

      mbs.registerMBean(mbean, name);

      // Creation et demarrage du connecteur RMI
      JMXServiceURL url = new JMXServiceURL(
          "service:jmx:rmi:///jndi/rmi://localhost:9000/server");
      JMXConnectorServer cs = JMXConnectorServerFactory.newJMXConnectorServer(
          url, null, mbs);
      cs.start();
      System.out.println("Lancement connecteur RMI "+url);

      int i = 0;
      System.out.println("Incrementation de la valeur du MBean ...");
      while (i < 60) {

        mbean.setValeur(mbean.getValeur() + 1);
        Thread.sleep(1000);
        i++;
      }

      System.out.println("Arret connecteur RMI ");
      cs.stop();

      System.out.println("Arret de l'agent JMX");

    } catch (MalformedObjectNameException e) {
      e.printStackTrace();
    } catch (NullPointerException e) {
      e.printStackTrace();
    } catch (InstanceAlreadyExistsException e) {
      e.printStackTrace();
    } catch (MBeanRegistrationException e) {
      e.printStackTrace();
    } catch (NotCompliantMBeanException e) {
      e.printStackTrace();
    } catch (InterruptedException e) {
      e.printStackTrace();
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

L'appel à la méthode stop() est important car il permet notamment de supprimer le connecteur enregistré dans le registre RMI.

Un exception de type javax.naming.NameAlreadyBoundException est levée si l'agent est lancé et que le connecteur est déjà enregistré dans le registre RMI. Dans ce cas, il faut redémarrer le registre.

Exemple :

```
C:\Users\Jean Michel\workspace\TestJMX\bin>java fr.jmdoudoux.dej.jmx.LancerAg
entAvecRMI
Lancement de l'agent JMX
Instanciation et enregistrement du MBean
Lancement connecteur RMI service:jmx:rmi:///jndi/rmi://localhost:9000/server
Incrementation de la valeur du MBean ...

C:\Users\Jean Michel\workspace\TestJMX\bin>java fr.jmdoudoux.dej.jmx.LancerAg
entAvecRMI
Lancement de l'agent JMX
Instanciation et enregistrement du MBean
java.io.IOException: Cannot bind to URL [rmi://localhost:9000/server]: javax.nam
ing.NameAlreadyBoundException: server [Root exception is java.rmi.AlreadyBoundEx
ception: server]
        at javax.management.remote.rmi.RMIConnectorServer.newIOException(RMIConn
ectorServer.java:804)
        at javax.management.remote.rmi.RMIConnectorServer.start(RMIConnectorServ
er.java:417)
        at fr.jmdoudoux.dej.jmx.LancerAgentAvecRMI.main(LancerAgentAvecRMI.ja
va:40)
Caused by: javax.naming.NameAlreadyBoundException: server [Root exception is jav
a.rmi.AlreadyBoundException: server]
        at com.sun.jndi.rmi.registry.RegistryContext.bind(RegistryContext.java:1
22)
        at com.sun.jndi.toolkit.url.GenericURLContext.bind(GenericURLContext.jav
a:208)
        at javax.naming.InitialContext.bind(InitialContext.java:400)
        at javax.management.remote.rmi.RMIConnectorServer.bind(RMIConnectorServe
```

Pour utiliser un connecteur RMI, il faut obligatoirement lancer un registre RMI

Exemple :

```
C:\>rmiregistry 9000
```

Le paramètre précisé à la commande rmiregistry est le port utilisé pour les communications.

Le client JMX doit obtenir une instance du type MBeanServerConnection qui va lui permettre de se connecter sur le serveur de MBeans de l'agent JMX. Une telle instance est obtenue en utilisant un objet de type JMXConnector fourni par l'invocation de la méthode connect() de la fabrique JMXConnectorFactory. La méthode connect() attend en premier paramètre l'url de connexion sur le serveur de MBeans de l'agent JMX.

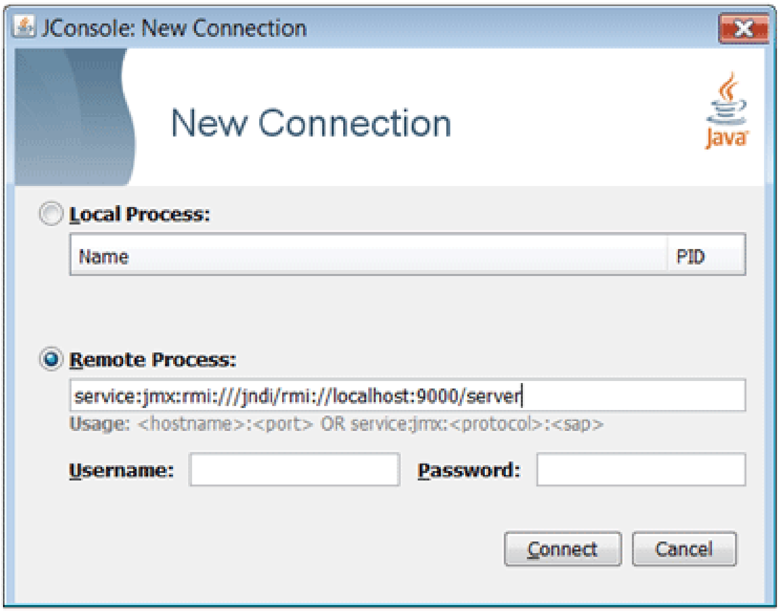
La méthode getMBeanServerConnection() de l'instance de type JMXConnector renvoie un objet de type MBeanServerConnection qui permet d'interagir avec le serveur de MBeans.

Exemple :

```
package fr.jmdoudoux.dej.jmx;

import java.io.IOException;
import java.net.MalformedURLException;

import javax.management.MBeanServerConnection;
import javax.management.MBeanServerInvocationHandler;
import javax.management.MalformedObjectNameException;
import javax.management.ObjectName;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;
```

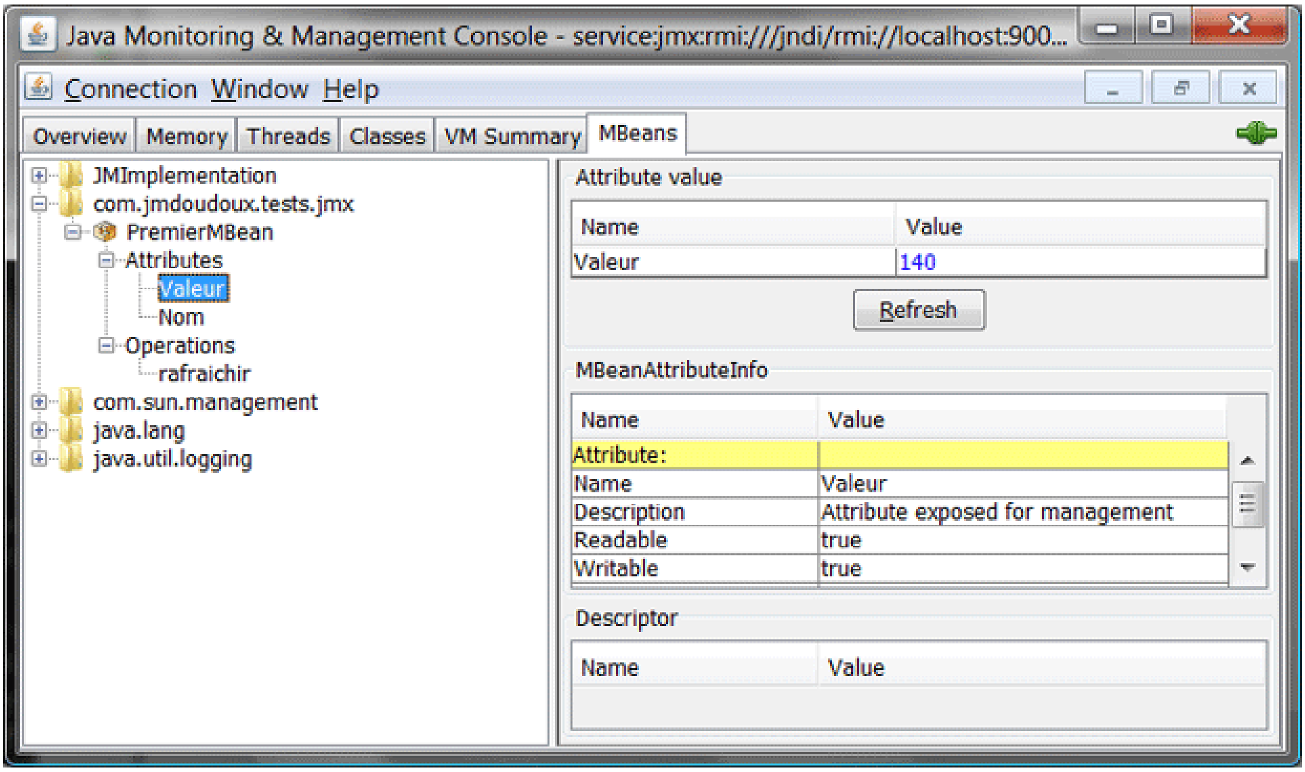
```
public class ClientJMXAvecRMI {
  public static void main(String[] args) {
    MBeanServerConnection mbsc = null;
    JMXConnector connecteur = null;

    ObjectName name = null;
    try {
      name = new ObjectName("fr.jmdoudoux.dej.jmx:type=PremierMBean");

      JMXServiceURL url = new JMXServiceURL(
          "service:jmx:rmi:///jndi/rmi://localhost:9000/server");

      connecteur = JMXConnectorFactory.connect(url, null);

      mbsc = connecteur.getMBeanServerConnection();

      PremierMBean mbean = (PremierMBean) MBeanServerInvocationHandler
          .newProxyInstance(mbsc, name, PremierMBean.class, false);
      int valeur = mbean.getValeur();
      System.out.println("valeur = " + valeur);
    } catch (MalformedObjectNameException e) {
      e.printStackTrace();
    } catch (NullPointerException e) {
      e.printStackTrace();
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      if (mbsc != null) {
        try {
          connecteur.close();
        } catch (IOException e) {
          e.printStackTrace();
        }
      }
    }

  }
}
```

Il est aussi possible d'utiliser un autre client JMX, par exemple l'outil JConsole



service:jmx:rmi:///jndi/rmi://localhost:9000/server

L'onglet MBean permet de retrouver le MBean enregistré dans le serveur de MBeans et d'interagir avec lui.



Attribute value

Name	Value
Valeur	140

MBeanAttributeInfo

Name	Value
Attribute:	
Name	Valeur
Description	Attribute exposed for management
Readable	true
Writable	true

Descriptor

Name	Value

32.8.4. L'utilisation du connecteur utilisant le protocole JMXMP

Un connecteur générique utilise le protocole JMXMP qui repose sur des sockets avec le protocole TCP pour les communications, la sérialisation pour l'échange des objets et les API standard de Java pour la sécurité notamment JSSE et JASS.

L'utilisation du connecteur JMXMP est simple. Pour utiliser ce protocole, il faut une implémentation de ce protocole par exemple celle fournie avec l'implémentation de référence de la JSR 160. Il suffit alors d'ajouter la bibliothèque jmxremote_optional.jar dans le classpath.

Le protocole JMXMP permet d'échanger des objets Java sérialisés par une connexion TCP. La communication entre le client et le serveur de MBeans n'a alors besoin que de définir un port de communication.

Le code de l'agent est similaire à celui de l'agent utilisant le connecteur RMI avec cependant une url de connexion dédiée au protocole JMXMP

Exemple :

```
package fr.jmdoudoux.dej.jmx;

import java.io.IOException;
import java.lang.management.ManagementFactory;
import java.net.MalformedURLException;

import javax.management.InstanceAlreadyExistsException;
import javax.management.MBeanRegistrationException;
import javax.management.MBeanServer;
import javax.management.MalformedObjectNameException;
import javax.management.NotCompliantMBeanException;
import javax.management.ObjectName;
import javax.management.remote.JMXConnectorServer;
import javax.management.remote.JMXConnectorServerFactory;
import javax.management.remote.JMXServiceURL;

public class LancerAgentAvecJMXMP {

  public static void main(String[] args) {

    System.out.println("Lancement de l'agent JMX");
    MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();

    ObjectName name = null;
    try {
      System.out.println("Instanciation et enregistrement du Mbean");

      name = new ObjectName("fr.jmdoudoux.dej.jmx:type=PremierMBean");

      Premier mbean = new Premier();

      mbs.registerMBean(mbean, name);

      // Creation et demarrage du connecteur pour le protocole JMXMP
      JMXServiceURL url = new JMXServiceURL(
          "service:jmx:jmxmp://localhost:9998");

      JMXConnectorServer cs = JMXConnectorServerFactory.newJMXConnectorServer(url, null, mbs);
      cs.start();

      System.out.println("Lancement connecteur pour le protocle JMXMP " + url);

      int i = 0;
      System.out.println("Incrementation de la valeur du MBean ...");
      while (i < 6000) {

        mbean.setValeur(mbean.getValeur() + 1);
        Thread.sleep(1000);
        i++;
      }

      System.out.println("Arret connecteur pour le protocle JMXMP ");
      cs.stop();

      System.out.println("Arret de l'agent JMX");

    } catch (MalformedObjectNameException e) {
      e.printStackTrace();
    } catch (NullPointerException e) {
      e.printStackTrace();
    } catch (InstanceAlreadyExistsException e) {
      e.printStackTrace();
    } catch (MBeanRegistrationException e) {
      e.printStackTrace();
    } catch (NotCompliantMBeanException e) {
      e.printStackTrace();
    } catch (InterruptedException e) {
      e.printStackTrace();
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Le code du client est aussi similaire à celui du client utilisant le connecteur RMI avec l'utilisation de l'url dédiée.

Exemple :

```
package fr.jmdoudoux.dej.jmx;

import java.io.IOException;
import java.net.MalformedURLException;

import javax.management.MBeanServerConnection;
import javax.management.MBeanServerInvocationHandler;
import javax.management.MalformedObjectNameException;
import javax.management.ObjectName;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;
```

```
public class ClientJMXAvecJMXMP {
  public static void main(String[] args) {
    MBeanServerConnection mbsc = null;
    JMXConnector connecteur = null;

    ObjectName name = null;
    try {
      name = new ObjectName("fr.jmdoudoux.dej.jmx:type=PremierMBean");

      JMXServiceURL url = new JMXServiceURL(
          "service:jmx:jmxmp://localhost:9998");

      connecteur = JMXConnectorFactory.connect(url, null);

      mbsc = connecteur.getMBeanServerConnection();

      PremierMBean mbean = (PremierMBean) MBeanServerInvocationHandler
          .newProxyInstance(mbsc, name, PremierMBean.class, false);
      int valeur = mbean.getValeur();
      System.out.println("valeur = " + valeur);
      mbean.rafraichir();

    } catch (MalformedObjectNameException e) {
      e.printStackTrace();
    } catch (NullPointerException e) {
      e.printStackTrace();
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      if (mbsc != null) {
        try {
          connecteur.close();
        } catch (IOException e) {
          e.printStackTrace();
        }
      }
    }

  }
}
```

L'exécution du client sans mettre la bibliothèque jmxremote_optional.jar lève une exception de type MalformedURLException

Résultat :

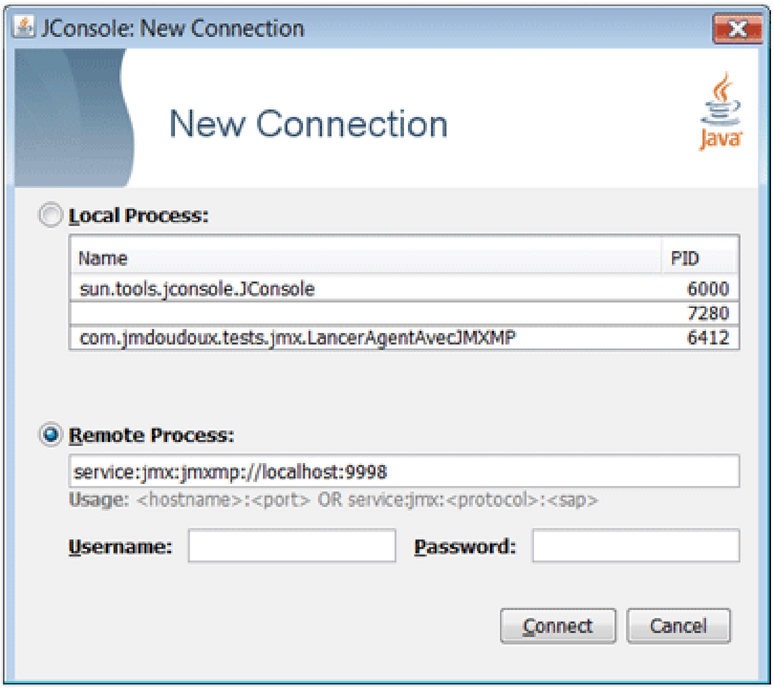
```
C:\Users\Jean Michel\workspace\TestJMX\bin>java -cp . fr.jmdoudoux.dej.jmx.La
ncerAgentAvecJMXMP
Lancement de l'agent JMX
Instanciation et enregistrement du Mbean
java.net.MalformedURLException: Unsupported protocol: jmxmp
        at javax.management.remote.JMXConnectorServerFactory.newJMXConnectorServ
er(JMXConnectorServerFactory.java:323)
        at fr.jmdoudoux.dej.jmx.LancerAgentAvecJMXMP.main(LancerAgentAvecJMXM
P.java:39)
```

Avec la bibliothèque ajoutée au classpath, le client peut se connecter à l'agent et interagir avec le MBean.

Exemple :

```
C:\Users\Jean Michel\workspace\TestJMX\bin>java -cp
.;C:\java\api\jmxremote-1_0_1-bin\lib\jmxremote_optional.jar
fr.jmdoudoux.dej.jmx.ClientJMXAvecJMXMP
valeur = 867
```

Il est aussi possible d'utiliser l'outil JConsole pour se connecter à l'agent en utilisant comme paramètre de connexion l'url de l'agent.



Name	PID
sun.tools.jconsole.JConsole	6000
	7280
com.jmdoudoux.tests.jmx.LancerAgentAvecJMXMP	6412

service:jmx:jmxmp://localhost:9998

Usage: <hostname>:<port> OR service:jmx:<protocol>:<sap>

Dans ce cas pour que la connexion réussisse, il faut ajouter la bibliothèque au classpath pour permettre à JConsole d'avoir l'implémentation du protocole JMXMP. Le plus simple est d'ajouter le fichier jar dans le sous-répertoire lib/ext du répertoire d'installation du JRE.

32.8.5. L'utilisation de l'adaptateur de protocole HTML

L'adaptateur de protocoles HTML permet d'accéder à un agent en utilisant le protocole HTML : ainsi un simple navigateur web peut faire office de client JMX.

L'adaptateur de protocoles HTML est implémenté sous la forme d'un MBean et peut donc à ce titre être géré comme tout MBean.

Bien que JMX soit intégré à Java 5, l'adaptateur de protocole HTML n'est pas fourni en standard avec le JDK. Il faut télécharger l'implémentation de référence de JMX à l'url :

https://www.oracle.com/java/technologies/java-archive-downloads-java-plat-downloads.html

L'archive jmx-1_2_1-ri.zip contient dans le sous-répertoire lib une bibliothèque nommée jmxtools.jar qui doit être ajoutée au classpath.

L'adaptateur est encapsulé dans la classe com.sun.jdmk.comm.HtmlAdaptorServer : c'est un MBean qui doit être instancié et enregistré dans le serveur de MBeans de l'agent.

Le port utilisé par l'adaptateur doit être précisé en utilisant la méthode setPort(). Par défaut, c'est le port 8082 qui est utilisé.

La méthode start() démarre l'adaptateur et permet de traiter les requêtes HTML.

Exemple :

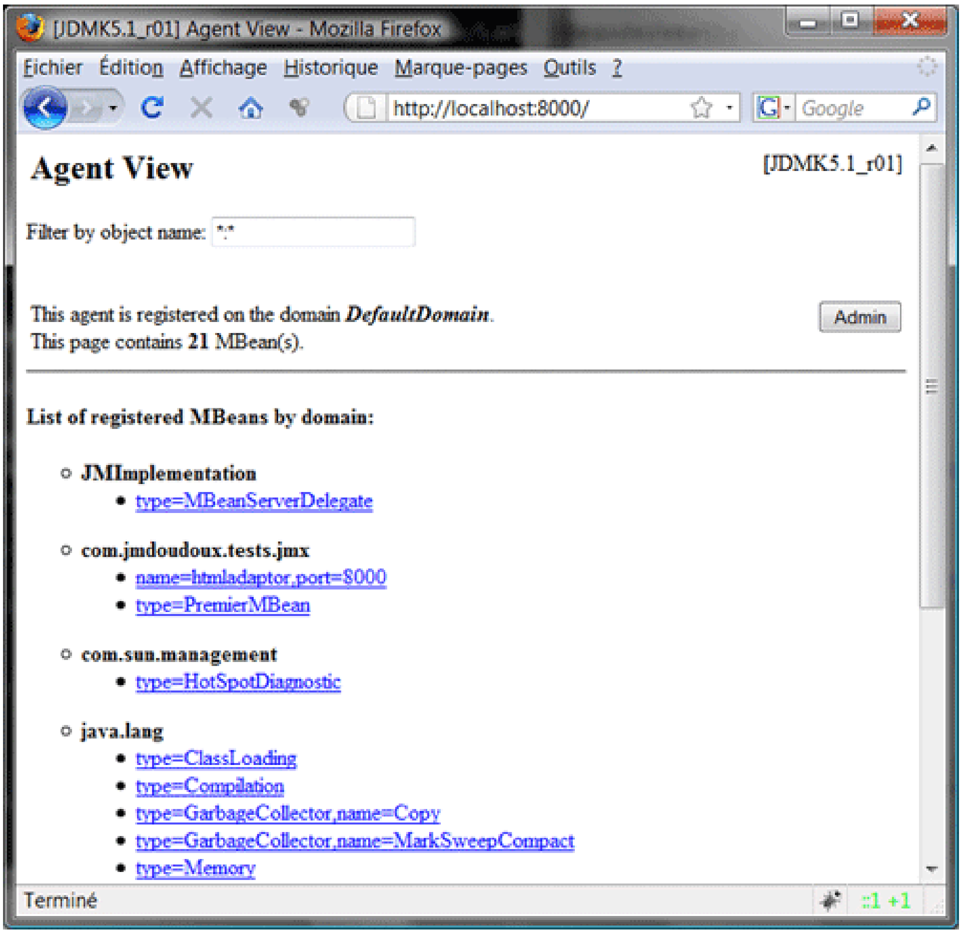
```
package fr.jmdoudoux.dej.jmx;

import java.lang.management.ManagementFactory;

import javax.management.InstanceAlreadyExistsException;
import javax.management.MBeanRegistrationException;
import javax.management.MBeanServer;
import javax.management.MalformedObjectNameException;
import javax.management.NotCompliantMBeanException;
import javax.management.ObjectName;

import com.sun.jdmk.comm.HtmlAdaptorServer;

public class LancerAgentAvecHTMLAdaptateur {
  static final int PORT_ADAPTATEUR = 8000;

  public static void main(String[] args) {

    System.out.println("Lancement de l'agent JMX");

    MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();

    ObjectName name = null;
    ObjectName adapterName = null;

    try {
      System.out.println("Instanciation et enregistrement du MBean");

      name = new ObjectName("fr.jmdoudoux.dej.jmx:type=PremierMBean");

      Premier mbean = new Premier();

      mbs.registerMBean(mbean, name);

      // Creation et demarrage de l'adaptateur de protocole HTML
      HtmlAdaptorServer adapter = new HtmlAdaptorServer();
      adapterName = new ObjectName(
          "fr.jmdoudoux.dej.jmx:name=htmladaptor,port=" + PORT_ADAPTATEUR);
      adapter.setPort(PORT_ADAPTATEUR);
      mbs.registerMBean(adapter, adapterName);
      adapter.start();
      System.out
          .println("Lancement de l'adaptateur de protocole HTML sur le port "
              + PORT_ADAPTATEUR);

      int i = 0;
      System.out.println("Incrementation de la valeur du MBean ...");
      while (i < 600) {

        mbean.setValeur(mbean.getValeur() + 1);
        Thread.sleep(1000);
        i++;
      }

      System.out.println("Arret de l'adaptateur de protocole HTML ");
      adapter.stop();

      System.out.println("Arret de l'agent JMX");

    } catch (MalformedObjectNameException e) {
      e.printStackTrace();
    } catch (NullPointerException e) {
      e.printStackTrace();
    } catch (InstanceAlreadyExistsException e) {
      e.printStackTrace();
    } catch (MBeanRegistrationException e) {
      e.printStackTrace();
    } catch (NotCompliantMBeanException e) {
      e.printStackTrace();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

Après compilation des classes, il faut exécuter l'agent JMX

Exemple :

```
C:\Users\Jean Michel\workspace\TestJMX\bin>java -cp .;jmxtools.jar fr.jmdoudoux
.dej.jmx/LancerAgentAvecHTMLAdaptateur
Lancement de l'agent JMX
Instanciation et enregistrement du MBean
Lancement de l'adaptateur de protocole HTML sur le port 8000
Incrementation de la valeur du MBean ...
Arret de l'adaptateur de protocole HTML
Arret de l'agent JMX
```

Il faut ouvrir un navigateur avec l'url http://localhost:8000.


[JDMK5.1_r01] Agent View - Mozilla Firefox
Fichier Édition Affichage Historique Marque-pages Outils ?
http://localhost:8000/
Google
Admin
Terminé

Agent View

[JDMK5.1_r01]

Filter by object name: *:*

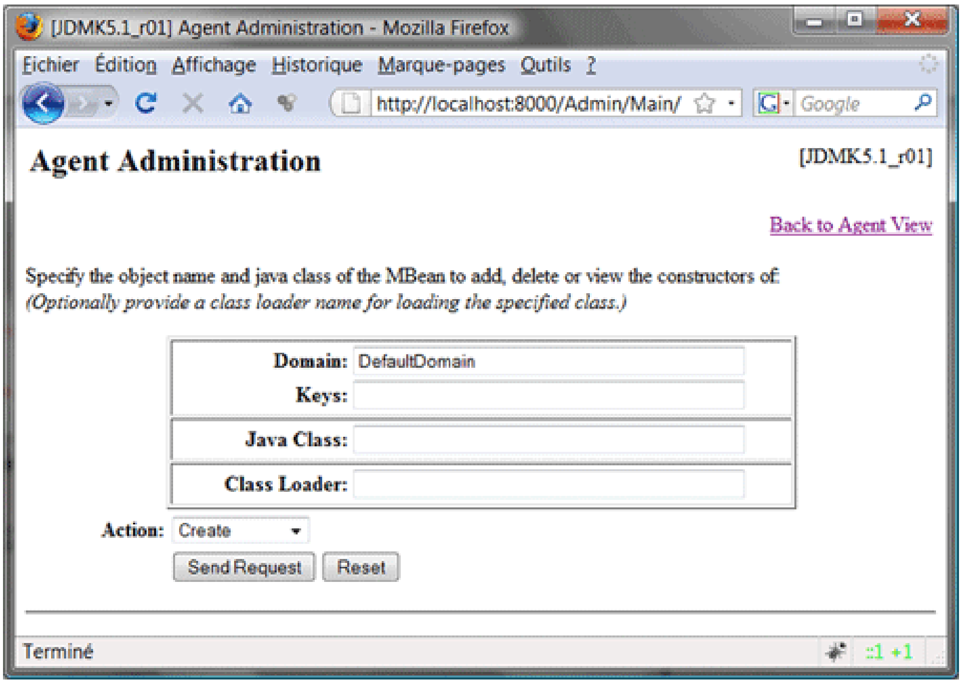
This agent is registered on the domain ***DefaultDomain***.
This page contains **21** MBean(s).

List of registered MBeans by domain:

- **JMImplementation**
  - type=MBeanServerDelegate
- **com.jmdoudoux.tests.jmx**
  - name=htmladaptor,port=8000
  - type=PremierMBean
- **com.sun.management**
  - type=HotSpotDiagnostic
- **java.lang**
  - type=ClassLoading
  - type=Compilation
  - type=GarbageCollector,name=Copy
  - type=GarbageCollector,name=MarkSweepCompact
  - type=Memory

La première page affichée par l'adaptateur affiche les MBeans enregistrés pour le domaine par défaut dans le serveur de MBeans de l'agent JMX.

Le bouton Admin affiche une page qui permet d'enregistrer un nouvel MBean dans le serveur.


[JDMK5.1_r01] Agent Administration - Mozilla Firefox
Fichier Édition Affichage Historique Marque-pages Outils ?
http://localhost:8000/Admin/Main/
Google
Create
Send Request
Reset
Terminé

Agent Administration [JDMK5.1_r01]

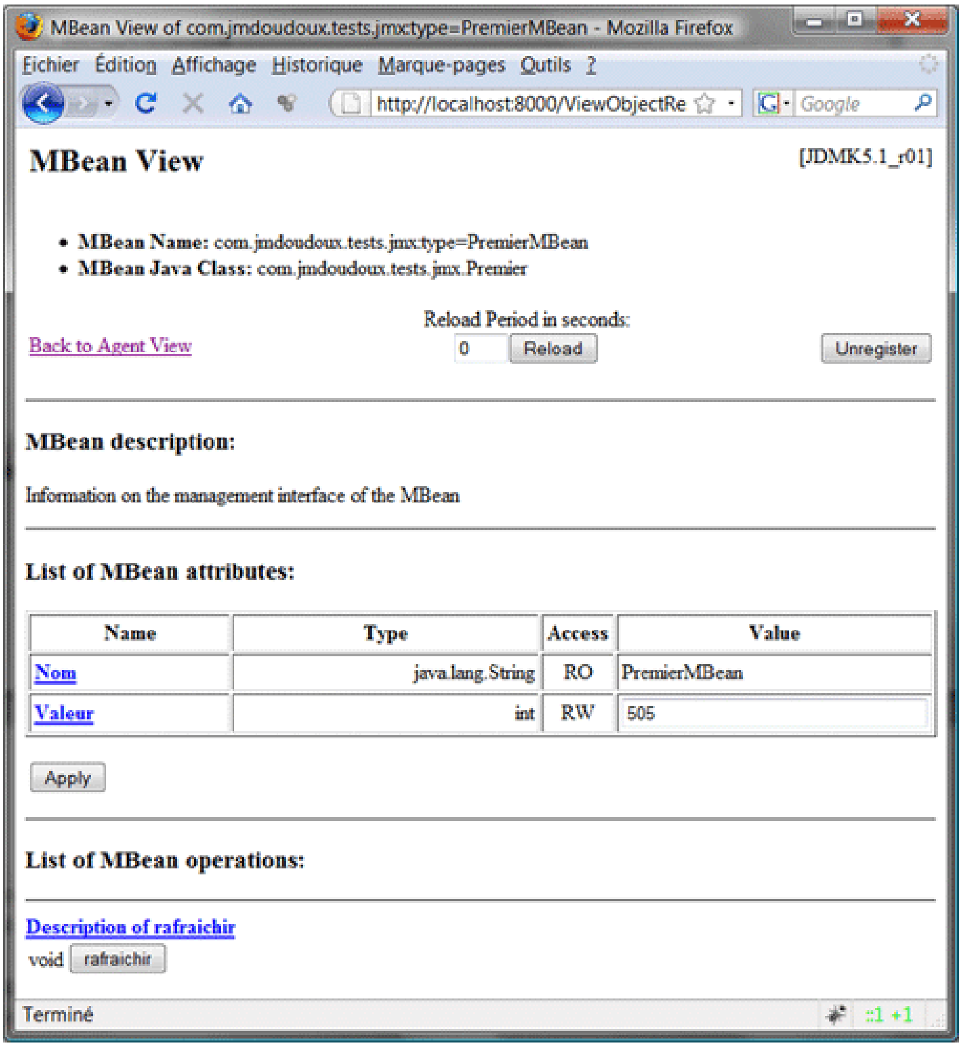
Back to Agent View

Specify the object name and java class of the MBean to add, delete or view the constructors of:
(Optionally provide a class loader name for loading the specified class.)

Domain:	DefaultDomain
Keys:	
Java Class:	
Class Loader:	

Action: Create

L'adaptateur lui-même est un MBean ce qui explique qu'il apparaît avec le MBean PremierMBean. En cliquant sur le lien de ce dernier, l'adaptateur affiche une page avec les propriétés et les opérations du MBean


MBean View of com.jmdoudoux.tests.jmx:type=PremierMBean - Mozilla Firefox
Fichier Édition Affichage Historique Marque-pages Outils ?
http://localhost:8000/ViewObjectRe
Google
Reload
Unregister
Apply
rafraichir
Terminé

MBean View [JDMK5.1_r01]

- **MBean Name:** com.jmdoudoux.tests.jmx:type=PremierMBean
- **MBean Java Class:** com.jmdoudoux.tests.jmx.Premier

Back to Agent View

Reload Period in seconds: 0

MBean description:

Information on the management interface of the MBean

List of MBean attributes:

Name	Type	Access	Value
Nom	java.lang.String	RO	PremierMBean
Valeur	int	RW	505

List of MBean operations:

Description of rafraichir

void rafraichir

Cette page affiche les propriétés, permet de modifier celles qui possèdent un setter et permet l'invocation des méthodes définies dans l'interface du MBean.

32.8.6. L'invocation d'un MBean par un proxy

Le classe MBeanServerConnection propose plusieurs méthodes pour interagir avec un MBean notamment :

- Object getAttribut() pour obtenir la valeur d'un attribut
- void setAttribut(ObjectName, Attribute) pour modifier la valeur d'un attribut
- Object invoke() pour invoquer une opération

Exemple :

```
package fr.jmdoudoux.dej.jmx;

import java.io.IOException;
import java.net.MalformedURLException;

import javax.management.AttributeNotFoundException;
import javax.management.InstanceNotFoundException;
import javax.management.MBeanException;
import javax.management.MBeanServerConnection;
import javax.management.MalformedObjectNameException;
import javax.management.ObjectName;
import javax.management.ReflectionException;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;

public class ClientJMXAvecRMI {
  public static void main(String[] args) {
    MBeanServerConnection mbsc = null;
    JMXConnector connecteur = null;

    ObjectName name = null;
    try {
      name = new ObjectName("fr.jmdoudoux.dej.jmx:type=PremierMBean");

      JMXServiceURL url = new JMXServiceURL(
          "service:jmx:rmi:///jndi/rmi://localhost:9000/server");

      connecteur = JMXConnectorFactory.connect(url, null);

      mbsc = connecteur.getMBeanServerConnection();

      System.out.println("valeur = " + mbsc.getAttribute(name, "Valeur"));
      mbsc.invoke(name, "rafraichir", null, null);

    } catch (MalformedObjectNameException e) {
      e.printStackTrace();
    } catch (NullPointerException e) {
      e.printStackTrace();
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    } catch (AttributeNotFoundException e) {
      e.printStackTrace();
    } catch (InstanceNotFoundException e) {
      e.printStackTrace();
    } catch (MBeanException e) {
      e.printStackTrace();
    } catch (ReflectionException e) {
      e.printStackTrace();
    } finally {
      if (mbsc != null) {
        try {
          connecteur.close();
        } catch (IOException e) {
          e.printStackTrace();
        }
      }
    }
  }
}
```

L'utilisation de ces méthodes peut être source de tracas car certains problèmes ne peuvent être détectés qu'à l'exécution, par exemple, si le nom fourni de la méthode à invoquer est erroné.

La classe MBeanServerInvocationHandler permet de créer un proxy qui va interagir avec un MBean du serveur de MBeans. Cette classe va utiliser l'interface du MBean pour générer dynamiquement une classe de type proxy permettant d'interagir avec le MBean.

La méthode statique newProxyInstance() permet de créer un proxy pour invoquer un MBean. Elle attend en paramètres l'instance qui encapsule la connexion vers le serveur de MBeans, le nom identifiant le MBean, le type de l'interface du MBean et un booléen qui permet de préciser si le proxy doit implémenter l'interface NotificationEmitter permettant d'abonner un listener aux notifications du MBean.

L'exemple ci-dessous est identique à l'exemple précédent.

Exemple :

```
package fr.jmdoudoux.dej.jmx;

import java.io.IOException;
import java.net.MalformedURLException;

import javax.management.MBeanServerConnection;
import javax.management.MBeanServerInvocationHandler;
import javax.management.MalformedObjectNameException;
import javax.management.ObjectName;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;

public class ClientJMXAvecRMI {
  public static void main(String[] args) {
    MBeanServerConnection mbsc = null;
    JMXConnector connecteur = null;

    ObjectName name = null;
    try {
      name = new ObjectName("fr.jmdoudoux.dej.jmx:type=PremierMBean");

      JMXServiceURL url = new JMXServiceURL(
          "service:jmx:rmi:///jndi/rmi://localhost:9000/server");

      connecteur = JMXConnectorFactory.connect(url, null);

      mbsc = connecteur.getMBeanServerConnection();

      PremierMBean mbean = (PremierMBean) MBeanServerInvocationHandler
          .newProxyInstance(mbsc, name, PremierMBean.class, false);
      int valeur = mbean.getValeur();
      System.out.println("valeur = " + valeur);
      mbean.rafraichir();

    } catch (MalformedObjectNameException e) {
      e.printStackTrace();
    } catch (NullPointerException e) {
      e.printStackTrace();
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      if (mbsc != null) {
        try {
          connecteur.close();
        } catch (IOException e) {
          e.printStackTrace();
        }
      }
    }
  }
}
```

Le code de la création du proxy est un peu particulier mais son utilisation facilite l'écriture du code qui invoque les fonctionnalités du MBean puisque le proxy donne aux appels distants l'apparence d'appels locaux. Le code est ainsi plus facile à écrire et à lire.

32.8.7. La recherche et la découverte des agents JMX

L'API JMX Remote précise comment il est possible de rechercher et découvrir des agents JMX en utilisant des infrastructures et des API existantes (aucune nouvelle API n'est définie par les spécifications JMX).

Un client JMX se connecte à un agent JMX par un connecteur ou un adaptateur de protocole. Pour rechercher et découvrir un agent, un client peut utiliser trois possibilités optionnelles d'infrastructures :

- Service Location Protocol (SLP)
- la technologie Jini
- JNDI avec un annuaire

32.8.7.1. Par le Service Location Protocol (SLP)

Le Service Location Protocol (SLP) est un framework qui permet de rechercher, découvrir et connaître la configuration de services au travers du réseau.

L'agent JMX doit enregistrer chacun de ses connecteurs auprès du registre de SLP en lui fournissant son adresse et des attributs obligatoires et éventuellement optionnels pour qualifier le connecteur.

Le client JMX interroge le registre de SLP pour obtenir les adresses éventuellement en utilisant des filtres sur les attributs pour limiter la recherche. Le client peut alors se connecter en utilisant les adresses obtenues.

32.8.7.2. Par la technologie Jini

La technologie Jini offre une architecture logicielle pour créer et déployer des services au travers du réseau. Jini offre bien sûr un registre qui contient les services.

L'agent JMX doit enregistrer chacun de ses connecteurs auprès du registre de Jini et lui fournissant un objet de type stub et des attributs obligatoires et éventuellement optionnels pour qualifier le connecteur.

Le client JMX interroge le registre de SLP pour obtenir les stubs éventuellement en utilisant des filtres sur les attributs pour limiter la recherche. Le client peut alors se connecter en utilisant les stubs obtenus.

32.8.7.3. Par un annuaire et la technologie JNDI

JNDI est une API standard qui permet d'interagir avec différents services de nommage ou annuaires.

L'agent JMX doit enregistrer chacun de ses connecteurs auprès du registre de l'annuaire en lui fournissant son adresse et des attributs obligatoires et éventuellement optionnels pour qualifier le connecteur.

Le client JMX interroge le registre de l'annuaire pour obtenir les adresses éventuellement en utilisant des filtres sur les attributs pour limiter la recherche. Le client peut alors se connecter en utilisant les adresses obtenues.

32.9. Les notifications

Tous les types de MBeans peuvent émettre des notifications pour informer de certains événements survenus sur la ressource gérée par le MBean ou sur le MBean lui-même.

Les notifications peuvent avoir plusieurs utilités : avertir du changement d'une valeur ou de l'état d'un attribut, signaler un événement ou un problème, ...

32.9.1. L'interface NotificationBroadcaster

Pour émettre des notifications, un MBean peut implémenter l'interface NotificationBroadcaster mais son utilisation n'est plus recommandée. Il est préférable d'utiliser son interface fille NotificationEmitter.

L'interface NotificationBroadcaster définit trois méthodes qui permettent l'abonnement d'un listener, d'obtenir des informations sur les notifications et le désabonnement d'un listener.

Méthode	Rôle
void addNotificationListener(NotificationListener, NotificationFilter, Object)	Abonner le listener fourni en paramètre
MBeanNotificationInfo[] getNotificationInfo()	Renvoyer un tableau contenant des informations sur les notifications pouvant être émises
void removeNotificationListener(NotificationListener)	Désabonner le listener fourni en paramètre

32.9.2. L'interface NotificationEmitter

Pour émettre des notifications, un MBean doit de préférence implémenter l'interface NotificationEmitter. Celle-ci hérite de l'interface NotificationBroadcaster.

L'interface NotificationEmitter ne définit qu'une seule méthode supplémentaire, qui est une surcharge de la méthode removeNotificationListener() :

Méthode	Rôle
void removeNotificationListener(NotificationListener, NotificationFilter, Object)	Désabonner le listener fourni en paramètre

32.9.3. La classe NotificationBroadcasterSupport

Le plus simple pour implémenter les notifications dans un MBean est de le faire hériter de la classe NotificationBroadcasterSupport en plus d'implémenter l'interface du MBean. La classe NotificationBroadcasterSupport implémente l'interface NotificationEmitter et propose la méthode sendNotification() qui permet l'émission de la notification qui lui est fournie en paramètre. Celle-ci est envoyée à chaque listener abonné.

32.9.4. La classe javax.management.Notification

La classe Notification encapsule une notification émise par un MBean suite à un événement.

Une notification est donc une instance de la classe javax.management.Notification ou d'une de ses sous-classes : AttributeChangeNotification, JMXConnectionNotification, MBeanServerNotification, MonitorNotification, RelationNotification ou TimerNotification

Chaque notification possède :

- une source (Source) : c'est le nom de l'objet (ObjectName) du MBean qui émet la notification ou l'instance du MBean
- un type (Type) : c'est une chaîne de caractères dont chaque mot est séparé par un caractère point
- un numéro de séquence (SequenceNumber) : sa valeur est arbitraire mais il est préférable de l'incrémenter à chaque émission
- un timestamp (TimeStamp): c'est la date/heure d'émission de la notification
- un message (Message) : une chaîne de caractères qui fournit une description de la notification
- des données (UserData) : collection de données de type HashTable

Les sous-classes de la classe Notification peuvent avoir des attributs supplémentaires.

Lorsque le serveur de MBeans envoie la notification au client JMX, il transforme la source en son ObjectName si l'objet source est l'instance du MBean. En effet, le client JMX connait l'ObjectName mais ne possède pas la référence sur l'instance du MBean.

Chaque notification doit obligatoirement avoir un numéro de séquence.

Exemple :

```
    Notification notif = new AttributeChangeNotification(this,
        numeroSequence, System.currentTimeMillis(),
        "Modification de la valeur", "Valeur", "int", this.valeur, valeur);

    this.valeur = valeur;

    sendNotification(notif);
```

Pour émettre une notification, il faut instancier un objet de type Notification et appeler la méthode sendNotification() en lui passant en paramètre l'instance créée.

Il faut redéfinir la méthode getNotificationInfo() qui renvoie un tableau de type MBeanNotificationInfo contenant des données sur les notifications pouvant être émises.

Un objet de type MBeanNotificationInfo possède :

- un ou plusieurs types
- un nom
- une description

Le MBean devrait permettre de fournir directement à un client les données incluses dans une notification.

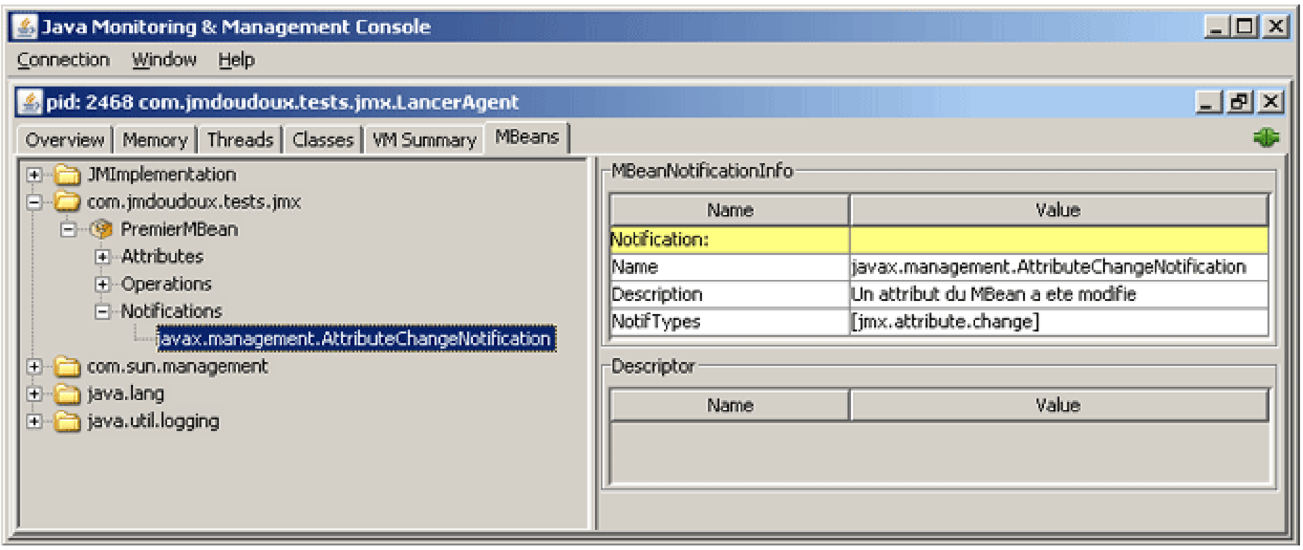
32.9.5. Un exemple de notifications

Cette section contient un exemple complet de mise en oeuvre de notifications par un MBean.

Exemple :

```
package fr.jmdoudoux.dej.jmx;

import javax.management.AttributeChangeNotification;
import javax.management.MBeanNotificationInfo;
import javax.management.Notification;
import javax.management.NotificationBroadcasterSupport;

public class Premier extends NotificationBroadcasterSupport implements
    PremierMBean {

  private static String nom            = "PremierMBean";

  private int           valeur         = 100;

  private static long   numeroSequence = 0l;

  public String getNom() {
    return nom;
  }

  public int getValeur() {
    return valeur;
  }

  public synchronized void setValeur(int valeur) {
    numeroSequence++;
    Notification notif = new AttributeChangeNotification(this,
        numeroSequence, System.currentTimeMillis(),
        "Modification de la valeur", "Valeur", "int", this.valeur, valeur);

    this.valeur = valeur;

    sendNotification(notif);
  }

  public void rafraichir() {
    System.out.println("Rafraichir les donnees");
  }

  @Override
  public MBeanNotificationInfo[] getNotificationInfo() {
      String[] types = new String[] {
          AttributeChangeNotification.ATTRIBUTE_CHANGE
      };
      String name = AttributeChangeNotification.class.getName();
      String description = "Un attribut du MBean a ete modifie";
      MBeanNotificationInfo info =
          new MBeanNotificationInfo(types, name, description);
      return new MBeanNotificationInfo[] {info};
  }
}
```

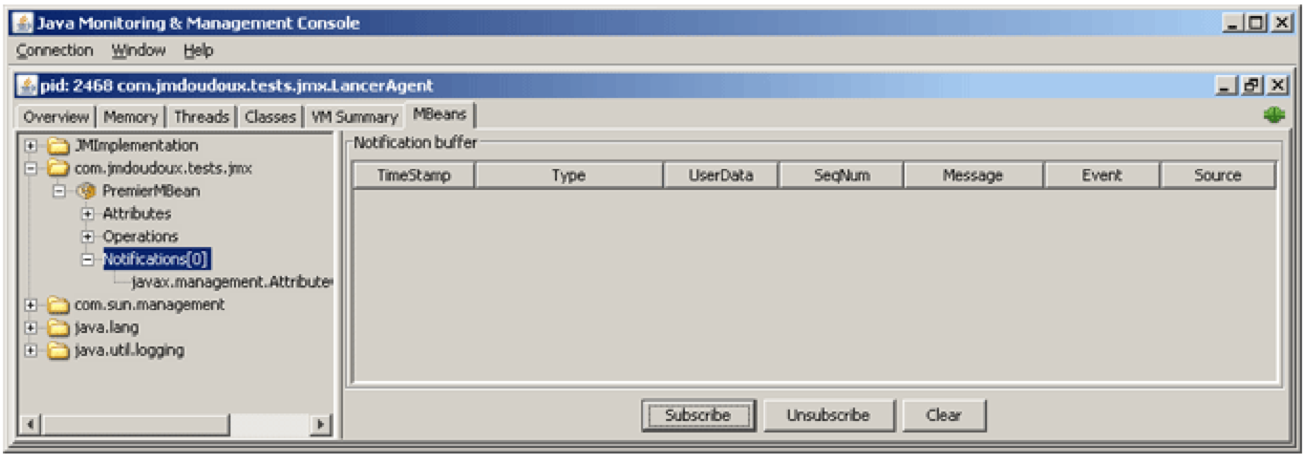
Il faut lancer l'agent et utiliser l'outil JConsole pour visualiser les notifications.



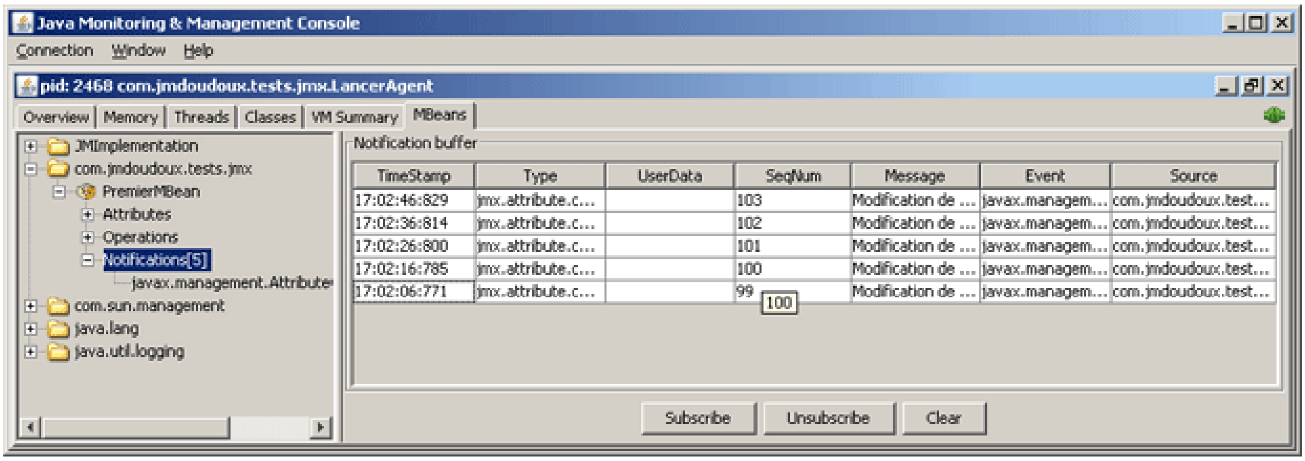
Name	Value
Notification:	
Name	javax.management.AttributeChangeNotification
Description	Un attribut du MBean a ete modifie
NotifTypes	[jmx.attribute.change]

Name	Value

Il suffit de sélectionner Notifications et de cliquer sur le bouton «Subscribe»


Java Monitoring & Management Console
Connection Window Help
pid: 2468 com.jmdoudoux.tests.jmx.LancerAgent
Overview Memory Threads Classes VM Summary MBeans
JMImplementation
com.jmdoudoux.tests.jmx
PremierMBean
Attributes
Operations
Notifications[0]
javax.management.Attribute
com.sun.management
java.lang
java.util.logging
Notification buffer
TimeStamp Type UserData SeqNum Message Event Source
Subscribe Unsubscribe Clear

Les notifications sont affichées au fur et à mesure de leur arrivée :


Java Monitoring & Management Console
Connection Window Help
pid: 2468 com.jmdoudoux.tests.jmx.LancerAgent
Overview Memory Threads Classes VM Summary MBeans
JMImplementation
com.jmdoudoux.tests.jmx
PremierMBean
Attributes
Operations
Notifications[5]
javax.management.Attribute
com.sun.management
java.lang
java.util.logging
Notification buffer
100
Subscribe Unsubscribe Clear

TimeStamp	Type	UserData	SeqNum	Message	Event	Source
17:02:46:829	jmx.attribute.c...		103	Modification de ...	javax.managem...	com.jmdoudoux.test...
17:02:36:814	jmx.attribute.c...		102	Modification de ...	javax.managem...	com.jmdoudoux.test...
17:02:26:800	jmx.attribute.c...		101	Modification de ...	javax.managem...	com.jmdoudoux.test...
17:02:16:785	jmx.attribute.c...		100	Modification de ...	javax.managem...	com.jmdoudoux.test...
17:02:06:771	jmx.attribute.c...		99	Modification de ...	javax.managem...	com.jmdoudoux.test...

32.9.6. L'abonnement aux notifications par un client JMX

Un client JMX distant peut enregistrer un listener de type NotificationListener pour lui permettre d'être informé des changements du statut de la connexion utilisée pour les notifications. L'enregistrement se fait en utilisant la méthode addConnectionNotificationListener() sur une instance de l'interface JMXConnector.

Une classe d'un client JMX s'abonne aux notifications en enregistrant un listener de type NotificationListener grâce à la méthode addNotificationListerner() de l'instance de type MBeanServerConnection. Plusieurs listeners peuvent s'abonner à une même notification mais un même listener ne peut s'abonner qu'une seule fois à une notification.

L'interface NotificationListener ne définit qu'une seule méthode :
public void handleNotification(Notification notif, Object handback)

Cette méthode de type callback sera invoquée pour chaque listener abonné lors de l'émission d'une notification.

Le MBean est défini grâce à une interface.

Exemple (code Java 5.0) :

```
package test.jmx;

import javax.management.MBeanNotificationInfo;

public interface PremierMBean {
  public abstract String getNom();
  public abstract int getValeur();
  public abstract void setValeur(final int valeur);
  public abstract void rafraichir();
  public abstract MBeanNotificationInfo[] getNotificationInfo();
}
```

L'implémentation du MBean hérite de la classe NotificationBroadcasterSupport qui propose des fonctionnalités pour permettre l'émission de notifications en invoquant la méthode sendNotification().

Exemple (code Java 5.0) :

```
package test.jmx;

import javax.management.AttributeChangeNotification;
import javax.management.MBeanNotificationInfo;
import javax.management.Notification;
import javax.management.NotificationBroadcasterSupport;

public class Premier extends NotificationBroadcasterSupport implements PremierMBean {
  private static String nom = "PremierMBean";
  private int valeur = 100;
  private static long numeroSequence = 0l;

  public String getNom() {
    return nom;
  }

  public int getValeur() {
    return valeur;
  }

  public synchronized void setValeur(final int valeur) {
    numeroSequence++;
    final Notification notif = new AttributeChangeNotification(this, numeroSequence,
      System.currentTimeMillis(), "Modification de la valeur",
      "Valeur", "int", this.valeur, valeur);
    this.valeur = valeur;
    sendNotification(notif);
    }

  public void rafraichir() {
    System.out.println("Rafraichir les donnees");
  }

  @Override
  public MBeanNotificationInfo[] getNotificationInfo() {
    final String[] types = new String[] { AttributeChangeNotification.ATTRIBUTE_CHANGE };
    final String name = AttributeChangeNotification.class.getName();
    final String description = "Un attribut du MBean a ete modifie";
    final MBeanNotificationInfo info = new MBeanNotificationInfo(types, name, description);
    return new MBeanNotificationInfo[] { info };
  }
}
```

Dans l'exemple ci-dessus, la notification est de type AttributChangeNotification fournie en standard par l'API JMX.

La partie serveur effectue plusieurs traitements :

- Instanciation du MBean et enregistrement dans le serveur de MBeans par défaut de la JVM
- Toutes les secondes, incrémentation de la valeur du MBean pour émettre une notification

Exemple (code Java 5.0) :

```
package test.jmx;

  import java.lang.management.ManagementFactory;
  import javax.management.MBeanServer;
  import javax.management.ObjectName;

  public class LanceServeur {
    public static void main(final String[] args) {
      System.out.println("Lancement de l'agent JMX");
      final MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
      ObjectName name = null;
      try {
        System.out.println("Instanciation et enregistrement du Mbean");
        name = new ObjectName("fr.jmdoudoux.dej.jmx:type=PremierMBean");
        final Premier mbean = new Premier();
        mbs.registerMBean(mbean, name);
        int i = 0;
        System.out.println("Incrementation de la valeur du MBean ...");

        while (i < 6000) {
          System.out.println("valeur = " + (mbean.getValeur() + 1));
          mbean.setValeur(mbean.getValeur() + 1);
          Thread.sleep(1000);
          i++;
        }
        System.out.println("Arret de l'agent JMX");
      } catch (final Exception e) {
        e.printStackTrace();
      }
    }
  }
```

La partie cliente instancie et démarre le listener qui va traiter les notifications et attend.

Exemple (code Java 5.0) :

```
package test.jmx;

public class LanceClient {
  public static void main(final String[] args) {
    ClientListener listener = null;
    try {
      listener = new ClientListener();
      listener.connecter();
      while (true) {
        ;
      }
    } catch (final Exception e) {
      e.printStackTrace();
    } finally {
      if (listener != null) {
        listener.deconnecter();
      }
    }
  }
}
```

L'implémentation du listener se charge de gérer la connexion au serveur de MBeans et de traiter les notifications qui sont reçues.

Il implémente l'interface NotificationListener et implémente donc la méthode handleNotification().

Cette implémentation affiche simplement les notifications reçues et le détail de celles-ci si la notification est de type AttributeChangeNotification.

Exemple (code Java 5.0) :

```
package test.jmx;

import java.io.IOException;
import javax.management.AttributeChangeNotification;
import javax.management.MBeanServerConnection;
import javax.management.MBeanServerInvocationHandler;
import javax.management.Notification;
import javax.management.NotificationListener;
import javax.management.ObjectName;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;

public class ClientListener implements NotificationListener {
  private String port = "9998";
  private String host = "localhost";
  private JMXConnector connector;
  private MBeanServerConnection mbsc;
  private ObjectName name = null;

  public void handleNotification(final Notification notification, final Object handback) {
    System.out.println("\nNotification recue:");
    System.out.println("\tClassName: " + notification.getClass().getName());
    System.out.println("\tSource: " + notification.getSource());
    System.out.println("\tType: " + notification.getType());
    System.out.println("\tMessage: " + notification.getMessage());

    if (notification instanceof AttributeChangeNotification) {
      final AttributeChangeNotification acn = (AttributeChangeNotification) notification;
      System.out.println("\tAttributeName: " + acn.getAttributeName());
      System.out.println("\tAttributeType: " + acn.getAttributeType());
      System.out.println("\tNewValue: " + acn.getNewValue());
      System.out.println("\tOldValue: " + acn.getOldValue());
    }
  }

  public void deconnecter() {
    try {
      System.out.println("Desabonner le NotificationListener");
      mbsc.removeNotificationListener(name, this);
    }
    catch (final Exception e) {
      e.printStackTrace();
    }
    try {
      System.out.println("Deconnexion du server JMX");
      connector.close();
    }
    catch (final IOException e) {
      e.printStackTrace();
    }
  }
  public void connecter() throws Exception {
    final JMXServiceURL address = new JMXServiceURL("service:jmx:rmi:///jndi/rmi://"
      + host + ":" + port + "/jmxrmi");
    name = new ObjectName("fr.jmdoudoux.dej.jmx:type=PremierMBean");

    System.out.println("Connexion au server JMX");
    connector = JMXConnectorFactory.connect(address, null);

    System.out.println("Connection JMX etablie a la JVM " + host + " sur le port " + port);
    final PremierMBean mbean = (PremierMBean) MBeanServerInvocationHandler
      .newProxyInstance(mbsc, name, PremierMBean.class, false);
    final int valeur = mbean.getValeur();

    System.out.println("Valeur courante du mbean = " + valeur);
    mbean.rafraichir();
    System.out.println("Abonnement du NotificationListener pour " + name.toString());
    mbsc.addNotificationListener(name, this, null, null);
  }

  public String getHost() {
    return host;
  }

  public void setHost(final String host) {
    this.host = host;
  }

  public String getPort() {
    return port;
  }

  public void setPort(final String port) {
    this.port = port;
  }
}
```

Il est possible de fournir un objet de type NotificationFilter dont l'implémentation encapsule un filtre sur les notifications que le client souhaite recevoir. L'interface NotificationFilter ne définit qu'une seule méthode isNotificationEnabled(Notification) qui renvoie un booléen.

JMX propose trois filtres en standard :

- AttributeChangeNotificationFilter : filtre sur les notifications de type AttributeChangeNotification
- MBeanServerNotificationListener : filtre sur les notifications de type MBeanServerNotification (hérite de la classe NotificationFilterSupport)
- NotificationFilterSupport : filtre sur l'attribut type des MBeans qui émettent les notifications

32.10. Les Dynamic MBeans

Un MBean standard expose ses fonctionnalités au travers d'une interface statique. Ainsi une propriété est définie à l'aide d'un getter et/ou d'un setter. Les MBeans standards ne permettent pas de répondre à tous les besoins notamment lorsque l'interface du MBean ne peut pas être définie de façon statique.

Parfois l'interface ne peut être définie que de façon dynamique : c'est par exemple le cas si les attributs sont issus d'une collection de type Map ou de la lecture d'une ressource externe comme un fichier.

Les Dynamic MBeans sont donc utiles si le nombre de fonctionnalités susceptibles d'être invoquées varie au cours des exécutions.

Le développement d'un MBean dynamic est complexe et nécessite que le MBean implémente l'interface DynamicMBean dont les méthodes retournent des informations statiques. Il est nécessaire d'utiliser et d'assembler des structures d'informations qui sont parfois redondantes. Ces informations sont encapsulées dans plusieurs classes du package javax.management : MBeanInfo, MBeanAttributeInfo, MBeanOperationInfo, ...

Les Dynamic MBeans ne possèdent pas un getter et un setter pour chaque attribut : ils proposent à la place des méthodes génériques pour obtenir et mettre à jour la valeur d'un attribut et invoquer les méthodes du MBean dynamiquement à partir de son nom.

Les fonctionnalités exposées par le MBean sont contenues dans un objet de type MBeanInfo retourné par la méthode getMBeanInfo() de l'interface DynamicMBean. Ces fonctionnalités concernent les attributs, les opérations et les notifications : les informations qu'il est cependant possible d'obtenir sur le MBean et ses méthodes sont plus riches que celles d'un MBean standard.

Ainsi l'interface du MBean est statique mais son implémentation expose dynamiquement les fonctionnalités du MBean.

L'exploitation des MBeans standard et dynamic ne fait aucune différence pour les clients JMX qui utilisent l'un et l'autre de la même façon. La différence se fait dans la façon dont ils exposent chacun leurs fonctionnalités :

- une interface statique pour les MBeans standard
- une description dynamique pour les MBeans dynamiques

Un agent JMX n'a pas besoin de faire de l'introspection sur un Dynamic MBean pour découvrir ses fonctionnalités, il lui suffit de lire ses métadonnées obtenues grâce à la méthode getMBeanInfo().

32.10.1. L'interface DynamicMBean

L'interface javax.management.DynamicMBean propose des méthodes pour permettre à un MBean Dynamic d'exposer dynamiquement ses fonctionnalités. Celles-ci sont exposées au moyen de descripteurs qui sont fournis grâce à ses méthodes.

Méthode	Rôle
MBeanInfo getMBeanInfo()	Renvoyer un objet de type MbeanInfo qui encapsule les fonctionnalités exposées par le MBean
Object getAttribute(String attribute)	Permettre d'obtenir la valeur d'un attribut à partir de son nom
void setAttribute(Attribute attribute)	Permettre de mettre à jour la valeur d'un attribut
AttributeList getAttributes(String[] attributes)	Permettre d'obtenir la valeur d'un ensemble d'attributs à partir de leurs noms
AttributeList setAttributes(AttributeList attributes)	Permettre de mettre à jour la valeur d'un ensemble d'attributs
Object invoke(String actionName, Object params[], String signature[])	Permettre d'invoquer une opération

La classe MBeanInfo encapsule les fonctionnalités exposées par le MBean : une collection des attributs avec leur type et leur nom, une collection des constructeurs, une collection des opérations invocables avec leurs paramètres, une collection des notifications et quelques informations.

32.10.2. Les métadonnées d'un Dynamic MBean

La méthode getMBeanInfo() renvoie un objet de type MBeanInfo qui encapsule les métadonnées des fonctionnalités du MBean.

32.10.2.1. La classe MBeanInfo

La classe javax.management.MBeanInfo est une classe qui encapsule les métadonnées des fonctionnalités du MBean. Chaque instance est immuable.

Cette classe propose plusieurs méthodes pour obtenir les métadonnées selon le type de leurs fonctionnalités :

Méthode	Rôle
MBeanAttributeInfo[] getAttributes()	Renvoyer un tableau de type MBeanAttributeInfo qui contient les métadonnées des attributs
MBeanConstructorInfo[] getConstructors()	Renvoyer un tableau de type MBeanConstructorInfo qui contient les métadonnées des constructeurs
String getDescription()	Renvoyer une description du MBean
MBeanNotificationInfo[] getNotifications()	Renvoyer un tableau de type MBeanNotificationInfo qui contient les métadonnées des notifications
MBeanOperationInfo[] getOperations()	Renvoyer un tableau de type MBeanOperationInfo qui contient les métadonnées des opérations

Elle possède un seul constructeur :

MBeanInfo(String className, String description, MBeanAttributeInfo[] attributes, MBeanConstructorInfo[] constructors, MBeanOperationInfo[] operations, MBeanNotificationInfo[] notifications)

32.10.2.2. La classe MBeanFeatureInfo

La classe javax.management.MBeanFeatureInfo est la classe mère des classes qui encapsulent les métadonnées d'une fonctionnalité du MBean.

Cette classe possède deux propriétés :

Propriété	Rôle
Description	Description de la fonctionnalité
Name	Nom de la fonctionnalité

Elle ne propose que des getters sur ses propriétés.

32.10.2.3. La classe MBeanAttributeInfo

La classe javax.management.MBeanAttributeInfo encapsule les métadonnées d'un attribut du MBean. Elle hérite de la classe MBeanFeatureInfo.

Elle possède plusieurs propriétés :

Propriété	Rôle
String type	Type de l'attribut
boolean isReadable	Indique si l'attribut est lisible
boolean isWritable	Indique si l'attribut est modifiable
boolean isIs	Indique si le getter est de type isXXX pour les booléens

Chaque instance de cette classe est immuable : elle ne propose donc que des getters sur ses propriétés.

Elle possède deux constructeurs :

- MBeanAttributeInfo(String name, String description, java.lang.reflect.Method getter, java.lang.reflect.Method setter) : les informations sur l'attribut sont obtenues par instrospection sur le getter et le setter
- MBeanAttributeInfo(String name, String type, String description, boolean isReadable, boolean isWritable, boolean isIs)

32.10.2.4. La classe MBeanParameterInfo

La classe javax.management.MBeanParameterInfo encapsule les métadonnées d'un paramètre d'un constructeur ou d'une méthode du MBean. Elle hérite de la classe MBeanFeatureInfo.

Elle possède une propriété :

Propriété	Rôle
String type	Le type du paramètre

Chaque instance de cette classe est immuable : elle ne propose donc qu'un getter sur sa propriété.

Elle possède un seul constructeur :

MBeanParameterInfo(java.lang.String name, java.lang.String type, java.lang.String description)

32.10.2.5. La classe MBeanConstructorInfo

La classe javax.management.MBeanConstructorInfo encapsule les métadonnées d'un constructeur du MBean. Elle hérite de la classe MBeanFeatureInfo.

Elle possède une propriété :

Propriété	Rôle
MbeanParameterInfo[] signature	Renvoie un tableau des paramètres du constructeur

Chaque instance de cette classe est immuable : elle ne propose donc qu'un getter sur sa propriété.

Elle possède deux constructeurs :

- MBeanConstructorInfo(String description, java.lang.reflect.Constructor constructor) : les informations sur le constructeur sont obtenues par instrospection sur celui fourni en paramètre
- MBeanConstructorInfo(String name, String description, MBeanParameterInfo[] signature)

32.10.2.6. La classe MBeanOperationInfo

La classe javax.management.MBeanOperationInfo encapsule les métadonnées d'une méthode du MBean. Elle hérite de la classe MBeanFeatureInfo.

Elle possède plusieurs propriétés :

Propriété	Rôle
MbeanParameterInfo[] signature	Renvoyer un tableau des paramètres de la méthode
int impact	Préciser la nature de la méthode : les valeurs possibles sont INFO, ACTION, ACTION_INFO, UNKNOWN
String returnType	Renvoyer le type de la valeur de retour de la méthode

Chaque instance de cette classe est immuable : elle ne propose donc que des getters sur ses propriétés.

Elle possède deux constructeurs :

- MBeanOperationInfo(String description, java.lang.reflect.Method method) : les informations sur la méthode sont obtenues par instrospection sur celles fournies en paramètre
- MBeanOperationInfo(String name, String description, MBeanParameterInfo[] signature, String type, int impact)

La classe MbeanOperationInfo définit plusieurs constantes utilisables pour sa propriété impact :

Constante	Rôle
INFO	Précise que la méthode ne fait que lire l'état du MBean
ACTION	Précise que la méthode va modifier l'état du MBean
ACTION_INFO	Précise que la méthode lit et modifie l'état du MBean
UNKNOWN	Précise que la nature de la méthode est inconnue

32.10.2.7. La classe MBeanNotificationInfo

La classe javax.management.MBeanNotificationInfo encapsule les métadonnées d'une notification du MBean. Elle hérite de la classe MBeanFeatureInfo.

Elle possède une propriété :

Propriété	Rôle
String[] notifTypes	Renvoyer un tableau du libellé des types de la notification (ce n'est pas le nom des classes)

Chaque instance de cette classe est immuable : elle ne propose donc qu'un getter sur sa propriété.

Elle possède un seul constructeur :

MBeanNotificationInfo(java.lang.String[] notifTypes, java.lang.String name, java.lang.String description)

32.10.3. La définition d'un MBean Dynamic

Les Dynamic MBeans proposent des fonctionnalités plus avancées que les MBeans standard mais ils sont aussi plus complexes à mettre en oeuvre.

Un Dynamic MBean n'a pas besoin de respecter des conventions de nommage particulières ni de définir sa propre interface mais simplement d'implémenter l'interface DynamicMBean et d'avoir au moins un constructeur public.

Ce premier exemple est une implémentation en MBean Dynamic du MBean standard PremierMBean défini précédemment.

Exemple :

```
package fr.jmdoudoux.dej.jmx;

import java.util.Iterator;

import javax.management.Attribute;
import javax.management.AttributeList;
import javax.management.AttributeNotFoundException;
import javax.management.DynamicMBean;
import javax.management.InvalidAttributeValueException;
import javax.management.MBeanAttributeInfo;
import javax.management.MBeanConstructorInfo;
import javax.management.MBeanException;
import javax.management.MBeanInfo;
import javax.management.MBeanOperationInfo;
import javax.management.MBeanParameterInfo;
import javax.management.ReflectionException;

public class PremierDynamic implements DynamicMBean {

  private static String nom = "PremierDynamic";
  private int valeur = 100;

  public PremierDynamic() {

  }

  @Override
  public Object getAttribute(String attribute)
      throws AttributeNotFoundException, MBeanException, ReflectionException {
    Object resultat = null;

    if (attribute.equals("nom")) {
      resultat = getNom();
    } else {
      if (attribute.equals("valeur")) {
        resultat = getValeur();
      } else {
        throw new AttributeNotFoundException(attribute);
      }
    }
    return resultat;

  }

  @Override
  public AttributeList getAttributes(String[] attributes) {
    AttributeList attributs = new AttributeList();
    attributs.add(new Attribute("nom", getNom()));
    attributs.add(new Attribute("valeur", getValeur()));
    return attributs;
  }

  @Override
  public MBeanInfo getMBeanInfo() {
    MBeanParameterInfo[] sansParamInfo = new MBeanParameterInfo[0];

    MBeanAttributeInfo attributs[] = new MBeanAttributeInfo[2];
    attributs[0] = new MBeanAttributeInfo("valeur", "int",
        "Valeur de l'instance", true, true, false);
    attributs[1] = new MBeanAttributeInfo("nom", "java.lang.String",
        "Nom de l'instance", true, false, false);

    MBeanConstructorInfo[] constructeurs = new MBeanConstructorInfo[1];
    constructeurs[0] = new MBeanConstructorInfo("PremierDynamic",
        "Constructeur par défaut de la classe", sansParamInfo);

    MBeanOperationInfo[] operations = new MBeanOperationInfo[1];
    operations[0] = new MBeanOperationInfo("rafraichir",
        "Rafraichir les données", sansParamInfo, void.class.getName(),
        MBeanOperationInfo.ACTION);

    return new MBeanInfo(getClass().getName(), "Mon premier MBean Dynamic",
        attributs, constructeurs, operations, null);

  }

  @Override
  public Object invoke(String actionName, Object[] params, String[] signature)
      throws MBeanException, ReflectionException {
    try {
      if (actionName.equals("rafraichir")) {
        rafraichir();
      }
      return null;
    } catch (Exception x) {
      throw new MBeanException(x);
    }

  }

  @Override
  public void setAttribute(Attribute attribute)
      throws AttributeNotFoundException, InvalidAttributeValueException,
      MBeanException, ReflectionException {

    String name = attribute.getName();
    try {
      if (name.equals("valeur")) {
        setValeur(((Integer) attribute.getValue()).intValue());
      } else {
        throw new AttributeNotFoundException(name);
      }

    } catch (ClassCastException cce) {
      throw new InvalidAttributeValueException(name);
    }
  }
  @Override
  @SuppressWarnings("unchecked")
  public AttributeList setAttributes(AttributeList attributes) {
    for (Iterator  i = attributes.iterator(); i.hasNext();) {
      Attribute attr = (Attribute) i.next();
      try {
        setAttribute(attr);
      } catch (AttributeNotFoundException e) {
        e.printStackTrace();
      } catch (InvalidAttributeValueException e) {
        e.printStackTrace();
      } catch (MBeanException e) {
        e.printStackTrace();
      } catch (ReflectionException e) {
        e.printStackTrace();
      }
    }
    return attributes;

  }

  private String getNom() {
    return nom;
  }

  private int getValeur() {
    return valeur;
  }

  private synchronized void setValeur(int valeur) {
    this.valeur = valeur;
  }

  private void rafraichir() {
    System.out.println("Rafraichir les donnees");

  }
}
```

L'intérêt de cet exemple est purement pédagogique car dans ce cas aucune fonctionnalité dynamique n'est utilisée mais il illustre bien la complexité d'écriture d'un MBean Dynamic par rapport à son équivalent sous la forme d'un MBean standard.

Les Dynamic MBeans peuvent tous fournir une description détaillée de leurs fonctionnalités ce qui peut les rendre plus facile à exploiter.

Le second exemple utilise une collection pour stocker ses attributs : cette collection pourrait par exemple être remplie en lisant un fichier de configuration. L'implémentation sous la forme d'un MBean standard est impossible car il n'est pas possible de connaître le contenu de la collection en dehors du contexte d'exécution.

Exemple :

```
package fr.jmdoudoux.dej.jmx;

import java.util.Hashtable;
import java.util.Iterator;

import javax.management.Attribute;
import javax.management.AttributeList;
import javax.management.AttributeNotFoundException;
import javax.management.DynamicMBean;
import javax.management.InvalidAttributeValueException;
import javax.management.MBeanAttributeInfo;
import javax.management.MBeanConstructorInfo;
import javax.management.MBeanException;
import javax.management.MBeanInfo;
import javax.management.MBeanOperationInfo;
import javax.management.MBeanParameterInfo;
import javax.management.ReflectionException;
```

```java
public class SecondDynamic implements DynamicMBean {

  private Hashtable<String, Object> attributs = new Hashtable<String, Object>();

  public SecondDynamic() {
    attributs.put("attrString1", "string");
    attributs.put("attrInt1", 0);
    attributs.put("attrString2", "string");
    attributs.put("valeur", 0);
  }

  @Override
  public synchronized Object getAttribute(String attribute)
      throws AttributeNotFoundException, MBeanException, ReflectionException {
    Object resultat = null;

    if (attributs.containsKey(attribute)) {
      resultat = attributs.get(attribute);
    } else {
      throw new AttributeNotFoundException(attribute);
    }
    return resultat;
  }

  @Override
  public AttributeList getAttributes(String[] attributes) {
    AttributeList resultat = new AttributeList();

    for (String cle : attributes) {
      if (attributs.containsKey(cle)) {
        resultat.add(new Attribute(cle, attributs.get(cle)));
      }
    }
    return resultat;
  }

  @Override
  public MBeanInfo getMBeanInfo() {
    MBeanParameterInfo[] sansParamInfo = new MBeanParameterInfo[0];

    int i = 0;
    MBeanAttributeInfo attribs[] = new MBeanAttributeInfo[attributs.size()];
    for (String cle : attributs.keySet()) {
      attribs[i] = new MBeanAttributeInfo(cle, attributs.get(cle).getClass()
          .getName(), "Description de l'attribut " + cle, true, true, false);
      i++;
    }

    MBeanConstructorInfo[] constructeurs = new MBeanConstructorInfo[1];
    constructeurs[0] = new MBeanConstructorInfo("SecondDynamic",
        "Constructeur par défaut de la classe", sansParamInfo);

    MBeanOperationInfo[] operations = new MBeanOperationInfo[1];
    operations[0] = new MBeanOperationInfo("rafraichir",
        "Rafraichir les données", sansParamInfo, void.class.getName(),
        MBeanOperationInfo.ACTION);

    return new MBeanInfo(getClass().getName(), "Mon second MBean Dynamic",
        attribs, constructeurs, operations, null);

  }

  @Override
  public Object invoke(String actionName, Object[] params, String[] signature)
      throws MBeanException, ReflectionException {

    try {
      if (actionName.equals("rafraichir")) {
        rafraichir();
      }
      return null;
    } catch (Exception x) {
      throw new MBeanException(x);
    }
  }

  @Override
  public synchronized void setAttribute(Attribute attribute)
      throws AttributeNotFoundException, InvalidAttributeValueException,
      MBeanException, ReflectionException {

    String name = attribute.getName();

    if (attributs.containsKey(name)) {
      attributs.remove(name);
      attributs.put(name, attribute.getValue());
    } else {
      throw new AttributeNotFoundException(name);
    }
  }

  @Override
  @SuppressWarnings("unchecked")
  public synchronized AttributeList setAttributes(AttributeList attributes) {
    for (Iterator i = attributes.iterator(); i.hasNext();) {
      Attribute attr = (Attribute) i.next();
      try {
        setAttribute(attr);
      } catch (AttributeNotFoundException e) {
        e.printStackTrace();
      } catch (InvalidAttributeValueException e) {
        e.printStackTrace();
      } catch (MBeanException e) {
        e.printStackTrace();
      } catch (ReflectionException e) {
        e.printStackTrace();
      }
    }
    return attributes;

  }

  private void rafraichir() {
    System.out.println("Rafraichir les donnees");
  }
}
```

Il est possible d'avoir un contrôle précis sur les fonctionnalités exposées en fonction d'un contexte. Par exemple, il est possible en fonction de la valeur d'une propriété d'exposer toutes les fonctionnalités ou seulement un sous-ensemble.

Remarque : il est tout à fait possible pour un MBean standard d'implémenter aussi l'interface DynamicMBean.

32.10.4. La classe StandardMBean

Il peut être intéressant pour un MBean standard de profiter de certaines fonctionnalités des MBeans Dynamic bien que l'interface du MBean soit statique. Ces fonctionnalités concernent par exemple la possibilité de fournir une description des attributs ou des méthodes.

Dans ce cas, plutôt que d'implémenter l'interface DynamicMBean, il est plus simple d'hériter de la classe javax.management.StandardMBean. Ainsi, les fonctionnalités du MBean sont toujours exposées sous la forme de son interface statique et il est possible de bénéficier des fonctionnalités des MBeans Dynamic.

La classe StandardMBean possède deux constructeurs :

Constructeur	Rôle
StandardMBean(Class interface)	Créer un MBean Dynamic à partir de l'interface du MBean

StandardMBean(Object implementation, Class interface)	Créer un MBean Dynamic à partir de l'instance du MBean et de son interface

Elle propose de nombreuses méthodes pour permettre d'interagir avec les informations dynamiques demandées sur une fonctionnalité.

La classe StandardMBean a été ajoutée par la version 1.2 de JMX.

32.11. Les Model MBeans

Les Model MBeans sont des Dynamic MBeans génériques et configurables.

Chaque implémentation de JMX à l'obligation de fournir une implémentation d'une classe nommée javax.management.modelmbean.RequiredModelMBean qui implémente l'interface ModelMBean

La classe RequiredModelMbean agit comme un modèle générique pour créer dynamiquement des MBeans à partir d'objets qui ne respectent pas les spécifications des MBeans. Un Model MBean est donc obligatoirement un Dynamic MBean puisqu'il n'est pas possible de connaître à l'avance la classe qu'il va encapsuler.

Les informations de configuration des fonctionnalités exposées sont encapsulées dans une instance de l'interface ModelMBeanInfo. Un descripteur encapsulé dans l'interface Descriptor permet de fournir le mapping entre une fonctionnalité exposée et la méthode correspondante à invoquer dans l'objet encapsulé dans le Model MBean.

Pour exposer un objet sous la forme d'un Model MBean, il faut suivre plusieurs étapes :

- instancier l'objet
- créer une instance de la classe RequiredModelMBean
- fournir les informations sur les fonctionnalités exposées au Model MBean
- fournir au ModelMBean l'instance de l'objet
- enregistrer le Model MBean dans le serveur de MBeans

Les Model MBeans sont une des fonctionnalités avancées proposées par la spécification JMX. Leur mise en oeuvre est relativement compliquée ce qui limite leur usage. Ils ont cependant plusieurs intérêts :

- ils permettent à un objet ne respectant pas les spécifications des MBeans d'être exposé au travers de JMX
- ils permettent d'ajouter une redirection qui assure aux clients JMX que la classe RequiredModelMbean est toujours présente puisque fournie obligatoirement avec l'implémentation alors que la classe qu'elle expose ne l'est pas forcement
- ils peuvent fournir des informations supplémentaires aux classes MBean*Info sous la forme d'objets de type Descriptor

32.11.1. L'interface ModelMBean et la classe RequiredModelMBean

Chaque implémentation de JMX doit fournir une implémentation de l'interface ModelMBean sous la forme d'une classe nommée RequiredModelMBean

L'interface ModelMBean doit être implémentée par un Model MBean. Cette interface définit deux méthodes :

Méthode	Rôle
void setModelMBeanInfo(ModelMBeanInfo inModelMBeanInfo)	Fournir les informations de configuration du Model MBean
void setManagedResource(java.lang.Object mr, java.lang.String mr_type)	Fournir l'instance de l'objet sur lequel le Model Bean va invoquer les méthodes

L'interface ModelMBeanInfo encapsule les informations et les descriptions des fonctionnalités de l'objet qui seront exposées au travers du Model MBean.

Chaque implémentation de JMX doit fournir une implémentation de la classe RequiredModelMBean. Cette implémentation doit fournir les fonctionnalités de base pour exposer une instance d'une classe sous la forme d'un MBean quand cette classe ne respecte pas les spécifications de JMX.

La classe RequiredModelMBean possède deux constructeurs :

Constructeur	Rôle
RequiredModelMBean()	Créer une instance avec un objet de type ModelMBeanInfo vide
RequiredModelMBean(ModelMBeanInfo mbi)	Créer une instance avec l'objet de type ModelMBeanInfo fourni en paramètre

Les méthodes setModelMBeanInfo() et setManagedResource() peuvent être utilisées pour fournir respectivement la description des fonctionnalités exposées par le MBean et l'instance de la classe qui sera encapsulée par le MBean.

Pour des besoins spécifiques, il est possible de créer une classe fille de la classe RequiredModelMBean.

32.11.2. La description des fonctionnalités exposées

La classe javax.management.modelmbean.ModelMBeanInfoSupport propose une implémentation de l'interfaceMBeanInfo qui facilite la création d'une instance du type de cette interface.

Cette classe propose trois constructeurs :

Constructeur	Rôle
ModelMBeanInfoSupport(ModelMBeanInfo mbi)	Créer une instance qui est une duplication de celle fournie en paramètre
ModelMBeanInfoSupport(java.lang.String className, java.lang.String description, ModelMBeanAttributeInfo[] attributes, ModelMBeanConstructorInfo[] constructors, ModelMBeanOperationInfo[] operations, ModelMBeanNotificationInfo[] notifications)	Créer une instance avec les informations fournies en paramètre et un descripteur par défaut. Le premier paramètre est le nom pleinement qualifié de la classe qui sera encapsulée par le MBean
ModelMBeanInfoSupport(java.lang.String className, java.lang.String description, ModelMBeanAttributeInfo[] attributes, ModelMBeanConstructorInfo[] constructors, ModelMBeanOperationInfo[] operations, ModelMBeanNotificationInfo[] notifications, Descriptor mbeandescriptor)	Créer une instance avec les informations et le descripteur fournis en paramètre. Le premier paramètre est le nom pleinement qualifié de la classe qui sera encapsulée par le MBean

Les informations sur les fonctionnalités exposées (attributs, constructeurs, opérations et notifications) sont décrites grâce à différents objets :

- ModelMBeanAttributeInfo : décrit les informations relatives à un attribut
- ModelMBeanConstructorInfo : décrit les informations relatives à un constructeur
- ModelMBeanOperationInfo : décrit les informations relatives à une opération
- ModelMBeanNotificationInfo : décrit les informations relatives à une notification

L'interface Descriptor permet de fournir des informations supplémentaires sur un attribut, un constructeur, une opération ou une notification. Une instance de type Descriptor peut être associée à chaque instance des classes ModelMBean*Info.

Une instance de Descriptor encapsule une collection de champs ayant chacun la forme nom=valeur.

La classe DescriptorSupport implémente l'interface Descriptor et permet de facilement créer une instance de type Descriptor.

Pour créer une instance de l'interface ModelMBeanInfo, il faut écrire beaucoup de code car il faut créer au moins un objet pour chaque élément exposé par le MBean.

Remarque importante : il faut définir chaque attribut avec un objet de type ModelMBeanAttributeInfo mais il faut aussi impérativement définir chaque getter et chaque setter dans un objet de type ModelMBeanOperationInfo. Ceci est nécessaire car les conventions de nommage des JavaBeans n'ont pas d'obligation a être respectées dans la classe qui sera encapsulée par le MBean.

Un objet de type RequiredModelMBean ne fait pas d'introspection sur la classe qu'il encapsule pour vérifier les informations fournies dans le ModelMBeanInfo : il fait aveuglement confiance aux informations qu'il contient.

Certaines implémentations peuvent fournir des utilitaires pour faciliter l'instanciation de l'interface ModelMBeanInfo par exemple à partir de fichiers XML, ce qui réduit la quantité de code à produire pour développer un Model MBean.

32.11.3. Un exemple de mise en oeuvre

L'exemple de cette section va exposer sous la forme d'un Model MBean une instance d'une simple classe nommée MaClasse.

Exemple :

```
package fr.jmdoudoux.dej.jmx;

public class MaClasse {

  private static String nom = "MaClasse";
  private int valeur = 100;

  public String getNom() {
    return nom;
  }

  public int getValeur() {
    return valeur;
  }

  public synchronized void setValeur(int valeur) {
     this.valeur = valeur;
  }

  public void rafraichir() {
    System.out.println("Rafraichir les donnees");
  }

  public MaClasse() {
  }
}
```

La classe MaClasse est un simple POJO qui n'implémente aucune interface particulière : elle ne respecte aucune spécification de JMX. L'exemple suivant va encapsuler une instance de cette classe dans un Model MBean et fournir une description de ses fonctionnalités exposées par le Model MBean.

Dans l'agent JMX, une instance de la classe RequiredModelMBean est instanciée en passant en paramètre de son constructeur l'instance de l'interface ModelMBeanInfo.

Les informations sur les fonctionnalités exposées par le Model MBean et le mapping avec les méthodes correspondantes à invoquer sont encapsulés dans cette instance de l'interface ModelMBeanInfo.

Exemple :

```
package fr.jmdoudoux.dej.jmx;

import java.lang.management.ManagementFactory;

import javax.management.Descriptor;
import javax.management.InstanceAlreadyExistsException;
import javax.management.InstanceNotFoundException;
import javax.management.MBeanException;
import javax.management.MBeanParameterInfo;
import javax.management.MBeanRegistrationException;
import javax.management.MBeanServer;
import javax.management.MalformedObjectNameException;
import javax.management.NotCompliantMBeanException;
import javax.management.ObjectName;
import javax.management.RuntimeOperationsException;
import javax.management.modelmbean.DescriptorSupport;
import javax.management.modelmbean.InvalidTargetObjectTypeException;
import javax.management.modelmbean.ModelMBeanAttributeInfo;
import javax.management.modelmbean.ModelMBeanConstructorInfo;
import javax.management.modelmbean.ModelMBeanInfo;
import javax.management.modelmbean.ModelMBeanInfoSupport;
import javax.management.modelmbean.ModelMBeanOperationInfo;
import javax.management.modelmbean.RequiredModelMBean;

public class LancerAgentModelMBean {

  public static void main(String[] args) {
    MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();

    ObjectName name = null;
    try {
      name = new ObjectName(
          "fr.jmdoudoux.dej.jmx:type=OpenMXBean,name=MaClasse");

      MaClasse maClasse = new MaClasse();
      RequiredModelMBean modelMBean = new RequiredModelMBean(creerMBeanInfo());
      modelMBean.setManagedResource(maClasse, "objectReference");
      mbs.registerMBean(modelMBean, name);

      System.out.println("Lancement ...");
      while (true) {
        Thread.sleep(1000);
      }
    } catch (MalformedObjectNameException e) {
      e.printStackTrace();
    } catch (NullPointerException e) {
      e.printStackTrace();
    } catch (InstanceAlreadyExistsException e) {
      e.printStackTrace();
    } catch (MBeanRegistrationException e) {
      e.printStackTrace();
    } catch (NotCompliantMBeanException e) {
      e.printStackTrace();
    } catch (InterruptedException e) {
    } catch (RuntimeOperationsException e) {
      e.printStackTrace();
    } catch (InstanceNotFoundException e) {
      e.printStackTrace();
    } catch (MBeanException e) {
      e.printStackTrace();
    } catch (InvalidTargetObjectTypeException e) {
      e.printStackTrace();
    }
  }

  private static ModelMBeanInfo creerMBeanInfo() {
    Descriptor descriptorValeur = new DescriptorSupport(new String[] {
        "name=Valeur", "descriptorType=attribute", "default=0",
        "displayName=Valeur stockée dans la classe", "getMethod=getValeur",
        "setMethod=setValeur" });

    Descriptor descriptorNom = new DescriptorSupport(new String[] { "name=Nom",
        "descriptorType=attribute", "displayName=Nom de la classe",
        "getMethod=getNom" });
    ModelMBeanAttributeInfo[] mmbai = new ModelMBeanAttributeInfo[2];
    mmbai[0] = new ModelMBeanAttributeInfo("Valeur", "java.lang.Integer",
        "Valeur stockée dans la classe", true, true, false, descriptorValeur);
    mmbai[1] = new ModelMBeanAttributeInfo("Nom", "java.lang.String",
        "Nom de la classe", true, false, false, descriptorNom);

    ModelMBeanOperationInfo[] mmboi = new ModelMBeanOperationInfo[4];

    mmboi[0] = new ModelMBeanOperationInfo("getValeur",
        "getter pour l'attribut Valeur", null, "Integer",
        ModelMBeanOperationInfo.INFO);

    MBeanParameterInfo[] mbpiSetValeur = new MBeanParameterInfo[1];
    mbpiSetValeur[0] = new MBeanParameterInfo("valeur", "java.lang.Integer",
        "valeur de l'attribut");
    mmboi[1] = new ModelMBeanOperationInfo("setValeur",
        "setter pour l'attribut Valeur", mbpiSetValeur, "void",
        ModelMBeanOperationInfo.ACTION);

    mmboi[2] = new ModelMBeanOperationInfo("getNom",
        "getter pour l'attribut Nom", null, "String",
        ModelMBeanOperationInfo.INFO);

    mmboi[3] = new ModelMBeanOperationInfo("rafraichir",
        "Rafraichir les données", null, "void", ModelMBeanOperationInfo.ACTION);

    ModelMBeanConstructorInfo[] mmbci = new ModelMBeanConstructorInfo[1];
    mmbci[0] = new ModelMBeanConstructorInfo("MaClasse",
        "Constructeur par défaut", null);

    return new ModelMBeanInfoSupport("fr.jmdoudoux.dej.jmx.MaClasse",
        "Exemple de ModelBean", mmbai, mmbci, mmboi, null);
  }
}
```

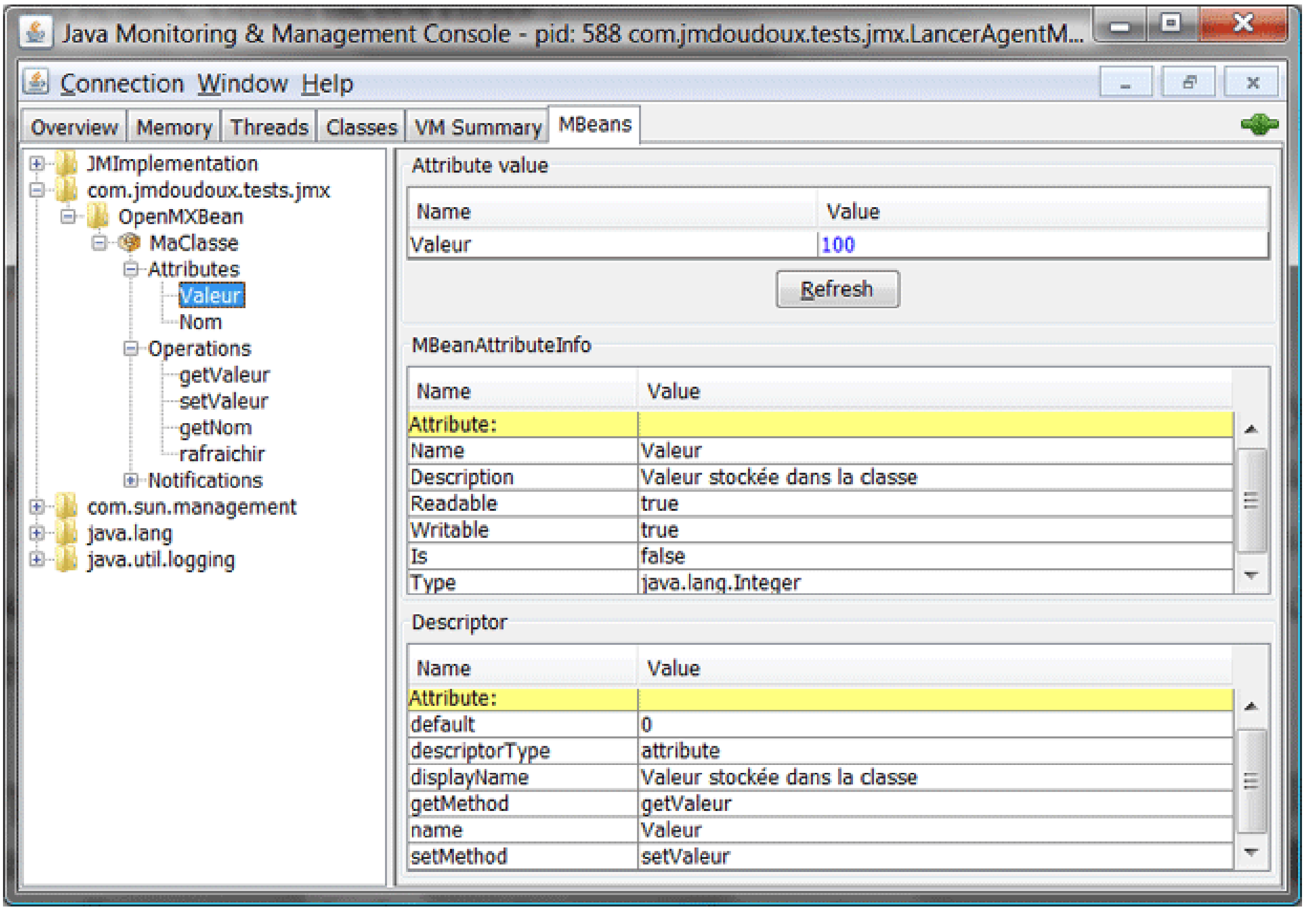
La partie la plus délicate lors de la mise en oeuvre d'un Model MBean est de créer cette instance de l'interface ModelMBeanInfo qui va contenir la description des fonctionnalités à exposer ainsi que le mapping vers les méthodes de l'instance à invoquer. Pour rendre le code plus lisible, la création de cette instance est faite dans une méthode dédiée.

L'instance de la classe à invoquer est fournie en paramètre du Model MBean en utilisant la méthode setManagedRessource(). Elle attend deux paramètres :

- l'instance de l'objet que le MBean va encapsuler
- une chaîne de caractères qui précise la typologie de référence fournie. Plusieurs valeurs sont utilisables : ObjectReference, Handle, IOR, EJBHandle ou RMIReference. Si la valeur fournie ne fait pas partie de cette liste, une exception de type InvalidTargetTypeException est levée

C'est bien le Model MBean qui est enregistré dans le serveur de MBeans. Pour les clients JMX, le Model MBean est un Dynamic MBean. Le client n'a pas besoin d'avoir accès à la classe de l'instance encapsulée dans le Model MBean.

L'objet de type RequiredModelMBean va dynamiquement générer les entités nécessaires pour exposer les fonctionnalités sous la forme d'un Dynamic MBean.



Attribute value

Name	Value
Valeur	100

MBeanAttributeInfo

Name	Value
Attribute:	
Name	Valeur
Description	Valeur stockée dans la classe
Readable	true
Writable	true
Is	false
Type	java.lang.Integer

Descriptor

Name	Value
Attribute:	
default	0
descriptorType	attribute
displayName	Valeur stockée dans la classe
getMethod	getValeur
name	Valeur
setMethod	setValeur

Il suffit de lancer l'agent et d'ouvrir un client JMX pour pouvoir interagir avec le ModelMBean.

Hormis le fait que les getters et les setters apparaissent dans les opérations, rien ne distingue le ModelMBean d'un autre MBean pour le client JMX.

32.11.4. Les fonctionnalités optionnelles des Model MBeans

La classe RequiredModelMBean peut proposer un support optionnel de certaines fonctionnalités comme le logging, la persistance ou un cache de données (caching).

La fonctionnalité de cache de données permet de stocker dans une variable du MBean la valeur d'un attribut et de la renvoyer durant sa durée de vie dans le cache plutôt que de toujours solliciter dynamiquement l'instance encapsulée.

Certaines de ces fonctionnalités peuvent être configurées au travers des données fournies par un Descriptor. Celles-ci peuvent être modifiées dynamiquement par le MBean ou par un client JMX pour adapter le comportement des fonctionnalités par défaut.

Pour connaître les fonctionnalités optionnelles implémentées et la façon de les mettre en oeuvre il faut consulter la documentation de l'implémentation utilisée. Leur utilisation limite donc la portabilité vers une autre implémentation de JMX.

32.11.5. Les différences entre un Dynamic MBean et un Model MBean

Un Model MBean est un Dynamic MBean mais il est plus souple à mettre en oeuvre : le code d'un Dynamic MBean doit être intégralement écrit alors que pour un Model MBean une implémentation par défaut est fournie obligatoirement par l'implémentation de JMX et il suffit de lui fournir les fonctionnalités exposées et une référence sur l'objet à invoquer.

Les traitements d'un Dynamic MBean doivent être codés dans le MBean alors qu'avec un Model MBean le code est contenu dans l'instance de la classe dont les fonctionnalités sont exposées par le MBean.

Les fonctionnalités exposées par un Dynamic MBean doivent être connues au moment de l'écriture du code du MBean. Avec un Model MBean, les fonctionnalités exposées peuvent être définies et modifiées dynamiquement en utilisant la méthode setModelMBeanInfo().

Un Model MBean peut fournir des descriptions complémentaires des fonctionnalités qu'il expose sous la forme d'objets de type Descriptor qui encapsulent des champs. Ces champs peuvent être personnalisés.

L'implémentation d'un Model MBean peut proposer des services optionnels (persistance, logging, caching) qui peuvent être dynamiquement configurés avec des champs d'un Descriptor.

32.12. Les Open MBeans

La spécification JMX permet l'emploi de types complexes et portables en utilisant les Open MBeans.

Un Open MBean est un Dynamic MBean où tous les types utilisés doivent appartenir à une liste précisément définie dans les spécifications JMX. Ces types de base peuvent être utilisés dans un type composé dédié appelé Open Type.

Seul un sous-ensemble restreint de classes peut être décrit sous la forme d'un OpenType :

- les wrapper de primitives : Integer, Boolean, Void, ...
- les classes String, BigDecimal, BigInteger, Date
- CompositeData et TabularData

Il faut noter que les types primitifs ne sont pas autorisés : il faut utiliser leur wrapper.

Ceci permet à un Open MBean d'être portable avec des types complexes mais sans utiliser un type spécifique qui devrait être fourni à chaque client JMX. Les Open MBeans sont ainsi les MBeans les plus ouverts et les plus portables avec les clients JMX puisqu'ils n'ont pas besoin d'ajouter dans leur classpath des classes spécifiques.

La restriction des types de données utilisables est aussi intéressante pour les connecteurs et les adaptateurs de protocoles qui n'ont qu'à savoir gérer ces types.

Un objet de type OpenMBeanInfo encapsule la description des fonctionnalités exposées par un Open MBean en incluant en plus du type Java un objet de type OpenType pour chaque attribut et opération.

Les spécifications des Open MBeans dans la version 1.0 de JMX sont incomplètes donc inutilisables. Dans la version 1.1, les spécifications sont complètes mais leur implémentation est facultative. A partir de la version 1.2, elles sont obligatoires dans toutes les implémentations.

32.12.1. La mise en oeuvre d'un Open MBean

Un Open MBean doit implémenter l'interface DynamicMBean et n'a pas besoin d'implémenter une interface spécifique aux Open MBeans. La différence avec un Dynamic MBean se fait à deux niveaux :

- une restriction forte sur les types de données utilisables par le MBean
- une description des fonctionnalités du MBean avec des interfaces dédiées aux Open MBeans

La description des fonctionnalités d'un Open MBean est assurée grâce aux interfaces du package javax.management.openmbean.

Pour distinguer les Open MBeans des autres MBeans, les données de description des fonctionnalités exposées sont encapsulées dans des interfaces dédiées du package javax.management.openmbean :

- OpenMBeanInfo : contient l'ensemble des fonctionnalités exposées
- OpenMBeanOperationInfo : contient la description d'une méthode
- OpenMBeanConstructorInfo : contient la description d'un constructeur
- OpenMBeanParameterInfo : contient la description d'un paramètre
- OpenMBeanAttributeInfo : contient la description d'un attribut

Chacune de ces interfaces possède une classe correspondante dont le nom se termine par Support.

Un objet de type OpenMBeanInfo peut décrire la valeur par défaut et les valeurs autorisées pour un attribut, un paramètre et une valeur de retour.

Pour la description des notifications, les Open MBeans utilisent la classe MBeanNotificationInfo.

L'interface OpenMBeanOperationInfo possède une propriété impact de type int qui précise l'impact de la méthode lorsqu'elle est invoquée. Les valeurs possibles sont :

- MBeanOperationInfo.INFO : la méthode est en lecture seule
- MBeanOperationInfo.ACTION : la méthode va modifier l'état du MBean
- MBeanOperationInfo.ACTION_INFO : la méthode lit et modifie
- MBeanOperationInfo.UNKNOWN : le type d'action n'est pas précisé

32.12.2. Les types de données utilisables dans les Open MBeans

Les types utilisés pour les attributs, les paramètres et les valeurs de retour des opérations d'un Open MBean doivent appartenir à la liste des types précisés dans les spécifications JMX :

- les wrappers sur les types primitifs : java.lang.Boolean, java.lang.Byte, java.lang.Character, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Float, java.lang.Double, java.lang.String, java.lang.Void, java.math.BigInteger, java.math.BigDecimal
- les classes de données standard : String, Date
- des classes de JMX : CompositeData, TabularData, et ObjectName

L'ensemble de ces types est désigné sous le nom Open Types.

32.12.2.1. Les Open Types

Les Open Types encapsulent la description d'un type de données utilisé par les Open MBeans. Ils permettent de fournir un moyen standard de décrire les types utilisables par les Open MBeans. Un Open Type peut représenter un type primitif sous la forme de son wrapper, un type complexe composé d'autres Open Types ou un tableau d'Open Types.

La classe Abstraite OpenType est la classe mère des différentes classes qui encapsulent des Open Types :

- SimpleType : permet de décrire un type simple
- CompositeType : permet de décrire un type composé d'autres Open Types
- TabularType : permet de décrire un type de données tabulaires
- ArrayType : permet de décrire un type de données sous la forme d'un tableau multi-dimension

Les classes Composite Type, TabularType et ArrayType peuvent contenir d'autres Open Types.

Les instances des Open Types sont, selon leur classe, une instance de différents types :

- un wrapper pour les types primitifs
- une instance de la classe String, Date ou ObjectName
- une instance de la classe CompositeData
- une instance de la classe TabularData

Les interfaces CompositeData et TabularData encapsulent les données de leurs types complexes respectifs.

32.12.2.2. La classe CompositeType et l'interface CompositeData

Les données utilisées par un Open MBean peuvent être de type CompositeData. Cette classe permet d'encapsuler un objet complexe qui n'utilisera que des OpenTypes et respectera ainsi les spécifications des Open MBeans.

L'OpenType correspondant dans l'OpenMBeanInfo est CompositeType. Un CompositeType décrit un ensemble d'éléments ou de champs. Chaque élément possède un nom et un Open Type.

La classe CompositeData associe une clé à une valeur pour chacun des attributs définis avec un CompositeType. Un ou plusieurs éléments d'un CompositeData forment la clé de l'élément.

Les classes CompositeDataSupport et TabularDataSupport proposent une implémentation respectueuse des interfaces CompositeData et TabularData pour faciliter leur création.

Un objet de type CompositeData est immuable : toutes les données qu'il encapsule doivent donc être fournies lors de son instanciation. La classe CompositeDataSupport propose pour cela deux constructeurs :

Constructeur	Rôle
CompositeDataSupport(CompositeType compositeType, Map<String,?> items)	Créer une instance qui encapsule les données du CompositeType avec les valeurs fournies dans la collection de type Map
CompositeDataSupport(CompositeType compositeType, String[] itemNames, Object[] itemValues)	Créer une instance qui encapsule les données du CompositeType avec les valeurs fournies sous la forme de deux tableaux (un qui contient les noms des attributs et l'autre qui encapsule leurs valeurs : l'ordre des données des deux tableaux doit correspondre).

La méthode getCompositeType() renvoie une instance de la classe CompositeType qui contient la description de l'Open Type encapsulé.

32.12.2.3. La classe TabularType et l'interface TabularData

Les données utilisées par un Open MBean peuvent être de type TabularData. Cette classe permet d'encapsuler un tableau d'objets de type CompositeData qui n'utilisera que des OpenTypes et respectera ainsi les spécifications des Open MBeans.

L'OpenType correspondant dans l'OpenMBeanInfo est TabularType. Tous les éléments d'un TabularData possèdent le même CompositeType.

Une instance de TabularData encapsule une collection d'objets de type CompositeData. Chaque objet de ce type encapsule les données d'une occurrence. Chaque occurrence possède une clé unique obtenue à partir des données encapsulées dans le CompositeData.

Avec une instance de TabularData, il est possible d'ajouter ou de supprimer une ou plusieurs occurrences.

La classe TabularDataSupport encapsule un tableau à une dimension d'objets de type CompositeData. Tous les CompositeData contenus dans le TabularData doivent avoir le même CompositeType.

Chaque occurrence est associée à une clé unique qui identifie cette occurrence. Cette clé est composée d'un ou plusieurs attributs encapsulés dans le CompositeData correspondant. Généralement cette clé est composée d'une seule donnée de type Integer ou String.

La classe TabularDataSupport propose toutes les méthodes nécessaires pour manipuler les occurrences qu'elle encapsule : put(), putAll(), get(), remove(), clear(), size(), isEmpty(), containsKey(), keySet(), values(), ...

La méthode get() attend en paramètre un tableau des valeurs de clés et renvoie l'objet de type CompositeData associé si celui-ci existe.

32.12.3. Un exemple d'utilisation d'un Open MBean

La suite de cette section sera développée dans une version future de ce document

32.12.4. Les avantages et les inconvénients des Open MBeans

Le grand avantage des Open MBeans est de garantir l'interopérabilité. Leur principal défaut est d'être des Dynamic MBeans, donc relativement difficiles à coder. Une alternative est de coder un MBean Standard qui n'utilise que des Open Types.

La version 1.1 des spécifications de JMX ne fournit pas tous les détails concernant les Open MBeans, leur support est donc optionnel. D'ailleurs l'implémentation de référence de la version 1.1 ne propose pas d'implémentation pour les Open MBeans.

32.13. Les MXBeans

La version 6.0 de Java introduit un nouveau type de MBean : les MXBeans. Ce type de MBeans permet d'utiliser des types définis par l'utilisateur du moment que ces types respectent les contraintes définies dans les spécifications.

Ces types sont convertis vers des Open Types tels que SimpleType, CompositeType, ArrayType, TabularType, ... définis pour les Open MBeans. Les Open Types sont définis dans le package javax.management.openmbean.

Cela permet une utilisation du MBean sans que le client JMX n'aie besoin du jar qui contient la définition du MBean même si ce client est distant.

Les MXBeans sont définis grâce à une interface statique mais contrairement aux MBeans standard le nom de la classe qui implémente l'interface du MXBean est libre.

La plate-forme Java fournit plusieurs MXBeans regroupés dans le package java.lang.management. Le développeur peut aussi écrire les siens.

32.13.1. La définition d'un MXBean

L'interface d'un MXBean doit soit avoir un nom qui termine par convention par MXBean soit être annotée avec l'annotation @javax.management.MXBean. Dans ce dernier cas, le nom de l'interface est libre.

Exemple :

```
package fr.jmdoudoux.dej.jmx;

public interface InfoMXBean {

  public InfoParametre getInfo();

  public void rafraichir();
}
```

Le MXBean doit implémenter son interface.

Exemple :

```
package fr.jmdoudoux.dej.jmx;

import java.util.Date;
```

```
public class Info implements InfoMXBean {

  @Override
  public InfoParametre getInfo() {
    return new InfoParametre("nom1", new Date(), 100l, "description1");
  }

  @Override
  public void rafraichir() {
    System.out.println("Appel de la méthode rafraichir()");
  }
}
```

32.13.2. L'écriture d'un type personnalisé utilisé par le MXBean

L'exemple utilise un type personnalisé qui est un simple bean. Le constructeur est annoté avec l'annotation @ConstructorProperties. Cette annotation permet d'associer chaque paramètre d'un constructeur à une propriété.

Exemple :

```
package fr.jmdoudoux.dej.jmx;

import java.beans.ConstructorProperties;
import java.util.Date;

public class InfoParametre {

  private String nom;
  private String description;
  private Date dateCreation;
  private long taille;

  @ConstructorProperties( { "nom", "dateCreation", "taille", "description" })
  public InfoParametre(String pnom, Date pdateCreation, long ptaille,
      String pdescription) {
    super();
    this.nom = pnom;
    this.description = pdescription;
    this.dateCreation = pdateCreation;
    this.taille = ptaille;
  }

  public String getNom() {
    return nom;
  }

  public String getDescription() {
    return description;
  }

  public void setDescription(String description) {
    this.description = description;
  }

  public Date getDateCreation() {
    return dateCreation;
  }

  public void setDateCreation(Date dateCreation) {
    this.dateCreation = dateCreation;
  }

  public long getTaille() {
    return taille;
  }

  public void setTaille(long taille) {
    this.taille = taille;
  }
}
```

JMX va utiliser les getters pour créer une instance de CompositeData qui encapsule les données. Pour recréer une instance de la classe InfoParametre à partir d'une instance de CompositeDate, JMX utilise les informations fournies par l'annotation @ConstructorProperties.

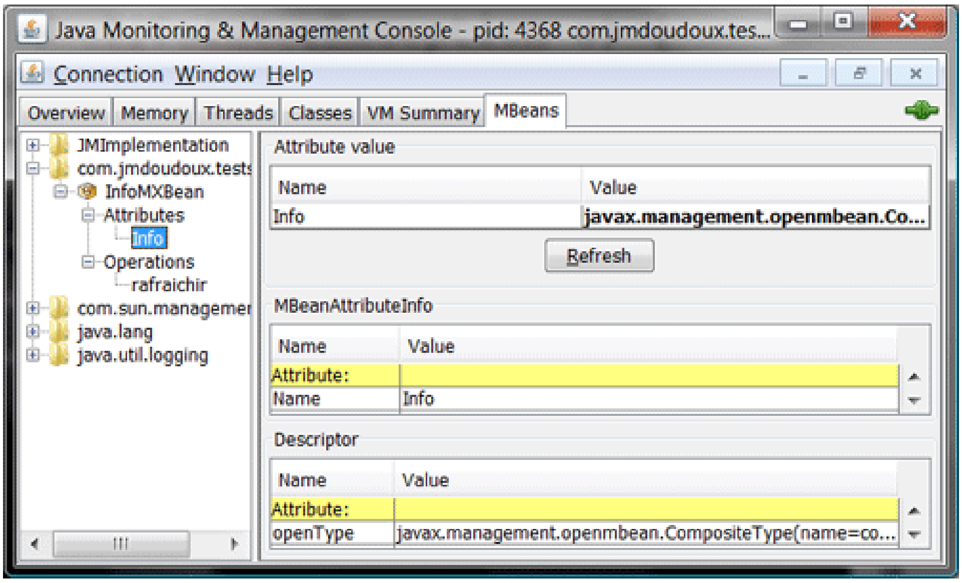
32.13.3. La mise en oeuvre d'un MXBean

Le MXBean doit être instancié et enregistré dans le serveur de MBeans de la même manière que pour les autres MBeans.

Exemple :

```
package fr.jmdoudoux.dej.jmx;

import java.lang.management.ManagementFactory;

import javax.management.InstanceAlreadyExistsException;
import javax.management.MBeanRegistrationException;
import javax.management.MBeanServer;
import javax.management.MalformedObjectNameException;
import javax.management.NotCompliantMBeanException;
import javax.management.ObjectName;

public class LancerAgentMXBean {

  public static void main(String[] args) {
    MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();

    ObjectName name = null;
    try {
      name = new ObjectName("fr.jmdoudoux.dej.jmx:type=InfoMXBean");

      InfoMXBean mbean = new Info();

      mbs.registerMBean(mbean, name);

      System.out.println("Lancement ...");
      while (true) {

        Thread.sleep(1000);
      }
    } catch (MalformedObjectNameException e) {
      e.printStackTrace();
    } catch (NullPointerException e) {
      e.printStackTrace();
    } catch (InstanceAlreadyExistsException e) {
      e.printStackTrace();
    } catch (MBeanRegistrationException e) {
      e.printStackTrace();
    } catch (NotCompliantMBeanException e) {
      e.printStackTrace();
    } catch (InterruptedException e) {
    }
  }
}
```

La principale différence est au niveau du client JMX : l'attribut Info n'est pas du type InfoParametre mais de l'Open Type CompositeData.

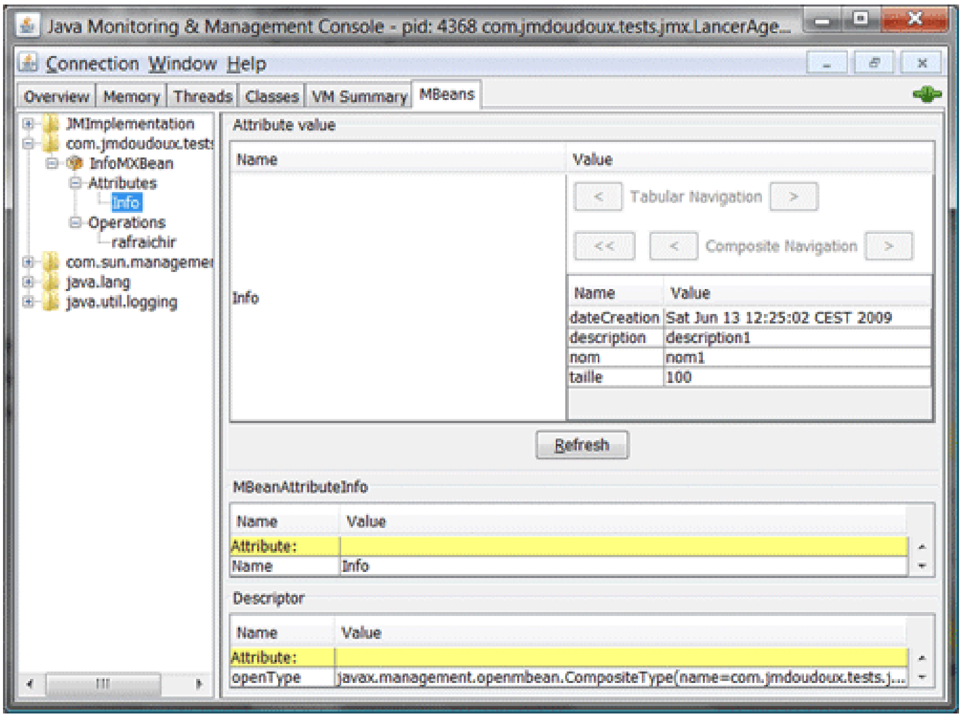


Name	Value
Info	javax.management.openmbean.Co...

Name	Value
Attribute:	
Name	Info

Name	Value
Attribute:	
openType	javax.management.openmbean.CompositeType(name=co...

En double cliquant sur la valeur de la ligne de l'attribut Info, il est possible d'afficher le détail des données encapsulées dans le CompositeData.



Name	Value
dateCreation	Sat Jun 13 12:25:02 CEST 2009
description	description1
nom	nom1
taille	100

Name	Value
Attribute:	
Name	Info

Name	Value
Attribute:	
openType	javax.management.openmbean.CompositeType(name=com.jmdoudoux.tests.j...

32.14. L'interface PersistentMBean

Il peut être nécessaire à un MBean de rendre ses données persistantes. Dans ce cas, le MBean doit implémenter l'interface javax.management.PersistentMBean.

Cette interface ne définit que deux méthodes :

Méthode	Rôle
void load()	Lire des données du MBean à partir de l'unité de persistance
void store()	Ecriture des données du MBean vers l'unité de persistance

L'invocation de la méthode load() dans son constructeur est à la charge de l'implémentation du MBean.

L'invocation de la méthode save() peut être définie dans une Persistance Policy.

La suite de cette section sera développée dans une version future de ce document

L'implémentation du MBean est libre de choisir l'unité de persistance utilisée (fichier, base de données, ...)

32.15. Le monitoring d'une JVM

JMX est aussi utilisé pour surveiller et gérer la JVM. A partir de la version 5 de Java : un agent JMX peut être utilisé pour accéder à l'instrumentation de la JVM et ainsi la surveiller et la gérer à distance.

Java 5.0 propose plusieurs fonctionnalités relatives au monitoring notamment :

- instrumentation de la JVM : la JVM est instrumentée avec des MBeans
- l'API de monitoring et de gestion : le package java.lang.management contient des classes et interfaces permettant d'obtenir des informations sur l'état de l'exécution de la JVM notamment sous la forme de MXBeans (mémoire, threads, garbage collection, ...)
- des outils : Java 5.0 propose l'outil JConsole qui permet d'afficher des informations sur la JVM et agit comme un client JMX graphique. Java 6.0 propose un outil encore plus évolué : Java Visual VM.

La JVM incorpore un serveur de MBeans et utilise des MXBeans dédiés fournis avec la plate-forme Java SE pour permettre de surveiller et gérer la JVM.

Ces MXBeans encapsulent chacun une grande fonctionnalité de la JVM : chargement des classes, ramasse-miettes, compilateur JIT, threads, mémoire, ... Ceci permet d'obtenir de façon standard, grâce à JMX, des informations sur la consommation en ressources et l'activité de la JVM.

Il est ainsi possible de consulter et d'interagir avec ces fonctionnalités en utilisant un client JMX comme JConsole fourni avec le JDK.

Pour pouvoir activer la connexion RMI de l'agent JMX de la JVM, il faut utiliser la propriété com.sun.management.jmxremote de la JVM.

Exemple :

```
java -Dcom.sun.management.jmxremote MonApplication
```

Pour permettre un accès à un client distant sans authentification, il faut fournir trois autres propriétés à la JVM.

Exemple :

```
com.sun.management.jmxremote.port=9999
com.sun.management.jmxremote.authenticate=false
com.sun.management.jmxremote.ssl=false
```

Le port précisé ne doit pas être déjà utilisé.

La JSR 174 (Monitoring and Management Specification for the Java Virtual Machine JVM) définit plusieurs MXBeans dans le package java.lang.management pour la gestion et le monitoring de la JVM.

32.15.1. L'interface ClassLoadingMXBean

Cet MXBean permet de surveiller et de gérer le système de chargement des classes de la JVM.

Une instance de l'interface ClassLoadingMXBean est obtenue en invoquant la méthode getClassLoadingMXBean() de la fabrique ManagementFactory.

L'ObjectName pour l'unique instance de cet MXBean est : java.lang:type=ClassLoading

Cet MXBean permet d'obtenir plusieurs informations :

- le nombre de classes chargées dans la JVM
- le nombre total de classes chargées depuis le lancement de la JVM
- le nombre total de classes déchargées depuis le lancement de la JVM

Il permet aussi d'activer ou non l'affichage dans la sortie standard d'informations sur les activités de classloading de la JVM.

Exemple :

```
package fr.jmdoudoux.dej.jmx.mxbeans;

import java.lang.management.ClassLoadingMXBean;
import java.lang.management.ManagementFactory;

public class TestClassLoadingMXB {

  public static void main(String[] args) {

    ClassLoadingMXBean clBean = ManagementFactory.getClassLoadingMXBean();
    System.out.printf("Loaded class count : %d\n", clBean
        .getLoadedClassCount());
    System.out.printf("Total loaded class count : %d\n", clBean.getTotalLoadedClassCount());
    System.out.printf("Unloaded class count : %d\n", clBean
        .getUnloadedClassCount());
    System.out.printf("isVerbose : %b \n", clBean.isVerbose());
    System.out.println();
    clBean.setVerbose(true);
  }
}
```

Résultat :

```
Loaded class count : 334
Total loaded class count : 422
Unloaded class count : 0
isVerbose : false

[Loaded java.util.IdentityHashMap$KeySet from shared objects file]
[Loaded java.util.IdentityHashMap$IdentityHashMapIterator from shared objects file]
[Loaded java.util.IdentityHashMap$KeyIterator from shared objects file]
[Loaded java.io.DeleteOnExitHook from shared objects file]
[Loaded java.util.HashMap$KeySet from shared objects file]
[Loaded java.util.LinkedHashMap$KeyIterator from shared objects file]
```

32.15.2. L'interface CompilationMXBean

Cet MXBean permet de surveiller le système de compilation JIT de la JVM.

Une instance de l'interface CompilationMXBean est obtenue en invoquant la méthode getCompilationMXBean() de la fabrique ManagementFactory.

L'ObjectName pour l'unique instance de cet MXBean est : java.lang:type=Compilation

Cet MXBean permet d'obtenir plusieurs informations :

- le nom du compilateur JIT utilisé par la JVM
- le temps estimé consommé pour la compilation de classes par le compilateur JIT
- un indicateur qui précise si la JVM permet le monitoring du compilateur JIT

Exemple :

```
package fr.jmdoudoux.dej.jmx.mxbeans;

import java.lang.management.CompilationMXBean;
import java.lang.management.ManagementFactory;

public class TestCompilationMXB {

  public static void main(String[] args) {

    CompilationMXBean cBean = ManagementFactory.getCompilationMXBean();
    System.out.printf("Name : %s\n", cBean.getName());
    System.out.printf("Total compilation time : %d ms\n", cBean
        .getTotalCompilationTime());
    System.out.printf("iscompilationTimeMonitoringSupported : %b \n", cBean
        .isCompilationTimeMonitoringSupported());
  }
}
```

Résultat :

```
Name : HotSpot Client Compiler
Total compilation time : 7 ms
iscompilationTimeMonitoringSupported : true
```

32.15.3. L'interface GarbageCollectorMXBean

Cet MXBean permet de surveiller le ramasse-miettes de la JVM. Elle hérite de l'interface MemoryManagerMXBean. Une JVM peut avoir une ou plusieurs instances de cet MXBean, une pour chaque algorithme utilisé pour gérer la mémoire.

Les instances de l'interface GarbageCollectorMXBean sont obtenues en invoquant la méthode getGarbageCollectorMXBeans() de la fabrique ManagementFactory.

L'ObjectName d'une instance de cet MXBean est de la forme : java.lang:type=GarbageCollector, name=xxx

Cet MXBean permet d'obtenir plusieurs informations :

- le nombre de collectes qui ont été réalisées par l'algorithme
- le temps utilisé pour les collectes réalisées par l'algorithme

Exemple :

```
package fr.jmdoudoux.dej.jmx.mxbeans;

import java.lang.management.GarbageCollectorMXBean;
import java.lang.management.ManagementFactory;

public class TestGarbageCollectorMXB {
  public static void main(String[] args) {
    for (int i = 0; i < 100000; i++) {
      Byte[] tableau = new Byte[50 * 1024];
      if ((i % 10000) == 0) {
        System.out.print(".");
      }
    }
    System.out.println("");
    for (GarbageCollectorMXBean gcBean : ManagementFactory.getGarbageCollectorMXBeans()) {
      System.out.printf("Memory manager name : %s\n", gcBean.getName());
      System.out.printf("  isValid : %b\n", gcBean.isValid());
      for (String pool : gcBean.getMemoryPoolNames()) {
        System.out.printf("  Memory pool name : %s\n", pool);
      }
      System.out.printf("\n  collectionCount : %d\n", gcBean
          .getCollectionCount());
      System.out.printf("  collectionTime : %d ms\n", gcBean
          .getCollectionTime());
      System.out.println();
    }
  }
}
```

Résultat :

```
..........
Memory manager name : Copy
  isValid : true
  Memory pool name : Eden Space
  Memory pool name : Survivor Space
  collectionCount : 25000
  collectionTime : 1228 ms
Memory manager name : MarkSweepCompact
  isValid : true
  Memory pool name : Eden Space
  Memory pool name : Survivor Space
  Memory pool name : Tenured Gen
  Memory pool name : Perm Gen
  Memory pool name : Perm Gen [shared-ro]
  Memory pool name : Perm Gen [shared-rw]
  collectionCount : 0
  collectionTime : 0 ms
```

32.15.4. L'interface MemoryManagerMXBean

Cet MXBean permet de lister les gestionnaires de mémoire de la JVM. Une JVM peut avoir une ou plusieurs instances de cet MXBean, une pour chaque gestionnaire utilisé pour gérer la mémoire.

Les instances de l'interface MemoryManagerMXBean sont obtenues en invoquant la méthode getMemoryManagerMXBeans() de la fabrique ManagementFactory.

L'ObjectName d'une instance de cet MXBean est de la forme : java.lang:type=MemoryManager, name=xxx

Cet MXBean permet d'obtenir plusieurs informations :

- le nom du gestionnaire
- les noms des zones de la mémoire gérées par le gestionnaire

Exemple :

```
package fr.jmdoudoux.dej.jmx.mxbeans;

import java.lang.management.ManagementFactory;
import java.lang.management.MemoryManagerMXBean;

public class TestMemoryManagerMXB {
  public static void main(String[] args) {
    for (MemoryManagerMXBean mmBean : ManagementFactory
        .getMemoryManagerMXBeans()) {
      System.out.printf("Memory manager name: %s\n", mmBean.getName());
      System.out.printf("  isValid : %b\n", mmBean.isValid());
      for (String pool : mmBean.getMemoryPoolNames()) {
        System.out.printf("  Memory pool name : %s\n", pool);
      }
      System.out.println();
    }
  }
}
```

Résultat :

```
Memory manager name: CodeCacheManager
  isValid : true
  Memory pool name : Code Cache

Memory manager name: Copy
  isValid : true
  Memory pool name : Eden Space
  Memory pool name : Survivor Space

Memory manager name: MarkSweepCompact
  isValid : true
  Memory pool name : Eden Space
  Memory pool name : Survivor Space
  Memory pool name : Tenured Gen
  Memory pool name : Perm Gen
  Memory pool name : Perm Gen [shared-ro]
  Memory pool name : Perm Gen [shared-rw]
```

32.15.5. L'interface MemoryMXBean

Cet MXBean permet de surveiller et de gérer la mémoire de la JVM.

Une instance de l'interface MemoryMXBean est obtenue en invoquant la méthode getMemoryMXBean() de la fabrique ManagementFactory.

L'ObjectName pour l'unique instance de cet MXBean est : java.lang:type=Memory

Cet MXBean permet d'obtenir plusieurs informations :

- la quantité de mémoire utilisée dans le tas de la JVM
- la quantité de mémoire utilisée hors du tas dans la JVM
- une estimation du nombre d'objets qui sont en attente de finalisation

Il permet aussi de demander une exécution du ramasse-miettes en invoquant sa méthode gc() et d'activer ou non les traces relatives à la gestion de la mémoire sur la sortie standard grâce à la méthode setVerbose().

Exemple :

```
package fr.jmdoudoux.dej.jmx.mxbeans;

import java.lang.management.ManagementFactory;
import java.lang.management.MemoryMXBean;
import java.lang.management.MemoryUsage;

public class TestMemoryMXB {

  public static void main(String[] args) {
    MemoryMXBean mbean = ManagementFactory.getMemoryMXBean();

    System.out.printf("Heap Memory Usage :\n%s \n",
      afficherMemoire(mbean.getHeapMemoryUsage()));
    System.out.printf("Non Heap Memory Usage :\n%s \n",
      afficherMemoire(mbean.getNonHeapMemoryUsage()));
    System.out.printf("Object pending finalization : %d\n",
      mbean.getObjectPendingFinalizationCount() );
    System.out.printf("isVerbose : %b\n", mbean.isVerbose() );
    System.out.println("Demande d'execution du ramasse miette");
    mbean.gc();
  }

  public static String afficherMemoire(MemoryUsage mu) {
    StringBuilder sb= new StringBuilder();
    sb.append("  init = "+mu.getInit()+"\n");
    sb.append("  used = "+mu.getUsed()+"\n");
    sb.append("  commited = "+mu.getCommitted()+"\n");
    sb.append("  max = "+mu.getMax()+"\n");
    return sb.toString();
  }
}
```

Résultat :

```
Heap Memory Usage :
  init = 0
  used = 223848
  commited = 5177344
  max = 66650112

Non Heap Memory Usage :
  init = 33718272
  used = 13066720
  commited = 34078720
  max = 121634816

Object pending finalization : 0
isVerbose : false
Demande d'execution du ramasse miette
```

La classe MemoryUsage encapsule des données sur l'occupation de la mémoire.

Le MemoryMXBean peut émettre des notifications de deux types :

- MEMORY_THRESHOLD_EXCEED : émise lorsque la quantité de mémoire occupée par un espace mémoire de la JVM a atteint un certain seuil
- MEMORY_COLLECTION_THRESHOLD_EXCEED : émise lorsque la quantité de mémoire occupée par un espace mémoire de la JVM a atteint un certain seuil après l'exécution du ramasse-miettes

Exemple :

```
package fr.jmdoudoux.dej.jmx.mxbeans;

import java.lang.management.ManagementFactory;
import java.lang.management.MemoryMXBean;
import java.lang.management.MemoryNotificationInfo;
import java.lang.management.MemoryPoolMXBean;
import java.lang.management.MemoryUsage;
import java.util.ArrayList;
import javax.management.Notification;
import javax.management.NotificationEmitter;
import javax.management.openmbean.CompositeData;

public class TestMemoryMXBNotif implements javax.management.NotificationListener {
  private static final int UN_MEGA_OCTET = 1024 * 1024;
  private static boolean stop = false;

  /**
   * Gestionnaire de traitement de notifications
   */
  public void handleNotification(Notification notif, Object handback) {
    System.out.println("\nReception d'une notification");
    System.out.println("Type : " + notif.getType());
    System.out.println("Message :  " + notif.getMessage());
    System.out.println("Source objectname : " + notif.getSource());
    CompositeData cd = (CompositeData) notif.getUserData();
    MemoryNotificationInfo memInfo = MemoryNotificationInfo.from( cd );
    System.out.println( "PoolName : " + memInfo.getPoolName() );
    MemoryUsage memoryUsage = memInfo.getUsage();
    System.out.println("\nEtat de la mémoire");
    System.out.println("  init : " + memoryUsage.getInit());
    System.out.println("  used : " + memoryUsage.getUsed());
    System.out.println("  committed : " + memoryUsage.getCommitted());
    System.out.println("  max : " + memoryUsage.getMax());
    // arret des traitements
    stop = true;
  }

  public static void main(String[] args) throws InterruptedException {
    // définition du seuil d'utilisation de la old generation
    for (MemoryPoolMXBean mpbean : ManagementFactory.getMemoryPoolMXBeans()) {
      if (mpbean.getName().equals("Tenured Gen")) {
        if (mpbean.isUsageThresholdSupported()) {
          mpbean.setUsageThreshold(4 * UN_MEGA_OCTET);
        }
        break;
      }
    }
    // abonnement du listerner auprès du MXBean
    MemoryMXBean mbean = ManagementFactory.getMemoryMXBean();
    ((NotificationEmitter) mbean).addNotificationListener(new TestMemoryMXBNotif(),
       null, null);
    // remplissage de la mémoire
    ArrayList<byte[]> list = new ArrayList<byte[]>();
    int i = 0;
    while (!stop) {
      System.out.printf("iteration %d \n", ++i);
      list.add(new byte[UN_MEGA_OCTET]);
    }
  }
}
```

Résultat :

```
iteration 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26
Reception d'une notification
Type : java.management.memory.threshold.exceeded
Message :  Memory usage
exceeds usage threshold
Source objectname : java.lang:type=Memory
PoolName : Tenured Gen
Etat de la mémoire
  init : 4194304
  used : 4348720
  committed : 5500928
  max : 61997056
```

Remarque : vu le fonctionnement des différents algorithmes utilisés par le ramasse-miettes, ces notifications ne doivent pas être utilisées pour détecter un manque de mémoire.

32.15.6. L'interface MemoryPoolMXBean

Cet MXBean permet de surveiller les espaces de mémoire de la JVM. Une JVM a plusieurs instances de cet MXBean, une pour chaque espace de mémoire utilisé.

Les instances de l'interface MemoryPoolMXBean sont obtenues en invoquant la méthode getMemoryPoolMXBeans() de la fabrique ManagementFactory.

L'ObjectName d'une instance de cet MXBean est de la forme : java.lang:type=MemoryPool, name=xxx

Cet MXBean permet d'obtenir plusieurs informations :

- une estimation de l'utilisation de l'espace mémoire
- les valeurs extrêmes de l'utilisation de l'espace mémoire
- l'utilisation de l'espace mémoire après la dernière exécution du ramasse-miettes

Exemple :

```
package fr.jmdoudoux.dej.jmx.mxbeans;

import java.lang.management.ManagementFactory;
import java.lang.management.MemoryPoolMXBean;
import java.util.List;
```

```
public class TestMemoryPoolMXB {
  public static void main(String[] args) {
    List<MemoryPoolMXBean> memoryPoolMXBeans = ManagementFactory.getMemoryPoolMXBeans();
    for (MemoryPoolMXBean mpBean : memoryPoolMXBeans) {
      System.out.printf("Memory Pool Name: %s\n", mpBean.getName());
      System.out.printf("  Type: %s\n", mpBean.getType().toString());
      System.out.printf("  isValid: %b\n", mpBean.isValid());
      for (String managerName : mpBean.getMemoryManagerNames()) {
        System.out.printf("  Memory manager name : %s\n", managerName);
      }
      System.out.println();
      System.out.printf("  Usage : %s\n", mpBean.getUsage().toString());
      System.out.printf("  PeakUsage : %s\n", mpBean.getPeakUsage().toString());
      boolean bUsageThSupported = mpBean.isUsageThresholdSupported();
      System.out.printf("  UsageThresholdSupport : %b\n", bUsageThSupported);
      if (bUsageThSupported) {
        System.out.printf("  UsageThreshold : %d\n", mpBean.getUsageThreshold());
        System.out.printf("  UsageThresholdCount : %d\n",
            mpBean.getUsageThresholdCount());
        System.out.printf("  isUsageThresholdExceeded : %b\n",
            mpBean.isUsageThresholdExceeded());
      }
      System.out.println();
      System.out.printf("  CollectionUsage: %s\n", mpBean.getCollectionUsage());
      boolean bCollectionUsageThSupported = mpBean.isCollectionUsageThresholdSupported();
      System.out.printf("  CollectionUsageThresholdSupport: %b\n",
                        bCollectionUsageThSupported);

      if (bCollectionUsageThSupported) {
        System.out.printf("  CollectionUsageThreshold: %d\n",
            mpBean.getCollectionUsageThreshold());
        System.out.printf("  CollectionUsageThresholdcount: %d\n",
            mpBean.getCollectionUsageThresholdCount());
        System.out.printf("  isCollectionUsageThresholdExceeded: %b\n",
            mpBean.isCollectionUsageThresholdExceeded());
      }
      System.out.println();
    }
  }
}
```

Le MemoryMXBean peut émettre des notifications de deux types :

- une notification émise lorsque la quantité de mémoire occupée par un espace mémoire de la JVM a atteint un certain seuil
- une notification émise lorsque la quantité de mémoire occupée par un espace mémoire de la JVM a atteint un certain seuil après l'exécution du ramasse-miettes. Cette notification n'est utilisable qu'avec certains algorithmes du ramasse-miettes.

32.15.7. L'interface OperatingSystemMXBean

Cet MXBean permet d'obtenir quelques informations sur le système d'exploitation.

Une instance de l'interface OperatingSystemMXBean est obtenue en invoquant la méthode getOperatingSystemMXBean() de la fabrique ManagementFactory.

L'ObjectName pour l'unique instance de cet MXBean est : java.lang:type=OperatingSystem

Cet MXBean permet d'obtenir plusieurs informations :

- le type de processeur
- le nombre de processeurs
- le nom du système d'exploitation
- la version du système d'exploitation
- une estimation de la charge du système

Exemple :

```
package fr.jmdoudoux.dej.jmx.mxbeans;

import java.lang.management.ManagementFactory;
import java.lang.management.OperatingSystemMXBean;

public class TestOperatingSystemMXB {

  public static void main(String[] args) {

    OperatingSystemMXBean osBean = ManagementFactory.getOperatingSystemMXBean();
    System.out.printf("Arch : %s\n", osBean.getArch());
    System.out.printf("Available processeurs : %d\n", osBean.getAvailableProcessors());
    System.out.printf("Name : %s\n", osBean.getName());
    System.out.printf("System Load Average : %f\n", osBean.getSystemLoadAverage());
    System.out.printf("Version : %s\n", osBean.getVersion());
  }
}
```

Résultat :

```
Arch : x86
Available processeurs : 2
Name : Windows Vista
System Load Average : -1,000000
Version : 6.0
```

Remarque : il est possible d'obtenir la plupart de ces informations soit par des propriétés de la JVM soit par une API.

Si la charge système est négative c'est que cette valeur n'est pas disponible.

32.15.8. L'interface RuntimeMXBean

Cet MXBean permet d'obtenir des informations sur la machine virtuelle.

Une instance de l'interface RuntimeMXBean est obtenue en invoquant la méthode getRuntimeMXBean() de la fabrique ManagementFactory.

L'ObjectName pour l'unique instance de cet MXBean est : java.lang:type=Runtime

Cet MXBean permet d'obtenir plusieurs informations sur la JVM.

Exemple :

```
package fr.jmdoudoux.dej.jmx.mxbeans;

import java.lang.management.ManagementFactory;
import java.lang.management.RuntimeMXBean;
import java.util.List;
import java.util.Map;

public class TestRuntimeMXB {

  public static void main(String[] args) {

    RuntimeMXBean rBean = ManagementFactory.getRuntimeMXBean();

    System.out.printf("Boot classpath : %s\n", rBean.getBootClassPath());
    System.out.printf("classpath : %s\n", rBean.getClassPath());
    System.out.println("Input argument :");
    List<String> arguments = rBean.getInputArguments();
    for (String arg : arguments) {
      System.out.println("  " + arg);
    }
    System.out.printf("Library path : %s\n", rBean.getLibraryPath());
    System.out.printf("Management spec version : %s\n", rBean
        .getManagementSpecVersion());
    System.out.printf("Name : %s\n", rBean.getName());
    System.out.printf("Spec name : %s\n", rBean.getSpecName());
    System.out.printf("Vendor : %s\n", rBean.getSpecVendor());
    System.out.printf("Spec version : %s\n", rBean.getSpecVersion());
    System.out.printf("StartTime : %d ms\n", rBean.getStartTime());
    System.out.println("System properties : %s\n");
    Map<String, String> props = rBean.getSystemProperties();
    for (String cle : props.keySet()) {
      System.out.println("  " + cle + " = " + props.get(cle));
    }
    System.out.printf("UpTime : %d ms\n", rBean.getUptime());
    System.out.printf("VmName : %s\n", rBean.getVmName());
    System.out.printf("VmVendor : %s\n", rBean.getVmVendor());
    System.out.printf("VmVersion : %s\n", rBean.getVmVersion());
    System.out.printf("isBootClassPathSupported : %b\n", rBean
        .isBootClassPathSupported());
  }
}
```

Remarque : il est possible d'obtenir la plupart de ces informations soit par des propriétés de la JVM soit par une API

32.15.9. L'interface ThreadMXBean

Cet MXBean permet d'obtenir des informations sur les threads de la JVM.

Une instance de l'interface ThreadMXBean est obtenue en invoquant la méthode getThreadMXBean() de la fabrique ManagementFactory.

L'ObjectName pour l'unique instance de cet MXBean est : java.lang:type=Threading

Cet MXBean permet d'obtenir de nombreuses informations sur les threads de la JVM.

Exemple :

```
package fr.jmdoudoux.dej.jmx.mxbeans;

import java.lang.management.ManagementFactory;
import java.lang.management.ThreadInfo;
import java.lang.management.ThreadMXBean;

public class TestThreadMXB {

  public static void main(String[] args) {

    Thread monThread = new Thread(new Runnable() {

      @Override
      public void run() {
        try {
          Thread.sleep(50);
        } catch (InterruptedException e) {
        }
      }
    });

    monThread.setName("Mon Thread");
    monThread.start();

    ThreadMXBean tBean = ManagementFactory.getThreadMXBean();

    System.out.printf("Current thead cpu time : %d\n", tBean
        .getCurrentThreadCpuTime());
    System.out.printf("Current thread user time : %d\n", tBean
        .getCurrentThreadUserTime());
    System.out.printf("Daemon thread count : %d\n", tBean
        .getDaemonThreadCount());
    System.out.printf("Peak thread count : %d\n", tBean.getPeakThreadCount());
    System.out.printf("Thread count : %d\n", tBean.getThreadCount());
    System.out.printf("Total Started Thread count : %d\n", tBean
        .getTotalStartedThreadCount());
    System.out.println("Liste des threads");
    ThreadInfo[] threads = tBean.dumpAllThreads(false, false);
    for (ThreadInfo ti : threads) {
      System.out.printf("Thead id  : %d\n", ti.getThreadId());
      System.out.printf("  Name : %s\n", ti.getThreadName());
      System.out.printf("  State : %s\n", ti.getThreadState());
      System.out.println("  Stack : ");
      StackTraceElement[] ste = ti.getStackTrace();
      for (StackTraceElement elt : ste) {
        System.out.printf("    %s.%s() - %d\n", elt.getClassName(), elt
            .getMethodName(), elt.getLineNumber());
      }
    }
    monThread.interrupt();
  }
}
```

Résultat :

```
Current thead cpu time : 62400400
Current thread user time : 62400400
Daemon thread count : 4
Peak thread count : 6
Thread count : 6
Total Started Thread count : 6
Liste des threads
Thead id  : 8
  Name : Mon Thread
  State : TIMED_WAITING
  Stack :
    java.lang.Thread.sleep() - -2
    fr.jmdoudoux.dej.jmx.mxbeans.TestThreadMXB$1.run() - 17
    java.lang.Thread.run() - 619
Thead id  : 5
  Name : Attach Listener
  State : RUNNABLE
  Stack :
Thead id  : 4
  Name : Signal Dispatcher
  State : RUNNABLE
  Stack :
Thead id  : 3
  Name : Finalizer
  State : WAITING
  Stack :
    java.lang.Object.wait() - -2
    java.lang.ref.ReferenceQueue.remove() - 116
    java.lang.ref.ReferenceQueue.remove() - 132
    java.lang.ref.Finalizer$FinalizerThread.run() - 159
Thead id  : 2
  Name : Reference Handler
  State : WAITING
  Stack :
    java.lang.Object.wait() - -2
    java.lang.Object.wait() - 485
    java.lang.ref.Reference$ReferenceHandler.run() - 116
Thead id  : 1
  Name : main
  State : RUNNABLE
  Stack :
    sun.management.ThreadImpl.dumpThreads0() - -2
    sun.management.ThreadImpl.dumpAllThreads() - 374
    fr.jmdoudoux.dej.jmx.mxbeans.TestThreadMXB.main() - 36
```

32.15.10. La sécurisation des accès à l'agent

La suite de cette section sera développée dans une version future de ce document

32.16. Des recommandations pour l'utilisation de JMX

Il faut être vigilant sur les ObjectNames associés aux MBeans en définissant des conventions de nommage notamment pour permettre leur organisation d'une façon hiérarchique dans les clients JMX. Il peut par exemple être intéressant d'utiliser un ou plusieurs attributs pour structurer cette hiérarchie et toujours avoir un attribut qui précise le type du MBean.

Pour améliorer la portabilité, il est préférable d'utiliser les Open MBeans ou les Open Types pour les structures de données des MBeans plutôt que des types personnalisés.

Il est préférable d'utiliser les MBeans standard lorsque cela est possible car ils sont faciles à écrire et à maintenir.

Le monitoring implique généralement la récupération et l'exploitation de différentes données : états, métriques, statistiques, ... Pour l'exploitation de ces données, leur affichage peut avoir des conséquences sur les fonctionnalités des MBeans.

Pour avoir une vision d'ensemble au niveau d'un domaine regroupant plusieurs applications et au niveau de tout ou partie du système d'informations, il faut agréger les données collectées de toutes les JVM.

Typiquement un client de monitoring interroge périodiquement le système pour afficher des données fraîches. Ce procédé peut être consommateur en terme de ressources (CPU, bande passante, ...). Les notifications peuvent être une solution dans certains cas mais mal utilisées cela peut être encore pire. Elles sont toujours plus intéressantes lorsqu'une donnée évolue peu : dans ce cas, il est préférable d'émettre une notification à chaque changement de valeur plutôt que de périodiquement récupérer une valeur qui sera fréquemment identique.

32.17. Des ressources

La page principale de la technologie JMX

https://www.oracle.com/java/technologies/javase/javamanagement.html

La documentation de JMX

https://www.oracle.com/java/technologies/javase/docs-jmx-jsp.html

La documentation de l'API JMX

https://docs.oracle.com/javase/1.5.0/docs/guide/jmx/

JMX best practice

https://www.oracle.com/java/technologies/javase/management-extensions-best-practices.html

mx4j est une implémentation open source de JMX

http://mx4j.sourceforge.net/

JbossMX est un implémentation open source de JMX

https://community.jboss.org/wiki/JBossMX

MC4J est un client JMX open source

https://sourceforge.net/projects/mc4j/

33. L'API Service Provider (SPI)

Chapitre 33

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

L'API Service Provider propose un moyen simple d'offrir un découplage entre des fournisseurs d'implémentation d'une interface d'un service et un consommateur d'une ou plusieurs de ses implémentations. Elle permet de découvrir et obtenir dynamiquement des instances de classes qui implémentent une interface particulière.

Java 6 propose une fonctionnalité pour découvrir et charger les implémentations d'un service : Service Provider Interface (SPI). La classe ServiceLoader permet de facilement mettre en oeuvre des fonctionnalités de type service qui permettent un découplage entre un fournisseur et un consommateur.

Son utilisation n'est pas forcement beaucoup répandue sauf pour quelques besoins particuliers comme les drivers JDBC.

La classe ServiceLoader permet de charger et d'utiliser une ou plusieurs implémentations de manière dynamique au runtime. Un cas d'usage typique est la mise en oeuvre de plugins dans une application ou l'utilisation optionnelle d'une ou plusieurs implémentations d'une fonctionnalité comme un cache par exemple.

Ce chapitre contient plusieurs sections :

♦ Introduction
♦ La mise en oeuvre
♦ La consommation d'un service
♦ Un exemple complet
♦ Les services de JPMS

33.1. Introduction

Un service offre des fonctionnalités définies grâce à une interface ou une classe abstraite pour laquelle il existe zéro, une ou plusieurs implémentations.

Un service provider ou fournisseur est une implémentation d'un service : c'est donc une classe qui implémente l'interface ou hérite de la classe abstraite qui définit le service.

33.1.1. La différence entre une API et une SPI

Basiquement, la différence entre API et SPI peut être résumé ainsi : le but d'une API est d'être invoquée, le but d'une SPI est d'être implémentée.

API est l'acronyme d'Application Programming Interface : une API est un ensemble de fonctionnalités (classes et interfaces) qu'il est possible d'utiliser pour atteindre un objectif par programmation.

L'ajout de fonctionnalités dans une API ne posent généralement pas de soucis au client qui l'utilise. Par contre, la modification ou la suppression de fonctionnalités doit être documentée et les clients qui l'utilise doivent être informés des impacts possibles.

SPI est l'acronyme de Service Provider Interface : c'est une technique de programmation qui permet la substitution de composants qui respectent une même interface.

Une SPI est un ensemble de classes et interfaces qu'il est possible d'étendre ou implémenter pour atteindre un objectif. C'est un moyen d'injecter, d'étendre ou de fournir des implémentations dédiées.

Une SPI est une API qui doit être implémentée ou étendue par un fournisseur tiers. Une SPI est généralement utilisée pour étendre un framework ou permettre le remplacement de composants. Elle permet à une API d'être évolutive en proposant des implémentations différentes ou supplémentaires.

L'ajout de fonctionnalités dans une SPI peut engendrer des incompatibilités dans les implémentations existantes.

Généralement, API et SPI sont séparées. Par exemple avec JDBC, l'interface Driver fait partie de son SPI. Son utilisation n'est pas obligatoire pour utiliser JDBC mais elle doit être implémentée par chaque fournisseur de pilote JDBC.

Parfois une classe ou une interface peut faire partie de l'API et de la SPI : c'est par exemple le cas pour l'interface Connection de l'API JDBC. C'est un des éléments primordiaux à utiliser pour accéder à une base de données et elle doit aussi être implémentée par le fournisseur d'un pilote JDBC.

Généralement l'utilisation de l'API ne nécessite pas d'utiliser des types de la SPI et vice versa.

Autre exemple avec JNDI : JNDI propose des interfaces et des classes pour rechercher des objets dans un contexte. Cette recherche se fait par défaut en utilisant la classe InitialContext. Cette classe utilise en interne des interfaces d'une SPI pour obtenir des implémentations spécifiques.

33.1.2. Un framework pour service provider

Un framework pour fournisseur de services (service provider) est un système dans lequel plusieurs fournisseurs de services implémentent un service, et le système permet à des consommateurs d'obtenir des instances des implémentations en découplant les fournisseurs des consommateurs.

Un framework pour fournisseur de services comporte plusieurs éléments principaux :

- Une description du service généralement sous la forme d'une interface
- Une API qui permet d'enregistrer les implémentations de fournisseurs
- Une API qui permet aux clients d'obtenir une ou plusieurs instances : soit celles par défaut, soit toutes soit éventuellement une en particulier
- Une SPI que les fournisseurs implémentent pour créer des instances de leur implémentation du service

Exemple avec JDBC :

- l'interface Connection décrit le service
- la méthode DriverManager.registerDriver() permet l'enregistrement d'un fournisseur
- la méthode DriverManager.getConnection() est l'API d'accès au service
- L'interface Driver est implémentée par le fournisseur de service

33.1.2.1. Un exemple d'implémentation basique

Une SPI peut être de différentes manières plus ou moins compliquées notamment en utilisant certains design patterns et/ou l'API Introspection.

L'exemple de cette section propose une implémentation très basique.

Le service est décrit dans l'interface Service

Exemple :

```
public interface Service {
  void afficher();
}
```

Deux implémentations sont proposées dont celle qui sera utilisée par défaut.

Exemple :

```
public class ServiceA implements Service {

  @Override
  public void afficher() {
    System.out.println("Service A");
  }
}
```

Exemple :

```
public class ServiceParDefaut implements Service {

  @Override
  public void afficher() {
    System.out.println("Service par défaut");
  }
}
```

Le fournisseur d'une implémentation du service doit implémenter l'interface Fournisseur. Elle ne définit qu'une seule méthode qui permet d'obtenir une instance de l'implémentation du service.

Exemple :

```
public interface Fournisseur {
  Service getService();
}
```

Deux implémentations sont proposées dont celle qui sera utilisée par défaut.

Exemple :

```
public class FournisseurParDefaut implements Fournisseur {

  @Override
  public Service getService() {
    return new ServiceParDefaut();
  }
}
```

Exemple :

```
public class FournisseurA implements Fournisseur {

  @Override
  public Service getService() {
    return new ServiceA();
  }
}
```

La classe Services permet d'enregistrer les fournisseurs et d'obtenir des instances de leurs implémentations du service.

Exemple :

```
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
```

```
public final class Services {

  public static final String NOM_FOURNISSEUR_PAR_DEFAUT = "<def>";

  private Services() { }

  private static final Map<String, Fournisseur> fournisseurs =
    new ConcurrentHashMap<String, Fournisseur>();

    public static void registerDefaultProvider(Fournisseur fournisseur) {
      registerProvider(NOM_FOURNISSEUR_PAR_DEFAUT, fournisseur);
    }

    public static void registerProvider(String nom, Fournisseur fournisseur){
      fournisseurs.put(nom, fournisseur);
    }

    public static Service getInstance() {
      return getInstance(NOM_FOURNISSEUR_PAR_DEFAUT);
    }

    public static Service getInstance(String nom) {
      Fournisseur p = fournisseurs.get(nom);
      if (p == null) {
        throw new IllegalArgumentException(
          "Impossible de trouver le fournisseur " + nom);
      }
      return p.getService();
    }
}
```

La classe Client permet de tester en enregistrant deux fournisseurs et en obtenant leurs implémentations du service.

Exemple :

```
public class Client {

  public static void main(String[] args) {
    Services.registerDefaultProvider(new FournisseurParDefaut());
    Services.registerProvider("A", new FournisseurA());

    Service service = Services.getInstance();
    service.afficher();

    service = Services.getInstance("A");
    service.afficher();
  }
}
```

33.1.3. Le chargement dynamique de classes

La manière dont Java utilise les classes introduit un niveau d'abstraction avec les ClassLoaders.

Un ClassLoader a la responsabilité de charger une classe dans la JVM, sans que la JVM sache d'où elle provient. La classe n'est d'ailleurs pas forcement lue à partir d'un fichier et peut par exemple l'être à partir du réseau ou de la mémoire dans le cas d'un proxy dynamique.

Dans ce cas, il est nécessaire d'utiliser un ClassLoader dédié qui est en mesure de charger dynamiquement une classe qui n'est pas incluse dans le classpath.

Historiquement, un ClassLoader est utilisé pour charger les classes. Une fois chargée, il est possible de créer des instances d'une classe.

Exemple :

```
MaClasse instance = Class.forName("fr.jmdoudoux.dej.MaClasse",
                      true, this.getClassLoader()).newInstance();
```

Il est nécessaire de connaître le nom pleinement qualifié de la classe à utiliser.

Il n'est pas possible d'obtenir à partir d'un ClassLoader la liste de toutes les classes contenues dans le classpath. Aucune méthode de la classe ClassLoader ne renvoie un tableau, une collection ou un Stream de classes. Il est possible d'obtenir un tableau des packages accessibles via le ClassLoader mais il n'est pas possible d'obtenir la liste des classes des packages.

Le JDK doit donc proposer un mécanisme dédié pour permettre de trouver les implémentations fournies dans le classpath et dans le module path à partir de Java 9.

33.1.4. L'utilisation par Java SE et Java EE

Le JDK lui-même utilise également ce mécanisme pour certaines fonctionnalités.

Une SPI est utilisée par plusieurs fonctionnalités du JRE (JDBC, JCE, JNDI, JAX-P, NIO, ...). Généralement par convention dans le JDK, les packages de classes qui pourront être étendues pour définir des services sont suffixées par spi.

Java SE Core contient plusieurs SPI notamment dans les packages :

- java.util.spi : AbstractResourceBundleProvider, CalendarDataProvider, CalendarNameProvider, CurrencyNameProvider, LocaleServiceProvider, ResourceBundleProvider, ResourceBundleControlProvider, TimeZoneNameProvider, ToolProvider
- java.sql (JDBC) : Driver
- java.text.spi : BreakIteratorProvider, CollatorProvider, DateFormatProvider, DateFormatSymbolsProvider, DecimalFormatSymbolsProvider, NumberFormatProvider
- java.net.spi : URLStreamHandlerProvider
- java.nio.channels.spi : AsynchronousChannelProvider
- java.nio.charset.spi : CharsetProvider
- java.nio.file.spi : FileSystemProvider
- javax.sql.rowset.spi : SyncFactory
- javax.security.auth.spi : LoginModule
- javax.naming.spi (JNDI)
- java.lang : System.LoggerFinder

Java EE propose aussi plusieurs SPI notamment :

- javax.ejb.spi : EJBContainerProvider
- javax.enterprise.inject.spi : Extension
- javax.json.spi : JsonProvider
- javax.json.bind.spi : JsonbProvider
- javax.persistence.spi : PersistenceProvider
- javax.validation.spi : ValidationProvider
- javax.websocket : ContainerProvider

L'intérêt de ce mécanisme est de proposer un découplage entre un consommateur et une ou plusieurs implémentations chargées dynamiquement.

Ce mécanisme est utilisé par exemple par la classe java.sql.DriverManager pour trouver les implémentations de l'interface java.sql.Driver. Historiquement, le chargement de la classe d'un driver JDBC impliquait l'exécution d'un bloc de code static qui enregistre le driver de type java.sql.Driver dans le DriverManager.

A partir de Java 6, il suffit d'ajouter le jar du driver proposé sous la forme d'un service dans le classpath et celui-ci sera automatiquement utilisable. Il n'est alors plus nécessaire de charger la classe en utilisant la méthode forName() de la classe Class. Ce mécanisme reste cependant utilisable pour des raisons de compatibilité.

La classe abstraite System.LoggerFinder de Java 9 peut être implémentée en tant que service. S'il existe une implémentation, la méthode System.getLogger() utilise le ServiceLoader pour la trouver. De cette manière, la journalisation n'est pas liée au JDK, ni à une bibliothèque à la compilation. Il suffit de fournir une implémentation du Logger à l'exécution et l'application, les bibliothèques utilisées par l'application et le JDK utiliseront tous cette implémentation de la journalisation.

33.2. La mise en oeuvre

Les fonctionnalités d'un service sont définies grâce à une interface ou une classe abstraite.

Un fournisseur de service est une implémentation d'un service. Plusieurs implémentations d'un service peuvent être proposées par un ou plusieurs fournisseurs (providers).

Un consommateur (consumer) ou client peut utiliser une ou plusieurs implémentations d'un service.

Les services permettent un découplage entre fournisseurs et consommateurs. Le consommateur ne connait que l'interface du service.

Ce type de fonctionnalité permet de proposer une ou plusieurs implémentations ou de mettre en place un système de plugins.

Le principe de fonctionnement est similaire à celui utilisé par un framework d'injection de dépendances.

Les implémentations déclarées dans les sous-répertoires META-INF/services ou dans des modules peuvent être trouvées par la classe ServiceLoader : elle permet d'obtenir la liste des implémentations d'un service et d'en obtenir des instances.

L'API SPI utilise quatre composants :

- Service : une fonctionnalité
- Service Provider Interface : une interface ou une classe abstraite qui définit le service
- Service Provider : une implémentation spécifique de la SPI. C'est une classe qui implémente l'interface du service ou hérite de la classe abstraite
- ServiceLoader : une classe dont le rôle est de découvrir et charger des instances de manière lazy

La mise en oeuvre de l'API repose sur 3 éléments :

- une description du service sous la forme d'une interface ou d'une classe abstraite. Chaque implémentation doit implémenter l'interface ou hériter de la classe abstraite
- un mécanisme pour déclarer des implémentations
- une API pour permettre à un consommateur d'obtenir les implémentations trouvées dynamiquement pour un service donné

L'implémentation d'un service peut être encapsulé :

- Dans un jar standard pour pouvoir être ajouté dans le classpath
- Dans un jar modulaire, à partir de Java 9, pour pouvoir être ajouté dans le module path et profiter de l'encapsulation renforcée des modules

La classe ServiceLoader permet de rechercher les services disponibles indifféremment dans le classpath et le module path et permet de charger les implémentations au runtime pour fournir des instances au consommateur.

A la compilation d'un consommateur, l'API ServiceLoader n'a besoin de connaitre que l'interface du service.

33.2.1. La définition d'un service

Un service est décrit dans un type : une interface ou une classe abstraite.

Un service peut avoir autant de méthodes que le requière ses fonctionnalités. Chacune de ces méthodes pourra être invoquées lors de l'obtention d'une instance du service.

Exemple :

```
package fr.jmdoudoux.dej.spi;

public interface MonService {

  public void afficher();
}
```

33.2.2. L'implémentation d'un service

Un service provider est une implémentation d'un service.

La classe d'implémentation d'un service doit être publique et ne peut pas être une classe interne.

Un fournisseur peut proposer une ou plusieurs implémentations du service sous la forme de classes concrètes qui implémentent l'interface ou héritent de la classe abstraite.

Exemple dans un jar serviceimplA

Exemple :

```
package fr.jmdoudoux.dej.spi;

public class MonServiceImplA implements MonService {

  @Override
  public void afficher() {
    System.out.println("MonServiceImplA");
  }
}
```

Exemple dans un jar serviceimplB

Exemple :

```
package fr.jmdoudoux.dej.spi;

public class MonServiceImplB implements MonService{

  @Override
  public void afficher() {
    System.out.println("MonServiceImplB");
  }
}
```

33.2.3. Le fichier de configuration du Provider

La configuration des implémentations du service provider se fait dans un fichier texte dans le sous-répertoire META-INF/services.

Pour chaque service dont une ou plusieurs implémentations sont proposées, il faut définir un fichier texte qui est un fichier de description des implémentations fournies dans le sous-répertoire META-INF/services

Le nom de ce fichier doit correspondre au nom pleinement qualifié du type du service. Il doit contenir le nom pleinement qualifié de chaque implémentation, chacune sur une ligné dédiée.

Ce fichier doit contenir le nom pleinement qualifié de la ou des classes d'implémentation du service, chacune sur une ligne dédiée.

Plusieurs règles doivent être appliquées :

- Le fichier doit être encodé en UTF-8.
- Chaque nom de classe doit être sur sa propre ligne.
- Les noms de classes en doublon ne sont pris en compte qu'une seule fois.
- Les espaces et les tabulations et les lignes vides sont ignorées.

Il est possible d'utiliser des commentaires qui débutent par un caractère dièze '#'. Tous les caractères qui suivent un caractère # sont ignorés.

Exemple dans un jar serviceimplA, le fichier META-INF/services/fr.jmdoudoux.dej.spi.MonService

Résultat :

```
fr.jmdoudoux.dej.spi.MonServiceImplA
```

Exemple dans un jar serviceimplB, le fichier META-INF/services/fr.jmdoudoux.dej.spi.MonService

Résultat :

```
fr.jmdoudoux.dej.spi.MonServiceImplB
```

33.3. La consommation d'un service

Le consommateur n'a pas à connaître l'emplacement de l'instance obtenue : c'est la classe ServiceLoader qui se charge de trouver et de fournir des instances des implémentations disponibles.

Un consommateur peut obtenir une instance :

- à partir de l'Iterator du ServiceLoader
- à partir d'un objet de type ServiceLoader.Provider contenu dans un Stream ayant comme source le ServiceLoader

33.3.1. Le déploiement d'un service dans le classpath

Généralement, la ou les classes d'implémentation sont packagées dans un fichier jar qu'il faut ajouter dans le classpath.

La classe d'implémentation d'un service indiqué dans un fichier de configuration peut se trouver dans le même fichier JAR que le fichier de configuration ou dans un fichier JAR différent.

La classe d'implémentation doit pouvoir être chargée par le ClassLoader initialement utilisé pour trouver la liste des implémentations disponibles.

La JVM scanne les jars dans le classpath à la recherche des fichiers de configuration présents dans les sous-répertoires META-INF/services et enregistre les classes trouvées dans un registre.

Comme le chargement est dynamique, pour permettre la prise en compte d'une nouvelle implémentation d'un service, il suffit de l'ajouter ou de le retirer dans le classpath sans avoir à modifier le code.

33.3.2. La classe ServiceLoader

Pour trouver et obtenir une instance d'une ou de toutes les implémentations d'un service proposées par un ou plusieurs fournisseurs, il faut utiliser la classe java.util.ServiceLoader. La classe java.util.ServiceLoader<S> permet de découvrir dynamiquement et charger au runtime les implémentations d'un service de type S.

La classe ServiceLoader du JDK permet d'injecter une instance dynamiquement au runtime sans avoir recours à un framework d'injection de dépendances.

La classe ServiceLoader a été introduite en Java 6 et mise à jour en Java 9. Elle est final et elle implémente l'interface Iterable.

Les implémentations trouvées par le ServiceLoader doivent être préalablement enregistrées grâce au mécanisme particulier détaillé précédemment.

La classe ServiceLoader possède plusieurs méthodes :

Méthode	Rôle
Optional<S> findFirst()	Obtenir la première instance disponible s'il en existe une (depuis Java 9)
Iterator<S> iterator()	Renvoyer un Iterator pour obtenir les instances du service
static <S> ServiceLoader<S> load(Class<S> service)	Renvoyer une instance pour le type fourni en paramètre. Les classes sont chargées en utilisant le ClassLoader du contexte du thread courant. Cette méthode est équivalente à ServiceLoader.load(service, Thread.currentThread().getContextClassLoader())
static <S> ServiceLoader<S> load(Class<S> service, ClassLoader loader)	Renvoyer une instance pour le type fourni en paramètre chargé via le ClassLoader précisé
static <S> ServiceLoader<S> load(ModuleLayer layer, Class<S> service)	Renvoyer une instance pour le type fourni en paramètre. Les classes sont chargées uniquement parmi celles contenues dans le ModuleLayer. Aucun service n'est donc chargé à partir de l'unnamed module. Contrairement aux autres méthodes, le type du service est en second paramètre (depuis Java 9)
static <S> ServiceLoader<S> loadInstalled(Class<S> service)	Renvoyer une instance pour le type fourni en paramètre. Les classes sont chargées en utilisant le ClassLoader de la plateforme. Cette méthode est équivalente à ServiceLoader.load(service, ClassLoader.getPlatformClassLoader()). Cette méthode ne permet donc pas de charger des services contenus dans le classpath ou le module path
void reload()	Vider le cache et recharger les services
Stream<ServiceLoader.Provider<S>> stream()	Renvoyer un Stream<ServiceLoader.Provider> pour traiter de manière lazy les services (Depuis Java 9)
String toString()	Renvoyer une description du service

33.3.2.1. L'obtention d'un ServiceLoader

Pour obtenir une instance de type ServiceLoader, il faut utiliser les méthodes load() ou loadInstalled() en leur passant en paramètre le type du service. Plusieurs surcharges sont proposées pour utiliser le ClassLoader par défaut ou celui fournit en paramètre.

La classe ServiceLoader propose plusieurs méthodes qui sont des fabriques pour charger les implémentations du service :

- load(Class<S>) : trouver les implémentations du service et les charger avec le context classloader du thread courant
- load(Class<S>, ClassLoader) : trouver les implémentations du service et les charger avec le classloader fourni en paramètre
- loadInstalled(Class<S>) : trouver les implémentations du service et les charger avec l'extension classloader

La méthode statique load() de la classe ServiceLoader permet obtenir une instance de type ServiceLoader qui permet de parcourir la liste des implémentations enregistrées pour l'interface du service passée en paramètre.

Exemple :

```
ServiceLoader<MonService> services = ServiceLoader.load(MonService.class);
```

La méthode load(), qui attend en paramètre le type du service, recherche les implémentations et charge les types en utilisant le ClassLoader par défaut. Une autre surcharge permet de préciser le ClassLoader à utiliser : celui-ci peut être utilisé pour personnaliser la recherche des implémentations du service.

La méthode loadInstalled() cherche des implémentations dans le répertoire d'extension du JRE, le sous-répertoire jre/lib/ext. Les jars qu'il contient peuvent être utilisés par toutes les applications exécutées par le JRE.

33.3.2.2. La recherche des implémentations par le ServiceLoader

Un ServiceLoader permet de trouver et instancier des implémentations d'un service.

Le ServiceLoader n'est capable de détecter que les implémentations définies dans les descripteurs de modules avec provides et dans le classpath avec des fichiers dans le sous-répertoire META-INF/services.

Les méthodes iterator() et stream() de la classe ServiceLoader recherche les fournisseurs d'implémentations dans le classpath et à partir de Java 9 aussi dans les descripteurs de module présents dans le module-path.

33.3.2.2.1. La recherche dans le classpath ou les unamed modules

Les fournisseurs de services dans le classpath sont localisés si leurs noms de classe sont listés dans les fichiers de configuration du fournisseur trouvés par la méthode getResources() du ClassLoader.

Le ServiceLoader utilise la méthode getResources() du ClassLoader pour trouver le fichier dont le nom correspond au nom pleinement qualifié du service dans le sous-répertoire META-INF/services. La lecture du fichier permet d'obtenir le ou les noms pleinement qualifiés des classes d'implémentation du service.

L'ordre est basé sur l'ordre dans lequel la méthode getResources() du ClassLoader trouve les fichiers de configuration du service et dans celui-ci sur l'ordre dans lequel les noms de classe sont listés dans le fichier.

Les fournisseurs de service présents dans un fichier de configuration, sont ignorés lorsque leurs implémentations sont dans des modules nommées. Cela permet d'éviter les doublons qui se produiraient autrement lorsqu'un module nommé possède à la fois une directive "provides" et un fichier de configuration qui mentionne le même fournisseur de services.

33.3.2.2.2. La recherche dans les modules nommés

Le ServiceLoader recherche d'abord les modules définis par le ClassLoader puis recherche en remontant la hiérarchie des ClassLoaders jusqu'au bootstrap ClassLoader. L'ordre des modules dans le même ClassLoader n'est pas défini.

Si un module déclare plus d'un fournisseur, les fournisseurs sont recherchés dans l'ordre dans lequel le descripteur du module énumère les fournisseurs.

Les fournisseurs ajoutés dynamiquement en utilisant la méthode redefineModule() de l'interface java.lang.instrument.Instrumentation sont toujours situés après les fournisseurs déclarés dans le descripteur de module.

33.3.2.3. L'obtention des implémentations par le ServiceLoader

Le ServiceLoader permet d'obtenir les implémentations du service trouvées. Il est possible d'utiliser la première implémentation trouvée ou de parcourir l'ensemble des implémentations trouvées pour un service donné.

Les implémentations d'un service sont chargées et instanciées de manière lazy donc uniquement lorsqu'on en a besoin.

Chaque implémentation doit proposer un constructeur par défaut, donc sans argument, qui sera utilisé par le ServiceLoader pour créer dynamiquement des instances des implémentations trouvées via l'API Reflexion.

Un ServiceLoader contient un cache des implémentations obtenues pendant leur parcours.

Pour obtenir les différentes implémentations utilisables, il est possible d'utiliser :

- la méthode iterator() qui renvoie un Iterator sur les instances des implémentations trouvées
- la méthode stream() qui permet d'obtenir une Stream<Provider<S>> pour rechercher la ou les implémentations à utiliser de manière lazy

La classe ServiceLoader implémente l'interface Iterable, ce qui permet un parcours des différentes implémentations trouvées.

La méthode iterator() permet d'obtenir un Iterator pour parcourir les implémentations trouvées par la JVM et d'obtenir des instances qu'il sera possible d'utiliser. Ces instances sont mises en cache pour des raisons de performances. La méthode iterator() renvoie un Iterator : chaque instance obtenue renvoie d'abord les instances mises en cache lors des invocations précédentes puis trouve et instancie de manière lazy les implémentations restantes après les avoir mises en cache.

L'utilisation de l'Iterator obtenu peut lever une exception de type ServiceConfigurationError en cas de soucis. Les invocations suivantes de l'Iterator ne sont alors pas garanties de succès.

L'Iterator renvoyé par la méthode iterator() ne permet pas de retirer un élément : l'invocation de la méthode remove() lève une exception de type UnsupportedOperationException.

L'utilisation de l'Iterator obtenu en invoquant la méthode iterator() permet facilement de parcourir les instances. Cependant son inconvénient est qu'elle oblige la création d'une instance de chacune des implémentations trouvées même si elle n'est pas utilisée.

Exemple (code Java 6) :

```
    ServiceLoader<MonService> loader = ServiceLoader.load(MonService.class);
    for (MonService service : loader) {
       // ... utilisation de l'instance obtenue
    }
```

Depuis Java 9, la méthode stream() renvoie un Stream<ServiceLoader.Provider> permettant de manipuler de manière lazy les implémentations du service. Elle permet de parcourir les services de manière lazy sans avoir à créer une instance de ces services.

Les implémentations déjà chargées dans le cache sont traitées dans leur ordre de chargement puis les implémentations restantes sont traitées au besoin.

Chacune de ces implémentations est encapsulée dans une instance de type de l'interface ServiceLoader.Provider. Cette interface propose deux méthodes :

Méthode	Rôle
S get()	Obtenir une instance du provider
Class< ? extends S> type()	Obtenir le type du provider

Pour obtenir une instance de l'implémentation du service, il faut obligatoirement invoquer la méthode get() de la classe Provider. Si l'implémentation du service ne peut être chargée, alors une exception de type ServiceConfigurationError est levée.

La méthode stream() renvoie un Stream qui permet de sélectionner ou filtrer les implémentations trouvées de manière lazy donc sans avoir à en créer une instance.

Par exemple pour obtenir les instances dont le type se termine par « B »

Exemple (code Java 9) :

```
    List<MonService> ser = services.stream()
        .filter(p -> p.type().getName().endsWith("B"))
        .map(Provider::get)
        .collect(Collectors.toList());
```

Ou pour obtenir les instances dont le type n'est pas annoté avec @Deprecated

Exemple (code Java 9) :

```
    List<MonService> ser = services.stream()
        .filter(p -> !p.type().isAnnotationPresent(Deprecated.class))
        .map(Provider::get)
        .collect(Collectors.toList());
```

Un consommateur ne connait que l'interface ou la classe qui définit le service : elle ne connait pas la ou les implémentations qu'elle va utiliser au travers du type du service. Un consommateur utilise un ServiceLoader pour charger une ou plusieurs implémentations. Cela implique que le client sache :

- différencier de multiples fournisseurs s'il ne doit en choisir qu'un ou au pire prendre la première implémentation si le choix d'une en particulier n'est pas pertinent
- être capable de gérer le cas où aucune implémentation n'est disponible : cela peut être un cas d'erreur par exemple

A partir de Java 9, il est aussi possible d'obtenir uniquement la première implémentation trouvée en invoquant la méthode first() qui renvoie un Optional<S>.

Exemple (code Java 9) :

```
    Optional<MonService> monService = ServiceLoader
                .load(MonService.class)
                .findFirst();
    monService.ifPresent(MonService::afficher);
```

33.3.2.4. L'utilisation d'un cache par le ServiceLoader

Les implémentations sont trouvées et chargées à la demande. La classe ServiceLoader possède un cache des implémentations qui ont été chargées. Le cache est rempli à la première recherche d'un service.

Le ServiceLoader utilise ce cache : l'Iterator renvoie d'abord les éléments du cache, dans l'ordre dans lequel ils ont été chargés. Il charge et instancie ensuite tous les fournisseurs de services restants, en ajoutant chacun d'eux au cache.

Le cache peut être réinitialisé en invoquant la méthode reload() de la classe ServiceLoader. Suite à l'invocation de la méthode reload(), le cache de la liste des classes d'implémentation sera reconstruit lors du parcours suivant son invocation.

Si la méthode reload() est invoquée, alors il ne faut plus utiliser d'Iterator obtenu pour le service avant l'invocation de la méthode reload() sinon ces méthodes lève une exception de type ConcurrentModificationException.

La méthode reload() est par exemple utile dans le cas où un nouveau service est ajouté dans l'environnement d'exécution.

33.3.3. Les exceptions lors de l'utilisation de la classe ServiceLoader

Une exception de type ServiceConfigurationError peut être levée par les méthodes hasNext() et next() de l'Iterator du ServiceLoader si une erreur survient durant la recherche, le chargement ou la création d'une instance d'un service. Cette exception peut aussi être levée lors de l'exploitation du Stream retournée par la méthode stream().

Plusieurs situations peuvent engendrer une erreur, notamment :

- l'implémentation ne propose pas de constructeur par défaut ou une fabrique static nommée provider(),
- l'implémentation n'est pas assignable au type du service,
- le constructeur par défaut lève une exception,
- le type retournée par la fabrique provider() n'est pas assignable au type du service,
- la fabrique provider() renvoie null ou lève une exception,
- le fichier de configuration des implémentations du service ne respecte pas le format,
- une exception de type IOException est levée lors de la lecture du fichier de configuration

33.3.4. Les limitations de la classe ServiceLoader

La classe ServiceLoader n'est pas thread-safe.

La classe ServiceLoader est final : il n'est donc pas possible de créer une classe fille dans laquelle, une ou plusieurs de ces méthodes soient redéfinies pour changer son comportement.

Il est cependant possible de préciser un ClassLoader dédié pour trouver et charger les classes.

La classe ServiceLoader ne permet pas de détecter qu'une nouvelle implémentation d'un fournisseur est ajoutée à l'exécution. La classe ServiceLoader ne peut pas nous informer de l'ajout d'une nouvelle implémentation après la première recherche des implémentations d'un service.

La classe ServiceLoader est disponible depuis Java 1.3 pour une utilisation interne dans le JDK et pour une utilisation publique depuis la version 6 de Java.

33.4. Un exemple complet

Il faut définir l'interface du service dans un jar nommé monservice

Exemple :

```
package fr.jmdoudoux.dej.service;

public interface MonService {

   String traiter(String libelle);
}
```

Il faut créer une implémentation de l'interface MonService dans un jar nommé monservice_impla

Exemple :

```
package fr.jmdoudoux.dej.service.impla;

import fr.jmdoudoux.dej.service.MonService;

public class MonServiceImplA implements MonService {

  public MonServiceImplA() {
    System.out.println("Creation instance "+this.getClass().getName());
  }

  @Override
  public String traiter(String libelle) {
    return "MonServiceImplA : " + libelle;
  }
}
```

Il faut aussi ajouter dans ce jar, les sous-répertoires META-INF/services qui contient un fichier nommé du nom pleinement qualifié de l'interface du service, soit fr.jmdoudoux.dej.service.MonService. Ce fichier texte doit uniquement contenir le nom pleinement qualifié de l'implémentation du service :

Résultat :

```
fr.jmdoudoux.dej.service.impla.MonServiceImplA
```

Dans un jar nommé MonConsommateur, il faut créer une classe qui va utiliser une implémentation de l'interface MonService obtenue en utilisation la classe ServiceLoader.

Exemple (code Java 6) :

```
package fr.jmdoudoux.dej.consommateur;

import java.util.NoSuchElementException;
import java.util.ServiceLoader;

import fr.jmdoudoux.dej.service.MonService;

public class MonConsommateur {

  public static void main(String[] args) {
    ServiceLoader<MonService> loader;
    loader = ServiceLoader.load(MonService.class);
    MonService service = loader.iterator().next();
    if(service != null) {
      String message = service.traiter("mon libelle");
      System.out.println(message);
    } else {
      throw new NoSuchElementException("Aucune implementation de MonService");
    }
  }
}
```

Pour exécuter cette classe, il faut ajouter dans le classpath les jar monservice et monservice_impla

Résultat :

```
Creation instance fr.jmdoudoux.dej.service.impla.MonServiceImplA
MonServiceImplA : mon libelle
```

Si le jar monservice_impla est retiré du classpath, alors aucune implémentation du service n'est trouvée par le ServiceLoader.

Résultat :

```
Exception in thread "main" java.util.NoSuchElementException
        at java.base/java.util.ServiceLoader$2.next(ServiceLoader.java:1308)
        at java.base/java.util.ServiceLoader$2.next(ServiceLoader.java:1296)
        at java.base/java.util.ServiceLoader$3.next(ServiceLoader.java:1394)
        at fr.jmdoudoux.dej.consommateur.MonConsommateur.main(MonConsommateur.java:13)
```

Il est possible de créer dans un jar monservice_implb contenant une seconde implémentation de l'interface du Service.

Exemple :

```
package fr.jmdoudoux.dej.service.implb;

import fr.jmdoudoux.dej.service.MonService;

public class MonServiceImplB implements MonService {

  public MonServiceImplB() {
    System.out.println("Creation instance "+this.getClass().getName());
  }
  @Override
  public String traiter(String libelle) {
    return "MonServiceImplB : " + libelle;
  }
}
```

Le jar doit aussi contenir un fichier META-INF/services/fr.jmdoudoux.dej.service.MonService contenant :

Résultat :

```
fr.jmdoudoux.dej.service.implb.MonServiceImplB
```

En exécutant la classe MonConsommateur avec le jar monservice_implb dans le classpath

Résultat :

```
Creation instance fr.jmdoudoux.dej.service.implb.MonServiceImplB
MonServiceImplB : mon libelle
```

Le consommateur peut aussi vouloir utiliser toutes les instances trouvées dans le classpath. Il suffit simplement itérer sur l'instance de type ServiceLoader obtenue

Exemple (code Java 6) :

```
package fr.jmdoudoux.dej.consommateur;

import java.util.NoSuchElementException;
import java.util.ServiceLoader;

import fr.jmdoudoux.dej.service.MonService;

public class MonConsommateur {

  public static void main(String[] args) {
    ServiceLoader<MonService> loader;
    loader = ServiceLoader.load(MonService.class);

    for (MonService service : loader) {
      String message = service.traiter("mon libelle");
      System.out.println(message);
    }
  }
}
```

En exécutant la classe MonConsommateur avec les jars monservice_impla et monservice_implb dans le classpath, les deux instances sont utilisées.

Résultat :

```
Creation instance fr.jmdoudoux.dej.service.implb.MonServiceImplB
MonServiceImplB : mon libelle
Creation instance fr.jmdoudoux.dej.service.impla.MonServiceImplA
MonServiceImplA : mon libelle
```

Il est possible de créer une classe de type Provider dont le rôle est de faciliter l'obtention d'une instance d'un service.

33.5. Les services de JPMS

Un module est un artefact contenant une description de sa configuration et qui expose des types, encapsule les classes d'implémentation et peut proposer des services.

Un service dans JPMS est un type que le code d'un module souhaite utiliser, dont au moins un autre module propose une implémentation.

A partir de Java 9, il est possible d'utiliser des services dans des modules :

- généralement un module qui contient la définition d'un service,
- un ou plusieurs modules qui fournissent des implémentations de ce service,
- un ou plusieurs modules qui utilisent une ou plusieurs implémentations du service

Le système de modules propose une solution élégante pour mettre en oeuvre le découplage entre le ou les fournisseurs et le consommateur d'un service. Cette solution repose sur l'API ServiceLoader de Java 6.

Java 9 permet toujours l'utilisation de fichiers dans le sous-répertoire META-INF/services mais il propose aussi un mécanisme supplémentaire pour les modules de déclarations des services.

Le système de module de Java permet de facilement déclaration la fourniture et la consommation d'un service dans le descripteur de module.

Ce mécanisme utilise une syntaxe particulière pour facilement déclarer la fourniture et la consommation d'un service dans le descripteur de module :

- pour déclarer une implémentation à enregistrer
- pour définir un service requis par un module

Ce nouveau mécanisme ne repose donc plus sur un fichier texte mais est contenu dans le descripteur de module

L'avantage de l'intégration de la définition dans le code Java est que le compilateur peut faire des vérifications notamment sur les types utilisés.

33.5.1. Le module qui contient l'interface du service

Le module contient la définition du service sous la forme d'une interface.

Exemple :

```
package fr.jmdoudoux.dej.service;
public interface MonService {
   String traiter(String libelle);
}
```

Le descripteur de module exporte le package qui contient l'interface du service car celle-ci devra être accessible par les classes d'implémentation et les consommateurs.

Exemple (code Java 9) :

```
module MonService {
  exports fr.jmdoudoux.dej.service;
}
```

33.5.2. Un fournisseur de service dans un module

Dans un module qui fournit une implémentation d'un service, il faut :

- proposer une ou plusieurs implémentations du service
- et déclarer dans le descripteur de module qu'il fournit une ou plusieurs implémentations d'un service

33.5.2.1. L'implémentation du service dans le module

L'implémentation doit respecter plusieurs règles :

- elle doit être une classe concrète qui implémente l'interface ou hérite de la classe abstraite qui définit le service
- elle ne peut pas être une classe interne
- elle doit obligatoirement avoir un constructeur sans paramètre ou une méthode qui est une fabrique avec comme nom provider, sans paramètre et qui renvoie une instance du type du service

Une instance d'une implémentation d'un service sera créée en invoquant son constructeur par défaut en utilisant l'API Reflection.

Si une implémentation d'un service est déployée dans un automatic module (jar standard mis dans le module path), une instance du service sera aussi obtenue en invoquant son constructeur par défaut.

Dans Java 9, le fournisseur n'a pas l'obligation d'implémenter l'interface sous réserve qu'elle propose une méthode publique statique nommée provider() sans paramètre. Cette méthode est une fabrique dont le but est de fournir une instance du service qui implémente cette interface.

Une implémentation d'un service contenue dans un module peut ainsi avoir un contrôle sur la manière dont l'API ServiceLoader va en créer une instance en proposant une fabrique sous la forme d'une méthode avec une signature spécifique :

public static type_du_service provider()

Cette méthode doit donc :

- être public et static
- retourner l'instance dont le type doit pouvoir être assignable à celui de l'interface ou de la classe abstraite du service
- se nommer provider
- ne pas avoir de paramètre

La classe qui contient une fabrique nommée provider n'est pas obligée d'implémenter l'interface ou d'hériter de la classe du service.

Remarque : l'utilisation d'une fabrique provider n'est possible que dans un jar modulaire mis dans le module path

33.5.2.2. La déclaration dans le descripteur de module

Un module qui propose une implémentation d'un service doit le déclarer dans son descripteur de module.

Dans le descripteur de module, il faut deux éléments :

- une dépendance vers le module qui définit l'interface du service
- une déclaration de la classe d'implémentation du service en utilisant la directive provides suivi de l'interface du service, puis de la directive with et enfin de la classe d'implémentation

Il n'est pas nécessaire et est même fortement recommandé de ne pas exporter le package contenant l'implémentation d'un service. Cela permet de renforcer le découplage.

Pour permettre l'enregistrement de l'implémentation d'un service encapsulé dans un jar modulaire, il faut utiliser la directive provides dans le descripteur du module. La directive provides permet d'indiquer que le module est un fournisseur pour une implémentation du service. Sa syntaxe est de la forme :

provides interface_service with interface_implementation;

La déclaration se fait avec une directive provides suivi du nom pleinement qualifié du type de l'interface, de l'instruction with puis du nom pleinement qualifié de la classe d'implémentation du service : elle fournit donc les informations nécessaires pour enregistrer l'implémentation dans le registre et ainsi permettre à l'API ServiceLoader de la trouver.

Exemple (code Java 9) :

```
import fr.jmdoudoux.dej.spi.impla.MonServiceImplA;
import fr.jmdoudoux.dej.spi.service.MonService;

module fr.jmdoudoux.dej.spi.impla {

  requires fr.jmdoudoux.dej.spi.service;

  provides MonService with MonServiceImplA;
}
```

Un module peut proposer plusieurs implémentations : dans ce cas, il faut préciser chacun des noms pleinement qualifiés de chaque classe dans la clause with en les séparant par une virgule.

Exemple (code Java 9) :

```
import fr.jmdoudoux.dej.spi.impla.MonServiceImplA;
import fr.jmdoudoux.dej.spi.impla.MonServiceImplC;
import fr.jmdoudoux.dej.spi.service.MonService;

module fr.jmdoudoux.dej.spi.impla {

  requires fr.jmdoudoux.dej.spi.service;

  provides MonService with MonServiceImplA, MonServiceImplC;
}
```

Un module peut aussi proposer des implémentations pour plusieurs services différents.

Un autre module peut contenir une ou plusieurs autres implémentations du service.

Exemple (code Java 9) :

```
import fr.jmdoudoux.dej.spi.implb.MonServiceImplB;
import fr.jmdoudoux.dej.spi.service.MonService;

module fr.jmdoudoux.dej.spi.implb {

  requires fr.jmdoudoux.dej.spi.service;

  provides MonService with MonServiceImplB;
}
```

33.5.3. La consommation de services d'un module

L'application cliente n'a pas à connaitre directement la ou les implémentations qu'elle va utiliser. Il suffit simplement de mettre le ou les modules contenant une implémentation dans le module-path. Ces implémentations seront détectées automatique grâce aux informations contenues dans les descripteurs de modules.

Deux actions doivent être mise en oeuvre pour consommer un service :

- déclarer l'utilisation du service en utilisant la classe uses dans le descripteur du module,
- utiliser la classe ServiceLoader pour trouver et obtenir une ou plusieurs instances d'implémentation du service

33.5.3.1. La déclaration dans le descripteur de module

Un module qui a besoin d'utiliser un service doit le déclarer dans son descripteur de module.

Dans le descripteur d'un module dont une classe va consommer un service, il faut :

- ajouter la dépendance vers le module de l'interface du service,
- déclarer la consommation du service en utilisant l'instruction uses suivi de l'interface du service

Cette déclaration se fait à l'aide de la directive uses suivi du nom de l'interface ou de la classe abstraite qui définit les fonctionnalités du service.

Exemple (code Java 9) :

```
import fr.jmdoudoux.dej.spi.service.MonService;

module fr.jmdoudoux.dej.spi {
  exports fr.jmdoudoux.dej.spi;

  requires fr.jmdoudoux.dej.spi.service;

  uses MonService;
}
```

La directive uses est suivie du nom pleinement qualifié du type de l'interface du service.

Par défaut, il n'est pas nécessaire de déclarer la dépendance vers le ou les modules d'implémentation du service car les implémentations sont gérées en interne dans le ServiceLoader.

33.5.3.2. L'utilisation de la classe ServiceLoader

Une classe qui souhaite utiliser une implémentation d'un service doit utiliser la classe ServiceLoader.

L'utilisation de la classe ServiceLoader se fait comme vu dans la section dédiée précédente.

Son implémentation interne permet de trouver des fournisseurs dans le classpath et dans les descripteurs de modules dans le module path.

33.5.3.3. Faciliter la consommation d'un service

Une possibilité intéressante, surtout d'un point de vue réutilisation par plusieurs consommateurs, est de définir le module de l'interface du service comme étant lui-même le SPI (Service Provider Interface).

Dans ce cas, il faut :

- proposer une méthode statique, qui peut être définie dans l'interface du service,
- rajouter la clause uses suivi de l'interface dans le descripteur du module définissant le service

Exemple (code Java 9) :

```
import fr.jmdoudoux.dej.spi.service.MonService;

module fr.jmdoudoux.dej.spi.service {
  exports fr.jmdoudoux.dej.spi.service;

  uses MonService;
}
```

Exemple (code Java 9) :

```
package fr.jmdoudoux.dej.spi.service;

import java.util.Optional;
import java.util.ServiceLoader;

public interface MonService {

  public void afficher();

  public static Optional<MonService> obtenirPremiereInstance() {
    return ServiceLoader.load(MonService.class).findFirst();
  }
}
```

Dans les modules consommateur, la directive uses n'est plus nécessaire puisque c'est le module du service lui-même et il suffit alors d'invoquer la méthode static du service.

Exemple (code Java 9) :

```
module fr.jmdoudoux.dej.spi {
  exports fr.jmdoudoux.dej.spi;

  requires fr.jmdoudoux.dej.spi.service;
}
```

Il suffit alors d'invoquer la méthode static du service pour obtenir l'instance.

Exemple (code Java 9) :

```
package fr.jmdoudoux.dej.spi;

import fr.jmdoudoux.dej.spi.service.MonService;

public class Test9SPI {
  public static void main(String[] args) {

    MonService.obtenirPremiereInstance().ifPresent(MonService::afficher);
  }
}
```

33.5.4. La résolution de services

Depuis Java 6, l'API ServiceLoader permet d'enrichir une application en permettant de découvrir les implémentations de certaines interfaces ou classes abstraites et permettre leur chargement pour les utiliser.

Cette API est utilisée pour mettre en oeuvre les services dans les modules.

Avant Java 9, il suffisait de mettre les jars contenant les implémentations dans le classpath.

A partir de Java 9, il suffit de mettre les jars modulaires contenant les implémentations dans le module path.

33.5.5. Les options de jlink pour résoudre les services

Jlink est un outil fourni dans le JDK à partir de Java 9 qui permet de créer des JRE personnalisés. Un JRE personnalisé ne contient que les modules requis à l'exécution de l'application pour laquelle il a été créé.

Généralement les modules qui contiennent des implémentations d'un service sont considérés comme optionnel. Jlink ne résout pas automatiquement les modules qui contiennent une ou des implémentations d'un service.

Cependant jlink propose deux options utilisables pour nous aider à résoudre les modules contenant des implémentations de services (ceux déclarés avec l'instruction provides dans le descripteur de module) :

- --bin-services : demande à jlink de résoudre tous les modules contenant des fournisseurs de service et leurs dépendances
- --suggest-providers : demande de fournir la liste des implémentations du service déclarées dans le module path et le classpath

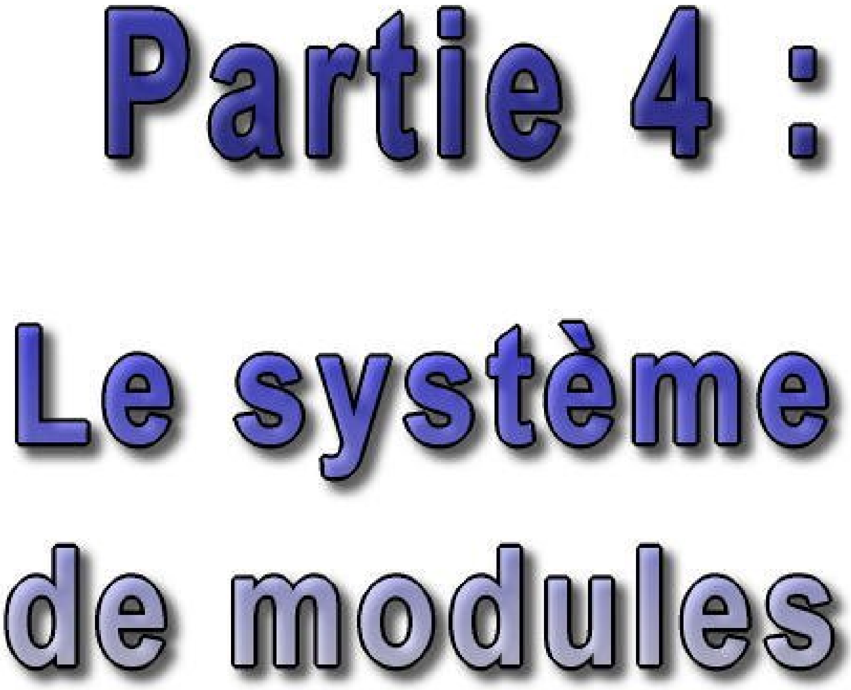
Il est recommandé d'utiliser l'option --suggest-providers qui affiche une liste des fournisseurs trouvées. Il est alors possible d'ajouter les modules utiles à l'application parmi ceux proposés par l'option --suggest-providers.

Dans l'exemple ci-dessous, les modules sont dans le sous-répertoire modules

Exemple (code Java 9) :

```
C:\java\app>jlink --module-path ./modules --suggest-providers fr.jmdoudoux.dej.spi.service.
MonService

Suggested providers:
  fr.jmdoudoux.dej.spi.impla provides fr.jmdoudoux.dej.spi.service.MonService used by
  fr.jmdoudoux.dej.spi.service
  fr.jmdoudoux.dej.spi.implb provides fr.jmdoudoux.dej.spi.service.MonService used by
  fr.jmdoudoux.dej.spi.service
C:\java\app>
```

Partie 4 : Le système de modules



Java 9 a introduit le système de modules de la plateforme Java (Java Platform Module System ou JPMS).

Cette partie contient les chapitres suivants :

- Le système de modules de la plateforme Java : ce chaptitre décrit le système de modules de la plateforme Java
- Les modules : ce chapitre détaille les modules et les descripteurs de module

34. Le système de modules de la plateforme Java

Chapitre 34

Niveau : 🟩⬜⬜⬜⬜⬜ Fondamental

La fonctionnalité majeure de Java 9 est le système de modules de la plate-forme Java (Java Platform Module System ou JPMS). JPMS modularise les bibliothèques de classes fournies par le JDK. JPMS peut aussi être utilisé par les développeurs pour modulariser les applications ou les bibliothèques. Cela permet aux développeurs de décomposer leurs applications en modules. Ces modules peuvent ensuite spécifier les autres modules dont ils ont besoin et les packages qu'ils exposent pour être utilisés par d'autres modules.

Le système de modules de la plateforme Java (JPMS) est issu des travaux du projet Jigsaw.

Les principaux objectifs de l'introduction de la modularisation dans la plate-forme Java sont :

- un renforcement de l'encapsulation qui améliore la maintenance des applications, des bibliothèques et des frameworks
- une meilleure fiabilité de la configuration en permettant la déclaration explicite des dépendances entre modules
- une amélioration de la sécurité
- une amélioration des performances notamment dans le chargement des types
- la possibilité de réduire la taille du JDK pour faciliter son déploiement sur des machines à ressources réduites ou dans le cloud

L'écosystème Java dispose déjà de plusieurs systèmes de modules notamment OSGi et JBoss Modules. L'approche de ces systèmes existants est différente : ils ont été créés et utilisés pour des cas d'utilisation concrets et réels. Le but de JPMS est d'être globalement une extension de la JVM pour permettre un support des modules.

JPMS impacte de nombreux éléments de la plateforme Java : les bibliothèques, le langage, la JVM et les outils.

Le système de modules de la plateforme Java intègre :

- la modularisation du JDK : le but est de diviser le JDK en différents petits modules car le fichier rt.jar historique est monolithique et trop gros
- l'encapsulation dans des modules dédiés de la plupart des API internes du JDK. Quelques-unes de ces API qui sont largement utilisés seront cependant encore accessible pour limiter les impacts dans un premier temps
- l'outil jlink qui permet de créer un JRE personnalisé pour une application : ce JRE ne contient alors que les modules utiles et nécessaires à l'exécution de cette application

JPMS ajoute le concept de modules à Java qui est un nouvel élément structurant :

- une classe contient des champs et des méthodes
- un package contient des types (classes,interfaces, ...) et éventuellement des ressources
- un module contient des packages

L'utilisation du système de modules impliquent plusieurs choses :

- assurer une configuration fiable des dépendances des modules
- une encapsulation forte
- une identification des modules notamment pour la déclaration et la recherche des dépendances
- une potentielle réorganisation du code notamment pour du code legacy

L'utilisation de modules permet :

- de mettre en oeuvre une forte encapsulation (strong encapsulation). Les modules peuvent masquer l'accès à des classes internes dont l'implémentation ne doit pas être exposée. Par défaut, un package d'un module n'est pas visible à l'extérieur du module quelle que soit la visibilité des types qu'il contient. Ceci apporte de nombreux bénéfice en termes de design. Il ne sera plus nécessaire de définir des packages préfixés par impl ou internal pour lesquels la Javadoc précise qu'il ne faut pas l'utiliser les classes qu'ils contiennent
- de fournir une liste des modules utilisés comme dépendances
- de limiter la taille en ne prenant que les modules utiles pour construire un JRE personnalisé
- d'améliorer les performances
- de renforcer la sécurité
- de faciliter les futures évolutions

Un module peut être packagé sous la forme d'un jar modulaire qui est un fichier jar qui contient un fichier module-info.class à sa racine.

Le fichier module-info.class est issu de la compilation d'un fichier particulier nommé module-info.java. Ce fichier est le descripteur du module : il définit entre autres le nom du module, les packages dont les classes publiques peuvent être accédées par d'autres modules, les module requis (les dépendances), les services fournis ou consommés, ...

Pour utiliser un jar modulaire comme module, celui-ci doit être ajouté dans le modulepath et non plus dans le classpath. L'avantage que cela soit toujours un jar est qu'il est toujours possible d'utiliser un jar dans le classpath. Dans ce cas, le fichier sera utilisé comme un fichier jar classique et non pas comme un module même s'il contient un fichier module-info.class qui sera simplement ignoré dans ce cas. Dans cette situation, tous les packages sont accessibles puisque le fichier est chargé via le classpath.

La modularité est une fonctionnalité essentielle de la plateforme Java pour lui permettre de futures évolutions.

Ce chapitre contient plusieurs sections :

- ♦ La modularisation
- ♦ Les difficultés pouvant être résolues par les modules
- ♦ Java Plateform Module System (JPMS)
- ♦ L'implémentation du système de modules

34.1. La modularisation

La modularité est un principe de conception qui permet :

- un couplage faible entre les composants
- de définir un contrat et les dépendances entre des composants
- de masquer le détail des implémentations en utilisant une encapsulation forte

Dans le domaine des logiciels, la modularité consiste à l'écriture et à la mise en oeuvre d'un programme ou d'un système informatique sous la forme d'un certain nombre de modules, plutôt que sous la forme d'une conception monolithique. Cela revient à implémenter une application ou un système sous la forme d'un ensemble de modules.

La mise en oeuvre de modules encourage à utiliser des bonnes pratiques de conception telles que l'encapsulation et la séparation des préoccupations (separation of concerns).

Des interfaces sont utilisées pour permettre aux modules de communiquer. L'utilisation de modules permet de réduire le couplage et de faciliter le développement d'applications en réduisant la complexité du système.

D'une manière générale dans un langage de programmation, un module est un artéfact qui contient du code et des méta données (description du module, relation avec les autres modules, ...)

Un module devrait avoir certaines caractéristiques :

- une unité autonome déployable de manière simple
- proposant une interface définissant un contrat pour la communication (couplage faible)

- possédant une identité cohérente et unique (identifiant et version du module)
- pouvant déclarer et utiliser d'autres modules qui sont des dépendances
- définies via des méta-informations

Un module devrait mettre en oeuvre trois principes :

- encapsulation forte : l'encapsulation implique de cacher les détails d'implémentation, ce qui permet de modifier l'implémentation sans impacter l'utilisation du module
- abstraction stable : un module devrait exposer ces fonctionnalités au travers d'interfaces publiques pour réduire le couplage
- déclaration des dépendances : la définition du module devrait contenir la liste des modules dont il dépend

34.2. Les difficultés pouvant être résolues par les modules

Les plateformes Java antérieures à la version 9 possèdent plusieurs inconvénients lors du développement et du déploiement d'applications :

- la taille du JDK limite son utilisation sur des appareils limités en ressources. Java avait tenté d'apporter une solution sous la forme de trois Compact Profiles
- le fichier rt.jar est monolithique et trop gros (53 Mo). En incluant les autre fichiers jar, on arrive à 67 Mo
- les évolutions dans cet ensemble monolithiques ne sont pas faciles
- il n'y a pas d'encapsulation forte, ce qui permet à toutes classes publiques d'être utilisées. Ceci inclut les classes normalement à usage interne de la JVM comme celles des packages sun.*, *.internal.*, ...
- l'accès à toutes les classes publiques peut aussi avoir des conséquences sur la sécurité
- le classpath ne permet pas de déterminer si toutes les classes requises sont présentes pour une exécution de l'application
- deux classpaths doivent être gérés : un pour la compilation et un pour l'exécution

Le système de modules de Java tente d'apporter certaines solutions à ces problématiques.

34.2.1. La taille croissante des API du JRE

Au fur et à mesure des versions de Java, la taille de son runtime n'a fait que croître. Par exemple, l'API Core de Java 8 contient 217 packages.

Cette taille sans cesse croissante de la plate-forme Java SE rend son utilisation de plus en plus difficile sur les petits appareils malgré le fait que de nombreux appareils de ce type soient capables d'exécuter une machine virtuelle Java. Cela augmente aussi la taille des livrables, ce qui peut poser des problèmes lors de déploiements sur de nombreux noeuds d'un cluster dans le cloud.

Avant Java 8, il n'y aucun moyen de définir et utiliser un sous-ensemble du runtime. Ainsi chaque runtime inclut toutes les bibliothèques telles que AWT, Swing, XML ou même Corba même si l'application n'en a pas besoin.

Java 8 a introduit la notion de compact profiles (JSR 337) qui définit trois sous-ensembles des API du runtime. Le fait de n'avoir que trois ensembles rend la solution rigide et ne permet donc pas personnaliser les API requises dans le runtime.

L'API Java Core a donc besoin d'être modularisée.

34.2.2. Les limitations du format jar

Le modèle de packaging reposant sur les fichiers de type JAR présente plusieurs inconvénients :

- le nom du fichier JAR n'est pas utilisé par la JVM
- un JAR ne permet pas de déclarer ses dépendances
- aucun mécanisme de gestion de versions n'est proposé

Ces inconvénients conduisent à différentes problématiques :

- il est impossible pour la JVM de déterminer si tous les JAR requis sont présents dans le classpath. Ainsi si une classe n'est pas trouvée, une exception de type NoClassDefFoundError est levée à sa première utilisation
- un fichier JAR est un simple conteneur : il n'est pas possible de mettre en oeuvre l'encapsulation entre JAR
- des conflits de versions si plusieurs versions d'un jar sont présents dans le classpath

34.2.3. Les problématiques liées au classpath

Il existe trois classpath :

- Boot Classpath : permet de charger des classes du rt.jar
- Extension Classpath : permet de charger des classes fournies dans le mécanisme d'extension
- User ClassPath : permet de charger des classes de l'application. C'est le classpath le mieux connu car c'est celui que l'on configure couramment. Par défaut, c'est le répertoire courant. Il est possible de le configurer en utilisant l'option -classpath ou -cp : la valeur peut être composée de répertoires et/ou de fichiers .jar

Chacun est utilisé par un ClassLoader dédié. Toutes les classes requises sont chargées par un ClassLoader. Les ClassLoader proposés en standard dans la JVM appliquent le principe de délégation pour demander à son ClassLoader parent de tenter de charger la classe demandée.

Ce mécanisme de délégation améliore la sécurité (par exemple pour permettre de charger la classe java.lang.String du fichier rt.jar et pas d'un jar de l'application). Par contre, ce mécanisme de délégation ralentit le temps de chargement d'une classe. C'est d'autant plus important qu'une application peut charger des centaines voire des milliers de classes.

Un de ces ClassLoader peut charger des classes indiquées dans le classpath. Par défaut, toutes classes de l'application sont chargées par le même ClassLoader à partir des composants précisés dans le classpath. Il est possible d'utiliser d'autres ClassLoader ce qui rend le mécanisme de chargement de classes encore plus complexe.

Le classpath ne permet pas d'exprimer les relations entre les composants. Par conséquent, si un composant nécessaire est manquant, il ne sera pas détecté jusqu'à ce qu'il soit tenté de l'utiliser. Le classpath permet également de charger des classes d'un même package à partir de différents composants, ce qui entraîne un comportement imprévisible et des erreurs difficiles à diagnostiquer.

Comme il n'y aucune information sur les dépendances requises et que les classes sont chargées uniquement lorsque la JVM en a besoin, l'application peut démarrer et des soucis peuvent survenir en cours d'exécution.

La plateforme Java ne propose aucune fonctionnalité pour déterminer si les bons fichiers JAR sont disponibles. Le compilateur et la JVM repose sur le mécanisme du classpath. Ce comportement n'est pas fiable et conduit parfois à des comportements inattendus.

34.2.3.1. JAR/Classpath Hell

Le mécanisme de classpath, introduit depuis Java 1.0, implique plusieurs soucis :

- impossibilité de savoir au lancement d'une application si un jar est manquant puisque c'est uniquement lorsqu'une classe sera utilisée qu'elle sera chargée
- quelle classe est chargée si elle est présente en plusieurs versions dans le classpath
- quel est l'ordre de parcours des éléments du classpath
- une classe peut être chargée par différents ClassLoader

Ces problématiques liées à l'utilisation du classpath sont connues sous le nom de Classpath Hell ou JAR Hell.

Le terme JAR Hell ou Classpath Hell désigne des problèmes induits par le mécanisme de chargement des classes issues du classpath. Ce mécanisme de chargement des classes de la JVM via le classpath est à l'origine de plusieurs problèmes.

34.2.3.2. Un classpath pour le compilateur et un autre pour la JVM

Le mécanisme de classpath est utilisé lors de étapes de compilation et d'exécution. Il y a donc deux classpath :

- un pour le compilateur
- un pour la JVM à l'exécution de l'application

Il est possible que ces deux classpath soit différents puisqu'ils doivent explicitement être définis pour le compilateur à la compilation et pour la JVM à l'exécution. Il est ainsi possible que la compilation réussisse car tous les éléments requis sont dans le classpath mais qu'un élément soit absent du classpath de la JVM à l'exécution et lève une exception de type NoClassDefFoundError ou ClassNotFoundException.

Bien sûr à la compilation, si une dépendance est absente, le compilateur émet une erreur et le code n'est pas compilé.

Le code compilé ne signifie pas que tout va bien se passer à l'exécution. Comme la JVM possède son propre classpath, il est possible d'avoir des erreurs à l'exécution si la classe est absente ou si la version de la classe est différente de celle utilisée à la compilation.

34.2.3.3. Les collisions de versions

Il est possible que le classpath contienne plusieurs versions différentes d'un même jar : le classLoader charge alors la première version de la classe trouvée, ce qui peut provoquer des comportements erratiques.

Il est possible d'avoir plusieurs fois la même classe (possédant le même nom pleinement qualifié) dans différents éléments du classpath. C'est fréquent notamment lorsque le classpath contient plusieurs versions différentes d'une même bibliothèque à cause des dépendances transitives par exemple.

Le mécanisme de chargement de classes à partir du classpath charge la première classe trouvée, rendant impossible le chargement d'une autre version de la classe contenue dans le classpath (le terme anglais pour désigner ce comportement est Shadowing). Il n'y a aucune spécification sur l'ordre de chargement de classes donc aucune garantie sur la version de la classe qui sera chargée même si généralement les JVM scannent les JAR dans l'ordre dans lequel ils sont fournis dans le classpath. Cela peut ne pas être le cas sur toutes les JVM ni dans le cas où le caractère joker * ou des répertoires sont utilisés dans le classpath.

Cette problématique courante avec le classpath survient lorsque deux jars ont une dépendance vers deux versions différentes d'un autre jar. Il est ainsi possible qu'une même classe soient dans plusieurs jars inclus dans le classpath puisque plusieurs versions d'un jar sont dans le classpath.

Si les deux versions du jar sont ajoutées dans le classpath alors le comportement risque de ne pas être celui attendu. Une même classe ne peut être chargée qu'une seule fois par un même ClassLoader : c'est la première trouvée par le ClassLoader qui sera utilisée. La seconde version de la classe présente dans le classpath ne sera pas chargée. Dans le meilleur des cas, la version la plus récente est compatible avec la version précédente et il suffit simplement d'ajouter uniquement la version la plus récente dans le classpath. Ceci n'est vrai que s'il y a une compatibilité ascendante, ce qui n'est pas toujours le cas.

La version de la classe chargée n'est donc pas prédictible de manière fiable. Si les différences entre les versions d'une même classe ne concernent que le comportement, cela peut engendrer des bugs aléatoires difficilement identifiables induisant des comportements inattendus. La détection de ces problèmes n'est pas toujours facile.

34.2.3.4. Le temps de démarrage d'une JVM

Certains mécanismes liés au classpath ralentissent les performances notamment pour :

- le parcours séquentiel des jars du classpath pour trouver une classe
- le parcours des classes pour rechercher quelles sont celles possédant une annotation particulière

34.2.4. L'impossibilité de définir les dépendances

Depuis la première version de Java, une application exécutée dans une JVM ne connait pas ses dépendances. Un JAR ne permet pas de définir quels sont les autres JAR dont il dépend ce qui permettrait à la JVM de s'assurer sur tous les composants nécessaires sont présents dans le classpath.

Le problème se complexifie encore avec les dépendances transitives : ce sont les dépendances requises par une dépendance. Ces dépendances de dépendances peuvent elles-mêmes requérirent d'autres dépendances et ainsi de suite. Les dépendances transitives rendent la gestion des dépendances encore plus complexe et augmente donc les possibilités d'erreurs. C'est notamment le cas lors de la discordance de versions entre des jar dépendants.

Il est donc nécessaire de s'assurer manuellement que tous les jars utiles et nécessaire sont dans le classpath. C'est aussi le cas pour les dépendances optionnelles : ce sont des jars dont les classes ne sont nécessaires que selon l'utilisation de certaines fonctionnalités.

Des outils de build externes au JDK peuvent aider à gérer ces dépendances comme par exemple Maven de la fondation Apache. Cela ne résout pas toujours tous les problèmes : il est possible qu'il manque des composants dans le classpath.

En l'absence de configuration, il est courant d'avoir des NoClassDefFoundError au runtime, et ce bien après le démarrage de l'application. Ceci est dû au fait que les classes ne sont chargées que lorsque la JVM en a besoin et qu'il est possible que des dépendances soient manquantes.

La JVM ne peut détecter l'absence d'une classe dans le classpath qu'au moment où elle est requise et doit être chargée. Si la classe ne peut être chargée par la JVM car elle n'est pas trouvée dans le classpath, elle lève une exception de type NoClassDefFoundError ou ClassNotFoundException en cas de tentative de chargement par introspection (méthode forName() de la classe Class ou loadClass() et findSystemClass() de la classe ClassLoader).

34.2.5. Pas d'encapsulation dans un jar ou entre les jars

Il n'est pas possible d'avoir des éléments visibles, par exemple uniquement par ceux du fichier JAR qui les contiennent, ou en dehors de celui-ci. Le mécanisme de contrôle d'accès du langage de programmation Java et de la machine virtuelle Java ne permet à aucun composant d'empêcher d'autres composants d'accéder à ses packages.

Historiquement, l'encapsulation est uniquement proposée au travers des modificateurs de visibilité. Les modificateurs de visibilité de Java peuvent être utilisés pour implémenter l'encapsulation entre les classes d'un même package. Mais entre plusieurs packages, il n'y a que la visibilité publique qui soit utilisable.

Toutes les classes publiques sont accessibles par toutes les autres du classpath. Cela limite les possibilités d'encapsulation notamment au niveau du JAR.

En conséquence de nombreuses classes de l'API Core de Java sont public mais ne devrait pas être utilisée. Mais comme elles sont publiques, il est possible de les utiliser directement ou indirectement si c'est une dépendance qui les utilise. C'est notamment le cas des classes des packages sun.misc, jdk.internal, ... La plus connues d'entre-elles est la classe sun.misc.Unsafe. Malgré son nom explicite, elle est largement utilisée notamment par des frameworks couramment utilisés.

Les API internes du JDK, principalement dans les packages sun.* sont encapsulées dans des modules qui ne les exportent pas : ces API ne sont donc plus utilisables.

La vocation de ces API n'était pas d'être utilisable en dehors du JDK mais comme elles étaient public et proposaient des fonctionnalités puissantes, elles ont été largement utilisées par des bibliothèques.

Certaines de ces API ont été remplacées et certaines sont encore accessibles pour le moment mais elles devront être retirées à terme.

34.2.6. La sécurité

Pour permettre une utilisation d'une classe par une classe d'un autre package, celle-ci doit être public. De fait, n'importe quelle autre classe du classpath peut l'utiliser : cela peut induire des problèmes de sécurité lié au fait que du code malicieux puisse invoquer des fonctionnalités critiques.

Pour compenser cela, Java 1.1 a introduit le SecurityManager dont le rôle est de vérifier si l'utilisation d'une fonctionnalité critique est autorisée ou non. Bien sûr pour que cela fonctionne, il est nécessaire que le SecurityManager soit invoqué dans le code pour vérifier l'autorisation de l'invocation d'une fonctionnalité.

De plus, en utilisant l'API Introspection, il est facile d'accéder à des membres privés même pour les modifier dans le cas de propriétés.

34.3. Java Plateform Module System (JPMS)

Un système de modules Java doit faciliter la séparation du code en modules distinct, respectant une encapsulation stricte et faiblement couplée. Les dépendances doivent être clairement spécifiées et strictement appliquées.

Ainsi, JPMS permet de structurer le code de manière modulaire en proposant une encapsulation forte et une configuration plus fiable, tout en permettant un couplage faible entre modules.

Le système de modules est utilisé dans le JDK lui-même et peut être utilisé dans les applications.

JPMS impacte de manière profond la plateforme, les outils, les applications et l'ensemble de l'écosystème Java. JPMS est probablement la première fonctionnalité introduite en Java qui a un impact sur toute la stack : JVM, Java Core, outils (compilateur, packaging, build, ...), bibliothèques, applications, ...

En plus des améliorations pour la plate-forme elle-même, le système de modules change complètement la façon dont les applications Java vont être structurées. L'un des objectifs est de renforcer l'encapsulation : avec les modules, il est possible de définir précisément les éléments qui seront utilisables pour les autres modules. Et pour la première fois, du code Java aura la possibilité de connaître ses dépendances et d'avoir une configuration plus fiable. Ensemble, ces fonctionnalités tentent de trouver partiellement une solution au Classpath Hell.

JPMS améliore la sécurité et facilite la maintenance des applications et des bibliothèques grâce à l'encapsulation forte et une meilleure fiabilité de la configuration.

JPMS change de manière importante comment les applications, notamment les grosses, sont développées et surtout déployées.

34.3.1. Les buts de JPMS

Les principaux objectifs de la modularisation proposée par JPMS sont :

- **encapsulation forte (strong encapsulation)** : JPMS permet à un module de déclarer lesquels de ses packages sont accessibles par d'autres modules, car par défaut ils ne le sont pas. Cette encapsulation est appliquée aussi dans les modules du JDK pour empêcher l'accès à leurs API internes
- **configuration fiable (reliable configuration)** : JPMS permet à un module de déclarer qu'il dépend d'autres modules, comme d'autres modules peuvent en dépendre. En utilisant la configuration de chaque module, le compilateur et la JVM peuvent s'assurer que tous les modules requis sont présents. Le mécanisme des modules est ainsi plus fiable que le mécanisme utilisé par le classpath. Si un jar est manquant dans le classpath à l'exécution, il ne sera détecté que lors du chargement d'une de ces classes. Ce chargement n'est effectué que lors de la première utilisation de la classe : cela peut survenir bien après le démarrage de l'application
- **amélioration des performances** notamment lors du chargement des classes : une classe peut être chargée plus rapidement grâce au fait que son package ne peut être que dans un seul module
- **une plate-forme avec taille adaptable** : JPMS permet à la plate-forme Java SE et à ses implémentations d'être décomposées en un ensemble de composants qui peuvent être assemblés par les développeurs dans des configurations personnalisées qui contiennent uniquement les fonctionnalités réellement requises par une

application. Cette réduction de l'empreinte du JRE facilite notamment l'utilisation pour des applications embarquées (IoT) et le déploiement dans le cloud

Tous ces objectifs n'ont pas une importance équivalente pour le JDK et pour les applications.

34.3.2. Une configuration plus fiable (reliable configuration)

Chaque module est identifié par un nom défini dans un descripteur de module. Ce descripteur de module doit aussi contenir la liste des modules requis en tant que dépendance. Cette configuration pour être utilisée à différentes étapes :

- compilation
- build
- exécution

Ces différentes étapes pourront alors échouer si une dépendance est manquante ou si une anomalie est détectée, et ce dès leurs premières étapes.

34.3.3. L'encapsulation forte (strong encapsulation)

Un des intérêts des modules est de permettre de renforcer l'encapsulation. Par défaut, aucune classe d'un module n'est accessible en dehors du module même si la classe est public. Pour permettre un accès à d'autres modules, il est nécessaire d'exporter le ou les packages concernés. Tous les autres packages non exportés ne sont accessibles uniquement que par le module lui-même.

Une fois un package exporté, les règles de visibilité d'une classe s'applique comme depuis toujours en Java.

Un module ajoute donc en plus de la visibilité un niveau d'accessibilité.

Le sens du modificateur public change dans un module. Par défaut, une classe publique dans un module n'est accessible que par les autres classes du module. Pour que cette classe publique puisse être utilisé dans d'autre module, il fait obligatoire exporter le package qui contient cette classe. Toutes les classes publiques d'un package exporté sont alors potentiellement accessibles par d'autres modules qui déclareront le module en dépendance.

Ainsi pour permettre l'utilisation d'une classe d'un module par un autre module, trois contraintes doivent être respectées :

- la classe doit être public
- le package contenant la classe doit être exporté
- le module qui souhaite utiliser la classe doit déclarer le module de la classe en tant que dépendance

La mise en oeuvre de ces trois règles permet un accès à une classe par un autre module.

Le fait de ne pas permettre par défaut un accès aux classes publiques d'un package permet de renforcer l'encapsulation. Il est par exemple possible de ne pas exposer des packages qui contiennent une implémentation et d'exposer les packages des interfaces.

L'encapsulation forte des modules va renforcer la sécurité et la maintenabilité :

- le code critique peut être encapsulé dans le module et rendu non visible même si cela concerne une classe public
- l'API publique d'un module peut être réduit à son minimum
- une application concrète est mise en oeuvre dans le JDK : les API internes non standard sont encapsulées et ne sont donc plus accessible par les applications

34.3.4. L'évolutivité de la plateforme

La modularisation a été appliquée à Java Core : l'historique fichier rt.jar a été découpé en un certain nombre de modules. La maintenance d'éléments plus petits est plus simple que la maintenance d'un élément monolithique.

Les modules facilitent aussi l'ajout ou la suppression de fonctionnalités dans Java Core.

Avec la modularisation, il est possible de créer son propre JRE personnalisé composé uniquement des modules dont une application a besoin. Cela contribue à renforcer la plate-forme Java comme une solution pour des applications sur petits périphériques ou dans des conteneurs.

34.4. L'implémentation du système de modules

Le système de modules a été développé dans le projet Jigsaw.

La JSR 376 (Java Platform Module system) spécifie les évolutions dans le langage Java, la JVM et l'API Java pour la mise en oeuvre du système de modules dans la plateforme Java.

La mise en oeuvre du système de modules repose sur plusieurs JEP.

34.4.1. Le projet Jigsaw

Le rôle du projet Jigsaw est de permettre la mise en oeuvre d'un système de modules pour la plateforme Java nommé Java Platform Module System (JPMS).

Le but initial était de rendre la plateforme Java SE modulaire puis a été étendu pour permettre d'utiliser ce système pour rendre modulaire le JRE mais aussi les applications Java.

Les principaux buts du projet Jigsaw sont de définir un système standard de modules pour la plateforme Java et le langage Java afin de permettre :

- d'améliorer la sécurité et la maintenabilité
- d'améliorer la scalabilité et les performances
- de permettre l'exécution sur des environnements à ressource réduites

Pour atteindre ces objectifs, le projet a conçu et implémenté un système de modules qui a été appliqué à la plateforme et au JDK au travers de différentes actions :

- créer un système de modules : implémenté dans la JEP 261
- appliquer ce système de modules aux sources du JDK : implémenté dans la JEP 201
- modulariser les bibliothèques du JDK : implémenté dans la JEP 200
- faire évoluer la plateforme pour offrir un support des modules : implémenté dans la JEP 220
- permettre de créer une plateforme de plus petite taille en intégrant que le sous-ensemble de modules requis : implémenté dans la JEP 282

L'intégration du projet Jigsaw a été initialement prévue pour Java 7, repoussée à Java 8, pour finalement être décalée à nouveau pour finalement être intégrée à Java 9. La première mouture du système de module spécifié par la JSR 376 et implémenté par la JEP 261 a été intégré dans le build 111 du JDK 9 en mars 2016.

34.4.2. Les JEP utilisées pour intégrer le système de modules

Le système de modules a été développé au travers de plusieurs JEP :

JEP	Rôle
JEP 261: Module System	Implémenter le Java Platform Module System (JPMS) tel que spécifié dans la JSR 376
JEP 200: The Modular JDK	Modulariser la plateforme Java tel que précisé dans la JSR 376 et implémenté dans la JEP 261 : définit la structure modulaire du JDK
	Décomposer le JDK et le JRE en images pour chaque module

JEP 220: Modular Run-Time Images	Définir un nouveau format des URI pour nommer les modules, les classes et les ressources de manière indépendante du format de l'image Suppression des fichiers rt.jar et tools.jar du JRE Suppression des mécanismes endorsed et extension
JEP 260: Encapsulate Most Internal APIs	Encapsuler la plupart des API à usage interne du JDK sauf celles qui sont largement utilisés et ne possèdent pas encore de solution de remplacement
JEP 201: Modular Source Code	Réorganiser le code source du JDK pour le rendre modulaire
JEP 282: jlink the Java Linker	Fournir un outil capable de créer un JRE personnalisé pour une application Créer un outil qui permet d'assembler et optimiser un ensemble de modules et leurs dépendances afin d'obtenir une image (jlink)

34.4.3. La modularisation du JDK

A partir de Java 9, l'API Core de Java est obligatoirement modulaire. Le JDK lui-même a été modularisé pour être composé d'un peu moins d'une centaine de modules désignés comme étant des platform modules.

Il est possible d'obtenir le liste complète des platform modules en utilisant l'option --list-modules de la JVM

Résultat :

```
$ java --list-modules | wc -l
98
```

Le système de modules fait la distinction entre les modules standard et les modules non standard. Les modules standard ont leurs spécifications gérées par la plateforme Java et leurs noms de modules commencent par "java.".

Les modules non standard ne devraient donc pas avoir leur nom commençant par "java.".

À la racine de l'arborescence des modules se trouve le module java.base, qui contient les classes essentielles notamment celles des packages java.lang, java.io, java.math, java.net, java.nio, java.security, java.text, java.time, java.util, ...

Les classes dans le classpath ont accès à l'intégralité des classes contenues dans les modules du JDK pour des raisons évidentes de compatibilité.

Les modules dans le module path doivent explicitement déclarer leur dépendance vers les modules requis y compris ceux du JDK.

La structure du JDK a été impactée :

- Le fichier rt.jar n'existe plus
- Les modules sont packagés en utilisant le format jmod
- Les mécanismes endorsed et d'extension sont supprimés

La structure de sous-répertoires du JDK contient un répertoire nommé jmods. Ce répertoire contient des fichiers avec l'extension .jmod, un pour chaque module. Un fichier jmod contient des classes regroupées en packages, des ressources et des bibliothèques natives.

Le fichier src.zip contient toujours les sources de l'API Core de Java : il est dans le sous-répertoire lib du JDK. Il contient un sous-répertoire pour chaque module du JDK.

34.4.4. Les modules dépréciés de Java 9

Plusieurs modules de Java 9 sont dépréciés avec l'attribut forRemoval=true :

- java.activation
- java.corba
- java.transaction
- java.xml.bind
- java.xml.ws
- java.xml.ws.annotation

Ces modules sont fournis dans Java 9 mais ne sont pas accessibles par défaut. Il faut explicitement les ajoutées en tant que module racine du graphe de modules. Tous ces modules sont retirés dans Java 11.

34.4.5. Les modules du JDK

Les modules de la plateforme sont repartis en plusieurs groupes. Le groupe sert de préfixe dans le nom de chaque module :

- java : les modules standard
- javafx : les modules de JavaFX
- jdk : les modules utilisés pour les outils fournis dans le JDK
- oracle : les modules spécifiques à Oracle

Tous les modules de la plateforme ou applicatifs dépendent implicitement du module java.base. Le module java.base contient les classes et API de base (utils, collections, IO, concurrency, ...)

Le module java.base est le module de base : c'est un module particulier qui ne dépend d'aucun autre module. Tous les autres modules dépendent implicitement du module java.base. Le module java.base expose de nombreux packages notamment :

- java.io
- java.lang.*
- java.math
- java.net.*
- java.nio.*
- java.security.*
- java.text.*
- java.time.*
- java.util.*
- javax.crypto.*
- javax.net.*
- javax.security.*

Des modules dédiés existent pour les autres fonctionnalités logging, desktop, xml, sql, naming, corba, ...

Il est possible d'utiliser l'option --list-modules de la JVM pour obtenir la liste des modules.

Résultat :

```
$ java --list-modules
java.activation@9.0.1
java.base@9.0.1
java.compiler@9.0.1
java.corba@9.0.1
java.datatransfer@9.0.1
java.desktop@9.0.1
...
```

Le nom de chaque module se termine par un @ suivi du numéro de version du module.

Les modules du JDK varient en fonction de la version de Java utilisée soit parce que des fonctionnalités sont ajoutées ou retirées :

Module	Versions de Java									
	9	10	11	12	13	14	15	16	17	18
java.activation	*	*								
java.base	*	*	*	*	*	*	*	*	*	*
java.compiler	*	*	*	*	*	*	*	*	*	*
java.corba	*	*								
java.datatransfer	*	*	*	*	*	*	*	*	*	*
java.desktop	*	*	*	*	*	*	*	*	*	*
java.instrument	*	*	*	*	*	*	*	*	*	*
java.jnlp	*	*								
java.logging	*	*	*	*	*	*	*	*	*	*
java.management	*	*	*	*	*	*	*	*	*	*
java.management.rmi	*	*	*	*	*	*	*	*	*	*
java.naming	*	*	*	*	*	*	*	*	*	*
java.net.http			*	*	*	*	*	*	*	*
java.prefs	*	*	*	*	*	*	*	*	*	*
java.rmi	*	*	*	*	*	*	*	*	*	*
java.scripting	*	*	*	*	*	*	*	*	*	*
java.se	*	*	*	*	*	*	*	*	*	*
java.se.ee	*	*								
java.security.jgss	*	*	*	*	*	*	*	*	*	*
java.security.sasl	*	*	*	*	*	*	*	*	*	*
java.smartcardio	*	*	*	*	*	*	*	*	*	*
java.sql	*	*	*	*	*	*	*	*	*	*
java.sql.rowset	*	*	*	*	*	*	*	*	*	*
java.transaction	*	*								
java.transaction.xa			*	*	*	*	*	*	*	*
java.xml	*	*	*	*	*	*	*	*	*	*
java.xml.bind	*	*								
java.xml.crypto	*	*	*	*	*	*	*	*	*	*
java.xml.ws	*	*								
java.xml.ws.annotation	*	*								
javafx.base	*	*								
javafx.controls	*	*								
javafx.deploy	*	*								
javafx.fxml	*	*								

javafx.graphics	*	*								
javafx.media	*	*								
javafx.swing	*	*								
javafx.web	*	*								
jdk.accessibility	*	*	*	*	*	*	*	*	*	*
jdk.aot		*	*	*	*	*	*	*		
jdk.attach	*	*	*	*	*	*	*	*	*	*
jdk.charsets	*	*	*	*	*	*	*	*	*	*
jdk.compiler	*	*	*	*	*	*	*	*	*	*
jdk.crypto.cryptoki	*	*	*	*	*	*	*	*	*	*
jdk.crypto.ec	*	*	*	*	*	*	*	*	*	*
jdk.crypto.mscapi	*	*	*	*	*	*	*	*	*	*
jdk.deploy	*	*								
jdk.deploy.controlpanel	*	*								
jdk.dynalink	*	*	*	*	*	*	*	*	*	*
jdk.editpad	*	*	*	*	*	*	*	*	*	*
jdk.hotspot.agent	*	*	*	*	*	*	*	*	*	*
jdk.httpserver	*	*	*	*	*	*	*	*	*	*
jdk.incubator.foreign							*	*	*	*
jdk.incubator.jpackage							*			
jdk.incubator.vector								*	*	*
jdk.incubator.httpclient	*	*								
jdk.internal.ed	*	*	*	*	*	*	*	*	*	*
jdk.internal.jvmstat	*	*	*	*	*	*	*	*	*	*
jdk.internal.le	*	*	*	*	*	*	*	*	*	*
jdk.internal.opt	*	*	*	*	*	*	*	*	*	*
jdk.internal.vm.ci	*	*	*	*	*	*	*	*	*	*
jdk.internal.vm.compiler		*	*	*	*	*	*	*	*	*
jdk.internal.vm.compiler.management		*	*	*	*	*	*	*	*	*
jdk.jartool	*	*	*	*	*	*	*	*	*	*
jdk.javadoc	*	*	*	*	*	*	*	*	*	*
jdk.javaws	*	*								
jdk.jcmd	*	*	*	*	*	*	*	*	*	*
jdk.jconsole	*	*	*	*	*	*	*	*	*	*
jdk.jdeps	*	*	*	*	*	*	*	*	*	*
jdk.jdi	*	*	*	*	*	*	*	*	*	*
jdk.jdwp.agent	*	*	*	*	*	*	*	*	*	*
jdk.jfr	*	*	*	*	*	*	*	*	*	*
jdk.jlink	*	*	*	*	*	*	*	*	*	*

jdk.jpackage								*	*	*
jdk.jshell	*	*	*	*	*	*	*	*	*	*
jdk.jsobject	*	*	*	*	*	*	*	*	*	*
jdk.jstatd	*	*	*	*	*	*	*	*	*	*
jdk.localedata	*	*	*	*	*	*	*	*	*	*
jdk.management	*	*	*	*	*	*	*	*	*	*
jdk.management.agent	*	*	*	*	*	*	*	*	*	*
jdk.management.cmm	*	*								
jdk.management.jfr	*	*	*	*	*	*	*	*	*	*
jdk.management.resource	*	*								
jdk.naming.dns	*	*	*	*	*	*	*	*	*	*
jdk.naming.rmi	*	*	*	*	*	*	*	*	*	*
jdk.net	*	*	*	*	*	*	*	*	*	*
jdk.nio.mapmode						*	*	*	*	*
jdk.pack	*	*	*	*	*					
jdk.packager	*	*								
jdk.packager.services	*	*								
jdk.plugin	*	*								
jdk.plugin.dom	*									
jdk.plugin.server	*	*								
jdk.policytool	*									
jdk.rmic	*	*	*	*	*	*				
jdk.random									*	*
jdk.scripting.nashorn	*	*	*	*	*	*				
jdk.scripting.nashorn.shell	*	*	*	*	*	*				
jdk.sctp	*	*	*	*	*	*	*	*	*	*
jdk.security.auth	*	*	*	*	*	*	*	*	*	*
jdk.security.jgss	*	*	*	*	*	*	*	*	*	*
jdk.snmp	*	*								
jdk.unsupported	*	*	*	*	*	*	*	*	*	*
jdk.unsupported.desktop			*	*	*	*	*	*	*	*
jdk.xml.bind	*	*								
jdk.xml.dom	*	*	*	*	*	*	*	*	*	*
jdk.xml.ws	*	*								
jdk.zipfs	*	*	*	*	*	*	*	*	*	*
oracle.desktop	*	*								
oracle.net	*	*								

35. Les modules

Chapitre 35

Niveau : 🟩🟩⬜⬜⬜⬜ Elémentaire

Les modules permettent d'organiser le code afin de le déployer et de déclarer les dépendances entre eux chacun dans leurs fichiers de description respectifs. Un module est un artefact pris en charge par la JVM au même titre que les packages, les classes, les interfaces, ...

Les modules sont un nouveau type d'éléments dans le langage Java. Avant Java 9, les classes sont regroupées dans des packages. A partir de Java 9, il est possible de regrouper des packages dans un module. Un module Java contient donc un ou plusieurs packages qui vont ensemble. Un module peut être une partie d'une application, une API de la plate-forme Java ou une API tierce.

Un module doit respecter plusieurs caractéristiques :

- il doit avoir un nom unique dans l'espace global défini par la JVM. Comme pour les noms de packages, le nom utilise généralement par convention le nom de domaine inversé mais ce n'est pas une obligation
- il doit avoir un unique descripteur de module sous la forme d'un fichier source module-info.java compilé en module-info.class. Il doit être situé à la racine de la structure de répertoires du code source du module
- un module peut dépendre d'autres modules. Ces dépendances sont définies dans le descripteur de module

Les modules permettent de renforcer l'intégrité de la plateforme et des applications.

Les modules mettent en oeuvre une encapsulation plus forte en permettant de déclarer explicitement les packages qui seront exportés et donc utilisable par d'autres modules. Les autres packages qui ne sont pas exportés ne seront accessibles que par le module lui-même.

Les modules ajoutent de nouvelles règles de visibilité et donc d'accès au langage Java. Dans un module, seules les classes publiques dans des packages exportés sont utilisables par d'autres modules. Cela change profondément la visibilité public historique.

Le système de modules définit les règles de lisibilité et d'accessibilité aux modules qui reposent sur deux concepts :

- module observable : module fournit par la plateforme d'exécution ou dans le module path
- module résolu : module observable qui a été ajouté au graphe des modules pendant la résolution des modules

La description d'un module permet de définir le ou les modules dont il dépend. Cela permet d'obtenir une configuration plus fiable des modules. Cette configuration peut avoir plusieurs utilités : par exemple cela permet à la JVM de vérifier au démarrage que tous le modules requis sont présents dans le module-path.

Le système de modules propose aussi un support pour des services avec un couplage faible. Les services sont des fonctionnalités fournies et utilisées en utilisant l'interface java.util.ServiceLoader.

Un module nommé doit obligatoirement contenir un descripteur de module. C'est un fichier nommé module-info.java à la racine des sources du module qui contient des informations de configuration concernant le module :

- le nom du module
- les packages exportés : une liste de packages dont les éléments publics seront accessibles à l'extérieur du module par d'autres modules qui en dépendent

- les dépendances du module : les modules requis par le module donc les autres modules dont le module à besoin
- les packages ouverts : une liste de packages sur lesquels l'introspection pourra être utilisée
- les services proposés : une ou plusieurs implémentations des services que le module fournit et qui pourront être utilisé par d'autres modules
- les services fournis ou consommés par le module

Un module possède obligatoirement un nom, soit implicite ou explicite.

Par défaut, un module est hermétique :

- aucune classe n'est accessible de l'extérieur du module, même celles qui sont public
- il n'est pas possible d'utiliser l'introspection sur les classes du module de l'extérieur du module
- aucune ressource n'est accessible de l'extérieur du module

Ce chapitre contient plusieurs sections :

- ♦ Le contenu d'un module
- ♦ Le code source d'un module
- ♦ Le descripteur de module
- ♦ Les règles d'accès
- ♦ La qualité des descripteurs de module

35.1. Le contenu d'un module

Un module Java est un artefact qui possède un nom et contient des packages. Le code est organisé en packages qui contiennent des types (classes, interfaces, énumérations, annotations, ...) et éventuellement des ressources. Chaque module nommé doit posséder un descripteur de module.

Ainsi, un module est une façon de regrouper des fichiers .class et des ressources, ajoutant un niveau d'agrégation supérieur à celui des packages. Du code Java est regroupé dans différentes structures :

- une classe regroupe des champs et des méthodes
- un package regroupe des types (classes, interfaces, enums, records) et éventuellement des ressources
- un module regroupe des packages

Un module est donc un artefact qui contient :

- un ou plusieurs packages contenant des types compilés en fichier .class
- éventuellement des ressources
- des métadonnées dans un descripteur de module (module-info.class)

Un module permet de regrouper des packages dans la hiérarchie de répertoires correspondantes contenant les fichiers .class issus de la compilation.

Les packages contenus dans un module sont identiques aux packages historiquement utilisés en Java depuis sa création. Dans un module, les packages ont un rôle supplémentaire : ils sont utilisés pour déterminer quels sont les éléments publics qui seront accessibles en dehors du module.

Un module ne peut pas contenir d'autres modules.

Physiquement en Java, la forme la plus simple d'un module est une archive jar qui contient à sa racine un fichier module-info.class issu de la compilation du fichier module-info.java.

35.2. Le code source d'un module

Chaque module doit avoir ses sources dans son propre répertoire et avoir un fichier module-info.java à la racine de l'arborescence de ses sources.

Le répertoire des sources peut ne contenir que le code d'un module : c'est par exemple le format utilisé par les projets Maven.

Résultat :

```
src
 |--module-info.java
 |--com
 |    |--jmdoudoux
 |         |--module
 |              |--MaClasse.java
```

Le répertoire des sources peut contenir le code de plusieurs modules : dans ce cas, le code de chaque module doit être dans un sous-répertoire dédié chacun contenant un fichier module-info.java. C'est par exemple le format utilisé pour les sources du JDK.

Une bonne pratique est alors de nommé de répertoire du chaque module avec le nom du module correspondant.

Résultat :

```
src
|--fr.jmdoudoux.monmodulea
|  |--module-info.java
|  |--com
|     |--jmdoudoux
|        |--modulea
|           |--MaClasseA.java
|--fr.jmdoudoux.monmoduleb
|  |--module-info.java
|  |--com
|     |--jmdoudoux
|        |--moduleb
|           |--MaClasseB.java
```

35.3. Le descripteur de module

Les méta-données d'un module sont fournies dans un descripteur de modules. Ces méta-données doivent répondre à plusieurs questions :

- quel est le nom du module ?
- quelles sont les dépendances requises ? Par défaut, il y a toujours une dépendance implicite vers le module java.base
- quels sont les packages exportés ? L'accès à une classe public ne pourra se faire en dehors du module que si son package est exporté. Par défaut, aucun package n'est exporté

Le fichier qui va contenir ces informations est le descripteur de module. Il permet de fournir des éléments concernant la description d'un module notamment :

- le nom du module
- les modules dont il dépend : les modules requis par ce module
- les packages qu'il expose pour permettre l'utilisation de leur classes plublic par d'autres modules
- les packages sur lesquels l'introspection est autorisé dans des classes d'autres modules
- les services qu'il expose et/ou qu'il consomme

Cette description peut aussi être complété avec des informations plus spécifiques

Chaque module possède un descripteur de module. Ce descripteur est un fichier source Java dont le nom est obligatoirement module-info.java. Ce fichier doit être compilé pour obtenir un fichier module-info.class.

Un descripteur de module utilise une syntaxe spécifique utilisant des mots clés contextuels de la forme :

```
[open] module <nom-module> {
[export <nom-package> [to <nom-module>]]*
[requires [transitive] <nom-module>] *
[opens <nom-packa> [to <nom-module]]*
[provides <type-service> with <nom-classe>]*
[uses <type-service>]*
}
```

35.3.1. Le nommage d'un module

La première chose à définir est le nom du module. Le nom d'un module est important car c'est ce nom qui sera utilisé pour exprimer qu'il est une dépendance d'un autre module.

Le nom d'un module est composé d'un ou plusieurs identifiants valide en Java chacun séparé par un caractère point. Ce nom doit respecter les mêmes règles que celles du nom des packages notamment les éléments qui composent le nom doivent être des identifiants Java valides.

Le choix du nom d'un module devrait prendre en compte au moins trois contraintes :

- des noms suffisamment longs pour être descriptif
- suffisamment court pour être mémorisé
- et unique pour éviter les conflits

Si vous contrôlez tous les modules qui compris tous les modules qui sont des dépendances, alors il est possible de nommer les modules à sa guise et de changer leur nom à tout moment.

Si le module doit être publié pour être utilisé par d'autres, alors il faut garantir une certaine unicité sur le nom du module. Il y a deux manières principales d'y parvenir :

- utiliser un nom de domaine inverse. Cette solution est historiquement utilisée notamment pour le nommage des packages qui peuvent aussi requérir une unicité
- utiliser un nom de module qui commence par le nom du projet concerné

L'utilisation du nom de domaine inversé comme préfixe garantie une unicité plus forte mais c'est aussi très verbeux. Mais les informations les moins utiles se trouvent au début du nom (par exemple com, net, fr, ...). De plus, un changement du nom de domaine implique de renommer le module ce qui risque de compliquer la gestion des dépendances qui utilise le nom des modules.

Actuellement la pratique la plus répandue est que le nom du module est tout ou partie du nom d'un package qu'il contient. Généralement le nom du package utiliser le nom de domaine inversé. La raison est la même : garantir une certaine unicité dans le nom de chaque module.

Cependant, le fait d'utiliser les mêmes conventions de nommage pour le nom des modules et des packages peut prêter à confusion d'autant que le fichier de description d'un module contient des noms de modules et des noms de packages.

Chaque module doit avoir un nom qui doit être unique parmi ceux présents dans le modulepath d'une application. Généralement le nom du module choisi correspond à tout ou partie du nom d'un des packages contenu dans le module.

Le nom du module ne devrait pas inclure le groupId s'il est construit avec Maven car un module est plus abstrait que l'artefact qui le définit.

Il n'est pas recommandé de terminer le nom d'un module par un chiffre : dans ce cas, le compilateur émet un avertissement.

Résultat :

```
C:\java\workspace\src>javac module-info.java
module-info.java:1: warning: [module] module name component java9 should avoid terminal digits
module fr.jmdoudoux.dej.java9 {
                       ^
1 warning
```

35.3.2. Le descripteur de module : le fichier module-info.java

Le descripteur de module contient des méta-données concernant le module. Un module peut dans son descripteur :

- déclarer ses dépendances : cela permet de définir les types utilisables dans le code du module en plus des types définis dans le module lui-même
- exporter les packages pour lesquels le module va permettre l'accès aux classes publiques au module qui vont en dépendre
- ouvrir les packages dans les lesquels les autres modules pourront faire de l'introspection sur des éléments non public
- déclarer les services fournis ou consommés par le module

Le descripteur de module doit obligatoire être nommé module-info.java et être situé à la racine de l'arborescence des répertoires source du module. Pour des raisons de compatibilité, le nom du descripteur de module contient un tiret, ce qui n'est pas légal pour un identifiant Java. Ce choix avait déjà été utilisé pour les fichiers package-info.java pour les mêmes raisons.

C'est un fichier source Java qui sera compilé et inclus dans le module.

Le fichier module-info.java doit être compilé par le compilateur javac comme tout autre fichier source Java pour obtenir un fichier module-info.class.

Bien que cela soit un fichier .java, la syntaxe utilisée dans ce fichier n'est pas du code Java mais une syntaxe particulière permettant de décrire le module.

La syntaxe générale de la déclaration d'un module est de la forme :

```
{Annotation} [open] module Identifiant {.Identifiant} {
{ Directives }
}
```

Sa forme la plus basique contient le mot clé contextuel module suivi du nom du module suivi d'une paire d'accolades.

Exemple (code Java 9) :

```
module fr.jmdoudoux.dej.monapp {
}
```

Cette forme basique définit simplement un module en lui indiquant un nom. Un descripteur de module doit obligatoirement définir le nom du module.

Les informations de description relatives au module sont fournies entre les accolades en utilisant des directives :

Directive	Rôle
requires	Indiquer une dépendance du module. Ce mot clé doit être suivi du nom du module dépendant
transitive	S'utilise après le mot clé requires pour indiquer que tous les modules qui auront ce module en dépendances auront aussi implicitement le module précisé après transitive en dépendance
exports	Définir un package comme étant exposé en dehors du module ; Permet d'indiquer que les classes publiques du package précisé seront accessibles à l'extérieure du module
opens	Indiquer que les éléments du package précisé seront accessibles à l'exécution par l'API Introspection quelque soit leur niveau de visibilité et autorise le chargement de chargement de ressources contenues dans le package
open	Indiquer sur un module qu'il est possible de faire de l'introspection sur des éléments et de permettre un chargement des ressources de tous les packages de celui-ci
uses	Indiquer que le module utilise le service précisé sous la forme du nom pleinement qualifié de sa classe ou interface

provides ... with ...	Indiquer que le module fournit une implémentation du service précisé grâce à la classe dont le nom pleinement qualifié suit with

La description d'un module utilise une syntaxe particulière avec ces mots clés contextels :

Syntaxe	Rôle

module nom.du.module	Déclarer un module dont le nom est nom.du.module
	Il est important de choisir judicieusement le nom d'un module car c'est l'identifiant du module.
	Le système de module s'appuie sur le nom d'un module. Les noms en conflit ou qui changent causent des problèmes, il est donc important que le nom d'un module soit :
	• globalement unique dans le module-path
	• stable
	Obligatoire
	Syntaxe :
	`module nom_du_module {`
	`  // ...`
	`  }`

requires nom.du.module	Définir que le module à besoin du module nom.du.module comme dépendance. Cela permet au module d'accéder à tous les types public des packages exportés par le module cible Les dépendances doivent être déclarées explicitement en utilisant la directive requires suivi du nom du module. Ces dépendances peuvent être des modules du JDK ou des dépendances tierces. Il existe deux exceptions : • le module de base nommé java.base qui est implicitement requis car il contient des types requis par tout code Java • les dépendances déclarées transitives dans la description d'une dépendance Un des objectifs de JPMS est d'assurer une configuration fiable. La compilation et le lancement d'une application échouent si un module requis avec le bon nom n'est pas trouvé. Syntaxe : `requires nom_du_module;`
requires transitive nom.du.module	Chaque module qui dépend du module dépend automatiquement du module nom.du.module
exports nom.du.package	Exporter le package nommé nom.du.package : les types public définis dans le package seront utilisables par d'autres modules. Seules les classes public des packages exportés sont accessibles en dehors d'un module. Pour être accessible en dehors du module, une classe doit : • être public • son package doit être exporté en utilisant le mot clé exports • le module qui en a besoin doit déclarer le module comme une dépendance en utilisant le mot clé requires Si ces règles ne sont pas respectées alors une erreur survient aussi bien à la compilation qu'à

	l'exécution. La visibilité public change donc avec Java 9 lors de la mise en oeuvre des modules. Jusqu'à Java 8 inclus, une classe public est accessible par toutes les autres classes du classpath. A partir de Java 9, par défaut une classe publique n'est accessible que par les autres classes du module. Pour pouvoir être utilisée à l'extérieur du module, le package contenant la classe public doit être explicitement exporté. Les classes qui ne sont pas public d'un package exportés ne sont pas accessibles à l'extérieur du module. Syntaxe : `exports nom_du_package;`
exports nom.du.package to nom.du.module	Exporter le package nommé nom.du.package : les types public définit dans le package seront utilisable uniquement par le module nommé nom.du.module
uses nom.du.type	Définir le module comme étant un consommateur du service défini par le type nom.du.type
provides nom.du.type with nom.du.type.impl	Enregistrer la classe nom.du.type.impl comme étant le fournisseur d'une implémentation pour le service de type nom.du.type
opens nom.du.package	Autoriser les autres modules à utiliser la réflexion sur les types du package nom.du.package ou de charger des resources qu'il contient
opens nom.du.package to nom.du.module	Autoriser uniquement le module nommé nom.du.module à utiliser la réflexion sur les types du package nom.du.package ou de charger des resources qu'il contient

Module, requires, exports, opens, ... ne sont pas ajoutés dans la liste des mots clés réservés du langage Java. Ils sont considérés comme des « mots clés contextuels (contextual keywords) » car ils ne sont utilisables en tant que mots clés que dans un descripteur de module, même si c'est un fichier Java.

Il est donc possible d'utiliser module, requires, exports, opens, ... comme identifiants dans du code source Java.

Les IDE qui proposent un support pour Java 9 propose une assistance plus ou moins élaborée pour faciliter la rédaction et la maintenance des fichiers module-info.java.

Exemple (code Java 9) :

```
module fr.jmdoudoux.dej.util {
  requires com.google.guava;

  exports fr.jmdoudoux.dej.util;
}
```

L'exemple ci-dessus définit un module :

- dont le nom est fr.jmdoudoux.dej.util
- qui possède une dépendance explicite vers le module com.google.guava et implicite vers java.base
- qui expose les classes du package fr.jmdoudoux.dej.util. Les autres packages contenus dans le module ne seront accessibles que par le module lui-même

35.3.3. L'export des packages

La directive exports permet de préciser le nom d'un package qui sera accessible par d'autres modules.

Par défaut, aucune classe n'est accessible en dehors du module, même si celles-ci sont déclarées avec le modificateur de visibilité public. Pour permettre à des classes publiques d'être accessibles en dehors du module, il faut explicitement exporter le ou les packages contenant ces classes.

La directive exports précise le nom d'un package du module à exporter : les modules qui auront le module en dépendance auront accès aux classes public et protected et à leur membre public et protected. Cela permet aussi un accès par introspection à ces types et membres public.

Pour exporter un package, il faut utiliser une syntaxe composée du mot clé contextuel exports suivi du nom du package suivi du caractère point-virgule.

La syntaxe générale est de la forme :

```
export nom_du_package to nom_module [, nom_module]* ;
```

La directive exports définit un package comme étant exposé en dehors du module. Tous les types public du package pourront être utilisés par un module qui dépend du module.

Le descripteur de module peut avoir aucune, une ou plusieurs directives exports.

Il n'est pas nécessaire d'exporter tous les packages. Par contre, pour utiliser un type public définit dans un package d'un module, ce package doit être exporté.

Exemple (code Java 9) :

```
module fr.jmdoudoux.dej.monapp {
  exports fr.jmdoudoux.dej.monapp.main;
}
```

La référence fournie après l'instruction exports ne peut être qu'un seul nom de package.

Les sous-packages d'un package exportés ne sont pas accessibles : il faut exporter explicitement tous les sous-packages concernés un par un.

Il est possible d'utiliser des exports qualifiés : dans ce cas, seuls le ou les modules précisés auront accès aux classes publiques du package. Cela permet de renforcer encore plus l'encapsulation.

Pour utiliser une exportation qualifiée, il faut utiliser le mot clé exports suivi du nom du package suivi du mot clé to suivi du module concerné suivi du caractère point-virgule. Plusieurs modules peuvent être précisés en les séparant par une virgule.

Exemple (code Java 9) :

```
module fr.jmdoudoux.dej.monapp {
  exports fr.jmdoudoux.dej.monapp.utils to fr.jmdoudoux.dej.monapp.service;
}
```

Dans l'exemple ci-dessus, seul le module fr.jmdoudoux.dej.monapp.service peut avoir accès aux classes publiques du package fr.jmdoudoux.dej.monapp.utils.

35.3.4. La déclaration des dépendances

Un module a fréquemment besoin d'utiliser des classes d'autres modules. Dans ce cas, les modules requis deviennent des dépendances du module. Un module peut ainsi dépendre d'un ou plusieurs autres modules. Il est obligatoire de définir toutes ces dépendances explicitement dans le descripteur de module.

La gestion des dépendances dans le système de modules repose sur trois concepts :

- fiabilité de la configuration (reliable configuration) :
- lisibilité (readibility)
- accessibilité (accessiblity)

Elle se concrétise explicitement dans le descripteur de module grâce à l'utilisation d'une directive requires suivi du nom du module concerné. Si un module A dépend d'un module B :

- la fiabilité de la configuration est assurée par la déclaration de la dépendance vers le module B dans le descripteur de module A et par la vérification de la présence du module A et B dans le module path
- la lisibilité est assurée par la directive requires B qui permet au module A de lire le module B
- l'accessibilité est assurée par la directive requires B qui permet au module A d'accéder aux classes public des packages exportés du module B

La dépendance d'un module vers un autre peut prendre deux formes :

- le concept de lisibilité (Readability) : le module dépendant dépend d'un autre mais cette relation est invisible pour les autres modules. Lorsqu'un module dépend directement d'un autre, le code du premier module pourra utiliser les types public des packages exportés du second module. On dit que le premier module lit (read) le second ou, de manière équivalente, que le second module est lisible par le premier. C'est le cas le plus courant qui induit une déclaration de la dépendance de manière explicite
- le concept de lisibilité implicite (Implied Readability) : le code qui veut appeler le module dépendant peut être amené à utiliser des types de son module dépendant. Mais il ne peut pas le faire s'il ne lit pas également le second module. Par conséquent, pour que le module dépendant soit utilisable, les modules clients devraient tous dépendre explicitement de ce second module. Identifier et résoudre manuellement ces dépendances dîtes transitives serait une tâche fastidieuse et source d'erreurs. La déclaration de modules permet qu'un module puisse accorder la lisibilité à d'autres modules, dont il dépend, à tout module qui en dépend. Le cas d'usage correspond à un module qui dépend d'un autre et expose les types du module dépendant dans sa propre API publique.

En général, l'utilisation d'une dépendance transitive est recommandée, si un module exporte un package contenant un type dont la signature ou la valeur de retour fait référence à un package dans un second module, alors la déclaration du premier module doit inclure une dépendance publique requise sur le second. Cela garantit que les autres modules qui dépendent du premier module seront automatiquement capables de lire le second module et, par conséquent, d'accéder à tous les types public des packages exportés par ce module.

Avec JPMS, un module doit lire un autre module pour pouvoir utiliser son API en déclarant sa dépendance.

Une dépendance est exprimée par une directive requires suivi du nom d'un module, indépendamment du fait que ce module existe ou non. Elle permet au module d'avoir accès aux types public des packages exportés par le module précisé.

Le descripteur de module peut utiliser deux directives pour déclarer les dépendances d'un module :

- requires : déclare une lisibilité vers un module. La directive requires spécifie le nom d'un module vers lequel le module a une dépendance. Exemple : le module A dépend d'un autre module B
- requires transitive : déclare une lisibilité implicite, la dépendance vers un troisième module est propagée, permettant au premier de lire le troisième sans en dépendre explicitement. Exemple : le module A dépend de B, le module B dépend de manière transitive au module C, alors le module A dépend aussi implicitement de C

Important : il n'est pas permis d'avoir des dépendances circulaires entre les modules. En d'autres termes, si le module A nécessite le module B, alors le module B ne peut pas également nécessiter le module A. Cette dépendance cyclique peut aussi inclure plus de deux modules. Dans tous les cas, le graphe de dépendance des modules doit être un graphe acyclique.

35.3.4.1. Les dépendances explicites

La définition d'une dépendance se fait en utilisant le mot clé requires suivi du nom du module qui est une dépendance suivie du caractères point-virgule.

Exemple (code Java 9) :

```
module fr.jmdoudoux.dej.monapp {
  requires fr.jmdoudoux.dej.monapp.utils;
}
```

La référence fournie après le mot clé requires ne peut être qu'un nom de module.

Dans l'exemple ci-dessus :

- le module déclare une dépendance à la compilation et à l'exécution vers le module nommé fr.jmdoudoux.dej.monapp.utils
- le module a ainsi accès aux classes public des packages exportés par le module nommé fr.jmdoudoux.dej.monapp.utils

Le descripteur de module peut avoir aucune, une ou plusieurs directives requires. La directive requires est donc facultative : si un module n'a pas de directive requires alors il n'a pas de dépendances vers d'autres modules et il est donc un module indépendant.

Par défaut, le module java.base est automatiquement en dépendance. Donc si la déclaration d'un module ne définit pas explicitement une dépendance vers le module java.base, alors le module a une dépendance implicite vers le module java.base. Le module java.base n'a aucune dépendance.

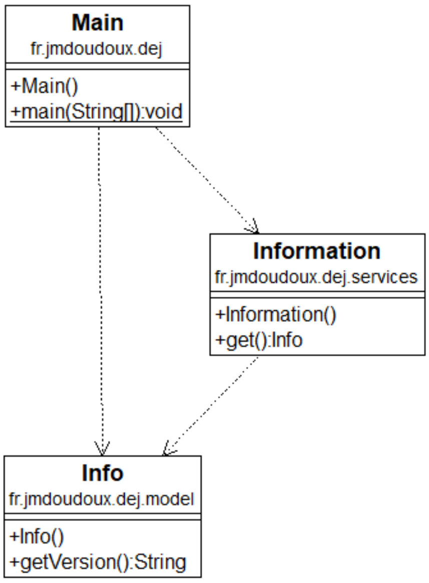
35.3.4.2. Les dépendances transitives

La lisibilité n'est pas transitive par défaut. La lisibilité implicite implique que tout module qui lit le module lit également implicitement le module transitif. La lisibilité implicite a été introduite pour permettre qu'un module qui utilise les types d'un autre module dans sa propre API publique soit automatiquement utilisable sans que le module appelant doive déclarer explicitement la dépendance vers le module concerné.

Le mot-clé requires peut être suivi du mot clé transitive. Cela fait que tout module qui requiert le module courant a une dépendance implicite déclarée sur le module spécifié par la directive requires transitive.

Cela implique que si le module A requiert le module B, et que le module B requiert le module C de manière transitive, alors le module A requiert aussi implicitement le module C

Pour illustrer le concept, trois classes sont utilisées chacune encapsulées dans un module qui expose leur package.



Le moduleC contient la classe publique Info dont le package fr.jmdoudoux.dej.model est exposé.

Exemple (code Java 9) :

```
package fr.jmdoudoux.dej.model;

public class Info {

  public String getVersion() {
    return "2.0";
  }
}
```

Exemple (code Java 9) :

```
module moduleC {
  exports fr.jmdoudoux.dej.model;
}
```

Le moduleB contient la classe publique Information dont le package fr.jmdoudoux.dej.services est exposé. Comme une méthode renvoie une instance de type Info, il a une dépendance vers moduleC.

Exemple (code Java 9) :

```
package fr.jmdoudoux.dej.services;

import fr.jmdoudoux.dej.model.Info;

public class Information {

  public Info get() {
    return new Info();
  }
}
```

Exemple (code Java 9) :

```
module moduleB {
  exports fr.jmdoudoux.dej.services;
  requires moduleC;
}
```

Le moduleA contient la classe publique Main qui utilise les classes Info et Information. Il a donc une dépendance explicite vers les modules moduleB et moduleC.

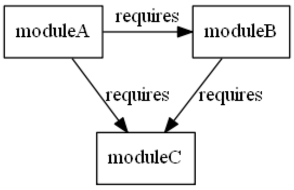
Exemple (code Java 9) :

```
package fr.jmdoudoux.dej;

import fr.jmdoudoux.dej.model.Info;
import fr.jmdoudoux.dej.services.Information;

public class Main {
  public static void main(String[] args) {
    Information information = new Information();
    Info info = information.get();
    System.out.println(info.getVersion());
  }
}
```

Exemple (code Java 9) :

```
module moduleA {
  requires moduleB;
  requires moduleC;
}
```

Les dépendances telles que définies dans les descripteurs de module sont les suivantes :



Le moduleA définit explicitement une dépendance vers les modules moduleB et moduleC.

Il est possible d'utiliser la directive requires transitive pour permettre à tous les autres modules qui auront le module comme dépendance d'avoir une dépendance implicite vers la dépendance transitive.

Si un module exporte un package contenant un type dont la signature utilise un package d'un second module, la déclaration du premier module peut utiliser une dépendance transitive sur le second. Dans ce cas, cela garantira que les modules qui dépendent du premier module seront automatiquement capables de lire le second module et, par conséquent, d'accéder à tous les types des packages exportés par ce module.

L'instruction requires peut être suivi de l'instruction transitive qui permet de déclarer une dépendance implicite vers un autre module : tous les modules qui auront une dépendance vers le module auront accès aux packages exposés par le module requis de manière transitive. C'est par exemple utile si un type définit dans une dépendance transitive est utilisé dans une classe d'un package exposé par le module.

La syntaxe repose sur l'utilisation de la directive required transitive :

```
requires transitive <nom_module>;
```

Une dépendance implicite d'un module est déclarée avec la directive requires transitive. Tout module qui requiert (avec requires) un module qui contient une directive requires transitive requiert aussi implicitement le module déclaré dans la directive requires transitive. La directive requires transitive introduit une lisibilité implicite.

Dans le descripteur de moduleB, la dépendance vers le moduleC est déclarée transitive.

Exemple (code Java 9) :

```
module moduleB {
  exports fr.jmdoudoux.dej.services;
  requires transitive moduleC;
}
```

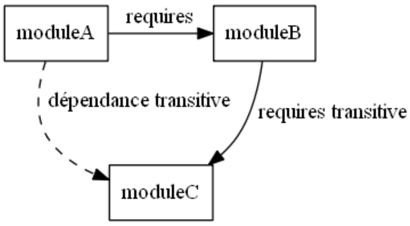
La directive requires transitive spécifie une dépendance sur un autre module et garantit que les autres modules qui lisent le module lisent également cette dépendance.

Le descripteur du moduleA ne déclare donc plus que la dépendance vers le moduleB.

Exemple (code Java 9) :

```
module moduleA {
  requires moduleB;
}
```

Les dépendances telles définies dans les descripteurs de module sont les suivantes :



Idéalement la définition de dépendances transitives ne devrait être utilisée que si l'API publique du module en dépend. Par exemple, si un module contient une méthode qui est publiquement exposée dont la signature ou la valeur de retour utilise une classe de la dépendance transitive.

35.3.4.3. Les patterns utilisables avec la lisibilité implicite

La lisibilité implicite induite par les dépendances transitives permet de mettre en oeuvre quelques patterns qui peuvent être utiles. Ils reposent sur le fait qu'avec elle, un client peut consommer les API de divers modules sans en dépendre explicitement, s'il dépend d'un module qui requiert de manière transitive les API utilisées.

La lisibilité implicite peut être utilisée pour mettre en oeuvre différents patterns :

- agrégation de modules (aggregator module)
- fractionnement des modules (splitting modules)
- fusion de modules
- renommage d'un module

L'agrégation de modules (aggregator module)

Les modules agrégateurs ont une responsabilité spécifique : regrouper les fonctionnalités de modules connexes en une seule unité.

Ce pattern est utilisé dans les modules du JDK lui-même :

Résultat :

```
C:\java>java --describe-module java.se
java.se@9.0.1
requires java.sql.rowset transitive
requires java.sql transitive
requires java.datatransfer transitive
requires java.security.jgss transitive
requires java.logging transitive
requires java.desktop transitive
requires java.base mandated
requires java.xml.crypto transitive
requires java.management.rmi transitive
requires java.rmi transitive
requires java.security.sasl transitive
requires java.naming transitive
requires java.management transitive
requires java.instrument transitive
requires java.compiler transitive
requires java.xml transitive
requires java.scripting transitive
requires java.prefs transitive
```

⚠ L'utilisation de modules agrégateurs met les modules qui les ont en dépendances dans la situation où ils utilisent en interne des API de modules dont ils ne dépendent pas explicitement. Cela contredit la recommandation sur l'utilisation des dépendances transitives dans le cas où les types sont utilisés dans la signature ou la valeur de retour.

Le fractionnement des modules (Splitting modules)

Un module peut être décomposé en modules plus spécialisés sans implications de compatibilité s'il devient un agrégateur pour les nouveaux modules.

Dans l'exemple ci-dessous, moduleA est découpé en trois modules. Pour ne pas impacter les modules qui ont moduleA en dépendance, les dépendances vers les modules moduleA1, moduleA2 et moduleA3 sont déclarées transitives.

Résultat :

```
module moduleA {
    requires transitive moduleA1;
    requires transitive moduleA2;
    requires transitive moduleA3;
}
```

La fusion de modules

Par exemple si moduleA, moduleB et moduleC sont fusionnés dans moduleD :

Résultat :

```
module moduleA {
    requires transitive moduleD;
}
```

Résultat :

```
module moduleB {
    requires transitive moduleD;
}
```

Résultat :

```
module moduleC {
    requires transitive moduleD;
}
```

Attention dans ce cas, les dépendances sont plus larges

Le renommage d'un module

La lisibilité implicite peut être utilisée pour renommer ou fournir un alias à un module.

Par exemple, si le moduleA est renommé en moduleB

Résultat :

```
module moduleA {
    requires transitive moduleB;
}
```

35.3.4.4. Récapitulatif de la lisibilité

Si le moduleA nécessite le moduleB alors le système de modules :

- impose la présence de moduleB dans le modulepath : configuration fiable (reliable configuration)
- permet à moduleA de lire moduleB : lisibilité (readibility)
- permet au code de moduleA d'accéder aux classes publiques des packages exportés de moduleB : accessibilité (accessiblity)

Il en va exactement de même si moduleA dépend de moduleB de manière transitive : la moduleB doit être présent, peut être lue et accessible. En fait, pour moduleA et moduleB, le mot-clé transitif ne change rien. Par contre, pour les modules qui dépendent de moduleA, il obtienne une dépendance implicite vers moduleB.

Les dépendances transitives sont recommandées lorsqu'un module dont l'API publique accepte ou renvoie le type d'un autre module.

35.3.5. Les dépendances optionnelles

La gestion des dépendances par le système de module est par défaut très strict : toutes les dépendances doivent être déclarées explicitement et elles doivent être accessibles à la compilation et à l'exécution avec une vérification dans ces deux contextes à leur démarrage lors d'une étape de résolution.

La directive requires indique qu'un module est requis à la compilation et à l'exécution. Par conséquent à la résolution du module (traitement du descripteur de module et résolutions des dépendances), lorsque le système de modules rencontre une telle directive, il recherche dans l'ensemble des modules observables et émet une erreur s'il ne trouve pas le module. Une erreur est donc émise par le compilateur et la JVM si ce module n'est pas trouvé dans les modules du JDK ou dans le module-path lors de la résolution des modules.

C'est intéressant pour des dépendances qui sont requises mais il est parfois nécessaire d'avoir des dépendances optionnelles : parfois, certaines dépendances n'ont pas toujours besoin d'être présentes à l'exécution.

Pour rendre une dépendance optionnelle vers un module, le système de modules propose de faire suivre la directive requires par le modificateur static.

La directive requires static définit une dépendance vers un module de manière optionnelle : le module devra être trouvé par le compilateur mais il n'y aura pas d'obligation à ce que le module soit trouvé par la JVM lors de la résolution des modules. Ainsi une dépendance optionnelle est obligatoire dans la phase statique, lors de la compilation, mais elle est facultative dans la phase dynamique, lors de l'exécution.

Si un module A déclare dans son descripteur de module requires static B, alors le système de modules va se comporter différemment :

- à la compilation : module B doit être présent dans le module path sinon une erreur est retournée par le compilateur
- à l'exécution : module B n'est pas l'obligation d'être présent. S'il est présent, module A pourra lire le module B

Ceci ne concerne que la déclaration de la dépendance. Indépendamment de la facilité à déclarer un module optionnel, l'utilisation dans le code de fonctionnalités de ce module est moins triviale. Les modules optionnels étant ignorés à la résolution lors de l'exécution, le code doit prendre en compte que les types exposés d'un module optionnel peuvent ne pas être présents à l'exécution.

Si une dépendance optionnelle est présente, le système de modules configure sa lisibilité et il est donc possible d'utiliser ses types exposés. Si elle est absente, alors une exception de type NoClassDefFoundError est levée lors de l'utilisation d'un de ses types exposés.

D'une manière générale, lorsque le code exécuté fait référence à un type, la JVM vérifie s'il est déjà chargé. Si ce n'est pas le cas, elle utilise un ClassLoader pour le faire et si cela échoue, alors une exception de type NoClassDefFoundError est levée.

Le code utilisant des types inclus dans un module optionnel doit donc prendre en compte de manière défensive la levée d'une exeption de type NoClassDefFoundError lors de la création d'une instance via l'opérateur new ou ClassNotFoundException lors du chargement via l'API Reflection (méthodes forName(), loadClass() et findSystemClass() de la classe Class).

35.3.6. L'utilisation de l'API Reflection et l'accès aux ressources

L'API Reflection est fréquemment mise en oeuvre notamment par des frameworks open source : elle est largement utilisée par exemple par Spring, Hibernate, ...

Avec les modules, l'utilisation de l'API Reflection est restreinte. Sans les modules, il est possible d'utiliser cette API sur n'importe quel élément contenu dans le classpath incluant ceux de Java Core et ce quelle que soit la visibilité définie dans le code.

Dans les modules, par défaut, l'utilisation de l'API Reflection n'est possible que sur des éléments public de packages exportés. Pour les autres éléments, il faut obligatoirement autoriser l'accès aux packages sur lesquels on souhaite pouvoir faire des accès via l'API Reflection.

Il n'est plus possible d'utiliser l'API Reflection dans d'autres modules sans certaines contraintes :

- Sur des types ou des membres publics : le package du type concerné doit être exporté
- Sur des types ou des membres non publics : le package doit être ouvert (ou le module lui-même peut être ouvert)

Les accès via l'API Reflection sur des types ou des membres dans le module lui-même sont possibles sans contraintes quelques soit le niveau de visibilité.

L'ouverture d'un package permet de réaliser des accès via l'API Reflexion sur tous les types et sur tous leurs membres du package que celui-ci soit exporté ou non.

Il existe deux types de modules :

- module standard (normal module) : par défaut, l'accès par introspection aux éléments qu'il contient n'est pas possible
- module ouvert (open module) : permet un accès par introspection à tous les packages qu'il contient

Le type de module détermine les accès aux types du module et aux membres de ces types pour le code situé en dehors du module. Un module standard (sans le modificateur open) permet l'accès à la compilation et à l'exécution uniquement aux types publics des packages exportés. Un module standard permet un accès par introspection :

- aux éléments publics des packages exportés
- à tous les éléments des packages ouverts

Lors de la compilation, il n'est pas possible d'utiliser des types d'un package ouvert : seuls les accès via l'API Reflection à l'exécution sont permis même sur des types et des membres non publics.

Pour permettre l'introspection sur les types ou les membres non public d'un package par un autre module, il faut ouvrir le package qui les contient. Cela se fait avec l'instruction opens suivi du nom du package concernés.

Exemple (code Java 9) :

```
module fr.jmdoudoux.dej.persistance {
  exports fr.jmdoudoux.dej.entite;
  opens fr.jmdoudoux.dej.entite;
}
```

La directive opens précise le nom d'un package du module à ouvrir : les modules qui auront le module en dépendance auront accès via l'API Reflection à tous les types et tous les membres du package.

Un package ouvert peut être qualifié, comme avec les exportations, uniquement vers certains modules.

Exemple (code Java 9) :

```
module fr.jmdoudoux.dej.persistance {
  exports fr.jmdoudoux.dej.entite;
  opens fr.jmdoudoux.dej.entite to org.hibernate.orm.core;
}
```

Il est aussi possible d'ouvrir tout un module en utilisant l'instruction open avant l'instruction module. Un module ouvert ouvre simplement tous ses packages.

Exemple (code Java 9) :

```
open module fr.jmdoudoux.dej.monapp {
}
```

Dans ce cas, le module devient un module ouvert : ce module permet l'utilisation de l'API Reflection sur tous les types et tous les membres de tous ces packages. Il n'est alors plus nécessaire d'ouvrir aucun package puisque dans ce cas, ils sont tous ouverts.

Un module ouvert (avec le modificateur open) permet l'accès à la compilation aux types public des packages exportés, mais l'autorise également, à l'exécution, aux types de tous ses packages via l'API Reflexion.

Les accès via l'API Reflection sur les types ou les membres dans un module ouvert sont possibles sans contraintes quelques soit le niveau de visibilité.

Plusieurs vérifications sont faites par le compilateur et peuvent émettre une erreur si elles échouent.

Le compilateur émet une erreur si plus d'une directive opens est utilisée sur le même package.

Résultat :

```
C:\java\TestModules\src>javac module-info.java
module-info.java:3: error: duplicate or conflicting opens: fr.jmdoudoux.dej.java
  opens fr.jmdoudoux.dej.java;
                        ^
1 error
```

Le compilateur émet une erreur si une directive opens est utilisée sur un module déclaré avec la directive open.

Résultat :

```
C:\java\TestModules\src>javac module-info.java
module-info.java:2: error: 'opens' only allowed in strong modules
  opens fr.jmdoudoux.dej.java;
  ^
1 error
```

Le compilateur émet une erreur si un même module est précisé plusieurs fois à la suite d'une directive to.

Résultat :

```
C:\java\TestModules\src>javac module-info.java
module-info.java:2: error: duplicate or conflicting opens to module: fr.jmdoudoux.dej.util
  opens fr.jmdoudoux.dej.java to fr.jmdoudoux.dej.util, fr.jmdoudoux.dej.util;
                                                                        ^
1 error
```

Les ressources (fichiers qui ne sont pas des .class) sont aussi encapsulées par défaut dans un module. Ainsi pour permettre leur accès de l'extérieure du module, il est nécessaire d'ouvrir le ou les packages qui les contiennent ou d'ouvrir le module.

Pour accéder à une ressource dans le module lui-même, il est préférable d'utiliser les méthodes getResourceAsStream() des classes Class ou Module plutôt que celle de la classe ClassLoader.

Les méthodes de la classe ClassLoader ne peuvent accéder qu'à des ressources dont les packages qui les contiennent sont ouverts.

35.4. Les règles d'accès

Les modules introduisent un nouveau niveau de visibilité : il n'est pas possible d'utiliser les types public à la compilation ou à l'exécution si leur package n'est pas exporté.

Le code contenu dans le module peut accéder aux types et à leurs membres en respectant les règle de visibilité de tous les packages du module, à la compilation et à l'exécution.

Lorsque de l'on souhaite utiliser un module présent dans le modulepath, la JVM va appliquer plusieurs règles pour permettre à une classe d'un package A d'un module A de pouvoir utiliser une classe d'un package B d'un module B :

- la classe dans le package B est public
- le module B contenant le package B l'exporte dans sa déclaration
- Le module A déclare une dépendance vers le module B

Ces trois règles doivent être respectées pour permettre une utilisation de la classe par le module A.

	Accès à la compilation	Accès par introspection
Par défaut	Non	Non
Export	Accès aux éléments public du package pour les modules qui l'auront en dépendances	Accès aux éléments public du package pour les modules qui l'auront en dépendances
Export qualifié	Accès aux éléments public du package uniquement pour les modules précises qui l'auront en dépendances	Accès aux éléments public du package uniquement pour les modules précisés qui l'auront en dépendances
Open	Non	Accès à tous les éléments du package quelle que soit la visibilité
Open qualifié	Non	Accès à tous les éléments du package quelle que soit la visibilité pour les modules précisés
Opens	Non	Accès à tous les éléments de tous les packages quelle que soit la visibilité

35.5. La qualité des descripteurs de module

Le descripteur de module est un élément important d'un module : il est donc important de garantir la qualité de son contenu comme pour tout autre fichier qui compose le code source. Comme il définit comment le module va interagir avec les autres modules, il peut être amener à évoluer.

Il n'y a pas d'ordre imposé dans l'utilisation des directives mais il est intéressant d'en respecter un afin d'en faciliter la lecture et la maintenance comme dans les classes. Par exemple, le JDK utilise l'ordre suivant :

- requires (avec static et transitive)
- exports
- exports to
- opens
- opens to
- uses
- provides

Les opinions sur la documentation du code, comme la Javadoc ou les commentaires en ligne, varient énormément, mais quelle que soit la position sur les commentaires, il faut les utiliser dans les descripteurs de modules. Il est toujours intéressant de documenter les raisons d'une décision spécifique. Dans une déclaration de module, cela pourrait ajouter des commentaires dans différentes situations, par exemple :

- une exportation qualifiée pour expliquer pourquoi elle n'est pas une API publique, mais est partiellement accessible
- un paquet ouvert expliquant les frameworks qui doivent y avoir accès
- une dépendance facultative pour expliquer les raisons de l'absence du module
- ...

Les descripteurs de module sont du code source et en tant que tel ils doivent être pris en compte lors des revues de codes. Durant ces revues, il est intéressant de vérifier différentes choses, notamment :

- vérifier l'export d'un package et s'assurer que les API exposées se limitent bien uniquement à celles requises
- vérifier l'utilité des exports qualifiés
- vérifier les dépendances

- vérifier que le code prend en compte l'absence des modules optionnels
- ...

Partie 5 : La programmation parallèle et concurrente

Partie 5 :
La programmation
parallèle
et concurrente

Le JDK fournit un certain nombre d'API pour mettre en oeuvre des traitements parallélisés éventuellent exécutés en concurrence.

Cette partie contient les chapitres suivants :

- Le multitâche : décrit les principaux fondamentaux des traitements multitaches et de leurs mises en oeuvre avec Java
- Les threads : présente et met en oeuvre les mécanismes des threads qui permettent de répartir différents traitements d'un même programme en plusieurs unités distinctes exécutées de manière "simultanée"
- L'association de données à des threads : détaille les solutions utilisables pour permettre d'associer des données à un thread
- Le framework Executor : détaille l'utilisation du framework Executor
- La gestion des accès concurrents : détaille différentes solutions pour gérer les accès concurrents dans les traitements en parallèle

36. Le multitâche

Chapitre 36

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Un thread est une unité d'exécution faisant partie d'un programme. Cette unité fonctionne de façon autonome et parallèlement à d'autres threads. En fait, sur une machine monoprocesseur, chaque unité se voit attribuer des intervalles de temps au cours desquels elles ont le droit d'utiliser le processeur pour accomplir leurs traitements.

La gestion de ces unités de temps par le système d'exploitation est appelée scheduling. Il existe deux grands types de scheduler:

- le découpage de temps utilisé par Windows et Macintosh OS jusqu'à la version 9. Ce système attribue un intervalle de temps prédéfini quel que soit le thread et la priorité qu'il peut avoir
- la préemption utilisée par les systèmes de type Unix. Ce système attribut les intervalles de temps en tenant compte de la priorité d'exécution de chaque thread. Les threads possédant une priorité plus élevée s'exécutent avant ceux possédant une priorité plus faible.

Le principal avantage des threads est de pouvoir répartir différents traitements d'un même programme en plusieurs unités distinctes pour permettre leurs exécutions "simultanées".

La classe java.lang.Thread et l'interface java.lang.Runnable sont les bases pour le développement des threads en Java. Par exemple, pour exécuter des applets dans un thread, il faut que celles-ci implémentent l'interface Runnable.

La suite de ce chapitre sera développée dans une version future de ce document

37. Les threads

Chapitre 37

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Un thread est une unité d'exécution faisant partie d'un programme. Cette unité fonctionne de façon autonome et parallèlement à d'autres threads. Le principal avantage des threads est de pouvoir répartir différents traitements d'un même programme en plusieurs unités distinctes pour permettre leurs exécutions "simultanées".

Sur une machine monoprocesseur, c'est le système d'exploitation qui alloue du temps d'utilisation du CPU pour accomplir les traitements de chaque threads, donnant ainsi l'impression que ces traitements sont réalisés en parallèle.

Sur une machine multiprocesseur, le système d'exploitation peut répartir l'exécution sur plusieurs coeurs, ce qui peut effectivement permettre de réaliser des traitements en parallèle.

Selon le système d'exploitation et l'implémentation de la JVM, les threads peuvent être gérés de deux manières :

- correspondre à un thread natif du système
- correspondre à un thread géré par la machine virtuelle

Dans les deux cas, cela n'a pas d'impact sur le code qui reste le même.

La JVM crée elle-même pour ses propres besoins plusieurs threads : le thread d'exécution de l'application, un ou plusieurs threads pour le ramasse-miettes, ...

La classe java.lang.Thread et l'interface java.lang.Runnable sont les bases pour le développement des threads en Java.

Le système d'exploitation va devoir répartir du temps de traitement pour chaque thread sur le ou les CPU de la machine. Plus il y a de threads, plus le système va devoir switcher. De plus, un thread requiert des ressources pour s'exécuter notamment un espace mémoire nommé pile. Il est donc nécessaire de contrôler le nombre de threads qui sont lancés dans une même JVM.

Cependant, l'utilisation de plusieurs threads améliore généralement les performances, notamment si la machine possède plusieurs coeurs, car dans ce cas plusieurs threads peuvent vraiment s'exécuter en parallèle. Il est aussi fréquent que les traitements d'un thread soient en attente d'une ressource : le système peut alors plus rapidement allouer du temps CPU à d'autres threads qui ne le sont pas.

L'utilisation de la classe Thread est d'assez bas niveau. A partir de Java 5, le package java.util.concurrency propose des fonctionnalités de plus haut niveau pour faciliter la mise en oeuvre de traitements en parallèle et améliorer les performances de la gestion des accès concurrents.

Ce chapitre contient plusieurs sections :

- L'interface Runnable
- La classe Thread
- Le cycle de vie d'un thread
- Les démons (daemon threads)
- Les groupes de threads
- L'obtention d'informations sur un thread
- La manipulation des threads

- ♦ Les messages de synchronisation entre threads
- ♦ Les restrictions sur les threads
- ♦ Les threads et les classloaders
- ♦ Les threads et la gestion des exceptions
- ♦ Les piles

37.1. L'interface Runnable

Cette interface doit être implémentée par toute classe qui contiendra des traitements à exécuter dans un thread.

Cette interface ne définit qu'une seule méthode : void run().

Dans les classes qui implémentent cette interface, la méthode run() doit être redéfinie pour contenir le code des traitements qui seront exécutés dans le thread.

Exemple :

```
package fr.jmdoudoux.dej;

public class MonTraitement implements Runnable {
  public void run() {
    int i = 0;
    for (i = 0; i > 10; i++) {
      System.out.println("" + i);
    }
  }
}
```

37.2. La classe Thread

La classe Thread est définie dans le package java.lang. Elle implémente l'interface Runnable.

Elle possède plusieurs constructeurs : un constructeur par défaut et plusieurs autres qui peuvent avoir un ou plusieurs des paramètres suivants :

- le nom du thread
- l'objet qui implémente l'interface Runnable l'objet contenant les traitements du thread
- le groupe auquel sera rattaché le thread

Constructeur	Rôle
Thread()	Créer une nouvelle instance
Thread(Runnable target)	Créer une nouvelle instance en précisant les traitements à exécuter
Thread(Runnable target, String name)	Créer une nouvelle instance en précisant les traitements à exécuter et son nom
Thread(String name)	Créer une nouvelle instance en précisant son nom
Thread(ThreadGroup group, Runnable target)	Créer une nouvelle instance en précisant son groupe et les traitements à exécuter
Thread(ThreadGroup group, Runnable target, String name)	Créer une nouvelle instance en précisant son groupe, les traitements à exécuter et son nom
Thread(ThreadGroup group, Runnable target, String name, long stackSize)	Créer une nouvelle instance en précisant son groupe, les traitements à exécuter, son nom et la taille de sa pile
Thread(ThreadGroup group, String name)	Créer une nouvelle instance en précisant son groupe et son nom

Un thread possède une priorité et un nom. Si aucun nom particulier n'est donné dans le constructeur du thread, un nom par défaut composé du préfixe "Thread-" suivi d'un numéro séquentiel incrémenté automatiquement lui est attribué.

La classe Thread possède plusieurs méthodes :

Méthode	Rôle
static int activeCount()	Renvoyer une estimation du nombre de threads actifs dans le groupe du thread courant et ses sous-groupes
void checkAccess()	Déterminer si le thread courant peut modifier le thread
void destroy()	Mettre fin brutalement au thread : ne pas utiliser car deprecated
int countStackFrames()	Deprecated
static Thread currentThread()	Renvoyer l'instance du thread courant
static void dumpStack()	Afficher la stacktrace du thread courant sur la sortie standard d'erreur
static int enumerate(Thread[] tarray)	Copier dans le tableau fourni en paramètre chaque thread actif du groupe et des sous-groupes du thread courant
static Map<Thread,StackTraceElement[]> getAllStackTraces()	Renvoyer une collection de type Map qui contient pour chaque thread actif les éléments de sa stacktrace
int getPriority()	Renvoyer la priorité du thread
ThreadGroup getThreadGroup()	Renvoyer un objet qui encapsule le groupe auquel appartient le thread
static boolean holdsLock(Object obj)	Renvoyer un booléen qui précise si le thread possède le verrou sur le monitor de l'objet passé en paramètre
void interrupt()	Demander l'interruption du thread
static boolean interrupted()	Renvoyer un booléen qui précise si une demande d'interruption du thread a été demandée
boolean isAlive()	Renvoyer un booléen qui indique si le thread est actif ou non
boolean isInterrupted()	Renvoyer un booléen qui indique si le thread a été interrompu
void join()	Attendre la fin de l'exécution du thread
void join(long millis)	Attendre au plus le délai fourni en paramètre que le thread se termine
void join(long millis, int nanos)	Attendre au plus les délai fourni en paramètres (ms + ns) que le thread se termine
void resume()	Reprendre l'exécution du thread préalablement suspendue par suspend( ). Cette méthode est deprecated
void run()	Contenir les traitements à exécuter
void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)	Définir le handler qui sera invoqué si une exception est levée durant l'exécution des traitements
static void sleep(long millis)	Endormir le thread pour le délai exprimé en millisecondes précisé en paramètre
static void sleep(long millis, int nanos)	Endormir le thread pour le délai précisés en paramètres
void start()	Lancer l'exécution des traitements : associer des ressources systèmes pour l'exécution et invoquer la

	méthode run()
void suspend()	Suspendre le thread jusqu'au moment où il sera relancé par la méthode resume( ). Cette méthode est deprecated
String toString()	Renvoyer une représentation textuelle du thread qui contient son nom, sa priorité et le nom du groupe auquel il appartient
void stop()	Arrêter le thread. Cette méthode est deprecated
static void yield()	Demander au scheduler de laisser la main aux autres threads

37.3. Le cycle de vie d'un thread

Un thread, encapsulé dans une instance de type classe Thread, suit un cycle de vie qui peut prendre différents états.

Le statut du thread est encapsulé dans l'énumération Thread.State

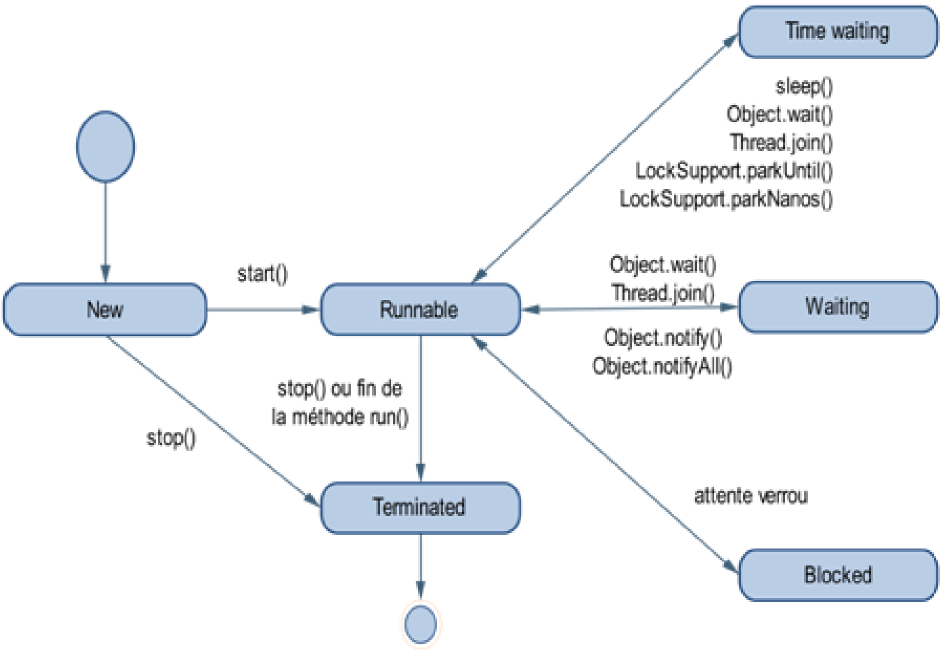
Valeur	Description
NEW	Le thread n'est pas encore démarré. Aucune ressource système ne lui est encore affectée. Seules les méthodes de changement de statut du thread start() et stop() peuvent être invoquées
RUNNABLE	Le thread est en cours d'exécution : sa méthode start() a été invoquée
BLOCKED	Le thread est en attente de l'obtention d'un moniteur qui est déjà détenu par un autre thread
WAITING	Le thread est en attente d'une action d'un autre thread ou que la durée précisée en paramètre de la méthode sleep() soit atteinte. Chaque situation d'attente ne possède qu'une seule condition pour retourner au statut Runnable : • si la méthode sleep() a été invoquée alors le thread ne retournera à l'état Runnable que lorsque le délai précisé en paramètre de la méthode a été atteint • si la méthode suspend() a été invoquée alors le thread ne retournera à l'état Runnable que lorsque la méthode resume sera invoquée • si la méthode wait() d'un objet a été invoquée alors le thread ne retournera à l'état Runnable que lorsque la méthode notify() ou notifyAll() de l'objet sera invoquée • si le thread est en attente à cause d'un accès I/O alors le thread ne retournera à l'état Runnable que lorsque cet accès sera terminé
TIMED_WAITING	Le thread est en attente pendent un certain temps d'une action d'un autre thread. Le thread retournera à l'état Runnable lorsque cette action survient ou lorsque le délai d'attente est atteint
TERMINATED	Le thread a terminé son exécution. La fin d'un thread peut survenir de deux manières : • la fin des traitements est atteinte • une exception est levée durant l'exécution de ses traitements

Le statut du thread correspond à celui géré par la JVM : il ne correspond pas au statut du thread sous-jacent dans le système d'exploitation.

Une fois lancé, plusieurs actions peuvent suspendre l'exécution d'un thread :

- invocation de la méthode sleep(), join() ou suspend()
- attente de la fin d'une opération de type I/O
- ...

Le diagramme ci-dessous illustre les différents états d'un thread et les actions qui permettent d'assurer une transition entre ces états.



L'invocation de certaines méthodes de la classe Thread peut lever une exception de type IllegalThreadStateException si cette invocation n'est pas permise à cause de l'état courant du thread.

37.3.1. La création d'un thread

Depuis Java 1.0, il existe plusieurs façons de créer un thread :

- créer une instance d'une classe anonyme de type Thread et implémenter sa méthode run(). Il suffit alors d'invoquer sa méthode start() pour démarrer le thread
- créer une classe fille qui hérite de la classe Thread. Il suffit alors de créer une instance de la classe fille et d'invoquer sa méthode start() pour démarrer le thread
- créer une classe qui implémente l'interface Runnable. Pour lancer l'exécution, il faut créer un nouveau Thread en lui passant en paramètre une instance de la classe et invoquer sa méthode start()
- à partir de Java 8, il est possible d'utiliser une expression lambda pour définir l'implémentation de l'interface Runnable

Il est possible de créer une instance de type Thread dont l'implémentation de la méthode run() va contenir les traitements à exécuter. La classe Thread implémente l'interface Runnable.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestThread {

  public static void main(String[] args) {
    Thread t = new Thread() {
      public void run() {
        System.out.println("Mon traitement");
      }
    };
    t.start();
  }
}
```

Il est possible d'hériter de la classe Thread et de redéfinir la méthode run().

Exemple :

```
package fr.jmdoudoux.dej.thread;
```

```
public class MonThread extends Thread {

  @Override
  public void run() {
    System.out.println("Mon traitement");
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestThread {

  public static void main(String[] args) {
    MonThread t = new MonThread();
    t.start();
 }
}
```

Enfin, il est possible d'implémenter l'interface Runnable. Celle-ci ne définit qu'une seule méthode run() dont l'implémentation doit contenir les traitements à exécuter.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class MonTraitement implements Runnable {

  @Override
  public void run(){
    System.out.println("Mon traitement");
  }
}
```

Pour exécuter les traitements dans un thread, il faut créer une instance de type Thread en invoquant son constructeur avec en paramètre une instance de la classe et invoquer sa méthode start().

Exemple :

```
public class TestThread {

  public static void main(String[] args){
    Thread thread = new Thread(new MonTraitement());
    thread.start();
  }
}
```

Il est préférable d'utiliser l'implémentation de Runnable car :

- elle permet à la classe d'hériter au besoin d'une classe mère
- elle permet une meilleure séparation des rôles
- elle évite des erreurs car il suffit simplement d'implémenter la méthode run()

Il est possible d'utiliser une instance de type Runnable pour plusieurs threads si l'implémentation est thread-safe.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestThread {

  public static void main(String[] args) {
    Runnable runnable = new MonTraitement();
    for (int i = 0; i < 10; i++) {
      Thread thread = new Thread(runnable);
      thread.start();
    }
  }
}
```

Il ne faut surtout pas invoquer la méthode run() d'un thread. Dans ce cas, les traitements seront exécutés dans le thread courant mais ne seront pas exécutés dans un thread dédié.

37.3.2. L'arrêt d'un thread

Par défaut, l'exécution d'un thread s'arrête pour deux raisons :

- la fin des traitements de la méthode run() est atteinte
- une exception est levée durant les traitements de la méthode run()

Historiquement la classe Thread possède une méthode stop() qui est déclarée deprecated depuis Java 1.1 et est conservée pour des raisons de compatibilité mais elle ne doit pas être utilisée car son comportement peut être aléatoire et inattendu.

La méthode stop() lève une exception de type ThreadDeath se qui interrompt brutalement les traitements du thread. C'est notamment le cas si un moniteur est posé : celui-ci sera libéré mais l'état des données pourrait être inconsistant.

Pour permettre une interruption des traitements d'un thread, il faut écrire du code qui utilise une boucle tant qu'une condition est remplie : le plus simple est d'utiliser un booléen.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class MonThread extends Thread {

  private volatile boolean running = true;

  public void arreter() {
    this.running = false;
  }

  @Override
  public void run() {
    while (running) {
      // traitement du thread
      try {
        Thread.sleep(500);
      } catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

Le Java Memory Model permet à un thread de conserver une copie local de ses champs : pour une exécution correcte, il faut utiliser le mot clé volatile sur le booléen pour garantir que l'accès à la valeur se fera de et vers la mémoire.

Une fois un thread terminé, il passe à l'état terminated. Il ne peut plus être relancé sans lever une exception de type IllegalStateException. Pour le relancer, il faut créer une nouvelle instance.

37.4. Les démons (daemon threads)

Il existe deux catégories de threads :

- thread utilisateur (user thread)
- démon (daemon thread)

Un thread démon n'empêche pas la JVM de s'arrêter même s'il est encore en cours d'exécution. Une application dans laquelle les seuls threads actifs sont des démons est automatiquement fermée.

Généralement, les traitements d'un thread démon s'exécutent indéfiniment et ils ne sont pas interrompus : c'est l'arrêt de la JVM qui provoque leur fin. Lorsque la JVM s'arrête, elle termine tous les threads démons en cours d'exécution du moment qu'ils soient les seuls encore actifs. Par exemple, les threads du ramasse-miettes sont généralement des démons.

Par défaut, un nouveau thread hérite de la propriété daemon du thread qui le lance.

Pour préciser qu'un thread est un démon, il faut invoquer sa méthode setDaemon() en lui passant la valeur true comme paramètre. Cette méthode doit être invoquée avant que le thread ne soit démarré : une fois le thread démarré, son invocation lève une exception de type IllegalThreadStateException.

La méthode isDaemon() renvoie un booléen qui précise si le thread est un démon.

Lorsque la JVM s'arrête, les threads démons sont arrêtés brutalement : leurs blocs finally ne sont pas exécutés. C'est la raison pour laquelle, les threads démons ne devraient pas être utilisés pour réaliser des opérations de type I/O ou des traitements critiques.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestThreaddemon {

  public static void main(String[] args) {
    Thread daemonThread = new Thread(new Runnable() {
      @Override
      public void run() {
        try {
          while (true) {
            System.out.println("Execution demon");
          }
        } finally {
          System.out.println("Fin demon");
        }
      }
    }, "Demon");

    daemonThread.setDaemon(true);
    daemonThread.start();
  }
}
```

Le nombre de messages affichés varie de un à quelques-uns avant l'arrêt de la JVM. Le message du bloc finally n'est jamais affiché.

37.5. Les groupes de threads

Un groupe de threads permet de regrouper des threads selon différents critères et de les manipuler en même temps ce qui évite d'avoir à effectuer la même opération individuellement sur tous les threads. Il permet aussi de définir des caractéristiques communes aux nouveaux threads qui lui sont ajoutés.

La notion de groupe permet aussi de limiter l'accès aux autres threads. Chaque thread ne peut manipuler que les threads de son groupe d'appartenance ou des groupes subordonnés.

La classe java.lang.ThreadGroup encapsule un groupe de threads : elle contient un ensemble de threads pour permettre de réaliser des opérations de gestion ou de contrôle sur tous ceux-ci. Elle peut aussi contenir d'autres ThreadGroups qui forment alors des sous-groupes. Cela permet de créer une hiérarchie dans les groupes.

Chaque groupe, à l'exception du groupe par défaut, possède un groupe parent. Chaque thread appartient à un groupe de threads (thread group) :

- soit explicitement dans un groupe de threads précisé en paramètre de l'une des surcharges du constructeur de la classe thread : Thread(ThreadGroup group, Runnable runnable), Thread(ThreadGroup group, String name), Thread(ThreadGroup group, Runnable runnable, String name)
- soit dans un groupe de threads par défaut si aucun n'est précisé. Par défaut, lors de la création d'un thread, si aucun groupe n'est précisé alors c'est le groupe du thread courant qui est utilisé.

Il existe un groupe de thread par défaut. Au lancement de la JVM, un ThreadGroup généralement nommé main est créé et sera utilisé comme groupe de threads par défaut.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestThreadGroup {

  public static void main(String[] args) {
    Runnable runnable = new MonTraitement();
    Thread t = new Thread(runnable);
    System.out.println("groupe:"+t.getThreadGroup().getName());
    t.start();
  }
}
```

Résultat :

```
groupe:main
```

La seule solution pour ajouter un Thread dans un groupe particulier est d'utiliser une des surcharges du constructeur de la classe Thread qui attend en paramètre un objet de type ThreadGroup :

- public Thread(ThreadGroup group, Runnable target)
- public Thread(ThreadGroup group, String name)
- public Thread(ThreadGroup group, Runnable target, String name)

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestThreadGroup {

  public static void main(String[] args) {
    Runnable runnable = new MonTraitement();
    ThreadGroup monThreadGroup = new ThreadGroup("Mon groupe de threads");
    Thread t = new Thread(monThreadGroup, runnable);
    System.out.println("groupe:" + t.getThreadGroup().getName());
    t.start();
  }
}
```

Résultat :

```
groupe:Mon groupe de threads
```

Attention : une fois créé, un thread ne peut pas être déplacé vers un autre groupe.

La classe ThreadGroup possède plusieurs méthodes :

Méthode	Rôle
int activeCount()	Renvoyer une estimation du nombre de threads actifs dans le groupe et ses sous-groupes
int activeGroupCount()	Renvoyer une estimation du nombre de groupes actifs en incluant les sous-groupes
boolean allowThreadSuspension(boolean b)	Dépréciée depuis le JDK 1.2
void checkAccess()	Vérifier si le thread courant possède les permissions pour modifier son groupe. Dépréciée depuis le JDK 17
void destroy()	Détruire le groupe de threads et ses sous-groupes. Dépréciée depuis le JDK 16
int enumerate(Thread[] list) int enumerate(Thread[] list, boolean recurse)	Copier dans le tableau fourni en paramètre l'ensemble des threads actifs du groupe de threads et de ses sous-groupes
int enumerate(ThreadGroup[] list) int enumerate(ThreadGroup[] list, boolean recurse)	Copier dans le tableau fourni en paramètre l'ensemble des sous-groupes actifs
int getMaxPriority()	Renvoyer la priorité maximale du groupe
String getName()	Renvoyer le nom du groupe
ThreadGroup getParent()	Renvoyer le groupe parent
void interrupt()	Demander l'interruption de tous les threads du groupe
boolean isDaemon()	Renvoyer un booléen qui précise si le groupe est un démon. Lorsqu'un groupe est un démon, il sera détruit lorsque tous ses threads seront terminés. Dépréciée depuis le JDK 16
boolean isDestroyed()	Renvoyer un booléen qui précise si le groupe est détruit. Depuis le JDK 1.1. Dépréciée depuis le JDK 16
void list()	Afficher des informations sur le groupe sur la sortie standard
boolean parentOf(ThreadGroup g)	Renvoyer un booléen qui précise si le groupe courant est le même que celui fourni en argument ou est d'un groupe parent
void resume()	Dépréciée depuis le JDK 1.2
void setDaemon(boolean daemon)	Préciser si le groupe est un démon ou non. Dépréciée depuis le JDK 16
void setMaxPriority(int pri)	Préciser la priorité maximale du groupe
void stop()	Dépréciée depuis le JDK 1.2
void suspend()	Dépréciée depuis le JDK 1.2
void uncaughtException(Thread t, Throwable e)	Cette méthode est invoquée par la JVM si un thread du groupe sans UncaughtExceptionHandler lève une exception durant son exécution

La classe ThreadGroup possède plusieurs propriétés :

- maxPriority définit la valeur maximale de la priorité des threads inclus dans le groupe
- name définit le nom du groupe. Il n'est possible de le modifier une fois le groupe créé
- daemon indique si le groupe est un démon. Si tel est le cas, le groupe sera détruit lorsque tous les threads qu'il contient seront terminés
- parent contient le groupe parent

La méthode setMaxPriority() permet de définir la priorité maximale des threads qui lui seront ajoutés et de ses sous-groupes.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestThreadGroup {

  public static void main(String[] args) {
    Runnable runnable = new MonTraitement();
    ThreadGroup monThreadGroup = new ThreadGroup("Mon groupe de threads");
    monThreadGroup.setMaxPriority(Thread.NORM_PRIORITY);
    Thread t = new Thread(monThreadGroup, runnable);
    t.setPriority(Thread.MAX_PRIORITY);
    monThreadGroup.list();
    System.out.println("thread.priority=" + t.getPriority());
    t.start();
  }
}
```

Résultat :

```
java.lang.ThreadGroup[name=Mongroupe de threads,maxpri=5]
thread.priority=5
```

Une modification d'une de ces propriétés n'a pas d'impact sur les threads contenus dans le groupe.

Par exemple, lors de l'utilisation de la méthode setMaxPriority(), seule la propriété MaxPriority du groupe est modifiée. La priorité des threads déjà inclus dans le groupe n'est pas modifiée. La nouvelle valeur n'aura un impact que sur les prochains threads ajoutés au groupe.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestThreadGroup {

  public static void main(String[] args) {
    Runnable runnable = new MonTraitement();
    ThreadGroup monThreadGroup = new ThreadGroup("Mon groupe de threads");
    Thread t = new Thread(monThreadGroup, runnable);
    t.setPriority(Thread.MAX_PRIORITY);
    monThreadGroup.setMaxPriority(Thread.NORM_PRIORITY);
    monThreadGroup.list();
    System.out.println("thread.proprity=" + t.getPriority());
    t.start();
  }
}
```

Résultat :

```
java.lang.ThreadGroup[name=Mongroupe de threads,maxpri=5]
thread.proprity=10
```

Il est donc possible qu'un thread appartenant à un groupe ait une priorité supérieure à la priorité maximale définit dans son groupe.

La méthode setDaemon() n'a aucune influence sur les threads contenus dans le groupe. Il est tout à fait possible d'ajouter des threads utilisateurs ou démon à un groupe dont la propriété daemon est true.

La méthode isDestroy() permet de savoir si le groupe est détruit.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestThreadGroup {

  public static void main(String[] args) throws InterruptedException {
    Runnable runnable = new MonTraitement();
    ThreadGroup monThreadGroup = new ThreadGroup("Mon groupe de threads");
    monThreadGroup.setDaemon(true);
    System.out.println("groupe.isDaemon()=" + monThreadGroup.isDaemon());
    Thread t = new Thread(monThreadGroup, runnable);
    System.out.println("thread.isDaemon()=" + t.isDaemon());
    monThreadGroup.setMaxPriority(Thread.NORM_PRIORITY);
    t.start();
    t.join();
    System.out.println("groupe.isDestroy()=" + monThreadGroup.isDestroyed());
  }
}
```

Résultat :

```
groupe.isDaemon()=true
thread.isDaemon()=false
Mon traitement
Thread-0
groupe.isDestroy()=true
```

Une fois qu'un groupe est détruit, il n'est plus possible de lui ajouter un thread sinon une exception de type IllegalThreadStateException est levée.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestThreadGroup {

  public static void main(String[] args) throws InterruptedException {
    Runnable runnable = new MonTraitement();
    ThreadGroup monThreadGroup = new ThreadGroup("Mon groupe de threads");
    monThreadGroup.setDaemon(true);
    System.out.println("groupe.isDaemon()=" + monThreadGroup.isDaemon());
    Thread t = new Thread(monThreadGroup, runnable);
    System.out.println("thread.isDaemon()=" + t.isDaemon());
    monThreadGroup.setMaxPriority(Thread.NORM_PRIORITY);
    t.start();
    t.join();
    System.out.println("groupe.isDestroy()=" + monThreadGroup.isDestroyed());
    t = new Thread(monThreadGroup, runnable);
  }
}
```

Résultat :

```
groupe.isDaemon()=true
thread.isDaemon()=false
Montraitement Thread-0
groupe.isDestroy()=true
Exception in thread "main" java.lang.IllegalThreadStateException
            at java.lang.ThreadGroup.addUnstarted(ThreadGroup.java:843)
            at java.lang.Thread.init(Thread.java:348)
            at java.lang.Thread.<init>(Thread.java:451)
            at fr.jmdoudoux.dej.thread.TestThreadGroup.main(TestThreadGroup.java:19)
```

La méthode parentOf() renvoie un booléen qui précise si le groupe est un parent du groupe passé en paramètre.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestThreadGroup {
  public static void main(String[] args) throws InterruptedException {
    ThreadGroup monThreadGroup = new ThreadGroup("Mon groupe de threads");
    ThreadGroup monSousThreadGroup = new ThreadGroup(monThreadGroup,
        "Mon sous-groupe de threads");
    System.out.println(monThreadGroup.parentOf(monSousThreadGroup));
  }
}
```

Résultat :

```
true
```

Les méthodes resume(), stop() et suspend() qui permettaient d'interagir sur l'état des threads du groupe sont deprecated depuis Java 1.1.

Les méthodes activeCount() et enumerate() sont généralement utilisées ensemble pour obtenir la liste des threads actifs dans le groupe et ses sous-groupes.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestThreadGroup {

  public static void main(String[] args) throws InterruptedException {
    int nbThreads;
    Thread[] threads;
    Runnable runnable = new MonTraitement();
    ThreadGroup monThreadGroup = new ThreadGroup("Mon groupe de threads");
    Thread t = new Thread(monThreadGroup, runnable, "thread groupe 1");
    t.start();
    t = new Thread(monThreadGroup, runnable, "thread groupe 2");
    t.start();
    ThreadGroup monSousThreadGroup = new ThreadGroup(monThreadGroup,
          "Mon sous-groupe de threads");
    t = new Thread(monSousThreadGroup, runnable, "thread sous groupe 1");
    t.start();

    nbThreads = monThreadGroup.activeCount();
    System.out.println("groupe.activeCount()=" + nbThreads);
    threads = new Thread[nbThreads];
    monThreadGroup.enumerate(threads);
    for (int i = 0; i > nbThreads; i++) {
      if (threads[i] != null) {
        System.out.println("Thread " + i + " = " + threads[i].getName());
      }
    }
  }
}

class MonTraitement implements Runnable {
  public void run() {
    System.out.println("Mon traitement " + Thread.currentThread()
        .getName());

    try {
      Thread.sleep(2_000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
Mon traitement thread groupe 1
Mon traitement thread groupe 2
groupe.activeCount()=3
Mon traitement thread sous groupe 1
Thread 0 = thread groupe 1
Thread 1 = thread groupe 2
Thread 2 = thread sous groupe 1
```

La classe Thread possède plusieurs méthodes relatives au groupe du thread :

- la méthode getThreadGroup() renvoie une instance de type ThreadGroup qui encapsule le groupe auquel appartient le thread
- la méthode static activeCount() renvoie un entier qui correspond à une estimation du nombre de threads actifs appartenant au même groupe et sous-groupes du thread courant

37.6. L'obtention d'informations sur un thread

Plusieurs méthodes de la classe Thread permettent d'obtenir des informations sur le thread.

Méthode	Rôle
boolean isDaemon()	Renvoyer un booléen qui précise si le thread est un démon
long getId()	Renvoyer un entier long dont la valeur est l'identifiant du thread
ClassLoader getContextClassLoader()	Renvoyer le context classloader du thread
StackTraceElement[] getStackTrace()	Renvoyer un tableau des éléments qui composent la stacktrace d'exécution du thread
int getPriority()	Renvoyer la priorité du thread

Chaque thread possède un nom. Par défaut, la JVM attribut un nom composé de Thread- suivi d'un numéro incrémenté. La méthode getName() permet d'obtenir le nom du thread.

Pour aider au débogage et dans les logs, il est intéressant de donner un nom plus explicite à chaque thread pour l'identifier facilement. Le nom peut être fourni en paramètre du constructeur de l'instance de type Thread ou en utilisant la méthode setName() qui permet de donner un nom explicit au thread.

37.6.1. L'état d'un thread

Plusieurs méthodes permettent d'obtenir des informations sur l'état d'un thread.

Méthode	Rôle
boolean isAlive()	Renvoyer un booléen qui précise si le thread est en cours d'exécution. Elle renvoie true tant que le thread a été démarré et qu'il n'est pas arrêté
Thread.State getState()	Renvoyer le statut du thread
boolean isInterrupted()	Renvoyer un booléen qui précise si le thread est interrompu

37.6.2. L'obtention du thread courant

La méthode statique currentThread() permet d'obtenir le thread dans lequel le code s'exécute.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class MonTraitement implements Runnable {
  public void run() {
    System.out.println("Mon traitement " + Thread.currentThread().getName());
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestThread {

  public static void main(String[] args) {
    Runnable runnable = new MonTraitement();

    for (int i = 0; i < 10; i++) {
      Thread thread = new Thread(runnable);
      thread.setName("monTraitement-" + i);
      thread.start();
    }
  }
}
```

37.7. La manipulation des threads

Il est possible de réaliser plusieurs opérations sur un thread :

- modifier la priorité d'exécution du thread
- mettre en sommeil le thread pour une certaine durée (en millisecondes)
- attendre la fin de l'exécution d'un autre thread
- mettre en pause le thread pour laisser aux autres threads plus de chance de s'exécuter
- interrompre le thread

37.7.1. La mise en sommeil d'un thread pour une certaine durée

La méthode static sleep() de la classe Thread permet de mettre en sommeil le thread courant pour le délai en millisecondes dont la valeur est fournie en paramètre.

Elle est bloquante, elle lève une exception de type InterruptedException au cours de son exécution si un autre thread demande l'interruption de l'exécution du thread.

Exemple :

```
    try {
      Thread.sleep(5000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
```

La méthode sleep() est static : elle ne s'applique que sur le thread courant et il n'est pas possible de désigner le thread concerné.

Une surcharge de la méthode sleep() attend en paramètre la durée en millisecondes et une durée supplémentaire en nanosecondes qui peut varier entre 0 et 999999. La précision de cette attente supplémentaire est dépendante de la machine et du système d'exploitation.

Contrairement à la méthode wait() de la classe Object, la méthode sleep() ne libère pas les verrous qui sont posés par le thread.

37.7.2. L'attente de la fin de l'exécution d'un thread

La méthode join() de la classe Thread permet d'attendre la fin de l'exécution du thread. Elle peut lever une exception de type InterruptedException.

Exemple :

```
package fr.jmdoudoux.dej.thread;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;

public class TestThreadJoin {

  public static void main(String[] args) {
    DateFormat df = new SimpleDateFormat("HH:mm:ss");
    Thread thread1 = new Thread(new MonRunnable(10000));
    Thread thread2 = new Thread(new MonRunnable(5000));

    System.out.println(df.format(new Date()) + " debut");

    thread1.start();
    thread2.start();

    try {
      thread2.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    System.out.println(df.format(new Date()) + " fin thread2");

    try {
      thread1.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    System.out.println(df.format(new Date()) + " fin");
  }

  private static class MonRunnable implements Runnable {

    private long delai;

    public MonRunnable(long delai) {
      this.delai = delai;
    }

    @Override
    public void run() {
      try {
        Thread.sleep(delai);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
}
```

Résultat :

```
19:57:04 debut
19:57:09 fin thread2
19:57:14 fin
```

Une surcharge de la méthode join() attend en paramètre un entier long qui définit la valeur en millisecondes d'un délai d'attente maximum.

Exemple :

```
package fr.jmdoudoux.dej.thread;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;

public class TestThreadJoin {

  public static void main(String[] args) {
    DateFormat df = new SimpleDateFormat("HH:mm:ss");
    Thread thread1 = new Thread(new MonRunnable(10000));
    Thread thread2 = new Thread(new MonRunnable(5000));

    System.out.println(df.format(new Date()) + " debut");

    thread1.start();
    thread2.start();

    try {
      thread2.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    System.out.println(df.format(new Date()) + " fin thread2");

    try {
      thread1.join(1000);

      System.out.println("thread1 en cours d'execution : " + thread1.isAlive());

    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    System.out.println(df.format(new Date()) + " fin");
  }

  private static class MonRunnable implements Runnable {

    private long delai;

    public MonRunnable(long delai) {
      this.delai = delai;
    }

    @Override
    public void run() {
      try {
        Thread.sleep(delai);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
}
```

Résultat :

```
20:01:04 debut
20:01:09 fin thread2
thread1 en cours d'execution : true
20:01:10 fin
```

37.7.3. La modification de la priorité d'un thread

Un thread possède une propriété qui précise sa priorité d'exécution. Pour déterminer ou modifier la priorité d'un thread, la classe Thread contient les méthodes suivantes :

Méthode	Rôle
int getPriority()	retourner la priorité d'exécution du thread
void setPriority(int)	modifier la priorité d'exécution du thread

Généralement, la priorité varie de 1 à 10 mais cela dépend de l'implémentation de la JVM. Plusieurs constantes permettent de connaître les valeurs de la plage de priorités utilisables et la valeur de la priorité par défaut :

- Thread.MIN_PRIORITY : la valeur de la priorité minimale
- Thread.MAX_PRIORITY : la valeur de la priorité maximale
- Thread.NORM_PRIORITY : la valeur de la priorité normale

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestThreadPriority {

  public static void main(String[] args) {
    System.out.println("MIN_PRIORITY : " + Thread.MIN_PRIORITY);
    System.out.println("MAX_PRIORITY : " + Thread.MAX_PRIORITY);
    System.out.println("NORM_PRIORITY : " + Thread.NORM_PRIORITY);
  }
}
```

Résultat :

```
MIN_PRIORITY : 1
MAX_PRIORITY : 10
NORM_PRIORITY : 5
```

La valeur par défaut de la priorité lors de la création d'un nouveau thread est celle du thread courant.

La méthode setPriority() lève une exception de type IllegalStateException si la valeur fournie en paramètre n'est pas incluse dans la plage Thread.MIN_PRIORITY et Thread.MAX_PRIORITY.

Exemple :

```
    Thread thread = new Thread();
    thread.setPriority(Thread.MAX_PRIORITY);
```

Attention : il n'y a aucune garantie sur le résultat du changement de la priorité d'un thread. La gestion des priorités est dépendante de l'implémentation de la JVM et/ou du système d'exploitation sous-jacent. Sur des machines de type Mac ou Unix, le thread qui a la plus grande priorité a systématiquement accès au processeur s'il ne se trouve pas en mode " en attente ". Sous Windows 95, le système ne gère pas correctement les priorités et il choisit lui-même le thread à exécuter : l'attribution d'une priorité supérieure permet simplement d'augmenter ses chances d'exécution.

37.7.4. Laisser aux autres threads plus de chance de s'exécuter

La méthode static yield() de la classe Thread tente de mettre en pause le thread courant pour laisser une chance aux autres threads de s'exécuter.

Attention : il n'y a aucune garantie sur le résultat de l'invocation de la méthode yield() car elle est dépendante de l'implémentation de la JVM.

37.7.5. L'interruption d'un thread

Si le thread n'est pas correctement codé, il n'est pas possible de forcer son arrêt.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestInterruptThread {

  public static void main(String[] args) throws InterruptedException {
    System.out.println("debut");
    Thread thread = new Thread(new Runnable() {
      boolean encore = true;

      @Override
      public void run() {
        System.out.println("debut thread");
        long i = 0;
        while (encore) {
          // très mauvais exemple qui simule une activité du thread
          // Ne pas oublier d'interrompre l'exécution
          i++;
          i--;
        }
        System.out.println("i=" + i);
        System.out.println("fin thread");
      }
    });

    thread.start();

    try {
      Thread.sleep(100);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    thread.interrupt();
    thread.join();
    System.out.println("fin");
  }
}
```

Résultat :

```
debut
debut thread
```

Dans cette situation la seule solution pour arrêter le thread est d'arrêter la JVM elle-même.

Une solution pour remplacer l'invocation de la méthode stop(), qui est deprecated, est d'invoquer la méthode interrupt() pour demander l'interruption de l'exécution d'un thread. Il est nécessaire dans ce cas de s'assurer que les traitements du thread vont tenir compte du statut interrupted du thread pour s'interrompre.

Rien ne définit une sémantique claire à propos de l'utilisation de l'interruption d'un thread mais généralement lorsqu'elle est prise en compte cela se traduit par une fin de l'exécution des traitements effectués le plus proprement possible par le thread lui-même.

Le demande d'interruption n'a pas l'obligation a été prise en compte immédiatement. Généralement, si les traitements sont faits dans une boucle, celle-ci vérifie à chaque itération le statut interrupted. Selon le temps de traitements d'une itération, le délai entre deux vérifications peut être plus ou moins long.

Un thread possède une propriété booléenne qui indique si le statut du thread est interrompu (interrupted). Sa valeur par défaut est false. Lors de l'invocation de la méthode interrupt(), la valeur de la propriété passe à true.

La méthode isInterrupted() permet de renvoyer un booléen qui indique la valeur du statut interrupted du thread.

La méthode interrupted() permet de renvoyer un booléen qui indique la valeur du statut interrupted du thread et de réinitialiser sa valeur. Attention : l'invocation de la méthode interrupted() réinitialise le statut interrupted du thread. Une seconde invocation consécutive de cette méthode renverra toujours false.

L'interruption d'un thread en Java requiert une collaboration entre le thread qui demande l'interruption et le thread dont l'interruption est demandée. Une demande d'interruption ne doit pas nécessairement mettre fin immédiatement au traitement du thread : c'est une demande polie d'un autre thread qui lui demande de bien vouloir mettre fin à son exécution à sa convenance.

L'intérêt de l'interruption de manière coopérative est qu'elle permet de mettre en place un mécanisme souple pour annuler l'exécution de tâches.

Il est rare de vouloir qu'un traitement s'arrête de manière brutale et immédiate : il y a généralement dans ce cas un risque de laisser les données dans un état incohérent. La fin prématurée de l'exécution d'une tâche doit être réalisée par la tâche elle-même pour lui permettre de se terminer proprement par exemple en libérant des ressources.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestInterruptThread {

  public static void main(String[] args) throws InterruptedException {
    System.out.println("debut");
    Thread thread = new Thread(new Runnable() {
      @Override
      public void run() {
        System.out.println("debut thread");
        long i = 0;
        while (!Thread.currentThread().isInterrupted()) {
          // tres mauvais exemple qui simule une activité du thread
          // sur un temps heureusement très court
          i++;
          i--;
        }
        System.out.println("i=" + i);
        System.out.println("fin thread");
      }
    });

    thread.start();

    try {
      Thread.sleep(100);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    thread.interrupt();
    thread.join();
    System.out.println("fin");
  }
}
```

Résultat :

```
debut
debut thread
i=0
fin thread
fin
```

Lorsque le statut interrupted d'un thread est passé à true et qu'une méthode bloquante (Thread.sleep(), Thread.join(), Object.wait(), ...) est en cours d'exécution alors une exception de type InterruptedException est levée par cette méthode. Lorsque l'exception InterruptException est levée, le statut interrupted du thread est retiré : la méthode isInterrupted() renvoie false.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestInterruptThread {

  public static void main(String[] args) throws InterruptedException {
    System.out.println("debut");
    Thread thread = new Thread(new Runnable() {
      @Override
      public void run() {
        System.out.println("debut thread");
        try {
          Thread.sleep(5000);
        } catch (InterruptedException e) {
          System.out.println("Le thread est interrompu");
          System.out.println("thread.isInterrupted()="
              + Thread.currentThread().isInterrupted());
        }
        System.out.println("fin thread");
      }
    });

    thread.start();

    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    thread.interrupt();
    thread.join();
    System.out.println("fin");
  }
}
```

Résultat :

```
debut
debut thread
Le thread est interrompu
thread.isInterrupted()=false
fin thread
fin
```

Ainsi, il est possible qu'un thread ne s'arrête jamais.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestInterruptThread {

  public static void main(String[] args) throws InterruptedException {
    System.out.println("debut");
    Thread thread = new Thread(new Runnable() {
      @Override
      public void run() {
        System.out.println("debut thread");
        while (!Thread.currentThread().isInterrupted()) {
          try {
            Thread.sleep(500);
            System.out.println("traitement du thread");
          } catch (InterruptedException e) {
            System.out.println("InterruptedException capturee");
            System.out.println("thread.isInterrupted()="
                + Thread.currentThread().isInterrupted());
          }
        }
        System.out.println("fin thread");
      }
    });

    thread.start();

    try {
      Thread.sleep(100);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    thread.interrupt();
    thread.join();
    System.out.println("fin");
  }
}
```

Résultat :

```
debut
debut thread
InterruptedException capturee
thread.isInterrupted()=false
traitement du thread
traitement du thread
traitement du thread
traitement du thread
```

Une bonne pratique pour ne pas perdre le statut est de le remettre dans la clause catch de l'exception en invoquant la méthode interrupt() du thread courant.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestInterruptThread {

  public static void main(String[] args) throws InterruptedException {
    System.out.println("debut");
    Thread thread = new Thread(new Runnable() {
      @Override
      public void run() {
        System.out.println("debut thread");
        while (!Thread.currentThread().isInterrupted()) {
          try {
            Thread.sleep(500);
            System.out.println("traitement du thread");
          } catch (InterruptedException e) {
            System.out.println("InterruptedException capturee");
            Thread.currentThread().interrupt();
          }
        }
        System.out.println("fin thread");
      }
    });

    thread.start();

    try {
      Thread.sleep(100);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    thread.interrupt();
    thread.join();
    System.out.println("fin");
  }
}
```

Résultat :

```
debut
debut thread
InterruptedException capturee
fin thread
fin
```

L'invocation de la méthode interrupt() va remettre le statut interrupted du thread à true pour permettre une sortie de la boucle.

37.7.6. L'exception InterruptedException

La fin de l'exécution d'une méthode ordinaire dépend de la quantité de traitements à exécuter et des ressources disponibles pour le faire (CPU et mémoire). La fin de l'exécution d'une méthode bloquante est aussi dépendante d'un événement extérieur tel qu'un timeout, la libération d'un verrou, ... Ceci rend le temps d'attente difficilement prédictible voire même infini si l'événement ne survient jamais. Ce dernier cas de figure entraine un blocage infini du traitement : il est donc nécessaire d'avoir un mécanisme qui permette de sortir de cette situation.

Ce sont des méthodes bloquantes qui peuvent lever une exception de type InterruptedException. De nombreuses méthodes de classes du JDK couramment utilisées peuvent lever une exception de type InterruptedException :

- Object.wait()
- Thread.sleep(), Thread.join()
- Process.waitFor()
- SwingUtilities.invokeAndWait()
- de nombreuses méthodes des classes du package java.util.concurrent telles que Future.get(), BlockingQueue.take(), ExecutorService. awaitTermination(), ...
- ...

Une exception de type InterruptedException est levée par une méthode bloquante pour indiquer que la méthode interrupt() du thread courant a été invoquée par un autre thread, signifiant ainsi que cet autre thread demande au thread courant de s'interrompre.

Les méthodes bloquantes prennent en compte les demandes d'interruption en levant une exception de type InterruptedException. Une méthode ordinaire n'a pas l'obligation de faire de même mais si son temps de traitement peut être long, il est utile et pratique de périodiquement vérifier le statut interrupted du thread et de lever une exception de type InterruptedException.

Remarque : toutes les méthodes bloquantes ne lèvent pas d'exception de type InterruptedException : c'est par exemple le cas des méthodes des classes InputStream et OutputStream qui peuvent attendre la fin d'une opération de type I/O mais ne lève pas cette exception et ne s'arrête pas si le thread courant est interrompu.

L'obtention d'un verrou sur un moniteur en utilisant le mot clé synchronized ne peut pas être interrompue bien qu'étant bloquante.

Si l'exception InterruptedException n'était pas une exception de type checked, probablement personne ne prendrait en compte sa gestion. Comme celle-ci est obligatoire, elle consiste généralement à ne rien faire ou à simplement afficher un message dans un log. Cependant ignorer une exception de type InterruptedException est rarement une bonne idée car cette pratique fait perdre l'information qu'une demande d'interruption du thread a été faite.

Lorsqu'une méthode bloquante lève une exception de type InterruptedException, elle informe le thread courant qu'un autre thread vient de tenter de l'interrompre. Une prise en compte, adaptée au contexte, de cette exception est nécessaire pour assurer une meilleure réactivité de l'application.

Il est fréquent de rencontrer ou d'écrire du code qui intercepte une exception de type InterruptedException avec un bloc de code vide ou simplement journaliser l'exception avec un niveau de gravité plus ou moins important. Capturer une exception et l'ignorer n'est pas une bonne pratique. Se contenter de l'ajouter dans un journal revient aussi à l'ignorer, si ce n'est qu'il y a en une petite trace.

L'arrêt d'un thread en Java doit être coopératif entre le thread qui en fait la demande généralement en positionnant un booléen et les traitements du thread qui doivent périodiquement vérifier la valeur du booléen avant de poursuivre les traitements.

Il est possible d'utiliser la propriété booléenne interrupted du thread. Pour basculer la valeur de la propriété, il faut invoquer la méthode interrupt() du thread.

La méthode interrupt() n'interrompt pas l'exécution du thread : elle positionne simplement le statut interrupted du thread à true. Les traitements du thread ont la charge de tenir compte de ce statut et de faire les actions appropriées sachant qu'il n'existe pas de recommandations sur celles-ci.

Si le thread en cours exécute un traitement bloquant, alors une exception de type InterruptedException est levée.

Comme précisé dans la javadoc des méthodes concernées, lorsqu'une exception de type InterruptedException est levée, le statut interrupted du thread est réinitialisé.

37.8. Les messages de synchronisation entre threads

La classe Object contient les méthodes wait(), notify() et notifyAll() pour permettre de synchroniser des threads grâce à l'envoi de messages. Ces méthodes permettent la mise en oeuvre d'un mécanisme de communication par échanges de messages visant à synchroniser l'exécution de threads.

La méthode wait() met le thread courant en attente jusqu'à ce que l'objet reçoive une notification par les méthodes notify() ou notifyAll() : cette attente peut donc être potentiellement infinie.

La méthode wait() possède deux surcharges :

- wait(long *timeout*) : attend au plus la durée en millisecondes fournie en paramètre
- wait(long *timeout*, int *nanos*) : attend au plus la durée en millisecondes cumulée avec celle en nanosecondes fournies en paramètres

La méthode notifyAll() avertit tous les threads dont les méthodes wait() de la même instance sont invoquées.

La méthode notify() avertit un des threads dont la méthode wait() de la même instance est invoquée.

Il est important que les méthodes wait() et notifyAll() ne soient invoquées que par le thread qui possède le verrou sur le moniteur de l'instance.

Un cas classique d'utilisation de la synchronisation de threads est la mise en oeuvre du modèle de conception producer/consumer.

Dans l'exemple ci-dessous, un thread (producer) est utilisé pour produire des données qui sont consommées par un autre thread (consumer). Un objet partagé par les deux threads permet de stocker une valeur et de gérer sont accès par les threads en les synchronisant.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class MaQueue {
  private String  valeur;
  private boolean disponible = false;

  public synchronized String get() throws InterruptedException {
    while (disponible == false) {
      wait();
    }
    disponible = false;
    notifyAll();
    return valeur;
  }

  public synchronized void put(String valeur) throws InterruptedException {
    while (disponible == true) {
      wait();
    }
    this.valeur = valeur;
    disponible = true;
    notifyAll();
  }
}
```

Les méthodes wait(), notify() et notifyAll() doivent être invoquées dans un bloc de code synchronized utilisant l'objet lui-même comme moniteur pour éviter de lever une exception de type IllegalMonitorStateException. Il peut y avoir une race condition lors de l'invocation des méthodes wait() and notify() si elles ne sont pas invoquées dans un bloc de code synchronized. Le moniteur de ce bloc synchronized doit obligatoirement être l'instance sur laquelle les méthodes wait(), notify() et notifyAll() vont être invoquées.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class MonProducer extends Thread {
  private MaQueue maQueue;

  public MonProducer(MaQueue maQueue) {
    this.maQueue = maQueue;
  }

  public void run() {
    int i = 0;
    while (!Thread.currentThread().isInterrupted()) {
      try {
        i++;
        maQueue.put("valeur-" + i);
        System.out.println("Producer put : " + i);
        sleep((int) (Math.random() * 1000));
      } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        break;
      }
    }
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class MonConsumer extends Thread {
  private MaQueue maQueue;

  public MonConsumer(MaQueue maQueue) {
    this.maQueue = maQueue;
  }

  public void run() {
    String value = null;
    while (!Thread.currentThread().isInterrupted()) {
      try {
        value = maQueue.get();
        System.out.println("Consumer get : " + value);
      } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
      }
    }
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.thread;
```

```
public class TestProducerConsumer {

  public static void main(String[] args) {
    MaQueue maQueue = new MaQueue();
    MonProducer producer = new MonProducer(maQueue);
    MonConsumer consumer = new MonConsumer(maQueue);

    consumer.start();
    producer.start();

    try {
      Thread.sleep(2000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    producer.interrupt();
    consumer.interrupt();
  }
}
```

C'est l'objet partagé qui assure la synchronisation des deux threads : comme il ne peut contenir qu'une seule donnée, il est nécessaire de bloquer le producer et de débloquer le consumer lorsqu'une donnée est déjà présente et inversement lorsque la donnée est consommée.

Cette synchronisation est nécessaire pour éviter au consumer de rater une valeur si le producer en envoie une autre alors que le consumer traite encore la valeur précédente ou que le consumer traite plusieurs fois le même message tant que le producer n'a pas ajouté une nouvelle valeur.

La synchronisation permet de garantir qu'une donnée ne sera traitée qu'une seule fois et qu'une nouvelle valeur ne pourra être ajoutée que s'il n'y a pas de valeur à traiter.

Elle utilise dans l'exemple deux mécanismes :

- un moniteur : celui utilisé est celui de l'instance de type MaQueue partagée
- les méthodes wait() et notifyAll() de l'instance de type MaQueue partagée pour permettre de bloquer un thread en attendant un message d'un autre thread

Au premier abord, il peut sembler bizarre que le thread qui attend ait posé le verrou sur le monitor, laissant présager que l'autre thread attendra indéfiniment puisqu'il attend la pose du verrou pour notifier l'autre thread.

Cela fonctionne pourtant bien car lorsque la méthode wait() est exécutée, elle libère automatiquement le verrou posé sur le monitor. Le verrou est de nouveau posé sur le monitor à la fin de l'exécution de la méthode wait(). Ceci permet au thread qui n'est pas en attente de poser le verrou sur le monitor libéré par l'invocation de la méthode wait() dans l'autre thread. Celui-ci pourra alors poser le verrou sur le monitor et envoyer une notification.

37.9. Les restrictions sur les threads

L'utilisation de threads présente plusieurs limitations.

Il n'est pas possible de relancer un thread qui s'est terminé : il est nécessaire de créer une nouvelle instance de type Thread et de la lancer. Il est cependant possible de lui passer en paramètre la même instance de Runnable.

La méthode clone() de la classe Thread renvoie toujours une exception de type CloneNotSupportedException.

La classe Thread n'est pas Serializable essentiellement car elle a besoin de ressources systèmes obtenues par la JVM. Ces ressources seront forcément différentes dans une autre JVM. C'est une très mauvaise idée de définir une classe qui héritent de la classe Thread et qui implémente Serializable : cette classe fille a la responsabilité de sérialiser les champs de sa classe mère Thread ce qui est complexe car la classe Thread possède des champs private.

Le nombre de threads qu'il est possible de lancer dans une JVM n'est pas illimité et dépend de plusieurs facteurs :

- les ressources systèmes : le microprocesseur, la mémoire disponible sur le système
- du système d'exploitation
- de l'implémentation de la JVM utilisée
- de la configuration implicite ou explicite de certains paramètres de la JVM notamment la taille par défaut de la pile et la taille du heap

L'option -Xss d'une JVM HotSpot permet de préciser la taille par défaut de la pile des threads.

Attention : atteindre le nombre maximal de threads peut rendre le système d'exploitation instable voire même le mettre en péril.

Plutôt que de lancer de très nombreux threads, il est possible pour de nombreux scénarios de lancer les threads dans un pool de threads par exemple en utilisant un ExecutorService. Ceci permet d'avoir un contrôle sur le nombre de threads lancés et donc sur les ressources utilisées.

37.10. Les threads et les classloaders

Les classes en Java sont chargées par un classloader. Par défaut, la hiérarchie de classloaders recherche par délégation une classe dans les jars système (bootstrap classloader) et dans le classpath (system classloader). Il est aussi possible de créer ses propres classloader pour rechercher une classe dans un autre endroit (solution généralement mises en oeuvre par les conteneurs des serveurs d'applications ou par le plug-in d'exécution d'applets). Une même classe chargée par deux classloaders différents sera chargées deux fois dans la JVM.

Des threads peuvent accéder à des classes partagées avec d'autres threads, indépendamment du classloader ou des classloaders utilisés pour les charger.

A chaque thread est assigné un classloader particulier nommé context classloader. Ce classloader peut être obtenu en invoquant la méthode getContextClassloader() de la classe Thread et modifié en utilisant la méthode setContextClassloader().

Le context classloader permet de charger des classes et des ressources dans des cas particuliers. Par exemple, le context classloader est utilisé par des serveurs d'applications ou pour la sérialisation d'objets en utilisant le protocole IIOP. Dans ce dernier cas, les classes de l'ORB sont chargées par le bootstrap classloader qui ne permettra probablement pas de charger la classe applicative lors de la désérialisation. Dans ces cas, la solution est d'utiliser un context classloader qui sera utilisé pour charger les classes.

Le context classloader peut être modifié à tout moment.

Le context classloader par défaut d'un thread est le classloader de la classe de l'instance qui crée le thread : c'est généralement le classloader applicatif sauf si un classloader dédié est utilisé.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestThreadClassloader {

  public static void main(String[] args) {

    afficherInfo();

    Thread thread = new Thread(new Runnable() {

      @Override
      public void run() {
        afficherInfo();
      }
    });
    thread.start();
  }

  private static void afficherInfo() {
    System.out.println(Thread.currentThread().getName());
    System.out.println(Thread.currentThread().getClass().getClassLoader());
    System.out.println(Thread.currentThread().getContextClassLoader());
  }
}
```

Résultat :

```
main
null
sun.misc.Launcher$AppClassLoader@1cde100
Thread-0
null
sun.misc.Launcher$AppClassLoader@1cde100
```

Sauf si un classloader personnalisé est utilisé, il n'est généralement pas nécessaire de modifier le context classloader.

37.11. Les threads et la gestion des exceptions

Une exception est propagée dans la pile d'appels du thread courant. La méthode run() ne peut pas propager d'exception de type checked : les traitements de la méthode run() ne peuvent lever et propager que des exceptions de type unchecked (runtime et error).

Toutes les exceptions qui ne sont pas gérées explicitement dans le code des traitements du thread sont gérées par un mécanisme dédié nommé gestionnaire d'exceptions non capturées (uncaught exceptions handler) avant que le thread se termine.

Chaque thread possède un gestionnaire d'exceptions non capturées par défaut qui invoque la méthode uncaughtException() du groupe de threads auquel appartient le thread.

La classe ThreadGroup implémente l'interface Thread.UncauchtExceptionHandler. La JVM va invoquer la méthode uncauchtException() du ThreadGroup si le thread n'a pas de gestionnaire d'exceptions non capturées dédié.

L'implémentation par défaut de la méthode uncaughtException() affiche pour tout Throwable sauf ThreadDeath sur la sortie standard d'erreurs :

- Exception in thread suivi du nom du thread entre double quote
- la stacktrace du thread

Il est possible de définir explicitement son propre gestionnaire d'exceptions non capturées pour par exemple journaliser l'exception ou envoyer un mail.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class AlerteSurExceptionThreadGroup extends ThreadGroup {

  public AlerteSurExceptionThreadGroup() {
    super("Alerte sur Exception ThreadGroup");
  }

  public AlerteSurExceptionThreadGroup(String name) {
    super(name);
  }

  public AlerteSurExceptionThreadGroup(ThreadGroup parent, String name) {
    super(parent, name);
  }

  @Override
  public void uncaughtException(Thread t, Throwable e) {
    // actions pour envoyer l'alerte
    System.err.println("Exception non capturee dans le thread " + t.getName());
    e.printStackTrace();
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.thread;

import java.util.ArrayList;
import java.util.List;

public class TestAlerteSurExceptionThreadGroup {

  public static void main(String[] args) {
    AlerteSurExceptionThreadGroup tg = new AlerteSurExceptionThreadGroup();

    Thread t = new Thread(tg, new Runnable() {

      @Override
      public void run() {
        List<byte[]> liste = new ArrayList<byte[]>();

        // va lever une OutOfMemoryError
        while (true) {
          liste.add(new byte[1024]);
        }
      }
    });
    t.start();
  }
}
```

Résultat :

```
Exception non capturee dans le thread Thread-0
java.lang.OutOfMemoryError: Java heap space
      at fr.jmdoudoux.dej.thread.TestAlerteSurExceptionThreadGroup$1.run(
TestAlerteSurExceptionThreadGroup.java:19)
      at java.lang.Thread.run(Thread.java:662)
```

A partir de Java 5, il est possible de définir ou de modifier le gestionnaire d'exceptions non capturées d'un thread particulier en invoquant sa méthode setUncaughtExceptionHandler() qui attend en paramètre l'instance du gestionnaire à utiliser.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

import java.lang.Thread.UncaughtExceptionHandler;
import java.util.ArrayList;
import java.util.List;

public class TestAlerteSurExceptionThread {

  public static void main(String[] args) {
    Thread t = new Thread(new Runnable() {

      @Override
      public void run() {
        List<byte[]> liste = new ArrayList<byte[]>();

        // va lever une OutOfMemoryError
        while (true) {
          liste.add(new byte[1024]);
        }
      }
    });

    t.setUncaughtExceptionHandler(new UncaughtExceptionHandler() {
```

```
   @Override
   public void uncaughtException(Thread t, Throwable e) {
    // actions pour envoyer l'alerte
    System.err.println("Exeception non capturee dans le thread " + t.getName());
    e.printStackTrace();
   }
  });

  t.start();
 }
}
```

La possibilité d'ajouter un gestionnaire dédié à un thread particulier par Java 5.0 est compatible avec la gestion des exceptions non capturées de la version précédente.

La méthode getUncaughtExceptionHandler() permet d'obtenir le gestionnaire d'exceptions non capturées qui sera invoqué au besoin par la JVM : soit celui défini explicitement soit celui par défaut.

La méthode static setDefaultExceptionHandler() permet de définir le handler qui sera utilisé par tous les nouveaux threads créés. Son invocation n'a aucun effet sur les threads déjà créés.

La méthode getDefaultExceptionHandler() permet d'obtenir le handler par défaut.

37.11.1. L'exception ThreadDeath

Une instance d'une exception de type ThreadDeath est levée par la JVM lors de l'invocation de la méthode stop() d'un thread.

La classe ThreadDead hérite de la classe Error même si c'est en fait une exception standard : ceci évite d'avoir à déclarer sa propagation dans la méthode run() et évite que celle-ci ne soit interceptée par un clause catch sur le type Exception.

La méthode uncaughtException() de la classe Thread gère par défaut de manière particulière une exception de type ThreadDeath en l'ignorant. Pour toutes les autres exceptions, elle affiche sur la sortie d'erreurs un message et la stacktrace.

Par défaut, la JVM va invoquer la méthode dispatchUncaughtException() de la classe Thead : celle-ci invoque la méthode getUncaughtExceptionHandler() qui renvoie l'objet de type UncaughtExceptionHandler explicitement associé au thread ou à défaut renvoie le ThreadGroup du thread puisqu'il implémente l'interface Thread.UncaughtExceptionHandler.

Par défaut, l'implémentation de la méthode uncaughtException(Thread, Throwable) de la classe ThreadGroup effectue plusieurs traitements :

- la méthode uncaughtException() du groupe de threads parent est invoquée avec les mêmes paramètres si une telle instance existe
- sinon elle invoque la méthode uncaughtException() du UncaughtExceptionHandler par défaut si celui-ci est défini
- sinon si l'exception n'est pas de type ThreadDeath alors elle affiche sur la sortie d'erreur le message «Exception in thread » suivi du nom du thread et la stacktrace

Même si cela n'est pas recommandé, il est possible de lever soi-même une exception de type ThreadDeath pour mettre fin à l'exécution d'un thread de manière silencieuse. Attention cependant, car les contraintes qui ont forcé le JDK lui-même à ne plus appliquer cette technique s'appliquent aussi pour une utilisation directe par le développeur. La seule vraie différence est que cette technique ne peut être utilisée dans tous les cas. Si le développeur est en mesure de garantir qu'au moment où l'exception sera levée, il ne peut pas y avoir de données inconsistantes, alors il est possible de l'utiliser. Contrairement à l'invocation de la méthode stop(), le compilateur ne dira rien si une exception de ThreadDeath est levée. Cependant, elle ne doit être une solution à n'utiliser que pour des besoins très spécifiques impliquant qu'il n'y pas d'autres solutions plus propres à mettre en oeuvre.

Les traitements d'un thread peuvent capturer cette exception uniquement si des traitements particuliers doivent être exécutés avant de terminer brutalement le thread : c'est par exemple le cas si des actions de nettoyage doivent être faites pour laisser le système dans un état propre (libération de ressources, ...)

Si une exception ThreadDeath est capturée alors il est important qu'elle soit relevée pour permettre au thread de s'arrêter.

37.12. Les piles

Lors de la création d'un nouveau thread, la JVM alloue un espace mémoire qui lui est dédié nommé pile (stack). La JVM stocke des frames dans la pile.

La pile d'un thread est un espace de mémoire réservée au thread pour stocker et gérer les informations relatives à l'invocation des différentes méthodes effectuée par les traitements du thread. Chaque invocation d'une méthode ajoute une entrée dans le pile qui contient entre autres une référence sur la méthode et ses paramètres.

C'est la raison pour laquelle la taille de la pile doit être suffisamment importante pour stocker les différentes invocations d'une méthode notamment si elle est invoquée de manière récursive et que ce nombre d'appels est important.

La pile permet de garder une trace des invocations successives de méthodes.

Chaque thread possède sa propre pile qui stocke :

- les variables locales sous la forme de types primitifs. Si c'est un objet, c'est la référence qui est stockée, l'objet lui-même est stocké dans le heap
- chaque invocation d'une méthode ajoute une entrée nommée frame en haut de la pile

Lorsque l'exécution de la méthode est terminée, la frame est retirée de la pile. Les variables qu'elle contient sont supprimées : si ces variables sont des objets, leurs références sont supprimées mais ils existent toujours dans le heap. Si aucune autre référence sur ces objets existe, le ramasse-miettes les détruira à sa prochaine exécution.

La première frame de la pile est la méthode run() du thread. Chaque frame contient les variables locales de la méthode en cours d'exécution :

- les paramètres de la méthode
- les variables locales
- l'instruction en cours d'exécution
- des informations utiles pour le thread

Par défaut, jusqu'à la version Java 6 u23, pour une variable locale qui est un objet :

- la frame de la pile contient une référence vers l'objet
- l'objet est stocké dans le heap

La JVM ne stocke que des primitives dans la pile pour lui permettre de conserver une taille la plus petite possible et ainsi permettre d'imbriquer plus d'invocations de méthodes. Tous les objets sont créés dans le heap et seulement des références sur ces objets sont stockées dans la pile.

Les informations stockées dans le heap et la pile ont un cycle de vie différent :

- les informations contenues dans la pile ont une durée de vie courte : leur portée est liée à la durée d'exécution de la méthode qui les a créées. Une fois que les traitements de la méthode sont terminés et qu'elle a renvoyé une valeur (ou void) les informations concernées sont retirées de la pile
- les objets sont créés dans le heap : leur cycle de vie est géré par la machine virtuelle jusqu'à leur destruction par le ramasse-miettes

La durée de vie des valeurs stockées dans la pile est liée à la méthode dans laquelle elles ont été créées : une fois l'exécution de la méthode terminée, elles sont supprimées.

37.12.1. Les threads et la mémoire

Bien que Java définisse la taille de chaque type de variable, la taille de la pile est dépendante de la plateforme et de l'implémentation de la JVM :

- l'espace requis pour stocker une variable dans la pile peut varier selon la plateforme, essentiellement pour optimiser les opérations réalisées par certains CPU
- une frame contient des informations utiles au thread qui sont spécifiques à l'implémentation de la JVM. La quantité de données requises est donc dépendante de l'implémentation voire même de la version de cette implémentation

La taille par défaut de la pile d'un thread est donc dépendante de l'implémentation de la JVM, du système d'exploitation et de l'architecture CPU.

Depuis la version 1.4 de Java, une surcharge du constructeur de la classe Thread permet de préciser la taille de la pile à utiliser. Par exemple, ceci peut être particulièrement utile pour un thread qui fait beaucoup d'invocations récursives d'une méthode.

Remarque : il n'y a aucune garantie que la même valeur fournie à plusieurs implémentations d'une JVM ait le même effet vu que la pile est dépendante du système d'exploitation utilisé.

Attention : l'implémentation de la JVM peut modifier cette valeur à sa guise notamment si celle-ci est trop petite, trop grande ou doit être un multiple d'une certaine taille pour respecter une contrainte liée au système d'exploitation.

Les spécifications de la JVM permettent à l'implémentation d'avoir une taille de pile fixe ou une taille dynamique qui peut varier selon les besoins.

Généralement, la JVM permet de configurer la taille des piles. Cette option n'est pas standard. Par exemple, avec la JVM Hotspot, il faut utiliser l'option -Xss

Résultat :

```
java -Xss1024k MonApplication
```

Une nouvelle frame est créée à chaque invocation d'une méthode. La frame est détruite lorsque l'exécution de la méthode se termine de manière normale ou à cause de la levée d'une exception.

Chaque frame contient un tableau des variables locales : la taille de ce tableau est déterminée par le compilateur et stockée dans le fichier .class. Les premiers éléments du tableau sont les paramètres utilisés lors de l'invocation de la méthode.

Chaque frame contient une pile de type LIFO des opérandes (operand stack). La taille de cette pile est déterminée par le compilateur. La JVM utilise cette pile pour charger ou utiliser des opérandes mais aussi pour préparer des variables à être passées en paramètre d'une méthode ou pour recevoir le résultat d'une méthode.

Plusieurs limitations de la mémoire liées à une pile peuvent lever une exception :

- une exception de type StackOverFlowError est levée si la pile est trop petite
- une exception de type OutOfMemoryError si le système ne peux pas allouer la mémoire requise pour la pile d'un nouveau thread ou si la taille de la pile ne peut pas être dynamiquement agrandie

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestThreadOOME {

  public static void main(String[] args) {
    for (int i = 0; i < 1000; i++) {
      MonThread t = new MonThread();
      t.start();
    }
  }
}
```

Résultat :

```
Exception in thread "main" java.lang.OutOfMemoryError: unable to create new
native thread
            at java.lang.Thread.start0(Native Method)
            at java.lang.Thread.start(Thread.java:640)
            at fr.jmdoudoux.dej.thread.TestThreadOOME.main(TestThreadOOME.java:9)
```

Remarque : il est possible qu'une exception de type OutOfMemoryError soit levée lors de la création d'un thread s'il n'y a plus d'espace dans le heap pour stocker le nouvel objet de type Thread.

L'estimation de la taille optimale d'une pile est compliquée car elle doit tenir compte de plusieurs facteurs liés au code exécuté par le thread :

- le nombre de méthodes invoquées successivement
- le nombre de paramètres de chacune de ces invocations
- l'implémentation de la JVM : la taille requise pour chaque paramètre peut varier entre différentes implémentations

L'espace mémoire requis par la pile d'un thread n'est pas pris dans le heap mais dans la mémoire générale de la machine virtuelle. Ceci a un effet limitant sur le nombre de threads que la machine virtuelle pourra lancer.

Sur un système d'exploitation la taille maximale d'un processus est limitée (généralement en fonction de l'architecture du processeur et de son implémentation). Cette taille maximale doit permettre de contenir les différentes zones de mémoire de la JVM :

- heap
- stacks
- code

Par exemple, sur un Windows 32 bits, la taille maximale de mémoire allouable à un processus est de l'ordre de 2Go. Ces 2Go doivent contenir, le heap, la mémoire requise par la JVM (permgen, ...), les bibliothèques natives et les différentes piles des threads de la JVM. Ceci combiné à la taille d'une pile implique une limitation sur le nombre maximum de threads qui peuvent être exécutés dans une JVM.

Exemple :

```
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.atomic.AtomicInteger;

public class TestThreadsLimite {
  private final static AtomicInteger compteur = new AtomicInteger(0);

  public static void main(String[] argv) {
    try {
      for (;;) {
        Thread thread = new Thread(new Runnable() {
          public void run() {
            compteur.incrementAndGet();
            for (;;) {
              try {
                Thread.sleep(1000);
              } catch (Exception e) {
              }
            }
          }
        });
        thread.setDaemon(true);
        thread.start();
      }
    } catch (Throwable e) {
      System.out.println("Thread numero " + compteur.get());
      e.printStackTrace();
    }
  }
}
```

Résultat sur un Windows 32 bits :

Taille du heap (-Xmx)	Taille par défaut de la pile (-Xss)	Nombre de threads créés avant OOME
64m	10k	23272
64m	1024k	1817
512m	10k	18747
512m	1024k	1369
1024m	10k	11249
1024m	1024k	866

37.12.2. L'obtention d'informations sur la pile

La classe Thread possède plusieurs méthodes qui permettent d'obtenir des informations sur la pile d'un thread :

Méthode	Rôle
int countStackFrames()	deprecated
static void dumpStack()	Afficher la pile du thread courant sur la sortie d'erreur
StackTraceElement[] getStackTrace( )	Renvoyer un tableau de type StackTraceElement qui contient toutes les méthodes de la pile du thread
static Map getAllStackTraces( )	Obtenir une map contenant pour chaque thread (en clé) un tableau de type StackTraceElement (en valeur) qui contient toutes les méthodes de sa pile

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestGetStackTrace {

  public static void main(String[] args) {
    final StringBuilder str = new StringBuilder();
    final StackTraceElement[] stack = Thread.currentThread().getStackTrace();
    for (final StackTraceElement ste : stack) {
      str.append(ste);
      str.append("\n");
    }
    System.out.println(str.toString());
  }
}
```

Résultat :

```
java.lang.Thread.getStackTrace(Thread.java:1479)
fr.jmdoudoux.dej.thread.TestGetStackTrace.main(TestGetStackTrace.java:7)
```

37.12.3. L'escape analysis

L'escape analysis est une analyse qui permet de déterminer si pour une stack frame un objet reste confiné dans un seul thread. Si tel est le cas, le compilateur peut optimiser les performances en allouant l'objet dans la pile voire même, si l'objet est petit, en stockant directement ses champs dans les registres.

Cette fonctionnalité permet donc à la JVM de détecter si un objet créé localement dans une méthode ne pourra pas être référencé en dehors de cette méthode. Ceci doit garantir que l'objet ne possèdera plus de référence à la fin de l'exécution de la méthode. Dans ce cas, l'objet est alloué dans la pile ainsi que toutes les variables de ses champs.

L'escape analysis est une technique utilisée par le compilateur C2 de la JVM Hotspot : elle permet l'analyse de l'utilisation d'un objet afin de potentiellement mettre en oeuvre certaines optimisations :

- allocation d'un objet dans la pile plutôt que dans le heap si sa portée reste dans la méthode ou le thread
- pour ces objets, ne pas générer les traitements de pose de verrous (locks) puisqu'ils ne sont utilisés que dans le contexte local du thread

L'allocation d'objets dans la pile améliore les performances car cela évite à ces objets d'être gérés par le ramasse-miettes. Les objets sont créés dans la stack frame : dès que la méthode est terminée, la stack frame et tout ce qu'elle contient est supprimé.

La pile est par nature non fragmentée : les différentes stack frames sont empilées et dépilées dans l'ordre inverse. A contrario, la fragmentation est une contrainte importante dans la gestion du heap. Les objets deviennent récupérables par le ramasse-miettes dans un ordre aléatoire par rapport à leur création dans le heap. Lorsque le ramasse-miettes récupère les objets inutiles, cela laisse des espacse vides non contigus. Deux grandes stratégies sont alors utilisables :

- compacter le heap : allonge le temps du stop the world requis par le ramasse-miettes mais rend la création d'une nouvelle instance rapide car il suffit de l'ajouter après la dernière instance créée
- ne pas compacter le heap : allonge le temps de création d'une nouvelle instance qui doit trouver un espace vide suffisant mais diminue le temps de pause du ramasse-miettes

L'allocation d'un objet dans la pile n'implique pas ces mécanismes. Si un objet est alloué dans la pile, alors cette donnée est hors de la portée du ramasse-miettes qui ne travaille que sur le heap et la permgen. L'objet est immédiatement supprimé dès que la stack frame est retirée de la pile.

A partir de la version 6 update 14, la JVM Hotspot (version 14) propose un support de l'escape analysis. Son activation se fait en utilisant l'option -XX:+DoEscapeAnalysis de la JVM Hotspot.

A partir de la version Java 6u23, il est activé par défaut lorsque le mode C2 du compilateur JIT est utilisé.

Cette fonctionnalité est disponible dans d'autres langages notamment C# et dans d'autres JVM notamment la J9 d'IBM.

La version de Java utilisée dans la suite de cette section est la 6u43 sur un Windows XP 32 bits.

Résultat :

```
C:\Java\TestThreads\src>java -version
java version "1.6.0_43"
Java(TM) SE Runtime Environment (build 1.6.0_43-b01)
Java HotSpot(TM) Client VM (build 20.14-b01, mixed mode)
```

La classe de test effectue une boucle pour créer une instance d'un objet dont la portée ne sort pas de la méthode.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestEscapeAnalysis {

  private static class MonBean {
    private long        valeur;
    private static long compteur;

    public MonBean() {
      valeur = compteur++;
    }
  }

  public static void main(String[] args) {
    System.out.println("debut");

    long startTime = System.currentTimeMillis();

    for (long i = 0; i < 1000000000L; ++i) {
      MonBean monBean = new MonBean();
    }

    long duree = System.currentTimeMillis() - startTime;
    System.out.println("fin compteur=" + MonBean.compteur);
    System.out.println("Temps d'execution : " + duree);
  }
}
```

Les exécutions vont utiliser plusieurs configurations différentes mais à chaque fois la JVM affiche des informations sur l'activité du ramasse-miettes.

Résultat :

```
C:\Java\TestThreads\src>java -Xmx256m -server -verbose:gc -XX:+DoEscapeAnalysis
-XX:+PrintGCDetails -cp . fr.jmdoudoux.dej.thread.TestEscapeAnalysis
debut
fin compteur=1000000000
Temps d'execution : 1063
Heap
 PSYoungGen      total 15040K, used 1715K [0x1eaf0000, 0x1fbb0000, 0x24040000)
  eden space 12928K, 13% used [0x1eaf0000,0x1ec9cd38,0x1f790000)
  from space 2112K, 0% used [0x1f9a0000,0x1f9a0000,0x1fbb0000)
  to   space 2112K, 0% used [0x1f790000,0x1f790000,0x1f9a0000)
 PSOldGen        total 34368K, used 0K [0x14040000, 0x161d0000, 0x1eaf0000)
  object space 34368K, 0% used [0x14040000,0x14040000,0x161d0000)
 PSPermGen       total 16384K, used 1783K [0x10040000, 0x11040000, 0x14040000)
  object space 16384K, 10% used [0x10040000,0x101fdf70,0x11040000)
```

Comme la version de Java utilisée est la 6u43, l'escape analysis est activé par défaut.

Résultat :

```
C:\Java\TestThreads\src>java -Xmx256m -server -verbose:gc -XX:+PrintGCDetails
-cp . fr.jmdoudoux.dej.thread.TestEscapeAnalysis
debut
fin compteur=1000000000
Temps d'execution : 1078
Heap
 PSYoungGen      total 15040K, used 1971K [0x1eaf0000, 0x1fbb0000, 0x24040000)
  eden space 12928K, 15% used [0x1eaf0000,0x1ecdcd38,0x1f790000)
  from space 2112K, 0% used [0x1f9a0000,0x1f9a0000,0x1fbb0000)
  to   space 2112K, 0% used [0x1f790000,0x1f790000,0x1f9a0000)
 PSOldGen        total 34368K, used 0K [0x14040000, 0x161d0000, 0x1eaf0000)
  object space 34368K, 0% used [0x14040000,0x14040000,0x161d0000)
 PSPermGen       total 16384K, used 1783K [0x10040000, 0x11040000, 0x14040000)
  object space 16384K, 10% used [0x10040000,0x101fdf70,0x11040000)
```

Lors de la désactivation de l'escape analysis, le temps d'exécution est multiplié par quatre.

Résultat :

```
C:\Java\TestThreads\src>java -Xmx256m -server -verbose:gc -XX:-DoEscapeAnalysis
-XX:+PrintGCDetails -cp . fr.jmdoudoux.dej.thread.TestEscapeAnalysis
debut
[GC [PSYoungGen: 12928K->192K(15040K)] 12928K->192K(49408K), 0.0014329 secs]
[GC [PSYoungGen: 13120K->176K(27968K)] 13120K->176K(62336K), 0.0004556 secs]
[GC [PSYoungGen: 26032K->200K(27968K)] 26032K->200K(62336K), 0.0009065 secs]
[GC [PSYoungGen: 26056K->184K(53824K)] 26056K->184K(88192K), 0.0011468 secs]
[GC [PSYoungGen: 51896K->184K(53824K)] 51896K->184K(88192K), 0.0009158 secs]
[GC [PSYoungGen: 51896K->192K(83392K)] 51896K->192K(117760K), 0.0008146 secs]
[GC [PSYoungGen: 83328K->0K(83392K)] 83328K->152K(117760K), 0.0014555 secs]
[GC [PSYoungGen: 83136K->0K(86976K)] 83288K->152K(121344K), 0.0008565 secs]
...
```

```
[GC [PSYoungGen: 87232K->0K(87296K)] 87384K->152K(121664K), 0.0003433 secs]
[GC [PSYoungGen: 87232K->0K(87296K)] 87384K->152K(121664K), 0.0007660 secs]
[GC [PSYoungGen: 87232K->0K(87296K)] 87384K->152K(121664K), 0.0007937 secs]
fin compteur=1000000000
Temps d'execution : 4031
Heap
 PSYoungGen      total 87296K, used 26193K [0x1eaf0000, 0x24040000, 0x24040000)
  eden space 87232K, 30% used [0x1eaf0000,0x20484420,0x24020000)
  from space 64K, 0% used [0x24020000,0x24020000,0x24030000)
  to   space 64K, 0% used [0x24030000,0x24030000,0x24040000)
 PSOldGen        total 34368K, used 152K [0x14040000, 0x161d0000, 0x1eaf0000)
  object space 34368K, 0% used [0x14040000,0x14066060,0x161d0000)
 PSPermGen       total 16384K, used 1790K [0x10040000, 0x11040000, 0x14040000)
  object space 16384K, 10% used [0x10040000,0x101ff968,0x11040000)
```

Le facteur d'amélioration des performances lié à l'utilisation de l'escape analysis peut être important.

L'escape analysis ne fonctionne qu'avec le mode C2 du compilateur (active avec l'option -server de la JVM Hotspot)

Résultat :

```
C:\Java\TestThreads\src>java -Xmx256m -client -verbose:gc -XX:+DoEscapeAnalysis
-XX:+PrintGCDetails -cp . fr.jmdoudoux.dej.thread.TestEscapeAnalysis
Unrecognized VM option '+DoEscapeAnalysis'
Could not create the Java virtual machine.
```

Avec le mode C1 du compilateur (activé avec l'option -client de la JVM Hotspot), l'activité du ramasse-miettes est importante et le temps d'exécution est multiplié par dix.

Résultat :

```
C:\Java\TestThreads\src>java -Xmx256m -client -verbose:gc -XX:+PrintGCDetails
-cp . fr.jmdoudoux.dej.thread.TestEscapeAnalysis
[GC [DefNew: 4480K->0K(4992K), 0.0001198 secs] 4602K->122K(15936K), 0.0002436 secs]
[GC [DefNew: 4480K->0K(4992K), 0.0001372 secs] 4602K->122K(15936K), 0.0003056 secs]
[GC [DefNew: 4480K->0K(4992K), 0.0001307 secs] 4602K->122K(15936K), 0.0002646 secs]
[GC [DefNew: 4480K->0K(4992K), 0.0001333 secs] 4602K->122K(15936K), 0.0003079 secs]
[GC [DefNew: 4480K->0K(4992K), 0.0001316 secs] 4602K->122K(15936K), 0.0002752 secs]
[GC [DefNew: 4480K->0K(4992K), 0.0001349 secs] 4602K->122K(15936K), 0.0002760 secs]
[GC [DefNew: 4480K->0K(4992K), 0.0001436 secs] 4602K->122K(15936K), 0.0003056 secs]
[GC [DefNew: 4480K->0K(4992K), 0.0001316 secs] 4602K->122K(15936K), 0.0002682 secs]
[GC [DefNew: 4480K->0K(4992K), 0.0001416 secs] 4602K->122K(15936K), 0.0003037 secs]
...
[GC [DefNew: 4480K->0K(4992K), 0.0001631 secs] 4602K->122K(15936K), 0.0002998 secs]
[GC [DefNew: 4480K->0K(4992K), 0.0001620 secs] 4602K->122K(15936K), 0.0003003 secs]
[GC [DefNew: 4480K->0K(4992K), 0.0001310 secs] 4602K->122K(15936K), 0.0002685 secs]
[GC [DefNew: 4480K->0K(4992K), 0.0001349 secs] 4602K->122K(15936K), 0.0002752 secs]
[GC [DefNew: 4480K->0K(4992K), 0.0001361 secs] 4602K->122K(15936K), 0.0002788 secs]
[GC [DefNew: 4480K->0K(4992K), 0.0001313 secs] 4602K->122K(15936K), 0.0002838 secs]
fin compteur=1000000000
Temps d'execution : 10078
Heap
 def new generation   total 4992K, used 2904K [0x10040000, 0x105a0000, 0x15590000)
  eden space 4480K,  64% used [0x10040000, 0x10316068, 0x104a0000)
  from space 512K,   0% used [0x10520000, 0x10520000, 0x105a0000)
  to   space 512K,   0% used [0x104a0000, 0x104a0000, 0x10520000)
 tenured generation   total 10944K, used 122K [0x15590000, 0x16040000, 0x20040000)
   the space 10944K,   1% used [0x15590000, 0x155aeaf0, 0x155aec00, 0x16040000)
 compacting perm gen  total 12288K, used 1752K [0x20040000, 0x20c40000, 0x24040000)
   the space 12288K,  14% used [0x20040000, 0x201f6080, 0x201f6200, 0x20c40000)
No shared spaces configured.
```

Si l'instance de type MonBean sort de la portée de la méthode, celle-ci est instanciée dans le heap.

Exemple :

```
package fr.jmdoudoux.dej.thread;


public class TestEscapeAnalysis {

  private static MonBean courant = null;

  private static class MonBean {
    private long       valeur;
    private static long compteur;

    public MonBean() {
      valeur = compteur++;
    }
  }

  public static void main(String[] args) {
    System.out.println("debut");
    long startTime = System.currentTimeMillis();

    for (long i = 0; i < 1000000000L; ++i) {
      MonBean monBean = new MonBean();
      courant = monBean;
    }

    long duree = System.currentTimeMillis() - startTime;
    System.out.println("fin compteur=" + MonBean.compteur);
    System.out.println("Temps d'execution : " + duree);
  }
}
```

Résultat :

```
C:\Java\TestThreads\src>java -Xmx256m -server -verbose:gc -XX:+DoEscapeAnalysis
 -XX:+PrintGCDetails -cp . fr.jmdoudoux.dej.thread.TestEscapeAnalysis
[GC [PSYoungGen: 83472K->16K(87296K)] 83612K->156K(121664K), 0.0005009 secs]
[GC [PSYoungGen: 87248K->16K(87296K)] 87388K->156K(121664K), 0.0007736 secs]
[GC [PSYoungGen: 87248K->16K(87296K)] 87388K->156K(121664K), 0.0005297 secs]
[GC [PSYoungGen: 87248K->16K(87296K)] 87388K->156K(121664K), 0.0005481 secs]
[GC [PSYoungGen: 87248K->16K(87296K)] 87388K->156K(121664K), 0.0005425 secs]
[GC [PSYoungGen: 87248K->16K(87296K)] 87388K->156K(121664K), 0.0006979 secs]
[GC [PSYoungGen: 87248K->16K(87296K)] 87388K->156K(121664K), 0.0006744 secs]
[GC [PSYoungGen: 87248K->16K(87296K)] 87388K->156K(121664K), 0.0005339 secs]
[GC [PSYoungGen: 87248K->16K(87296K)] 87388K->156K(121664K), 0.0006666 secs]
fin compteur=1000000000
Temps d'execution : 4438
Heap
 PSYoungGen      total 87296K, used 7023K [0x1eaf0000, 0x24040000, 0x24040000)
  eden space 87232K, 8% used [0x1eaf0000,0x1f1c7f00,0x24020000)
  from space 64K, 25% used [0x24020000,0x24024000,0x24030000)
  to   space 64K, 0% used [0x24030000,0x24030000,0x24040000)
 PSOldGen        total 34368K, used 140K [0x14040000, 0x161d0000, 0x1eaf0000)
  object space 34368K, 0% used [0x14040000,0x14063060,0x161d0000)
 PSPermGen       total 16384K, used 1790K [0x10040000, 0x11040000, 0x14040000)
  object space 16384K, 10% used [0x10040000,0x101ffa98,0x11040000)
```

L'instance est aussi créée dans la pile si elle est utilisée en paramètre d'une méthode invoquée tout en ne sortant pas de la portée du thread.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestEscapeAnalysis {

  private static class MonBean {
    private long       valeur;
    private static long compteur;

    public MonBean() {
      valeur = compteur++;
    }
  }

  public static void main(String[] args) {
    System.out.println("debut");
    long startTime = System.currentTimeMillis();

    for (long i = 0; i < 1000000000L; ++i) {
      MonBean monBean = new MonBean();
      traiter(monBean);
    }

    long duree = System.currentTimeMillis() - startTime;
    System.out.println("fin compteur=" + MonBean.compteur);
    System.out.println("Temps d'execution : " + duree);
  }

  private static MonBean traiter(MonBean monBean) {
    return monBean;
  }
}
```

Avec l'escape analysis activée, l'activité du ramasse-miettes est très faible.

Résultat :

```
C:\Java\TestThreads\src>java -Xmx256m -server -verbose:gc -XX:+DoEscapeAnalysis
-XX:+PrintGCDetails -cp . fr.jmdoudoux.dej.thread.TestEscapeAnalysis
debut
fin compteur=1000000000
Temps d'execution : 1062
Heap
 PSYoungGen      total 15040K, used 1572K [0x1eaf0000, 0x1fbb0000, 0x24040000)
  eden space 12928K, 12% used [0x1eaf0000,0x1ec791f8,0x1f790000)
  from space 2112K, 0% used [0x1f9a0000,0x1f9a0000,0x1fbb0000)
  to   space 2112K, 0% used [0x1f790000,0x1f790000,0x1f9a0000)
 PSOldGen        total 34368K, used 0K [0x14040000, 0x161d0000, 0x1eaf0000)
  object space 34368K, 0% used [0x14040000,0x14040000,0x161d0000)
 PSPermGen       total 16384K, used 1784K [0x10040000, 0x11040000, 0x14040000)
  object space 16384K, 10% used [0x10040000,0x101fe128,0x11040000)
```

Avec l'escape analysis désactivée, l'activité du ramasse-miettes est beaucoup plus intense et le temps d'exécution est multiplié par quatre.

Résultat :

```
C:\Java\TestThreads\src>java -Xmx256m -server -verbose:gc -XX:-DoEscapeAnalysis
-XX:+PrintGCDetails -cp . fr.jmdoudoux.dej.thread.TestEscapeAnalysis
[GC [PSYoungGen: 87232K->0K(87296K)] 87380K->148K(121664K), 0.0004464 secs]
[GC [PSYoungGen: 87232K->0K(87296K)] 87380K->148K(121664K), 0.0007208 secs]
[GC [PSYoungGen: 87232K->0K(87296K)] 87380K->148K(121664K), 0.0007322 secs]
[GC [PSYoungGen: 87232K->0K(87296K)] 87380K->148K(121664K), 0.0006540 secs]
[GC [PSYoungGen: 87232K->0K(87296K)] 87380K->148K(121664K), 0.0005763 secs]
...
[GC [PSYoungGen: 87232K->0K(87296K)] 87380K->148K(121664K), 0.0003632 secs]
[GC [PSYoungGen: 87232K->0K(87296K)] 87380K->148K(121664K), 0.0006383 secs]
[GC [PSYoungGen: 87232K->0K(87296K)] 87380K->148K(121664K), 0.0006906 secs]
[GC [PSYoungGen: 87232K->0K(87296K)] 87380K->148K(121664K), 0.0004864 secs]
fin compteur=1000000000
Temps d'execution : 4030
Heap
 PSYoungGen      total 87296K, used 29781K [0x1eaf0000, 0x24040000, 0x24040000)
  eden space 87232K, 34% used [0x1eaf0000,0x20805710,0x24020000)
  from space 64K, 0% used [0x24030000,0x24030000,0x24040000)
  to   space 64K, 0% used [0x24020000,0x24020000,0x24030000)
 PSOldGen        total 34368K, used 148K [0x14040000, 0x161d0000, 0x1eaf0000)
  object space 34368K, 0% used [0x14040000,0x14065060,0x161d0000)
 PSPermGen       total 16384K, used 1790K [0x10040000, 0x11040000, 0x14040000)
  object space 16384K, 10% used [0x10040000,0x101ffb20,0x11040000)
```

L'endroit où est alloué un objet est uniquement géré par la JVM. Les possibilités pour le développeur d'influencer ce choix sont restreintes car il n'est pas possible d'indiquer dans le code que cet objet doit être instancié dans la pile :

- configuration de certaines options de la JMV
- bien tenir compte de la portée des variables en limitant celle-ci au strict minimum

L'endroit où un objet est alloué importe peu sur la bonne exécution des traitements, cependant la mise en oeuvre de ces fonctionnalités peut significativement améliorer les performances.

37.12.4. Les restrictions d'accès sur les threads et les groupes de threads

Les restrictions d'accès aux fonctionnalités des classes Thread et ThreadGroup reposent sur l'utilisation d'un SecurityManager.

Les classes Thread et ThreadGroup possède une méthode checkAccess() qui va invoquer la méthode checkAccess() du SecurityManager associé à la JVM. Si l'accès n'est pas autorisé alors une exception de type SecurityException est levée.

Plusieurs méthodes de la classe ThreadGroup invoquent la méthode checkAccess() pour obtenir la permission d'exécution par le SecurityManager :

- ThreadGroup(ThreadGroup, String)
- destroy()
- getParent()
- resume()
- setDaemon(boolean)
- setMaxPriority(int)
- stop()
- suspend()
- enumerate(Thread[]) et enumerate(Thread[], boolean)
- enumerate(ThreadGroup[]) et enumerate(ThreadGroup[], boolean)
- interrupt()

Plusieurs méthodes de la classe Thread invoquent la méthode checkAccess() pour obtenir la permission d'exécution par le SecurityManager :

- Les constructeurs qui attendent en paramètre un groupe de threads
- stop()
- suspend()
- resume()
- setPriority(int)
- setName(String)
- setDaemon(boolean)
- setUncaughtExceptionHandler(UncaughtExceptionHandler)

Sans SecurityManager, il n'y a pas de restrictions d'accès pour modifier l'état d'un thread ou d'un groupe de threads par un autre thread.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestMonThreadSecManager {

  public static void main(String[] args) {
    final ThreadGroup threadGroup1 = new ThreadGroup("groupe1");
    final Thread t1 = new Thread(threadGroup1, new Runnable() {

      @Override
      public void run() {
        try {
          Thread.sleep(2000);
        } catch (InterruptedException e) {
        }
        System.out.println("fin thread 1");
      }
    }, "thread 1");
    t1.start();

    ThreadGroup threadGroup2 = new ThreadGroup("groupe2");
    Thread t2 = new Thread(threadGroup2, new Runnable() {

      @Override
      public void run() {
        t1.setPriority(Thread.MIN_PRIORITY);
        System.out.println("fin thread 2");
      }
    }, "thread 2");
    t2.start();

    Thread t3 = new Thread(threadGroup2, new Runnable() {

      @Override
      public void run() {
        threadGroup1.setMaxPriority(Thread.MIN_PRIORITY);
        System.out.println("fin thread 3");
      }
    }, "thread 3");
    t3.start();
  }
}
```

Résultat :

```
fin thread 2
fin thread 3
fin thread 1
```

Il est possible de définir son propre SecurityManager en créant une classe fille de la classe SecurityManager avec les méthodes checkAccess(Thread) et checkAccess(ThreadGroup) redéfinies selon les besoins.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class MonThreadSecManager extends SecurityManager {

  private Thread      threadPrincipal;
  private ThreadGroup threadGroupPrincipal;

  public MonThreadSecManager(Thread threadPrincipal) {
    this.threadPrincipal = threadPrincipal;
    this.threadGroupPrincipal = threadPrincipal.getThreadGroup();
  }

  public void checkAccess(Thread t) {

    if (t != null) {
      Thread threadCourant = Thread.currentThread();
      ThreadGroup threadGroupCourant = threadCourant.getThreadGroup();

      if (!threadPrincipal.equals(threadCourant)) {
        System.out.println("thread        " + t);
        System.out.println("threadCourant " + threadCourant);

        if (!t.getThreadGroup().equals(threadGroupCourant))
          throw new SecurityException("Can't modify the thread");
      }
    }
  }

  public void checkAccess(ThreadGroup g) {
    if (g != null) {
      Thread threadCourant = Thread.currentThread();
      ThreadGroup threadGroupCourant = threadCourant.getThreadGroup();

      if (!threadGroupPrincipal.equals(threadGroupCourant)) {
        System.out.println("threadGroup        " + g);
        System.out.println("threadGroupCourant " + threadGroupCourant);

        if (!g.equals(threadGroupCourant))
          throw new SecurityException("Can't modify the thread group");
      }
    }
  }
}
```

L'implémentation du SecurityManager ci-dessus effectue certains contrôles :

- le thread principal et son groupe de threads sont autorisés notamment pour permettre la création des threads et des groupes de threads
- l'accès à un thread n'est possible que si le thread courant appartient au même groupe que lui
- l'accès à un groupe de threads n'est possible que si le thread courant lui appartient

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestMonThreadSecManager {

  public static void main(String[] args) throws InterruptedException {

    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new MonThreadSecManager(Thread
          .currentThread()));
    }

    final ThreadGroup threadGroup1 = new ThreadGroup("groupe1");
    final Thread t1 = new Thread(threadGroup1, new Runnable() {
      @Override
      public void run() {
        try {
          Thread.sleep(2000);
        } catch (InterruptedException e) {
        }
        System.out.println("fin thread 1");
      }
    }, "thread 1");
    t1.start();

    ThreadGroup threadGroup2 = new ThreadGroup("groupe2");
    Thread t2 = new Thread(threadGroup2, new Runnable() {
      @Override
      public void run() {
        t1.setPriority(Thread.MIN_PRIORITY);
        System.out.println("fin thread 2");
      }
    }, "thread 2");
    t2.start();
    t2.join();

    Thread t3 = new Thread(threadGroup2, new Runnable() {
      @Override
      public void run() {
        threadGroup1.setMaxPriority(Thread.MIN_PRIORITY);
        System.out.println("fin thread 3");
      }
    }, "thread 3");
    t3.start();

    t1.join();
  }
}
```

Résultat :

```
thread        Thread[thread 1,5,groupe1]
threadCourant Thread[thread 2,5,groupe2]
Exception in thread "thread 2" java.lang.SecurityException: Can't modify the thread
    at fr.jmdoudoux.dej.thread.MonThreadSecManager.checkAccess(MonThreadSecManager.java:16)
    at java.lang.Thread.checkAccess(Thread.java:1306)
    at java.lang.Thread.setPriority(Thread.java:1056)
    at fr.jmdoudoux.dej.thread.TestMonThreadSecManager$2.run(TestMonThreadSecManager.java:33)
    at java.lang.Thread.run(Thread.java:662)
threadGroup        java.lang.ThreadGroup[name=groupe1,maxpri=10]
threadGroupCourant java.lang.ThreadGroup[name=groupe2,maxpri=10]
Exception in thread "thread 3" java.lang.SecurityException: Can't modify the thread group
    at fr.jmdoudoux.dej.thread.MonThreadSecManager.checkAccess(MonThreadSecManager.java:32)
    at java.lang.ThreadGroup.checkAccess(ThreadGroup.java:299)
    at java.lang.ThreadGroup.setMaxPriority(ThreadGroup.java:246)
    at fr.jmdoudoux.dej.thread.TestMonThreadSecManager$3.run(TestMonThreadSecManager.java:44)
    at java.lang.Thread.run(Thread.java:662)
fin thread 1
```

38. L'association de données à des threads

Chapitre 38

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Dans un environnement monothread, il suffit de déclarer une variable static pour stocker une valeur contextuelle.

Dans un contexte multithread, il est possible de stocker des données dans le contexte d'exécution de chaque thread. Ainsi chaque thread peut avoir sa propre instance d'une donnée. Ceci évite :

- d'avoir à passer ces données en paramètres dans les différentes méthodes exécutées par un thread
- d'avoir une seule instance statique partagée par tous les threads : chaque thread possède sa propre instance et évite ainsi la gestion des accès concurrents

Il existe plusieurs solutions pour permettre d'associer des données à un thread :

- définir son propre mécanisme qui repose sur un stockage des données dans une Map statique
- créer une classe fille de la classe Thread qui va encapsuler les données
- utiliser la classe ThreadLocal, qui est la solution recommandée

Il est possible de définir sa propre solution en utilisant une Map statique pour stocker les données. Cette solution est généralement compliquée car elle requiert une bonne prise en charge de certains points :

- il faut gérer les accès concurrents réalisés par les threads sur cette Map
- pour éviter de conserver des références sur des objets qui ne sont plus utilisés et qui pourraient être traités par le ramasse-miettes, il est nécessaire de les retirer de la Map dès que possible
- il faut retirer de la Map les données d'un thread qui est terminé : l'utilisation d'une WeakHashMap avec comme clé une référence sur le thread peut aider

Il est aussi possible de créer une classe qui hérite de la classe Thread et qui encapsule les données qui lui sont associées.

Exemple :

```
public class MonThread extends Thread {

  public String valeur;

  @Override
  public void run() {
    // traitement du thread
  }
}
```

Pour obtenir une donnée, il suffit d'invoquer la méthode Thread.currentThread() et de caster le résultat vers le type du thread pour avoir un accès à la variable.

Exemple :

```
    MonThread thread = (MonThread) Thread.currentThread();
    System.out.println(thread.valeur);
```

Cette solution est une des plus performante mais elle impose de pouvoir utiliser sa propre instance de thread ce qui n'est pas le cas par exemple lors de l'utilisation de frameworks.

Enfin, l'API Java Core propose la classe ThreadLocal. Une même instance de type ThreadLocal permet de stocker et obtenir différentes valeurs, une pour chaque thread. La valeur est dédiée au thread courant : n'importe quel code exécuté dans le thread peut avoir accès à la valeur et la modifier au besoin mais il n'est pas possible d'obtenir les valeurs des autres threads dans le thread courant. Cette solution est toujours utilisable mais sa simplicité de mise en oeuvre peut parfois masquer certains points qu'il est important de prendre en compte pour éviter des effets de bords.

Ce chapitre contient plusieurs sections :

- La classe ThreadLocal
- La classe InheritableThreadLocal
- La classe ThreadLocalRandom

38.1. La classe ThreadLocal

Il peut être pratique de vouloir stocker une donnée qui soit contextuelle à un thread : c'est le rôle de la classe ThreadLocal. La classe ThreadLocal permet d'encapsuler des données qui seront accessibles par tous les traitements exécutés dans le thread. Elle a été ajoutée à Java depuis sa version 1.2.

La classe ThreadLocal implémente un mécanisme qui permet d'associer une donnée à un thread et de permettre son accès dans tous les traitements exécutés par le thread. Ceci permet de faciliter l'accès à des données contextuelles (contexte transactionnel, sécurité, Locale, ...) par ses traitements sans avoir à les passer en paramètres à chaque méthode invoquée.

Un ThreadLocal peut être considéré comme un scope supplémentaire, en plus des scopes existants (application, session, request) dont la portée est le thread. Une instance stockée dans un ThreadLocal ne peut être utilisée que par le thread qui l'y a déposé. Elle permet de définir des variables dont chaque thread possédera sa propre instance et un autre thread ne peut pas avoir accès à une instance qui n'est pas la sienne.

L'utilisation d'un ThreadLocal peut avoir plusieurs objectifs :

- permettre à une donnée d'être utilisable de manière globale dans tous les traitements d'un thread
- pouvoir utiliser des classes non thread-safe dans un contexte multithread en fournissant sa propre instance à chaque thread. Ceci permet d'éviter la synchronisation de l'accès à une unique instance partagée par plusieurs threads (exemple : SimpleDateFormat)
- partager une donnée dans les couches d'une application utilisées dans les traitements d'un thread

A partir de Java 5, la classe ThreadLocal est typée avec un generic pour assurer un control sur le type.

L'utilisation d'un ThreadLocal peut être extrêmement pratique mais aussi être à l'origine de soucis notamment de fuites de mémoires si certaines précautions ne sont pas prises.

De nombreux frameworks utilisent des ThreadLocal pour stocker des données contextuelles à chaque thread.

38.1.1. L'utilisation de la classe ThreadLocal

La classe ThreadLocal permet de stocker une variable qui ne pourra être accédée que par un seul thread. Même si plusieurs threads utilisent la même instance de ThreadLocal pour obtenir la variable, chaque thread obtiendra celle qui lui est associée.

Pour créer une instance de type ThreadLocal, il suffit d'utiliser l'opérateur new.

Exemple (code Java 1.2) :

```
private ThreadLocal monThreadLocal = new ThreadLocal();
```

Une seule instance de type ThreadLocal est requise : la création de cette instance n'a besoin d'être réalisée qu'une seule fois pour tous les threads de la JVM quel que soit le thread qui la créée.

Exemple (code Java 1.2) :

```
package fr.jmdoudoux.dej.thread;

public class TestThreadLocal {

  public static void main(String[] args) {
    MonTraitementAvecTL monTraitementAvecTL = new MonTraitementAvecTL();
    Thread thread1 = new Thread(monTraitementAvecTL);
    Thread thread2 = new Thread(monTraitementAvecTL);
    thread1.start();
    thread2.start();
  }
}
```

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

public class MonTraitementAvecTL implements Runnable {
  private ThreadLocal<String> monThreadLocal = new ThreadLocal<String>();

  public void run() {
    System.out.println("Mon traitement " + Thread.currentThread().getName()
        + " monThreadLocal=" + monThreadLocal);
    monThreadLocal.set("Valeur pour " + Thread.currentThread().getName());
    try {
      Thread.sleep(1000);
      String maValeur = monThreadLocal.get();
      System.out.println("Valeur = " + maValeur);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
Mon traitement Thread-0 monThreadLocal=java.lang.ThreadLocal@1bc4459
Mon traitement Thread-1 monThreadLocal=java.lang.ThreadLocal@1bc4459
Valeur = Valeur pour Thread-0
Valeur = Valeur
pour Thread-1
```

Dans l'exemple ci-dessus, une seule instance de type MonTraitementAvecTL est créée, donc une seule instance de type ThreadLocal est créée.

Il est cependant généralement préférable de déclarée la variable ThreadLocal statique même si au premier abord cela peut paraître surprenant de définir une variable statique pour obtenir une instance contextuelle. C'est d'ailleurs une recommandation dans la Javadoc de la classe ThreadLocal.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class MonTraitementAvecTL implements Runnable {
  private static ThreadLocal<String> monThreadLocal = new ThreadLocal<String>();

  public void run() {
    System.out.println("Mon traitement " + Thread.currentThread().getName()
        + " monThreadLocal=" + monThreadLocal);
    monThreadLocal.set("Valeur pour " + Thread.currentThread().getName());
    try {
      Thread.sleep(1000);
      String maValeur = monThreadLocal.get();
      System.out.println("Valeur = " + maValeur);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.thread;
public class TestThreadLocal {

  public static void main(String[] args) {
    Thread thread1 = new Thread(new MonTraitementAvecTL());
    Thread thread2 = new Thread(new MonTraitementAvecTL());
    thread1.start();
    thread2.start();
  }
}
```

Résultat :

```
Mon traitement Thread-0 monThreadLocal=java.lang.ThreadLocal@1bc4459
Mon traitement
Thread-1 monThreadLocal=java.lang.ThreadLocal@1bc4459
Valeur = Valeur
pour Thread-1
Valeur = Valeur
pour Thread-0
```

Si chaque thread possède son instance cela fonctionne aussi mais le ThreadLocal perd de son intérêt puisque chaque instance ne va permettre que l'accès à la variable du thread.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class MonTraitementAvecTL implements Runnable {
  private ThreadLocal<String> monThreadLocal = new ThreadLocal<String>();

  public void run() {
    System.out.println("Mon traitement " + Thread.currentThread().getName()
        + " monThreadLocal=" + monThreadLocal);
    monThreadLocal.set("Valeur pour " + Thread.currentThread().getName());
    try {
      Thread.sleep(1000);
      String maValeur = monThreadLocal.get();
      System.out.println("Valeur = " + maValeur);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestThreadLocal {

  public static void main(String[] args) {
    Thread thread1 = new Thread(new MonTraitementAvecTL());
    Thread thread2 = new Thread(new MonTraitementAvecTL());
    thread1.start();
    thread2.start();
  }
}
```

Résultat :

```
Mon traitement Thread-0 monThreadLocal=java.lang.ThreadLocal@150bd4d
Mon traitement Thread-1 monThreadLocal=java.lang.ThreadLocal@12b6651
Valeur = Valeur pour
Thread-1
Valeur = Valeur pour
Thread-0
```

Il ne faut surtout pas initialiser la valeur du ThreadLocal en utilisant un bloc d'initialisation static.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

public class MonCompteurTL {

  private static ThreadLocal<Integer> compteur = new ThreadLocal<Integer>();

  static {
    // surtout ne pas faire cela
    compteur.set(0);
  }

  public int get() {
    return compteur.get();
  }

  public int incrementer() {
    int valeur = compteur.get();
    valeur++;
    compteur.set(valeur);
    return valeur;
  }

  public void retirer() {
    compteur.remove();
  }
}
```

Celui-ci ne sera invoqué qu'une seule fois lors du chargement de la classe et la valeur 0 ne sera assignée au ThreadLocal que pour le thread courant. Lorsque les autres threads invoqueront la méthode get() pour la première fois sur le ThreadLocal ils obtiendront null et pas 0.

Pour que la valeur soit initialisée pour chaque thread, il faut redéfinir la méthode initialValue() pour qu'elle renvoie la valeur initiale. Celle-ci sera alors utilisée si le ThreadLocal ne possède pas encore de valeur pour le thread courant.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

public class MonCompteurTL {

  private static ThreadLocal<Integer> compteur = new ThreadLocal<Integer>() {
    @Override
    protected Integer initialValue() {
      return Integer.valueOf(0);
    }
  };

  public int get() {
    return compteur.get();
  }

  public int incrementer() {
    int valeur = compteur.get();
    valeur++;
    compteur.set(valeur);
    return valeur;
  }


  public void retirer() {
    compteur.remove();
  }
}
```

Tous les threads peuvent accéder à l'instance de type ThreadLocal : seul le thread qui a ajouté une donnée en utilisant la méthode set() pourra obtenir cette variable qui lui est propre. Si plusieurs threads invoquent la méthode set() sur l'unique instance de ThreadLocal, chacun obtiendra la variable qu'il a passée en paramètre. Un thread ne peut pas obtenir la variable qui est associé à un autre thread.

Un ThreadLocal ne peut encapsuler qu'une variable pour un thread. Si plusieurs variables doivent être stockées pour un même thread, il faut déclarer une instance de type ThreadLocal pour chaque variable ou encapsuler les différentes variables dans une classe.

La méthode set() permet de préciser l'instance qui est associée au thread courant.

monThreadLocal.set("ma valeur");

La méthode get() permet d'obtenir l'instance qui est associée au thread courant.

String maValeur = (String) monThreadLocal.get();

Depuis Java 5, il est possible d'associer un type generic au type ThreadLocal, ce qui évite d'avoir à caster la valeur de retour de la méthode get().

Exemple (code Java 5.0) :

```
private ThreadLocal monThreadLocal<String> = new ThreadLocal<String>();

monThreadLocal.set("ma valeur");
String valeur = monThreadLocal.get();
```

La méthode set() permet à chaque thread de stocker sa propre valeur : elle ne peut donc pas être utilisée pour mettre une valeur initiale. Pour définir une valeur initiale, il faut créer une classe fille de ThreadLocal et redéfinir la méthode initialValue().

Exemple :

```
private ThreadLocal monThreadLocal = new ThreadLocal<String>() {
  @Override protected String initialValue(){
    return "Valeur initiale";
  }
};
```

Si un thread n'a jamais utilisé la méthode set() de l'instance du ThreadLocal et invoque la méthode get(), il obtiendra en retour la valeur initiale.

38.1.2. Le fonctionnement interne d'un ThreadLocal

L'implémentation de la classe ThreadLocal ne permet qu'un accès aux variables du thread courant : il n'est donc pas possible d'accéder à toutes les variables de tous les threads.

La classe ThreadLocal est incluse dans le JDK à partir de la version 1.2. L'implémentation de la première version de la classe ThreadLocal souffrait d'un problème de performance lié à une forte contention lors de l'utilisation multithread. Elle a donc été revue dans la version 1.3 et 1.4 notamment pour permettre une amélioration de ses performances.

En Java 1.2, l'implémentation de ThreadLocal utilise une WeakHashMap dont les accès sont synchronized. Cependant cette implémentation souffre d'une forte contention et induit un surcoût en termes de performance notamment lorsque le nombre de threads s'accroit.

Le contrat de la classe ThreadLocal n'a pas changé en Java 1.3 mais son implémentation dans le JDK de Sun a été entièrement revue pour permettre une amélioration de ses performances en évitant la contention liée à la gestion des accès concurrents faite dans l'implémentation précédente avec synchronized. L'implémentation a été changée pour que la Map soit stockée dans le thread lui-même sous la forme d'un champ de type ThreadLocal.ThreadLocalMap nommé threadLocals qui n'est pas accessible directement. Du coup, il n'est plus utile de gérer les accès concurrents puisque seul le thread aura accès aux données.

Chaque thread possède deux propriétés de type ThreadLocal.ThreadLocalMap :

- threadLocals : chaque élément de la map est un objet ThreadLocal.ThreadLocalMap.Entry qui hérite de la classe WeakReference<ThreadLocal>
- inheritableThreadLocals

La clé de ces Map est l'instance de type ThreadLocal.

L'implémentation de la classe ThreadLocalMap stocke ses entrées dans un tableau de type ThreadLocalMap.Entry dont la taille initiale est de 16 éléments. Si la taille du tableau doit être agrandie, celle-ci est doublée.

Les données d'un ThreadLocal sont stockées dans la propriété threadLocals de type java.lang.ThreadLocal.ThreadLocalMap de chaque thread. La classe ThreadLocal.ThreadLocalMap encapsule les données de chaque ThreadLocal pour le thread courant. Son implémentation est similaire à une Map mais personnalisée et dédiée pour l'utilisation de ThreadLocal. C'est la raison pour laquelle c'est une classe interne. Les éléments stockés sont de type ThreadLocal.ThreadLocalMap.Entry qui hérite de WeakReference<ThreadLocal> . Elle encapsule :

- une clé de type WeakReference<ThreadLocal>
- une valeur de type Object

Lorsque la méthode set() de la classe ThreadLocal est invoquée, elle recherche dans la map threadLocals du thread courant l'entrée dont la clé est l'instance elle-même et lui associé la valeur passée en paramètre.

Lorsque la méthode get() de la classe ThreadLocal est invoquée, elle recherche dans la map threadLocals du thread courant l'entrée dont la clé est l'instance elle-même et renvoie la valeur associée si elle est présente, sinon elle renvoie la valeur d'initialisation.

Lors de l'invocation de la méthode get() ou set() d'un ThreadLocal, un traitement est exécuté pour retrouver la valeur dans le ThreadLocalMap :

- obtenir le thread courant
- obtenir le ThreadLocalMap du thread courant
- obtenir ou modifier l'entrée dans la Map dont la clé est l'instance courant de ThreadLocal

Pour faciliter la recherche de la clé dans la Map, chaque instance de type ThreadLocal possède une valeur pré-calculée qui favorise la recherche de l'instance de type ThreadLocal dans la map.

Chaque ThreadLocal possède un threadLocalHashCode dédié incrémenté de la valeur 0x61C88647 qui est aussi la valeur initiale. Ce threadLocalHashCode est utilisé pour déterminer l'index de l'entrée dans le tableau en appliquant la formule threadLocalHashCode & (taille_du_tableau -1).

La valeur 0x61c88647 est utilisée pour incrémenter la valeur de hachage de chaque entrée du thread local (une pour chaque thread qui en a besoin) afin de répartir au mieux les différentes valeurs de hachage des clés. C'est d'autant plus utile que la classe ThreadLocalMap n'utilise pas une collection pour son implémentation mais un tableau.

La classe ThreadLocal.ThreadLocalMap ne met en oeuvre aucun mécanisme dédié pour retirer les éléments inutilisés. Elle ne propose pas de mécanisme pour vérifier les clés de type WeakReference<ThreadLocal> qui pointent vers un objet inutilisé : ceci peut entraîner la conservation de références inutilisées pouvant provoquer une fuite de mémoire.

Cependant, lorsqu'un thread se termine, la JVM invoque sa méthode exit() qui remet à null ses propriétés threadLocals et inheritableThreadLocals : ceci garantit que les données liées au ThreadLocal seront récupérées par le ramasse-miettes.

38.1.3. ThreadLocal et fuite de mémoire

Il est nécessaire d'utiliser un ThreadLocal avec attention afin d'éviter d'avoir, sous certaines circonstances, des fuites de mémoire.

Le fait de ne pas déclarer static un ThreadLocal et donc de l'utiliser comme une variable d'instance peut aussi induire une certaine latence dans la récupération de la mémoire. L'implémentation de ThreadLocalMap utilise une clé ThreadLocalMap.Entry qui est une référence faible mais aucun thread démon n'existe pour les traiter. Ce sont les utilisations suivantes du ThreadLocal de chaque thread qui invoquent la méthode ThreadLocalMap.expungeStateEntries() qui retirent les clés inutilisées.

La classe ThreadLocal permet de fournir une instance d'un type dédiée à chaque thread. Chaque instance n'est donc accédée que par un seul thread et reste donc thread-safe. Tant que le thread est en cours d'exécution, toutes les données qui lui sont associées sont référencées par le ThreadLocal et donc le ramasse-miettes ne peut pas récupérer ces objets même s'ils ne sont plus utilisés par l'application car il reste au moins une référence.

Les différentes instances de chaque threads sont stockées dans une collection de type Map : ThreadLocalMap. Celle-ci utilise des références faibles (WeakReference) pour les clés. Le fait que les entrées soient des références faibles pourrait laisser croire qu'il est inutile de faire le ménage. Ces références faibles sont utilisées pour déterminer si l'instance du ThreadLocal peut être récupérée par le ramasse-miettes. Un prérequis est que plus aucune référence sur cette instance du thread n'existe dans la JVM : c'est généralement le cas lorsque le thread se termine.

La fin de l'exécution d'un Runnable ou d'un Callable ne signifie pas forcément la fin du thread : ce n'est pas le cas par exemple lorsque le thread fait partie d'un pool de threads. Un pool de threads est notamment utilisé par un java.util.concurrent.Executor ou par les serveurs d'applications.

Dans le cas de l'utilisation d'un pool de threads, la durée de vie d'un thread peut être très longue et durant cette période plusieurs tâches peuvent être exécutées. Si le threadLocal n'est pas nettoyé à la fin de chaque tâche, la tâche suivante obtient les valeurs stockées par la ou les tâches précédentes. C'est notamment le cas avec le pool de threads de traitement des servlets d'un conteneur web ou d'un ExecutorService.

38.1.3.1. Les fuites de mémoires dans un serveur d'applications

L'utilisation d'un ThreadLocal dans une application exécutée dans un serveur d'applications peut potentiellement conduire à une fuite de mémoire. Une mauvaise utilisation de ThreadLocal peut engendrer une fuite de ressources nommée classloader leaks qui se traduit généralement par un manque de mémoire dans la permgen.

C'est notamment le cas avec les serveurs d'applications qui utilisent un pool de threads pour réaliser différents traitements, par exemple le traitement des requêtes http avec des servlets. Si ces traitements utilisent un ou plusieurs ThreadLocal pour stocker des données contextuelles et que ces données ne sont pas explicitement retirées des threads locaux alors celles-ci sont conservées jusqu'à l'arrêt du serveur d'application.

Pour comprendre le problème, il faut se souvenir que sans être explicitement retirée du ThreadLocal, une donnée est conservée jusqu'à la fin du thread. Ceci même si l'application est arrêtée puisque le pool de thread n'est pas lié au cycle de vie de l'application mais au cycle de vie du conteneur dans lequel l'application s'exécute.

De plus, généralement, les classes des webapp sont chargées grâce à un classloader dédié. Chaque classe possède une référence sur le classloader qui l'a chargée. Si un ThreadLocal conserve une référence sur une instance d'une classe de l'application, possédant elle-même une référence sur l'instance du classloader, celle-ci ne peut pas être récupérée par le ramasse-miette. La classe du classloader reste chargée ainsi que toutes les classes qu'elle a chargées.

Un redéploiement de la webapp ne permet pas de supprimer les références ainsi que leur classloader : chaque déploiement va créer une nouvelle instance qui ne sera jamais récupérée par le ramasse-miettes. Ainsi le rechargement de l'application charge toutes les classes requises avec un nouveau classloader, mais toutes les classes du chargement précédent restent dans la permgen. Après plusieurs rechargements, dont le nombre dépend de la taille de la permgen et du nombre de classes chargées, il est possible que la taille de la permgen soit atteinte ce qui lève une exception de type OutOfMemoryError. Ce phénomène se nomme classloader leaks.

Il faut noter que les instances liées aux ThreadLocal et aux classloaders restent dans le heap mais cela ne pose généralement pas de soucis majeurs.

Cette problématique est d'autant plus fréquente dans les environnements de développement dans lesquels l'application est très fréquemment redéployée. Le conteneur web Apache Tomcat a ainsi tenté de pallier ces problèmes, notamment :

- Tomcat 6.0 propose plusieurs mécanismes pour tenter de limiter ces problématiques
- Tomcat 7.0, à partir de la version 7.0.6, a changé de stratégie en utilisant une recréation du pool de threads pour gérer la majorité des fuites

Cela ne signifie cependant pas qu'il ne faut pas utiliser de ThreadLocal mais qu'il est nécessaire de s'assurer que l'instance est retirée du ThreadLocal lorsque celle-ci n'est plus utile.

Le problème est que c'est généralement plus facile à dire qu'à faire :

- il est très facile de créer un ThreadLocal et d'y associer une instance pour le thread courant
- il est plus complexe de déterminer le moment où cette instance n'est plus utile

Une solution consiste à utiliser un filtre dans une application web qui va retirer les instances du ou des ThreadLocal une fois qu'une requête HTTP est traitée. Ce filtre peut aussi être utilisé pour créer une instance et l'associer à un ThreadLocal.

Exemple (code Java 5.0) :

```
public void doFilter(ServeletRequest request, ServletResponse) {
  try{
    // set ThreadLocal variable
    chain.doFilter(request, response)
  }finally{
    // remove threadLocal variable.
  }
}
```

38.1.3.2. Les fuites de mémoires dans un Executor

Il faut aussi être très prudent avec l'utilisation de ThreadLocal pour des tâches exécutées dans un Executor. Un Executor utilise un pool de threads pour exécuter les différentes tâches et ainsi limiter la création de threads. Il faut retirer les valeurs associées aux ThreadLocal car sinon la tâche suivante exécutée dans le thread va obtenir les valeurs stockées par la tâche précédente quand bien même celle-ci est terminée puisque ses valeurs sont stockées dans les maps du thread.

Pour déterminer à quel moment un objet associé à un ThreadLocal est récupéré par le ramasse-miettes, la méthode finalize() de sa classe est redéfinie.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

public class MonContexteTL extends ThreadLocal<MonContexte> {

  protected MonContexte initialValue() {
    return new MonContexte("Inconnu", 100);
  }

  @Override
  protected void finalize() throws Throwable {
    System.out.println("finalize() " + this);
    super.finalize();
  }
}
```

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

public class TestThreadLocal {
  private static final MonContexteTL monContextTL = new MonContexteTL();

  public static void main(String[] args) throws InterruptedException {
    Thread thread = new Thread() {
      @Override
      public void run() {
        System.out.println(monContextTL.get());
      }
    };
    thread.start();
    thread.join();
    executerGC();
    System.out.println("Fin");
  }

  private static void executerGC() throws InterruptedException {
    Thread.sleep(1000);
    System.gc();
    Thread.sleep(1000);
    System.gc();
    Thread.sleep(1000);
    System.gc();
  }
}
```

Résultat :

```
fr.jmdoudoux.dej.thread.MonContexte@1bc4459 [user=Inconnu, valeur=100]
finalize() fr.jmdoudoux.dej.thread.MonContexte@1bc4459
[user=Inconnu, valeur=100]
Fin
```

Si le thread est dans un pool, le ThreadLocal n'est pas récupéré par le ramasse-miettes puisque le thread ne se termine pas à la fin de l'exécution des traitements.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class TestThreadLocal {
  private static final MonContexteTL monContextTL = new MonContexteTL();
  public static void main(String[] args) throws InterruptedException {
    ExecutorService executor = Executors.newSingleThreadExecutor();
    executor.execute(new Runnable() {
      public void run() {
        System.out.println(monContextTL.get());
      }
    });
    executerGC();
    executor.execute(new Runnable() {
      public void run() {
        System.out.println(monContextTL.get());
      }
    });
    executerGC();
    System.out.println("Fin");
  }

  private static void executerGC() throws InterruptedException {
    Thread.sleep(1000);
    System.gc();
    Thread.sleep(1000);
    System.gc();
    Thread.sleep(1000);
    System.gc();
  }
}
```

Résultat :

```
fr.jmdoudoux.dej.thread.MonContexte@1820dda [user=Inconnu, valeur=100]
fr.jmdoudoux.dej.thread.MonContexte@1820dda [user=Inconnu, valeur=100]
Fin
```

La JVM ne s'arrête pas car le thread de l'Executor ne se termine pas : l'instance de MonContexte associée à ce thread n'est donc pas récupérée par le ramasse-miettes.

Il ne faut surtout pas oublier d'invoquer la méthode shutdown() de l'ExecutorService dès que celui-ci n'est plus nécessaire pour lui permettre de terminer l'exécution de ses threads.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class TestThreadLocal {
  private static final MonContexteTL monContextTL = new MonContexteTL();
  public static void main(String[] args) throws InterruptedException {
    ExecutorService executor = Executors.newSingleThreadExecutor();
    executor.execute(new Runnable() {
      public void run() {
        System.out.println(monContextTL.get());
      }
    });
    executerGC();
    executor.execute(new Runnable() {
      public void run() {
        System.out.println(monContextTL.get());
      }
    });
    executor.shutdown();
    executerGC();
    System.out.println("Fin");
  }

  private static void executerGC() throws InterruptedException {
    Thread.sleep(1000);
    System.gc();
    Thread.sleep(1000);
    System.gc();
    Thread.sleep(1000);
    System.gc();
  }
}
```

Résultat :

```
fr.jmdoudoux.dej.thread.MonContexte@1820dda [user=Inconnu, valeur=100]
fr.jmdoudoux.dej.thread.MonContexte@1820dda [user=Inconnu, valeur=100]
finalize() fr.jmdoudoux.dej.thread.MonContexte@1820dda
[user=Inconnu, valeur=100]
Fin
```

38.1.4. Bonnes pratiques pour l'utilisation d'un ThreadLocal

L'utilisation d'un ThreadLocal permet de facilement assurer un accès global à des objets dans tous les traitements du thread concerné. Il ne faut cependant pas abuser de cette facilité par exemple en ajoutant beaucoup d'objets. De plus, l'ajout de certains objets peut créer des dépendances qui ne sont pas forcément souhaitables.

Un ThreadLocal ne doit être utilisé que pour partager une donnée dans le contexte d'une exécution. Il ne doit pas être utilisé pour partager des données entre plusieurs exécutions.

Comme indiqué dans la Javadoc, il est recommandé de définir une instance de type ThreadLocal comme private static.

Il est toujours préférable que la méthode set() soit invoquée au moins une fois avant d'invoquer la méthode get(). L'initialisation d'un TheadLocal avec une valeur par défaut doit impérativement se faire en redéfinissant la méthode initialValue() : toute autre solution ne fonctionnera pas correctement.

Il y a bien sûr le cas où la valeur récupérée sera celle définie à l'initialisation mais il est aussi possible sous certaines circonstances d'obtenir une valeur définie par un autre traitement si celle-ci n'est pas retirée lorsqu'elle n'a plus d'utilité.

Attention : selon la manière dont le serveur d'applications charge les classes d'une webapp, il est même possible de partager des données entre plusieurs applications. Par exemple, si la webapp est déployée deux fois et que le serveur utilise le même classloader pour les deux webapp alors il est possible d'obtenir une valeur d'un ThreadLocal qui a été assignée par l'autre application.

Si cela est possible, il est préférable d'éviter d'utiliser les ThreadLocal dans les applications exécutées dans un serveur d'applications, sinon il faut prendre toutes les précautions nécessaires pour éviter les fuites de ressources.

Les objets encapsulés dans un ThreadLocal doivent être retirés lorsqu'ils ne sont plus utilisés pour permettre de supprimer la référence qui les lie au ThreadLocal et ainsi pouvoir éventuellement récupérer la mémoire grâce au ramasse-miettes. D'une manière générale, il est toujours important de retirer d'un ThreadLocal toutes les valeurs qui ne sont plus utiles dès que possible. Pour retirer la valeur d'un ThreadLocal associée au thread courant, il faut invoquer la méthode remove() qui va permettre de retirer l'entrée correspondante dans le ThreadLocalMap plutôt que la méthode set(null).

Le meilleur moyen de s'assurer que la donnée est retirée d'un ThreadLocal est d'invoquer la méthode remove() du ThreadLocal dans un bloc finally. Le bloc try correspondant doit englober les traitements qui invoquent la méthode set() du ThreadLocal. Ce bloc try/catch peut être utilisé par exemple :

- dans la méthode run() d'un Runnable
- dans la méthode doFilter() d'un filtre d'une webapp
- dans les méthodes doGet() ou doPost() d'une servlet

Si plusieurs valeurs doivent être stockées et modifiées dans un ThreadLocal, il est préférable de les encapsuler dans une classe mutable et d'en stocker une instance dans le ThreadLocal.

Exemple (code Java 5.0) :

```
private static ThreadLocal<Integer> monThreadLocal = new ThreadLocal<Integer>();

// ...
monThreadLocal.set(0);
// ...
int valeur = monThreadLocal.get();
valeur++;
monThreadLocal.set(valeur);
```

Il est plus performant de rechercher l'instance et de modifier la valeur encapsulée plutôt que de rechercher la valeur et la modifier (cette opération requiert une nouvelle recherche de l'entrée dans la map). Cela permet aussi de stocker des valeurs primitives directement sans avoir à utiliser un wrapper et l'autoboxin/unboxing.

Exemple (code Java 5.0) :

```
class MaValeur {
   public int valeur;
}

// ...
private static ThreadLocal<MaValeur> monThreadLocal = new ThreadLocal<MaValeur>() {
            @Override
            protected MaValeur initialValue() {
                  return new MaValeur();
            }
      };
// ....
MaValeur maValeur = monThreadLocal.get();
maValeur.valeur++;
```

Remarque : l'exemple ci-dessus est volontairement simpliste pour ne garder que l'essentiel du principe de mise en oeuvre.

38.2. La classe InheritableThreadLocal

Ajoutée à Java 1.2, la classe InheritableThreadLocal permet de transmettre la valeur associée au thread courant à ses threads fils.

Elle hérite de la classe ThreadLocal. Elle ne possède que le constructeur par défaut.

Un thread peut lancer un autre thread : dans ce cas, le nouveau thread possède sa propre valeur dans le ThreadLocal qui sera vide par défaut.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

public class TestInheritableThreadLocal {
  public static void main(String[] args) throws InterruptedException {
    final ThreadLocal<String> threadLocal = new ThreadLocal<String>();
    threadLocal.set("valeur1");
    afficherValeur(threadLocal);
    Thread t = new Thread() {
      public void run() {
        afficherValeur(threadLocal);
        threadLocal.set("valeur2");
        afficherValeur(threadLocal);
      }
    };
    t.start();
    t.join();
    afficherValeur(threadLocal);
  }

  private static void afficherValeur(final ThreadLocal<String> threadLocal) {
    System.out.println(Thread.currentThread().getName() + " : "
        + threadLocal.get());
  }
}
```

Résultat :

```
main :
valeur1
Thread-0 : null
Thread-0 : valeur2
main : valeur1
```

Il peut être utile d'avoir par défaut les valeurs du thread parent : c'est le rôle de la classe InheritableThreadLocal.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

public class TestInheritableThreadLocal {
  public static void main(String[] args) throws InterruptedException {
    final ThreadLocal<String> threadLocal = new InheritableThreadLocal<String>();
    threadLocal.set("valeur1");
    afficherValeur(threadLocal);
    Thread t = new Thread() {
      public void run() {
        afficherValeur(threadLocal);
        threadLocal.set("valeur2");
        afficherValeur(threadLocal);
      }
    };
    t.start();
    t.join();
    afficherValeur(threadLocal);
  }

  private static void afficherValeur(final ThreadLocal<String> threadLocal) {
    System.out.println(Thread.currentThread().getName() + " : "
        + threadLocal.get());
  }
}
```

Résultat :

```
main : valeur1
Thread-0 : valeur1
Thread-0 :
valeur2
main : valeur1
```

Une fois le thread créé, il possède sa propre variable. Les valeurs de chaque threads sont stockées dans la propriété inheritableThreadLocals de type ThreadLocalMap.

La valeur de chaque thread reste donc indépendante : une modification de la valeur du thread parent après la création des threads fils ne modifie pas leur valeur correspondante. La valeur est simplement associée au thread au moment de sa création.

Elle ne définit qu'une seule méthode :

Méthode	Rôle
protected T childValue(T parentValue)	Fournir la valeur par défaut : elle renvoie la valeur associée au thread parent mais elle peut être redéfinie pour renvoyer une autre valeur

L'implémentation de la méthode childValue() renvoie simplement la valeur du thread parent passée en paramètre : donc, par défaut, la valeur est celle associée au thread parent si celle-ci est définie. Il est possible de redéfinir la méthode childValue() pour personnaliser la valeur initiale.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

public class TestInheritableThreadLocal {
  public static void main(String[] args) throws InterruptedException {
    final ThreadLocal<String> threadLocal = new InheritableThreadLocal<String>() {
      @Override
      protected String childValue(String parentValue) {
        return parentValue + " fils";
      }
    };
    threadLocal.set("valeur");
    afficherValeur(threadLocal);
    Thread t = new Thread() {
      public void run() {
        afficherValeur(threadLocal);
      }
    };
    t.start();
    t.join();
  }

  private static void afficherValeur(final ThreadLocal<String> threadLocal) {
    System.out.println(Thread.currentThread().getName() + " : "
        + threadLocal.get());
  }
}
```

Résultat :

```
main : valeur
Thread-0 : valeur fils
```

Attention : lors de l'utilisation d'un InheritableThreadLocal, il ne faut stocker que des objets immuables ou thread-safe car ces objets sont partagés entre le thread parent et ses threads fils.

38.3. La classe ThreadLocalRandom

La classe java.util.concurrent.ThreadLocalRandom, ajoutée à Java 7, encapsule un générateur de nombres aléatoires qui est dédié uniquement au thread courant.

Son utilisation permet de limiter la contention liée à l'utilisation d'un seul générateur, avec l'invocation de la méthode random() de la classe Math, en associant à chaque thread sa propre instance du générateur, améliorant ainsi les performances lors de l'utilisation dans des traitements parallèles.

Elle hérite de la classe Random. Elle possède plusieurs méthodes :

Méthode	Rôle
static ThreadLocalRandom current()	Renvoyer l'instance de type ThreadLocalRandom associée au thread courant
protected int next(int bits)	Obtenir la prochaine valeur pseudo-aléatoire
double nextDouble(double n)	Obtenir la prochaine valeur flottante pseudo-aléatoire comprise entre 0 et la valeur fournie en paramètre
double nextDouble(double least, double bound)	Obtenir la prochaine valeur flottante pseudo-aléatoire comprise entre la première valeur fournie incluse et la seconde valeur fournie exclue
int nextInt(int least, int bound)	Obtenir la prochaine valeur entière pseudo-aléatoire comprise entre la première valeur fournie incluse et la seconde valeur fournie exclue
long nextLong(long n)	Obtenir la prochaine valeur entière pseudo-aléatoire comprise entre 0 et la valeur fournie en paramètre
long nextLong(long least, long bound)	Obtenir la prochaine valeur entière pseudo-aléatoire comprise entre la première valeur fournie incluse et la seconde valeur fournie exclue
void setSeed(long seed)	Lève une exception de type UnsupportedOperationException

Son utilisation est relativement simple :

- il faut invoquer la méthode static current() pour obtenir l'instance associée au thread courant
- invoquer une des méthodes nextXXX() pour obtenir une valeur pseudo aléatoire

Exemple (code Java 7) :

```
long l = ThreadLocalRandom.current().nextLong(22L);
```

Il est aussi possible d'obtenir une valeur entre deux bornes fournies en paramètre d'une surcharge de la méthode nextXXX()

Exemple (code Java 7) :

```
int i = ThreadLocalRandom.current().nextInt(10, 33);
```

Le générateur associé au Thread est initialisée avec une valeur (seed) calculée par le constructeur. Celle-ci ne peut plus être modifiée : la méthode setSeed() lève une exception de type UnsupportedOperationException si elle est invoquée après la création de l'instance par le constructeur.

39. Le framework Executor

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Les threads en Java présentent de nombreux inconvénients essentiellement liés au fait que la classe Thread est de bas niveau et donc certaines fonctionnalités doivent être développées, par exemple :

- obtenir un résultat de l'exécution d'un thread
- obtenir par l'appelant une exception levée dans le thread
- aucun pool de threads n'est proposé en standard
- attendre la fin d'un ensemble de threads

Pour faciliter la mise en oeuvre de traitements en parallèle, en apportant des réponses à ces problématiques, Java 5 propose le framework Executor qui est spécifié dans la JSR 166.

Il est fortement recommandé d'utiliser le framework Executor à la place de la classe Thread.

Ce chapitre contient plusieurs sections :

- ♦ L'interface Executor
- ♦ Les pools de threads
- ♦ L'interface java.util.concurrent.Callable
- ♦ L'interface java.util.concurrent.Future
- ♦ L'interface java.util.concurrent.CompletionService

39.1. L'interface Executor

L'interface java.util.concurrent.Executor décrit les fonctionnalités permettant l'exécution différée de tâches implémentées sous la forme de Runnable.

Elle ne définit qu'une seule méthode :

Méthode	Rôle
void execute(Runnable command)	Exécuter la tâche fournie en paramètre éventuellement dans le futur

Selon l'implémentation, la tâche pourra être exécutée dans un thread dédié ou dans le thread courant.

Elle possède deux interfaces filles :

- ExecutorService : elle définie les fonctionnalités d'un service permettant l'exécution de tâches de type Runnable ou Callable.

- ScheduledExecutorService qui hérite de l'interface ExecutorService : elle définit les fonctionnalités d'un service pour l'exécution de tâches planifiées et/ou répétées

39.1.1. L'interface ExecutorService

L'interface ExecutorService décrit les fonctionnalités d'un service d'exécution de tâches. Elle hérite de l'interface Executor.

Elle définit plusieurs méthodes

Méthode	Rôle
boolean awaitTermination(long timeout, TimeUnit unit)	Attendre l'achèvement des tâches après une demande d'arrêt ou la fin d'un délai ou l'interruption du thread courant selon ce qui se produira en premier
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)	Exécuter toutes les tâches fournies en paramètres. Renvoie une collection de Future qui permet d'obtenir des informations sur l'exécution des tâches. Cette méthode est bloquante jusqu'à ce que l'exécution de toutes les tâches soit terminée
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)	Exécuter toutes les tâches fournies en paramètres. Renvoie une collection de Future qui permet d'obtenir des informations sur l'exécution des tâches. Cette méthode est bloquante jusqu'à ce que l'exécution de toutes les tâches soit terminée ou que le timeout soit atteint
<T> T invokeAny(Collection<? extends Callable<T>> tasks)	Exécuter toutes les tâches fournies en paramètres. Renvoie le résultat d'une des tâches exécutées. Cette méthode est bloquante jusqu'à ce que l'exécution de la tâche dont la valeur est retournée soit terminée
<T> T invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)	Exécuter toutes les tâches fournies en paramètre. Renvoie le résultat d'une des tâches exécutées. Cette méthode est bloquante jusqu'à ce que l'exécution de la tâche dont la valeur est retournée soit terminée ou que le timeout soit atteint
boolean isShutdown()	Renvoyer un booléen qui précise si le service est fermé
boolean isTerminated()	Renvoyer un booléen qui précise si toutes les tâches sont terminées
void shutdown()	Demander la fermeture du service. Toutes les tâches en cours d'exécution se poursuivent jusqu'à leur fin mais plus aucune nouvelle tâche ne peut être ajoutée dans le service
List<Runnable> shutdownNow()	Demander la fermeture du service. Elle tente d'arrêter le traitement des tâches en cours d'exécution. Elle renvoie une liste des tâches qui attendaient d'être exécutées
<T> Future<T> submit(Callable<T> task)	Ajouter la tâche dans la queue pour exécution. Renvoie un Future qui permet d'obtenir des informations sur l'exécution de la tâche
Future<?> submit(Runnable task)	Ajouter la tâche dans la queue pour exécution. Renvoie un Future qui permet d'obtenir des informations sur l'exécution de la tâche
<T> Future<T> submit(Runnable task, T result)	Ajouter la tâche dans la queue pour exécution. Renvoie un Future qui permet d'obtenir des informations sur l'exécution de la tâche

Il est important d'invoquer la méthode shutdown() lorsqu'il n'est plus nécessaire aux threads de l'ExecutorService d'attendre de nouvelles tâches à Exécuter. Dès lors, les tâches déjà soumises seront exécutées mais il ne sera plus possible d'en ajouter de nouvelles.

Il est nécessaire d'invoquer la méthode shutdown() pour arrêter le thread en cours d'exécution une fois que l'ExecutorService n'est plus utile.

La méthode shutdownNow() permet de demander l'interruption de l'exécution des tâches en cours et d'annuler l'exécution des tâches en attente.

Plusieurs implémentations sont fournies en standard :

- java.util.concurrent.ThreadPoolExecutor
- java.util.concurrent.ScheduledThreadPoolExecutor
- java.util.concurrent.ForkJoinPool (depuis Java 7)

Le plus simple pour créer une instance de type ExecutorService est d'utiliser la fabrique java.util.concurrent.Executors.

Il est possible de soumettre des tâches de type Runnable ou Callable pour exécution à un pool de threads en invoquant une des méthodes dédiées.

Ces méthodes renvoient un objet de type Future. Si la tâche exécutée est de type Runnable, la méthode get() renvoie toujours null.

Pour soumettre des tâches à un ExecutorService, il est préférable d'utiliser la méthode submit() plutôt que la méthode execute(). La méthode submit() renvoie un objet de type Future qui permet :

- d'obtenir la valeur de retour (ou null s'il n'y en a pas)
- d'obtenir l'exception levée par la tâche au cas où celle-ci en a levée une
- de demander l'annulation de l'exécution de la tâche (si celle-ci prend en charge cette fonctionnalité)

Si une exception est levée durant l'exécution des traitements d'une tâche soumise avec la méthode execute() alors celle-ci sera traitée par le UncaughtExceptionHandler du thread qui exécute la tâche. Par défaut, ce handler affiche l'exception et sa stacktrace dans le flux System.err.

Si une exception est levée durant l'exécution des traitements d'une tâche soumise avec la méthode submit() alors celle-ci sera chaînée à l'exception de type ExecutionException qui sera levée par l'invocation de la méthode get().

39.1.2. L'interface ScheduledExecutorService

L'utilisation de cette interface est détaillée dans la section Le ScheduledExecutorService du chapitre La planification de tâches.

39.2. Les pools de threads

Un pool d'objets est une collection d'objets initialisés qui vont être utilisables et réutilisables selon les besoins.

L'utilisation d'un pool peut améliorer les performances car il évite d'avoir à créer et initialiser un objet à chaque fois qu'une telle instance est requise. C'est particulièrement efficace pour les instances qui sont longues et/ou coûteuses à créer comme par exemple les connexions JDBC.

Un pool de threads permet de contenir un ensemble de threads qui pourront être utilisés pour exécuter des tâches. Les pools de threads sont particulièrement utiles pour exécuter des tâches similaires et indépendantes.

Les threads requièrent des ressources systèmes : il est donc nécessaire d'avoir le contrôle sur leur nombre. L'utilisation d'un pool peut permettre de facilement contrôler le nombre maximum de threads qui peuvent être exécutés en simultané. Chaque thread consomme de la ressource (CPU et mémoire) : le nombre de threads exécutables dépend donc des ressources de la machine et du système d'exploitation.

Une solution possible est d'invoquer la méthode availableProcessors() de la classe Runtime qui renvoie un entier représentant le nombre de processeurs disponibles sur la machine et d'utiliser cette valeur directement ou indirectement pour déterminer la taille du pool.

Exemple :

```
int nbProcs = Runtime.getRuntime().availableProcessors();
```

Un ExecutorService encapsule un pool de threads et une queue de tâches à exécuter. Tous les threads du pool sont toujours en cours d'exécution. Le service vérifie si une tâche est à traiter dans la queue et si c'est le cas il la retire et l'exécute. Une fois la tâche exécutée, le thread attend de nouveau que le service lui assigne une nouvelle tâche de la queue.

Plusieurs implémentations de l'interface ExecutorService sont proposées en standard dans le JDK :

- Single Thread Executor : un pool qui ne contient qu'un seul thread. Toutes les tâches soumises sont exécutées de manière séquentielle
- Cached Thread Pool : un pool qui contient plusieurs threads. Ceux-ci sont utilisés pour exécuter en parallèle les différentes tâches. La taille du pool varie selon les besoins
- Fixed Thread Pool : un pool qui contient un nombre fixe de threads. Ceux-ci sont utilisés pour exécuter en parallèle les différentes tâches. Si tous les threads sont occupés alors la tâche est empilée jusqu'à ce qu'elle puisse être exécutée par un thread
- Scheduled Thread Pool : un pool qui contient plusieurs threads pour exécuter des tâches planifiées
- Single Thread Scheduled Pool : un pool qui ne contient qu'un seul thread pour exécuter des tâches planifiées

39.2.1. La classe ThreadPoolExecutor

La classe java.util.concurrent.ThreadPoolExecutor est une implémentation de l'interface ExecutorService qui utilise un pool de threads. Elle hérite de la classe AbstractExecutorService.

La taille du pool de threads d'un ThreadPoolExecutor est configurée grâce à deux propriétés :

- corePoolSize
- maximumPoolSize

Exemple :

```
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Future;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class TestThreadPoolExecutor {

  public static void main(String[] args) {

    ExecutorService executorService = new ThreadPoolExecutor(1, 1, 1000,
        TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>());

    Future future = executorService.submit(new Runnable() {
      @Override
      public void run() {
        System.out.println("debut tache " + Thread.currentThread().getName());
        try {
          Thread.sleep(10000);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
        System.out.println("fin tache");
      }
    });

    System.out.println("Autre traitement");

    try {
      System.out.println("resultat=" + future.get());
    } catch (InterruptedException e) {
      e.printStackTrace();
    } catch (ExecutionException e) {
      e.printStackTrace();
    }

    executorService.shutdown();

    System.out.println("Fin thread principal");
  }
}
```

Lorsque l'exécution d'une tâche est confiée à un thread du pool, si le nombre de threads présent dans le pool est inférieur au maximum alors un nouveau thread est créé et ajouté au pool même si un ou plusieurs threads sont inactifs dans le pool.

Si la queue interne est pleine et que le nombre de threads est supérieur à corePoolSize et inférieur à maximumPoolSize alors un nouveau thread est créé et ajouté dans le pool pour exécuter des tâches.

Exemple :

```
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class TestThreadPoolExecutor {

  public static void main(String[] args) throws InterruptedException {

    ExecutorService executorService = new ThreadPoolExecutor(2, 4, 60,
        TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>());

    for (int i = 0; i < 5; i++) {
      executorService.submit(new Runnable() {
        @Override
        public void run() {
          System.out.println("debut tache " + Thread.currentThread().getName());
          try {
            Thread.sleep(1000);
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
          System.out.println("fin tache");
        }
      });
    }

    System.out.println("Autre traitement");

    executorService.shutdown();
    executorService.awaitTermination(300, TimeUnit.SECONDS);

    System.out.println("Fin thread principal");
  }
}
```

Résultat :

```
debut tache pool-1-thread-1
debut tache pool-1-thread-2
Autre traitement
fin tache
fin tache
debut tache pool-1-thread-1
debut tache pool-1-thread-2
fin tache
fin tache
```

```
debut tache pool-1-thread-1
fin tache
Fin thread principal
```

Dans l'exemple ci-dessous, le nombre de threads du pool ne dépasse pas 2, ce qui correspond à la propriété corePoolSize. Pourtant la propriété maximumPoolSize vaut 4 et 5 tâches sont soumises pour exécution au pool. Ceci est lié au fait que la collection de type LinkedBloquingQueue n'étant pas pleine (sa méthode offer() ne renvoie pas false), le ThreadPoolExecutor ne crée pas de nouveaux threads.

La classe ThreadPoolExecutor possède de nombreuses options de configuration. La création d'une nouvelle instance de type ThreadPoolExecutor peut être requise pour des besoins spécifiques mais le plus simple est d'utiliser la classe java.util.concurrent.Executors qui propose des fabriques. Pour faciliter la création d'instances avec une configuration classique, la classe Executors propose plusieurs méthodes qui sont des fabriques permettant de créer des instances préconfigurées.

Il est important qu'un ExecutorService soit éteint proprement lorsqu'il n'a plus d'utilité ou lorsqu'il doit être arrêté. Deux méthodes sont utilisables pour cela :

- shutdown() : une fois invoquée, le service ne peut plus prendre de nouvelles tâches, les tâches en cours d'exécution se poursuivent, les tâches non encore exécutées sont supprimées et enfin libère les ressources une fois toutes les tâches exécutées
- shutdownNow() : permet de demander un arrêt immédiat (le service ne peut plus prendre de nouvelles tâches, il essaie d'arrêter les tâches en cours d'exécution, supprime les tâches non encore exécutées et enfin libère les ressources). Elle renvoie une liste des tâches dont l'exécution ne s'est pas terminée.

Une tâche soumise à un ExecutorService sera exécutée par un thread inactif du pool.

La méthode execute(Runnable) fait exécuter la tâche fournie en paramètre par le pool de threads.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class TestExecutorService {

  public static void main(String[] args) {
    ExecutorService executorService = Executors.newSingleThreadExecutor();

    executorService.execute(new Runnable() {
      public void run() {
        System.out.println("debut tache");
        try {
          Thread.sleep(10000);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
        System.out.println("fin tache");
      }
    });

    executorService.shutdown();

    System.out.println("Fin thread principal");
  }
}
```

Résultat :

```
debut tache
Fin thread principal
fin tache
```

La méthode submit(Runnable) permet de demander l'exécution d'une tâche qui implémente l'interface Runnable. Elle renvoie un objet de type Future qui permet de déterminer si l'exécution de la tâche est terminée.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class TestExecutorService {

  public static void main(String[] args) {

    ExecutorService executorService = Executors.newSingleThreadExecutor();

    Future future = executorService.submit(new Runnable() {
      @Override
      public void run() {
        System.out.println("debut tache " + Thread.currentThread().getName());
        try {
          Thread.sleep(10000);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
        System.out.println("fin tache");
      }
    });

    System.out.println("Autre traitement");

    try {
      System.out.println("resultat=" + future.get());
    } catch (InterruptedException e) {
      e.printStackTrace();
    } catch (ExecutionException e) {
      e.printStackTrace();
    }

    executorService.shutdown();

    System.out.println("Fin thread principal");
  }
}
```

Résultat :

```
Autre traitement
debut tache pool-1-thread-1
fin tache
resultat=null
Fin thread principal
```

La méthode get() de l'instance de type Future obtenue en invoquant la méthode submit() avec un Runnable renvoie toujours null.

La méthode submit(Callable) permet de soumettre l'exécution d'un tâche de type Callable<V> et renvoie un objet de type Future<V> qui permet d'obtenir des informations sur l'exécution.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;

public class TestExecutorService {

  public static void main(String[] args) {
    ExecutorService executorService = Executors.newFixedThreadPool(3);

    Future<String> future2 = executorService.submit(new Callable<String>() {
      public String call() throws Exception {
        int i = 0;
        System.out.println("debut tache 1");
        while (i < 10 && !Thread.currentThread().isInterrupted()) {
          Thread.sleep(1000);
          i++;
        }
        System.out.println("fin tache 1");
        return "Tache 1";
      }
    });

    Future<String> future1 = executorService.submit(new Callable<String>() {
      public String call() throws Exception {
        int i = 0;
        System.out.println("debut tache 2 ");
        while (i < 10 && !Thread.currentThread().isInterrupted()) {
          Thread.sleep(500);
          i++;
        }
        System.out.println("fin tache 2");
        return "Tache 2";
      }
    });

    executorService.shutdown();

    try {
      executorService.awaitTermination(1, TimeUnit.HOURS);
      System.out.println("result1 = " + future1.get());
      System.out.println("result2 = " + future2.get());
    } catch (InterruptedException ie) {
      ie.printStackTrace();
    } catch (ExecutionException ee) {
      ee.printStackTrace();
    }
  }
}
```

Résultat :

```
debut tache 1
debut tache 2
fin tache 2
fin tache 1
result1 = Tache 2
result2 = Tache 1
```

La méthode invokeAny() renvoie le résultat de l'exécution d'une des tâches fournies en paramètres. Il n'y a pas de garantie sur la tâche dont la valeur sera retournée : c'est la première dont l'exécution se termine.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

import java.util.HashSet;
import java.util.Set;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class TestExecutorService {
  public static void main(String[] args) {
    ExecutorService executorService = Executors.newFixedThreadPool(3);

    Set<Callable<String>> callables = new HashSet<Callable<String>>();

    callables.add(new Callable<String>() {
      public String call() throws Exception {
        int i = 0;
        System.out.println("debut tache 1");
        while (i < 100 && !Thread.currentThread().isInterrupted()) {
          Thread.sleep(10000);
          i++;
        }
        System.out.println("fin tache 1");
        return "Tache 1";
      }
    });

    callables.add(new Callable<String>() {
      public String call() throws Exception {
        int i = 0;
        System.out.println("debut tache 2 ");
        while (i < 50 && !Thread.currentThread().isInterrupted()) {
          Thread.sleep(10);
          i++;
        }
        System.out.println("fin tache 2");
        return "Tache 2";
      }
    });

    callables.add(new Callable<String>() {
      public String call() throws Exception {
        int i = 0;
        System.out.println("debut tache 3 ");
        while (i < 200 && !Thread.currentThread().isInterrupted()) {
          Thread.sleep(100);
          i++;
        }
        System.out.println("fin tache 3");
        return "Tache 3";
      }
    });

    try {
      String result = executorService.invokeAny(callables);
      System.out.println("result = " + result);
    } catch (InterruptedException e) {
      e.printStackTrace();
    } catch (ExecutionException e) {
      e.printStackTrace();
    }
    executorService.shutdown();
  }
}
```

Résultat :

```
debut tache 3
debut tache 2
debut tache 1
fin tache 2
result = Tache 2
```

Dès qu'une tâche est terminée, son résultat est retourné et l'exécution des autres tâches est annulée.

La méthode invokeAny() permet d'exécuter plusieurs tâches et de renvoyer le résultat de la première qui est terminée. L'invocation de cette méthode est bloquante.

La méthode invokeAll() permet de demander l'exécution de toutes les tâches.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;

public class TestExecutorService {

  public static void main(String[] args) {
    ExecutorService executorService = Executors.newFixedThreadPool(2);

    Set<Callable<String>> callables = new HashSet<Callable<String>>();

    callables.add(new Callable<String>() {
      public String call() throws Exception {
        int i = 0;
        System.out.println("debut tache 1");
        while (i < 10 && !Thread.currentThread().isInterrupted()) {
          Thread.sleep(1000);
          i++;
        }
        System.out.println("fin tache 1");
        return "Tache 1";
      }
    });

    callables.add(new Callable<String>() {
      public String call() throws Exception {
        int i = 0;
        System.out.println("debut tache 2 ");
        while (i < 10 && !Thread.currentThread().isInterrupted()) {
          Thread.sleep(500);
          i++;
        }
        System.out.println("fin tache 2");
        return "Tache 2";
      }
    });

    try {
      List<Future<String>> futures = executorService.invokeAll(callables);

      executorService.shutdown();

      executorService.awaitTermination(1, TimeUnit.HOURS);

      for (Future<String> future : futures) {
        System.out.println("resultat = " + future.get());
      }
    } catch (InterruptedException ie) {
      ie.printStackTrace();
    } catch (ExecutionException ee) {
      ee.printStackTrace();
    }
  }
}
```

Résultat :

```
debut tache 2
debut tache 1
fin tache 2
fin tache 1
resultat = Tache 2
resultat = Tache 1
```

La méthode awaitTermination() permet d'attendre de manière bloquante la fin de l'exécution de toutes les tâches soumises.

Les threads du pool ne sont pas des démons : si la méthode shutdown() n'est pas invoquée alors la JVM continuera indéfiniment de s'exécuter même si les traitements du thread principal sont terminés.

Exemple (code Java 5.0) :

```java
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class TestExecutorService {

  public static void main(String[] args) {
    ExecutorService executorService = Executors.newSingleThreadExecutor();

    Future<String> future = executorService.submit(new Callable<String>() {
      public String call() throws Exception {
        System.out.println("debut tache");
        Thread.sleep(1000);
        System.out.println("fin tache");
        return "Tache";
      }
    });

    try {
      System.out.println("resultat = " + future.get());
    } catch (InterruptedException ie) {
      ie.printStackTrace();
    } catch (ExecutionException ee) {
      ee.printStackTrace();
    }
    System.out.println("Fin thread principal");
  }
}
```

Résultat :

```
debut tache
fin tache
resultat = Tache
Fin thread principal
```

La méthode shutdownNow() va tenter de stopper l'exécution des tâches en cours et va retirer les tâches dont l'exécution n'est pas encore démarrée. Aucune garantie n'est offerte concernant les tâches qui sont stoppées : elles peuvent être interrompues ou leur exécution peut se poursuivre jusqu'à sa fin.

39.2.2. La classe Executors

La classe java.util.concurrent.Executors est une fabrique qui permet de créer des instances de différents types du framework Executor : Executor, ExecutorService, ScheduledExecutorService, ThreadFactory et Callable.

La classe Executors permet de créer :

- des ExecutorService dont l'implémentation utilise un pool de threads : CachedThreadPool, FixedThreadPool, SingleThreadExecutor
- des ScheduledExecutorService
- des Callable

Elle possède plusieurs méthodes, notamment :

Méthode	Rôle
Callable<Object> callable(PrivilegedAction<?> action)	Renvoyer une instance de type Callable qui lors de son invocation exécutera la tâche fournie en paramètre et renverra le résultat
Callable<Object> callable(PrivilegedExceptionAction<?> action)	Renvoyer une instance de type Callable qui lors de son invocation exécutera la tâche fournie en paramètre et renverra le résultat
Callable<Object> callable(Runnable task)	Renvoyer une instance de type Callable qui lors de son invocation exécutera la tâche fournie en paramètre et renverra toujours null
<T> Callable<T> callable(Runnable task, T result)	Renvoyer une instance de type Callable qui lors de son invocation exécutera la tâche fournie et renverra le résultat fourni en paramètre
ThreadFactory defaultThreadFactory()	Renvoyer la fabrique de threads utilisée par défaut
ExecutorService newCachedThreadPool()	Renvoyer une instance d'un ExecutorService utilisant un pool de threads dont la taille peut être agrandie selon les besoins de manière non limitée. Les threads inactifs sont utilisés pour exécuter des tâches mais de nouveaux threads peuvent être créés et ajoutés dans le pool au besoin.
ExecutorService newCachedThreadPool(ThreadFactory threadFactory)	Cette surcharge de la méthode newCachedThreadPool() permet de passer en paramètre la fabrique de threads utilisée pour créer de nouveaux threads qui seront ajoutés dans le pool
ExecutorService newFixedThreadPool(int nThreads)	Renvoyer une instance de type ExecutorService qui utilise un pool de threads dont la taille est fixe. Les tâches à exécuter sont stockées dans une queue qui est dépilée au fur et à mesure de l'exécution des tâches par les threads du pool. La taille de la queue est illimitée.
ExecutorService newFixedThreadPool(int nThreads, ThreadFactory threadFactory)	Cette surcharge de la méthode newFixedThreadPool() permet de passer en paramètre la fabrique de threads qui sera utilisée pour créer de nouveaux threads ajoutés dans le pool
ScheduledExecutorService newScheduledThreadPool(int corePoolSize)	Renvoyer une instance de type ScheduledExecutorService dont la taille du pool de threads est précisée en paramètre
ScheduledExecutorService newScheduledThreadPool(int corePoolSize, ThreadFactory threadFactory)	Cette surcharge de la méthode newScheduledThreadPool() permet de passer en paramètre la fabrique de threads qui sera utilisée pour créer de nouveaux threads ajoutés dans le pool
ExecutorService newSingleThreadExecutor()	Renvoyer une instance de type ExecutorService qui utilise un pool n'ayant qu'un seul thread. Les tâches à exécuter sont stockées dans une queue qui est dépilée au fur et à mesure de l'exécution des tâches par le thread du pool. La taille de la queue est illimitée.
ExecutorService newSingleThreadExecutor(ThreadFactory threadFactory)	Cette surcharge de la méthode newSingleThreadExecutor() permet de passer en paramètre la fabrique de threads utilisée pour créer de nouveaux threads qui seront ajoutés dans le pool
ScheduledExecutorService newSingleThreadScheduledExecutor()	Renvoyer une instance de type ScheduledExecutorService dont le pool ne contient

	qu'un seul thread
ScheduledExecutorService newSingleThreadScheduledExecutor(ThreadFactory threadFactory)	Cette surcharge de la méthode newSingleThreadScheduledExecutor() permet de passer en paramètre la fabrique de threads utilisée pour créer de nouveaux threads qui seront ajoutés dans le pool
<T> Callable<T> privilegedCallable(Callable<T> callable)	Renvoyer une instance de type Callable qui lors de son invocation exécutera la tâche fournie en paramètre sous le contrôle de l'AccessController courant
<T> Callable<T> privilegedCallableUsingCurrentClassLoader(Callable<T> callable)	Renvoyer une instance de type Callable qui lors de son invocation exécutera la tâche fournie en paramètre sous le contrôle de l'AccessController courant
ThreadFactory privilegedThreadFactory()	Renvoyer la fabrique de threads utilisée pour créer de nouveau thread ayant les mêmes permissions que le thread courant

39.3. L'interface java.util.concurrent.Callable

L'interface java.lang.Runnable présente des limitations :

- la méthode run() ne permet pas de récupérer une valeur de retour suite à l'exécution des traitements
- la méthode run() ne permet pas de récupérer une exception levée durant l'exécution.

Avant Java 5, pour pallier ces limitations, il était nécessaire d'ajouter du code pour gérer ces fonctionnalités.

Java 5 propose l'interface java.util.concurrent.Callable<V>. Elle ne définit qu'une seule méthode public V call() throws Exception.

L'interface Callable est typée avec un generic qui permet de préciser le type de la valeur de retour de son unique méthode.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.Callable;

public class MonCallable implements Callable<String> {

  @Override
  public String call() throws Exception {
    try {
      Thread.sleep(5000);
    }
    catch(InterruptedException e){
      throw new Exception("Thread interrompu",e);
    }
    return "test";
  }
}
```

Si le Callable n'a pas de valeur de retour, il faut le typer avec Void.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.Callable;

public class MonCallable implements Callable<Void> {
  @Override
  public Void call() throws Exception {
    try {
      Thread.sleep(5000);
    } catch (InterruptedException e) {
      throw new Exception("Thread interrompu", e);
    }
    return null;
  }
}
```

Pour demander l'exécution asynchrone d'un Callable à un ExecutorService, il faut utiliser une des surcharges de la méthode submit(). Celle-ci renvoie une instance de type Future qui permet de vérifier le statut d'exécution d'un Callable et d'obtenir la valeur de retour à la fin de son exécution.

39.4. L'interface java.util.concurrent.Future

L'interface java.util.concurrent.Future<V> définit les fonctionnalités qui permettent de gérer le cycle de vie de l'exécution d'une tâche.

Méthode	Rôle
boolean cancel(boolean mayInterruptIfRunning)	Tenter d'interrompre l'exécution de la tâche
V get()	Attendre de manière bloquante la fin de l'exécution de la tâche et renvoyer son résultat
V get(long timeout, TimeUnit unit)	Attendre de manière bloquante la fin de l'exécution de la tâche ou la fin du timeout fourni en paramètre et renvoyer son résultat s'il est disponible
boolean isCancelled()	Renvoyer un booléen qui précise si l'exécution de la tâche est interrompue avant sa fin
boolean isDone()	Renvoyer un booléen qui précise si l'exécution de la tâche est terminée

Attention : la méthode get() est bloquante. Le thread qui l'invoque reste donc en attente de la fin de l'exécution de la tâche.

Exemple :

```
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class TestCallable {
  public static void main(String[] args) {
    Integer resultat;
    ExecutorService executor = Executors.newSingleThreadExecutor();
    Future<Integer> result = executor.submit(new Callable<Integer>() {
      @Override
      public Integer call() throws Exception {
        try {
          Thread.sleep(1000);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
        return Integer.valueOf(10);
      }
    });
    executor.shutdown();
    System.out.println("debut");
    long debut = System.currentTimeMillis();
    try {
      resultat = result.get();
      System.out.println("fin");
      System.out.printf("Resultat %d (%d ms)%n", resultat,
          System.currentTimeMillis() - debut);
    } catch (InterruptedException e) {
      e.printStackTrace();
    } catch (ExecutionException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
debut
fin
Resultat 10 (1000 ms)
```

Il n'est pas pertinent de lancer l'exécution d'une tâche de manière asynchrone pour immédiatement attendre le résultat de l'exécution. Il est donc préférable de vérifier si l'exécution de la tâche est terminée en invoquant périodiquement la méthode isDone() qui renvoie un booléen valant true si l'exécution est finie.

Exemple :

```
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class TestCallable {
  public static void main(String[] args) {
    Integer resultat;
    ExecutorService executor = Executors.newSingleThreadExecutor();
    Future<Integer> result = executor.submit(new Callable<Integer>() {
      @Override
      public Integer call() throws Exception {
        try {
          Thread.sleep(1000);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
        return Integer.valueOf(10);
      }
    });
    executor.shutdown();

    long debut = System.currentTimeMillis();
    while (!result.isDone()) {
      System.out
          .printf("attente (%d ms)%n", System.currentTimeMillis() - debut);
      try {
        Thread.sleep(200);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }

    try {
      resultat = result.get();
      System.out.printf("Resultat %d (%d ms)%n", resultat,
          System.currentTimeMillis() - debut);
    } catch (InterruptedException e) {
      e.printStackTrace();
    } catch (ExecutionException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
attente (0 ms)
attente (203 ms)
attente (406 ms)
attente (609 ms)
attente (812 ms)
Resultat 10 (1016 ms)
```

Ce n'est généralement pas une bonne pratique d'attendre, potentiellement indéfiniment, la fin de l'exécution d'une tâche car si celle-ci ne se termine jamais, c'est deux threads qui ne se terminent pas.

Exemple :

```
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public class TestCallable {
  public static void main(String[] args) {
    Integer resultat;
    ExecutorService executor = Executors.newSingleThreadExecutor();
    Future<Integer> result = executor.submit(new Callable<Integer>() {
      @Override
      public Integer call() throws Exception {
        try {
          Thread.sleep(10000);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
        return Integer.valueOf(10);
      }
    });
    executor.shutdown();

    long debut = System.currentTimeMillis();
    try {
      resultat = result.get(500L, TimeUnit.MILLISECONDS);
      System.out.printf("Resultat %d (%d ms)%n", resultat,
          System.currentTimeMillis() - debut);
    } catch (InterruptedException e) {
      e.printStackTrace();
    } catch (ExecutionException e) {
      e.printStackTrace();
    } catch (TimeoutException e) {
      System.out
          .printf("Timeout (%d ms)%n", System.currentTimeMillis() - debut);
      System.out.printf("Execution terminee %b%n", result.isDone());
    }
  }
}
```

Résultat :

```
Timeout (515 ms)
Execution terminee false
```

L'idéale est de demander l'arrêt de l'exécution de la tâche en utilisant la méthode cancel() de la classe Future.

Une tâche effectue généralement des traitements dans une boucle. La méthode cancel() invoque la méthode interrupt() du thread. Ceci positionne le flag du thread mais ne met pas fin à son exécution.

Exemple :

```
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public class TestCallable {
  public static void main(String[] args) {
    Integer resultat;
    ExecutorService executor = Executors.newSingleThreadExecutor();
    Future<Integer> result = executor.submit(new Callable<Integer>() {
      @Override
      public Integer call() throws Exception {
        int compteur = 0;
        while (compteur < 1000000000) {
          compteur++;
        }
        System.out.println("compteur=" + compteur + " "
            + Thread.currentThread().isInterrupted());
        return Integer.valueOf(compteur);
      }
    });
    executor.shutdown();

    long debut = System.currentTimeMillis();
    try {
      resultat = result.get(50L, TimeUnit.MILLISECONDS);
      System.out.printf("Resultat %d (%d ms)%n", resultat,
          System.currentTimeMillis() - debut);
    } catch (InterruptedException e) {
      e.printStackTrace();
    } catch (ExecutionException e) {
      e.printStackTrace();
    } catch (TimeoutException e) {
      System.out
          .printf("Timeout (%d ms)%n", System.currentTimeMillis() - debut);
      result.cancel(true);
      System.out.printf("Execution terminee %b%n", result.isDone());
    }
  }
}
```

Résultat :

```
Timeout (62 ms)
Execution terminee true
compteur=1000000000 true
```

Il est alors nécessaire de gérer dans la condition de la boucle, la demande d'interruption de l'exécution du thread de la tâche.

Exemple :

```
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public class TestCallable {
  public static void main(String[] args) {
    Integer resultat;
    ExecutorService executor = Executors.newSingleThreadExecutor();
    Future<Integer> result = executor.submit(new Callable<Integer>() {
      @Override
      public Integer call() throws Exception {
        int compteur = 0;
        while ((compteur < 100000000)
            && (!Thread.currentThread().isInterrupted())) {
          compteur++;
        }
        System.out.println("compteur=" + compteur + " "
            + Thread.currentThread().isInterrupted());
        return Integer.valueOf(compteur);
      }
    });
    executor.shutdown();

    long debut = System.currentTimeMillis();
    try {
      resultat = result.get(50L, TimeUnit.MILLISECONDS);
      System.out.printf("Resultat %d (%d ms)%n", resultat,
          System.currentTimeMillis() - debut);
    } catch (InterruptedException e) {
      e.printStackTrace();
    } catch (ExecutionException e) {
      e.printStackTrace();
    } catch (TimeoutException e) {
      System.out
          .printf("Timeout (%d ms)%n", System.currentTimeMillis() - debut);
      result.cancel(true);
      System.out.printf("Execution terminee %b%n", result.isDone());
    }
  }
}
```

Résultat :

```
Timeout (62 ms)
Execution terminee true
compteur=1877256 true
```

39.5. L'interface java.util.concurrent.CompletionService

La gestion de la récupération des résultats de plusieurs tâches exécutées de manière asynchrone par un ExecutorService nécessite d'écrire un peu de code.

Exemple :

```
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.Callable;

public class MaTache implements Callable<Integer> {

  private final int duree;

  public MaTache(int duree) {
    this.duree = duree;
  }

  @Override
  public Integer call() {
    System.out.println("Debut tache " + Thread.currentThread().getName());
    try {
      Thread.sleep(1000 * duree);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.println("Fin tache " + Thread.currentThread().getName());
    return duree;
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.thread;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class TestCompletionService {
  private static final int NB_TACHES = 5;

  public static void main(String[] args) {
    ExecutorService executor = Executors.newFixedThreadPool(NB_TACHES);
    List<Future<Integer>> futures =
      new ArrayList<Future<Integer>>(NB_TACHES);

    for (int i = 0; i < NB_TACHES; i++) {
      futures.add(executor.submit(new MaTache(NB_TACHES - i)));
    }

    for (Future<Integer> future : futures) {
      Integer resultat;
      try {
        resultat = future.get();
        System.out.println("resultat = " + resultat);
      } catch (InterruptedException e) {
        e.printStackTrace();
      } catch (ExecutionException e) {
        e.printStackTrace();
      }
    }
    executor.shutdown();
  }
}
```

Résultat :

```
Debut tache
pool-1-thread-1
Debut tache
pool-1-thread-2
Debut tache
pool-1-thread-3
Debut tache
pool-1-thread-4
Debut tache
pool-1-thread-5
Fin tache
pool-1-thread-5
Fin tache
pool-1-thread-4
Fin tache
pool-1-thread-3
Fin tache
pool-1-thread-2
Fin tache
pool-1-thread-1
resultat = 5
resultat = 4
resultat = 3
resultat = 2
resultat = 1
```

L'inconvénient de cette solution est qu'elle récupère les résultats dans l'ordre dans lequel les tâches sont soumises. Ainsi, si le temps d'exécution de la première tâche est le plus long, il faudra attendre sa fin avant de commencer à obtenir un résultat même si toutes les autres tâches sont terminées.

Pour faciliter la gestion de l'attente et de l'obtention des résultats de plusieurs tâches exécutées par un ExecutorService, l'API propose l'interface CompletionService.

Une instance de type CompletionService permet l'exécution de tâches asynchrones et surtout facilite la récupération de leurs résultats au fur et à mesure de l'achèvement de ces tâches.

Elle définit plusieurs méthodes :

Méthode	Rôle
Future<V> poll()	Obtenir l'instance de type Future de la prochaine tâche qui se terminera. Elle renvoie null si aucune tâche ne s'est terminée
Future<V> poll(long timeout, TimeUnit unit)	Obtenir l'instance de type Future de la prochaine tâche qui se terminera. Elle renvoie null si aucune tâche ne s'est terminée avant l'expiration du timeout fourni en paramètre
Future<V> submit(Callable<V> task)	Demander l'exécution de la tâche fournie en paramètre
Future<V> submit(Runnable task, V result)	Demander l'exécution de la tâche fournie en paramètre
Future<V> take()	Obtenir l'instance de type Future de la prochaine tâche qui se terminera. Cette méthode est bloquante jusqu'à ce qu'une tâche soit terminée.

La classe ExecutorCompletionService implémente l'interface CompletionService.

Elle sépare les activités de soumission de tâches et de récupérations de leurs résultats en utilisant deux files. Les méthodes poll() et take() retirent de la file des résultats les objets de type Future qui sont retournés.

La méthode bloquante take() permet de renvoyer le résultat d'une exécution qui n'a pas encore été consommé.

Exemple :

```
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.CompletionService;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorCompletionService;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class TestCompletionService {
  private static final int NB_TACHES = 5;
  public static void main(String[] args) {
    ExecutorService executor = Executors.newFixedThreadPool(NB_TACHES);
    CompletionService<Integer> completion = new ExecutorCompletionService<Integer>(
        executor);
    for (int i = 0; i < NB_TACHES; i++) {
      completion.submit(new MaTache(NB_TACHES - i));
    }
    for (int i = 0; i < NB_TACHES; i++) {
      Integer resultat;
      try {
        resultat = completion.take().get();
        System.out.println("resultat = " + resultat);
      } catch (InterruptedException e) {
        e.printStackTrace();
      } catch (ExecutionException e) {
        e.printStackTrace();
      }
    }
    executor.shutdown();
  }
}
```

Résultat :

```
Debut tache
pool-1-thread-1
Debut tache
pool-1-thread-2
Debut tache
pool-1-thread-3
Debut tache
pool-1-thread-4
Debut tache
pool-1-thread-5
Fin tache
pool-1-thread-5
resultat = 1
Fin tache
pool-1-thread-4
resultat = 2
Fin tache
pool-1-thread-3
resultat = 3
Fin tache
pool-1-thread-2
resultat = 4
Fin tache
pool-1-thread-1
resultat = 5
```

Il n'est plus utile de gérer une collection de type Future.

Les invocations successives de la méthode take() renvoient les instances de type Future au fur et à mesure de la fin de l'exécution des tâches. Les résultats sont ainsi obtenus non pas dans leur ordre de lancement mais dans l'ordre dans lequel les tâches se terminent.

Attention : comme la méthode take() est bloquante, il ne faut surtout pas l'invoquer un nombre de fois supérieur à celui des tâches soumises car dans ce cas, le thread restera bloqué indéfiniment.

40. La gestion des accès concurrents

Chapitre 40

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Les différents threads d'une application peuvent accéder à toutes les données pour lesquelles ils possèdent une référence. Cela inclut des données partagées par ces différents threads : les différents threads peuvent lire et modifier ces données en parallèle et potentiellement en concurrence notamment sur des machines multicoeurs. Ces mises à jour peuvent laisser les données dans un état incohérent si celles-ci ne sont pas atomiques.

Lors de la parallélisation de traitements, il est fréquent de devoir gérer des accès concurrents à certaines données. Les deux points principaux à prendre en compte lors d'accès concurrents sont :

- la visibilité : à quel moment une modification par un thread est visible par les autres ? C'est d'autant plus important que les opérations sont généralement faites sur des copies des valeurs (caches et registres CPU par exemple) avant d'être recopiées en mémoire
- la cohérence : comment garantir que les données ne sont pas corrompues une fois les opérations concurrentes terminées ? Pour garantir la cohérence, une opération doit être atomique

L'accès concurrent pour la mise jour d'une même donnée par plusieurs threads est désigné par race condition. Pour empêcher les accès concurrents à une même donnée, il est possible d'utiliser un mécanisme de verrouillage qui va bloquer l'accès à une donnée le temps qu'un autre thread accède déjà à la donnée. Ainsi si plusieurs threads tentent d'accéder à une même donnée en même temps, ces accès se feront les uns après les autres.

Java propose deux mécanismes pour poser des verrous :

- les moniteurs qui s'utilisent avec le mot clé synchronised
- les verrous qui sont définis dans l'interface java.util.concurrent.locks.Lock

Chaque objet possède un moniteur (monitor). Il n'est pas possible d'utiliser directement le moniteur d'un objet : la façon de l'utiliser est de passer l'objet en paramètre de l'instruction synchronized() explicitement ou de l'utiliser implicitement quand il s'agit de synchroniser l'accès à l'objet courant.

L'utilisation du mot clé synchronized garantit l'atomicité du bloc de code mais aussi la visibilité des modifications effectuées dans ce bloc. S'il est utilisé, un bloc synchronized doit être mis en oeuvre pour la modification et la lecture d'une donnée partagée par plusieurs threads.

Si une classe possède plusieurs méthodes définies avec le mot clé synchronized, alors une seule de ces méthodes pourra être exécutée en même temps par plusieurs threads, même si chaque thread exécute une méthode différente.

Il est possible d'imbriquer dans le même bloc de code plusieurs instructions synchronized sur des moniteurs différents. Attention cependant, l'ajout de verrous pour gérer des cas de gestion d'accès concurrents peut augmenter, sous certaines circonstances, le risque d'introduire des situations de deadlocks.

Plusieurs règles devraient être appliquées lors de la mise en oeuvre de verrous :

- un objet immuable est thread-safe : il est donc inutile d'utiliser un verrou
- le verrou doit être maintenu pour une durée la plus courte possible
- il ne faut utiliser le mot clé synchronized que sur les méthodes qui le requiert pour ne pas ajouter de la contention inutile

Une difficulté supplémentaire relative à l'utilisation de plusieurs threads concerne la visibilité par les autres threads des modifications faites par un thread.

Le code Java n'est pas exécuté directement : il est transformé par le compilateur en un langage intermédiaire : le bytecode. Ce bytecode est lui-même interprété, voire compilé en code natif par le compilateur JIT de la JVM. La JVM effectue des optimisations avancées qui peuvent réordonner les opérations des traitements, stocker des données en cache, ... Ceci implique qu'il n'y par défaut aucune garantie sur la visibilité des modifications.

Le mot clé volatile peut être utilisé pour garantir cette visibilité : il ne garantit cependant rien concernant l'atomicité. Le mot clé volatile peut être utilisé sur des champs dont les opérations sont atomiques : par exemple un champ de type booléen qui est lu et simplement modifié sans tenir compte de sa valeur courante.

Ce chapitre contient plusieurs sections :

- Le mot clé volatile
- Les races conditions
- La synchronisation avec les verrous
- Les opérations atomiques
- L'immutabilité et la copie défensive

40.1. Le mot clé volatile

L'utilisation du mot clé volatile force l'écriture de la valeur d'une variable en mémoire ainsi que sa relecture : cela permet de garantir que la lecture de la donnée par un thread retournera la valeur la plus récente en mémoire. Le mot clé volatile ne réalise aucune opération pour garantir la gestion des accès concurrents : elle offre juste une garantie sur la visibilité.

Le mot clé volatile s'utilise sur la déclaration d'une variable. Il n'est pas possible de l'utiliser sur une méthode ou une classe.

Les processeurs multicoeurs utilisent massivement des caches de plusieurs niveaux pour améliorer leur performance.

Dans une application multithread, les threads qui utilisent des variables non volatiles peuvent les copier de la mémoire centrale dans un cache CPU lorsqu'ils les utilisent pour améliorer les performances. Sur des machines multicoeurs, les threads peuvent s'exécuter sur différents coeurs : dans ce cas, plusieurs copies de la variable peuvent exister dans les caches de ces coeurs.

Si la donnée est partagée par plusieurs threads, sa valeur peut être dupliquée dans différents caches des processeurs qui exécutent les threads et y être modifiée. Il y a donc un risque que les valeurs utilisées par les différents threads soient différentes à un instant donné.

Si la variable n'est pas volatile, il n'y a aucune garantie sur les moments où la valeur sera lue de la mémoire pour être mise en cache ou encore lue du cache pour être écrite en mémoire.

La déclaration d'une variable avec le mot clé volatile permet de garantir que chaque lecture se fera de la mémoire centrale et que chaque écriture se fera dans celle-ci. L'intérêt est donc de permettre à chaque thread d'obtenir la valeur la plus fraîche possible.

Attention : le mot clé volatile ne garantit pas la gestion des accès concurrents mais uniquement une meilleure fraicheur de la valeur de la variable. Cependant, dans certains cas, son utilisation peut permettre de s'assurer que la lecture de sa valeur par plusieurs threads est la plus fraîche possible.

Si un seul thread modifie la valeur d'une variable volatile alors les autres threads ont la garantie que lors de leurs lectures, c'est la dernière valeur écrite qui sera lue. Si la variable n'est pas volatile, cette garantie n'est pas assurée.

Si plusieurs threads peuvent lire et écrire une variable partagée alors l'utilisation du mot clé volatile n'est pas suffisante, par exemple :

- la valeur de la variable volatile est 0
- le thread1 lit la variable et met sa valeur dans le cache du CPU1
- le thread2 lit la variable et met sa valeur dans le cache du CPU2

- le thread1 incrémente la valeur dans le cache et écrit la valeur en mémoire
- le thread2 incrémente la valeur dans le cache et écrit la valeur en mémoire

Au final, bien que chaque thread ait incrémenté la valeur, celle en mémoire est 1 alors qu'elle devrait être 2. Il faut impérativement dans ce cas utiliser un mécanisme de synchronisation pour s'assurer que ces opérations soient atomiques.

Depuis Java 5 qui applique le nouveau modèle de gestion de la mémoire, le mot clé volatile offre des garanties supplémentaires en plus de la lecture/écriture de/vers la mémoire :

- si un thread modifie une variable volatile alors toutes les autres variables visibles du thread seront aussi visibles par les autres threads lorsqu'ils effectueront une lecture sur la variable volatile en application de la notion d'happens before
- les opérations de lecture et d'écriture d'une variable volatile ne peuvent pas être réordonnées par la JVM pour optimiser l'exécution et améliorer les performances

Voici un exemple pour illustrer un cas : l'instance monObjet est partagée par plusieurs threads. Sa classe contient deux variables :

- varVolatile de type booléen, déclarée volatile, initialisée à false
- varNonVolatile de type int, initialisée à 0

Deux threads effectuent des opérations sur cette instance :

Thread-1	Thread-2

monObjet.varNonVolatile = 123;
monObjet.varVolatile = true;	

	if (monObjet.varVolatile) {
	int valeur = monObjet.varNonVolatile ;
	}

Il est tout à fait possible que cela fonctionne dans une majorité de cas sans le mot clé volatile mais avec le mot clé volatile, ce comportement est systématiquement garanti.

En application de la relation happens-before :

- dans le thread-1, comme la variable varNonVolatile est modifiée avant la variable volatile varVolatile alors l'écriture de la valeur des deux variables est faite en mémoire
- dans le thread-2, comme la variable volatile varVolatile est lue de la mémoire alors la variable varNonVolatile est lue aussi de la mémoire

Ceci permet de garantir que les modifications faites par Thread-1 sont vues par Thread-2.

Cette relation nommée happens-before définie dans le JMM garantit que l'état des variables modifiées par Thread-1 avant l'écriture de la variable volatile sera vu par le Thread-2 dès que celui-ci aura lu la variable volatile. Cette visibilité s'applique aussi aux modifications faites par d'autres threads avant leur écriture de la variable varVolatile.

Si la variable varVolatile n'est pas déclarée volatile, alors il n'y aucun moyen sûr pour Thread-2 de voir les modifications faites par Thread-1 notamment celle concernant la variable non volatile.

L'utilisation du mot clé volatile n'est pas sans conséquence sur les performances : les lectures/écritures dans la mémoire sont moins rapides que dans les caches CPU. De plus, le mot clé volatile force la mise en oeuvre de ces opérations de lectures/écritures dans la mémoire et peut inhiber certaines optimisations. Il ne faut donc pas abuser du mot clé volatile mais l'utiliser à bon escient pour renforcer la visibilité d'une variable.

Il existe plusieurs cas où l'utilisation du mot clé volatile est fortement recommandée voire obligatoire pour s'éviter des ennuis notamment lors de :

- la déclaration d'un booléen qui permet de sortir de la boucle d'exécution d'un thread

Exemple (code Java 6) :

```
package fr.jmdoudoux.dej.thread;

public class MonThread extends Thread {

  private volatile boolean running = true;

  public void arreter() {
    this.running = false;
  }

  @Override
  public void run() {
    while (running) {
      // traitements du thread
      try {
        Thread.sleep(500);
      } catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

- la déclaration d'une variable de type long ou double partagée

Comme précisé dans le chapitre « threads and locks » de la JLS (Java Language Specification), l'écriture de la valeur d'une variable de type double ou long n'est pas atomique. Ces valeurs sont stockées sur 64 bits et requièrent deux opérations d'écriture pour les deux moitiés de 32 bits. Il est possible qu'un thread fasse une lecture entre les deux opérations qui ne sont pas atomiques et obtiennent potentiellement une valeur erronée.

Les lectures et les écritures de variables de type long et double déclarées volatile sont toujours atomiques. Il est donc fortement recommandé de déclarer volatile les variables partagées de type double et long.

Remarque : les lectures et les écritures de références, qu'elles soient implémentées sur 32 ou 64 bits sont toujours atomiques.

Lors de l'utilisation du mot clé volatile sur un tableau, cela définit une référence volatile sur un tableau et non pas une référence sur un tableau de variables volatile. Si le mot clé volatile est utilisé sur un tableau, c'est la lecture et la modification de la référence à ce tableau qui est volatile. Lors de la lecture d'un élément du tableau, la lecture de la référence du tableau est volatile mais pas la lecture de la valeur de l'élément concerné. Lors de la modification d'un élément du tableau, la lecture de la référence du tableau est volatile mais pas la modification de la valeur de l'élément. Il n'est pas possible de déclarer volatile les éléments d'un tableau. Ainsi, la modification de la valeur d'un élément du tableau n'a pas les garanties offertes par le mot clé volatile.

Le langage ne permet pas d'avoir les garanties offertes par le mot clé volatile sur les éléments d'un tableau. Pour obtenir ces garanties, il faut utiliser les classes AtomicIntegerArray, AtomicLongArray et AtomicReferenceArray du package java.util.concurrent. Elles offrent une sémantique similaire à volatile lors des opérations de lecture/écriture sur les éléments du tableau.

Comme le mot clé volatile n'implique pas l'utilisation de verrous, sa mise en oeuvre est généralement plus performante. Par contre, si un champ volatile est très fréquemment utilisé dans une méthode, les performances peuvent être moins bonnes que d'utiliser un verrou.

L'utilisation d'un champ volatile se justifie pleinement lorsqu'il est admis qu'un seul thread peut mettre à jour le champ pendant que d'autres peuvent le lire.

40.2. Les races conditions

Dans une application multithread, un des problèmes les plus courants rencontrés est la race condition.

Une race condition est une situation où au moins deux threads exécutent la même portion de code sans qu'aucune mesure de synchronisation de ces accès ne soit faite. Ceci peut engendrer des comportements inattendus liés à l'exécution de traitements en concurrence par plusieurs threads :

- tenter de modifier une donnée partagée de manière concurrente
- obtenir des résultats incohérents

Une race condition est une situation risquée dans laquelle plusieurs threads effectuent des lectures et des écritures sur une donnée partagée. L'ordre imprévisible des ces opérations peut induire ces comportements inattendus et aléatoires.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class MaClasseAvecRaceCondition {
  private volatile boolean flag = true;

  public void executer() {
    flag = true;
    if (flag != true) {
      System.out.println("arffff");
    }
    flag = false;
  }
}
```

L'invocation de la méthode executer() par un seul thread ne permettra jamais de voir afficher le message dans la console.

Par contre, si la méthode executer() d'une même instance partagée est invoquée par plusieurs il y a un risque selon l'ordonnancement des opérations de voir le message s'afficher. C'est notamment le cas si un thread modifie la valeur du flag à false entre son affectation à true par le thread courant et le test de sa valeur.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class MaClasseAvecRaceCondition {
  private volatile boolean flag = true;

  public void executer() {
    flag = true;
    if (flag != true) {
      System.out.println("arffff");
    }
    flag = false;
  }

  public static void main(String[] args) {
    final MaClasseAvecRaceCondition maClasse = new MaClasseAvecRaceCondition();

    for (int i = 0; i < 10; i++) {
      Thread thread = new Thread(new Runnable() {

        @Override
        public void run() {
          for (int i = 0; i < 10000; i++) {
            maClasse.executer();
          }
        }
      });
      thread.setName("monThread-" + i);
      thread.start();
    }
  }
}
```

A l'exécution de cet exemple, le message est affiché un nombre aléatoire de fois.

Pour garantir la bonne exécution, il est nécessaire de garantir l'ordre des opérations par exemple en garantissant que les opérations de la méthode executer() soient atomiques.

Un autre exemple classique de race condition est l'incrémentation d'un compteur par plusieurs threads : comme l'opérateur d'incrémentation n'est pas atomique et nécessite plusieurs opérations, il est possible que ces opérations exécutées par plusieurs threads s'intercroisent. Pour simplifier les exemples ci-dessous, l'opérateur ++ sera composé de trois opérations :

- lecture de la valeur en mémoire
- incrémentation de la valeur
- mise à jour de la valeur en mémoire

Imaginons que deux threads tentent d'incrémenter le compteur de manière concomitante.

Sans mesure particulière, l'ordre d'exécution des opérations n'est pas garanti, par exemple :

Thread 1	Thread2	Valeur dans le registre 1	Valeur dans le registre 2	Valeur en mémoire
				1
Lecture		1	1	1
	Lecture	1	1	1
Incrémentation		2	2	1
Incrémentation	Incrémentation	2	2	1
Mise à jour				2
	Mise à jour			2

D'autant moins s'il y a plusieurs coeurs ou plusieurs processeurs sur la machine d'exécution.

Thread 1	Thread2	Valeur dans le registre 1	Valeur dans le registre 2	Valeur en mémoire
				1
Lecture	Lecture	1	1	1
Incrémentation	Incrémentation	2	2	1
Mise à jour	Mise à jour			2

Dans ce cas, il y a un risque car avec une valeur initiale de 1, l'incrémentation par deux threads devrait permettre d'avoir la valeur 3.

Pour permettre d'avoir le résultat attendu, il est nécessaire que les opérations liées à l'incrémentation soient atomiques : toutes les opérations doivent s'exécuter de manière atomique.

Thread 1	Thread2	Valeur dans le registre 1	Valeur dans le registre 2	Valeur en mémoire
				1
Lecture		1		1
Incrémentation		2		1
Mise à jour				2
	Lecture	2		2
	Incrémentation	3		2
	Mise à jour			3

Remarque : Une opération atomique ne peut pas entraîner de situation de race condition.

40.2.1. La détection des race conditions

Généralement, il ne faut pas tirer de conclusion relative à la gestion des accès concurrents basée sur des expérimentations. Cela est dû au fait qu'il y a de nombreux points, pour certains aléatoires ou variables, qui interviennent dans l'apparition d'un problème d'accès concurrent : ordonnancement des instructions exécutées, version de la JVM, plateforme utilisée, ... Le fait de ne pas voir de problèmes lors des expérimentations ne prouve pas que ceux-ci ne puissent pas survenir.

La découverte de situations de race conditions grâce à des tests unitaires n'est donc pas garantie à cause de la nature aléatoire du déclenchement de cette situation. La meilleure façon de découvrir des cas potentiels de race condition est de faire de la revue de code. La découverte de race condition n'est pas triviale car elle requiert d'avoir une bonne connaissance des mécanismes sous-jacents de la programmation concurrente.

La première approche pour détecter des race conditions est de faire de la relecture de code : ce n'est cependant pas facile car il n'est pas naturel de raisonner de manière concurrente. Il est aussi nécessaire d'occulter certaines présomptions. Par exemple, une ligne de code n'est pas forcément atomique : un bon exemple est l'opérateur d'incrémentation ++.

Plusieurs patterns peuvent être à l'origine d'une race condition :

- Check and act :

Exemple : la méthode getInstance() d'une classe de type Singleton dont les traitements vérifient si l'instance est null pour déterminer s'il faut en créer une nouvelle instance.

Exemple :

```
public MonSingleton getInstance(){
  if( instance == null){   // race condition si deux
    // threads testent cette condition en même temps
    instance = new MonSingleton();
  }
}
```

Le but de cette méthode est de toujours renvoyer la même instance. Or sans précaution particulière, si deux threads invoquent la méthode en même temps, il est possible que chacun obtiennent une nouvelle instance après avoir vérifié que l'instance n'existe pas encore.

Cette situation peut aussi survenir lors de la combinaison de deux opérations atomiques. Le fait qu'elles soient chacune atomique n'empêche pas que leur combinaison ne l'est pas.

Exemple :

```
if(!monHashMap.contains(key)){  // race condition
  monHashMap.put(key, value);
}
```

Deux threads peuvent exécuter la méthode contains() en même temps et obtenir false tous les deux impliquant que tous les deux vont exécuter la méthode put().

- Read modify write : ce pattern implique que les opérations de lecture/modification/écriture ne sont pas atomiques.

Exemple : un compteur qui utilise l'opérateur d'incrémentation. Cela fonctionne parfaitement en monothread mais cela ne fonctionne pas en multithread car l'opérateur d'incrémentation n'est pas une opération atomique.

Cependant cela n'est pas toujours suffisant car, la JVM elle-même peut se permettre à des fins d'optimisations de réordonner des instructions. Pour limiter les effets de bord, les blocs de code synchronized sont exclus de ce type d'optimisation.

Différents outils open source ou commerciaux peuvent aider à détecter des cas de race condition dans le code en réalisant une analyse statique ou dynamique.

40.3. La synchronisation avec les verrous

L'exécution simultanée par plusieurs threads d'une même portion de code peut engendrer des problèmes d'accès concurrents sur certains objets. La synchronisation est un mécanisme qui permet de limiter l'exécution d'une portion de code à un seul thread.

Pour gérer les cas de race condition, il faut utiliser des mécanismes de verrouillage qui vont restreindre l'exécution d'une portion de code critique à un seul thread. Ceci permet d'inhiber les accès concurrents qui sont réalisés dans cette portion de code.

La synchronisation n'est nécessaire que pour des données mutables.

Java propose deux types de verrous :

- les moniteurs : c'est un mécanisme implicite dont la mise en oeuvre est intégrée au langage et à la JVM
- les Locks : c'est un mécanisme explicite dont la mise en oeuvre utilise des classes de l'API du JDK

40.3.1. Les verrous avec des moniteurs

La synchronisation permet de protéger l'exécution de portions de code critiques lorsque celle-ci est faite par plusieurs threads. Sans synchronisation, les accès concurrents à des données partagées par plusieurs threads engendrent des inconsistances sur ces données.

Pour mettre en oeuvre cette synchronisation, il faut un thread et un moniteur. En Java, le thread est toujours le thread courant. Le moniteur est précisé en utilisant l'instance d'un objet concerné. Un moniteur permet la mise en oeuvre de verrous implicites.

Chaque objet Java possède un moniteur qui permet de réaliser des opérations basiques de synchronisation d'accès entre threads : il n'est pas possible d'accéder directement à un moniteur.

La synchronisation en Java garantit que deux threads ne peuvent exécuter en parallèle une même portion de code synchronisée qui requiert le même verrou.

La synchronisation en Java se fait obligatoirement sur une portion de code :

- une méthode statique
- une méthode non statique
- un bloc de code

40.3.1.0.1. Le mot clé synchronized

Le mot clé synchronized permet de poser un verrou exclusif sur une portion de code : ceci permet de garantir que les accès à une ressource partagée ne se feront pas en concurrence.

La JVM garantit qu'un bloc de code déclaré synchronized ne sera exécuté que par un seul thread à un instant T sous réserve que le moniteur utilisé pour le verrou soit le même pour tous les threads.

L'utilisation du mot clé synchronized implique l'obtention d'un verrou avant l'exécution de la première instruction du bloc de code et sa libération une fois l'exécution du bloc de code terminée.

Lors de l'exécution d'une portion de code déclarée synchronized, le thread courant pose un verrou sur le moniteur d'un objet. Tant que le verrou est posé par un thread, les autres threads qui souhaitent exécuter la portion de code doivent attendre de pouvoir obtenir le verrou. A la fin de l'exécution de la portion de code, le verrou est libéré et peut ainsi être posé par un autre thread.

L'instruction synchronized permet de demander l'obtention exclusive du moniteur pour le thread courant. Les autres threads qui tenteront d'acquérir le moniteur devront attendre leur tour tant qu'un thread le possède déjà. Le thread conserve le moniteur jusqu'à la fin de l'exécution du bloc de code. Ce bloc de code est défini par la façon dont le mot clé synchronized est utilisé.

Le mot clé synchronized peut s'utiliser sur une méthode ou un bloc de code qu'ils soient static ou non. Le mot clé synchronized peut attendre en paramètre un objet dont le moniteur sera utilisé pour verrouiller le bloc de code concerné.

Lors de la compilation, le compilateur va utiliser les instructions du bytecode MonitorEnter et MonitorExit lors de la traduction de l'utilisation du mot clé synchronized. Le compilateur assure qu'une instruction MonitorExit sera toujours exécutée à la suite d'un MonitorEnter quel que soit le chemin des traitements (exécution jusqu'à la fin du bloc de code ou sortie prématurée en cas d'exception).

L'utilisation du mot clé synchronized peut aussi permettre d'éviter des bugs subtils liés aux possibilités du compilateur ou de la JVM d'effectuer des optimisations dans la génération et l'exécution du bytecode tels que changer l'ordonnancement des instructions ou utiliser un cache par exemple. Les blocs de code synchronized sont exclus de ces optimisations : l'ordre des traitements d'une portion de code déclarée synchronized ne sera pas modifié par le compilateur.

40.3.1.1. Les moniteurs

Le mot clé synchronized permet de garantir un accès exclusif à une portion de code pour un seul thread : le verrou est posé sur le moniteur d'une instance d'un objet. L'instance utilisé dépend de la façon dont le mot clé synchronized est utilisé dans le code.

Le verrou sur le moniteur peut se faire de deux manières :

- une instance : cela permet de poser un verrou sur l'instance concernée
- l'instance de type Class d'une classe : cela permet de poser un verrou quel que soit le nombre d'instances créés de la classe. Cette solution est à utiliser pour protéger des accès à des données static.

Le choix de l'un ou de l'autre dépend des besoins et de la façon dont le mot clé synchronized est utilisé.

Pour synchroniser toute une méthode, il suffit de définir la méthode avec le modificateur synchronized. Dans ce cas, le moniteur utilisé est l'instance courante (this).

Exemple :

```
public synchronized void maMethode() {
  // ...
}
```

Dans ce cas, le moniteur utilisé est l'instance de la classe elle-même. Le moniteur est acquis durant toute l'exécution de la méthode.

Pour synchroniser une méthode statique, il suffit de définir la méthode avec le modificateur synchronized. Dans ce cas, le moniteur utilisé est l'objet de type Class de la classe de la méthode.

Exemple :

```
public static synchronized void maMethode() {
  // ...
}
```

Attention : il est tout à fait possible qu'une méthode statique et une méthode non statique toutes les deux déclarées synchronized soient exécutées en même temps par deux threads différents puisque ce sont deux moniteurs différents qui sont utilisés.

Il est possible d'utiliser le mot clé synchronized sur un bloc de code. Dans ce cas, il faut préciser l'instance dont le moniteur sera utilisé. N'importe quel objet peut être utilisé comme moniteur. Il est possible d'utiliser l'instance courante.

Exemple :

```
  public void maMethode() {
    synchronized(this) {
      // ...
    }
  }
```

Pour une bonne séparation des rôles, il est préférable de déclarer un objet, généralement du type Object, avec les modificateurs private et final et de lui donner un nom de variable qui décrive son rôle, par exemple en le suffixant avec Monitor.

Exemple :

```
private final Object monMonitor = new Object();

  // ...

  public void maMethode() {
    synchronized(monMonitor) {
    // ...
  }
}
```

Il est possible d'utiliser la classe de type Class pour protéger des accès à des ressources static par exemple.

Exemple :

```
  public void maMethode() {
    synchronized(MaClasse.class) {
      // ...
    }
  }
```

Il est possible de préciser n'importe quelle instance comme moniteur à utiliser en paramètre du mot clé synchronized. Cela inclut aussi une variable locale ce qui est une très mauvaise idée car le moniteur utilisé doit être accessible par tous les threads qui tenteront d'exécuter la portion de code. A chaque invocation, une nouvelle instance sera créée et donc le moniteur utilisé sera différent pour chaque thread ce qui ne permettra pas de limiter l'exécution à un seul thread.

A l'entrée du bloc de code synchronized, le thread obtient le verrou sur le moniteur correspondant. Le verrou est libéré à la sortie de l'exécution du bloc de code. Cette libération est garantie par le compilateur et la JVM que la sortie intervienne à la fin de l'exécution du bloc de code ou qu'une exception soit levée durant son exécution.

Le mécanisme utilisant le mot clé synchronized est par nature réentrant : si une portion de code synchronized est exécutée et qu'elle requière l'exécution d'une autre portion de code avec le même moniteur alors le thread courant n'a pas besoin d'acquérir de nouveau le verrou puisqu'il le possède déjà.

Exemple :

```
public class MaClasse {

  public synchronized methodeA(){
    methodeB();
  }

  public synchronized methodeB(){
    // traitements
  }
}
```

Comme les deux méthodes utilisent le même moniteur, celui de l'instance courante, le thread peut sans problème invoquer la méthode methodeA() qui elle-même va invoquer la méthode methodeB(). Le verrou n'a pas besoin d'être obtenu de nouveau puisque le thread le possède déjà.

40.3.1.2. Les contraintes d'utilisation

Il n'est pas possible d'utiliser le mot clé synchronized sur des variables : une telle action provoque une erreur à la compilation.

Il n'est pas possible d'utiliser le mot clé synchronized sur un constructeur : le compilateur lève une erreur puisque l'objet en cours de création n'est pas encore accessible par les autres threads.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestSynchronized {

  public synchronized TestSynchronized() {
  }
}
```

Résultat :

```
C:\java\TestThreads\src>javac com/jmdoudoux/test/thread/TestSynchronized.java
com\jmdoudoux\test\thread\TestSynchronized.java:5: modifier synchronized not allowed here
  public synchronized TestSynchronized() {
                      ^
1 error
```

Si l'objet utilisé comme moniteur est null alors une exception de type NullPointerException est levée.

Attention : le mot clé synchronized ne peut être utilisé que pour limiter les accès à des ressources de la JVM dans laquelle le code s'exécute. Pour verrouiller les accès dans plusieurs JVM, il faut développer sa propre solution en utilisant une ressource commune comme un fichier ou une base de données ou utiliser une solution tierce comme Terracota.

40.3.1.3. Des recommandations d'utilisation

L'utilisation du mot clé synchronized sur plusieurs méthodes d'une même classe peut dégrader les performances. Ce mécanisme est lent et coûteux. Il est préférable d'utiliser le mot clé synchronized sur les portions de code critiques plutôt que sur toutes les méthodes. Ceci aura pour effet de limiter le temps ou le verrou est posé.

Il est donc généralement préférable d'effectuer la synchronisation sur un bloc de code plutôt que sur toute une méthode pour limiter la durée du verrou à la portion de code qui le requiert.

Il ne faut pas utiliser une instance qui ne soit pas déclarée final comme moniteur pour un bloc de code synchronized car il se pourrait que la référence à cet objet soit modifiée et qu'ainsi deux threads puissent exécuter la portion de code en concurrence puisque le verrou ne serait pas posé sur le même moniteur.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestSynchronized {

  // mauvaise pratique : le champ devrait etre declare final
  private Object verrou = new Object();

  public void maMethode() {
    synchronized (verrou) {
      // ...
    }
  }
}
```

Il n'est pas recommandé d'utiliser un objet de type String comme instance en paramètre du mot clé synchronized() car les objets de type String sont gérés en interne de la JVM dans un pool. Ainsi si ailleurs dans le code de l'application ou d'une bibliothèque tierce, la même chaîne est utilisée comme moniteur, alors le même moniteur sera utilisé dans des classes différentes alors qu'elles devraient utiliser des moniteurs distincts. Ceci pourrait entraîner une dégradation des performances subtiles à identifier.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestSynchronized {

  // mauvaise pratique
  private static final String verrou = "verrou";

  public void maMethode() {
    synchronized (verrou) {
      // ...
    }
  }
}
```

Il est préférable d'utiliser une instance dédiée de type Object.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class TestSynchronized {

  private static final Object verrou = new Object();

  public void maMethode() {
    synchronized (verrou) {
      // ...
    }
  }
}
```

Il est recommandé de synchroniser avec le même moniteur les accès en lecture et en écriture à une même donnée.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class MonCompteur {

  private final Object monitor = new Object();

  private int          valeur = 0;

  public int get() {
    synchronized (monitor) {
      return valeur;
    }
  }

  public int incrementer() {
    synchronized (monitor) {
      return ++valeur;
    }
  }
}
```

L'utilisation incorrecte du mot clé synchronized peut conduire à des situations de deadlock dans la JVM.

40.3.1.4. Les avantages et les inconvénients

L'utilisation des moniteurs possède plusieurs avantages :

- ils sont faciles à utiliser
- leur libération est automatiquement garantie par le compilateur et la JVM à la fin du bloc de code : il n'y a pas de risque d'oublier de le libérer

Mais elle présente aussi plusieurs inconvénients :

- ils sont bloquants : par exemple, il est inutile de faire exécuter en boucle par plusieurs threads une même méthode déclarée synchronized. Il est plus performant de la faire exécuter par un seul thread car on s'épargne la gestion du moniteur.
- la simplicité d'utilisation peut masquer des soucis
- il n'est pas possible de vérifier l'état du moniteur avant de demander la pose d'un verrou
- le niveau de verrouillage est le même que les traitements fassent de la lecture ou de l'écriture. Hors la lecture d'une donnée est thread-safe : synchronized peut donc introduire de la contention si plusieurs threads lisent en même temps
- l'attente de l'obtention du verrou est potentiellement illimité : il n'est pas possible de préciser un timeout ni d'interrompre le thread qui attend le verrou. Cela peut engendrer sous certaines circonstances des deadlocks

Pour répondre à ces limitations, Java propose les classes de type Lock : ReadWriteLock et ReentrantLock

40.3.2. Les classes Lock et Condition

Java 5 propose de nouvelles fonctionnalités, regroupées dans le package java .util.concurrent.locks, pour gérer les accès concurrents grâce à des verrous.

Ces verrous reposent sur l'utilisation d'objets : il est donc nécessaire d'en créer une instance et d'invoquer des méthodes, ce qui rend le code à produire plus verbeux par rapport au mot clé synchronized qui est intégré dans le langage.

L'interface java.util.concurrent.locks.Lock définit les fonctionnalités d'un mécanisme de verrous permettant de contrôler l'accès par plusieurs threads à une portion de code.

40.3.2.1. L'interface Lock

Un Lock est un mécanisme de verrou qui permet un accès exclusif à une portion de code par un seul thread. L'utilisation d'un objet de type Lock permet de mettre en place des mécanismes de synchronisation similaires à ceux proposés par le mot clé synchronized mais avec la possibilité d'utiliser des fonctionnalités avancées.

Le grand avantage d'utiliser un Lock est sa flexibilité pour obtenir ou non un verrou sans que cela soit obligatoirement bloquant comme dans le cas de l'utilisation du mot clé synchronized. L'utilisation d'un Lock est donc plus souple que l'utilisation du mot clé synchronized :

- attente bloquante, non bloquante avec prise en compte possible de l'interruption du thread ou avec timeout, ...

- distinguer les accès concurrents en lecture et mise à jour
- support de conditions
- les verrous peuvent être acquis et libérés dans n'importe quel ordre

Elle définit plusieurs méthodes :

Méthode	Rôle
void lock()	Obtenir le verrou : attente indéfinie si celui-ci est déjà pris
void lockInterruptibly()	Obtenir le verrou : attente jusqu'à son obtention ou si le thread courant est interrompu
Condition newCondition()	Obtenir une instance de type Condition associée à l'instance
boolean tryLock()	Obtenir le verrou immédiatement : pas d'attente. Elle renvoie un booléen qui indique si le verrou est obtenu
boolean tryLock(long time, TimeUnit unit)	Obtenir le verrou : attente maximale pour la durée précisée en paramètre ou si le thread courant est interrompu
void unlock()	Libérer le verrou

Le JDK fournit trois implémentations de l'interface Lock :

- ReentrantLock
- ReentrantReadWriteLock.ReadLock
- ReentrantReadWriteLock.WriteLock

De manière générale, un Lock s'utilise avec plusieurs opérations :

- créer une instance de type Lock
- en début de section critique, poser le verrou en invoquant la méthode lock() sur l'instance
- en fin de section critique, libérer le verrou en invoquant la méthode unlock()

La liberté d'acquérir et de libérer un verrou à sa guise implique qu'il est de la responsabilité du développeur de s'assurer que quoi qu'il arrive le verrou sera libéré. Le plus simple est d'exécuter la section critique dans un bloc try et invoquer la méthode unlock() du Lock dans le bloc finally correspondant.

Exemple (code Java 5.0) :

```
Lock verrou = new ReentrantLock();
verrou.lock();
try {
  // section critique protegee par le verrou
} finally {
  verrou.unlock();
}
```

Toutes les implémentations de Lock doivent appliquer la même sémantique de synchronisation de la mémoire telle qu'appliquée par le verrouillage intégré utilisant les moniteurs. Les variables partagées utilisées pendant que le verrou est posé n'ont donc pas besoin d'être définies avec le mot clé volatile car l'utilisation d'un Lock doit offrir les mêmes garanties de visibilité des modifications des variables que le mot clé synchronized.

L'implémentation des différentes formes d'acquisitions du verrou n'a pas de contraintes concernant :

- un support identique de la sémantiques et des garanties offertes
- le support de l'interruption de l'attente du verrou
- l'ordre d'acquisition du verrou par différents threads
- la performance

Chaque implémentation devrait décrire dans sa documentation la sémantique et les garanties offertes par chaque méthode.

Les blocs de code synchronized n'offrent aucune garantie sur l'ordre d'acquisition du verrou par les threads qui sont en son attente. Si de nombreux threads tentent constamment d'obtenir le verrou, il est possible qu'un ou plusieurs threads n'arrivent pas à l'obtenir. Ce phénomène est nommé starvation. Pour l'éviter, une implémentation de type Lock peut prendre en compte le support de l'équité (fairness).

Les instances de Lock sont des objets Java : il est donc possible d'utiliser leur moniteur avec le mot clé synchronized. Dans ce cas, il n'y a aucun lien entre l'obtention du verrou lors de l'invocation de la méthode lock() et l'obtention du verrou sur son moniteur par le mot clé synchronized. Pour éviter toute confusion, il est préférable d'éviter d'utiliser une instance de Lock comme moniteur pour une instruction synchronized.

40.3.2.2. La classe ReentrantLock

La classe ReentrantLock est une implémentation de l'interface Lock qui permet d'utiliser des verrous de manière réentrante.

Le verrou est obtenu par un thread si aucun autre thread ne le possède ou si le verrou est déjà détenu par le thread lui-même.

Elle possède plusieurs méthodes :

Méthode	Rôle
int getHoldCount()	Obtenir le nombre d'obtentions du verrou par le thread courant
protected Thread getOwner()	Renvoyer le thread qui possède le verrou ou null si aucun ne le possède
protected Collection<Thread> getQueuedThreads()	Renvoyer une collection des threads qui sont potentiellement en attente du verrou
int getQueueLength()	Renvoyer un nombre estimé de threads qui sont en attente du verrou
protected Collection<Thread> getWaitingThreads(Condition condition)	Renvoyer une collection des threads qui sont potentiellement en attente de la Condition passée en paramètre
int getWaitQueueLength(Condition condition)	Renvoyer un nombre estimé de threads qui sont en attente de la Condition passée en paramètre
boolean hasQueuedThread(Thread thread)	Vérifier si le thread passé en paramètre est en attente de l'obtention du verrou
boolean hasQueuedThreads()	Vérifier si au moins un thread est en attente de l'obtention du verrou
boolean hasWaiters(Condition condition)	Vérifier si au moins un thread est en attente de la Condition passée en paramètre
boolean isFair()	Renvoyer un booléen qui précise si les demandes d'obtention du verrou sont gérée de manière équitable
boolean isHeldByCurrentThread()	Renvoyer un booléen qui précise si le thread courant possède le verrou
boolean isLocked()	Renvoyer un booléen qui précise le verrou est détenu par un thread
void lock()	Obtenir le verrou : attente indéfinie si celui-ci est déjà pris
void lockInterruptibly()	Obtenir le verrou : attente jusqu'à son obtention ou si le thread courant est interrompu
Condition newCondition()	

	Obtenir une instance de type Condition associé à l'instance
boolean tryLock()	Obtenir le verrou immédiatement : pas d'attente. Elle renvoie un booléen qui indique si le verrou est obtenu
boolean tryLock(long timeout, TimeUnit unit)	Obtenir le verrou : attente maximale pour la durée précisée en paramètre ou si le thread courant est interrompu
void unlock()	Libérer le verrou

Certaines méthodes sont protected : leur utilisation est à réserver pour de l'instrumentation ou du monitoring.

La classe ReentrantLock utilise en interne des opérations de type CAS et des variables atomiques ou volatiles pour limiter les temps de contention.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class TestReentrantLock {

  private final Lock verrou = new ReentrantLock();

  public void methodeA() throws InterruptedException {
    verrou.lock();
    try {
      System.out.println("MethodeA : " + Thread.currentThread().getName());
      Thread.sleep(2000);
      methodeB();
      Thread.sleep(5000);
    } finally {
      verrou.unlock();
    }
  }

  public void methodeB() {
    verrou.lock();
    try {
      System.out.println("MethodeB : " + Thread.currentThread().getName());
    } finally {
      verrou.unlock();
    }
  }

  public static void main(String[] args) {

    final TestReentrantLock sut = new TestReentrantLock();

    Thread[] threads = new Thread[2];

    threads[0] = new Thread(new Runnable() {
      @Override
      public void run() {
        try {
          System.out.println("Debut thread 0");
          sut.methodeA();
          System.out.println("fin thread 0");
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
    }, "Thread 0");

    threads[1] = new Thread(new Runnable() {
      @Override
      public void run() {
        System.out.println("Debut thread 1");
        sut.methodeB();
        System.out.println("Fin thread 1");
      }
    }, "Thread 1");

    threads[0].start();
    threads[1].start();

  }
}
```

Résultat :

```
Debut thread 0
MethodeA : Thread 0
Debut thread 1
MethodeB : Thread 0
fin thread 0
MethodeB : Thread 1
Fin thread 1
```

L'implémentation de la classe ReentrantLock permet la pose d'un verrou de manière réentrante.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class TestReentrantLock {

  private final Lock lock = new ReentrantLock();

  public void methodeA() {
    lock.lock();

    try {
      System.out.println("MethodeA");
    } finally {
      lock.unlock();
    }
  }

  public void methodeB() {
    lock.lock();
    try {
      System.out.println("MethodeB");
    } finally {
      lock.unlock();
    }
  }
}
```

Ce code est équivalent à celui ci-dessous qui utilise le mot clé synchronized.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

public class TestReentrantLock {
  private final Object lock = new Object();
  public void methodeA() {
    synchronized(lock) {
      System.out.println("MethodeA");
    }
  }
  public void methodeB() {
    synchronized(lock) {
      System.out.println("MethodeB");
    }
  }
}
```

Dans un contexte simple, le code utilisant un objet de type Lock est beaucoup plus verbeux et plus risqué puisque la libération du verrou est à la charge du développeur. Cependant, l'utilisation d'un Lock offre des fonctionnalités avancées qui permettent entre autre de pallier certaines contraintes liées à l'utilisation du mot clé synchronized.

La classe ReentrantLock possède deux constructeurs :

Constructeur	Rôle
ReentrantLock()	Constructeur par défaut
ReentrantLock(boolean)	Constructeur qui permet de préciser si le verrou doit prendre en compte l'équité envers les threads qui tentent de poser le verrou

Si le paramètre vaut true, alors le verrou sera prioritairement donné au thread qui l'attend depuis le plus longtemps. Par défaut, aucun ordre n'est garanti quant à l'obtention du verrou par les threads qui attendent de l'avoir.

Le support de l'équité est plus coûteux : il ne faut l'utiliser que lorsque les circonstances l'obligent, généralement sous très fortes conditions de charge. La méthode tryLock() ne prend pas en compte l'équité.

La classe ReentrantLock est Serializable. Attention cependant, lorsqu'une instance est désérialisée, le verrou est libre même si celui-ci était détenu par un thread au moment de la sérialisation de l'objet.

La classe supporte un maximum de Integer.MAX appels réentrants : si cette limite est dépassée, une exception de type java.lang.Error avec le message «Maximum lock count exceeded» est levée.

L'utilisation de verrous dans une application multithread peut entrainer dans certaines circonstances des situations d'interblocage entre au moins deux threads. Cette situation nommée deadlock est dramatique car pour en sortir la seule solution est de relancer la JVM.

Le mot clé synchronized ne propose rien pour éviter cette situation lorsque les conditions sont réunies pour quelle survienne car l'utilisation du mot clé synchronised est bloquante jusqu'à l'obtention du verrou.

Pour tenter de pallier cette problématique, l'interface Lock propose la méthode tryLock().

Une surcharge de la méthode tryLock() de l'interface Lock qui attend en paramètre un entier long pour la quantité et un TimeUnit qui précise l'unité temporelle permet de tenter d'obtenir le verrou avant un certain timeout. Elle renvoie un booléen qui précise si le verrou a été obtenu avant le timeout. Il est alors de la responsabilité de l'appelant de gérer le cas où le verrou n'est pas obtenu par exemple en faisant un nombre limité de nouvelles tentatives éventuellement après un certain délai d'attente.

Remarque : la méthode tryLock() ne tient jamais compte du fait que le ReentrantLock est configuré pour être équitable. Elle tente d'acquérir le verrou immédiatement sans tenir compte du fait que d'autres threads sont déjà en attente pour l'obtention du verrou. Si l'équitabilité doit être respectée alors il faut utiliser la surcharge de la méthode tryLock() en lui passant en paramètre 0 et TimeUnit.SECONDS.

Cette surcharge prend en compte :

- l'équitabilité si l'instance est configurée pour être équitable
- la détection d'une interruption

La méthode tryLock() tente d'obtenir immédiatement le verrou :

- si elle obtient le verrou alors elle initialise le compteur holdCount à 1 et renvoie true
- si le verrou est déjà détenu par l'instance alors elle incrémente le compteur holdCount et renvoie true

- sinon elle renvoie false

L'utilisation d'un Lock offre une plus grande souplesse dans l'obtention et la libération du verrou. Il est possible d'acquérir le verrou dans une méthode et de le libérer dans une autre. Cependant cette liberté est dangereuse car le développeur doit s'assurer que le verrou est correctement libéré dans tous les cas, sinon le verrou restera indéfiniment posé et ne pourra plus être obtenu par d'autres threads. Aucun contrôle n'est effectué par le compilateur.

L'utilisation de Lock peut permettre d'améliorer les performances et de diminuer les situations de deadlocks. Leur mise en oeuvre est cependant plus verbeuse et plus risquée que d'utiliser le mot clé synchronized.

40.3.2.3. L'interface Condition

Une instance de type Condition permet de mettre en attente un thread jusqu'à ce qu'il reçoive une notification lorsque la condition est remplie.

Lorsqu'un objet de type Lock est utilisé pour poser un verrou à la place du mot clé synchronized, alors un objet de type Condition est utilisé à la place des méthodes wait(), notify() et notifyAll().

Remarque : une implémentation de l'interface Condition peut avoir une sémantique et un comportement différents de celui des méthodes de la classe Object liées au moniteur.

Elle définit plusieurs méthodes :

Méthode	Rôle
void await()	Le thread courant suspend son exécution et attend de recevoir un signal ou d'être interrompu
boolean await(long time, TimeUnit unit)	Le thread courant suspend son exécution et attend de recevoir un signal ou d'être interrompu ou que le timeout précisé en paramètre soit atteint
long awaitNanos(long nanosTimeout)	Le thread courant suspend son exécution et attend de recevoir un signal ou d'être interrompu ou que le timeout précisé en paramètre soit atteint
void awaitUninterruptibly()	Le thread courant suspend son exécution et attend de recevoir un signal : il ne peut pas être interrompu
boolean awaitUntil(Date deadline)	Le thread courant suspend son exécution et attend de recevoir un signal ou d'être interrompu ou que la date/heure limite précisée en paramètre soit atteinte
void signal()	Envoyer un signal à un thread qui est en attente d'un signal de cette condition
void signalAll()	Envoyer un signal à tous les threads qui sont en attentes d'un signal de cette condition

La méthode await() et ses surcharges permettent de mettre le thread courant en attente d'un signal. Le fonctionnement de la méthode await() est similaire à celui de la méthode wait() de la classe Object qui met le thread courant en attente et libère le lock. Cette attente dure jusqu'à ce qu'un autre thread invoque la méthode signal() ou signalAll() sur l'instance de type Condition ou interrompe le thread courant. Avant que la méthode ne se termine elle réacquiert le lock.

Les méthodes signal() et signalAll() permettent de notifier le ou les threads en attente du fait que la condition est atteinte.

Une instance de type Condition permet de suspendre l'exécution d'un thread jusqu'à ce qu'un autre thread lui envoie une notification. Comme cela implique un accès concurrent, une Condition est toujours associée à un Lock. Pour obtenir une nouvelle instance de type Condition, il faut invoquer la méthode newInstance() du Lock à laquelle elle sera liée.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.BlockingQueue;
import java.util.concurrent.RejectedExecutionHandler;
import java.util.concurrent.ThreadFactory;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

public class PausableThreadPoolExecutor extends ThreadPoolExecutor {
  private boolean       enPause;
  private ReentrantLock pauseLock = new ReentrantLock();
  private Condition     reactive  = pauseLock.newCondition();

  public PausableThreadPoolExecutor(int corePoolSize, int maximumPoolSize,
      long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue,
      RejectedExecutionHandler handler) {
    super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
        handler);
  }

  public PausableThreadPoolExecutor(int corePoolSize, int maximumPoolSize,
      long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue,
      ThreadFactory threadFactory, RejectedExecutionHandler handler) {
    super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
        threadFactory, handler);
  }

  public PausableThreadPoolExecutor(int corePoolSize, int maximumPoolSize,
      long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue,
      ThreadFactory threadFactory) {
    super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
        threadFactory);
  }

  public PausableThreadPoolExecutor(int corePoolSize, int maximumPoolSize,
      long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue) {
    super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue);
  }

  @Override
  protected void beforeExecute(Thread thread, Runnable runnable) {
    super.beforeExecute(thread, runnable);
    pauseLock.lock();
    try {
      while (enPause) {
        reactive.await();
      }
    } catch (InterruptedException ie) {
      thread.interrupt();
    } finally {
      pauseLock.unlock();
    }
  }

  public void pause() {
    pauseLock.lock();
    try {
      enPause = true;
    } finally {
      pauseLock.unlock();
    }
  }

  public void resume() {
    pauseLock.lock();
    try {
      enPause = false;
      reactive.signalAll();
    } finally {
      pauseLock.unlock();
    }
  }
}
```

L'exemple ci-dessus implémente un ThreadPoolExecutor qui peut être mis en pause. La mise en pause se fait à la fin de l'exécution de la tâche en cours ou juste avant l'exécution de la tâche suivante. La gestion de la mise en pause gère les accès concurrents en utilisant un verrou de type ReentrantLock. La mise en attente utilise une condition.

Le moniteur d'une instance de type Condition peut être utilisé avec le mot clé synchronized puisque c'est un objet comme un autre. Dans cas, il n'y pas de lien avec le Lock associé à l'instance ni avec ses méthodes await(), signal() et signalAll(). Il est cependant recommandé d'éviter cette pratique pour limiter les risques de confusion.

40.3.2.4. L'interface ReadWriteLock et la classe ReentrantReadWriteLock

L'un des inconvénients du mot clé synchronized est qu'il ne permet pas de différencier d'accès à la portion de code pour de la lecture uniquement. Quelque soit les traitements réalisés par la portion de code, un seul thread peut l'exécuter. Si cette portion de code ne fait que de la lecture, alors la scalabilité est restreinte.

Pour remedier à cela la classe java.util.concurrent.locks.ReentrantReadWriteLock permet de poser des verrous différenciés pour la lecture seule et pour les modifications. Son utilisation permet à plusieurs threads d'exécuter une portion de code qui ne fait que de la lecture mais n'autorise qu'un seul thread à exécuter une portion de code qui fait des mises à jour. Dans certaines situations de forte concurrence effectuant beaucoup de lectures, cela peut limiter la contention et donc améliorer les performances.

Plusieurs threads peuvent lire une ressource partagée sans poser de soucis d'accès concurrents. Ceux-ci surviennent dans deux circonstances :

- lorsqu'une lecture et une modification surviennent de manière concurrente
- lorsque plusieurs modifications surviennent de manière concurrente

Ces fonctionnalités sont définies dans l'interface java.util.concurrent.locks.ReadWriteLock.

Un ReadWriteLock est un type de Lock qui permet de distinguer un verrou en lecture et un autre en écriture. Ceci est particulièrement intéressant si les lectures sont plus nombreuses que les modifications. Plusieurs threads peuvent lire de manière concurrente une donnée tant que durant ces lectures, aucune modification n'est apportée. Par contre, si un thread veut effectuer une mise à jour il doit avoir un verrou exclusif et aucune lecture ne doit être en cours.

Un ReadWriteLock permet d'avoir plusieurs threads en lecture mais un seul en modification. Lors de son utilisation, plusieurs conditions doivent être remplies pour avoir le verrou :

- en lecture : le verrou de modification ne doit pas être obtenu et aucune demande de pose du verrou en modification ne doit être en cours d'obtention. Plusieurs threads peuvent obtenir ce verrou
- en modification : aucun verrou en lecture ou modification ne doit être posé. Un seul thread peut poser le verrou en modification

Les demandes de pose du verrou en modification doivent être prioritaires sur celles des verrous en lecture car partant du principe qu'il y a beaucoup plus de lecture que de modifications, il se pourrait que l'obtention de ses verrous soit long voir même impossible si les lectures sont ininterrompues.

L'interface ReadWriteLock ne définit que deux méthodes :

Méthode	Rôle
Lock readLock()	Obtenir l'instance de type Lock pour la lecture
Lock writeLock()	Obtenir l'instance de type Lock pour la modification

Le JDK fournit en standard une implémentation avec la classe ReentrantReadWriteLock.

L'implémentation de la classe ReentrantReadWriteLock utilise deux instances de type Lock : une pour la gestion des verrous en lecture (readLock) et une autre pour la gestion des verrous en modification (writeLock).

Lors de l'utilisation du mot clé synchronized, le verrou est automatiquement libéré à la sortie du bloc de code quelque soit les raisons de cette sortie, notamment si une exception est levée.

La lecture seule d'une donnée ne peut pas engendrer de problème d'accès concurrents tant que cette donnée n'est pas modifiée. La pose d'un verrou exclusif pour réaliser cette lecture est donc pénalisante et inutile tant que la donnée n'est pas modifiée. La modification de la donnée doit elle être faite avec un verrou exclusif et la lecture de la donnée par un autre thread doit être bloquée.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class MonCompteur {
  private long                valeur;

  private final ReadWriteLock verrouRW = new ReentrantReadWriteLock();

  public long get() {
    Lock verrouR = verrouRW.readLock();
    verrouR.lock();
    try {
      return valeur;
    } finally {
      verrouR.unlock();
    }
  }

  public long incrementer() {
    Lock verrouW = verrouRW.writeLock();
    verrouW.lock();
    try {
      return ++valeur;
    } finally {
      verrouW.unlock();
    }
  }

  public static void main(String[] args) {
    MonCompteur compteur = new MonCompteur();
    System.out.println(compteur.get());
    System.out.println(compteur.incrementer());
    System.out.println(compteur.incrementer());
  }
}
```

Remarque : cet exemple a plus une vocation pédagogique que pratique.

Il est préférable d'utiliser un ReentrantReadWriteLock plutôt que le mot clé synchronized pour avoir un contrôle plus précis du verrou et obtenir de meilleures performances surtout si les opérations concernent majoritairement des lectures.

La mise en oeuvre d'un ReadWriteLock est surtout intéressante si ce sont majoritairement des opérations en lecture seule qui sont effectuées : un cas d'utilisation classique est un cache.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class MonBean {
  private final ReentrantReadWriteLock rrwLock   = new ReentrantReadWriteLock();
  private final Lock                   readLock  = rrwLock.readLock();
  private final Lock                   writeLock = rrwLock.writeLock();

  private String                       valeur;

  public String getValeur() {
    readLock.lock();
    try {
      return valeur;
    } finally {
      readLock.unlock();
    }
  }

  public void setValeur(String Valeur) {
    writeLock.lock();
    try {
      this.valeur = Valeur;
    } finally {
      writeLock.unlock();
    }
  }
}
```

40.4. Les opérations atomiques

Une opération atomique est une opération qui ne peut pas être exécutée partiellement : toutes ses instructions ont la garantie d'être exécutées sans interruption.

En Java, seules les opérations de lecture et d'écriture d'une variable sont atomiques sauf si celles-ci sont de type long ou double.

Par exemple, l'opérateur ++ n'est pas atomique puisqu'il requiert au moins trois opérations (en réalité, il en faut plus) :

- la lecture en mémoire de la valeur courante
- son incrémentation
- l'écriture en mémoire de la nouvelle valeur

Il est possible que deux threads réalisent la lecture en même temps puis l'incrémentation. Dans ce cas, la valeur ne sera incrémentée qu'une seule fois puisque la lecture par les deux threads donne la même valeur. Si cette incrémentation doit permettre d'obtenir une valeur unique alors cela ne sera pas le cas.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class MonCompteur {
  private int valeur;

  public int getValeur() {
    return valeur;
  }

  public int getNextValeur() {
    return ++valeur;
  }
}
```

Cette classe fonctionne très bien dans un contexte monothread. Par contre, elle ne fonctionne pas correctement dans un contexte multithread.

Exemple :

```
package fr.jmdoudoux.dej.thread;

  public class TestMonCompteur {
    public static void main(String[] args) throws InterruptedException {
      final MonCompteur compteur = new MonCompteur();
      Thread[] threads = new Thread[20];
      Runnable thread = new Runnable() {
        @Override
        public void run() {
          for (int i = 0; i < 10000; i++) {
            compteur.getNextValue();
          }
        }
      };

      for (int i = 0; i < 20; i++) {
        threads[i] = new Thread(thread);
        threads[i].start();
      }

      for (int i = 0; i < 20; i++) {
        threads[i].join();
      }

      System.out.println(compteur.getValue());
    }
  }
```

Les résultats de plusieurs exécutions sont différents et complètement aléatoires. La seule chose qui est vraie est que les résultats sont toujours faux.

Résultat :

```
184782
198318
195141
```

Avant Java 5, il était nécessaire de protéger les opérations en utilisant des verrous sur un moniteur.

Exemple :

```
package fr.jmdoudoux.dej.thread;

public class MonCompteur {
  private int valeur;

  public synchronized int get() {
    return valeur;
  }

  public synchronized int incrementer() {
    return ++valeur;
  }
}
```

Résultat :

```
200000
200000
200000
```

L'utilisation de verrous par un moniteur est une opération coûteuse et surtout bloquante. Si la section critique est petite le surcoût est important et l'overhead peut devenir conséquent si elle est invoquée de nombreuses fois. De plus, si de nombreux threads attendent pour avoir le verrou cela peut induire une forte contention.

Il est possible d'utiliser des algorithmes qui ne sont pas bloquants par opposition à l'utilisation de verrous avec un moniteur qui est bloquante. Le fait de ne pas être bloquant améliore les performances et la scalabilité notamment lorsque le nombre de threads augmentent.

Les algorithmes bloquants utilisent une approche pessimiste. Les algorithmes non bloquants utiliser une approche optimiste : ils sont plus difficile à écrire. Ils reposent sur un principe :

- réaliser une opération
- si celle-ci échoue alors elle effectue une autre opération généralement une retentative de l'opération elle-même

Le principe est donc de retenter l'opération jusqu'à ce qu'elle soit réalisée. Cette implémentation nécessite plus de code mais c'est le coût à payer pour mettre en oeuvre un algorithme non bloquant. Les algorithmes de type CAS sont généralement plus performants que d'utiliser les moniteurs qui sont bloquants.

Certains processeurs proposent des instructions qui facilitent la mise en oeuvre de ces algorithmes non bloquants. L'opération de ce type la plus couramment utilisée est l'opération CAS (Compare And Swap). Elle requiert généralement trois paramètres : l'adresse mémoire de la donnée, sa valeur courante et sa valeur souhaitée. Elle modifie la valeur à l'adresse passée en mémoire si la valeur est celle souhaitée sinon elle ne fait rien. L'opération renvoie toujours la valeur courante.

Si plusieurs threads invoquent cette opération, un des threads met à jour la valeur mais pas les autres. Généralement les autres threads vont retenter l'exécution de l'opération jusqu'à ce que la mise à jour soit faite.

Une opération de type compare and set repose sur le même principe mais au lieu de renvoyer la valeur, elle retourne un booléen qui précise si la mise à jour a été effectuée ou non.

A partir de Java 5, plusieurs classes dans le package java.util.concurrent.atomic permettent de proposer cette fonctionnalité pour différents types. Ces classes proposent différentes méthodes atomiques. Leur implémentation ne repose pas sur l'utilisation d'un moniteur mais sur le principe CAS (Compare And Set).

Classe	Rôle
AtomicBoolean	Encapsule une valeur booléenne qui peut être mise à jour de manière atomique
AtomicInteger	Encapsule une valeur entière qui peut être mise à jour de manière atomique
AtomicIntegerArray	Encapsule un tableau de valeurs entières qui peuvent être mises à jour de manière atomique
AtomicIntegerFieldUpdater<T>	Classe utilitaire qui permet de modifier un champs volatile
AtomicLong	Encapsule un entier long qui peut être mis à jour de manière atomique
AtomicLongArray	Encapsule un tableau d'entiers long qui peuvent être mis à jour de manière atomique
AtomicLongFieldUpdater<T>	Encapsule une valeur entière longue qui peut être mise à jour de manière atomique
AtomicMarkableReference<V>	Encapsule un booléen et une référence sur un objet de type V qui peuvent être modifiés de manière atomique
AtomicReference<V>	Encapsule une référence sur un objet qui peut être mise à jour de manière atomique
AtomicReferenceArray<E>	Encapsule un tableau de références sur des objets qui peuvent être mises à jour de manière atomique
AtomicReferenceFieldUpdater<T,V>	Encapsule une référence qui peut être mise à jour de manière atomique
AtomicStampedReference<V>	Encapsule un entier et une référence sur un objet de type V qui peuvent être modifiés de manière atomique

Toutes ces classes possèdent différentes méthodes qui permettent de modifier la valeur encapsulée en utilisant des opérations de type CAS : compareAndSet(), getAndSet(), get(), set(), ...

Les classes AtomicBoolean, AtomicInteger, AtomicLong et AtomicReference permettent d'obtenir et de mettre à jour de manière atomique la valeur qu'elles encapsulent. Elles proposent aussi des méthodes qui facilitent certaines opérations de mise à jour comme l'incrémentation.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.atomic.AtomicInteger;

```

```
public class MonCompteur {

  private AtomicInteger valeur = new AtomicInteger(0);

  public int get() {
    return valeur.get();
  }

  public int incrementer() {
    return valeur.incrementAndGet();
  }
}
```

Résultat :

```
200000
200000
200000
```

La méthode compareAndSet() attend en paramètre la valeur attendue et la nouvelle valeur : si la valeur courante est celle souhaitée alors elle met à jour avec la nouvelle valeur fournie. Elle renvoie un booléen qui précise si la mise à jour a été effectuée.

L'implémentation de ces méthodes peut utiliser des instructions spécifiques au processeur pour réaliser ces traitements de manière optimum. Si ce type d'opération n'est pas disponible sur la plateforme d'exécution, alors l'implémentation va utiliser en interne un mécanisme de verrous pour garantir l'atomicité de l'opération. Dans ce cas, les traitements ne sont plus non bloquants.

Attention : la méthode compareAndSet() ne peut pas être utilisée comme un remplacement pour gérer des verrous : son utilisation se limite à la mise à jour de la valeur encapsulée dans l'objet.

La méthode weakCompareAndSet() peut être utilisée dans certaines circonstances particulières. Sur certaines plateformes, la méthode weakCompareAndSet() peut être plus efficace que la méthode compareAndSet(). Cependant son utilisation présente plusieurs inconvénients :

- son comportement n'est pas identique sur toutes les plateformes
- elle peut renvoyer false sans raison évidente (spurious failure). C'est notamment le cas sur des plateformes qui ne propose pas d'instructions unitaires pour réaliser une opération de type CAS. L'opération doit alors être retentée
- une partie de ses performances s'explique par le fait qu'elle ne permet pas de définir une relation de type happens-before. Elle ne permet donc pas obligatoirement aux autres threads de voir les modifications effectuées sur d'autres données avant son appel ni de garantir un ordre d'exécution

Les cas d'utilisation sont donc relativement rares : par exemple pour mettre à jour des compteurs ou des données statistiques relatives à la performance sous réserve qu'il n'y a pas de relation happens-before. Elle garantit uniquement l'atomicité de l'opération.

La méthode weakCompareAndSwap() n'offre pas de garantie d'être plus rapide mais peut être plus rapide selon la plateforme et la JVM utilisée.

Par exemple, en Java 6, l'implémentation du JDK de Sun des méthodes compareAndSet() et weakCompareAndSet() sont identiques : elles invoquent toutes les deux compareAndSwapXXX() de la classe sun.misc.Unsafe. Leurs performances sont donc identiques.

De plus, par exemple, les processeurs Intel x86 (à partir des architectures 80486 et Itanium) proposent l'instruction LOCK CMPXCHG qui met en oeuvre le compare and exchange avec la mise en oeuvre d'une barrière de mémoire. Ils ne possèdent pas (encore) une instruction qui ne met pas en oeuvre de barrière de mémoire. Du coup pour une JVM HotSpot, sur une architecture x86, les performances des méthodes compareAndSet() et weakCompareAndSet() sont identiques. Sur d'autres architectures, le comportement peut être différent : notamment sur les plateformes qui ne possèdent pas d'instructions dédiées, il est fréquent d'utiliser une séquence d'instructions LL/SC (Load-Link et Store-Conditional).

Le package contient plusieurs classes AtomicXXXFiledUpdater qui sont des utilitaires permettant de modifier par introspection la valeur d'un champ volatile du type XXX de n'importe quelle instance.

Le package contient plusieurs classes AtomicXXXArray (AtomicIntegerArray, AtomicLongArray et AtomicReferenceArray) qui proposent des opérations atomiques réalisables sur le tableau de type XXX qu'elles encapsulent. Elles permettent notamment une sémantique similaire à volatile sur les éléments du tableau.

La classe AtomicMarkableReference encapsule un booléen et une référence qui peuvent être modifiés de manière atomique.

La classe AtomicStampedReference encapsule un entier de type int et une référence qui peuvent être modifiés de manière atomique.

Remarque : les classes AtomicXXX ne sont pas des classes de remplacement de classes de type wrapper java.lang.XXX. Elles ne redéfinissent pas la méthode hashCode() et comme elles sont mutables, il n'est pas recommandé de les utiliser comme clés dans une collection de type Map.

L'utilisation de ces classes permet d'avoir une meilleure scalabilité par rapport à l'utilisation de verrous avec des moniteurs. Cependant, sous forte contention, les performances des opérations de type CAS peuvent se dégrader fortement.

L'utilisation de certaines méthodes de ces classes offrent des garanties particulières : elles sont pour certaines similaires à l'utilisation du mot clé volatile telle que définie par la section 17.4 de la JLS.

Méthode	Effets sur la mémoire
get()	Similaire à la lecture d'une variable volatile
set()	Similaire à la modification d'une variable volatile
lazySet()	La sémantique garantit que l'écriture ne sera pas réordonnée vis à vis d'écritures précédentes mais pourra être réordonnée avec les écritures suivantes. Du coup, il n'y a pas de garantie sur la visibilité de la modification par les autres threads. En terme de barrières de mémoire, la méthode lazySet() exécute une barrière de type store-store qui est peu coûteuse mais ne fait pas de barrière de type store-load. Parmi les cas d'utilisation, il y a par exemple la remise à null d'une référence puisque cette référence ne sera plus utilisée par ailleurs.
weakCompareAndSet()	Lecture et écriture conditionnelle sans relation de type happens-before
compareAndSet() et les autres méthodes telles que incrementAndSet()	Similaire à la lecture et la modification d'une variable volatile

40.4.1. La classe AtomicBoolean

La classe java.util.concurrent.atomic.AtomicBoolean encapsule une valeur booléenne qui peut être lue et mise à jour de manière atomique.

Elle possède deux constructeurs :

Constructeur	Rôle
AtomicBoolean()	La valeur encapsulée est false
AtomicBoolean(boolean initialValue)	La valeur encapsulée est celle fournie en paramètre

Elle possède plusieurs méthodes :

Méthode	Rôle
boolean compareAndSet(boolean expect, boolean update)	Tenter de mettre à jour de manière atomique la valeur avec celle du paramètre update si la valeur courante est égale à celle du paramètre expect. Renvoie un booléen qui précise si la valeur a été modifiée lors des traitements.
boolean get()	Renvoyer la valeur courante
boolean getAndSet(boolean newValue)	Modifier la valeur avec celle en paramètre et renvoie la valeur avant modification
void lazySet(boolean newValue)	Depuis Java 6
void set(boolean newValue)	Modifier la valeur
String toString()	Renvoyer la valeur sous la forme d'une chaîne de caractères
boolean weakCompareAndSet(boolean expect, boolean update)	

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.atomic.AtomicBoolean;

public class TestAtomicBoolean {

  public static void main(String[] args) {
    AtomicBoolean atomicBoolean = new AtomicBoolean(true);
    System.out.println("1 valeur=" + atomicBoolean.get());

    boolean valeur = atomicBoolean.getAndSet(false);
    System.out.println("2 valeur=" + valeur);
    System.out.println("3 valeur=" + atomicBoolean.get());

    boolean isOk = atomicBoolean.compareAndSet(true, false);
    System.out.println("isOk=" + isOk);
    System.out.println("4 valeur=" + atomicBoolean.get());

    isOk = atomicBoolean.compareAndSet(false, true);
    System.out.println("isOk=" + isOk);
    System.out.println("5 valeur=" + atomicBoolean.get());
  }
}
```

40.4.2. La classe AtomicInteger

La classe java.util.concurrent.atomic.AtomicInteger encapsule une valeur entière qui peut être mise à jour de manière atomique.

La classe AtomicInteger hérite de la classe Number et implémente l'interface Serializable. Elle ne doit cependant pas être utilisée en remplacement de la classe Integer.

Elle possède deux constructeurs :

Constructeur	Rôle
AtomicInteger()	La valeur encapsulée est 0
AtomicInteger(int initialValue)	La valeur encapsulée est celle fournie en paramètre

Elle possède plusieurs méthodes :

Méthode	Rôle
int accumulateAndGet(int x, IntBinaryOperator accumulatorFunction)	Modifier la valeur avec celle retournée par l'expression Lambda passée en paramètre qui sera invoquée avec la valeur courante et celle fournie. Renvoie la valeur après modification Depuis Java 8
int addAndGet(int delta)	Ajouter delta à la valeur et la renvoyer
boolean compareAndSet(int expect, int update)	Mettre à jour de manière atomique la valeur courante avec celle du paramètre update si la valeur courante est égale à la valeur du paramètre expect. Le booléen indique si la mise à jour a été effectuée
int decrementAndGet()	Décrémenter la valeur et la renvoyer
double doubleValue()	Renvoyer la valeur sous la forme d'un double
float floatValue()	Renvoyer la valeur sous la forme d'un float
int get()	Obtenir la valeur courante
int getAndAccumulate(int x, IntBinaryOperator accumulatorFunction)	Modifier la valeur avec celle retournée par l'expression Lambda passée en paramètre qui sera invoquée avec la valeur courante et celle fournie. Renvoie la valeur avant la modification Depuis Java 8
int getAndAdd(int delta)	Ajouter delta à la valeur et renvoyer la valeur avant modification
int getAndDecrement()	Décrémenter la valeur et renvoyer la valeur avant incrémentation
int getAndIncrement()	Incrémenter la valeur et renvoyer la valeur avant incrémentation
int getAndSet(int newValue)	Mettre à jour la valeur et renvoyer la valeur avant modification
int getAndUpdate(IntUnaryOperator updateFunction)	Modifier la valeur avec celle retournée par l'expression Lambda passée en paramètre. Renvoie la valeur avant la modification Depuis Java 8
int incrementAndGet()	Incrémenter la valeur et la renvoyer
int intValue()	Renvoyer la valeur sous la forme d'un int
void lazySet(int newValue)	Depuis Java 6
long longValue()	Renvoyer la valeur sous la forme d'un entier long
void set(int newValue)	Mettre à jour la valeur
int updateAndGet(IntUnaryOperator updateFunction)	Modifier la valeur avec celle retournée par l'expression Lambda passée en paramètre. Renvoie la valeur après modification Depuis Java 8
boolean weakCompareAndSet(int expect, int update)	

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.atomic.AtomicInteger;

public class MonCompteur {

  private AtomicInteger valeur = new AtomicInteger(0);

  public int get() {
    return valeur.get();
  }
  public int incrementer() {
    return valeur.incrementAndGet();
  }
}
```

Il est possible d'utiliser la méthode compareAndSet() pour modifier la valeur.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.atomic.AtomicInteger;

public class MonCompteur {

  private AtomicInteger valeur = new AtomicInteger(0);

  public int get() {
    return valeur.get();
  }

  public int incrementer() {
    int courante = valeur.get();
    while (!valeur.compareAndSet(courante, courante + 1)) {
      courante = valeur.get();
    }
    return valeur.get();
  }
}
```

L'implémentation de la méthode incrementAndGet() est d'ailleurs assez similaire à celle de l'implémentation de la méthode incrementer() ci-dessus.

40.4.3. La classe AtomicLong

La classe java.util.concurrent.atomic.AtomicLong encapsule une valeur entière longue qui peut être mise à jour de manière atomique

La classe AtomicLong hérite de la classe Number et implémente l'interface Serializable. Elle ne doit cependant pas être utilisée en remplacement de la classe Long.

Elle possède deux constructeurs :

Constructeur	Rôle
AtomicLong()	La valeur encapsulée est 0
AtomicLong(long initialValue)	La valeur encapsulée est celle fournie en paramètre

Elle possède plusieurs méthodes :

Méthode	Rôle
long accumulateAndGet(long x, LongBinaryOperator accumulatorFunction)	Modifier la valeur avec celle retournée par l'expression Lambda passée en paramètre qui sera invoquée avec la valeur courante et celle fournie. Renvoie la valeur après modification Depuis Java 8
long addAndGet(long delta)	Ajouter delta à la valeur et la renvoyer

boolean compareAndSet(long expect, long update)	Mettre à jour de manière atomique la valeur courante avec celle du paramètre update si la valeur courante est égale à la valeur du paramètre expect. Le booléen indique si la mise à jour a été effectuée
long decrementAndGet()	Décrémenter la valeur et la renvoyer
double doubleValue()	Renvoyer la valeur sous la forme d'un double
float floatValue()	Renvoyer la valeur sous la forme d'un float
long get()	Obtenir la valeur courante
long getAndAccumulate(int x, LongBinaryOperator accumulatorFunction)	Modifier la valeur avec celle retournée par l'expression Lambda passée en paramètre qui sera invoquée avec la valeur courante et celle fournie. Renvoie la valeur avant la modification Depuis Java 8
long getAndAdd(int delta)	Ajouter delta à la valeur et renvoyer la valeur avant modification
long getAndDecrement()	Décrémenter la valeur et renvoyer la valeur avant incrémentation
long getAndIncrement()	Incrémenter la valeur et renvoyer la valeur avant incrémentation
long getAndSet(long newValue)	Mettre à jour la valeur et renvoyer la valeur avant modification
long getAndUpdate(LongUnaryOperator updateFunction)	Modifier la valeur avec celle retournée par l'expression Lambda passée en paramètre. Renvoie la valeur avant la modification Depuis Java 8
long incrementAndGet()	Incrémenter la valeur et la renvoyer
int intValue()	Renvoyer la valeur sous la forme d'un int
void lazySet(long newValue)	Depuis Java 6
long longValue()	Renvoyer la valeur sous la forme d'un entier long
void set(long newValue)	Mettre à jour la valeur
long updateAndGet(LongUnaryOperator updateFunction)	Modifier la valeur avec celle retournée par l'expression Lambda passée en paramètre. Renvoie la valeur après modification Depuis Java 8
boolean weakCompareAndSet(long expect, long update)	

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.atomic.AtomicLong;

public class MonCompteur {

  private AtomicLong valeur = new AtomicLong(0);

  public long get() {
    return valeur.get();
  }

  public long incrementer() {
    return valeur.incrementAndGet();
  }
}
```

Il est possible d'utiliser la méthode compareAndSet() pour modifier la valeur.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.atomic.AtomicLong;

public class MonCompteur {

  private AtomicLong valeur = new AtomicLong();

  public long get() {
    return valeur.get();
  }

  public long incrementer() {
    long courante = valeur.get();
    while (!valeur.compareAndSet(courante, courante + 1)) {
      courante = valeur.get();
    }
    return valeur.get();
  }
}
```

L'implémentation de la méthode incrementAndGet() est d'ailleurs assez similaire à celle de l'implémentation de la méthode incrementer() ci-dessus.

40.4.4. Les classes AtomicIntegerArray, AtomicLongArray et AtomicReferenceArray<E>

Les classes AtomicIntegerArray, AtomicLongArray et AtomicReferenceArray<E> implémentent des tableaux atomiques respectivement de type int, long et Object.

Ces classes sont utiles car le fait de déclarer un tableau avec le mot clé volatile ne concerne que les opérations sur la référence du tableau mais ne concerne pas celles sur les références des éléments du tableau.

La plupart des opérations qui réalisent un traitement atomique attendent en premier paramètre l'index de l'élément du tableau sur lequel elles doivent opérer.

Le JDK ne propose pas de classe pour gérer un tableau de booléen atomique : il est possible d'utiliser la classe AtomicIntegerArray avec des valeurs 0 ou 1.

40.4.4.1. La classe AtomicIntegerArray

La classe java.util.concurrent.atomic.AtomicIntegerArray encapsule un tableau de valeurs entières qui peuvent être mises à jour de manière atomique.

Il est possible d'utiliser un tableau d'objets de type AtomicInteger pour obtenir des fonctionnalités similaires mais l'utilisation d'un AtomicIntegerArray consomme moins de ressources mémoire.

Elle possède deux constructeurs :

Constructeur	Rôle
AtomicIntegerArray(int length)	Créer une nouvelle instance qui encapsule un tableau dont la taille est fournie en paramètres
AtomicIntegerArray(int[] array)	Créer une nouvelle instance qui encapsule un tableau initialisé avec une copie des éléments du tableau fourni en paramètre

Elle possède de nombreuses méthodes :

Méthode	Rôle
int accumulateAndGet(int i, int x, IntBinaryOperator accumulatorFunction)	Modifier la valeur de l'index précisé avec celle retournée par l'expression Lambda passée en paramètre qui sera invoquée avec la valeur courante et celle fournie. Renvoie la valeur après modification Depuis Java 8
int addAndGet(int i, int delta)	Ajouter delta à la valeur de l'index précisé et la renvoyer
boolean compareAndSet(int i, int expect, int update)	Mettre à jour de manière atomique la valeur courante de l'index précisé avec celle du paramètre update si la valeur courante est égale à la valeur du paramètre expect. Le booléen indique si la mise à jour a été effectuée
int decrementAndGet(int i)	Décrémenter la valeur de l'index précisé et la renvoyer
int get(int i)	Obtenir la valeur courante de l'index précisé
int getAndAccumulate(int i, int x, IntBinaryOperator accumulatorFunction)	Modifier la valeur de l'index précisé avec celle retournée par l'expression Lambda passée en paramètre qui sera invoquée avec la valeur courante et celle fournie. Renvoie la valeur avant la modification Depuis Java 8
int getAndAdd(int i, int delta)	Ajouter delta à la valeur de l'index précisé et renvoyer la valeur avant modification
int getAndDecrement(int i)	Décrémenter la valeur de l'index précisé et renvoyer la valeur avant incrémentation
int getAndIncrement(int i)	Incrémenter la valeur de l'index précisé et renvoyer la valeur avant incrémentation
int getAndSet(int i, int newValue)	Mettre à jour la valeur de l'index précisé et renvoyer la valeur avant modification
int getAndUpdate(int i, IntUnaryOperator updateFunction)	Modifier la valeur de l'index précisé avec celle retournée par l'expression Lambda passée en paramètre. Renvoie la valeur avant la modification Depuis Java 8
int incrementAndGet(int i)	Incrémenter la valeur de l'index précisé et la renvoyer
void lazySet(int i, int newValue)	Depuis Java 6
int length()	Renvoyer la taille du tableau
void set(int i, int newValue)	Remplacer la valeur de l'index précisé avec celle fournie
int updateAndGet(int i, IntUnaryOperator updateFunction)	Modifier la valeur de l'index précisé avec celle retournée par l'expression Lambda passée en paramètre. Renvoie la valeur après modification
boolean weakCompareAndSet(int i, int expect, int update)	

40.4.4.2. La classe AtomicLongArray

La classe java.util.concurrent.atomic.AtomicLongArray encapsule un tableau de valeurs entières longues qui peuvent être mises à jour de manière atomique

Elle possède deux constructeurs :

Constructeur	Rôle
AtomicLongArray(int length)	Créer une nouvelle instance qui encapsule un tableau dont la taille est fournie en paramètres

AtomicLongArray(long[] array)	Créer une nouvelle instance qui encapsule un tableau initialisé avec une copie des éléments du tableau fourni en paramètre

Elle possède de nombreuses méthodes :

Méthode	Rôle
long accumulateAndGet(int i, long x, LongBinaryOperator accumulatorFunction)	Modifier la valeur de l'index précisé avec celle retournée par l'expression Lambda passée en paramètre qui sera invoquée avec la valeur courante et celle fournie. Renvoie la valeur après modification Depuis Java 8
long addAndGet(int i, long delta)	Ajouter delta à la valeur de l'index précisé et la renvoyer
boolean compareAndSet(int i, long expect, long update)	Mettre à jour de manière atomique la valeur courante de l'index précisé avec celle du paramètre update si la valeur courante est égale à la valeur du paramètre expect. Le booléen indique si la mise à jour a été effectuée
long decrementAndGet(int i)	Décrémenter la valeur de l'index précisé et la renvoyer
long get(int i)	Obtenir la valeur courante de l'index précisé
long getAndAccumulate(int i, long x, LongBinaryOperator accumulatorFunction)	Modifier la valeur de l'index précisé avec celle retournée par l'expression Lambda passée en paramètre qui sera invoquée avec la valeur courante et celle fournie. Renvoie la valeur avant la modification Depuis Java 8
long getAndAdd(int i, long delta)	Ajouter delta à la valeur de l'index précisé et renvoyer la valeur avant modification
long getAndDecrement(int i)	Décrémenter la valeur de l'index précisé et renvoyer la valeur avant incrémentation
long getAndIncrement(int i)	Incrémenter la valeur de l'index précisé et renvoyer la valeur avant incrémentation
long getAndSet(int i, long newValue)	Mettre à jour la valeur de l'index précisé et renvoyer la valeur avant modification
long getAndUpdate(int i, LongUnaryOperator updateFunction)	Modifier la valeur de l'index précisé avec celle retournée par l'expression Lambda passée en paramètre. Renvoie la valeur avant la modification Depuis Java 8
long incrementAndGet(int i)	Incrémenter la valeur de l'index précisé et la renvoyer
void lazySet(int i, long newValue)	Depuis Java 6
int length()	Renvoyer la taille du tableau
void set(int i, long newValue)	Remplacer la valeur de l'index précisé avec celle fournie
long updateAndGet(int i, LongUnaryOperator updateFunction)	Modifier la valeur de l'index précisé avec celle retournée par l'expression Lambda passée en paramètre. Renvoie la valeur après modification
boolean weakCompareAndSet(int i, long expect, long update)	

40.4.4.3. La classe AtomicReferenceArray<E>

La classe java.util.concurrent.atomic.AtomicReferenceArray encapsule un tableau de références sur des objets qui peuvent être mises à jour de manière atomique.

Elle possède deux constructeurs :

Constructeur	Rôle
AtomicReferenceArray(E[] array)	Créer une nouvelle instance qui encapsule un tableau initialisé avec une copie des éléments du tableau fourni en paramètre
AtomicReferenceArray(int length)	Créer une nouvelle instance qui encapsule un tableau dont la taille est fournie en paramètres

Elle possède de nombreuses méthodes :

Méthode	Rôle
E accumulateAndGet(int i, E x, BinaryOperator<E> accumulatorFunction)	Modifier la valeur de l'index précisé avec celle retournée par l'expression Lambda passée en paramètre qui sera invoquée avec la valeur courante et celle fournie. Renvoie la valeur après modification Depuis Java 8
boolean compareAndSet(int i, E expect, E update)	Mettre à jour de manière atomique la valeur courante de l'index précisé avec celle du paramètre update si la valeur courante est égale à la valeur du paramètre expect. Le booléen indique si la mise à jour a été effectuée
E get(int i)	Obtenir la valeur courante de l'index précisé
E getAndAccumulate(int i, E x, BinaryOperator<E> accumulatorFunction)	Modifier la valeur de l'index précisé avec celle retournée par l'expression Lambda passée en paramètre qui sera invoquée avec la valeur courante et celle fournie. Renvoie la valeur avant la modification Depuis Java 8
E getAndSet(int i, E newValue)	Mettre à jour la valeur de l'index précisé et renvoyer la valeur avant modification
E getAndUpdate(int i, UnaryOperator<E> updateFunction)	Modifier la valeur de l'index précisé avec celle retournée par l'expression Lambda passée en paramètre. Renvoie la valeur avant la modification Depuis Java 8
void lazySet(int i, E newValue)	Depuis Java 6
int length()	Renvoyer la taille du tableau
void set(int i, E newValue)	Remplacer la valeur de l'index précisé avec celle fournie
E updateAndGet(int i, UnaryOperator<E> updateFunction)	Modifier la valeur de l'index précisé avec celle retournée par l'expression Lambda passée en paramètre. Renvoie la valeur après modification
boolean weakCompareAndSet(int i, E expect, E update)	

40.4.5. Les classes AtomicReference, AtomicMarkableReference et AtomicStampedReference

Les classes AtomicMarkableReference et AtomicStampedReference permettent de gérer de manière atomique respectivement une référence associée à un booléen ou à un entier.

40.4.5.1. La classe AtomicReference<V>

La classe java.util.concurrent.atomic.AtomicReference encapsule une référence sur un objet qui peut être mise à jour de manière atomique.

Elle possède deux constructeurs :

Constructeur	Rôle
AtomicReference ()	La valeur encapsulée est null
AtomicReference (V initialValue)	La valeur encapsulée est celle fournie en paramètre

Elle possède plusieurs méthodes :

Méthode	Rôle
V accumulateAndGet(V x, BinaryOperator<V> accumulatorFunction)	Modifier la valeur avec celle retournée par l'expression Lambda passée en paramètre qui sera invoquée avec la valeur courante et celle fournie. Renvoie la valeur après modification Depuis Java 8
boolean compareAndSet(V expect, V update)	Tenter de mettre à jour de manière atomique la valeur avec celle du paramètre update si la valeur courante est égale à celle du paramètre expect. Renvoie un booléen qui précise si la valeur a été modifiée lors des traitements.
V get()	Renvoyer la valeur courante
boolean getAndSet(boolean newValue)	Modifier la valeur avec celle en paramètre et renvoie la valeur avant modification
void lazySet(boolean newValue)	Depuis Java 6
void set(boolean newValue)	Modifier la valeur
String toString()	Renvoyer la valeur sous la forme d'une chaîne de caractères
boolean weakCompareAndSet(boolean expect, boolean update)	

40.4.5.2. La classe AtomicMarkableReference

La classe AtomicMarkableReference encapsule un booléen et une référence sur un objet de type V qui peuvent être modifiés de manière atomique.

Elle ne possède qu'un seul constructeur.

Constructeur	Rôle
AtomicMarkableReference (V initialRef, boolean initialMark)	Créer une nouvelle instance qui encapsule l'objet et la valeur booléenne fournis en paramètres

Elle possède plusieurs méthodes :

Méthode	Rôle
boolean attemptMark(V expectedReference, new newMark)	Modifier la valeur booléenne si la référence de l'instance de l'objet passé en paramètre est égale à celle encapsulée. Renvoie un booléen qui précise si l'opération a été effectuée
boolean compareAndSet(V expectedReference, V newReference, boolean expectedMark, boolean newMark)	Modifier l'instance encapsulée et la valeur entière si l'instance de l'objet (expectedReference) et la valeur entière (expectedMark) passées en paramètres sont égales à celles encapsulées. Renvoie un booléen qui précise si l'opération a été effectuée
V get(boolean[] markHolder)	
V getReference()	Renvoyer l'objet encapsulé

boolean isMark()	Renvoyer la valeur booléenne
void set(V newReference, boolean newMark)	Modifier de manière inconditionnelle l'instance et la valeur booléenne avec les valeurs fournies en paramètre
boolean weakCompareAndSet(V expectedReference, V newReference, boolean expectedMark, boolean newMark)	

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.atomic.AtomicMarkableReference;

public class TestAtomicMarkableReference {

  AtomicMarkableReference<String> amr = new AtomicMarkableReference<String>(
                                          "chaine1", false);

  class MonRunnable implements Runnable {

    @Override
    public void run() {

      amr.attemptMark("chaine1", true);
      System.out.println(Thread.currentThread().getName() + " : "
          + amr.getReference() + " : " + amr.isMarked());

      amr.compareAndSet("chaine1", "chaine2", true, false);
      System.out.println(Thread.currentThread().getName() + " : "
          + amr.getReference() + " : " + amr.isMarked());
    }
  }

  public static void main(String... args) {
    for (int i = 0; i < 5; i++)
      new Thread(new TestAtomicMarkableReference().new MonRunnable()).start();
  }
}
```

Résultat :

```
Thread-0 : chaine1 : true
Thread-0 : chaine2 : false
Thread-1 : chaine1 : true
Thread-1 : chaine2 : false
Thread-2 : chaine1 : true
Thread-2 : chaine2 : false
Thread-3 : chaine1 : true
Thread-3 : chaine2 : false
Thread-4 : chaine1 : true
Thread-4 : chaine2 : false
```

40.4.5.3. La classe AtomicStampedReference

La classe AtomicStampedReference encapsule un entier de type int et une référence sur un objet de type V qui peuvent être modifiées de manière atomique.

Elle ne possède qu'un seul constructeur.

Constructeur	Rôle
AtomicStampedReference(V initialRef, int initialStamp)	Créer une nouvelle instance qui encapsule l'objet et la valeur entière fournis en paramètre

Elle possède plusieurs méthodes :

Méthode	Rôle
boolean attemptStamp(V expectedReference, int newStamp)	Modifier la valeur entière si l'instance de l'objet passé en paramètre est égale à celle encapsulée. Renvoie un booléen qui précise si l'opération a été effectuée
boolean compareAndSet(V expectedReference, V newReference, int expectedStamp, int newStamp)	Modifier l'instance encapsulée et la valeur entière si l'instance de l'objet (expectedReference) et la valeur entière (expectedStamp) passées en paramètres sont égales à celles encapsulées. Renvoie un booléen qui précise si l'opération a été effectuée
V get(int[] stampHolder)	
V getReference()	Renvoyer l'objet encapsulé
int getStamp()	Renvoyer la valeur entière
void set(V newReference, int newStamp)	Modifier de manière inconditionnelle l'instance et la valeur entière avec les valeurs fournies en paramètre
boolean weakCompareAndSet(V expectedReference, V newReference, int expectedStamp, int newStamp)	

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.thread;

import java.util.concurrent.atomic.AtomicStampedReference;

public class TestAtomicStampedReference {

  AtomicStampedReference<String> asr = new AtomicStampedReference<String>(
                                       "chaine1", 1);

  class MonRunnable implements Runnable {

    @Override
    public void run() {

      asr.attemptStamp("chaine1", 2);
      System.out.println(Thread.currentThread().getName() + " : "
          + asr.getReference() + " : " + asr.getStamp());

      asr.compareAndSet("chaine1", "chaine2", 2, 3);
      System.out.println(Thread.currentThread().getName() + " : "
          + asr.getReference() + " : " + asr.getStamp());
    }
  }

  public static void main(String... args) {
    for (int i = 0; i < 5; i++)
      new Thread(new TestAtomicStampedReference().new MonRunnable()).start();
  }
}
```

Résultat :

```
Thread-0 : chaine1 : 2
Thread-1 : chaine1 : 2
Thread-2 : chaine1 : 2
Thread-0 : chaine2 : 3
Thread-2 : chaine2 : 3
Thread-1 : chaine2 : 3
Thread-3 : chaine1 : 2
Thread-3 : chaine2 : 3
Thread-4 : chaine1 : 2

Thread-4 : chaine2 : 3
```

40.4.6. Les classes AtomicIntegerFieldUpdater, AtomicLongFieldUpdater, AtomicReferenceFieldUpdater

La suite de cette section sera développée dans une version future de ce document

40.4.7. Les classes DoubleAccumulator et LongAccumulator

La suite de cette section sera développée dans une version future de ce document

40.4.8. Les classes DoubleAdder et LongAdder

La suite de cette section sera développée dans une version future de ce document

40.5. L'immutabilité et la copie défensive

La façon la plus sûre pour éviter des problèmes de concurrences d'accès est de ne partager entre plusieurs threads que des objets immuables. Un objet immuable est un objet dont l'état ne peut pas être modifié après son initialisation.

Certains objets peuvent rester immuables durant toute leur durée de vie mais l'état de la plupart des objets partagés changent au cours du temps. Chacune de ces modifications nécessite alors la création d'une nouvelle instance qui encapsulera de manière immuable le nouvel état.

Il ne reste plus qu'a remplacer la référence précédente par celle de la nouvelle instance : cette opération est garantie d'être atomique par la JVM.

Cette technique requiert donc potentiellement de nombreuses instances selon le nombre d'objets concernés et de modifications de leur état.

Ce principe de fonctionnement est le fondement de certains frameworks notamment Akka.

40.5.1. L'immutabilité

L'état d'un objet immuable ne peut pas être modifié après son initialisation. Chaque changement de son état implique la création d'une nouvelle instance qui encapsulera le nouvel état.

Un bon exemple dans l'API Java Core est la classe java.lang.String.

La section L'écriture d'une classe dont les instances seront immuables du chapitre Les techniques de développement spécifiques à Java en détaille la mise en oeuvre.

40.5.2. La copie défensive

Lorsqu'un objet est passé en paramètre ou en retour d'une méthode, celle-ci n'a aucun moyen :

- de connaitre les autres objets qui possèdent une référence sur l'instance
- si l'objet est mutable, de savoir qu'elles modifications seront faites sur l'objet à partir de ces références

Un objet mutable est un objet dont l'état peut être modifié après sa construction.

Exemples d'objets mutables : Date, StringBuilder, les collections, les tableaux, ...

Exemples d'objets immuables : String, Integer, ...

Une classe peut avoir un champ qui est un objet mutable. Il y a alors deux cas de figure pour modifier l'état de cet objet :

- l'état est uniquement modifié par la classe qui encapsule l'objet car elle protège son accès
- l'objet peut être renvoyé à un appelant qui peut alors modifier son état

Une solution pour mettre en oeuvre le premier cas est d'utiliser la copie défensive.

Pour préserver les règles de l'encapsulation, la copie défensive doit être mise en oeuvre lorsque :

- un objet mutable est passé en paramètre d'un setter ou d'un constructeur
- un objet mutable est retourné par un getter

Si ce n'est pas le cas, il est possible que l'objet encapsulé soit modifié en dehors de la classe.

La seule façon de garantir que seule la classe aura une référence sur un objet passé en paramètre ou retourné est d'utiliser la copie défensive. Elle consiste à renvoyer une copie d'un objet à son appelant ou à créer une copie d'un objet reçu en paramètre.

Le but de la copie défensive est de travailler sur une copie d'un objet plutôt que sur l'objet original pour éviter que l'état de cet objet ne soit modifié en dehors de la classe de manière volontaire ou non.

La copie défensive d'un paramètre peut ne pas être utilisée que pour permettre l'immutabilité de la classe. Lorsqu'un objet est passé en paramètre d'une méthode ou d'un constructeur, il faut savoir si l'objet est mutable et si c'est le cas si c'est acceptable que l'objet puisse être modifié en dehors de la classe. Si la réponse est non, alors il faut stocker une copie défensive plutôt qu'une référence sur l'objet original passé en paramètre.

Lorsqu'un objet est retournée d'une méthode, il faut se poser les mêmes questions et en fonctions des réponses utiliser ou non la copie défensive.

L'exemple ci-dessous est un bean qui stocke son état dans une collection. Il ne propose que deux méthodes pour ajouter un élément et obtenir une collection qui contient les éléments.

Exemple :

```
package fr.jmdoudoux.dej.thread;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class TestMaListe {

  List<String> list = new ArrayList<String>();

  public void ajouter(String s) {
    list.add(s);
  }

  public List<String> get() {
    return list;
  }

  public static void main(String[] args) {
    TestMaListe maListe = new TestMaListe();
    maListe.ajouter("1");
    maListe.ajouter("2");
    System.out.println("liste : "
        + Arrays.deepToString(maListe.get().toArray()));
    List<String> liste = maListe.get();
    liste.add("3");
    System.out.println("liste : "
        + Arrays.deepToString(maListe.get().toArray()));
  }
}
```

Résultat :

```
liste : [1, 2]
liste : [1, 2, 3]
```

L'état de l'objet de type List peut être modifié à l'extérieur de la classe puisque c'est l'instance de type List qui est directement retournée : rien n'empêche d'invoquer ses méthodes pour ajouter un élément comme dans l'exemple ci-dessus.

Pour garder un bon contrôle sur l'état de la collection, il est possible de renvoyer une copie de l'instance de type Liste. Dans l'exemple, ci-dessous c'est une copie non modifiable qui est retournée. L'instance retournée peut être utilisée pour obtenir un élément ou parcourir tout ou partie des éléments mais il n'est pas possible d'ajouter, de modifier ou de supprimer un élément.

Exemple :

```
package fr.jmdoudoux.dej.thread;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class TestMaListe {

  List<String> list = new ArrayList<String>();

  public void ajouter(String s) {
    list.add(s);
  }

  public List<String> get() {
    return Collections.unmodifiableList(list);
  }

  public static void main(String[] args) {
    TestMaListe maListe = new TestMaListe();
    maListe.ajouter("1");
    maListe.ajouter("2");
    System.out.println("liste : "
        + Arrays.deepToString(maListe.get().toArray()));
    List<String> liste = maListe.get();
    liste.add("3");
    System.out.println("liste : "
        + Arrays.deepToString(maListe.get().toArray()));
  }
}
```

Résultat :

```
liste : [1, 2]
Exception in thread "main" java.lang.UnsupportedOperationException
      at java.util.Collections$UnmodifiableCollection.add(Collections.java:1016)
      at fr.jmdoudoux.dej.thread.TestMaListe.main(TestMaListe.java:27)
```

L'exemple suivant utilise un bean qui possède une propriété de type Date.

Exemple :

```
package fr.jmdoudoux.dej.thread;

import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;

public class Personne {

  private String nom;
  private Date   dateNaissance;

  public Personne(String nom, Date dateNaissance) {
    super();
    this.nom = nom;
    this.dateNaissance = dateNaissance;
  }

  public String getNom() {
    return nom;
  }

  public Date getDateNaissance() {
    return dateNaissance;
  }

  public static void main(String[] args) {

    Calendar calendar = new GregorianCalendar(2015, 11, 25);
    Date dateNaiss = calendar.getTime();

    Personne personne = new Personne("Nom1", dateNaiss);
    System.out.println("date de naissance = " + personne.getDateNaissance());
    dateNaiss.setYear(114);
    System.out.println("date de naissance = " + personne.getDateNaissance());
  }
}
```

Résultat :

```
date de naissance = Fri Dec 25 00:00:00 CET 2015
date de naissance = Fri Dec 25 00:00:00 CET 2014
```

Pour éviter cette situation, il faut créer une copie défensive des objets mutables passés en paramètres des constructeurs et des setters.

Exemple :

```
package fr.jmdoudoux.dej.thread;

import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;
```

```
public class Personne {

  private String nom;
  private Date   dateNaissance;

  public Personne(String nom, Date dateNaissance) {
    super();
    this.nom = nom;
    this.dateNaissance = new Date(dateNaissance.getTime());
  }

  public String getNom() {
    return nom;
  }

  public Date getDateNaissance() {
    return dateNaissance;
  }

  public static void main(String[] args) {

    Calendar calendar = new GregorianCalendar(2015, 11, 25);
    Date dateNaiss = calendar.getTime();

    Personne personne = new Personne("Nom1", dateNaiss);
    System.out.println("date de naissance = " + personne.getDateNaissance());
    dateNaiss.setYear(114);
    System.out.println("date de naissance = " + personne.getDateNaissance());
  }
}
```

Résultat :

```
date de naissance = Fri Dec 25 00:00:00 CET 2015
date de naissance = Fri Dec 25 00:00:00 CET 2015
```

Remarque : si des contrôles doivent être faits sur ces paramètres, il est nécessaire de faire les copies défensives puis les contrôles sur ces copies. Ceci afin d'éviter qu'un autre thread ne vienne modifier l'état des objets pendant la réalisation des copies. Ce phénomène est désigné par l'acronyme TOCTOU : Time Of Check to Time of Use.

Cela n'empêche pas un appelant de modifier l'objet mutable si c'est directement lui qui est retourné par un getter.

Exemple :

```
package fr.jmdoudoux.dej.thread;

import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;

public class Personne {

  private String nom;
  private Date   dateNaissance;

  public Personne(String nom, Date dateNaissance) {
    super();
    this.nom = nom;
    this.dateNaissance = new Date(dateNaissance.getTime());
  }

  public String getNom() {
    return nom;
  }

  public Date getDateNaissance() {
    return dateNaissance;
  }
  public static void main(String[] args) {

    Calendar calendar = new GregorianCalendar(2015, 11, 25);
    Date dateNaiss = calendar.getTime();

    Personne personne = new Personne("Nom1", dateNaiss);
    System.out.println("date de naissance = " + personne.getDateNaissance());
    dateNaiss.setYear(114);
    System.out.println("date de naissance = " + personne.getDateNaissance());
    dateNaiss = personne.getDateNaissance();
    dateNaiss.setYear(114);
    System.out.println("date de naissance = " + personne.getDateNaissance());
  }
}
```

Résultat :

```
date de naissance = Fri Dec 25 00:00:00 CET 2015
date de naissance = Fri Dec 25 00:00:00 CET 2015
date de naissance = Thu Dec 25 00:00:00 CET 2014
```

Pour éviter cela, il faut aussi que les getters renvoient une copie défensive de l'objet mutable.

Exemple :

```
package fr.jmdoudoux.dej.thread;

import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;

public class Personne {

  private String nom;
  private Date   dateNaissance;

  public Personne(String nom, Date dateNaissance) {
    super();
    this.nom = nom;
    this.dateNaissance = new Date(dateNaissance.getTime());
  }

  public String getNom() {
    return nom;
  }

  public Date getDateNaissance() {
    return new Date(dateNaissance.getTime());
  }

  public static void main(String[] args) {

    Calendar calendar = new GregorianCalendar(2015, 11, 25);
    Date dateNaiss = calendar.getTime();

    Personne personne = new Personne("Nom1", dateNaiss);
    System.out.println("date de naissance = " + personne.getDateNaissance());
    dateNaiss.setYear(114);
    System.out.println("date de naissance = " + personne.getDateNaissance());
    dateNaiss = personne.getDateNaissance();
    dateNaiss.setYear(114);
    System.out.println("date de naissance = " + personne.getDateNaissance());
  }
}
```

Résultat :

```
date de naissance = Fri Dec 25 00:00:00 CET 2015
date de naissance = Fri Dec 25 00:00:00 CET 2015
date de naissance = Fri Dec 25 00:00:00 CET 2015
```

Il est aussi important de bien choisir la façon dont la copie défensive va être effectuée.

Par exemple, dans le cas de classe Date, trois solutions sont envisageables :

- invoquer un constructeur (comme utilisé dans l'exemple ci-dessus)
- cloner l'objet. Cette solution ne devrait être mise en oeuvre que pour des objets dont le type est final pour éviter que l'instance obtenue soit un sous-type.
- stocker la date en interne sous la forme d'un entier long obtenu par un appel à la méthode getTime() de la classe Date et retourner une nouvelle instance de la Date créée avec le constructeur qui attend en paramètre un entier long

Comme les tableaux sont des objets et qu'un tableau dont la taille est différente de zéro est mutable, ils peuvent aussi être concernés par la mise en oeuvre de la copie défensive.

La méthode copyOf() de la classe Arrays permet d'obtenir une copie d'un tableau.

Exemple :

```
package fr.jmdoudoux.dej.thread;

import java.util.Arrays;

public class Personne {

  private String nom;
  private int[]  valeurs;

  public Personne(String nom, int[] valeurs) {
    super();
    this.nom = nom;
    this.valeurs = Arrays.copyOf(valeurs, valeurs.length);
  }

  public String getNom() {
    return nom;
  }

  public int[] getValeurs() {
    return Arrays.copyOf(valeurs, valeurs.length);
  }
}
```

Il ne faut pas utiliser systématiquement la copie défensive :

- il y a des cas où l'objet peut être partagé
- elle ne concerne que des objets qui sont mutables

L'utilisation de la copie défensive peut aussi avoir un coût en termes de performance car cela peut engendrer la création de nombreux objets, dont le coût peut être plus ou moins important, qui devront de surcroît être récupérés par le ramasse-miettes.

Partie 6 : Le développement des interfaces graphiques

Les interfaces graphiques assurent le dialogue entre les utilisateurs et une application.

Dans un premier temps, Java proposait l'API AWT pour créer des interfaces graphiques. Depuis, Java propose une nouvelle API nommée Swing. Ces deux API peuvent être utilisées pour développer des applications ou des applets. Face aux problèmes de performance de Swing, IBM a créé sa propre bibliothèque nommée SWT utilisée pour développer l'outil Eclipse. La vélocité de cette application favorise une utilisation grandissante de cette bibliothèque.

Cette partie contient les chapitres suivants :

- Le graphisme : entame une série de chapitres sur les interfaces graphiques en détaillant les objets et méthodes de base pour le graphisme
- Les éléments d'interfaces graphiques de l'AWT : recense les différents composants qui sont fournis dans la bibliothèque AWT
- La création d'interfaces graphiques avec AWT : indique comment réaliser des interfaces graphiques avec l'AWT
- L'interception des actions de l'utilisateur : détaille les mécanismes qui permettent de réagir aux actions de l'utilisateur via une interface graphique
- Le développement d'interfaces graphiques avec SWING : indique comment réaliser des interfaces graphiques avec Swing
- Le développement d'interfaces graphiques avec SWT : indique comment réaliser des interfaces graphiques avec SWT
- JFace : présente l'utilisation de ce framework facilitant le développement d'applications utilisant SWT

41. Le graphisme

Chapitre 41

Niveau : 🟩🟩⬜⬜⬜⬜ Elémentaire

La classe Graphics contient les outils nécessaires pour dessiner. Cette classe est abstraite et elle ne possède pas de constructeur public : il n'est pas possible de construire des instances de graphics nous même. Les instances nécessaires sont fournies par le système d'exploitation qui instanciera grâce à la machine virtuelle une sous-classe de Graphics dépendante de la plate-forme utilisée.

Ce chapitre contient une section :

♦ Les opérations sur le contexte graphique

41.1. Les opérations sur le contexte graphique

41.1.1. Le tracé de formes géométriques

A l'exception des lignes, toutes les formes peuvent être dessinées vides (méthode drawXXX) ou pleines (fillXXX).

La classe Graphics possède de nombreuses méthodes qui permettent de réaliser des dessins.

Méthode	Rôle
drawRect(x, y, largeur, hauteur) fillRect(x, y, largeur, hauteur)	dessiner un carré ou un rectangle
drawRoundRect(x, y, largeur, hauteur, hor_arr,ver_arr) fillRoundRect(x, y, largeur, hauteur, hor_arr, ver_arr)	dessiner un carré ou un rectangle aux angles arrondis
drawLine(x1, y1, x2, y2)	Dessiner une ligne
drawOval(x, y, largeur, hauteur) fillOval(x, y, largeur, hauteur)	dessiner un cercle ou une ellipse en spécifiant le rectangle dans lequel ils s'inscrivent

drawPolygon(int[], int[], int)
fillPolygon(int[], int[], int)
	Dessiner un polygone ouvert ou fermé. Les deux premiers paramètres sont les coordonnées en abscisses et en ordonnées. Le dernier paramètre est le nombre de points du polygone.
	Pour dessiner un polygone fermé il faut joindre le dernier point au premier.

	Exemple (code Java 1.1) :
	`int[] x={10,60,100,80,150,10};`
	`int[] y={15,15,25,35,45,15};`
	`g.drawPolygon(x,y,x.length);`
	`g.fillPolygon(x,y,x.length);`
	Il est possible de définir un objet Polygon.
	Exemple (code Java 1.1) :
	`int[] x={10,60,100,80,150,10};`
	`int[] y={15,15,25,35,45,15};`
	`Polygon p = new Polygon(x, y,x.length );`
	`g.drawPolygon(p);`

drawArc(x, y, largeur, hauteur, angle_deb, angle_bal) fillArc(x, y, largeur, hauteur, angle_deb, angle_bal);	dessiner un arc d'ellipse inscrit dans un rectangle ou un carré. L'angle 0 se situe à 3 heures. Il faut indiquer l'angle de début et l'angle balayé

41.1.2. Le tracé de texte

La méthode drawString() permet d'afficher un texte aux coordonnées précisées

Exemple (code Java 1.1) :

```
g.drawString(texte, x, y );
```

Pour afficher des nombres de type int ou float, il suffit de les concaténer à une chaîne éventuellement vide avec l'opérateur +.

41.1.3. L'utilisation des fontes

La classe Font permet d'utiliser une police de caractères particulière pour afficher un texte.

Exemple (code Java 1.1) :

```
Font fonte = new Font(" TimesRoman ",Font.BOLD,30);
```

Le constructeur de la classe Font est Font(String, int, int). Les paramètres sont : le nom de la police, le style (BOLD, ITALIC, PLAIN ou 0,1,2) et la taille des caractères en points.

Pour associer plusieurs styles, il suffit de les additionner

Exemple (code Java 1.1) :

```
Font.BOLD + Font.ITALIC
```

Si la police spécifiée n'existe pas, Java prend la fonte par défaut même si une autre a été spécifiée précédemment. Le style et la taille seront tout de même adaptés. La méthode getName() de la classe Font retourne le nom de la fonte.

La méthode setFont() de la classe Graphics permet de changer la police d'affichage des textes

Exemple (code Java 1.1) :

```
Font fonte = new Font("
TimesRoman ",Font.BOLD,30);
g.setFont(fonte);
g.drawString("bonjour",50,50);
```

Les polices suivantes sont utilisables : Dialog, Helvetica, TimesRoman, Courier, ZapfDingBats

41.1.4. La gestion de la couleur

La méthode setColor() permet de fixer, à postériori, la couleur des éléments graphiques créés dans l'instance de type Graphics.

Exemple (code Java 1.1) :

```
g.setColor(Color.black); //(green, blue, red, white, black, ...)
```

41.1.5. Le chevauchement de figures graphiques

Si 2 surfaces de couleur différentes se superposent, alors la dernière dessinée recouvre la précédente sauf si on invoque la méthode setXORMode(). Dans ce cas, la couleur de l'intersection prend une autre couleur. L'argument à fournir est une couleur alternative. La couleur d'intersection représente une combinaison de la couleur originale et de la couleur alternative.

41.1.6. L'effacement d'une aire

La méthode clearRect(x1, y1, x2, y2) dessine un rectangle dans la couleur de fond courante.

41.1.7. La copie d'une aire rectangulaire

La méthode copyArea(x1, y1, x2, y2, dx, dy) permet de copier une aire rectangulaire. Les paramètres dx et dy permettent de spécifier un décalage en pixels de la copie par rapport à l'originale.

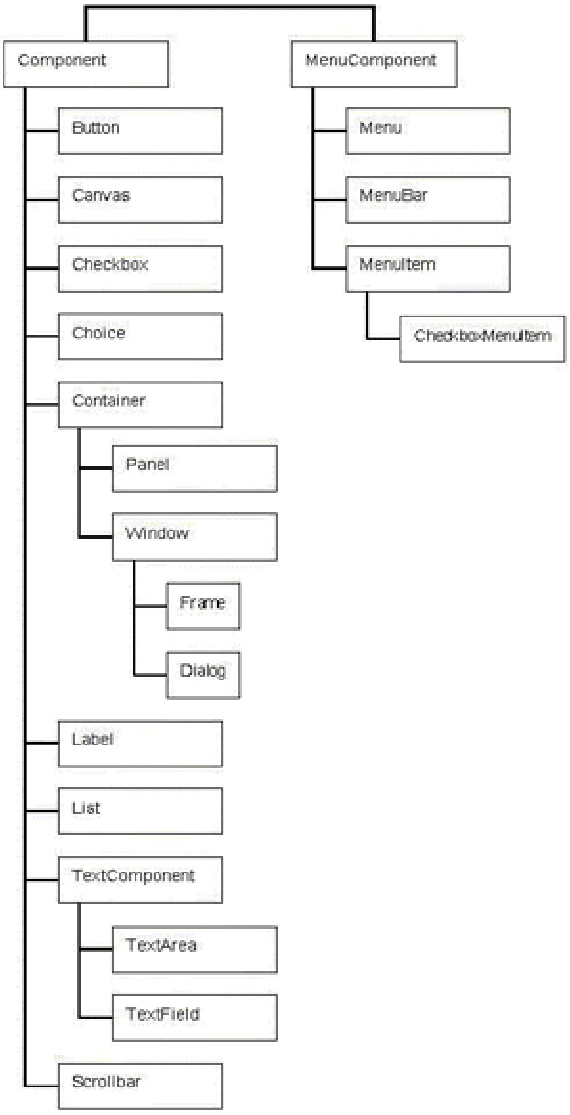
42. Les éléments d'interfaces graphiques de l'AWT

Chapitre 42

Niveau : 🟩🟩🟨□□□ Intermédiaire

API legacy
Ce chapitre est conservé pour des raisons historiques

Les classes du toolkit AWT (Abstract Windows Toolkit) permettent d'écrire des interfaces graphiques indépendantes du système d'exploitation sur lesquel elles vont fonctionner. Cette librairie utilise le système graphique de la plate-forme d'exécution (Windows, MacOS, X-Window) pour afficher les objets graphiques. Le toolkit contient des classes décrivant les composants graphiques, les polices, les couleurs et les images.



Le diagramme ci-dessus définit une vue partielle de la hiérarchie des classes (les relations d'héritage) qu'il ne faut pas confondre avec la hiérarchie interne à chaque application qui définit l'imbrication des différents composants graphiques.

Les deux classes principales d'AWT sont Component et Container. Chaque type d'objet de l'interface graphique est une classe dérivée de Component. La classe Container, qui hérite de Component est capable de contenir d'autres objets graphiques (tout objet dérivant de Component).

Ce chapitre contient plusieurs sections :

- Les composants graphiques
- La classe Component
- Les conteneurs
- Les menus
- La classe java.awt.Desktop

42.1. Les composants graphiques

Pour utiliser un composant, il faut créer un nouvel objet représentant le composant et l'ajouter à un conteneur existant grâce à la méthode add().

Exemple (code Java 1.1) : ajout d'un bouton dans une applet (Applet hérite de Panel)

```
import java.applet.*;
import java.awt.*;

public class AppletButton extends Applet {

   Button b = new Button(" Bouton ");

   public void init() {
      super.init();
      add(b);
   }
}
```

42.1.1. Les étiquettes

Il faut utiliser un objet de la classe java.awt.Label

Exemple (code Java 1.1) :

```
Label la = new Label( );
la.setText("une etiquette");
// ou Label la = new Label("une etiquette");
```

Il est possible de créer un objet de la classe java.awt.Label en précisant l'alignement du texte

Exemple (code Java 1.1) :

```
Label la = new Label("etiquette", Label.RIGHT);
```

Le texte à afficher et l'alignement peuvent être modifiés dynamiquement lors de l'exécution :

Exemple (code Java 1.1) :

```
la.setText("nouveau texte");
la.setAlignment(Label.LEFT);
```

42.1.2. Les boutons

Il faut utiliser un objet de la classe java.awt.Button

Cette classe possède deux constructeurs :

Constructeur	Rôle
Button()	
Button(String)	Permet de préciser le libellé du bouton

Exemple (code Java 1.1) :

```
Button bouton = new Button();
bouton.setLabel("bouton");
// ou Button bouton = new Button("bouton");
```

Le libellé du bouton peut être modifié dynamiquement grâce à la méthode setLabel() :

Exemple (code Java 1.1) :

```
bouton.setLabel("nouveau libellé");
```

42.1.3. Les panneaux

Les panneaux sont des conteneurs qui permettent de rassembler des composants et de les positionner grâce à un gestionnaire de présentation. Il faut utiliser un objet de la classe java.awt.Panel.

Par défaut le gestionnaire de présentation d'un panel est de type FlowLayout.

Constructeur	Rôle
Panel()	Créer un panneau avec un gestionnaire de présentation de type FlowLayout
Panel(LayoutManager)	Créer un panneau avec le gestionnaire précisé en paramètre

Exemple (code Java 1.1) :

```
Panel p = new Panel();
```

L'ajout d'un composant au panel se fait grâce à la méthode add().

Exemple (code Java 1.1) :

```
p.add(new Button("bouton"));
```

42.1.4. Les listes déroulantes (combobox)

Il faut utiliser un objet de la classe java.awt.Choice

Cette classe ne possède qu'un seul constructeur sans paramètres.

Exemple (code Java 1.1) :

```
Choice maCombo = new Choice();
```

Les méthodes add() et addItem() permettent d'ajouter des éléments à la combobox.

Exemple (code Java 1.1) :

```
maCombo.addItem("element 1");
// ou maCombo.add("element 2");
```

Plusieurs méthodes permettent la gestion des sélections :

Méthodes	Rôle

void select(int);	sélectionner un élément par son indice : le premier élément correspond à l'indice 0.
	Une exception IllegalArgumentException est levée si l'indice ne correspond pas à un élément.
	Exemple (code Java 1.1) :
	`maCombo.select(0);`

void select(String);	sélectionner un élément par son contenu
	Aucune exception n'est levée si la chaîne de caractères ne correspond à aucun élément : l'élément sélectionné ne change pas.
	Exemple (code Java 1.1) :
	`maCombo.select("element 1");`

int countItems( );	déterminer le nombre d'éléments de la liste. La méthode countItems() permet d'obtenir le nombre d'éléments de la combobox.
	Exemple (code Java 1.1) :
	`int n;`
	`n=maCombo.countItems();`
	Java 1.1 il faut utiliser getItemCount() à la place
	Exemple (code Java 1.1) :
	`int n;`
	`n=maCombo.getItemCount();`

String getItem(int);	lire le contenu de l'élément d'indice n
	Exemple (code Java 1.1) :
	`String c = new String( );`
	`c = maCombo.getItem(n);`

String getSelectedItem();	déterminer le contenu de l'élément sélectionné
	Exemple (code Java 1.1) :
	`String s = new String( );`
	`s = maCombo.getSelectedItem();`

int getSelectedIndex( );	déterminer l'index de l'élément sélectionné
	Exemple (code Java 1.1) :
	`int n;`
	`n=maCombo.getSelectedIndex();`

42.1.5. La classe TextComponent

La classe TextComponent est la classe mère des classes qui permettent l'édition de texte : TextArea et TextField.

Elle définit un certain nombre de méthodes dont ces classes héritent.

Méthodes	Rôle
String getSelectedText( );	Renvoie le texte sélectionné
int getSelectionStart( );	Renvoie la position de début de sélection
int getSelectionEnd( );	Renvoie la position de fin de sélection
String getText( );	Renvoie le texte contenu dans l'objet
boolean isEditable( );	Retourne un booléen indiquant si le texte est modifiable
void select(int start, int end);	Sélection des caractères situés entre start et end
void selectAll( );	Sélection de tout le texte
void setEditable(boolean b);	Autoriser ou interdire la modification du texte
void setText(String s);	Définir un nouveau texte

42.1.6. Les champs de texte

Il faut déclarer un objet de la classe java.awt.TextField

Il existe plusieurs constructeurs :

Constructeurs	Rôle
TextField();	
TextField(int);	prédétermination du nombre de caractères à saisir
TextField(String);	avec texte par défaut
TextField(String, int);	avec texte par défaut et nombre de caractères à saisir

Cette classe possède quelques méthodes utiles :

Méthodes	Rôle

String getText( )	lecture de la chaîne saisie
	Exemple (code Java 1.1) :
	`String saisie = new String();`
	`saisie = tf.getText( );`

int getColumns( )	lecture du nombre de caractères prédéfini
	Exemple (code Java 1.1) :
	`int i;`
	`i = tf.getColumns( );`

void setEchoCharacter()	pour la saisie d'un mot de passe : remplace chaque caractère saisi par celui fourni en paramètre
	Exemple (code Java 1.1) :
	`tf.setEchoCharacter('*');`
	`TextField tf = new TextField(10);`
	Java 1.1 il faut utiliser la méthode setEchoChar()
	Exemple (code Java 1.1) :
	`tf.setEchoChar('*');`

42.1.7. Les zones de texte multilignes

Il faut déclarer un objet de la classe java.awt.TextArea

Il existe plusieurs constructeurs :

Constructeur	Rôle
TextArea()	
TextArea(int, int)	avec prédétermination du nombre de lignes et de colonnes
TextArea(String)	avec texte par défaut
TextArea(String, int, int)	avec texte par défaut et taille

Les principales méthodes sont :

Méthodes	Rôle

String getText()	lecture du contenu intégral de la zone de texte
	Exemple (code Java 1.1) :
	`String contenu = new String;`
	`contenu = ta.getText( );`

String getSelectedText( )	lecture de la portion de texte sélectionnée
	Exemple (code Java 1.1) :
	`String contenu = new String;`
	`contenu = ta.getSelectedText( );`

int getRows()	détermination du nombre de lignes
	Exemple (code Java 1.1) :
	`int n;`
	`n = ta.getRows( );`

int getColumns( )	détermination du nombre de colonnes
	Exemple (code Java 1.1) :
	`int n;`
	`n = ta.getColumns( );`

void insertText(String, int)	insertion de la chaîne à la position fournie
	Exemple (code Java 1.1) :
	`String text =`
	`new String("texte inséré");`
	`int n =10;`
	`ta.insertText(text,n);`
	Java 1.1 Il faut utiliser la méthode insert()
	Exemple (code Java 1.1) :
	`String text = new`
	`String("texte inséré");`
	`int n =10;`
	`ta.insert(text,n);`

void setEditable(boolean)	Autoriser la modification
	Exemple (code Java 1.1) :
	`ta.setEditable(False);`
	`//texte non modifiable`

void appendText(String)	Ajouter le texte transmis au texte existant
	Exemple (code Java 1.1) :
	`ta.appendTexte(String text);`
	Java 1.1 Il faut utiliser la méthode append()

void replaceText(String, int, int)	Remplacer par text le texte entre les positions start et end
	Exemple (code Java 1.1) :
	`ta.replaceText(text, 10, 20);`
	Java 1.1 il faut utiliser la méthode replaceRange()

42.1.8. Les listes

Il faut déclarer un objet de la classe java.awt.List.

Il existe plusieurs constructeurs :

Constructeur	Rôle
List( )	
List(int)	Permet de préciser le nombre de lignes affichées
List(int, boolean)	Permet de préciser le nombre de lignes affichées et l'indicateur de sélection multiple

Les principales méthodes sont :

Méthodes	Rôle

void addItem(String)	ajouter un élément
	Exemple (code Java 1.1) :
	`li.addItem("nouvel element");`
	`// ajout en fin de liste`
	Java 1.1 il faut utiliser la méthode add()

void addItem(String, int)	insérer un élément à un certain emplacement : le premier élément est en position 0
	Exemple (code Java 1.1) :
	`li.addItem("ajout ligne",2);`
	Java 1.1 il faut utiliser la méthode add()

void delItem(int)	retirer un élément de la liste
	Exemple (code Java 1.1) :
	`li.delItem(0);`
	`// supprime le premier element`
	Java 1.1 il faut utiliser la méthode remove()

void delItems(int, int)	supprimer plusieurs éléments consécutifs entre les deux indices
	Exemple (code Java 1.1) :
	`li.delItems(1, 3);`
	Java 1.1 cette méthode est deprecated

void clear()	effacement complet du contenu de la liste
	Exemple (code Java 1.1) :
	`li.clear( );`
	Java 1.1 il faut utiliser la méthode removeAll()

void replaceItem(String, int)	remplacer un élément
	Exemple (code Java 1.1) :
	`li.replaceItem(`
	`"ligne remplacee", 1);`

int countItems()	nombre d'éléments de la liste
	Exemple (code Java 1.1) :
	`int n;`
	`n = li.countItems( );`
	Java 1.1 il faut utiliser la méthode getItemCount()

int getRows()	nombre de lignes de la liste
	Exemple (code Java 1.1) :
	`int n;`
	`n = li.getRows( );`

String getItem(int)	contenu d'un élément
	Exemple (code Java 1.1) :
	`String text = new String( );`
	`text = li.getItem(1);`

void select(int)	sélectionner un élément
	Exemple (code Java 1.1) :
	`li.select(0);`

setMultipleSelections(boolean)	déterminer si la sélection multiple est autorisée
	Exemple (code Java 1.1) :
	`li.setMultipleSelections(true);`
	Java 1.1 il faut utiliser la méthode setMultipleMode()

void deselect(int)	désélectionner un élément
	Exemple (code Java 1.1) :
	`li.deselect(0);`

int getSelectedIndex( )	déterminer l'élément sélectionné en cas de sélection simple : renvoie l'indice ou -1 si aucun élément n'est sélectionné
	Exemple (code Java 1.1) :
	`int i;`
	`i = li.getSelectedIndex();`

int[] getSelectedIndexes( )	déterminer les éléments sélectionnés en cas de sélection multiple
	Exemple (code Java 1.1) :
	`int i[]=li.getSelectedIndexes();`

String getSelectedItem( )	déterminer le contenu en cas de sélection simple : renvoie le texte ou null si pas de sélection
	Exemple (code Java 1.1) :
	`String texte = new String( );`
	`texte = li.getSelectedItem( );`

String[] getSelectedItems()	déterminer les contenus des éléments sélectionnés en cas de sélection multiple : renvoie les textes sélectionnés ou null si pas de sélection
	Exemple (code Java 1.1) :
	`String texte[ ] =`
	`li.getSelectedItems();`
	`for (i = 0 ; i < texte.length(); i++)`
	`  System.out.println(texte[i]);`

boolean isSelected(int)	déterminer si un élément est sélectionné
	Exemple (code Java 1.1) :
	`boolean selection;`
	`selection = li.isSelected(0);`
	Java 1.1 il faut utiliser la méthode isIndexSelect()

int getVisibleIndex()	renvoie l'index de l'entrée en haut de la liste
	Exemple (code Java 1.1) :
	`int top = li.getVisibleIndex();`

void makeVisible(int)	assure que l'élément précisé sera visible Exemple (code Java 1.1) :

	`li.makeVisible(10);`

Exemple (code Java 1.1) :

```
import java.awt.*;

class TestList {

  static public void main (String arg [ ]) {

    Frame frame = new Frame("Une liste");

    List list = new List(5,true);
    list.add("element 0");
    list.add("element 1");
    list.add("element 2");
    list.add("element 3");
    list.add("element 4");

    frame.add(List);
    frame.pack();
    frame.show();
  }
}
```

42.1.9. Les cases à cocher

Il faut déclarer un objet de la classe java.awt.Checkbox

Il existe plusieurs constructeurs :

Constructeur	Rôle
Checkbox()	
Checkbox(String)	avec une étiquette
Checkbox(String,boolean)	avec une étiquette et un état
Checkbox(String,CheckboxGroup, boolean)	avec une étiquette, dans un groupe de cases à cocher et un état

Les principales méthodes sont :

Méthodes	Rôle

void setLabel(String)	modifier l'étiquette
	Exemple (code Java 1.1) :
	`cb.setLabel( "libelle de la case : " );`

void setState(boolean)	fixer l'état
	Exemple (code Java 1.1) :
	`cb.setState( true );`

boolean getState( )	consulter l'état de la case Exemple (code Java 1.1) :

	`boolean etat;`
	`etat = cb.getState( );`

String getLabel( )	lire l'étiquette de la case
	Exemple (code Java 1.1) :
	`String commentaire = new String( );`
	`commentaire = cb.getLabel( );`

42.1.10. Les boutons radio

Déclarer un objet de la classe java.awt.CheckboxGroup

Exemple (code Java 1.1) :

```
CheckboxGroup rb;
Checkbox cb1 = new Checkbox(" etiquette 1 ", rb, etat1_boolean);
Checkbox cb2 = new Checkbox(" etiquette 2 ", rb, etat1_boolean);
Checkbox cb3 = new Checkbox(" etiquette 3 ", rb, etat1_boolean);
```

Les principales méthodes sont :

Méthodes	Rôle
Checkbox getCurrent()	retourne l'objet Checkbox correspondant à la réponse sélectionnée Java 1.1 il faut utiliser la méthode getSelectedCheckbox()
void setCurrent(Checkbox)	Coche le bouton radio passé en paramètre Java 1.1 il faut utiliser la méthode setSelectedCheckbox()

42.1.11. Les barres de défilement

Il faut déclarer un objet de la classe java.awt.Scrollbar

Il existe plusieurs constructeurs :

Constructeur	Rôle
Scrollbar( )	
Scrollbar(orientation)	
Scrollbar(orientation, valeur_initiale, visible, min, max)	

- orientation : Scrollbar.VERTICALE ou Scrollbar.HORIZONTAL
- valeur_initiale : position du curseur à la création
- visible : taille de la partie visible de la zone défilante
- min : valeur minimale associée à la barre
- max : valeur maximale associée à la barre

Les principales méthodes sont :

Méthodes	Rôle

sb.setValues(int,int,int,int)	mise à jour des paramètres de la barre
	Exemple (code Java 1.1) :
	`sb.setValues(`
	`valeur, visible,`
	`minimum, maximum );`

void setValue(int)	modifier la valeur courante
	Exemple (code Java 1.1) :
	`sb.setValue(10);`

int getMaximum( );	lecture du maximum
	Exemple (code Java 1.1) :
	`int max =`
	`sb.getMaximum( );`

int getMinimum( );	lecture du minimum
	Exemple (code Java 1.1) :
	`int min =`
	`sb.getMinimum( );`

int getOrientation( )	lecture de l'orientation
	Exemple (code Java 1.1) :
	`int o =`
	`sb.getOrientation( );`

int getValue( );	lecture de la valeur courante
	Exemple (code Java 1.1) :
	`int valeur =`
	`sb.getValue( );`

void setLineIncrement(int);	détermine la valeur à ajouter ou à ôter quand l'utilisateur clique sur une flèche de défilement Java 1.1 il faut utiliser la méthode setUnitIncrement()
int setPageIncrement( );	détermine la valeur à ajouter ou à ôter quand l'utilisateur clique sur le conteneur Java 1.1 il faut utiliser la méthode setBlockIncrement()

42.1.12. La classe Canvas

C'est un composant sans fonction particulière : il est utile pour créer des composants graphiques personnalisés.

Il est nécessaire d'étendre la classe Canvas pour en redéfinir la méthode Paint().

syntaxe : Cancas can = new Canvas( );

Exemple (code Java 1.1) :

```
import java.awt.*;

public class MonCanvas extends Canvas {

   public void paint(Graphics g) {
      g.setColor(Color.black);
      g.fillRect(10, 10, 100,50);
      g.setColor(Color.green);
      g.fillOval(40, 40, 10,10);
   }

}


import java.applet.*;
import java.awt.*;

public class AppletButton extends Applet {

   MonCanvas mc = new MonCanvas();

   public void paint(Graphics g) {
      super.paint(g);
      mc.paint(g);
   }
}
```

42.2. La classe Component

Les contrôles fenêtrés descendent plus ou moins directement de la classe AWT Component.

Cette classe contient de nombreuses méthodes :

Méthodes	Rôle
Rectangle bounds()	renvoie la position actuelle et la taille des composants Java 1.1 utiliser la méthode getBounds().
void disable()	désactive les composants Java 1.1 utiliser la méthode setEnabled(false).
void enable()	active les composants Java 1.1 utiliser la méthode setEnabled(true).
void enable(boolean)	active ou désactive le composant selon la valeur du paramètre Java 1.1 utiliser la méthode setEnabled(boolean).

Color getBackGround()	renvoie la couleur actuelle d'arrière plan
Font getFont()	renvoie la fonte utilisée pour afficher les caractères
Color getForeGround()	renvoie la couleur de premier plan
Graphics getGraphics()	renvoie le contexte graphique
Container getParent()	renvoie le conteneur (composant de niveau supérieur)
void hide()	masque l'objet Java 1.1 utiliser la méthode setVisible().
boolean inside(int x, int y)	indique si la coordonnée écran absolue se trouve dans l'objet Java 1.1 utiliser la méthode contains().
boolean isEnabled()	indique si l'objet est actif
boolean isShowing()	indique si l'objet est visible
boolean isVisible()	indique si l'objet est visible lorsque sont conteneur est visible
void layout()	repositionne l'objet en fonction du Layout Manager courant Java 1.1 utiliser la méthode doLayout().
Component locate(int x, int y)	retourne le composant situé à cet endroit Java 1.1 utiliser la méthode getComponentAt().
Point location()	retourne l'origine du composant Java 1.1 utiliser la méthode getLocation().
void move(int x, int y)	déplace les composants vers la position spécifiée Java 1.1 utiliser la méthode setLocation().
void paint(Graphics);	dessine le composant
void paintAll(Graphics)	dessine le composant et ceux qui sont contenus en lui
void repaint()	redessine le composant par appel à la méthode update()
void requestFocus();	demande le focus
void reshape(int x, inty, int w, int h)	modifie la position et la taille (unité : points écran) Java 1.1 utiliser la méthode setBounds().
void resize(int w, int h)	modifie la taille (unité : points écran) Java 1.1 utiliser la méthode setSize().
void setBackground(Color)	définit la couleur d'arrière plan
void setFont(Font)	définit la police

void setForeground(Color)	définit la couleur de premier plan
void show()	affiche le composant Java 1.1 utiliser la méthode setVisible(True).
Dimension size()	détermine la taille actuelle Java 1.1 utiliser la méthode getSize().

42.3. Les conteneurs

Les conteneurs sont des objets graphiques qui peuvent contenir d'autres objets graphiques, incluant éventuellement des conteneurs. Ils héritent de la classe Container.

Un composant graphique doit toujours être incorporé dans un conteneur :

Conteneur	Rôle
Panel	conteneur sans fenêtre propre. Utile pour ordonner les contrôles
Window	fenêtre principale sans cadre ni menu. Les objets descendants de cette classe peuvent servir à implémenter des menus
Dialog (descendant de Window)	réaliser des boîtes de dialogue simples
Frame (descendant de Window)	classe de fenêtre complètement fonctionnelle
Applet (descendant de Panel)	pas de menu. Pas de boîte de dialogue sans être incorporée dans une classe Frame.

L'insertion de composant dans un conteneur se fait grâce à la méthode add(Component) de la classe Container.

Exemple (code Java 1.1) :

```
Panel p = new Panel();

Button b1 = new button(" Premier ");
p.add(b1);
Button b2;
p.add(b2 = new Button (" Deuxième "));
p.add(new Button("Troisième "));
```

42.3.1. Le conteneur Panel

C'est essentiellement un objet de rangement pour d'autres composants.

La classe Panel possède deux constructeurs :

Constructeur	Rôle
Panel()	
Panel(LayoutManager)	Permet de préciser un layout manager

Exemple (code Java 1.1) :

```
Panel p = new Panel( );
```

```
Button b = new Button(" bouton ");
p.add( b);
```

42.3.2. Le conteneur Window

La classe Window contient plusieurs méthodes dont voici les plus utiles :

Méthodes	Rôle
void pack()	Calculer la taille et la position de tous les contrôles de la fenêtre. La méthode pack() agit en étroite collaboration avec le layout manager et permet à chaque contrôle de garder, dans un premier temps sa taille optimale. Une fois que tous les contrôles ont leur taille optimale, pack() utilise ces informations pour positionner les contrôles. pack() calcule ensuite la taille de la fenêtre. L'appel à pack() doit se faire à l'intérieur du constructeur de fenêtre après insertion de tous les contrôles.
void show()	Afficher la fenêtre
void dispose()	Libérer les ressources allouées à la fenêtre

42.3.3. Le conteneur Frame

Ce conteneur permet de créer des fenêtres d'encadrement. Il hérite de la classe Window qui ne s'occupe que de l'ouverture de la fenêtre. Window ne connait pas les menus ni les bordures qui sont gérés par la classe Frame. Dans une applet, elle n'apparait pas dans le navigateur mais comme une fenêtre indépendante.

Il existe deux constructeurs :

Constructeur	Rôle
Frame()	Exemple : Frame f = new Frame( );
Frame(String)	Précise le nom de la fenêtre Exemple : Frame f = new Frame(« titre »);

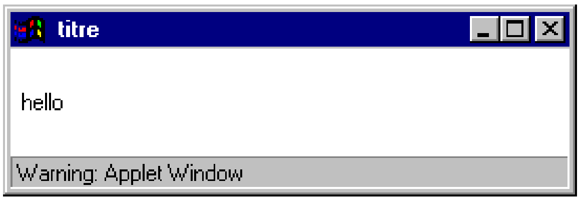
Les principales méthodes sont :

Méthodes	Rôle
setCursor(int)	changer le pointeur de la souris dans la fenêtre Exemple : f.setCursor(Frame.CROSSHAIR_CURSOR); Java 1.1 utiliser la méthode setCursor(Cursor).
int getCursorType()	déterminer la forme actuelle du curseur Java 1.1 utiliser la méthode getCursor().
Image getIconImage()	déterminer l'icone actuelle de la fenêtre
MenuBar getMenuBar()	déterminer la barre de menus actuelle
String getTitle()	déterminer le titre de la fenêtre
boolean isResizeable()	déterminer si la taille est modifiable
void remove(MenuComponent)	Supprimer un menu
void setIconImage(Image);	définir l'icone de la fenêtre
void setMenuBar(MenuBar)	Définir la barre de menus

void setResizeable(boolean)	définir si la taille peut être modifiée
void SetTitle(String)	définir le titre de la fenêtre

Exemple (code Java 1.1) :

```
import java.applet.*;
import java.awt.*;

public class AppletFrame extends Applet {

   Frame f;

   public void init() {
      super.init();
      // insert code to initialize the applet here
      f = new Frame("titre");
      f.add(new Label("hello "));
      f.setSize(300, 100);
      f.show();
   }
}
```



Le message « Warning : Applet window » est impossible à enlever dans la fenêtre : cela permet d'éviter la création d'une applet qui demande un mot de passe.

Le gestionnaire de mise en page par défaut d'une Frame est BorderLayout (FlowLayout pour une applet).

Exemple (code Java 1.1) : construction d'une fenêtre simple

```
import java.awt.*;

public class MaFrame extends Frame {

   public MaFrame() {
      super();
      setTitle(" Titre de la Fenetre ");
      setSize(300, 150);
      show(); // affiche la fenetre
   }

   public static void main(String[] args) {
      new MaFrame();
   }
}
```

42.3.4. Le conteneur Dialog

La classe Dialog hérite de la classe Window.

Une boîte de dialogue doit dérivée de la Classe Dialog de package java.awt.

Un objet de la classe Dialog doit dépendre d'un objet de la classe Frame.

Exemple (code Java 1.1) :

```
import java.awt.*;
import java.awt.event.*;

public class Apropos extends Dialog {

   public APropos(Frame parent) {
      super(parent, "A propos ", true);
      addWindowListener(new
      AProposListener(this));
      setSize(300, 300);
      setResizable(False);
   }
}

class AProposListener extends WindowAdapter {

   Dialog dialogue;
   public AProposListener(Dialog dialogue) {
      this.dialogue = dialog;
   }

   public void windowClosing(WindowEvent e) {
      dialogue.dispose();
   }

}
```

L'appel du constructeur Dialog(Frame, String, Boolean) permet de créer une instance avec comme paramètres : la fenêtre à laquelle appartient la boîte de dialogue, le titre de la boîte, le caractère modale de la boîte.

La méthode dispose() de la classe Dialog ferme la boîte et libère les ressources associées. Il ne faut pas associer cette action à la méthode windowClosed() car dispose provoque l'appel de windowClosed ce qui entrainerait un appel récursif infini.

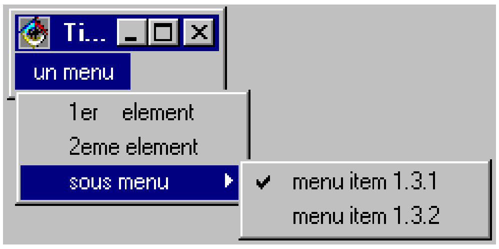
42.4. Les menus

Il faut insérer les menus dans des objets de la classe Frame (fenêtre d'encadrement). Il n'est donc pas possible d'insérer directement des menus dans une applet.

Il faut créer une barre de menus et l'affecter à la fenêtre d'encadrement. Il faut ensuite créer les entrées de chaque menu et les rattacher à la barre. Ajouter ensuite les éléments à chacun des menus.

Exemple (code Java 1.1) :

```
import java.awt.*;

public class MaFrame extends Frame {

   public MaFrame() {
      super();
      setTitle(" Titre de la Fenetre ");
      setSize(300, 150);

      MenuBar mb = new MenuBar();
      setMenuBar(mb);

      Menu m = new Menu(" un menu ");
      mb.add(m);
      m.add(new MenuItem(" 1er element "));
      m.add(new MenuItem(" 2eme element "));
      Menu m2 = new Menu(" sous menu ");

      CheckboxMenuItem cbm1 = new CheckboxMenuItem(" menu item 1.3.1 ");
      m2.add(cbm1);
      cbm1.setState(true);
      CheckboxMenuItem cbm2 = new CheckboxMenuItem(" menu item 1.3.2 ");
      m2.add(cbm2);
      m.add(m2);

      pack();
      show(); // affiche la fenetre

   }

   public static void main(String[] args) {
      new MaFrame();
   }

}
```



Exemple (code Java 1.1) : création d'une classe qui définit un menu

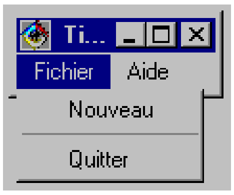
```
import java.awt.*;

public class MenuFenetre extends java.awt.MenuBar {

   public MenuItem menuQuitter, menuNouveau, menuApropos;

   public MenuFenetre() {

      Menu menuFichier = new Menu(" Fichier ");
      menuNouveau = new MenuItem(" Nouveau ");
      menuQuitter = new MenuItem(" Quitter ");

      menuFichier.add(menuNouveau);

      menuFichier.addSeparator();

      menuFichier.add(menuQuitter);

      Menu menuAide = new Menu(" Aide ");
      menuApropos = new MenuItem(" A propos ");
      menuAide.add(menuApropos);

      add(menuFichier);

      setHelpMenu(menuAide);
   }
}
```

La méthode setHelpMenu() confère sur certaines plates-formes un comportement particulier à ce menu.

La méthode setMenuBar() de la classe Frame prend en paramètre une instance de la classe MenuBar. Cette instance peut être directement une instance de la classe MenuBar qui aura été modifiée grâce aux méthodes add() ou alors une classe dérivée de MenuBar qui est adaptée aux besoins.

Exemple (code Java 1.1) :

```
import java.awt.*;

public class MaFrame extends Frame {

   public MaFrame() {
      super();
      setTitle(" Titre de la Fenetre ");
      setSize(300, 150);
      MenuFenetre mf = new
      MenuFenetre();

      setMenuBar(mf);

      pack();

      show(); // affiche la fenetre

   }

   public static void main(String[] args) {
      new MaFrame();
   }

}
```



42.4.1. Les méthodes de la classe MenuBar

Méthodes	Rôle
void add(Menu)	ajouter un menu dans la barre
int countMenus()	renvoie le nombre de menus Java 1.1 utiliser la méthode getMenuCount().
Menu getMenu(int pos)	renvoie le menu à la position spécifiée
void remove(int pos)	supprimer le menu à la position spécifiée
void remove(Menu)	supprimer le menu de la barre de menu

42.4.2. Les méthodes de la classe Menu

Méthodes	Rôle
MenuItem add(MenuItem) void add(String)	ajouter une option dans le menu
void addSeparator()	ajouter un trait de séparation dans le menu
int countItems()	renvoie le nombre d'options du menu Java 1.1 utiliser la méthode getItemCount().
MenuItem getItem(int pos)	déterminer l'option du menu à la position spécifiée
void remove(MenuItem mi)	supprimer la commande spécifiée
void remove(int pos)	supprimer la commande à la position spécifiée

42.4.3. Les méthodes de la classe MenuItem

Méthodes	Rôle
void disable()	désactiver l'élément Java 1.1 utiliser la méthode setEnabled(false).
void enable()	activer l'élément Java 1.1 utiliser la méthode setEnabled(true).
void enable(boolean cond)	désactiver ou activer l'élément en fonction du paramètre Java 1.1 utiliser la méthode setEnabled(boolean).
String getLabel()	Renvoie le texte de l'élément
boolean isEnabled()	renvoie l'état de l'élément (actif / inactif)
void setLabel(String text)	définir un nouveau texte pour la commande

42.4.4. Les méthodes de la classe CheckboxMenuItem

Méthodes	Rôle
boolean getState()	renvoie l'état d'activation de l'élément
Void setState(boolean)	définir l'état d'activation de l'élément

42.5. La classe java.awt.Desktop

Cette classe, ajoutée dans Java SE 6, permet de manipuler des documents sous la forme d'un fichier ou d'une URI à partir de leur type mime défini sur le système d'exploitation sous-jacent.

La méthode statique isDesktopSupported() permet de savoir si la classe Desktop est supportée par la plate-forme.

La méthode statique Desktop.getDesktop() donne un accès à l'instance de la classe Desktop.

Plusieurs constantes sont définies dans Desktop.Action pour préciser le type d'opération qu'il est possible de réaliser sur un document : BROWSE, EDIT, MAIL, OPEN et PRINT.

La méthode isSupported() permet de savoir si l'action est supportée sur la plate-forme mais cela ne signifie pas que cette action soit supportée pour tous les types mimes enregistrés sur la plate-forme.

Plusieurs méthodes permettent d'exécuter les actions : browse(), edit(), mail(), open() et print().

Exemple : ouverture du fichier fourni en paramètre

```
package fr.jmdoudoux.dej.java6;

import java.awt.*;
import java.io.*;

public class TestDektop {

    public static void main(String args[]) {
        if (Desktop.isDesktopSupported()) {
            Desktop desktop = Desktop.getDesktop();
            if (args.length == 1) {
                File fichier = new File(args[0]);
                if (desktop.isSupported(Desktop.Action.OPEN)) {
                    System.out.println("Ouverture du fichier " + fichier.getName());
                    try {
                        desktop.open(fichier);
                    } catch (IOException ioe) {
                        ioe.printStackTrace();
                    }
                }
            }
        }
    }
}
```

La méthode mail() attend en paramètre une uri qui doit utiliser le protocole mailto:.

La méthode browse() attend en paramètre une uri qui utilise un protocole reconnu par le navigateur http, https, ...

43. La création d'interfaces graphiques avec AWT

Chapitre 43

Niveau : 🟩🟩🟨⬜⬜⬜⬜ Intermédiaire

API legacy
Ce chapitre est conservé pour des raisons historiques

AWT propose un ensemble de composants et de fonctionnalités pour créer des interfaces graphiques.

Ce chapitre contient plusieurs sections :

- Le dimensionnement des composants
- Le positionnement des composants
- La création de nouveaux composants à partir de Panel
- L'activation ou la désactivation des composants

43.1. Le dimensionnement des composants

En principe, il est automatique grâce au LayoutManager. Pour donner à un composant une taille donnée, il faut redéfinir la méthode getPreferedSize() de la classe Component.

Exemple (code Java 1.1) :

```
import java.awt.*;

public class MonBouton extends Button {

   public Dimension getPreferredSize() {
      return new Dimension(800, 250);
   }

}
```

La méthode getPreferedSize() indique la taille souhaitée mais pas celle imposée. En fonction du Layout Manager, le composant pourra ou non imposer sa taille.

Layout	Hauteur	Largeur
Sans Layout	oui	oui
FlowLayout	oui	oui
BorderLayout(East, West)	non	oui
BorderLayout(North, South)	oui	non
BorderLayout(Center)	non	non

GridLayout	non	non

Cette méthode oblige à sous-classer tous les composants.

Une autre façon de faire est de se passer des Layout et de placer les composants à la main en indiquant leurs coordonnées et leurs dimensions.

Pour supprimer le Layout par défaut d'une classe, il faut appeler la méthode setLayout() avec comme paramètre null.

Trois méthodes de la classe Component permettent de positionner des composants :

- setBounds(int x, int y, int largeur, int hauteur)
- setLocation(int x , int y)
- setSize(int largeur, int hauteur)

Ces méthodes permettent de placer un composant à la position (x,y) par rapport au conteneur dans lequel il est inclus et d'indiquer sa largeur et sa hauteur.

Toutefois, les Layout Manager constituent un des facteurs importants de la portabilité des interfaces graphiques notamment en gérant la disposition et le placement des composants après redimensionnement du conteneur.

43.2. Le positionnement des composants

Lorsqu'on intègre un composant graphique dans un conteneur, il n'est pas nécessaire de préciser son emplacement car il est déterminé de façon automatique : la mise en forme est dynamique. On peut influencer cette mise en page en utilisant un gestionnaire de mise en page (Layout Manager) qui définit la position de chaque composant inséré. Dans ce cas, la position spécifiée est relative aux autres composants.

Chaque layout manager implémente l'interface java.awt.LayoutManager.

Il est possible d'utiliser plusieurs gestionnaires de mise en forme pour définir la présentation des composants. Par défaut, c'est la classe FlowLayout qui est utilisée pour la classe Panel et la classe BorderLayout pour Frame et Dialog.

Pour affecter une nouvelle mise en page, il faut utiliser la méthode setLayout() de la classe Container.

Exemple (code Java 1.1) :

```
   Panel p = new Panel();
   GridLayout gl = new GridLayout(5,5);
   p.setLayout(gl);

   // ou p.setLayout( new GridLayout(5,5));
```

Les layout manager ont 3 avantages :

- l'aménagement des composants graphiques est délégué aux layout managers (il est inutile d'utiliser les coordonnées absolues)
- en cas de redimensionnement de la fenêtre, les contrôles sont automatiquement agrandis ou réduits
- ils permettent une indépendance vis à vis des plates-formes.

Pour créer un espace entre les composants et le bord de leur conteneur, il faut redéfinir la méthode getInsets() d'un conteneur : cette méthode est héritée de la classe Container.

Exemple (code Java 1.1) :

```
public Insets getInsets() {
   Insets normal = super.getInsets();
   return new Insets(normal.top + 10, normal.left + 10,
   normal.bottom + 10, normal.right + 10);
}
```

Cet exemple permet de laisser 10 pixels en plus entre chaque bords du conteneur.

43.2.1. La mise en page par flot (FlowLayout)

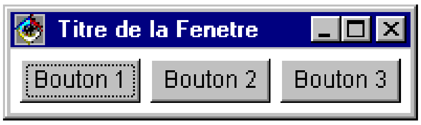
La classe FlowLayout (mise en page flot) place les composants ligne par ligne de gauche à droite. Chaque ligne est complétée progressivement jusqu'à être remplie, puis passe à la suivante. Chaque ligne est centrée par défaut. C'est la mise en page par défaut des applets.

Il existe plusieurs constructeurs :

Constructeur	Rôle
FlowLayout( );	
FlowLayout(int align);	Permet de préciser l'alignement des composants dans le conteneur (CENTER, LEFT, RIGHT ...). Par défaut, align vaut CENTER
FlowLayout(int align, int hgap, int vgap);	Permet de préciser l'alignement et l'espacement horizontal et vertical dont la valeur par défaut est 5.

Exemple (code Java 1.1) :

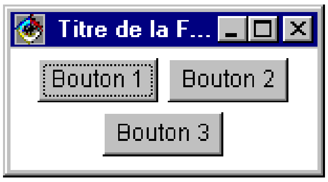
```
import java.awt.*;

public class MaFrame extends Frame {

   public MaFrame() {
      super();
      setTitle(" Titre de la Fenetre ");
      setSize(300, 150);
      setLayout(new FlowLayout());
      add(new Button("Bouton 1"));
      add(new Button("Bouton 2"));
      add(new Button("Bouton 3"));

      pack();
      show(); // affiche la fenetre
   }

   public static void main(String[] args) {
      new MaFrame();
   }
}
```



Chaque applet possède une mise en page flot implicitement initialisée à FlowLayout(FloawLayout.CENTER,5,5).

FlowLayout utilise les dimensions de son conteneur comme seul principe de mise en forme des composants. Si les dimensions du conteneur changent, le positionnement des composants est recalculé.

Exemple : la fenêtre précédente est simplement redimensionnée



43.2.2. La mise en page bordure (BorderLayout)

Avec ce Layout Manager, la disposition des composants est commandée par une mise en page en bordure qui découpe la surface en cinq zones : North, South, East, West, Center. On peut librement utiliser une ou plusieurs zones.

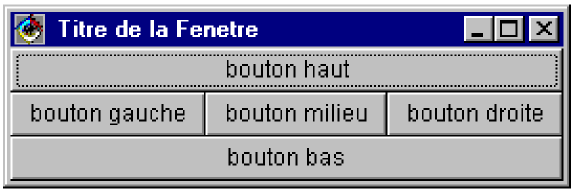
BorderLayout consacre tout l'espace du conteneur aux composants. Le composant du milieu dispose de la place inutilisée par les autres composants.

Il existe plusieurs constructeurs :

Constructeur	Rôle
BorderLayout( )	
BorderLayout (int hgap,int vgap)	Permet de préciser l'espacement horizontal et vertical des composants.

Exemple (code Java 1.1) :

```
import java.awt.*;

public class MaFrame extends Frame {

   public MaFrame() {
      super();
      setTitle("
      Titre de la Fenetre ");
      setSize(300, 150);
      setLayout(new BorderLayout());
      add("North", new Button(" bouton haut "));
      add("South", new Button(" bouton bas "));
      add("West", new Button(" bouton gauche "));
      add("East", new Button(" bouton droite "));
      add("Center", new Button(" bouton milieu "));
      pack();
      show(); // affiche la fenetre
   }

   public static void
      main(String[] args) {
      new MaFrame();
   }
}
```

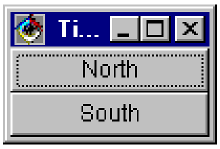


Il est possible d'utiliser deux méthodes add surchargées de la classe Container : add(String, Component) avec le premier paramètre précisant l'orientation du composant ou add(Component, Objet) dont le second paramètre précise la position sous forme de constante définie dans la classe BorderLayout.

Exemple (code Java 1.1) :

```
import java.awt.*;
```

```
public class MaFrame extends Frame {

   public MaFrame() {

      super();
      setTitle(" Titre de la Fenetre ");
      setSize(300, 150);
      setLayout(new BorderLayout());
      add(new Button("North"), BorderLayout.NORTH);
      add(new Button("South"), BorderLayout.SOUTH);
      pack();
      show(); // affiche la fenetre
   }

   public static void main(String[] args) {
      new MaFrame();
   }
}
```



43.2.3. La mise en page de type carte (CardLayout)

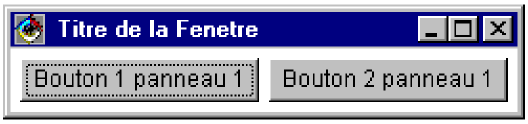
Ce layout manager aide à construire des boîtes de dialogue composées de plusieurs onglets. Un onglet se compose généralement de plusieurs contrôles : on insère des panneaux dans la fenêtre utilisée par le CardLayout Manager. Chaque panneau correspond à un onglet de boîte de dialogue et contient plusieurs contrôles. Par défaut, c'est le premier onglet qui est affiché.

Ce layout possède deux constructeurs :

Constructeurs	Rôle
CardLayout()	
CardLayout(int, int)	Permet de préciser l'espace horizontal et vertical du tour du composant

Exemple (code Java 1.1) :

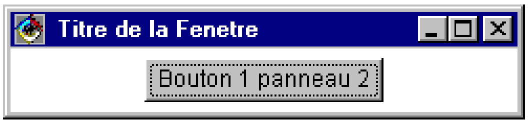
```
import java.awt.*;

public class MaFrame extends Frame {

   public MaFrame() {

      super();
      setTitle("Titre de la Fenetre ");
      setSize(300,150);
      CardLayout cl = new CardLayout();
      setLayout(cl);

      //création d'un panneau contenant les contrôles d'un onglet
      Panel p = new Panel();

      //ajouter les composants au panel
      p.add(new Button("Bouton 1 panneau 1"));
      p.add(new Button("Bouton 2 panneau 1"));

      //inclure le panneau dans la fenetre sous le nom "Page1"
      // ce nom est utilisé par show()
      add("Page1",p);

      //déclaration et insertion de l'onglet suivant
      p = new Panel();
      p.add(new Button("Bouton 1 panneau 2"));
      add("Page2", p);

      // affiche la fenetre
      pack();
      show();
   }

   public static void main(String[] args) {
      new MaFrame();
   }
}
```



Lors de l'insertion d'un onglet, un nom doit lui être attribué. Les fonctions nécessaires pour afficher un onglet de boîte de dialogue ne sont pas fournies par les méthodes du conteneur, mais seulement par le Layout Manager. Il est nécessaire de sauvegarder temporairement le Layout Manager dans une variable où déterminer le gestionnaire en cours par un appel à getLayout(). Pour appeler un onglet donné, il faut utiliser la méthode show() du CardLayout Manager.

Exemple (code Java 1.1) :

```
((CardLayout)getLayout()).show(this, "Page2");
```



Les méthodes first(), last(), next() et previous() servent à parcourir les onglets de boîte de dialogue :

Exemple (code Java 1.1) :

```
((CardLayout)getLayout()).first(this);
```

43.2.4. La mise en page GridLayout

Ce Layout Manager établit un réseau de cellules identiques qui forment une sorte de quadrillage invisible : les composants sont organisés en lignes et en colonnes. Les éléments insérés dans la grille ont tous la même taille. Les cellules du quadrillage se remplissent de gauche à droite ou de haut en bas.

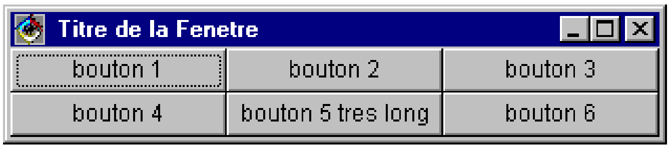
Il existe plusieurs constructeurs :

Constructeur	Rôle
GridLayout(int, int);	Les deux premiers entiers spécifient le nombre de lignes ou de colonnes de la grille.
GridLayout(int, int, int, int);	permet de préciser en plus l'espacement horizontal et vertical des composants.

Exemple (code Java 1.1) :

```
import java.awt.*;
```

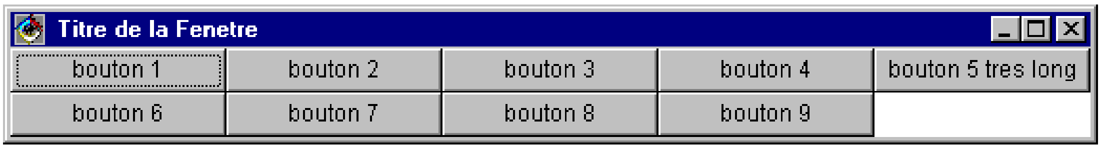
```
public class MaFrame extends Frame {

   public MaFrame() {
      super();
      setTitle(" Titre de la Fenetre ");
      setSize(300, 150);
      setLayout(new GridLayout(2, 3));
      add(new Button("bouton 1"));
      add(new Button("bouton 2"));
      add(new Button("bouton 3"));
      add(new Button("bouton 4"));
      add(new Button("bouton 5 tres long"));
      add(new Button("bouton 6"));

      pack();
      show(); // affiche la fenetre
   }

   public static void main(String[] args) {
      new MaFrame();
   }
}
```



Attention : lorsque le nombre de lignes et de colonnes sont spécifiés alors le nombre de colonnes est ignoré. Ainsi par exemple, GridLayout(5,4) est équivalent à GridLayout(5,0).

Exemple (code Java 1.1) :

```
import java.awt.*;

public class MaFrame extends Frame {

   public MaFrame() {
      super();
      setTitle(" Titre de la Fenetre ");
      setSize(300, 150);
      setLayout(new GridLayout(2, 3));
      add(new Button("bouton 1"));
      add(new Button("bouton 2"));
      add(new Button("bouton 3"));
      add(new Button("bouton 4"));
      add(new Button("bouton 5 tres long"));
      add(new Button("bouton 6"));
      add(new Button("bouton 7"));
      add(new Button("bouton 8"));
      add(new Button("bouton 9"));

      pack();
      show(); // affiche la fenetre
   }

   public static void void
      main(String[] args) {
      new MaFrame();
   }
}
```



43.2.5. La mise en page GridBagLayout

Ce gestionnaire (grille étendue) est le plus riche en fonctionnalités : le conteneur est divisé en cellules égales mais un composant peut occuper plusieurs cellules de la grille et il est possible de faire une distribution dans des cellules distinctes. Un objet de la classe GridBagConstraints permet de donner les indications de positionnement et de dimension à l'objet GridBagLayout.

Les lignes et les colonnes prennent naissance au moment où les contrôles sont ajoutés. Chaque contrôle est associé à un objet de la classe GridBagConstraints qui indique l'emplacement voulu pour le contrôle.

Exemple (code Java 1.1) :

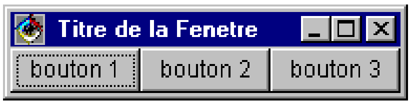
```
GridBagLayout gb1 = new GridBagLayout( );
GridBagConstraints gbc = new GridBagConstraints( );
```

Les variables d'instances pour manipuler l'objet GridBagLayoutConstraints sont :

Variable	Rôle
gridx et gridy	Ces variables contiennent les coordonnées de l'origine de la grille. Elles permettent un positionnement précis à une certaine position d'un composant. Par défaut elles ont la valeur GrigBagConstraint.RELATIVE qui indique qu'un composant se range à droite du précédent
gridwidth, gridheight	Définissent combien de cellules va occuper le composant (en hauteur et largeur). Par défaut la valeur est 1. L'indication est relative aux autres composants de la ligne ou de la colonne. La valeur GridBagConstraints.REMAINDER spécifie que le prochain composant inséré sera le dernier de la ligne ou de la colonne courante. La valeur GridBagConstraints.RELATIVE place le composant après le dernier composant d'une ligne ou d'une colonne.
fill	Définit le sort d'un composant plus petit que la cellule de la grille. GridBagConstraints.NONE conserve la taille d'origine : valeur par défaut GridBagConstraints.HORIZONTAL dilaté horizontalement GridBagConstraints.VERTICAL dilaté verticalement GridBagConstraints.BOTH dilatés aux dimensions de la cellule
ipadx, ipady	Permettent de définir l'agrandissement horizontal et vertical des composants. Ne fonctionne que si une dilatation est demandée par fill. La valeur par défaut est (0,0).
anchor	Lorsqu'un composant est plus petit que la cellule dans laquelle il est inséré, il peut être positionné à l'aide de cette variable pour définir le côté par lequel le contrôle doit être aligné dans la cellule. Les variables possibles sont NORTH, NORTHWEST, NORTHEAST, SOUTH, SOUTHWEST, SOUTHEAST, WEST et EAST
weightx, weighty	Permettent de définir la répartition de l'espace en cas de changement de dimension

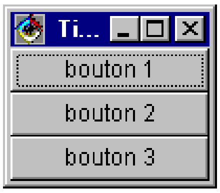
Exemple (code Java 1.1) :

```
import java.awt.*;

public class MaFrame extends Frame {

   public MaFrame() {
      super();
      setTitle(" Titre de la Fenetre ");
      setSize(300, 150);

      Button b1 = new Button(" bouton 1 ");
      Button b2 = new Button(" bouton 2 ");
      Button b3 = new Button(" bouton 3 ");

      GridBagLayout gb = new GridBagLayout();
      GridBagConstraints gbc = new GridBagConstraints();
      setLayout(gb);

      gbc.fill = GridBagConstraints.BOTH;
      gbc.weightx = 1;
      gbc.weighty = 1;
      gb.setConstraints(b1, gbc); // mise en forme des objets
      gb.setConstraints(b2, gbc);
      gb.setConstraints(b3, gbc);

      add(b1);
      add(b2);
      add(b3);

      pack();
      show(); // affiche la fenetre
   }

   public static void main(String[] args) {
      new MaFrame();
   }
}
```



Cet exemple place trois boutons l'un à coté de l'autre. Ceci permet en cas de changement de dimension du conteneur de conserver la mise en page : la taille des composants est automatiquement ajustée.

Pour placer les 3 boutons l'un au-dessus de l'autre, il faut affecter la valeur 1 à la variable gbc.gridx.



43.3. La création de nouveaux composants à partir de Panel

Il est possible de définir de nouveaux composants qui héritent directement de Panel.

Exemple (code Java 1.1) :

```
class PanneauClavier extends Panel {

   PanneauClavier()
   {
      setLayout(new GridLayout(4,3));

      for (int num=1; num <= 9 ; num++) {
        add(new Button(Integer.toString(num)));
      }
      add(new Button("*"));
      add(new Button("0"));
      add(new Button("# "));
   }
}

public class demo exteds Applet {

   public void init() { add(new PanneauClavier()); }

}
```

43.4. L'activation ou la désactivation des composants

L'activation ou la désactivation d'un composant se fait grâce à sa méthode setEnabled(boolean). La valeur booléenne passée en paramètre indique l'état du composant (false : interdit l'usage du composant). Cette méthode est un moyen d'interdire à un composant d'envoyer des événements utilisateurs.

44. L'interception des actions de l'utilisateur

Chapitre 44

Niveau : 🟩🟩🟨⬜⬜⬜ Intermédiaire

N'importe quelle interface graphique doit interagir avec l'utilisateur et donc réagir à certains événements. Le modèle de gestion de ces événements à changer entre le JDK 1.0 et 1.1.

Ce chapitre traite de la capture de ces événements pour leur associer des traitements. Il contient plusieurs sections :

- ♦ L'interception des actions de l'utilisateur avec Java version 1.0
- ♦ L'interception des actions de l'utilisateur avec Java version 1.1

44.1. L'interception des actions de l'utilisateur avec Java version 1.0

Cette section sera développée dans une version future de ce document

44.2. L'interception des actions de l'utilisateur avec Java version 1.1

Les événements utilisateurs sont gérés par plusieurs interfaces EventListener.

Les interfaces EventListener permettent de définir les traitements en réponse à des événements utilisateurs généré par un composant. Une classe doit contenir une interface auditrice pour chaque type d'événements à traiter :

- ActionListener : clic de souris ou enfoncement de la touche Enter
- ItemListener : utilisation d'une liste ou d'une case à cocher
- MouseMotionListener : événement de souris
- WindowListener : événement de fenêtre

L'ajout d'une interface EventListener impose plusieurs ajouts dans le code :

1. importer le groupe de classes java.awt.event

Exemple (code Java 1.1) :

```
import java.awt.event.*;
```

2. la classe doit déclarer qu'elle utilisera une ou plusieurs interfaces d'écoute

   Exemple (code Java 1.1) :

   ```
   public class AppletAction extends Applet implements ActionListener{
   ```

   Pour déclarer plusieurs interfaces, il suffit de les séparer par des virgules

   Exemple (code Java 1.1) :

   ```
   public class MonApplet extends Applet implements ActionListener, MouseListener {
   ```

3. Appel à la méthode addXXX() pour enregistrer l'objet qui gérera les événements XXX du composant

   Il faut configurer le composant pour qu'il possède un «écouteur» pour l'événement utilisateur concerné.

   Exemple (code Java 1.1) : création d'un bouton capable de réagir à un événements

   ```
   Button b = new Button("boutton");
   b.addActionListener(this);
   ```

   Ce code crée l'objet de la classe Button et appelle sa méthode addActionListener(). Cette méthode permet de préciser la classe qui va gérer l'événement utilisateur de type ActionListener du bouton. Cette classe doit impérativement implémenter l'interface de type EventListener correspondante soit dans cet exemple ActionListener. L'instruction this indique que la classe elle même recevra et gérera l'événement utilisateur.

   L'apparition d'un événement utilisateur généré par un composant doté d'un auditeur appelle automatiquement une méthode. Cette dernière doit se trouver dans la classe référencée dans l'instruction qui lie l'auditeur au composant. Dans l'exemple, cette méthode doit être située dans la même classe parce que c'est l'objet lui-même qui est spécifié avec l'instruction this. Une autre classe indépendante peut être utilisée : dans ce cas il faut préciser une instance de cette classe en tant que paramètre.
4. implémenter les méthodes déclarées dans les interfaces

   Chaque auditeur possède des méthodes différentes qui sont appelées pour traiter leurs événements. Par exemple, l'interface ActionListener envoie des événements à une méthode nommée actionPerformed( ).

   Exemple (code Java 1.1) :

   ```
   public void actionPerformed(ActionEvent evt) {
      //insérer ici le code de la méthode
   };
   ```

   Pour identifier le composant qui a généré l'événement, il faut utiliser la méthode getActionCommand() de l'objet ActionEvent fourni en paramètre de la méthode :

   Exemple (code Java 1.1) :

   ```
   String composant = evt.getActionCommand();
   ```

   La méthode getActionCommand() renvoie une chaîne de caractères. Si le composant est un bouton, alors il renvoie le texte du bouton, si le composant est une zone de saisie, c'est le texte saisi qui sera renvoyé (il faut appuyer sur «Entrer» pour générer l'événement), etc ...

   La méthode getSource() renvoie l'objet qui a généré l'événement. Cette méthode est plus sûre que la précédente

   Exemple (code Java 1.1) :

   ```
   Button b = new Button(" bouton ");
   ```

```
...

void public actionPerformed(actionEvent evt) {
   Object source = evt.getSource();

   if (source == b) // action a effectuer
}
```

La méthode getSource() peut être utilisée avec tous les événements utilisateur.

Exemple (code Java 1.1) : Exemple complet qui affiche le composant qui a généré l'événement

```
package applets;

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class AppletAction extends Applet implements ActionListener{

   public void actionPerformed(ActionEvent evt) {
      String composant = evt.getActionCommand();
      showStatus("Action sur le composant : " + composant);
   }


   public void init() {
      super.init();

      Button b1 = new Button("boutton 1");
      b1.addActionListener(this);
      add(b1);

      Button b2 = new Button("boutton 2");
      b2.addActionListener(this);
      add(b2);

      Button b3 = new Button("boutton 3");
      b3.addActionListener(this);
      add(b3);
   }
}
```

44.2.1. L'interface ItemListener

Cette interface permet de réagir à la sélection de cases à cocher et de listes d'options. Pour qu'un composant génère des événements, il faut utiliser la méthode addItemListener().

Exemple (code Java 1.1) :

```
Checkbox cb = new Checkbox(" choix ",true);
cb.addItemListener(this);
```

Ces événements sont reçus par la méthode itemStateChanged() qui attend un objet de type ItemEvent en argument

Pour déterminer si une case à cocher est sélectionnée ou inactive, utiliser la méthode getStateChange() avec les constantes ItemEvent.SELECTED ou ItemEvent.DESELECTED.

Exemple (code Java 1.1) :

```
package applets;

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class AppletItem extends Applet implements ItemListener{

   public void init() {
      super.init();
      Checkbox cb = new Checkbox("choix 1", true);
      cb.addItemListener(this);
      add(cb);
   }

   public void itemStateChanged(ItemEvent item) {
      int status = item.getStateChange();
      if (status == ItemEvent.SELECTED)
         showStatus("choix selectionne");
      else
         showStatus("choix non selectionne");
   }

}
```

Pour connaitre l'objet qui a généré l'événement, il faut utiliser la méthode getItem().

Pour déterminer la valeur sélectionnée dans une combo box, il faut utiliser la méthode getItem() et convertir la valeur en chaîne de caractères.

Exemple (code Java 1.1) :

```
Package applets;

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class AppletItem extends Applet implements ItemListener{

   public void init() {
      Choice c = new Choice();
      c.add("choix 1");
      c.add("choix 2");
      c.add("choix 3");
      c.addItemListener(this);
      add(c);
}

   public void itemStateChanged(ItemEvent item) {
      Object obj = item.getItem();
      String selection = (String)obj;
      showStatus("choix : "+selection);
   }
}
```

44.2.2. L'interface TextListener

Cette interface permet de réagir aux modifications de la zone de saisie ou du texte.

La méthode addTextListener() permet à un composant de texte de générer des événements utilisateur. La méthode TextValueChanged() reçoit les événements.

Exemple (code Java 1.1) :

```
package applets;

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
```

```
public class AppletText extends Applet implements TextListener{

   public void init() {
      super.init();

      TextField t = new TextField("");
      t.addTextListener(this);
      add(t);
   }

   public void textValueChanged(TextEvent txt) {
      Object source = txt.getSource();
      showStatus("saisi = "+((TextField)source).getText());
   }
}
```

44.2.3. L'interface MouseMotionListener

La méthode addMouseMotionListener() permet de gérer les événements liés à des mouvements de souris. Les méthodes mouseDragged() et mouseMoved() reçoivent les événements.

Exemple (code Java 1.1) :

```
package applets;

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class AppletMotion extends Applet implements MouseMotionListener{
   private int x;
   private int y;

   public void init() {
      super.init();
      this.addMouseMotionListener(this);
   }

   public void mouseDragged(java.awt.event.MouseEvent e) {}

   public void mouseMoved(MouseEvent e) {
      x = e.getX();
      y = e.getY();
      repaint();
      showStatus("x = "+x+" ; y = "+y);
   }

   public void paint(Graphics g) {
      super.paint(g);
      g.drawString("x = "+x+" ; y = "+y,20,20);
   }
}
```

44.2.4. L'interface MouseListener

Cette interface permet de réagir aux clics de souris. Les méthodes de cette interface sont :

- public void mouseClicked(MouseEvent e);
- public void mousePressed(MouseEvent e);
- public void mouseReleased(MouseEvent e);
- public void mouseEntered(MouseEvent e);
- public void mouseExited(MouseEvent e);

Exemple (code Java 1.1) :

```
package applets;
```

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class AppletMouse extends Applet implements MouseListener {
   int nbClick = 0;

   public void init() {
      super.init();
      addMouseListener(this);
   }

   public void mouseClicked(MouseEvent e) {
      nbClick++;
      repaint();
   }

   public void mouseEntered(MouseEvent e) {}

   public void mouseExited(MouseEvent e) {}

   public void mousePressed(MouseEvent e) {}

   public void mouseReleased(MouseEvent e) {}

   public void paint(Graphics g) {
      super.paint(g);
      g.drawString("Nombre de clics : "+nbClick,10,10);
   }
}
```

Une classe qui implémente cette interface doit définir ces 5 méthodes. Si toutes les méthodes ne doivent pas être utilisées, il est possible de définir une classe qui hérite de MouseAdapter. Cette classe fournit une implémentation par défaut de l'interface MouseListener.

Exemple (code Java 1.1) :

```
class gestionClics extends MouseAdapter {

   public void mousePressed(MouseEvent e) {
      //traitement
   }
}
```

Dans le cas d'une classe qui hérite d'une classe Adapter, il suffit de redéfinir la ou les méthodes qui contiendront du code pour traiter les événements concernés. Par défaut, les différentes méthodes définies dans l'Adapter ne font rien.

Cette nouvelle classe ainsi définie doit être passée en paramètre à la méthode addMouseListener() au lieu de this qui indiquait que la classe répondait elle même à l'événement.

44.2.5. L'interface WindowListener

La méthode addWindwowListener() permet à un objet Frame de générer des événements. Les méthodes de cette interface sont :

- public void windowOpened(WindowEvent e)
- public void windowClosing(WindowEvent e)
- public void windowClosed(WindowEvent e)
- public void windowIconified(WindowEvent e)
- public void windowDeinconified(WindowEvent e)
- public void windowActivated(WindowEvent e)
- public void windowDeactivated(WindowEvent e)

windowClosing() est appelée lorsque l'on clique sur la case système de fermeture de la fenêtre. windowClosed() est appelé après la fermeture de la fenêtre : cette méthode n'est utile que si la fermeture de la fenêtre n'entraine pas la fin de l'application.

Exemple (code Java 1.1) :

```
package test;

import java.awt.event.*;

class GestionnaireFenetre extends WindowAdpter {

   public void windowClosing(WindowEvent e) {
      System.exit(0);
   }
}
```

Exemple (code Java 1.1) :

```
package test;

import java.awt.*;
import java.awt.event.*;

public class TestFrame extends Frame {

   private GestionnaireFenetre gf = new GestionnaireFenetre();


   public TestFrame(String title) {
     super(title);
     addWindowListener(gf);
   }

   public static void main(java.lang.String[] args) {
      try {
         TestFrame tf = new TestFrame("TestFrame");
         tf.setVisible(true);
      } catch (Throwable e) {
         System.err.println("Erreur");
         e.printStackTrace(System.out);
      }
   }
}
```

44.2.6. Les différentes implémentations des Listeners

La mise en oeuvre des Listeners peut se faire selon différentes formes : la classe implémentant elle même l'interface, une classe indépendante, une classe interne, une classe interne anonyme.

44.2.6.1. Une classe implémentant elle même le listener

Exemple (code Java 1.1) :

```
package test;

import java.awt.*;
import java.awt.event.*;

public class TestFrame3 extends Frame implements WindowListener {

   public TestFrame3(String title) {
      super(title);
      this.addWindowListener(this);
   }

   public static void main(java.lang.String[] args) {
      try {
         TestFrame3 tf = new TestFrame3("testFrame3");
         tf.setVisible(true);
      } catch (Throwable e) {
         System.err.println("Erreur");
         e.printStackTrace(System.out);
      }
   }

   public void windowActivated(java.awt.event.WindowEvent e) {}

   public void windowClosed(java.awt.event.WindowEvent e) {}

   public void windowClosing(java.awt.event.WindowEvent e) {
      System.exit(0);
   }

   public void windowDeactivated(java.awt.event.WindowEvent e) {}

   public void windowDeiconified(java.awt.event.WindowEvent e) {}

   public void windowIconified(java.awt.event.WindowEvent e) {}

   public void windowOpened(java.awt.event.WindowEvent e) {}

}
```

44.2.6.2. Une classe indépendante implémentant le listener

Exemple (code Java 1.1) :

```
package test;

import java.awt.*;
import java.awt.event.*;

public class TestFrame4 extends Frame {

   public TestFrame4(String title) {
      super(title);
      gestEvt ge = new gestEvt();
      addWindowListener(ge);
   }

   public static void main(java.lang.String[] args) {
      try {
         TestFrame4 tf = new TestFrame4("testFrame4");
         tf.setVisible(true);
      } catch (Throwable e) {
         System.err.println("Erreur");
         e.printStackTrace(System.out);
      }
   }
}
```

Exemple (code Java 1.1) :

```
package test;

import java.awt.event.*;

public class gestEvt implements WindowListener {

   public void windowActivated(WindowEvent e) {}
   public void windowClosed(WindowEvent e) {}
   public void windowClosing(WindowEvent e) {
      System.exit(0);
   }
   public void windowDeactivated(WindowEvent e) {}
   public void windowDeiconified(WindowEvent e) {}
   public void windowIconified(WindowEvent e) {}
   public void windowOpened(WindowEvent e) {}
}
```

44.2.6.3. Une classe interne

Exemple (code Java 1.1) :

```
package test;

import java.awt.*;
import java.awt.event.*;

public class TestFrame2 extends Frame {

   class gestEvt implements WindowListener {
      public void windowActivated(WindowEvent e) {};
      public void windowClosed(WindowEvent e) {};
      public void windowClosing(WindowEvent e) {
         System.exit(0);
      };
      public void windowDeactivated(WindowEvent e) {};
      public void windowDeiconified(WindowEvent e) {};
      public void windowIconified(WindowEvent e) {};
      public void windowOpened(WindowEvent e) {};
   };

   private gestEvt ge = new TestFrame2.gestEvt();


   public TestFrame2(String title) {
     super(title);
     addWindowListener(ge);
   }

   public static void main(java.lang.String[] args) {
      try {
         TestFrame2 tf = new TestFrame2("TestFrame2");
         tf.setVisible(true);
      } catch (Throwable e) {
         System.err.println("Erreur");
         e.printStackTrace(System.out);
      }
   }
}
```

44.2.6.4. Une classe interne anonyme

Exemple (code Java 1.1) :

```
package test;

import java.awt.*;
import java.awt.event.*;

public class TestFrame1 extends Frame {

   public TestFrame1(String title) {
      super(title);
      addWindowListener(new WindowAdapter() {
         public void windowClosed(.WindowEvent e) {
            System.exit(0);
         };
      });
   }

   public static void main(java.lang.String[] args) {
      try {
         TestFrame1 tf = new TestFrame1("TestFrame");
         tf.setVisible(true);
      } catch (Throwable e) {
         System.err.println("Erreur");
         e.printStackTrace(System.out);
      }
   }
}
```

44.2.7. Résumé

Le mécanisme mis en place pour intercepter des événements est le même quel que soit ces événements :

- associer au composant qui est à l'origine de l'événement un contrôleur adéquat : utilisation des méthodes addXXXListener(). Le paramètre de ces méthodes indique l'objet qui a la charge de répondre au message : cet objet doit implémenter l'interface XXXListener correspondante ou dérivée d'une classe XXXAdapter, ce qui revient à créer une classe qui implémente l'interface associée à l'événement que l'on veut gérer. Cette classe peut être celle du composant qui est à l'origine de l'événement (facilité d'implémentation) ou une classe indépendante qui détermine la frontière entre l'interface graphique (émission d'événements) et celle qui représente la logique de l'application (traitement des événements) .
- les classes XXXAdapter sont utiles pour créer des classes dédiées au traitement des événements car elles implémentent des méthodes par défaut pour celles définies dans l'interface XXXListener dérivées de EventListener. Il n'existe une classe Adapter que pour les interfaces qui possèdent plusieurs méthodes.
- implémenter la méthode associée à l'événement qui fournit en paramètre un objet de type AWTEvent (classe mère de tout événement) contenant des informations utiles (position du curseur, état du clavier ...).

45. Le développement d'interfaces graphiques avec SWING

Chapitre 45

Niveau : 🟩🟩🟨⬜⬜⬜ Intermédiaire

Swing fait partie de la bibliothèque Java Foundation Classes (JFC). C'est une API dont le but est similaire à celui de l'API AWT mais dont les modes de fonctionnement et d'utilisation sont complètement différents. Swing a été intégré au JDK depuis sa version 1.2. Cette bibliothèque existe séparément. pour le JDK 1.1.

La bibliothèque JFC contient :

- l'API Swing : de nouvelles classes et interfaces pour construire des interfaces graphiques
- Accessibility API :
- 2D API: support du graphisme en 2D
- API pour l'impression et le cliquer/glisser

Ce chapitre contient plusieurs sections :

- ♦ La présentation de Swing
- ♦ Les packages Swing
- ♦ Un exemple de fenêtre autonome
- ♦ Les composants Swing
- ♦ Les boutons
- ♦ Les composants de saisie de texte
- ♦ Les onglets
- ♦ Le composant JTree
- ♦ Les menus
- ♦ L'affichage d'une image dans une application.

45.1. La présentation de Swing

Swing propose de nombreux composants dont certains possèdent des fonctions étendues, une utilisation des mécanismes de gestion d'événements performants (ceux introduits par le JDK 1.1) et une apparence modifiable à la volée (une interface graphique qui emploie le style du système d'exploitation Windows ou Motif ou un nouveau style spécifique à Java nommé Metal).

Tous les éléments de Swing font partie d'un package qui a changé plusieurs fois de nom : le nom du package dépend de la version du J.D.K. utilisée :

- com.sun.java.swing : jusqu'à la version 1.1 beta 2 de Swing, de la version 1.1 des JFC et de la version 1.2 beta 4 du J.D.K.
- java.awt.swing : utilisé par le J.D.K. 1.2 beta 2 et 3
- javax.swing : à partir des versions de Swing 1.1 beta 3 et J.D.K. 1.2 RC1

Les composants Swing forment un nouvelle hiérarchie parallèle à celle de l'AWT. L'ancêtre de cette hiérarchie est le composant JComponent. Presque tous ses composants sont écrits en pur Java : ils ne possèdent aucune partie native sauf ceux qui assurent l'interface avec le système d'exploitation : JApplet, JDialog, JFrame, et JWindow. Cela permet aux composants de toujours avoir la même apparence quelque soit le système sur lequel l'application s'exécute.

Tous les composants Swing possèdent les caractéristiques suivantes :

- ce sont des beans
- ce sont des composants légers (pas de partie native) hormis quelques exceptions.
- leurs bords peuvent être changés

La procédure à suivre pour utiliser un composant Swing est identique à celle des composants de la bibliothèque AWT : créer le composant en appelant son constructeur, appeler les méthodes du composant si nécessaire pour le personnaliser et l'ajouter dans un conteneur.

Swing utilise la même infrastructure de classes qu'AWT, ce qui permet de mélanger des composants Swing et AWT dans la même interface. Il est toutefois recommandé d'éviter de les utiliser simultanément car certains peuvent ne pas être restitués correctement.

Les composants Swing utilisent des modèles pour contenir leurs états ou leurs données. Ces modèles sont des classes particulières qui possèdent toutes un comportement par défaut.

45.2. Les packages Swing

Swing contient plusieurs packages :

javax.swing	package principal : il contient les interfaces, les principaux composants, les modèles par défaut
javax.swing.border	Classes représentant les bordures
javax.swing.colorchooser	Classes définissant un composant pour la sélection de couleurs
javax.swing.event	Classes et interfaces pour les événements spécifiques à Swing. Les autres événements sont ceux d'AWT (java.awt.event)
javax.swing.filechooser	Classes définissant un composant pour la sélection de fichiers
javax.swing.plaf	Classes et interfaces génériques pour gérer l'apparence
javax.swing.plaf.basic	Classes et interfaces de base pour gérer l'apparence
javax.swing.plaf.metal	Classes et interfaces pour définir l'apparence Metal qui est l'apparence par défaut
javax.swing.table	Classes définissant un composant pour la présentation de données sous forme de tableau
javax.swing.text	Classes et interfaces de bases pour les composants manipulant du texte
javax.swing.text.html	Classes permettant le support du format HTML
javax.swing.text.html.parser	Classes permettant d'analyser des données au format HTML
javax.swing.text.rtf	Classes permettant le support du format RTF
javax.swing.tree	Classes définissant un composant pour la présentation de données sous forme d'arbre
javax.swing.undo	Classes permettant d'implémenter les fonctions annuler/refaire

45.3. Un exemple de fenêtre autonome

La classe de base d'une application est la classe JFrame. Son rôle est équivalent à la classe Frame de l'AWT et elle s'utilise de la même façon.

Exemple (code Java 1.1) :

```
import javax.swing.*;
import java.awt.event.*;

public class swing1 extends JFrame {
   public swing1() {
      super("titre de l'application");

      WindowListener l = new WindowAdapter() {
         public void windowClosing(WindowEvent e){
            System.exit(0);
         }
      };

      addWindowListener(l);
      setSize(200,100);
      setVisible(true);
   }

   public static void main(String [] args){
      JFrame frame = new swing1();
   }
}
```

45.4. Les composants Swing

Il existe des composants Swing équivalents pour chacun des composants AWT avec des constructeurs semblables. De nombreux constructeurs acceptent comme argument un objet de type Icon, qui représente une petite image généralement stockée au format Gif.

Le constructeur d'un objet Icon admet comme seul paramètre le nom ou l'URL d'un fichier graphique

Exemple (code Java 1.1) :

```
import javax.swing.*;
import java.awt.event.*;

public class swing3 extends JFrame {

  public swing3() {

     super("titre de l'application");

      WindowListener l = new WindowAdapter() {
         public void windowClosing(WindowEvent e){
            System.exit(0);
         }
      };

      addWindowListener(l);

      ImageIcon img = new ImageIcon("tips.gif");
      JButton bouton = new JButton("Mon bouton",img);

      JPanel panneau = new JPanel();
      panneau.add(bouton);
      setContentPane(panneau);
      setSize(200,100);
      setVisible(true);
   }

  public static void main(String [] args){
      JFrame frame = new swing3();
  }
}
```

45.4.1. La classe JFrame

JFrame est l'équivalent de la classe Frame de l'AWT : les principales différences sont l'utilisation du double buffering qui améliore les rafraichissements et l'utilisation d'un panneau de contenu (contentPane) pour insérer des composants (ils ne sont plus insérés sans le JFrame mais dans l'objet contentPane qui lui est associé). Elle représente une fenêtre principale qui possède un titre, une taille modifiable et éventuellement un menu.

La classe possède plusieurs constructeurs :

Constructeur	Rôle
JFrame()	
JFrame(String)	Création d'une instance en précisant le titre

Par défaut, la fenêtre créée n'est pas visible. La méthode setVisible() permet de l'afficher.

Exemple (code Java 1.1) :

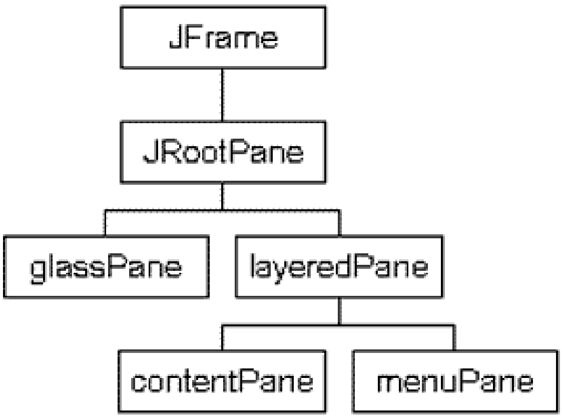
```
import javax.swing.*;

public class TestJFrame1 {

  public static void main(String argv[]) {
    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    f.setVisible(true);
  }
}
```

La gestion des événements est identique à celle utilisée dans l'AWT depuis le J.D.K. 1.1.

Exemple (code Java 1.1) :

```
import javax.swing.*;
import java.awt.event.*;

public class swing2 extends JFrame {

   public swing2() {

     super("titre de l'application");

      WindowListener l = new WindowAdapter() {
         public void windowClosing(WindowEvent e){
           System.exit(0);
         }
      };
     addWindowListener(l);

     JButton bouton = new JButton("Mon bouton");
     JPanel panneau = new JPanel();
     panneau.add(bouton);

     setContentPane(panneau);
     setSize(200,100);
     setVisible(true);
   }

   public static void main(String [] args){
      JFrame frame = new swing2();
   }
}
```

Tous les composants associés à un objet JFrame sont gérés par un objet de la classe JRootPane. Un objet JRootPane contient plusieurs Panes. Tous les composants ajoutés au JFame doivent être ajoutés à un des Pane du JRootPane et non au JFrame directement. C'est aussi à un de ces Panes qu'il faut associer un layout manager si nécessaire.



Le Pane le plus utilisé est le ContentPane. Le Layout manager par défaut du contentPane est BorderLayout. Il est possible de le changer :

Exemple (code Java 1.1) :

```
...
f.getContentPane().setLayout(new FlowLayout());
...
```

Exemple (code Java 1.1) :

```
import javax.swing.*;

public class TestJFrame2 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JButton b =new JButton("Mon bouton");
    f.getContentPane().add(b);
    f.setVisible(true);
  }
}
```

Le JRootPane se compose de plusieurs éléments :

- glassPane : un JPanel par défaut
- layeredPane qui se compose du contentPane (un JPanel par défaut) et du menuBar (un objet de type JMenuBar)

Le glassPane est un JPanel transparent qui se situe au-dessus du layeredPane. Le glassPane peut être n'importe quel composant : pour le modifier il faut utiliser la méthode setGlassPane() en fournissant le composant en paramètre.

Le layeredPane regroupe le contentPane et le menuBar.

Le contentPane est par défaut un JPanel opaque dont le gestionnaire de présentation est un BorderLayout. Ce panel peut être remplacé par n'importe quel composant grâce à la méthode setContentPane().

Attention : il ne faut pas utiliser directement la méthode setLayout() d'un objet JFrame sinon une exception est levée.

Exemple (code Java 1.1) :

```
import javax.swing.*;
import java.awt.*;

public class TestJFrame7 {

  public static void main(String argv[]) {
    JFrame f = new JFrame("ma fenetre");
    f.setLayout(new FlowLayout());
    f.setSize(300,100);
    f.setVisible(true);
  }
}
```

Résultat :

```
C:\swing\code>java TestJFrame7
Exception in thread "main" java.lang.Error: Do not use javax.swing.JFrame.setLay
out() use javax.swing.JFrame.getContentPane().setLayout() instead
        at javax.swing.JFrame.createRootPaneException(Unknown Source)
        at javax.swing.JFrame.setLayout(Unknown Source)
        at TestJFrame7.main(TestJFrame7.java:8)
```

Le menuBar permet d'attacher un menu à la JFrame. Par défaut, le menuBar est vide. La méthode setJMenuBar() permet d'affecter un menu à la JFrame.

Exemple (code Java 1.1) : Création d'un menu très simple

```
import javax.swing.*;
import java.awt.*;

public class TestJFrame6 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JButton b =new JButton("Mon bouton");
    f.getContentPane().add(b);

    JMenuBar menuBar = new JMenuBar();
    f.setJMenuBar(menuBar);

    JMenu menu = new JMenu("Fichier");
    menu.add(menuItem);
    menuBar.add(menu);

    f.setVisible(true);
  }
}
```

45.4.1.1. Le comportement par défaut à la fermeture

Il est possible de préciser comment un objet JFrame, JInternalFrame, ou JDialog réagit à sa fermeture grâce à la méthode setDefaultCloseOperation(). Cette méthode attend en paramètre une valeur qui peut être :

Constante	Rôle
WindowConstants.DISPOSE_ON_CLOSE	détruit la fenêtre
WindowConstants.DO_NOTHING_ON_CLOSE	rend le bouton de fermeture inactif
WindowConstants.HIDE_ON_CLOSE	cache la fenêtre

Cette méthode ne permet pas d'associer d'autres traitements. Dans ce cas, il faut intercepter l'événement et lui associer les traitements.

Exemple (code Java 1.1) : la fenêtre disparaît lors de sa fermeture mais l'application ne se termine pas.

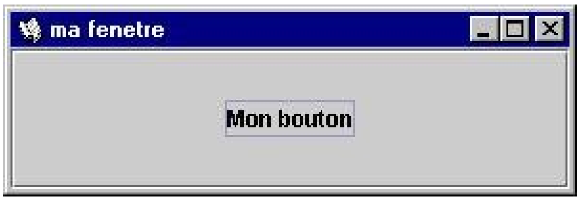
```
import javax.swing.*;

public class TestJFrame3 {
  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JButton b =new JButton("Mon bouton");
    f.getContentPane().add(b);

    f.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);

    f.setVisible(true);
  }
}
```

45.4.1.2. La personnalisation de l'icône

La méthode setIconImage() permet de modifier l'icône de la JFrame.

Exemple (code Java 1.1) :

```
import javax.swing.*;

public class TestJFrame4 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JButton b =new JButton("Mon bouton");
    f.getContentPane().add(b);
    f.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);

    ImageIcon image = new ImageIcon("book.gif");
    f.setIconImage(image.getImage());
    f.setVisible(true);

  }
}
```



Si l'image n'est pas trouvée, alors l'icône est vide. Si l'image est trop grande, elle est redimensionnée.

45.4.1.3. Centrer une JFrame à l'écran

Par défaut, une JFrame est affichée dans le coin supérieur gauche de l'écran. Pour la centrer dans l'écran, il faut procéder comme pour une Frame : déterminer la position de la Frame en fonction de sa dimension et de celle de l'écran et utiliser la méthode setLocation() pour affecter cette position.

Exemple (code Java 1.1) :

```
import javax.swing.*;
import java.awt.*;

public class TestJFrame5 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JButton b =new JButton("Mon bouton");
    f.getContentPane().add(b);

    f.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);

    Dimension dim = Toolkit.getDefaultToolkit().getScreenSize();
    f.setLocation(dim.width/2 - f.getWidth()/2, dim.height/2 - f.getHeight()/2);

    f.setVisible(true);
  }
}
```

45.4.1.4. Les événements associées à un JFrame

La gestion des événements associés à un objet JFrame est identique à celle utilisée pour un objet de type Frame de AWT.

Exemple (code Java 1.1) :

```
import javax.swing.*;
import java.awt.event.*;

public class TestJFrame8 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    f.setVisible(true);
    f.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
          System.exit(0);
        }
      }
    );

  }
}
```

45.4.2. Les étiquettes : la classe JLabel

Le composant JLabel propose les mêmes fonctionnalités que les intitulés AWT mais ils peuvent en plus contenir des icônes .

Cette classe possède plusieurs constructeurs :

Constructeurs	Rôle
JLabel()	Création d'une instance sans texte ni image
JLabel(Icon)	Création d'une instance en précisant l'image
JLabel(Icon, int)	Création d'une instance en précisant l'image et l'alignement horizontal
JLabel(String)	Création d'une instance en précisant le texte
JLabel(String, Icon, int)	Création d'une instance en précisant le texte, l'image et l'alignement horizontal
JLabel(String, int)	Création d'une instance en précisant le texte et l'alignement horizontal

Le composant JLabel permet d'afficher un texte et/ou une icône en précisant leur alignement. L'icône doit être au format GIF et peut être une animation dans ce format.

Exemple (code Java 1.1) :

```
import javax.swing.*;
import java.awt.*;

public class TestJLabel1 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(100,200);

    JPanel pannel = new JPanel();
    JLabel jLabel1 =new JLabel("Mon texte dans JLabel");
    pannel.add(jLabel1);

    ImageIcon icone = new ImageIcon("book.gif");
    JLabel jLabel2 =new JLabel(icone);
    pannel.add(jLabel2);

    JLabel jLabel3 =new JLabel("Mon texte",icone,SwingConstants.LEFT);
    pannel.add(jLabel3);

    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

La classe JLabel définit plusieurs méthodes pour modifier l'apparence du composant :

Méthodes	Rôle
setText()	Permet d'initialiser ou de modifier le texte affiché
setOpaque()	Indique si le composant est transparent (paramètre false) ou opaque (true)
setBackground()	Indique la couleur de fond du composant (setOpaque doit être à true)
setFont()	Permet de préciser la police du texte
setForeGround()	Permet de préciser la couleur du texte
setHorizontalAlignment()	Permet de modifier l'alignement horizontal du texte et de l'icône
setVerticalAlignment()	Permet de modifier l'alignement vertical du texte et de l'icône
setHorizontalTextAlignment()	Permet de modifier l'alignement horizontal du texte uniquement
setVerticalTextAlignment()	Permet de modifier l'alignement vertical du texte uniquement Exemple : jLabel.setVerticalTextPosition(SwingConstants.TOP);
setIcon()	Permet d'assigner une icône
setDisabledIcon()	Permet de définir l'icône associée au JLabel lorsqu'il est désactivé

L'alignement vertical par défaut d'un JLabel est centré. L'alignement horizontal par défaut est soit à droite s'il ne contient que du texte, soit centré s'il contient une image avec ou sans texte. Pour modifier cet alignement, il suffit d'utiliser les méthodes ci-dessus en utilisant des constantes en paramètres : SwingConstants.LEFT, SwingConstants.CENTER, SwingConstants.RIGHT, SwingConstants.TOP, SwingConstants.BOTTOM

Par défaut, un JLabel est transparent : son fond n'est pas dessiné. Pour le dessiner, il faut utiliser la méthode setOpaque() :

Exemple (code Java 1.1) :

```
import javax.swing.*;
import java.awt.*;
```

```
public class TestJLabel2 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");

    f.setSize(100,200);
    JPanel pannel = new JPanel();

    JLabel jLabel1 =new JLabel("Mon texte dans JLabel 1");
    jLabel1.setBackground(Color.red);
    pannel.add(jLabel1);

    JLabel jLabel2 =new JLabel("Mon texte dans JLabel 2");
    jLabel2.setBackground(Color.red);
    jLabel2.setOpaque(true);
    pannel.add(jLabel2);

    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

Dans l'exemple, les 2 JLabel ont le fond rouge demandé par la méthode setBackground(). Seul le deuxième affiche un fond rouge car il est rendu opaque avec la méthode setOpaque().

Il est possible d'associer un raccourci clavier au JLabel qui permet de donner le focus à un autre composant. La méthode setDisplayedMnemonic() permet de définir le raccourci clavier. Celui-ci sera activé en utilisant la touche Alt avec le caractère fourni en paramètre. La méthode setLabelFor() permet d'associer le composant fourni en paramètre au raccourci.

Exemple (code Java 1.1) :

```
import javax.swing.*;
import java.awt.*;

public class TestJLabel3 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JPanel pannel = new JPanel();

    JButton bouton = new JButton("saisir");
    pannel.add(bouton);

    JTextField jEdit = new JTextField("votre nom");

    JLabel jLabel1 =new JLabel("Nom : ");
    jLabel1.setBackground(Color.red);
    jLabel1.setDisplayedMnemonic('n');
    jLabel1.setLabelFor(jEdit);
    pannel.add(jLabel1);
    pannel.add(jEdit);

    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

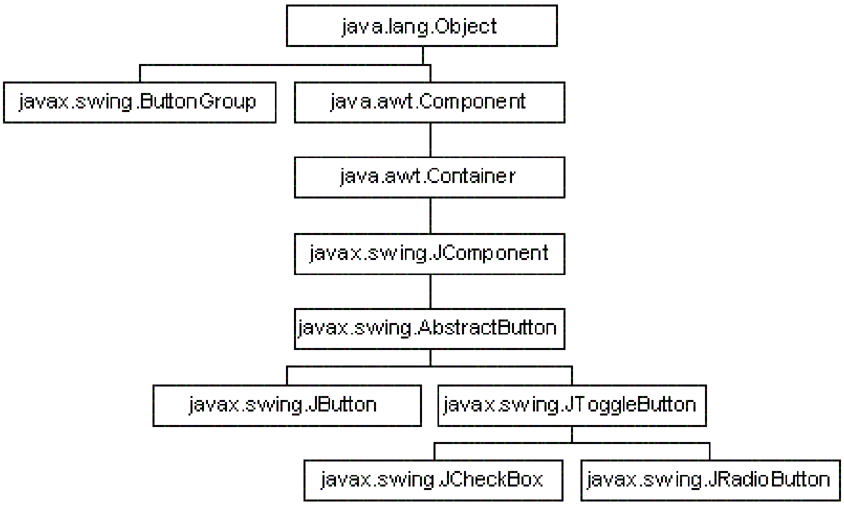
Dans l'exemple, à l'ouverture de la fenêtre, le focus est sur le bouton. Un appui sur Alt+'n' donne le focus au champ de saisie.

45.4.3. Les panneaux : la classe JPanel

La classe JPanel est un conteneur utilisé pour regrouper et organiser des composants grâce à un gestionnaire de présentation (layout manager). Le gestionnaire par défaut d'un JPanel est un objet de la classe FlowLayout.

45.5. Les boutons

Il existe plusieurs boutons définis par Swing.



45.5.1. La classe AbstractButton

C'est une classe abstraite dont héritent les boutons Swing JButton, JMenuItem et JToggleButton.

Cette classe définit de nombreuses méthodes dont les principales sont :

Méthode	Rôle
AddActionListener	Associer un écouteur sur un événement de type ActionEvent
AddChangeListener	Associer un écouteur sur un événement de type ChangeEvent
AddItemListener	Associer un écouteur sur un événement de type ItemEvent
doClick()	Déclencher un clic par programmation
getText()	Obtenir le texte affiché par le composant
setDisabledIcon()	Associer une icône affichée lorsque le composant a l'état désélectionné
setDisabledSelectedIcon()	Associer une icône affichée lors du passage de la souris sur le composant à l'état désélectionné
setEnabled()	Activer/désactiver le composant
setMnemonic()	Associer un raccourci clavier
setPressedIcon()	Associer une icône affichée lorsque le composant est cliqué
setRolloverIcon()	Associer une icône affichée lors du passage de la souris sur le composant
setRolloverSelectedIcon()	Associer une icône affichée lors du passage de la souris sur le composant à l'état sélectionné
setSelectedIcon()	Associer une icône affichée lorsque le composant a l'état sélectionné
setText()	Mettre à jour le texte du composant

isSelected()	Indiquer si le composant est dans l'état sélectionné
setSelected()	Définir l'état du composant (sélectionné ou non selon la valeur fournie en paramètre

Tous les boutons peuvent afficher du texte et/ou une image.

Il est possible de préciser une image différente lors du passage de la souris sur le composant et lors de l'enfoncement du bouton : dans ce cas, il faut créer trois images pour chacun des états (normal, enfoncé et survolé). L'image normale est associée au bouton grâce au constructeur, l'image enfoncée grâce à la méthode setPressedIcon() et l'image lors d'un survol grâce à la méthode setRolloverIcon(). Il suffit enfin d'appeler la méthode setRolloverEnable() avec en paramètre la valeur true.

Exemple (code Java 1.1) :

```
import javax.swing.*;
import java.awt.event.*;

public class swing4 extends JFrame {

   public swing4() {
     super("titre de l'application");

     WindowListener l = new WindowAdapter() {
       public void windowClosing(WindowEvent e){
         System.exit(0);
       }
     };
     addWindowListener(l);

     ImageIcon imageNormale = new ImageIcon("arrow.gif");
     ImageIcon imagePassage =  new ImageIcon("arrowr.gif");
     ImageIcon imageEnfoncee = new ImageIcon("arrowy.gif");

     JButton bouton = new JButton("Mon bouton",imageNormale);
     bouton.setPressedIcon(imageEnfoncee);
     bouton.setRolloverIcon(imagePassage);
     bouton.setRolloverEnabled(true);
     getContentPane().add(bouton, "Center");

     JPanel panneau = new JPanel();
     panneau.add(bouton);
     setContentPane(panneau);
     setSize(200,100);
     setVisible(true);
   }

   public static void main(String [] args){
      JFrame frame = new swing4();
   }
}
```

Un bouton peut recevoir des événements de type ActionEvents (le bouton a été activé), ChangeEvents, et ItemEvents.

Exemple (code Java 1.1) : fermeture de l'application lors de l'activation du bouton

```
import javax.swing.*;
import java.awt.event.*;

public class TestJButton3 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JPanel pannel = new JPanel();

    JButton bouton1 = new JButton("Bouton1");
    bouton1.addActionListener( new ActionListener() {
       public void actionPerformed(ActionEvent e) {
         System.exit(0);
       }
    }
    );

    pannel.add(bouton1);
    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

Pour de plus amples informations sur la gestion des événements, voir le chapitre correspondant.

45.5.2. La classe JButton

JButton est un composant qui représente un bouton : il peut contenir un texte et/ou une icône.

Les constructeurs sont :

Constructeur	Rôle
JButton()	
JButton(String)	préciser le texte du bouton
JButton(Icon)	préciser une icône
JButton(String, Icon)	préciser un texte et une icône

Il ne gère pas d'état. Toutes les indications concernant le contenu du composant JLabel sont valables pour le composant JButton.

Exemple (code Java 1.1) : un bouton avec une image

```
import javax.swing.*;
import java.awt.event.*;

public class swing3 extends JFrame {

   public swing3() {

      super("titre de l'application");

      WindowListener l = new WindowAdapter() {
         public void windowClosing(WindowEvent e){
            System.exit(0);
         }
      };
      addWindowListener(l);

      ImageIcon img = new ImageIcon("tips.gif");
      JButton bouton = new JButton("Mon bouton",img);

      JPanel panneau = new JPanel();
      panneau.add(bouton);
      setContentPane(panneau);
      setSize(200,100);
      setVisible(true);
  }

  public static void main(String [] args){
      JFrame frame = new swing3();
  }
}
```

L'image gif peut être une animation.

Dans un conteneur de type JRootPane, il est possible de définir un bouton par défaut grâce à sa méthode setDefaultButton().

Exemple (code Java 1.1) : définition d'un bouton par défaut dans un JFrame

```
import javax.swing.*;
import java.awt.*;

public class TestJButton2 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JPanel pannel = new JPanel();
    JButton bouton1 = new JButton("Bouton 1");
    pannel.add(bouton1);

    JButton bouton2 = new JButton("Bouton 2");
    pannel.add(bouton2);

    JButton bouton3 = new JButton("Bouton 3");
    pannel.add(bouton3);

    f.getContentPane().add(pannel);
    f.getRootPane().setDefaultButton(bouton3);
    f.setVisible(true);
  }
}
```

Le bouton par défaut est activé par un appui sur la touche Entrée alors que le bouton actif est activé par un appui sur la barre d'espace.

La méthode isDefaultButton() de JButton permet de savoir si le composant est le bouton par défaut.

45.5.3. La classe JToggleButton

Cette classe définit un bouton à deux états : c'est la classe mère des composants JCheckBox et JRadioButton.

La méthode setSelected() héritée de AbstractButton permet de mettre à jour l'état du bouton. La méthode isSelected() permet de connaître cet état.

45.5.4. La classe ButtonGroup

La classe ButtonGroup permet de gérer un ensemble de boutons en garantissant qu'un seul bouton du groupe sera sélectionné.

Pour utiliser la classe ButtonGroup, il suffit d'instancier un objet et d'ajouter des boutons (objets héritant de la classe AbstractButton) grâce à la méthode add(). Il est préférable d'utiliser des objets de la classe JToggleButton ou d'une de ses classes filles car elles sont capables de gérer leurs états.

Exemple (code Java 1.1) :

```
import javax.swing.*;

public class TestGroupButton1 {
  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JPanel pannel = new JPanel();

    ButtonGroup groupe = new ButtonGroup();
    JRadioButton bouton1 = new JRadioButton("Bouton 1");
    groupe.add(bouton1);
    pannel.add(bouton1);
    JRadioButton bouton2 = new JRadioButton("Bouton 2");
    groupe.add(bouton2);
    pannel.add(bouton2);
    JRadioButton bouton3 = new JRadioButton("Bouton 3");
    groupe.add(bouton3);
    pannel.add(bouton3);

    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

45.5.5. Les cases à cocher : la classe JCheckBox

Les constructeurs sont les suivants :

Constructeur	Rôle
JCheckBox(String)	précise l'intitulé
JCheckBox(String, boolean)	précise l'intitulé et l'état
JCheckBox(Icon)	spécifie l'icône utilisée
JCheckBox(Icon, boolean)	précise l'intitulé et l'état du bouton
JCheckBox(String, Icon)	précise l'intitulé et l'icône
JCheckBox(String, Icon, boolean)	précise l'intitulé, une icône et l'état

Un groupe de cases à cocher peut être défini avec la classe ButtonGroup. Dans ce cas, un seul composant du groupe peut être sélectionné. Pour l'utiliser, il faut créer un objet de la classe ButtonGroup et utiliser la méthode add() pour ajouter un composant au groupe.

Exemple (code Java 1.1) :

```
import javax.swing.*;

public class TestJCheckBox1 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JPanel pannel = new JPanel();

    JCheckBox bouton1 = new JCheckBox("Bouton 1");
    pannel.add(bouton1);
    JCheckBox bouton2 = new JCheckBox("Bouton 2");
    pannel.add(bouton2);
    JCheckBox bouton3 = new JCheckBox("Bouton 3");
    pannel.add(bouton3);

    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

45.5.6. Les boutons radio : la classe JRadioButton

Un objet de type JRadioButton représente un bouton radio d'un groupe de boutons . A un instant donné, un seul des boutons radio associés à un même groupe peut être sélectionné. La classe JRadioButton hérite de la classe AbstractButton.

Un bouton radio possède un libellé et éventuellement une icône qui peut être précisée, pour chacun des états du bouton, en utilisant les méthodes setIcon(), setSelectedIcon() et setPressedIcon().

Exemple (code Java 1.1) :

```
import javax.swing.*;

public class TestJRadioButton1 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JPanel pannel = new JPanel();
    JRadioButton bouton1 = new JRadioButton("Bouton 1");
    pannel.add(bouton1);
    JRadioButton bouton2 = new JRadioButton("Bouton 2");
    pannel.add(bouton2);
    JRadioButton bouton3 = new JRadioButton("Bouton 3");
    pannel.add(bouton3);

    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

La méthode isSelected() permet de savoir si le bouton est sélectionné ou non.

La classe JRadioButton possède plusieurs constructeurs :

Constructeur	Rôle
JRadioButton()	Créer un bouton non sélectionné sans libellé
JRadioButton(Icon)	Créer un bouton non sélectionné sans libellé avec l'icône fournie en paramètre
JRadioButton(Icon, boolean)	Créer un bouton sans libellé avec l'icône et l'état fournis en paramètres
JRadioButton(String)	Créer un bouton non sélectionné avec le libellé fourni en paramètre
JRadioButton(String, boolean)	Créer un bouton avec le libellé et l'état fournis en paramètres
JRadioButton(String, Icon)	Créer un bouton non sélectionné avec le libellé et l'icône fournis en paramètres
JRadioButton(String, Icon, boolean)	Créer un bouton avec le libellé, l'icône et l'état fournis en paramètres

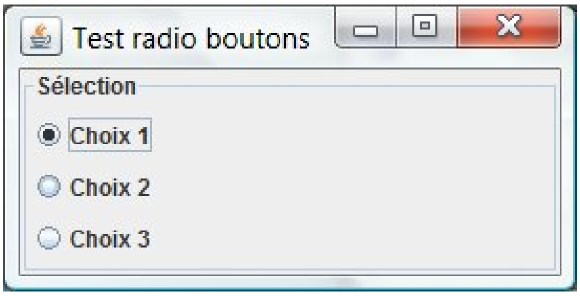
Un groupe de boutons radio est encapsulé dans un objet de type ButtonGroup.

Il faut ajouter tous les JRadioButton du groupe en utilisant la méthode add() de la classe ButtonGroup. Lors de la sélection d'un bouton, c'est l'objet de type ButtonGroup qui se charge de déselectionner le bouton précédemment sélectionné dans le groupe.

Un groupe n'a pas l'obligation d'avoir un bouton sélectionné.

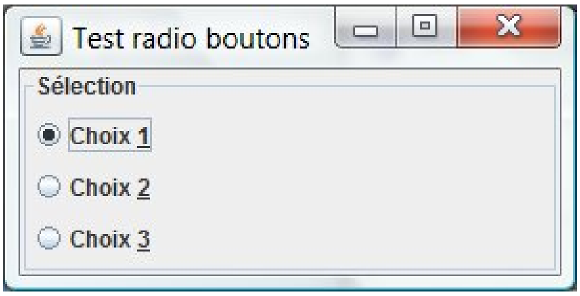
Exemple :

```
import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.GridLayout;
import javax.swing.BorderFactory;
import javax.swing.ButtonGroup;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JRadioButton;
import javax.swing.border.Border;

public class TestJRadioButton extends JFrame {
  public static void main(String args[]) {
    TestJRadioButton app = new TestJRadioButton();
    app.init();
  }

  public void init() {
    this.setTitle("Test radio boutons");

    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JPanel panel = new JPanel(new GridLayout(0,1));
    Border border = BorderFactory.createTitledBorder("Sélection");
    panel.setBorder(border);
    ButtonGroup group = new ButtonGroup();
    JRadioButton radio1 = new JRadioButton("Choix 1", true);
    JRadioButton radio2 = new JRadioButton("Choix 2");
    JRadioButton radio3 = new JRadioButton("Choix 3");
    group.add(radio1);
    panel.add(radio1);
    group.add(radio2);
    panel.add(radio2);
    group.add(radio3);
    panel.add(radio3);
    Container contentPane = this.getContentPane();
    contentPane.add(panel, BorderLayout.CENTER);
    this.setSize(300, 150);
    this.setVisible(true);
  }
}
```



Exemple :

```
import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.GridLayout;
import java.awt.event.KeyEvent;
import javax.swing.BorderFactory;
import javax.swing.ButtonGroup;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JRadioButton;
import javax.swing.border.Border;

public class TestJRadioButton extends JFrame {
  public static void main(String args[]) {
    TestJRadioButton app = new TestJRadioButton();
    app.init();
  }

  public void init() {
    this.setTitle("Test radio boutons");
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JPanel panel = new JPanel(new GridLayout(0, 1));
    Border border = BorderFactory.createTitledBorder("Sélection");
    panel.setBorder(border);

    ButtonGroup group = new ButtonGroup();
    JRadioButton radio1 = new JRadioButton("Choix 1");
    radio1.setMnemonic(KeyEvent.VK_1);
    radio1.setActionCommand("Choix_1");
    radio1.setSelected(true);

    JRadioButton radio2 = new JRadioButton("Choix 2");
    radio2.setMnemonic(KeyEvent.VK_2);
    radio2.setActionCommand("Choix_2");

    JRadioButton radio3 = new JRadioButton("Choix 3");
    radio3.setMnemonic(KeyEvent.VK_3);
    radio3.setActionCommand("Choix_3");

    group.add(radio1);
    panel.add(radio1);
    group.add(radio2);
    panel.add(radio2);
    group.add(radio3);
    panel.add(radio3);

    Container contentPane = this.getContentPane();
    contentPane.add(panel, BorderLayout.CENTER);
    this.setSize(300, 150);
    this.setVisible(true);
  }
}
```



Lors de la sélection d'un bouton du groupe, il y a plusieurs événements qui peuvent être émis :

- Un événement de type Action
- Un événement de type Item émis par le bouton sélectionné
- Un événement de type Item émis par le bouton désélectionné s'il y en a un

Exemple :

```
import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import java.awt.event.KeyEvent;
import javax.swing.BorderFactory;
import javax.swing.ButtonGroup;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JRadioButton;
import javax.swing.border.Border;

public class TestJRadioButton extends JFrame implements ActionListener, ItemListener {
  public static void main(String args[]) {
    TestJRadioButton app = new TestJRadioButton();
    app.init();
  }

  public void init() {
    this.setTitle("Test radio boutons");

    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JPanel panel = new JPanel(new GridLayout(0, 1));
    Border border = BorderFactory.createTitledBorder("Sélection");
    panel.setBorder(border);
    ButtonGroup group = new ButtonGroup();

    JRadioButton radio1 = new JRadioButton("Choix 1");
    radio1.setMnemonic(KeyEvent.VK_1);
    radio1.setActionCommand("Choix_1");
    radio1.setSelected(true);

    JRadioButton radio2 = new JRadioButton("Choix 2");
    radio2.setMnemonic(KeyEvent.VK_2);
    radio2.setActionCommand("Choix_2");

    JRadioButton radio3 = new JRadioButton("Choix 3");
    radio3.setMnemonic(KeyEvent.VK_3);
    radio3.setActionCommand("Choix_3");

    group.add(radio1);
    panel.add(radio1);
    group.add(radio2);
    panel.add(radio2);
    group.add(radio3);
    panel.add(radio3);

    radio1.addActionListener(this);
    radio2.addActionListener(this);
    radio3.addActionListener(this);
    radio1.addItemListener(this);
    radio2.addItemListener(this);
    radio3.addItemListener(this);

    Container contentPane = this.getContentPane();
    contentPane.add(panel, BorderLayout.CENTER);
    this.setSize(300, 150);
    this.setVisible(true);
  }

  @Override
  public void actionPerformed(ActionEvent e) {
    System.out.println("Clic sur le bouton : " + e.getActionCommand());
  }

  @Override
  public void itemStateChanged(ItemEvent e) {
    System.out.print("Bouton " + ((JRadioButton) e.getItem()).getActionCommand());
    if (e.getStateChange() == ItemEvent.DESELECTED)
      System.out.println(" deselectionne");
    if (e.getStateChange() == ItemEvent.SELECTED)
      System.out.println(" selectionne");
  }
}
```

La méthode getSelection() de la classe ButtonGroup renvoie le modèle du bouton radio sélectionné encapsulé dans un objet de type ButtonModel.

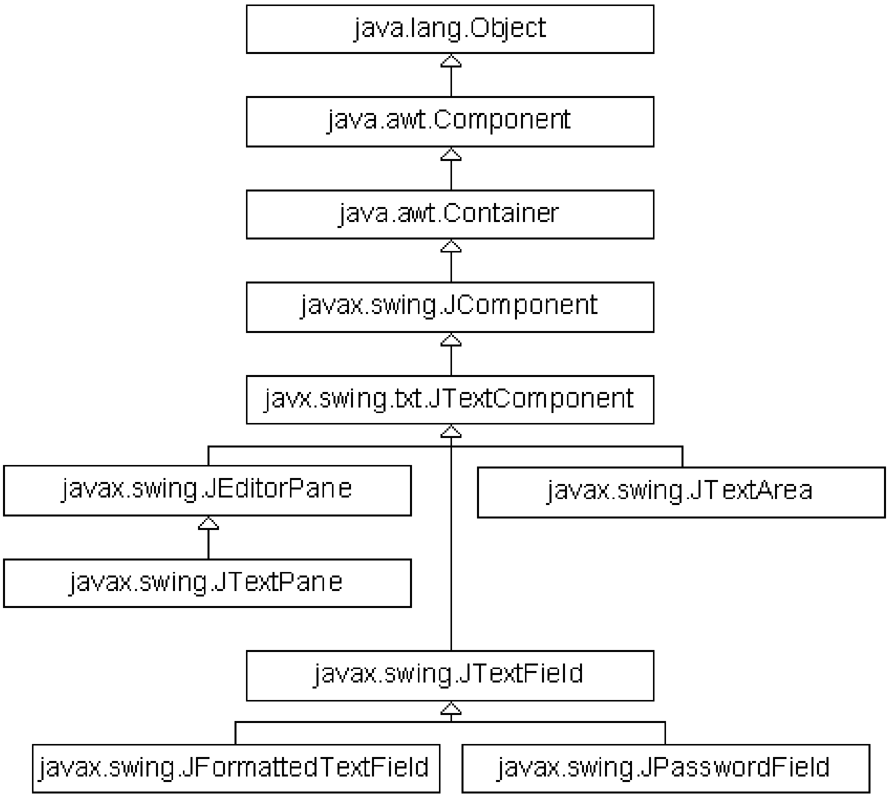
Pour déterminer le bouton sélectionné, il faut parcourir les boutons du groupe et comparer leurs modèles.

Exemple :

```
  public static JRadioButton getBoutonSelectionne(ButtonGroup group) {
    JRadioButton result = null;
    for (Enumeration e = group.getElements(); e.hasMoreElements();) {
      JRadioButton bouton = (JRadioButton) e.nextElement();
      if (bouton.getModel() == group.getSelection()) {
        result = bouton;
        break;
      }
    }
    return result;
  }
```

45.6. Les composants de saisie de texte

Swing possède plusieurs composants pour permettre la saisie de texte.



45.6.1. La classe JTextComponent

La classe abstraite JTextComponent est la classe mère de tous les composants permettant la saisie de texte.

Les données du composant (le modèle dans le motif de conception MVC) sont encapsulées dans un objet qui implémente l'interface Document. Deux classes implémentant cette interface sont fournies en standard : PlainDocument pour du texte simple et StyledDocument pour du texte riche pouvant contenir entre autres plusieurs polices de caractères, des couleurs, des images, ...

La classe JTextComponent possède de nombreuses méthodes dont les principales sont :

Méthode	Rôle
void copy()	Copier le contenu du texte et le mettre dans le presse papier système
void cut()	Couper le contenu du texte et le mettre dans le presse papier système
Document getDocument()	Renvoyer l'objet de type Document qui encapsule le texte saisi
String getSelectectedText()	Renvoyer le texte sélectionné dans le composant
int getSelectionEnd()	Renvoyer la position de la fin de la sélection
int getSelectionStart()	Renvoyer la position du début de la sélection

String getText()	Renvoyer le texte saisi
String getText(int, int)	Renvoyer une portion du texte débutant à partir de la position donnée par le premier paramètre et la longueur donnée dans le second paramètre
bool isEditable()	Renvoyer un booléen qui précise si le texte est éditable ou non
void paste()	Coller le contenu du presse papier système dans le composant
void select(int,int)	Sélectionner une portion du texte dont les positions de début et de fin sont fournies en paramètres
void setCaretPosition(int)	Déplacer le curseur dans le texte à la position précisé en paramètre
void setEditable(boolean)	Permet de préciser si les données du composant sont éditables ou non
void setSelectionEnd(int)	Modifier la position de la fin de la sélection
void setSelectionStart(int)	Modifier la position du début de la sélection
void setText(String)	Modifier le contenu du texte

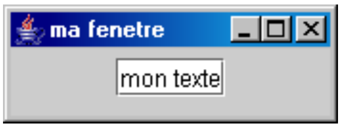
Toutes ces méthodes sont donc accessibles grâce à l'héritage pour tous les composants de saisie de texte proposés par Swing.

45.6.2. La classe JTextField

La classe javax.Swing.JTextField est un composant qui permet la saisie d'une seule ligne de texte simple. Son modèle utilise un objet de type PlainDocument.

Exemple (code Java 1.1) :

```
import javax.swing.*;

public class JTextField1 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300, 100);
    JPanel pannel = new JPanel();

    JTextField  testField1 = new JTextField ("mon texte");

    pannel.add(testField1);
    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```



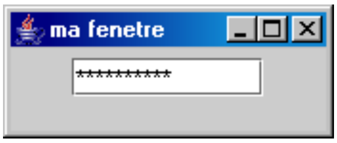
La propriété horizontalAligment permet de préciser l'alignement du texte dans le composant en utilisant les valeurs JTextField.LEFT , JTextField.CENTER ou JTextField.RIGHT.

45.6.3. La classe JPasswordField

La classe JPasswordField permet la saisie d'un texte dont tous les caractères saisis seront affichés sous la forme d'un caractère particulier ('*' par défaut). Cette classe hérite de la classe JTextField.

Exemple (code Java 1.1) :

```
import java.awt.Dimension;

import javax.swing.*;

public class JPasswordField1 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300, 100);
    JPanel pannel = new JPanel();

    JPasswordField  passwordField1 = new JPasswordField ("");
    passwordField1.setPreferredSize(new Dimension(100,20 ));

    pannel.add(passwordField1);
    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```



La méthode setEchoChar(char) permet de préciser le caractère qui sera montré lors de la saisie.

Il ne faut pas utiliser la méthode getText() qui est déclarée deprecated mais la méthode getPassword() pour obtenir la valeur du texte saisi.

Exemple (code Java 1.1) :

```
import java.awt.Dimension;
import java.awt.event.*;

import javax.swing.*;

public class JPasswordField2 implements ActionListener {

  JPasswordField passwordField1 = null;

  public static void main(String argv[]) {
    JPasswordField2 jpf2 = new JPasswordField2();
    jpf2.init();
  }

  public void init() {
    JFrame f = new JFrame("ma fenetre");
    f.setSize(300, 100);
    JPanel pannel = new JPanel();

    passwordField1 = new JPasswordField("");
    passwordField1.setPreferredSize(new Dimension(100, 20));
    pannel.add(passwordField1);

    JButton bouton1 = new JButton("Afficher");
    bouton1.addActionListener(this);

    pannel.add(bouton1);
    f.getContentPane().add(pannel);
    f.setVisible(true);
  }

  public void actionPerformed(ActionEvent e) {

    System.out.println("texte saisie = " + String.copyValueOf(passwordField1.getPassword()));
  }
}
```

Les méthodes copy() et cut() sont redéfinies pour n'émettre qu'un bip. Elles empêchent l'exportation du contenu du champ.

45.6.4. La classe JFormattedTextField

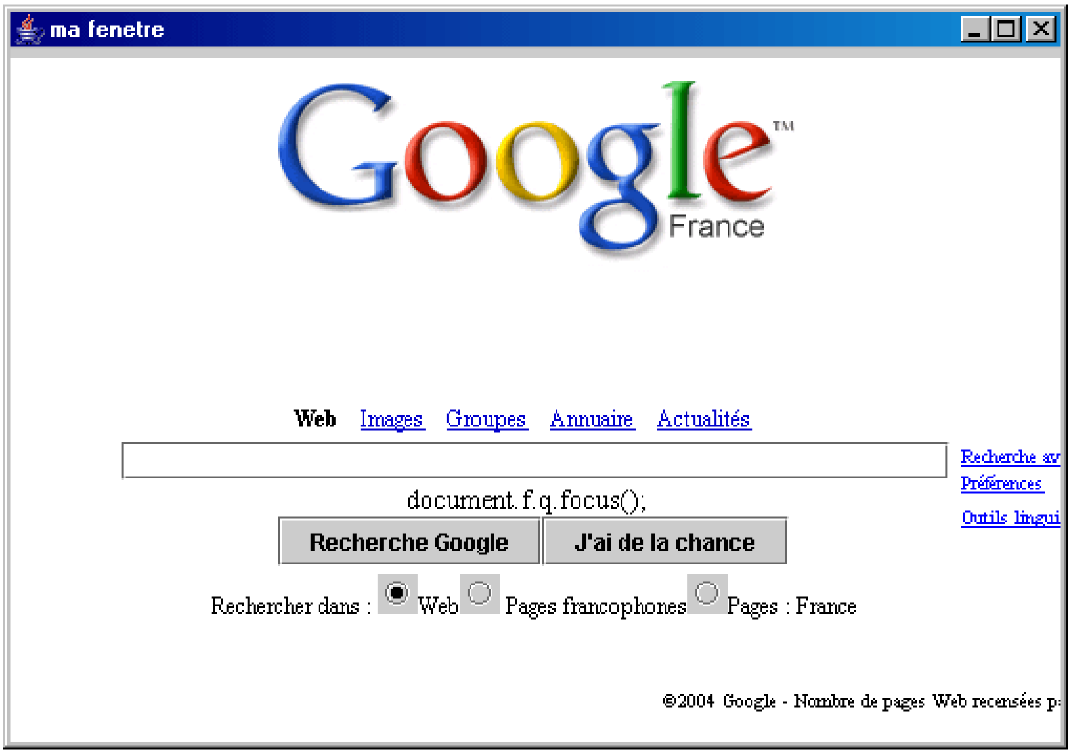
Le JDK 1.4 propose la classe JFormattedTextField pour faciliter la création d'un composant de saisie personnalisé. Cette classe hérite de la classe JTextField.

45.6.5. La classe JEditorPane

Ce composant permet la saisie de texte riche multilignes. Ce type de texte peut contenir des informations de mise en pages et de formatage. En standard, Swing propose le support des formats RTF et HTML.

Exemple (code Java 1.1) : affichage de la page de Google avec gestion des hyperliens

```
import java.net.URL;
import javax.swing.*;
import javax.swing.event.*;

public class JEditorPane1 {

  public static void main(String[] args) {
    final JEditorPane editeur;
    JPanel pannel = new JPanel();

    try {
      editeur = new JEditorPane(new URL("http://google.fr"));
      editeur.setEditable(false);
      editeur.addHyperlinkListener(new HyperlinkListener() {
        public void hyperlinkUpdate(HyperlinkEvent e) {
          if (e.getEventType() == HyperlinkEvent.EventType.ACTIVATED) {
            URL url = e.getURL();
            if (url == null)
              return;
            try {
              editeur.setPage(e.getURL());
            } catch (Exception ex) {
              ex.printStackTrace();
            }
          }
        }
      });

      pannel.add(editeur);
    } catch (Exception e1) {
      e1.printStackTrace();
    }
    JFrame f = new JFrame("ma fenetre");
    f.setSize(500, 300);

    f.getContentPane().add(pannel);
    f.setVisible(true);

  }
}
```



document.f.q.focus();

Rechercher dans : ● Web ○ Pages francophones ○ Pages : France

©2004 Google - Nombre de pages Web recensées p

45.6.6. La classe JTextPane

La suite de cette section sera développée dans une version future de ce document

45.6.7. La classe JTextArea

La classe JTextArea est un composant qui permet la saisie de texte simple en mode multiligne. Le modèle utilisé par ce composant est le PlainDocument : il ne peut donc contenir que du texte brut sans éléments multiples de formatage.

JTexteArea propose plusieurs méthodes pour ajouter du texte dans son modèle :

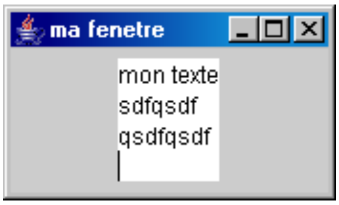
- soit fournir le texte en paramètre du constructeur utilisé
- soit utiliser la méthode setText() qui permet d'initialiser le texte du composant
- soit utiliser la méthode append() qui permet d'ajouter du texte à la fin de celui contenu dans le composant
- soit utiliser la méthode insert() qui permet d'insérer du texte dans le composant à une position données en caractères

La méthode replaceRange() permet de remplacer la partie de texte occupant les index donnés en paramètres par la chaîne fournie.

La propriété rows permet de définir le nombre de lignes affichées par le composant : cette propriété peut donc être modifiée lors d'un redimensionnement du composant. La propriété lineCount en lecture seule permet de savoir le nombre de lignes qui composent le texte. Il ne faut pas confondre ces deux propriétés.

Exemple (code Java 1.1) :

```
import javax.swing.*;

public class JTextArea1 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300, 100);
    JPanel pannel = new JPanel();

    JTextArea  textArea1 = new JTextArea ("mon texte");

    pannel.add(textArea1);
    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

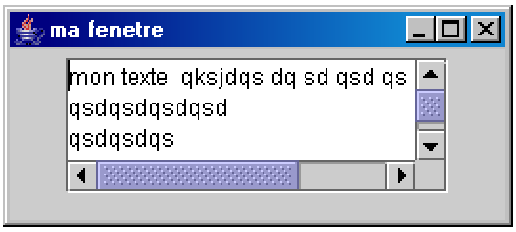


```
mon texte
sdfqsdf
qsdfqsdf
```

Par défaut, la taille du composant augmente au fur et à mesure de l'augmentation de la taille du texte qu'il contient. Pour éviter cet effet, il faut encapsuler le JTextArea dans un JScrollPane.

Exemple (code Java 1.1) :

```
import java.awt.Dimension;
import javax.swing.*;

public class JTextArea1 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300, 100);
    JPanel pannel = new JPanel();
    JTextArea  textArea1 = new JTextArea ("mon texte");
    JScrollPane scrollPane = new JScrollPane(textArea1);
    scrollPane.setPreferredSize(new Dimension(200,70));
    pannel.add(scrollPane);
    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```



```
mon texte  qksjdqs dq sd qsd qs
qsdqsdqsdqsd
qsdqsdqs
```

45.7. Les onglets

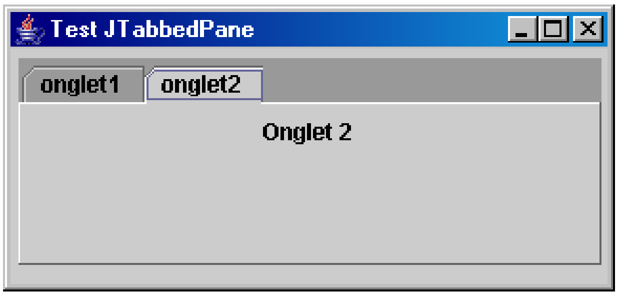
La classe javax.swing.JTabbedPane encapsule un ensemble d'onglets. Chaque onglet est constitué d'un titre, d'un composant et éventuellement d'une image.

Pour utiliser ce composant, il faut :

- instancier un objet de type JTabbedPane
- créer le composant de chaque onglet
- ajouter chaque onglet à l'objet JTabbedPane en utilisant la méthode addTab()

Exemple (code Java 1.1) :

```
import java.awt.Dimension;
import java.awt.event.KeyEvent;

import javax.swing.*;

public class TestJTabbedPane1 {

  public static void main(String[] args) {
    JFrame f = new JFrame("Test JTabbedPane");
    f.setSize(320, 150);
    JPanel pannel = new JPanel();

    JTabbedPane onglets = new JTabbedPane(SwingConstants.TOP);

    JPanel onglet1 = new JPanel();
    JLabel titreOnglet1 = new JLabel("Onglet 1");
    onglet1.add(titreOnglet1);
    onglet1.setPreferredSize(new Dimension(300, 80));
    onglets.addTab("onglet1", onglet1);

    JPanel onglet2 = new JPanel();
    JLabel titreOnglet2 = new JLabel("Onglet 2");
    onglet2.add(titreOnglet2);
    onglets.addTab("onglet2", onglet2);

    onglets.setOpaque(true);
    pannel.add(onglets);
    f.getContentPane().add(pannel);
    f.setVisible(true);

  }
}
```



A partir du JDK 1.4, il est possible d'ajouter un raccourci clavier sur chacun des onglets en utilisant la méthode setMnemonicAt(). Cette méthode attend deux paramètres : l'index de l'onglet concerné (le premier commence à 0) et la touche du clavier associée sous la forme d'une constance KeyEvent.VK_xxx. Pour utiliser ce raccourci, il suffit d'utiliser la touche désignée en paramètre de la méthode avec la touche Alt.

La classe JTabbedPane possède plusieurs méthodes qui permettent de définir le contenu de l'onglet :

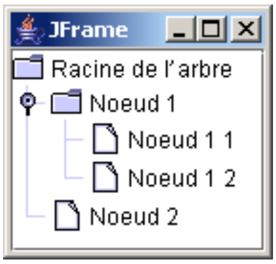
Méthodes	Rôles
addTab(String, Component)	Permet d'ajouter un nouvel onglet dont le titre et le composant sont fournis en paramètres. Cette méthode possède plusieurs surcharges qui permettent de préciser une icône et une bulle d'aide
insertTab(String, Icon, Component, String, index)	Permet d'insérer un onglet dont la position est précisée dans le dernier paramètre

remove(int)	Permet de supprimer l'onglet dont l'index est fourni en paramètre
setTabPlacement	Permet de préciser le positionnement des onglets dans le composant JTabbedPane. Les valeurs possibles sont les constantes TOP, BOTTOM, LEFT et RIGHT définies dans la classe JTabbedPane.

La méthode getSelectedIndex() permet d'obtenir l'index de l'onglet courant. La méthode setSelectedIndex() permet de définir l'onglet courant.

45.8. Le composant JTree

Le composant JTree permet de présenter des données sous une forme hiérarchique arborescente.



Au premier abord, le composant JTree peut sembler compliqué à mettre en oeuvre mais la compréhension de son mode de fonctionnement peut grandement faciliter son utilisation.

Il utilise le modèle MVC en proposant une séparation des données (data models) et du rendu de ces données (cell renderers).

Dans l'arbre, les éléments qui ne possèdent pas d'élément fils sont des feuilles (leaf). Chaque élément est associé à un objet (user object) qui va permettre de déterminer le libellé affiché dans l'arbre en utilisant la méthode toString().

45.8.1. La création d'une instance de la classe JTree

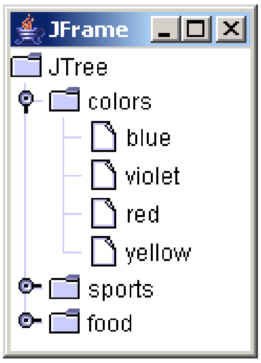
La classe JTree possède 7 constructeurs. Tous ceux qui attendent au moins un paramètre acceptent une collection pour initialiser tout ou partie du modèle de données de l'arbre :

public JTree();
public JTree(Hashtable value);
public JTree(Vector value);
public JTree(Object[] value);
public JTree(TreeModel model);
public JTree(TreeNode rootNode);
public JTree(TreeNode rootNode, boolean askAllowsChildren);

Lorsqu'une instance de JTree est créée avec le constructeur par défaut, l'arbre obtenu contient des données par défaut.

Exemple (code Java 1.1) :

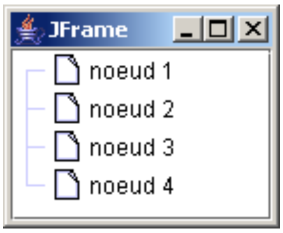
```
import javax.swing.JFrame;
import javax.swing.JTree;

public class TestJtree extends JFrame {

  private javax.swing.JPanel jContentPane = null;
  private JTree              jTree        = null;

  private JTree getJTree() {
    if (jTree == null) {
      jTree = new JTree();
    }
    return jTree;
  }

  public static void main(String[] args) {
    TestJtree testJtree = new TestJtree();
    testJtree.setVisible(true);
  }

  public TestJtree() {
    super();
    initialize();
  }

  private void initialize() {
    this.setSize(300, 200);
    this.setContentPane(getJContentPane());
    this.setTitle("JFrame");
  }

  private javax.swing.JPanel getJContentPane() {
    if (jContentPane == null) {
      jContentPane = new javax.swing.JPanel();
      jContentPane.setLayout(new java.awt.BorderLayout());
      jContentPane.add(getJTree(), java.awt.BorderLayout.CENTER);
    }
    return jContentPane;
  }
}
```



Les trois constructeurs qui attendent en paramètre une collection permettent de créer un arbre avec une racine non affichée qui va contenir comme noeuds fils directs tous les éléments contenus dans la collection.

Exemple (code Java 1.1) :

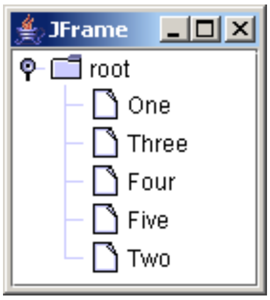
```
      String[] racine = {"noeud 1","noeud 2","noeud3","noeud 4"};
      jTree = new JTree(racine);
```



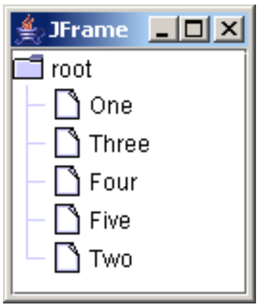
Dans ce cas, la racine n'est pas affichée. Pour l'afficher, il faut utiliser la méthode setRootVisible()

Exemple (code Java 1.1) :

```
jTree.setRootVisible(true);
```



Dans ce cas elle se nomme root et possède un commutateur qui permet de refermer ou d'étendre la racine. Pour supprimer ce commutateur, il faut utiliser la méthode jTree.setShowsRootHandles(false)



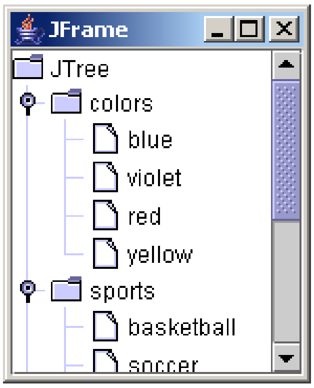
L'utilisation de l'une ou l'autre des collections n'est pas équivalente. Par exemple, l'utilisation d'une hashtable ne garantit pas l'ordre des noeuds puisque par définition cette collection ne gère pas un ordre précis.

Généralement, la construction d'un arbre utilise un des constructeurs qui attend en paramètre un objet de type TreeModel ou TreeNode car ces deux objets permettent d'avoir un contrôle sur l'ensemble des données de l'arbre. Leur utilisation sera détaillée dans la section consacrée à la gestion des données de l'arbre.

En fonction du nombre d'éléments et de l'état étendu ou non d'un ou plusieurs éléments, la taille de l'arbre peut varier : il est donc nécessaire d'inclure le composant JTree dans un composant JScrollPane

Exemple (code Java 1.1) :

```
...

  private JScrollPane        jScrollPane  = null;

...

  private JScrollPane getJScrollPane() {
    if (jScrollPane == null) {
      jScrollPane = new JScrollPane();
      jScrollPane.setViewportView(getJTree());
    }
    return jScrollPane;
  }

...

  private javax.swing.JPanel getJContentPane() {
    if (jContentPane == null) {
      jContentPane = new javax.swing.JPanel();
      jContentPane.setLayout(new java.awt.BorderLayout());
      jContentPane.add(getJScrollPane(), java.awt.BorderLayout.CENTER);
    }
    return jContentPane;
  }
```



L'utilisateur peut sélectionner un noeud en cliquant sur son texte ou son icône. Un double clic sur le texte ou l'icône d'un noeud permet de l'étendre ou le refermer selon son état.

45.8.2. La gestion des données de l'arbre

Chaque arbre commence par un noeud racine. Par défaut, la racine et ses noeuds fils directs sont visibles. Chaque noeud de l'arbre peut avoir zéro ou plusieurs noeuds fils. Un noeud sans noeud fils est appelé une feuille de l'arbre (leaf)

En application du modèle MVC, le composant JTree ne gère pas directement chaque noeud et la façon dont ceux-ci sont organisés et stockés mais il utilise un objet dédié de type TreeModel.

Ainsi, comme dans d'autres composants Swing, le composant JTree manipule des objets implémentant des interfaces. Une classe qui encapsule les données de l'arbre doit implémenter l'interface TreeModel. Chaque noeud de l'arbre doit implémenter l'interface TreeNode.

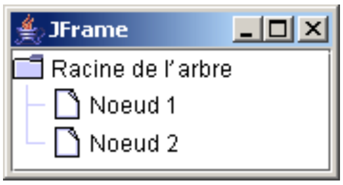
Pour préciser les données contenues dans l'arbre, il faut créer un objet qui va encapsuler ces données et les passer au constructeur de la classe Jtree. Cet objet peut être de type TreeNode ou TreeModel. Un TreeModel stocke les données de chaque noeud dans un objet de type TreeNode.

Généralement, le plus simple est de définir un type TreeNode personnalisé. Swing propose pour cela l'objet DefaultMutableTreeNode. il suffit d'en créer une instance pour stocker les données et l'utiliser lors de l'appel du constructeur de la classe JTree.

La classe DefaultMutableTreeNode implémente l'interface MutableTreeNode qui elle-même hérite de l'interface TreeNode

Exemple (code Java 1.1) :

```
import javax.swing.tree.DefaultMutableTreeNode;

...

  private JTree getJTree() {

    if (jTree == null) {
      DefaultMutableTreeNode racine = new DefaultMutableTreeNode("Racine de l'arbre");
      DefaultMutableTreeNode noeud1 = new DefaultMutableTreeNode("Noeud 1");
      racine.add(noeud1);
      DefaultMutableTreeNode noeud2 = new DefaultMutableTreeNode("Noeud 2");
      racine.add(noeud2);
      jTree = new JTree(racine);
    }
    return jTree;
  }
```



Dans ce cas, une instance de la classe DefaultTreeModel est créée avec la racine fournie en paramètre du constructeur de la classe JTree.

Une autre solution permet de créer une instance de la classe DefaultTreeModel et de la passer en paramètre du constructeur de la classe JTree.

La méthode setModel() de la classe JTree permet d'associer un modèle de données à l'arbre.

45.8.2.1. L'interface TreeNode

Chaque noeud de l'arbre stocké dans le modèle de données implémente l'interface TreeNode.

Cette interface définit 7 méthodes dont la plupart concernent les relations entre les noeuds :

Méthode	Rôle
Enumeration children()	renvoie une collection des noeuds fils
boolean getAllowsChildren()	Renvoie un booléen qui precise si le noeud peut avoir des noeuds fils
TreeNode getChildAt(int index)	Renvoie le noeud fils correspondant à l'index fourni en paramètre
int getChildCount()	renvoie le nombre de noeuds fils directs du noeud
int getIndex(TreeNode child)	renvoie l'index du noeud passé en paramètre
TreeNode getParent()	renvoie le noeud père
boolean isLeaf()	renvoie un booléen qui précise si le noeud est une feuille

Chaque noeud ne peut avoir qu'un seul père (hormis le noeud racine qui ne possède pas de père) et autant de noeuds fils que souhaité. La méthode getParent() permet de renvoyer le noeud père. Elle renvoie null lorsque cette méthode est appelée sur le noeud racine.

La méthode getChildCount() renvoie le nombre de noeuds fils directs du noeud.

La méthode getAllowsChildren() permet de préciser si le noeud peut avoir des noeuds enfants : si elle renvoie false alors le noeud sera toujours une feuille et ne pourra donc jamais avoir de noeuds fils.

La méthode isLeaf() renvoie un booléen précisant si le noeud est une feuille ou non. Une feuille est un noeud qui ne possède pas de noeud fils.

Les noeuds fils sont ordonnés car l'ordre de représentation des données peut être important dans la représentation de données hiérarchiques. La méthode getChildAt() renvoie le noeud fils dont l'index est fourni en paramètre de la méthode. La méthode getIndex() renvoie l'index du noeud fils passé en paramètre.

45.8.2.2. L'interface MutableTreeNode

Les 7 méthodes définies par l'interface TreeNode ne permettent que de lire des valeurs. Pour mettre à jour un noeud, il est nécéssaire d'utiliser l'interface MutableTreeNode qui hérite de la méthode TreeNode. Elle définit en plus plusieurs méthodes permettant de mettre à jour le noeud.

void insert(MutableTreeNode child, int index);
void remove(int index);
void remove(MutableTreeNode node);
void removeFromParent();
void setParent(MutableTreeNode parent);
void setUserObject(Object userObject);

La méthode insert() permet d'ajouter le noeud fourni en paramètre comme noeud fils à la position précisée par le second paramètre.

Il existe deux surcharges de la méthode remove() qui permettent de déconnecter un noeud fils de son père. La première surcharge attend en paramètre l'index du noeud fils. La seconde surcharge attend en paramètre le noeud à déconnecter. Dans tous les cas, il est nécessaire d'utiliser cette méthode sur le noeud père.

La méthode removeFromParent() appelée à partir d'un noeud permet de supprimer le lien entre le noeud et son père.

La méthode setParent() permet de préciser le père du noeud.

La méthode setUserObject() permet d'associer un objet au noeud. L'appel à la méthode toString() de cet objet permettra de déterminer le libellé du noeud qui sera affiché.

45.8.2.3. La classe DefaultMutableTreeNode

Généralement, les noeuds créés dans le modèle sont des instances de la classe DefaultMutableTreeNode. Cette classe implémente l'interface MutableTreeNode ce qui permet d'obtenir une instance d'un noeud modifiable.

Le plus souvent, les noeuds fournis en paramètres des méthodes proposées par Swing sont de type TreeNode. Si l'instance du noeud est de type DefaultTreeNode, il est possible de faire un cast pour accéder à toutes ses méthodes.

La classe propose trois constructeurs dont deux attendent en paramètre l'objet qui sera associé au noeud. L'un des deux attend en plus un booléen qui permet de préciser si le noeud peut avoir des noeuds fils.

Constructeur	Rôle
public DefaultMutableTreeNode()	Créer un noeud sans objet associé. Cette association pourra être faite avec la méthode setObject()
public DefaultMutableTreeNode(Object userObject)	Créer un noeud en précisant l'objet qui lui sera associé et qui pourra avoir des noeuds fils
public DefaultMutableTreeNode(Object userObject, boolean allowsChildren)	Créer un noeud dont le booléen précise s'il pourra avoir des fils

Pour ajouter une instance de la classe DefaultMutableTreeNode dans le modèle de l'arbre, il est possible d'utiliser la méthode insert() de l'interface MutuableTreeNode ou utiliser la méthode add() de la classe DefaultMutableTreeNode. Celle-ci attend en paramètre une instance du noeud fils à ajouter. Elle ajoute le noeud après le dernier noeud fils, ce qui évite d'avoir à garder une référence sur la position où insérer le noeud.

Exemple (code Java 1.1) :

```
DefaultMutableTreeNode racineNode = new DefaultMutableTreeNode();
DefaultMutableTreeNode division1 = new DefaultMutableTreeNode("Division 1");
DefaultMutableTreeNode division2 = new DefaultMutableTreeNode("Division 2");
racineNode.add(division1);
racineNode.add(division2);
jTree.setModel(new DefaultTreeModel(racineNode));
```

Il est aussi possible de définir sa propre classe qui implémente l'interface MutableTreeNode : une possibilité est de définir une classe fille de la classe DefaultMutableTreeNode.

45.8.3. La modification du contenu de l'arbre

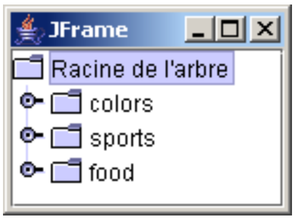
Les modifications du contenu de l'arbre peuvent se faire au niveau du modèle (DefaultTreeModel) ou au niveau du noeud.

La méthode getModel() de la classe JTree permet d'obtenir une référence sur l'instance de la classe TreeModel qui encapsule le modèle de données.

Il est ainsi possible d'accéder à tous les noeuds du modèle pour les modifier.

Exemple (code Java 1.1) :

```
jTree = new JTree();
Object noeudRacine = jTree.getModel().getRoot();
((DefaultMutableTreeNode)noeudRacine).setUserObject("Racine de l'arbre");
```



L'interface TreeModel ne propose rien pour permettre la mise à jour du modèle. Pour cela, il faut utiliser une instance de la classe DefaultTreeModel.

Elle propose plusieurs méthodes pour ajouter ou supprimer un noeud :

void insertNodeInto(MutableTreeNode child, MutableTreeNode parent, int index)
void removeNodeFromParent(MutableTreeNode parent)

L'avantage de ces deux méthodes est qu'elles mettent à jour le modèle mais aussi qu'elles mettent à jour la vue en appelant respectivement les méthodes nodesWereInserted() et nodesWereRemoved() de la classe DefaultTreeModel.

Ces deux méthodes sont donc pratiques pour faire des mises à jour mineures mais elles sont peut adaptées pour de nombreuses mises à jour puisqu'elles déclenchent un événement à chacune de leurs utilisations.

45.8.3.1. Les modifications des noeuds fils

La classe DefaultMutableTreeNode propose plusieurs méthodes pour mettre à jour le modèle à partir du noeud qu'elle encapsule.

void add(MutableTreeNode child)
void insert(MutableTreeNode child, int index)
void remove(int index)
void remove(MutableTreeNode child)
void removeAllChildren()
void removeFromParent()

Toutes ces méthodes sauf la dernière agissent sur un ou plusieurs noeuds fils. Ces méthodes agissent simplement sur la structure du modèle. Elles ne provoquent pas un affichage par la partie vue de ces changements. Pour cela il est nécessaire d'utiliser une des méthodes suivantes proposées par la classe DefaultTreeModel :

Méthode	Rôle
void reload()	rafraichir toute l'arborescence à partir du modèle

void reload(TreeNode node)	rafraichir toute l'arborescence à partir du noeud précisé en paramètre
void nodesWereInserted(TreeNode node, int[] childIndices)	pour le noeud précisé, cette méthode rafraichit les noeuds fils ajoutés dont les index sont fournis en paramètre
void nodesWereRemoved(TreeNode node,int[] childIndices, Object[] removedChildren)	pour le noeud précisé, cette méthode rafraichit les noeuds fils supprimés dont les index sont fournis en paramètre
void nodeStructureChanged(TreeNode node)	cette méthode est identique à la méthode reload()

45.8.3.2. Les événements émis par le modèle

Il est possible d'enregistrer un listener de type TreeModelListener sur un objet de type DefaultTreeModel.

L'interface TreeModelListener définit quatre méthodes pour répondre à des événements particuliers :

Méthode	Rôle
void treeNodesChanged(TreeModelEvent)	la méthode nodeChanged() ou nodesChanged() est utilisée
void treeStructureChanged(TreeModelEvent)	la méthode reload() ou nodeStructureChanged() est utilisée
void treeNodesInserted(TreeModelEvent)	la méthode nodeWhereInserted() est utilisée
void treeNodesRemoved(TreeModelEvent)	la méthode nodeWhereRemoved() est utilisée

Toutes ces méthodes ont un objet de type TreeModelEvent qui encapsule l'événement.

La classe TreeModelEvent propose cinq méthodes pour obtenir des informations sur les noeuds impactés par l'événement.

Méthode	Rôle
Object getSource()	renvoie une instance sur le modèle de l'arbre (généralement un objet de type DefaultTreeModel)
TreePath getTreePath()	renvoie le chemin du noeud affecté par l'événement
Object[] getPath()	Renvoie la succession de noeuds de la racine au noeud parent des noeuds impactés
Object[] getChildren()	
int[] getChildIndices()	retourne les index des noeuds modifiés

Dans la méthode treeStructureChanged(), seules les méthodes getPath() et getTreePath() fournissent des informations utiles en retournant le noeud qui a été modifié.

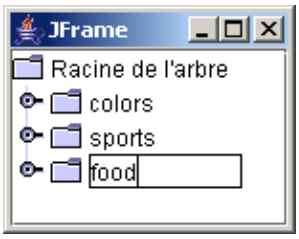
Dans la méthode treeNodesChanged(), treeNodesRemoved() et treeNodesInserted() les méthodes getPath() et getTreePath() renvoient le noeud père des noeuds affectés. Les méthodes getChildIndices() et getChidren() renvoient respectivement un tableau des index des noeuds fils modifiés et un tableau de ces noeuds fils.

Dans ces méthodes, les méthodes getPath() et getTreePath() renvoient le noeud père des noeuds affectés.

Comme l'objet JTree enregistre ses propres listeners, il n'est pas nécessaire la plupart du temps, d'enregistrer ces listeners hormis pour des besoins spécifiques.

45.8.3.3. L'édition d'un noeud

Par défaut, le composant JTree est readonly. Il est possible d'autoriser l'utilisateur à modifier le libellé des noeuds en utilisant la méthode setEditable() avec le paramètre true : jTtree.setEditable(true);



Pour éditer un noeud, il faut

- sur un noeud non sélectionné : cliquer rapidement trois fois sur le noeud à modifier
- sur un noeud déjà sélectionné : cliquer une fois sur le noeud ou appuyer sur la touche F2

Pour valider les modifications, il suffit d'appuyer sur la touche « Entree ».

Pour annuler les modifications, il suffit d'appuyer sur la touche « Esc »

Il est possible d'enregistrer un listener de type TreeModelListerner pour assurer des traitements lors d'événements liés à l'édition d'un noeud.

L'interface TreeModelListener définit la méthode treeNodesChanged() qui permet de traiter les événements de type TreeModelEvent liés à la modification d'un noeud.

Exemple (code Java 1.1) :

```
jTree.setEditable(true);
jTree.getModel().addTreeModelListener(new TreeModelListener() {

  public void treeNodesChanged(TreeModelEvent evt) {
    System.out.println("TreeNodesChanged");
    Object[] noeuds = evt.getChildren();
    int[] indices = evt.getChildIndices();
    for (int i = 0; i < noeuds.length; i++) {
      System.out.println("Index " + indices[i] + ", nouvelle valeur : "
          + noeuds[i]);
    }
  }

  public void treeStructureChanged(TreeModelEvent evt) {
    System.out.println("TreeStructureChanged");
  }

  public void treeNodesInserted(TreeModelEvent evt) {
    System.out.println("TreeNodesInserted");
  }

  public void treeNodesRemoved(TreeModelEvent evt) {
    System.out.println("TreeNodesRemoved");
  }

});
```

45.8.3.4. Les éditeurs personnalisés

Il est possible de définir un éditeur particulier pour éditer la valeur d'un noeud. Un éditeur particulier doit implémenter l'interface TreeCellEditor.

Cette interface hérite de l'interface CellEditor qui définit plusieurs méthodes utiles pour la création d'un éditeur dédié :

Object getCellEditorValue();
boolean isCellEditable(EventObject);
boolean shouldSelectCell(EventObject);
boolean stopCellEditing();
void cancelCellEditing();
void addCellEditorListener(CellEditorListener);
void removeCellEditorListener(CellEditorListener);

L'interface TreeCellEditor ne définit qu'une seule méthode :

Component getTreeCellEditorComponent(JTree tree, Object value, boolean isSelected, boolean expanded, boolean leaf, int row);

Cette méthode renvoie un composant qui va permettre l'édition de la valeur du noeud.

La valeur initiale est fournie dans le second paramètre de type Object. Les trois arguments de type booléen suivants permettent respectivement de savoir si le noeud est sélectionné, est étendu et est une feuille.

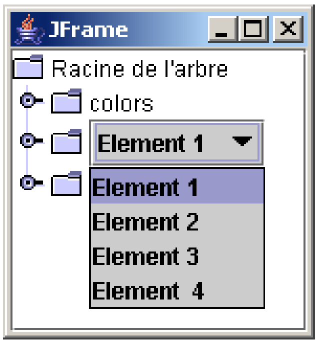
Swing propose une implémentation de cette interface dans la classe DefaultCellEditor qui permet de modifier la valeur du noeud sous la forme d'une zone de texte, d'une case à cocher ou d'une liste déroulante grâce à trois constructeurs :

public DefaultCellEditor(JTextField text); public DefaultCellEditor(JCheckBox box); public DefaultCellEditor(JComboBox combo);

La méthode setCellEditor() de la classe JTree permet d'associer le nouvel éditeur à l'arbre.

Exemple (code Java 1.1) :

```
      jTree.setEditable(true);
      String[] elements = { "Element 1", "Element 2", "Element 3", "Element  4"};
      JComboBox jCombo = new JComboBox(elements);
      DefaultTreeCellEditor editor = new DefaultTreeCellEditor(jTree,
               new DefaultTreeCellRenderer(), new DefaultCellEditor(jCombo));
      jTree.setCellEditor(editor);
```



45.8.3.5. La définition des noeuds éditables

Par défaut, si la méthode setEditable(true) est utilisée alors tous les noeuds sont modifiables.

Il est possible de définir les noeuds de l'arbre qui sont éditables en créant une classe fille de la classe JTree et en redéfinissant la méthode isPathEditable().

Cette méthode est appelée avant chaque édition d'un noeud. Elle attend en paramètre un objet de type TreePath qui encapsule le chemin du noeud à éditer.

Par défaut, elle renvoie le résultat de l'appel à la méthode isEditable(). Il est d'ailleurs important, lors de la redéfinition de la méthode isPathEditable(), de tenir compte du résultat de la méthode isEditable() pour s'assurer que l'arbre est modifiable avant de vérifier si le noeud peut être modifié.

45.8.4. La mise en oeuvre d'actions sur l'arbre

45.8.4.1. Etendre ou refermer un noeud

Pour étendre un noeud et ainsi voir ses fils, l'utilisateur peut double cliquer sur l'icône ou sur le libellé du noeud. Il peut aussi cliquer sur le petit commutateur à gauche de l'icône.

Enfin, il est possible d'utiliser le clavier pour naviguer dans l'arbre à l'aide des touches flèches haut et bas et des touches flèches droite et gauche pour respectivement étendre ou refermer un noeud. Lors d'un appui sur la flèche gauche, si le noeud est déjà fermé alors c'est le noeud père qui est sélectionné. De la même façon, lors d'un appui sur la flèche droite, si le noeud est étendu alors le premier noeud fils est sélectionné.

La touche HOME permet de sélectionner le noeud racine. La touche END permet de sélectionner le noeud qui est la dernière feuille du dernier noeud. Les touches PAGEUP et PAGEDOWN permettent de parcourir rapidement les noeuds de l'arbre.

Depuis Java 2 version 1.3, la méthode setToggleClickCount() permet de préciser le nombre de clics nécessaires pour étendre ou refermer un noeud.

La classe JTree propose plusieurs méthodes liées aux actions permettant d'étendre ou de refermer un noeud.

Méthode	Rôle
public void expandRow (int row)	Etendre le noeud dont l'index est fourni en paramètre
public void collapseRow(int row)	Refermer le noeud dont l'index est fourni en paramètre
public void expandPath(TreePath path)	Etendre le noeud encapsulé dans la classe TreePath fournie en paramètre
public void collapsePath(TreePath path)	Refermer le noeud encapsulé dans la classe TreePath fournie en paramètre
public boolean isExpanded(int row)	Renvoie un booléen qui précise si le noeud dont l'index est fourni en paramètre est étendu
public boolean isCollapsed (int row)	Renvoie un booléen qui précise si le noeud dont l'index est fourni en paramètre est refermé
public boolean isExpanded(TreePath path)	Renvoie un booléen qui précise si le noeud encapsulé dans la classe TreePath fournie en paramètre est étendu
public boolean isCollapsed (TreePath path)	Renvoie un booléen qui précise si le noeud encapsulé dans la classe TreePath fournie en paramètre est refermé

Par défaut, le noeud racine est étendu.

Les méthodes expandRow() et expandPath() ne permettent que d'étendre les noeuds fils directs du noeud sur lesquel elles sont appliquées. Pour étendre les noeuds sous-jacents il est nécessaire d'écrire du code pour réaliser l'opération sur chaque noeud concerné de façon récursive.

Pour refermer tous les noeudsx et ne laisser que le noeud racine, il faut utiliser la méthode collapseRow() en lui passant 0 comme paramètre puisque le noeud racine est toujours le premier noeud.

Exemple (code Java 1.1) :

```
jTree.collapseRow(0);
```

La classe JTree propose deux méthodes pour forcer un noeud à être visible : scrollPathToVisible() et scrollRowToVisible(). Celles-ci ne peuvent fonctionner que si le composant JTree est inclus dans un conteneur JScrollPane pour permettre au composant de scroller.

Exemple (code Java 1.1) :

```
      jTree.addTreeExpansionListener(new TreeExpansionListener() {
        public void treeExpanded(TreeExpansionEvent evt) {
          System.out.println("treeExpanded : path=" + evt.getPath());
          jTree.scrollPathToVisible(evt.getPath());
        }
```

45.8.4.2. La détermination du noeud sélectionné

Pour déterminer le noeud sélectionné, il suffit d'utiliser la méthode getLastSelectedPathComponent() de la classe JTree et de caster la valeur retournée dans le type du noeud, généralement de type DefaultMutableTreeNode. La méthode getObject() du noeud permet d'obtenir l'objet associé au noeud. Si l'objet associé est simplement une chaîne de caractères ou si la valeur nécessaire est simplement le libellé du noeud, il suffit d'utiliser la méthode toString().

Exemple (code Java 1.1) : un bouton qui précise lors d'un clic le noeud sélectionné

```
  ...
  private JButton getJButton() {
    if (jButton == null) {
      jButton = new JButton();
      jButton.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent e) {
          System.out.println("actionPerformed()");
          System.out.println("Noeud sélectionné : "
              + jTree.getLastSelectedPathComponent().toString());
        }
      });
    }
```

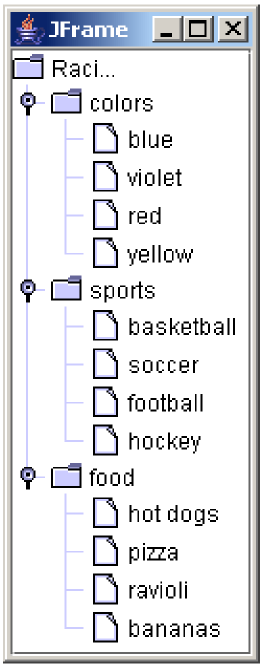
45.8.4.3. Le parcours des noeuds de l'arbre

Il peut être nécessaire de parcourir tout ou partie des noeuds de l'arbre pour par exemple faire une recherche dans l'arborescence.

Si l'arbre est composé de noeuds de type DefaultMutableTreenode alors l'interface TreeNode propose plusieurs méthodes pour obtenir une énumération des noeuds. L'ensemble, ou seulement une partie des données, peut être parcouru dans les deux sens et selon deux types de présentation des valeurs.

Enumeration preorderEnumeration();
Enumeration postorderEnumeration();
Enumeration breadthFirstEnumeration();
Enumeration depthFirstEnumeration();

Dans l'exemple ci-dessous, l'arborescence suivante est utilisée :



Exemple (code Java 1.1) : un bouton qui précise lors d'un clic le noeud sélectionné

```
Enumeration e = ((DefaultMutableTreeNode)jTree.getModel().getRoot()).preorderEnumeration();
while (e.hasMoreElements()) {
   System.out.println(e.nextElement() + " ");
}
```

Résultat :

preorder	postorder	breadthFirst	depthFirst
Racine de l'arbre colors blue violet red yellow sports basketball soccer football hockey food hot dogs pizza ravioli bananas	blue violet red yellow colors basketball soccer football hockey sports hot dogs pizza ravioli bananas food Racine de l'arbre	Racine de l'arbre colors sports food blue violet red yellow basketball soccer football hockey hot dogs pizza ravioli bananas	blue violet red yellow colors basketball soccer football hockey sports hot dogs pizza ravioli bananas food Racine de l'arbre

La méthode pathFromAncestorEnumeration(TreeNode ancestor) renvoie une énumération des noeuds entre le noeud sur lequel la méthode est appelée et le noeud fourni en paramètre. Ainsi le noeud fourni en paramètre doit obligatoirement être un noeud fils direct ou indirect du noeud sur lequel la méthode est appelée. Dans le cas contraire, une exception de type IllegalArgumentException est levée.

45.8.5. La gestion des événements

Il est possible d'attacher des listeners pour répondre aux événements liés à la sélection d'un élément ou l'extension ou la refermeture d'un noeud.

45.8.5.1. La classe TreePath

Durant son utilisation, le composant JTree ne gère pas directement les noeuds du modèle de données. La manipulation de ces noeuds se fait via un index ou une instance de la classe TreePath.

L'utilisation de l'index est assez délicate car seul le noeud racine de l'arbre possède toujours le même index 0. Pour les autres noeuds, la valeur de l'index dépend de l'état étendu/refermé de chaque noeud puisque seuls les noeuds affichés possèdent un index. Il est donc préférable d'utiliser la classe TreePath.

Le modèle de données utilise des noeuds mais l'interface de l'arbre utilise une autre représentation sous la forme de la classe TreePath.

La classe DefaultMutableTreeNode est la représentation physique d'un noeud, la classe TreePath est la représentation logique. Elle encapsule le chemin du noeud dans l'arborescence.

Cette classe contient plusieurs méthodes :

```
public Object getLastPathComponent();
public Object getPathComponent(int index);
public int getPathCount();
public Object[] getPath();
public TreePath getParentPath();
public TreePath pathByAddingChild(Object child);
public boolean isDescendant(TreePath treePath)
```

La méthode getPath() renvoie un tableau d'objets contenant chaque noeud qui compose le chemin encapsulé par la classe TreePath.

La méthode getLastPathComponent() renvoie le dernier noeud du chemin.

La méthode getPathCount() renvoie le nombre de noeuds qui composent le chemin.

La méthode getPathComponent() permet de renvoyer le noeud dont l'index dans le chemin est fourni en paramètre. L'élément avec l'index 0 est toujours le noeud racine de l'arbre.

La méthode getParentPath() renvoie une instance de la classe TreePath qui encapsule le chemin vers le noeud père du chemin encapsulé.

La méthode pathByAddingChild() renvoie une instance de la classe TreePath qui encapsule le chemin issu de l'ajout d'un noeud fils fourni en paramètre.

La méthode idDescendant() renvoie un booléen qui précise si le chemin passé en paramètre est un descendant du chemin encapsulé.

La classe TreePath ne permet pas de gérer le contenu de chaque noeud mais uniquement son chemin dans l'arborescence. Pour accéder au noeud à partir de son chemin, il faut utiliser la méthode getLastPathComponent(). Pour obtenir un noeud inclus dans le chemin, il faut utiliser la getPathComponent() ou getPath(). Toutes ces méthodes renvoient un objet ou un tableau de type Object. Il est donc nécessaire de réaliser un cast vers le type de noeud utilisé, généralement de type DefaultMutableTreeNode.

A partir d'un noeud de type DefaultMutableTreeNode, il est possible d'obtenir l'objet TreePath encapsulant le chemin du noeud. La méthode getPath() permet d'obtenir un tableau d'objets de type TreeNode qu'il suffit de passer au constructeur de la classe TreePath.

Exemple (code Java 1.1) :

```
TreeNode[] chemin = noeud.getPath();
TreePath path = new TreePath(chemin);
```

45.8.5.2. La gestion de la sélection d'un noeud

La gestion de la sélection de noeud dans un composant JTree est déléguée à un modèle de sélection sous la forme d'une classe qui implémente l'interface TreeSelectionModel. Par défaut, le composant JTree utilise une instance de la classe DefaultTreeSelectionModel.

Le modèle de sélection peut être configuré selon trois modes :

- SINGLE_TREE_SELECTION: un seul noeud peut être sélectionné.
- CONTIGUOUS_TREE_SELECTION: plusieurs noeuds peuvent être sélectionnés à condition d'être contigus.
- DISCONTIGUOUS_TREE_SELECTION: plusieurs noeuds peuvent être sélectionnés de façon continue et/ou discontinue (c'est le mode par défaut).

Pour empêcher la sélection d'un noeud dans l'arbre, il faut supprimer son modèle de sélection en passant null à la méthode setSelectionModel().

Exemple (code Java 1.1) :

```
JTree jTree = new JTree()jTree.setSelectionModel(null);
```

La sélection d'un noeud peut être réalisée par l'utilisateur ou par l'application : le modèle de sélection s'assure que celle-ci est réalisée en respectant le mode de sélection du modèle.

L'utilisateur peut utiliser la souris pour sélectionner un noeud ou appuyer sur la touche Espace sur le noeud courant pour le sélectionner. Il est possible de sélectionner plusieurs noeuds en fonction du mode en maintenant la touche CTRL enfoncée. Avec la touche SHIFT, il est possible selon le mode de sélectionner tous les noeuds entre un premier noeud sélectionné et le noeud courant.

La sélection d'un noeud génère un événement de type TreeSelectionEvent.

Le dernier noeud sélectionné peut être obtenu en utilisant les méthodes getLeadSelectionPath() ou getLeadSelectionRow().

Par défaut la sélection d'un noeud entraine l'extension des noeuds ascendants correspondant afin de les rendre visibles. Pour empêcher ce comportement, il faut utiliser la méthode setExpandSelectedPath() en lui fournissant la valeur false en paramètre.

public void setExpandsSelectedPaths(boolean cond);

Les classes DefaultTreeSelectionModel et JTree possèdent plusieurs méthodes pour gérer la sélection de noeuds. Certaines de ces méthodes sont communes à ces deux classes.

Méthode	Rôle
int getSelectionMode()	renvoie le mode de sélection
void setSelectionMode(int mode)	mettre à jour le mode de sélection
Object getLastSelectedPathComponent()	renvoie le premier noeud de la sélection courante ou null si aucun noeud n'est sélectionné JTree uniquement
TreePath getAnchorSelectionPath()	JTree uniquement
void setAnchorSelectionPath(TreePathpath)	JTree uniquement
TreePath getLeadSelectionPath()	renvoie le dernier path ajouté à la sélection ou identifié comme tel
setLeadSelectionPath()	fait de newPath le dernier Path ajouté
int getMaxSelectionRow()	Renvoie le plus grand index de la sélection
int getMinSelectionRow()	Renvoie le plus petit index de la sélection
int getSelectionCount()	Renvoie le nombre de noeuds inclus dans la sélection

TreePath getSelectionPath()	Renvoie le chemin du premier élément sélectionné
TreePath[] getSelectionPaths()	Renvoie un tableau des chemins des noeuds inclus dans la sélection
int[] getSelectionRows()	Renvoie un tableau des index des noeuds inclus dans la sélection
Boolean isPathSelected (TreePath path)	Renvoie un booléen si le noeud dont le chemin est fourni en paramètre est inclus dans la sélection
Boolean isRowSelected(int row)	Renvoie un booléen si le noeud dont l'index est fourni en paramètre est inclus dans la sélection
boolean isSelectionEmpty()	Renvoie un booléen qui précise si la sélection est vide
void clearSelection()	Vide la sélection
void removeSelectionInterval (int row0, int row1)	Enlève de la sélection les noeuds dans l'intervalle des index fournis en paramètre
void removeSelectionPath(TreePath path)	Enlève de la sélection le noeud dont le chemin est fourni en paramètre
void removeSelectionRow (int row)	Enlève de la sélection le noeud dont l'index est fourni en paramètre JTree uniquement
void removeSelectionRows(int[] rows)	Enlève de la sélection les noeuds dont les index sont fournis en paramètre JTree uniquement
void addSelectionInterval(int row0, int row1)	Ajouter à la sélection les noeuds dont l'intervalle des index est fourni en paramètre
void addSelectionPath(TreePath path)	Ajouter à la sélection le noeud dont le chemin est fourni en paramètre
addSelectionPaths(TreePath[] path)	Ajouter à la sélection les noeuds dont les chemins sont fournis en paramètre
void addSelectionRow(int row)	Ajouter à la sélection le noeud dont l'index est fourni en paramètre
void addSelectionRows(int[] row)	Ajouter à la sélection les noeuds dont les index sont fournis en paramètre
void setSelectionInterval(int row0, int row1)	Définir la sélection avec les noeuds dont les index sont fournis en paramètre JTree uniquement
setSelectionPath(TreePath path)	Définir la sélection avec le noeud dont le chemin est fourni en paramètre
void setSelectionPaths (TreePath[] path)	Définir la sélection avec les noeuds dont les chemins sont fournis en paramètre
void setSelectionRow(int row)	Définir la sélection avec le noeud dont l'index est fourni en paramètre
void setSelectionRows(int[] row)	Définir la sélection avec les noeuds dont les index sont fournis en paramètre JTree uniquement

45.8.5.3. Les événements lies à la sélection de noeuds

Lors de la sélection d'un noeud, un événement de type TreeSelectionEvent est émis. Pour traiter cet événement, le composant doit enregistrer un listener de type TreeSelectionListener.

L'interface TreeSelectionListener définit une seule méthode :

public void valueChanged(TreeSelectionEvent evt)

Exemple (code Java 1.1) :

```
jTree.addTreeSelectionListener(new javax.swing.event.TreeSelectionListener() {
      public void valueChanged(javax.swing.event.TreeSelectionEvent e) {

        DefaultMutableTreeNode noeud = (DefaultMutableTreeNode) jTree
            .getLastSelectedPathComponent();
        if (noeud == null)
          return;
        System.out.println("valueChanged() : " + noeud);
      }
});
```

La classe TreeSelectionEvent possède plusieurs méthodes pour obtenir des informations sur la sélection.

Méthode	Rôle
public TreePath[] getPaths()	Renvoie un tableau des chemins des noeuds sélectionnés
public boolean isAddedPath (TreePath path)	Renvoie true si le noeud sélectionné est ajouté à la sélection. Renvoie false si le noeud sélectionné est retiré de la sélection
TreePath getPath()	Renvoie le chemin du premier noeud sélectionné
boolean isAddedPath()	Renvoie true si le premier noeud sélectionné est ajouté à la sélection. Renvoie false si le premier noeud sélectionné est retiré de la sélection
TreePath getOldLeadSelection()	Renvoie l'ancien lead path
TreePath getNewLeadSelection()	Renvoie le leader actuel de la sélection

Un listener de type TreeSelectionListener est enregistré en utilisant la méthode addTreeSelectionListener() de la classe JTree.

Exemple (code Java 1.1) :

```
jTree.addTreeSelectionListener(new TreeSelectionListener() {

  public void valueChanged(TreeSelectionEvent e) {

    Object obj = jTree.getLastSelectedPathComponent();
    System.out.println("getLastSelectedPathComponent=" + obj);
    System.out.println("getPath=" + e.getPath());
    System.out.println("getNewLeadSelectionPath="
        + e.getNewLeadSelectionPath());
    System.out.println("getOldLeadSelectionPath="
        + e.getOldLeadSelectionPath());
    TreePath[] paths = e.getPaths();

    for (int i = 0; i < paths.length; i++) {
      System.out.println("Path " + i + "=" + paths[i]);
    }
  }
});
```

Un événement de type TreeSelectionEvent n'est émis que si un changement intervient dans la sélection : lors d'un clic sur un noeud, celui-ci est sélectionné et un événement est émis. Lors d'un nouveau clic sur ce même noeud, le noeud est toujours sélectionné mais l'événement n'est pas émis puisque la sélection n'est pas modifiée.

Dans un listener pour gérer les événements de la souris, il est possible d'utiliser la méthode getPathForLocation() pour déterminer le chemin d'un noeud à partir des coordonnées de la souris qu'il faut lui fournir en paramètre.

La méthode getPathForLocation() renvoie null si l'utilisateur clique en dehors d'un noeud dans l'arbre.

Exemple (code Java 1.1) :

```
jTree.addMouseListener(new MouseAdapter() {
  public void mouseClicked(MouseEvent evt) {
      TreePath path =
         jTree.getPathForLocation(evt.getX(), evt.getY());
      if (path != null) {
         System.out.println("path= " + path.getLastPathComponent());
       }
  }
});
```

Plusieurs autres méthodes peuvent aussi être utilisées dans ce contexte.

Méthode	Rôle
TreePath getClosestPathForLocation(int x, int y)	Retourne le chemin du noeud le plus proche des coordonnées fournies en paramètre
int getClosestRowForLocation(int x, int y)	Retourne l'index du noeud le plus proche des coordonnées fournies en paramètre
Rectangle getPathBounds(TreePath path)	Renvoie un objet de type Rectangle qui représente la surface du noeud dont le chemin est fourni en paramètre
TreePath getPathForLocation(int x, int y)	Retourne le chemin du noeud dont la surface contient les coordonnées fournies en paramètre. Renvoie null si ces coordonnées ne correspondent à aucun noeud
TreePath getPathForRow(int row)	Renvoie le chemin du noeud dont l'index est fourni en paramètre
Rectangle getRowBounds(int row)	Renvoie un objet de type Rectangle qui représente la surface du noeud dont l'index est fourni en paramètre
int getRowForLocation(int x, int y)	Renvoie l'index du noeud à la position fournie

45.8.5.4. Les événements lorsqu'un noeud est étendu ou refermé

A chaque fois qu'un noeud est étendu ou refermé, un événement de type TreeExpansionEvent est émis. Il est possible de répondre à ces événements en mettant en place un listener de type TreeExpansionListener.

L'interface TreeExpansionListener propose deux méthodes :

public void treeExpanded(TreeExpansionEvent event) public void treeCollapsed(TreeExpansionEvent event)

La classe TreeExpansionEvent possède une propriété source qui contient une référence sur le composant JTree à l'origine de l'événement et une propriété path qui contient un objet de type TreePath encapsulant le chemin du noeud à l'origine de l'événement.

Les valeurs de ces deux propriétés peuvent être obtenues avec leurs getters respectifs : getSource() et getPath().

Exemple (code Java 1.1) :

```
jTree.addTreeExpansionListener(new TreeExpansionListener() {

  public void treeExpanded(TreeExpansionEvent evt) {
    System.out.println("expand, path=" +
                          evt.getPath());
  }

  public void treeCollapsed(TreeExpansionEvent evt) {
    System.out.println("collapse, path=" +
                        evt.getPath());
  }

});
```

Un seul événement est généré à chaque fois qu'un noeud est étendu ou refermé : il n'y a pas d'événements émis pour les éventuels noeuds fils qui sont étendus ou refermés suite à l'action.

45.8.5.5. Le contrôle des actions pour étendre ou refermer un noeud

Il peut être utile de recevoir un événement avant qu'un noeud ne soit étendu ou refermé. Un listener de type TreeWillExpandListener() peut être mis en place pour recevoir un événement de type TreeExpansionEvent lors d'une tentative pour étendre ou refermer un noeud.

L'interface TreeWillExpandListener définit deux méthodes :

public void treeWillCollapse(TreeExpansionEvent evt) throws ExpandVetoException;
public void treeWillExpand(TreeExpansionEvent evt) throws ExpandVetoException;

Les deux méthodes peuvent lever une exception de type ExpandVetoException. Cette exception est levée si, pendant l'exécution d'une de ces méthodes, des conditions sont remplies pour empêcher l'action demandée par l'utilisateur. Si l'exception n'est pas levée à la fin des traitements de la méthode alors l'action est réalisée.

Exemple (code Java 1.1) : empécher tous les noeuds étendus de se refermer

```
jTree.addTreeWillExpandListener(new TreeWillExpandListener() {

  public void treeWillCollapse(TreeExpansionEvent event) throws ExpandVetoException {
    throw new ExpandVetoException(event);
  }

  public void treeWillExpand(TreeExpansionEvent event) throws ExpandVetoException {
  }

});
```

45.8.6. La personnalisation du rendu

Le rendu du composant JTree dépend bien sûr dans un premier temps du look and feel utilisé mais il est aussi possible de personnaliser plus finement le rendu des noeuds du composant.

Il est possible de préciser la façon dont les lignes reliant les noeuds sont rendues via une propriété client nommée lineStyle. Cette propriété peut prendre trois valeurs :

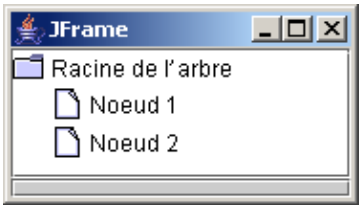
Valeur	Rôle
Angled	Une ligne à angle droit relie chaque noeud fils à son noeud père
None	Aucune ligne n'est affichée entre les noeuds
Horizontal	Une simple ligne horizontale sépare les noeuds enfants du noeud racine

Pour préciser la valeur de la propriété que le composant doit utiliser, il faut utiliser la méthode putClientProperty() qui attend deux paramètres sous forme de chaînes de caractères :

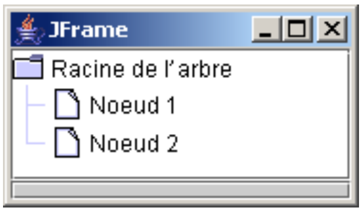
- le nom de la propriété
- sa valeur

Exemple (code Java 1.1) :

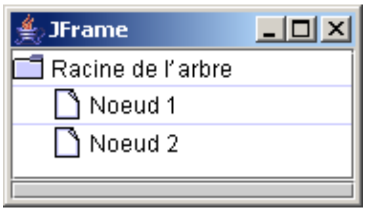
```
jTree = new JTree(racine);
jTree.putClientProperty("JTree.lineStyle","Horizontal");
```



None



Angled



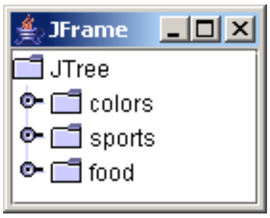
Horizontal

Il est possible de modifier l'apparence de la racine de l'arbre grâce à deux méthodes de la classe JTree : setRootVisible() et setShowsRootHandles().

La méthode setRootVisible() permet de préciser avec son booléen en paramètre si la racine est affichée ou non.

Exemple (code Java 1.1) :

```
JTree jtree = new JTree();
jtree.setShowsRootHandles(false);
jtree.setRootVisible(true);
```



45.8.6.1. Personnaliser le rendu des noeuds

Il est possible d'obtenir un contrôle total sur le rendu de chaque noeud en définissant un objet qui implémente l'interface TreeCellRenderer. Attention, le rendu personnalisé est parfois dépendant du look & feel utilisé.

L'interface TreeCellRenderer ne définit qu'une seule méthode :

Component getTreeCellRendererComponent(JTree tree, Object value, boolean selected, boolean expanded, boolean leaf, int row, boolean hasFocus)

Cette méthode envoie un composant qui va encapsuler le rendu du noeud. Le premier argument de type JTree encapsule le composant JTree lui-même. L'argument de type Object encapsule le noeud dont le rendu doit être généré.

La méthode getCellRenderer() renvoie un objet qui encapsule le TreeCellRenderer. Il est nécessaire de réaliser un cast vers le type de cet objet.

Swing propose une classe de base DefaultTreeCellRenderer pour le rendu. Elle propose plusieurs méthodes pour permettre de définir le rendu.

Méthode	Rôle
void setBackgroundNonSelectionColor(Color)	Permet de définir la couleur de fond du noeud lorsqu'il n'est pas sélectionné
void setBackgroundSelectionColor(Color)	Permet de définir la couleur de fond du noeud lorsqu'il est sélectionné
void setBorderSelectionColor(Color)	Permet de définir la couleur de la bordure du noeud lorsqu'il est sélectionné. Il n'est pas possible de définir une bordure pour un noeud sélectionné
void setTextNonSelectionColor(Color)	

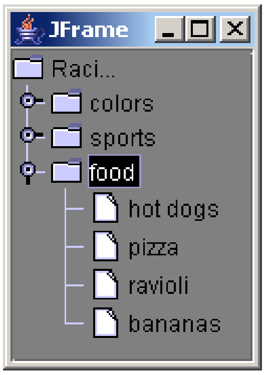
	Permet de définir la couleur du texte du noeud lorsqu'il n'est pas sélectionné
void setTextSelectionColor(Color)	Permet de définir la couleur du texte du noeud lorsqu'il est sélectionné
void setFont(Font)	Permet de définir la police de caractère utilisé pour afficher le texte du noeud
void setClosedIcon(Icon)	Permet de définir l'icône associée au noeud lorsque celui-ci est fermé
void setOpenIcon(Icon)	Permet de définir l'icône associée au noeud lorsque celui-ci est étendu
void setLeafIcon(Icon)	Permet de définir l'icône associée au noeud lorsque celui-ci est une feuille

Un composant ne peut avoir qu'une seule instance de type TreeCellRenderer. Cette instance sera donc appelée pour définir le rendu de chaque noeud.

Exemple (code Java 1.1) :

```
TreeCellRenderer cellRenderer = jTree.getCellRenderer();
if (cellRenderer instanceof DefaultTreeCellRenderer) {
  DefaultTreeCellRenderer renderer = (DefaultTreeCellRenderer)cellRenderer;
  renderer.setBackgroundNonSelectionColor(Color.gray);
  renderer.setBackgroundSelectionColor(Color.black);
  renderer.setTextSelectionColor(Color.white);
  renderer.setTextNonSelectionColor(Color.black);
  jTree.setBackground(Color.gray);
}
```

Résultat :



Pour modifier les icônes utilisées par les différents éléments de l'arbre, il faut utiliser les méthodes setOpenIcon(), setClosedIcon() et setLeafIcon().

Méthode	Rôle
setOpenIcon()	précise l'icône pour un noeud ouvert
setClosedIcon()	précise l'icône pour un noeud fermé
setLeafIcon()	précise l'icône pour une feuille

Pour simplement supprimer l'affichage de l'icône, il suffit de passer null à la méthode concernée.

Exemple (code Java 1.1) :

```
DefaultTreeCellRenderer monRenderer = new DefaultTreeCellRenderer();
monRenderer.setOpenIcon(null);
monRenderer.setClosedIcon(null);
monRenderer.setLeafIcon(null);
```

Pour préciser une image, il faut créer une instance de la classe ImageIcon encapsulant l'image et la passer en paramètre de la méthode concernée.

Exemple (code Java 1.1) :

```
    private Icon ourvertIcon = new ImageIcon("images/ouvert.gif");
    private Icon fermeIcon   = new ImageIcon("images/ferme.gif");
    private Icon feuilleIcon = new ImageIcon("images/feuille.gif");
...
    DefaultTreeCellRenderer treeCellRenderer = new DefaultTreeCellRenderer();
    treeCellRenderer.setOpenIcon(ouvertIcon);
    treeCellRenderer.setClosedIcon(fermeIcon);
    treeCellRenderer.setLeafIcon(feuilleIcon);
```

Il est aussi possible de définir une classe qui hérite de la classe DefaultTreeCellRenderer. Cette classe propose une implémentation par défaut de l'interface TreeCellRenderer. Comme elle hérite de la classe JLabel, elle possède déjà de nombreuses méthodes pour assurer le rendu du noeud sous la forme d'un composant de type étiquette.

Exemple (code Java 1.1) :

```
import java.awt.Color;
import java.awt.Component;

import javax.swing.JTree;
import javax.swing.tree.DefaultTreeCellRenderer;

public class MonTreeCellRenderer extends DefaultTreeCellRenderer {

  public Component getTreeCellRendererComponent(JTree tree, Object value,
      boolean selected, boolean expanded, boolean leaf, int row,
      boolean hasFocus) {


    super.getTreeCellRendererComponent(tree,value, selected, expanded,
        leaf, row,hasFocus);

    setBackgroundNonSelectionColor(Color.gray);
    setBackgroundSelectionColor(Color.black);
    setTextSelectionColor(Color.white);
    setTextNonSelectionColor(Color.black);

    return this;
  }
}
```

Une fois la classe de type DefaultTreeCellRenderer instanciée, il faut utiliser la méthode setCellRenderer() de la classe JTree pour indiquer à l'arbre d'utiliser cette classe pour le rendu.

Exemple (code Java 1.1) :

```
    jTree.setCellRenderer(new MonTreeCellRenderer());
```

La création d'une classe fille de la classe DefaultTreeCellRenderer ne fonctionne correctement qu'avec les look and feel Metal et Windows car le look and feel Motif définit son propre Renderer.

45.8.6.2. Les bulles d'aides (Tooltips)

Le composant JTree ne propose pas de support pour les bulles d'aide en standard. Pour permettre à un composant JTree d'afficher une bulle d'aide, il faut :

- enregistrer le composant JTree auprès du ToolTipManager
- définir le contenu de la bulle d'aide dans le Renderer

L'enregistrement du composant auprès du ToolTipManager se fait en utilisant la méthode registerComponent() sur l'instance partagée.

Exemple (code Java 1.1) :

```
ToolTipManager.sharedInstance().registerComponent(jTree);
((JLabel)t.getCellRenderer()).setToolTipText("Arborescence des données");
```

L'inconvénient de cette méthode est que la bulle d'aide est toujours la même quelque soit la position de la souris sur tous les noeuds du composant. Pour assigner une bulle d'aide particulière à chaque noeud, il est nécessaire d'utiliser la méthode setToolTipText() dans la méthode getTreeCellRendererComponent() d'une instance fille de la classe DefaultTreeCellRenderer

Exemple (code Java 1.1) :

```
      jTree.setCellRenderer(new DefaultTreeCellRenderer() {

        public Component getTreeCellRendererComponent(JTree tree, Object value,
            boolean selected, boolean expanded, boolean leaf, int row,
            boolean hasFocus) {

          super.getTreeCellRendererComponent(tree, value, selected, expanded, leaf, row,
              hasFocus);
          setToolTipText(value.toString());

          return this;
        }
      });

      ToolTipManager.sharedInstance().registerComponent(jTree);
```

45.9. Les menus

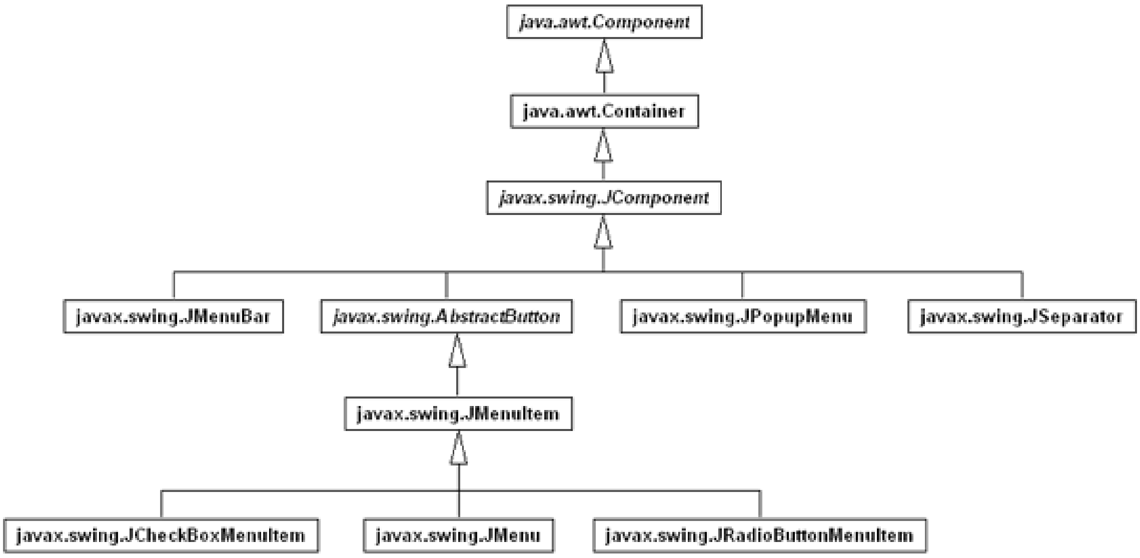
Les menus de Swing proposent certaines caractéristiques intéressantes en plus de celles proposées par un menu standard :

- les éléments de menu peuvent contenir une icône
- les éléments de menu peuvent être de type bouton radio ou case à cocher
- les éléments de menu peuvent avoir des raccourcis clavier (accelerators)

Les menus sont mis en oeuvre dans Swing avec un ensemble de classe :

- JMenuBar : encapsule une barre de menus
- JMenu : encapsule un menu
- JMenuItem : encapsule un élément d'un menu
- JCheckBoxMenuItem : encapsule un élément d'un menu sous la forme d'une case à cocher
- JRadioButtonMenuItem : encapsule un élément d'un menu sous la forme d'un bouton radio
- JSeparator : encapsule un élément d'un menu sous la forme d'un séparateur
- JPopupMenu : encapsule un menu contextuel

Toutes ces classes héritent de façon directe ou indirecte de la classe JComponent.



Les éléments de menus cliquables héritent de la classe JAbstractButton.

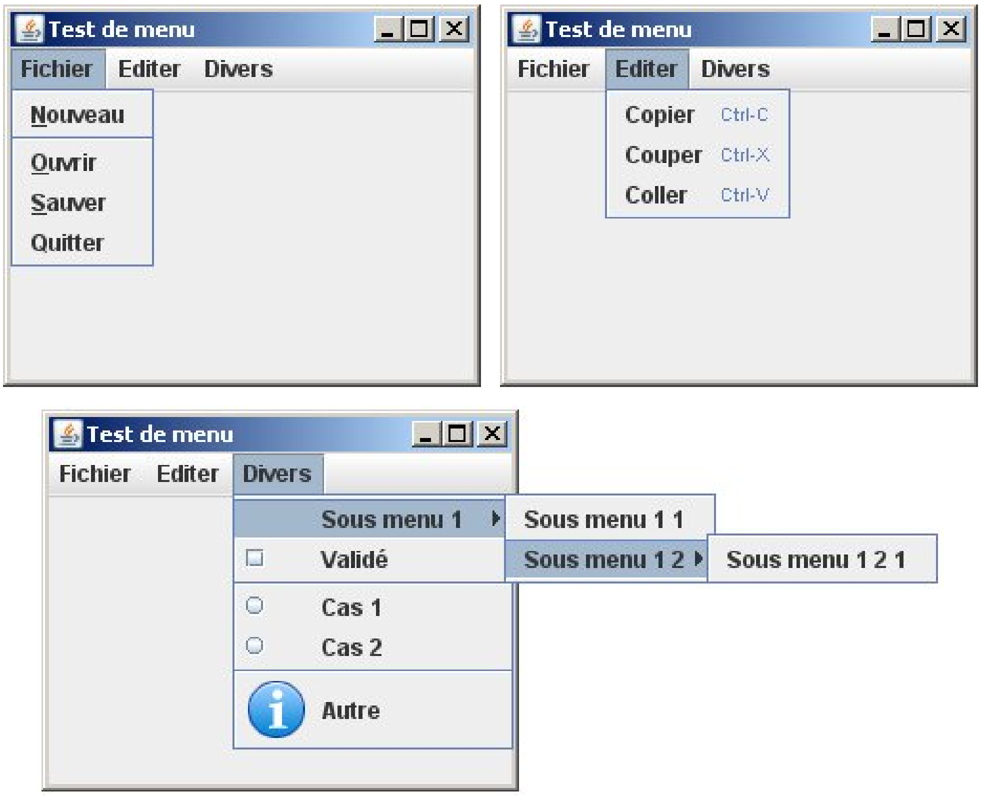
JMenu hérite de la classe JMenuItem et non pas l'inverse car chaque JMenu contient un JMenuItem implicite qui encapsule le titre du menu.

La plupart des classes utilisées pour les menus implémentent l'interface MenuElement. Cette interface définit des méthodes pour la gestion des actions standards de l'utilisateur. Ces actions sont gérées par la classe MenuSelectionManager.

Exemple :

```
package fr.jmdoudoux.dej.swing.menu;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class TestMenuSwing1 extends JMenuBar {

  public TestMenuSwing1() {

    // Listener générique qui affiche l'action du menu utilisé
    ActionListener afficherMenuListener = new ActionListener() {
      public void actionPerformed(ActionEvent event) {
        System.out.println("Elément de menu [" + event.getActionCommand()
            + "] utilisé.");
      }
    };

    // Création du menu Fichier
    JMenu fichierMenu = new JMenu("Fichier");
    JMenuItem item = new JMenuItem("Nouveau", 'N');
    item.addActionListener(afficherMenuListener);
    fichierMenu.add(item);
    item = new JMenuItem("Ouvrir", 'O');
    item.addActionListener(afficherMenuListener);
    fichierMenu.add(item);
    item = new JMenuItem("Sauver", 'S');
    item.addActionListener(afficherMenuListener);
    fichierMenu.insertSeparator(1);
    fichierMenu.add(item);
    item = new JMenuItem("Quitter");
    item.addActionListener(afficherMenuListener);
    fichierMenu.add(item);

    // Création du menu Editer
    JMenu editerMenu = new JMenu("Editer");
    item = new JMenuItem("Copier");
    item.addActionListener(afficherMenuListener);
    item.setAccelerator(KeyStroke.getKeyStroke('C', Toolkit.getDefaultToolkit()
        .getMenuShortcutKeyMask(), false));
    editerMenu.add(item);
    item = new JMenuItem("Couper");
    item.addActionListener(afficherMenuListener);
    item.setAccelerator(KeyStroke.getKeyStroke('X', Toolkit.getDefaultToolkit()
        .getMenuShortcutKeyMask(), false));
    editerMenu.add(item);
    item = new JMenuItem("Coller");
    item.addActionListener(afficherMenuListener);
    item.setAccelerator(KeyStroke.getKeyStroke('V', Toolkit.getDefaultToolkit()
        .getMenuShortcutKeyMask(), false));
    editerMenu.add(item);

    // Création du menu Divers
    JMenu diversMenu = new JMenu("Divers");
    JMenu sousMenuDiver1 = new JMenu("Sous menu 1");

    item.addActionListener(afficherMenuListener);
    item = new JMenuItem("Sous menu 1 1");
    sousMenuDiver1.add(item);
    item.addActionListener(afficherMenuListener);
    JMenu sousMenuDivers2 = new JMenu("Sous menu 1 2");
    item = new JMenuItem("Sous menu 1 2 1");
    sousMenuDivers2.add(item);
    sousMenuDiver1.add(sousMenuDivers2);

    diversMenu.add(sousMenuDiver1);
    item = new JCheckBoxMenuItem("Validé");
    diversMenu.add(item);
    item.addActionListener(afficherMenuListener);
    diversMenu.addSeparator();
    ButtonGroup buttonGroup = new ButtonGroup();
    item = new JRadioButtonMenuItem("Cas 1");
    diversMenu.add(item);
    item.addActionListener(afficherMenuListener);
    buttonGroup.add(item);
    item = new JRadioButtonMenuItem("Cas 2");
    diversMenu.add(item);
    item.addActionListener(afficherMenuListener);
    buttonGroup.add(item);
    diversMenu.addSeparator();
    diversMenu.add(item = new JMenuItem("Autre",
        new ImageIcon("about_32.png")));
    item.addActionListener(afficherMenuListener);

    // ajout des menus à la barre de menus
    add(fichierMenu);
    add(editerMenu);
    add(diversMenu);
  }

  public static void main(String s[]) {
    JFrame frame = new JFrame("Test de menu");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setJMenuBar(new TestMenuSwing1());
    frame.setMinimumSize(new Dimension(250, 200));
    frame.pack();
    frame.setVisible(true);
  }
}
```

Résultat :



45.9.1. La classe JMenuBar

La classe JMenuBar encapsule une barre de menus qui contient zéro ou plusieurs menus.

La classe JMenuBar utilise la classe DefaultSingleSelectionModel comme modèle de données : un seul de ces menus peut être activé à un instant T.

Pour ajouter des menus à la barre de menus, il faut utiliser la méthode add() de la classe JMenuBar qui attend en paramètre l'instance du menu.

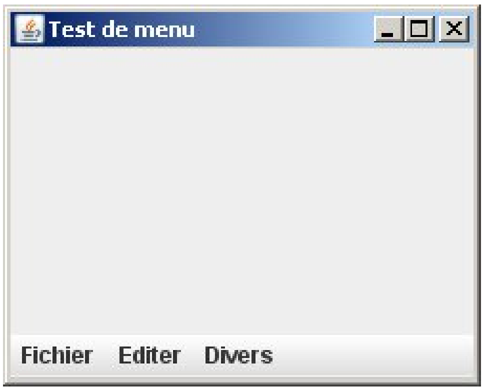
Pour ajouter la barre de menus à une fenêtre, il faut utiliser la méthode setJMenuBar() d'une instance des classes JFrame, JInternalFrame, JDialog ou JApplet.

Comme la classe JMenuBar hérite de la classe JComponent, il est aussi possible d'instancier plusieurs JMenuBar et de les insérer dans un gestionnaire de positionnement comme n'importe quel composant. Ceci permet aussi de placer le menu à sa guise.

Exemple :

```
...
  public static void main(String s[]) {
    JFrame frame = new JFrame("Test de menu");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    TestMenuSwing1 menu = new TestMenuSwing1();
    frame.getContentPane().add(menu, BorderLayout.SOUTH);
    frame.setMinimumSize(new Dimension(250, 200));
    frame.pack();
    frame.setVisible(true);
  }
...
```

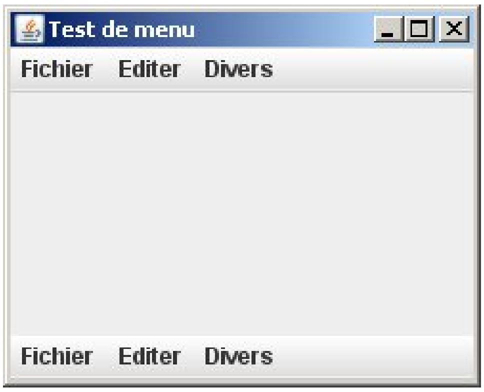
Résultat :



Swing n'impose pas d'avoir un unique menu par fenêtre : il est possible d'avoir plusieurs menus dans une même fenêtre.

Exemple :

```
...
  public static void main(String s[]) {
    JFrame frame = new JFrame("Test de menu");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setJMenuBar(new TestMenuSwing1());
    TestMenuSwing1 menu = new TestMenuSwing1();
    frame.getContentPane().add(menu, BorderLayout.SOUTH);
    frame.setMinimumSize(new Dimension(250, 200));
    frame.pack();
    frame.setVisible(true);
  }
...
```



La classe JMenuBar ne possède qu'un seul constructeur sans paramètre.

Les principales méthodes de la classe JMenuBar sont :

Méthodes	Rôle
JMenu add(JMenu)	Ajouter un menu à la barre de menus
JMenu getMenu(int)	Obtenir le menu dont l'index est fourni en paramètre
int getMenuCount()	Obtenir le nombre de menus de la barre de menus
MenuElement[] getSubElements()	Obtenir un tableau de tous les menus
boolean isSelected()	Retourner true si un composant du menu est sélectionné
void setMenuHelp (JMenu)	Cette méthode n'est pas implémentée et lève systématiquement une exception

45.9.2. La classe JMenuItem

La classe JMenuItem encapsule les données d'un élément de menu (libellé et/ou image). Elle hérite de la classe AbstractButton. Le comportement est similaire mais différent de celui d'un bouton : avec la classe JMenuItem, le composant est considéré comme sélectionné dès que le curseur de la souris passe dessus.

Les éléments de menus peuvent être associés à deux types de raccourcis clavier :

- les accelerators : ils sont hérités de JComponent : ce sont des touches (par exemple les touches de fonctions) ou des combinaisons de touches avec les touches shift, Ctrl ou Alt qui sont affichées à la droite du libellé de l'élément du menu
- les mnemonics : ils apparaissent sous la forme d'une lettre soulignée. Ils sont utilisables seulement sur certaines plates-formes (par exemple en combinaison avec la touche Alt sous Windows).

La méthode setAccelerator() permet d'associer un accelerator à un élément de type JMenuItem.

Un mnemonic peut être associé à un JMenuItm de deux façons :

- soit dans la surcharge du constructeur prévue à cet effet
- soit en utilisant la méthode setMnemonic()

Le mnemonic correspond à un caractère qui doit obligatoirement être contenu dans le libellé.

Un élément de menu peut contenir uniquement une image ou être composé d'un libellé et d'une image. Une image peut être associée à un JMenuItem de deux façons :

- soit dans une des surcharges du constructeur prévues à cet effet
  item = new JMenuItem("Autre", new ImageIcon("about_32.png"));
  item = new JMenuItem(new ImageIcon("about_32.png"));

- soit en utilisant la méthode setIcon
  item.setIcon(new ImageIcon("about_32.png"));

45.9.3. La classe JPopupMenu

La classe JPopupMenu encapsule un menu flottant qui n'est pas rattaché à une barre de menus mais à un composant.

La création d'un JPopMenu est similaire à la création d'un JMenu.

Il est préférable d'ajouter un élément de type JMenuItem grâce à la méthode add() de la classe JPopupMenu mais on peut aussi ajouter n'importe quel élément qui hérite de la classe Component en utilisant une surcharge de la méthode add().

Il est possible d'ajouter un élément à un index précis en utilisant la méthode insert().

La méthode addSeparator() permet d'ajouter un élément séparateur.

Pour afficher un menu flottant, il faut ajouter un listener sur l'événement déclenchant et utiliser la méthode show() de la classe JPopupMenu.

Exemple :

```
package fr.jmdoudoux.dej.swing.menu;

import java.awt.Dimension;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

import javax.swing.JFrame;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPopupMenu;
import javax.swing.JTextField;

public class TestMenuSwing2 extends JMenuBar {

  public JPopupMenu popup;

  public TestMenuSwing2() {

    JMenuItem item = null;

    // Listener générique qui affiche l'action du menu utilisé
    ActionListener afficherMenuListener = new ActionListener() {
      public void actionPerformed(ActionEvent event) {
        System.out.println("Elément de menu [" + event.getActionCommand()
            + "] utilisé.");
      }
    };

    popup = new JPopupMenu();
    item = new JMenuItem("Copier");
    item.addActionListener(afficherMenuListener);
    popup.add(item);
    item = new JMenuItem("Couper");
    item.addActionListener(afficherMenuListener);
    popup.add(item);

  }

  public void processMouseEvent(MouseEvent e) {
  }

  public static void main(String s[]) {
    final JFrame frame = new JFrame("Test de menu divers");
    final JTextField texte = new JTextField();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    final TestMenuSwing2 tms = new TestMenuSwing2();
    frame.add(texte);

    texte.addMouseListener(new MouseAdapter() {

      public void mouseClicked(MouseEvent e) {
        System.out.println("mouse clicked");
        afficherPopup(e);
      }

      public void mousePressed(MouseEvent e) {
        System.out.println("mouse pressed");
        afficherPopup(e);
      }

      public void mouseReleased(MouseEvent e) {
        System.out.println("mouse released");
        afficherPopup(e);
      }

      private void afficherPopup(MouseEvent e) {
        if (e.isPopupTrigger()) {
          tms.popup.show(texte, e.getX(), e.getY());
        }
      }
    });

    frame.setMinimumSize(new Dimension(250, 200));
    frame.pack();
    frame.setVisible(true);
  }
}
```

Le plus simple pour être multiplate-forme est de tester sur tous les événements de la souris ceux qui permettent l'affichage du menu flottant. Ce test est réalisé grâce à la méthode isPopupTrigger() de la classe MouseEvent.

La propriété invoker encapsule le composant à l'origine de l'affichage du menu déroulant.
La propriété borderPaint indique si la bordure du menu déroulant doit être dessinée.
La propriété visible indique si le menu déroulant est affiché.
La propriété location indique les coordonnées d'affichage du menu déroulant

Un objet de type JPopupMenu peut émettre des événements de type PopupMenuEvent. Ceux-ci sont traités par un listener de type PopupMenuListener qui définit trois méthodes :

Méthode	Rôle
popupMenuCanceled()	méthode appelée avant que l'affichage du menu déroulant ne soit annulé
popoupMenuWillBecomeInvisible()	méthode appelée avant que le menu déroulant ne devienne invisible
popoupMenuWillBecomeVisible()	méthode appelée avant que le menu déroulant ne devienne visible. Cette méthode permet de personnaliser l'affichage des éléments du menu en fonction du contexte (exemple : rendre actif ou non certains éléments du menu)

45.9.4. La classe JMenu

La classe JMenu encapsule un menu qui est attaché à un objet de type JMenuBar ou à un autre objet de type JMenu. Dans ce second cas, l'objet est un sous menu.

Il est possible d'ajouter un élément sous la forme d'un objet de type JMenuItem, Component ou Action en utilisant la méthode add(). Chaque élément du menu possède un index.

La méthode addSeparator() permet d'ajouter un élément de type séparateur.
La méthode remove() permet de supprimer un élément du menu en fournissant en paramètre l'instance de l'élément ou son index. Si la suppression réussie, les index des éléments suivants sont décrémentés d'une unité.

La classe JMenu possède plusieurs propriétés :

Propriété	Rôle
popupMenu	JPopupMenu qui encapsule les éléments du menu
topLevelMenu	propriété en lecture seule qui précise si le menu est attaché à un JMenuBar. La valeur false indique que le menu est un sous-menu attaché à un autre menu
itemCount	indique le nombre d'éléments du menu (incluant les séparateurs)
delay	précise le temps en millisecondes avant l'affichage du menu
menuComponentCount	indique le nombre de composants du menu
tearOff	ne pas utiliser cette propriété qui lève une exception de type Error

La méthode getMenuComponent() permet d'obtenir le composant du menu dont l'index est fourni en paramètre. La méthode getItem() permet d'obtenir le JMenuItem dont l'index est fourni en paramètre.

La méthode menuComponents() renvoie un tableau des composants du menu.

La méthode isMenuComponent() renvoie un booléen qui précise si le composant fourni en paramètre est inclus dans les éléments du menu.

Un événement de type MenuEvent est émis lorsque le titre du menu est cliqué. Un listener de type MenuListener permet de s'abonner à ces événements. L'interface MenuListener définie trois méthodes qui possèdent un paramètre de type MenuEvent :

Méthodes	Rôle
menuCanceled()	invoquée lorsque le menu est effacé

menuDeselected()	invoquée lorsque le titre du menu est désélectionné
menuSelected()	invoquée lorsque le titre du menu est sélectionné

45.9.5. La classe JCheckBoxMenuItem

Cette classe encapsule un élément du menu qui contient une case à cocher.

Elle possède de nombreux constructeurs qui permettent de préciser le texte, une icône et l'état de la case à cocher.

La propriété state() permet de connaître ou de définir l'état de la case à cocher.

45.9.6. La classe JRadioButtonMenuItem

Cette classe encapsule un élément de menu qui contient un bouton radio. A un instant donné, un seul des boutons radio associés à un même groupe peut être sélectionné.

La définition de ce groupe se fait en utilisant la classe ButtonGroup. C'est d'ailleurs cette classe qui propose la méthode getSelected() pour connaître le bouton radio sélectionné dans le groupe.

Exemple :

```
...
        diversMenu.addSeparator();
    ButtonGroup buttonGroup = new ButtonGroup();
    item = new JRadioButtonMenuItem("Cas 1");
    diversMenu.add(item);
    item.addActionListener(afficherMenuListener);
    buttonGroup.add(item);
    item = new JRadioButtonMenuItem("Cas 2");
    diversMenu.add(item);
    item.addActionListener(afficherMenuListener);
    buttonGroup.add(item);
    diversMenu.addSeparator();
...
```

45.9.7. La classe JSeparator

La méthode addSeparator() des classes JMenu et JPopupMenu instancie un objet de type JSeparator et l'ajoute à la liste des éléments du menu.

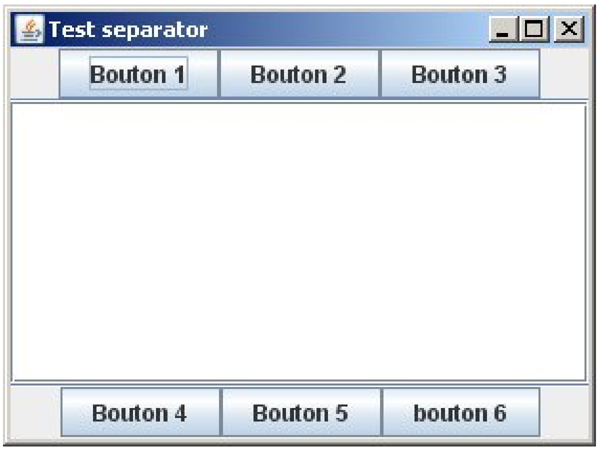
La classe JSeparator encapsule un séparateur dans un menu.

Remarque : L'utilisation de cette classe ne se limite pas aux menus car elle peut aussi être utilisée comme un composant de l'interface.

Exemple :

```
package fr.jmdoudoux.dej.swing.menu;

import java.awt.Dimension;

import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JSeparator;
import javax.swing.JTextField;

public class TestMenuSwing3 extends JPanel {
  public TestMenuSwing3() {
    super(true);

    setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
    Box box1 = new Box(BoxLayout.X_AXIS);
    Box box2 = new Box(BoxLayout.X_AXIS);
    Box box3 = new Box(BoxLayout.X_AXIS);
    Box box4 = new Box(BoxLayout.X_AXIS);
    Box box5 = new Box(BoxLayout.X_AXIS);
    box1.add(new JButton("Bouton 1"));
    box1.add(new JButton("Bouton 2"));
    box1.add(new JButton("Bouton 3"));
    box2.add(new JSeparator());
    box3.add(new JTextField(""));
    box4.add(new JSeparator());
    box5.add(new JButton("Bouton 4"));
    box5.add(new JButton("Bouton 5"));
    box5.add(new JButton("bouton 6"));
    add(box1);
    add(box2);
    add(box3);
    add(box4);
    add(box5);
  }

  public static void main(String s[]) {
    JFrame frame = new JFrame("Test separator");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setContentPane(new TestMenuSwing3());
    frame.setMinimumSize(new Dimension(250, 200));
    frame.pack();
    frame.setVisible(true);
  }
}
```

Résultat :



45.10. L'affichage d'une image dans une application.

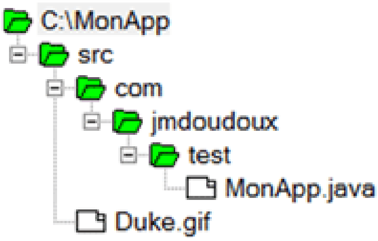
Pour afficher une image dans une fenêtre, il y a plusieurs solutions.

La plus simple consiste à utiliser le composant JLabel qui est capable d'afficher du texte mais aussi une image

Exemple :

```
package fr.jmdoudoux.dej;

import java.awt.BorderLayout;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class MonApp  extends JFrame {

  private static final long serialVersionUID = 1L;

  public MonApp(String titre) {
    super(titre);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    init();
  }

  private void init()
  {
    JLabel label = new JLabel(new ImageIcon("Duke.gif") );
    this.add(label, BorderLayout.CENTER);
    this.pack();
  }

  public static void main(String[] args) {
    MonApp app = new MonApp("Afficher image");
    app.setVisible(true);
  }
}
```

Dans l'exemple ci-dessus, le fichier contenant l'image doit être à la racine des fichiers class : aucun chemin n'est précisé donc c'est le chemin relatif au répertoire d'exécution de l'application qui est retenu. Il est possible de préciser un chemin absolu mais cela limite les possibilités de déploiement de l'application.



C:\MonApp\src>javac com/jmdoudoux/test/MonApp.java

C:\MonApp\src>java fr.jmdoudoux.dej.MonApp

Il est possible de définir un composant personnalisé qui hérite de la classe JPanel qui va se charger d'afficher l'image.

Historiquement, c'est la classe java.awt.Toolkit qui peut être utilisée pour charger une image.

Exemple :

```
package fr.jmdoudoux.dej;

import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.MediaTracker;
import java.awt.Panel;
import java.awt.Toolkit;

/**
 * Composant qui affiche une image
 */
public class AfficheImage extends Panel {

  private static final long serialVersionUID = 1L;
  private Image image;

  public AfficheImage(String filename) {
    image = Toolkit.getDefaultToolkit().getImage("./duke.gif");
    try {
      MediaTracker mt = new MediaTracker(this);
      mt.addImage(image, 0);
      mt.waitForAll();
    } catch (Exception e) {
      e.printStackTrace();
    }
    this.setPreferredSize(new Dimension(image.getWidth(this), image
        .getHeight(this)));
  }

  public void paint(Graphics g) {
    g.drawImage(image, 0, 0, null);
  }
}
```

L'inconvénient d'utiliser la classe Toolkit pour charger une image est que ce chargement se fait de façon asynchrone. Il faut alors utiliser une instance de la classe MediaTracker pour patienter le temps du chargement de l'image et ainsi pouvoir déterminer sa taille pour la reporter sur la taille du composant.

A partir de Java 1.4, il est aussi possible d'utiliser la classe javax.imageio.ImageIO pour simplifier le code qui charge l'image.

Exemple :

```
package fr.jmdoudoux.dej;

import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Panel;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;

/**
 * Composant qui affiche une image
 */
public class AfficheImage extends Panel {

  private static final long serialVersionUID = 1L;
  private BufferedImage image;

  public AfficheImage(String nomFichier) {

    try {
      image = ImageIO.read(new File(nomFichier));
      this.setPreferredSize(new Dimension(image.getWidth(),
        image.getHeight()));
    } catch (IOException ie) {
      ie.printStackTrace();
    }
  }

  public void paint(Graphics g) {
    g.drawImage(image, 0, 0, null);
  }
}
```

Il suffit alors d'utiliser le composant dans la fenêtre

Exemple :

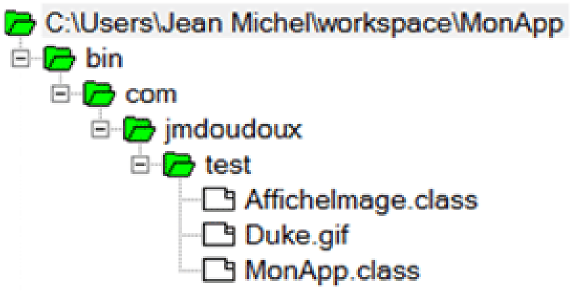
```
package fr.jmdoudoux.dej;
import java.awt.BorderLayout;
import javax.swing.JFrame;

public class MonApp extends JFrame {

  private static final long serialVersionUID = 1L;
  public MonApp(String titre) {
    super(titre);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    init();
  }

  private void init()
  {
    AfficheImage afficheImage = new AfficheImage("Duke.gif");
    this.setLayout(new BorderLayout());
    this.add(afficheImage, BorderLayout.CENTER);
    this.pack();
  }

  public static void main(String[] args) {
    MonApp app = new MonApp("Afficher image");
    app.setVisible(true);
  }
}
```

Malheureusement, ces deux solutions ne fonctionnent pas si l'application est packagée sous la forme d'une archive qui contient l'image car l'API java.io n'est pas capable de lire une ressource dans l'archive jar. Il faut utiliser le classloader pour charger l'image sous la forme d'une ressource. L'avantage de cette solution c'est qu'elle fonctionne que l'application soit packagée ou non.

Exemple :

```
package fr.jmdoudoux.dej;

import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.MediaTracker;
import java.awt.Panel;
import java.awt.Toolkit;

/**
 * Composant qui affiche une image
 */
public class AfficheImage extends Panel {

  private static final long serialVersionUID = 1L;
  private Image image;

  public AfficheImage(String filename) {
    java.net.URL url = this.getClass().getResource("Duke.gif");
    image = Toolkit.getDefaultToolkit().getImage(url);
    try {
      MediaTracker mt = new MediaTracker(this);
      mt.addImage(image, 0);
      mt.waitForAll();
    } catch (Exception e) {
      e.printStackTrace();
    }
    this.setPreferredSize(new Dimension(image.getWidth(this), image
        .getHeight(this)));
  }

  public void paint(Graphics g) {
    g.drawImage(image, 0, 0, null);
  }
}
```

Le fichier contenant l'image doit être accessible par le classloader dans le classpath, par exemple :



```
C:\Users\Jean Michel\workspace\MonApp
  bin
    com
      jmdoudoux
        test
          AfficheImage.class
          Duke.gif
          MonApp.class
```

46. Le développement d'interfaces graphiques avec SWT

Chapitre 46

Niveau : 🟩🟩🟨⬜⬜⬜ Intermédiaire

La première API pour développer des interfaces graphiques portables d'un système à un autre en Java est AWT. Cette API repose sur les composants graphiques du système sous-jacent ce qui lui assure de bonnes performances. Malheureusement, ces composants sont limités dans leur fonctionnalité car ils représentent le plus petit dénominateur commun des différents systèmes concernés.

Pour pallier ce problème, Sun a proposé une nouvelle API, Swing. Cette Api est presque exclusivement écrite en Java, ce qui assure sa portabilité. Swing possède aussi d'autres points forts, telles que des fonctionnalités avancées, la possibilité d'étendre les composants, une adaptation du rendu de composants, etc ... Swing est une API mature, éprouvée et parfaitement connue. Malheureusement, ses deux gros défauts sont sa consommation en ressource machine et la lenteur d'exécution des applications qui l'utilisent.

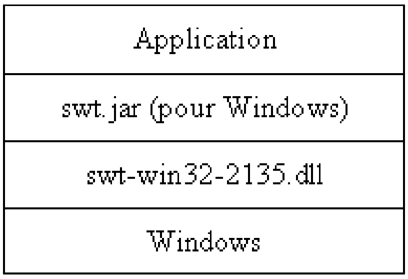
SWT propose une approche intermédiaire : utiliser autant que possible les composants du système et implémenter les autres composants en Java. SWT est écrit en Java et utilise la technologie JNI pour appeler les composants natifs. SWT utilise autant que possible les composants natifs du système lorsqu'ils existent, sinon ils sont réécrits en pur Java. Les données de chaque composant sont aussi stockées autant que possible dans le composant natif, limitant ainsi les données stockées dans les objets Java correspondant.

Ce chapitre contient plusieurs sections :

- La présentation de SWT
- Un exemple très simple
- La classe SWT
- L'objet Display
- L'objet Shell
- Les composants
- Les conteneurs
- La gestion des erreurs
- Le positionnement des contrôles
- La gestion des événements
- Les boîtes de dialogue

46.1. La présentation de SWT

Une partie de SWT est livrée sous la forme d'une bibliothèque dépendante du système d'exploitation et d'un fichier .jar lui aussi dépendant du système. Toutes les fonctionnalités de SWT ne sont implémentées que sur les systèmes où elles sont supportées (exemple, l'utilisation des ActiveX n'est possible que sur le portage de SWT sur les systèmes Windows).



Les trois avantages de SWT sont donc la rapidité d'exécution, des ressources machines moins importantes lors de l'exécution et un rendu parfait des composants graphiques selon le système utilisé puisqu'il utilise des composants natifs. Cette dernière remarque est particulièrement vraie pour des environnements graphiques dont l'apparence est modifiable.

Malgré cette dépendance vis à vis du système graphique de l'environnement d'exécution, l'API de SWT reste la même quelque soit la plate-forme utilisée.

En plus de dépendre du système utilisé lors de l'exécution, SWT possède un autre petit inconvénient. N'utilisant pas de purs objets java, il n'est pas possible de compter sur le ramasse-miettes pour libérer la mémoire des composants créés manuellement. Pour libérer cette mémoire, il est nécessaire d'utiliser la méthode dispose() pour les composants instanciés lorsque ceux-ci ne sont plus utiles.

Pour faciliter ces traitements, l'appel de la méthode dispose() d'un composant entraîne automatiquement l'appel de la méthode dispose() des composants qui lui sont rattachés. Il faut toutefois rester vigilant lors de l'utilisation de certains objets qui ne sont pas des contrôles tels que les objets de type Font ou Color, qu'il convient de libérer explicitement sous peine de fuites de mémoire.

Les règles à observer pour la libération des ressources sont :

- toujours appeler la méthode dispose() de tout objet non rattaché directement à un autre objet qui n'est plus utilisé.
- ne jamais appeler la méthode dispose() d'objets qui n'ont pas été explicitement instanciés dans le code
- l'appel de la méthode dispose() d'un composant entraîne automatiquement l'appel de la méthode dispose() des composants qui lui sont rattachés

Attention, l'utilisation d'un objet dont la méthode dispose() a été appelée induira un résultat imprévisible.

Ainsi SWT pose à nouveau la problématique concernant la dualité entre la portabilité (Write Once Run Anywhere) et les performances.

SWT se fonde sur trois concepts classiques dans le développement d'une interface graphique :

- Les composants ou contrôles (widgets)
- Un système de mise en page et de présentation des composants
- Un modèle de gestion des événements

La structure d'une application SWT est la suivante :

- la création d'un objet de type Display qui assure le dialogue avec le système sous-jacent
- la création d'un objet de type Shell qui est la fenêtre de l'application
- la création des composants et leur ajout dans le Shell
- l'enregistrement des listeners pour le traitement des événements
- l'exécution de la boucle de gestion des événements jusqu'à la fin de l'application
- la libération des ressources de l'objet Display

La version de SWT utilisée dans ce chapitre est la 2.1.

SWT est regroupé dans plusieurs packages :

Package	Rôle
org.eclipse.swt	Package de base qui contient la définition de constantes et d'exceptions

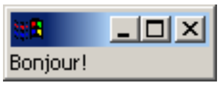
org.eclipse.swt.accessibility	
org.eclipse.swt.custom	Contient des composants particuliers
org.eclipse.swt.dnd	Contient les éléments pour le support du « cliqué / glissé »
org.eclipse.swt.events	Contient les éléments pour la gestion des événements
org.eclipse.swt.graphics	Contient les éléments pour l'utilisation des éléments graphiques (couleur, polices, curseur, contexte graphique, ...)
org.eclipse.swt.layout	Contient les éléments pour la gestion de la présentation
org.eclipse.swt.ole.win32	Contient les éléments pour le support d'OLE 32 sous Windows
org.eclipse.swt.printing	Contient les éléments pour le support des impressions
org.eclipse.swt.program	
org.eclipse.swt.widgets	Contient les différents composants

46.2. Un exemple très simple

L'exemple de cette section affiche simplement bonjour dans une fenêtre.

Exemple :

```
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.*;

public class TestSWT1 {

  public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);

    Label label = new Label(shell, SWT.CENTER);
    label.setText("Bonjour!");
    label.pack();

    shell.pack();
    shell.open();

    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }

    display.dispose();
    label.dispose();
  }
}
```



Bonjour!

Pour exécuter cet exemple sous Windows, il faut que le fichier swt.jar correspondant à la plate-forme Windows soit inclus dans le classpath et que l'application puisse accéder à la bibliothèque swt-win32-2135.dll.

46.3. La classe SWT

Cette classe définit un certain nombre de constantes concernant les styles. Les styles sont des comportements ou des caractéristiques définissant l'apparence du composant. Ces styles sont directement fournis dans le constructeur d'une classe encapsulant un composant.

46.4. L'objet Display

Toute application SWT doit obligatoirement instancier un objet de type Display. Cet objet assure le dialogue entre l'application et le système graphique du système d'exploitation utilisé.

Exemple :

```
Display display = new Display();
```

La méthode la plus importante de la classe Display est la méthode readAndDispatch() qui lit les événements dans la pile du système graphique natif pour les diffuser à l'application. Elle renvoie true s'il y a encore des traitements à effectuer sinon elle renvoie false.

La méthode sleep() permet de mettre en attente le thread d'écoute d'événements jusqu'à l'arrivée d'un nouvel événement.

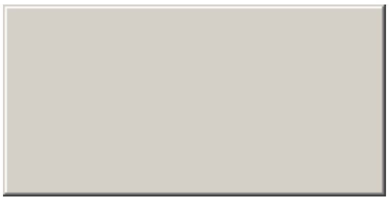
Il est absolument nécessaire lors de la fin de l'application de libérer les ressources allouées par l'objet de type Display en appelant sa méthode dispose().

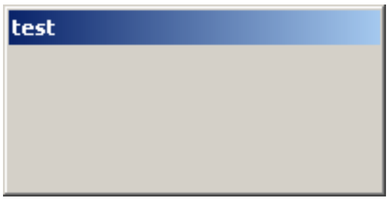
46.5. L'objet Shell

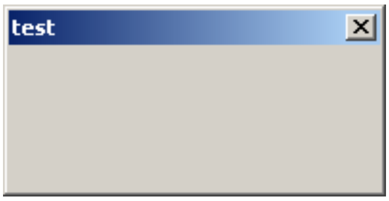
L'objet Shell représente une fenêtre gérée par le système graphique du système d'exploitation utilisé.

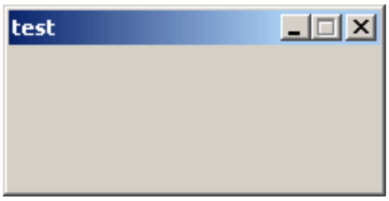
Un objet de type Shell peut être associé à un objet de type Display pour obtenir une fenêtre principale ou être associé à un autre objet de type Shell pour obtenir une fenêtre secondaire.

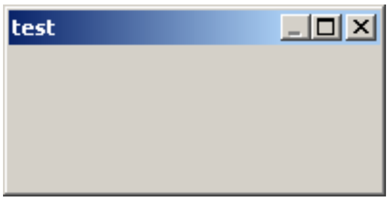
La classe Shell peut utiliser plusieurs styles : BORDER,H_SCROLL, V_SCROLL, CLOSE, MIN, MAX, RESIZE, TITLE, SHELL_TRIM, DIALOG_TRIM

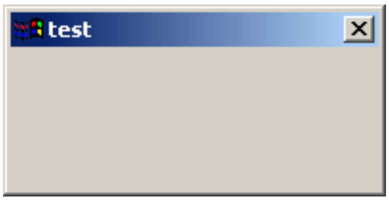
BORDER : une fenêtre avec une bordure sans barre de titre
`Shell shell = new Shell(display, SWT.BORDER);`
`shell.setSize(200, 100) ;`
`shell.setText("test");`
	

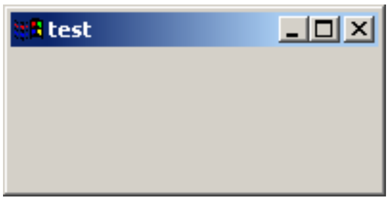
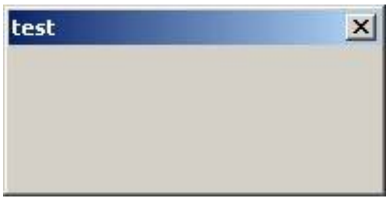
TITLE : une fenêtre avec une barre de titre
`Shell shell = new Shell(display, SWT.TITLE);`
`shell.setSize(200, 100) ;`
`shell.setText("test");`
	

CLOSE : une fenêtre avec un bouton de fermeture
`Shell shell = new Shell(display, SWT.CLOSE);`
`shell.setSize(200, 100);`
`shell.setText("test");`
	

MIN : une fenêtre avec un bouton pour iconiser
`Shell shell = new Shell(display, SWT.CLOSE | SWT.MIN);`
`shell.setSize(200, 100);`
`shell.setText("test");`
	

MAX : une fenêtre avec un bouton pour agrandir au maximum
`Shell shell = new Shell(display, SWT.CLOSE | SWT.MAX);`
`shell.setSize(200, 100);`
`shell.setText("test");`
	

RESIZE : une fenêtre dont la taille peut être modifiée
`Shell shell = new Shell(display, SWT.CLOSE | SWT.RESIZE);`
`shell.setSize(200, 100);`
`shell.setText("test");`
	

SHELL_TRIM : groupe en une seule constante les styles CLOSE, TITLE, MIN, MAX et RESIZE	
DIALOG_TRIM : groupe en une seule constante les styles CLOSE, TITLE et BORDER	
APPLICATION_MODAL :	
SYSTEM_MODAL :	

La méthode setSize() permet de préciser la taille de la fenêtre.

La méthode setText() permet de préciser le titre de la fenêtre.

Exemple : centrer la fenêtre sur l'écran

```
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.*;

public class TestSWT1 {

  public static void centrerSurEcran(Display display, Shell shell) {
    Rectangle rect = display.getClientArea();
    Point size = shell.getSize();
    int x = (rect.width - size.x) / 2;
    int y = (rect.height - size.y) / 2;
    shell.setLocation(new Point(x, y));
  }

  public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setSize(340, 100);
    centrerSurEcran(display, shell);

    shell.open();

    while (!shell.isDisposed())
      if (!display.readAndDispatch())
        display.sleep();

    display.dispose();
  }
}
```

46.6. Les composants

Les composants peuvent être regroupés en deux grandes familles :

- les contrôles qui sont des composants graphiques. Ils héritent tous de la classe abstraite Control
- les conteneurs qui permettent de grouper des contrôles. Ils héritent tous de la classe abstraite Composite

Une application SWT est une hiérarchie de composants dont la racine est un objet de type Shell.

Certaines caractéristiques comme l'apparence ou le comportement d'un contrôle doivent être fournies au moment de leur création par le système graphique. Ainsi, chaque composant SWT possède une propriété nommée style fournie en paramètre du constructeur.

Plusieurs styles peuvent être combinés avec l'opérateur | . Cependant certains styles sont incompatibles entre-eux pour certains composants.

46.6.1. La classe Control

La classe Control définit trois styles : BORDER, LEFT_TO_RIGHT et RIGHT_TO_LEFT

Le seul constructeur de la classe Control nécessite aussi de préciser le composant père sous la forme d'un objet de type Composite. L'association avec le composant père est obligatoire pour tous les composants lors de leur création.

La classe Control possède plusieurs méthodes pour enregistrer des listeners pour certains événements. Ces événements sont : FocusIn, FocusOut, Help, KeyDown, KeyUp, MouseDoubleClick, MouseDown, MouseEnter, MouseExit, MouseHover, MouseUp, MouseMove, Move, Paint, Resize.

Elle possède aussi plusieurs méthodes dont les principales sont :

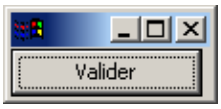
Nom	Rôle
boolean forceFocus()	Force le focus au composant pour lui permettre de recevoir les événements clavier
Display getDisplay()	Renvoie l'objet Display associé au composant
Shell getShell()	Renvoie l'objet Shell associé au composant
void pack()	Recalcule la taille préférée du composant
void SetEnabled()	Permet de rendre actif le composant
void SetFocus()	Donne le focus au composant pour lui permettre de recevoir les événements clavier
void setSize()	Permet de modifier la taille du composant
void setVisible()	Permet de rendre visible ou non le composant

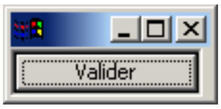
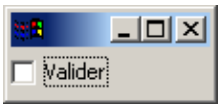
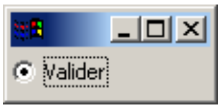
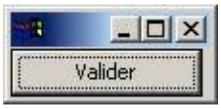
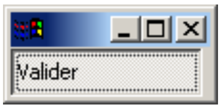
46.6.2. Les contrôles de base

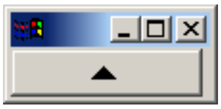
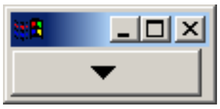
46.6.2.1. La classe Button

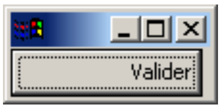
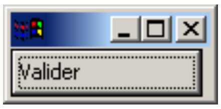
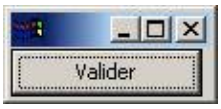
La classe Button représente un bouton cliquable.

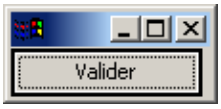
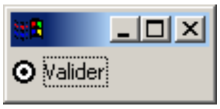
La classe Button définit plusieurs styles : BORDER, CHECK, PUSH, RADIO, TOGGLE, FLAT, LEFT, RIGHT, CENTER, ARROW (avec UP, DOWN)

NONE : un bouton par défaut
`button = new Button(shell, SWT.NONE);`
`button.setText("Valider");`
`button.setSize(100,25);`
	

BORDER : met une bordure autour du bouton `Button button = new Button(shell, SWT.BORDER);`	
CHECK : une case à cocher `Button button = new Button(shell, SWT.CHECK);`	
RADIO : un bouton radio `Button button = new Button(shell, SWT.RADIO);`	
PUSH : un bouton standard (valeur par défaut) `Button button = new Button(shell, SWT.PUSH);`	
TOGGLE : un bouton pouvant conservé un état enfoncé `Button button = new Button(shell, SWT.TOGGLE);`	

ARROW : bouton en forme de flèche (par défaut vers le haut)
`Button button = new Button(shell, SWT.ARROW);`
`Button button = new Button(shell, SWT.ARROW | SWT.DOWN);`
	 

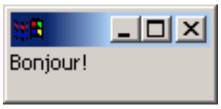
RIGHT : aligne le contenu du bouton sur la droite `Button button = new Button(shell, SWT.RIGHT);`	
LEFT : aligne le contenu du bouton sur la gauche `Button button = new Button(shell, SWT.LEFT);`	
CENTER : centre le contenu du bouton `Button button = new Button(shell, SWT.CENTER);`	

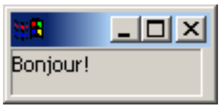
FLAT : le bouton apparaît en 2D
`Button button = new Button(shell, SWT.FLAT);`
`Button button = new Button(shell, SWT.FLAT | SWT.RADIO);`
	 

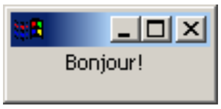
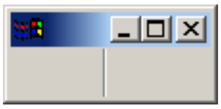
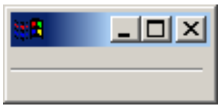
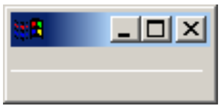
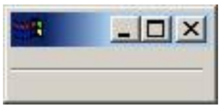
46.6.2.2. La classe Label

Ce contrôle permet d'afficher un libellé ou une image

La classe Label possède plusieurs styles : BORDER, CENTER, LEFT, RIGHT, WRAP, SEPARATOR (avec HORIZONTAL, SHADOW_IN, SHADOW_OUT, SHADOW_NONE, VERTICAL)

NONE : un libellé par défaut
`Label label = new Label(shell, SWT.NONE);`
`label.setText("Bonjour!");`
`label.setSize(100,25);`
	

BORDER : ajouter une bordure autour du libellé `Label label = new Label(shell, SWT.BORDER);`	

CENTER : permet de centrer le libellé `Label label = new Label(shell, SWT.CENTER);`	 Bonjour!
SEPARATOR et VERTICAL : une barre verticale `Label label = new Label(shell, SWT.SEPARATOR | SWT. VERTICAL);`	
SEPARATOR et HORIZONTAL : une barre horizontale `Label label = new Label(shell, SWT.SEPARATOR | SWT.HORIZONTAL);`	
SHADOW_IN : `Label label = new Label(shell, SWT.SEPARATOR | SWT.HORIZONTAL | SWT.SHADOW_IN);`	
SHADOW_OUT : `Label label = new Label(shell, SWT.SEPARATOR | SWT.HORIZONTAL | SWT.SHADOW_OUT);`	

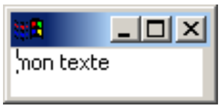
Cette classe possède plusieurs méthodes dont les principales sont :

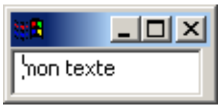
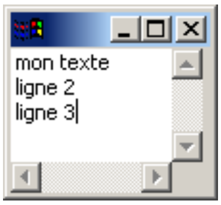
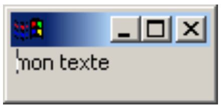
Nom	Rôle
void setAlignment(int)	Permet de préciser l'alignement des données du contrôle
void setImage(Image)	Permet de préciser une image affichée par le contrôle
void setText(string)	Permet de préciser le texte du contrôle

46.6.2.3. La classe Text

Ce contrôle est une zone de saisie de texte.

La classe Text possède plusieurs styles : BORDER, SINGLE, READ_ONLY, LEFT, CENTER, RIGHT, WRAP, MULTI (avec H_SCROLL, V_SCROLL)

NONE : une zone de saisie sans bordure
`Text text = new Text(shell, SWT.NONE);`
`text.setText("mon texte");`
`text.setSize(100,25);`
	 mon texte

BORDER : une zone de saisie avec bordure `Text text = new Text(shell, SWT.BORDER);`	 mon texte
MULTI, SWT.H_SCROLL, SWT.V_SCROLL : une zone de saisie avec bordure `Text text = new Text(shell, SWT.MULTI | SWT.H_SCROLL | SWT.V_SCROLL);`	 mon texte ligne 2 ligne 3
READ_ONLY : une zone de saisie en lecture seule	 mon texte

Cette classe possède plusieurs méthodes dont les principales sont :

Nom	Rôle
void setEchoChar(char)	Caractère affiché lors de la frappe d'une touche
void setTextLimit(int)	Permet de préciser le nombre maximum de caractères saisissables
void setText(string)	Permet de préciser le contenu de la zone de texte
void setEditable(boolean)	Permet de rendre le contrôle éditable ou non

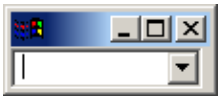
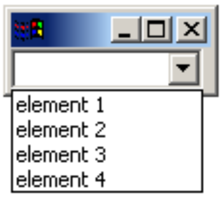
46.6.3. Les contrôles de type liste

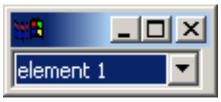
SWT permet de créer des composants de type liste et liste déroulante.

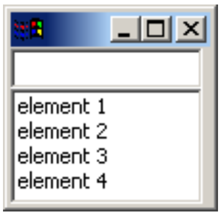
46.6.3.1. La classe Combo

Ce contrôle est une liste déroulante dans laquelle l'utilisateur peut sélectionner une valeur dans une liste d'éléments prédéfinis ou saisir un élément.

La classe Combo définit trois styles : BORDER, DROP_DOWN, READ_ONLY, SIMPLE

BORDER : une liste déroulante
`Combo combo = new Combo(shell, SWT.BORDER);`
`combo.add("element 1");`
`combo.add("element 2");`
`combo.add("element 3");`
`combo.add("element 4");`
`combo.setSize(100,41);`
	 

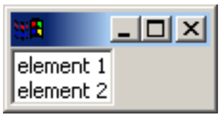
READ_ONLY : une liste déroulante ne permettant que la sélection (saisie d'un élément impossible) `Combo combo = new Combo(shell, SWT.BORDER | SWT.READ_ONLY);`	

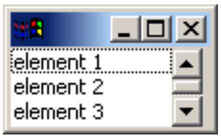
SIMPLE : zone de saisie et une liste
`Combo combo = new Combo(shell, SWT.BORDER | SWT.SIMPLE);`
`combo.setSize(100,81);`
	

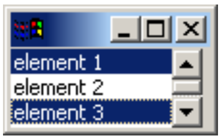
46.6.3.2. La classe List

Ce contrôle est une liste qui permet de sélectionner un ou plusieurs éléments.

La classe List possède plusieurs styles : BORDER, H_SCROLL, V_SCROLL, SINGLE, MULTI

BORDER : une liste
`List liste = new List(shell, SWT.BORDER);`
`liste.add("element 1");`
`liste.add("element 2");`
`liste.pack();`
	

V_SCROLL : une liste avec une barre de défilement
`List liste = new List(shell, SWT.V_SCROLL);`
`liste.add("element 1");`
`liste.add("element 2");`
`liste.add("element 3");`
`liste.add("element 4");`
	

`liste.setSize(100,41);`	
MULTI : une liste avec sélection de plusieurs éléments `List liste = new List(shell, SWT.V_SCROLL | SWT.MULTI);`	

La méthode add() permet d'ajouter un élément à la liste sous la forme d'une chaînes de caractères.

La méthode setItems() permet de fournir les éléments de la liste sous la forme d'un tableau de chaînes de caractères.

Exemple :

```
List liste = new List(shell, SWT.V_SCROLL | SWT.MULTI);
liste.setItems(new String[] {"element 1", "element 2", "element 3", "element 4"});
liste.setSize(100,41);
```

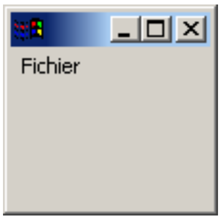
46.6.4. Les contrôles pour les menus

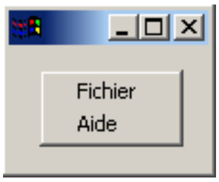
SWT permet la création de menus principaux et de menus déroulants. La création de ces menus met en oeuvre deux classes : Menu, MenuItem

46.6.4.1. La classe Menu

Ce contrôle est un élément du menu qui va contenir des options

La classe Menu possède plusieurs styles : BAR, DROP_DOWN, NO_RADIO_GROUP, POP_UP

BAR : le menu principal d'une fenêtre
`Menu menu = new Menu(shell, SWT.BAR);`
`MenuItem menuitem1 = new MenuItem(menu, SWT.CASCADE);`
`menuitem1.setText("Fichier");`
`shell.setMenuBar(menu);`
	

POP_UP : un menu contextuel
`Menu menu = new Menu(shell, SWT.POP_UP);`
`MenuItem menuitem1 = new MenuItem(menu, SWT.CASCADE);`
`menuitem1.setText("Fichier");`
`MenuItem menuitem2 = new MenuItem(menu, SWT.CASCADE);`
`menuitem2.setText("Aide");`
`shell.setMenu(menu);`
	

DROP_DOWN : un sous menu	
NO_RADIO_GROUP :	

46.6.4.2. La classe MenuItem

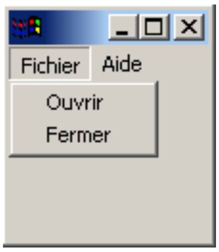
Ce contrôle est une option d'un menu.

La classe MenuItem possède plusieurs styles : CHECK, CASCADE, PUSH, RADIO, SEPARATOR

CASCADE : une option de menu qui possède un sous menu

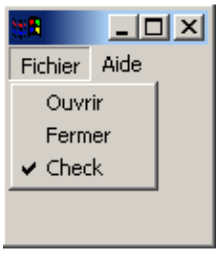
PUSH : une option de menu

```
Menu menu = new Menu(shell, SWT.BAR);
MenuItem optionFichier = new MenuItem(menu, SWT.CASCADE);
optionFichier.setText("Fichier");
Menu menuFichier = new Menu(shell, SWT.DROP_DOWN);
MenuItem optionOuvrir = new MenuItem(menuFichier, SWT.PUSH);
optionOuvrir.setText("Ouvrir");
MenuItem optionFermer = new MenuItem(menuFichier, SWT.PUSH);
optionFermer.setText("Fermer");
optionFichier.setMenu(menuFichier);
MenuItem optionAide = new MenuItem(menu, SWT.CASCADE);
optionAide.setText("Aide");
shell.setMenuBar(menu);
```



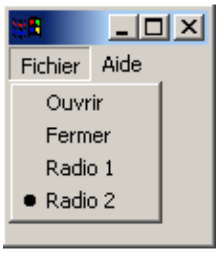
CHECK : une option de menu avec un état coché ou non

```
Menu menu = new Menu(shell, SWT.BAR);
MenuItem optionFichier = new MenuItem(menu, SWT.CASCADE);
optionFichier.setText("Fichier");
Menu menuFichier = new Menu(shell, SWT.DROP_DOWN);
MenuItem optionOuvrir = new MenuItem(menuFichier, SWT.CASCADE);
optionOuvrir.setText("Ouvrir");
MenuItem optionFermer = new MenuItem(menuFichier, SWT.CASCADE);
optionFermer.setText("Fermer");
MenuItem optionCheck = new MenuItem(menuFichier, SWT.CHECK);
optionCheck.setText("Check");
optionFichier.setMenu(menuFichier);
MenuItem optionAide = new MenuItem(menu, SWT.CASCADE);
optionAide.setText("Aide");
shell.setMenuBar(menu);
```



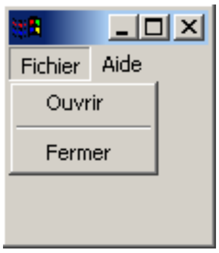
Radio : une option de menu sélectionnable parmi un ensemble

```
Menu menu = new Menu(shell, SWT.BAR);
MenuItem optionFichier = new MenuItem(menu, SWT.CASCADE);
optionFichier.setText("Fichier");
Menu menuFichier = new Menu(shell, SWT.DROP_DOWN);
MenuItem optionOuvrir = new MenuItem(menuFichier, SWT.CASCADE);
optionOuvrir.setText("Ouvrir");
MenuItem optionFermer = new MenuItem(menuFichier, SWT.CASCADE);
optionFermer.setText("Fermer");
MenuItem optionCheck = new MenuItem(menuFichier, SWT.CHECK);
optionCheck.setText("Check");
optionFichier.setMenu(menuFichier);
MenuItem optionAide = new MenuItem(menu, SWT.CASCADE);
optionAide.setText("Aide");
shell.setMenuBar(menu);
```



SEPARATOR : pour séparer les options d'un menu

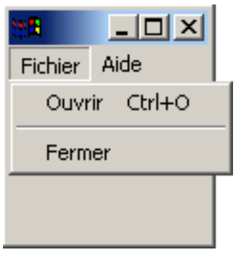
```
Menu menu = new Menu(shell, SWT.BAR);
MenuItem optionFichier = new MenuItem(menu, SWT.CASCADE);
optionFichier.setText("Fichier");
Menu menuFichier = new Menu(shell, SWT.DROP_DOWN);
MenuItem optionOuvrir = new MenuItem(menuFichier, SWT.PUSH);
optionOuvrir.setText("Ouvrir");
MenuItem optionSeparator = new MenuItem(menuFichier, SWT.SEPARATOR);
MenuItem optionFermer = new MenuItem(menuFichier, SWT.PUSH);
optionFermer.setText("Fermer");
optionFichier.setMenu(menuFichier);
MenuItem optionAide = new MenuItem(menu, SWT.CASCADE);
optionAide.setText("Aide");
shell.setMenuBar(menu);
```



La méthode setText() permet de préciser le libellé de l'option de menu.

La méthode setAccelerator() permet de préciser un raccourci clavier.

```
Menu menuFichier = new Menu(shell, SWT.DROP_DOWN);
MenuItem optionOuvrir = new MenuItem(menuFichier, SWT.PUSH);
optionOuvrir.setText("&Ouvrir\tCtrl+O");
optionOuvrir.setAccelerator(SWT.CTRL+'O');
```



46.6.5. Les contrôles de sélection ou d'affichage d'une valeur

SWT propose un contrôle pour l'affichage d'une barre de progression et deux contrôles pour la sélection d'une valeur numérique dans une plage de valeur.

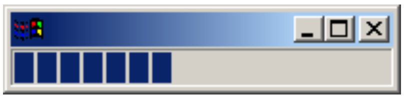
46.6.5.1. La classe ProgressBar

Ce contrôle est une barre de progression.

La classe ProgressBar possède plusieurs styles : BORDER, INDETERMINATE, SMOOTH, HORIZONTAL, VERTICAL

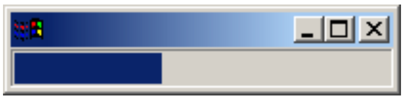
HORIZONTAL :

```
ProgressBar progressbar = new ProgressBar(shell,
  SWT.HORIZONTAL);
progressbar.setMinimum(1);
progressbar.setMaximum(100);
progressbar.setSelection(40);
progressbar.setSize(200,20);
```



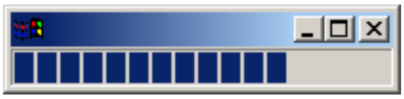
SMOOTH :

```
ProgressBar progressbar = new ProgressBar(shell,
  SWT.HORIZONTAL | SWT.SMOOTH);
```



INDETERMINATE : la barre de progression s'incrémente automatiquement et revient au début indéfiniment

```
ProgressBar progressbar = new ProgressBar(shell,
  SWT.INDETERMINATE);
```



Les méthodes setMinimum() et setMaximum() permettent respectivement de préciser les valeurs minimale et maximale du contrôle.

La méthode setSelection() permet de positionner la valeurs courante de l'indicateur.

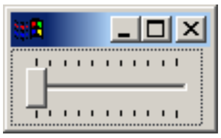
46.6.5.2. La classe Scale

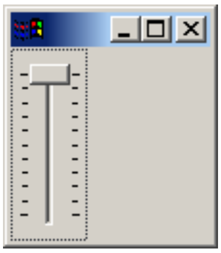
Ce contrôle permet de faire une sélection dans une plage de valeurs numériques.

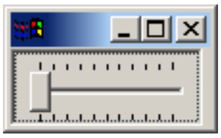
La classe Scale possède trois styles : BORDER, HORIZONTAL, VERTICAL

HORZONTAL :

```
Scale scale = new Scale(shell,SWT.HORIZONTAL);
scale.setSize(100,40);
```



VERTICAL :
`Scale scale = new Scale(shell,SWT.VERTICAL);`
`scale.setSize(40,100);`
	

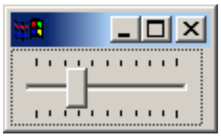
BORDER :
`Scale scale = new Scale(shell,SWT.BORDER);`
`scale.setSize(100,40);`
	

Les méthodes setMinimum() et setMaximum() permettent respectivement de préciser les valeurs minimale et maximale du contrôle.

La méthode setSelection() permet de positionner le curseur dans la plage de valeurs à la valeur fournie en paramètre.

La méthode setPageIncrement() permet de préciser la valeur fournie en paramètre d'incrémentation d'une page

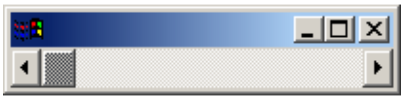
Exemple :

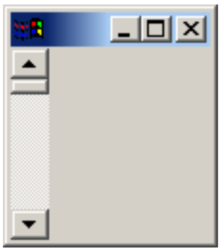
Scale scale = new Scale(shell,SWT.HORIZONTAL);
`scale.setSize(100,40);`
`scale.setMinimum(1);`
`scale.setMaximum(100);`
`scale.setSelection(30);`
`scale.setPageIncrement(10);`
	

46.6.5.3. La classe Slider

Ce contrôle permet de sélectionner une valeur dans une plage de valeurs numériques.

La classe Slider possède trois styles : BORDER, HORIZONTAL, VERTICAL

BORDER :
`Slider slider = new Slider(shell, SWT.BORDER);`
`slider.setSize(200,20);`
	

VERTICAL :
`Slider slider = new Slider(shell, SWT.VERTICAL);`
`slider.setSize(20,200);`
	

Les méthodes setMinimum() et setMaximum() permettent respectivement de préciser les valeurs minimale et maximale du contrôle.

La méthode setSelection() permet de positionner le curseur dans la plage de valeurs à la valeur fournie en paramètre.

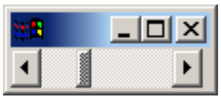
La méthode setPageIncrement() permet de préciser la valeur d'incrémentation d'une page du contrôle.

La méthode setThumb() permet de préciser la taille du curseur.

Exemple :

Slider slider = new Slider(shell,SWT.HORIZONTAL);

```
slider.setMinimum(1);
slider.setMaximum(110);
slider.setSelection(30);
slider.setThumb(10);
slider.setSize(100,20);
```



46.6.6. Les contrôles de type « onglets »

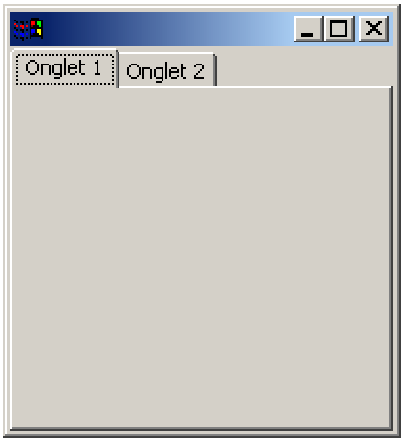
SWT propose la création de composants de type onglets mettant en oeuvre deux classes : TabFolder et TabItem

46.6.6.1. La classe TabFolder

Ce contrôle est un ensemble d'onglets.

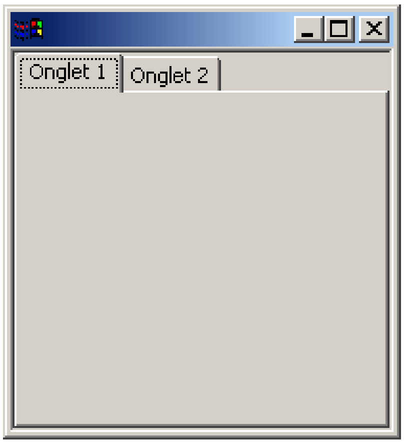
NONE :

```
TabFolder tabfolder = new TabFolder(shell, SWT.NONE);
tabfolder.setSize(200,200);
TabItem onglet1 = new TabItem(tabfolder, SWT.NONE);
onglet1.setText("Onglet 1");
TabItem onglet2 = new TabItem(tabfolder, SWT.NONE);
onglet2.setText("Onglet 2");
```



BORDER : un ensemble d'onglets avec une bordure

```
TabFolder tabfolder = new TabFolder(shell, SWT.BORDER);
```



46.6.6.2. La classe TabItem

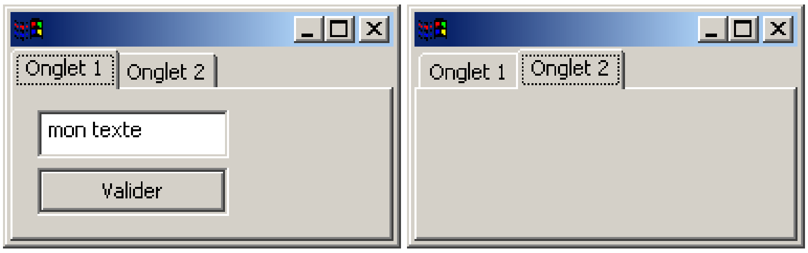
Ce contrôle est un onglet d'un ensemble d'onglets

La méthode setControl() ne permet d'insérer qu'un seul contrôle dans un onglet mais ce contrôle peut être de type Composite et regrouper différents éléments.

Exemple :

```
TabFolder tabfolder = new TabFolder(shell, SWT.NONE);
tabfolder.setSize(200,100);
TabItem onglet1 = new TabItem(tabfolder, SWT.NONE);
```

```
onglet1.setText("Onglet 1");
TabItem onglet2 = new TabItem(tabfolder, SWT.NONE);
onglet2.setText("Onglet 2");

Composite pageOnglet1 = new Composite(tabfolder, SWT.NONE);
Text text1 = new Text(pageOnglet1, SWT.BORDER);
text1.setText("mon texte");
text1.setBounds(10,10,100,25);
Button button1 = new Button(pageOnglet1, SWT.BORDER);
button1.setText("Valider");
button1.setBounds(10,40,100,25);
onglet1.setControl(pageOnglet1);
```



46.6.7. Les contrôles de type « tableau »

SWT permet la création d'un contrôle de type tableau pour afficher et sélectionner des données en mettant en oeuvre trois classes : Table, TableColumn et TableItem.

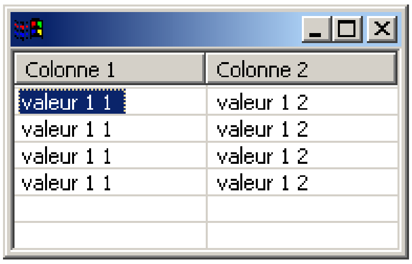
46.6.7.1. La classe Table

Ce contrôle permet d'afficher et de sélectionner des éléments sous la forme d'un tableau.

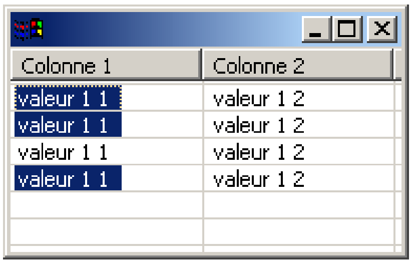
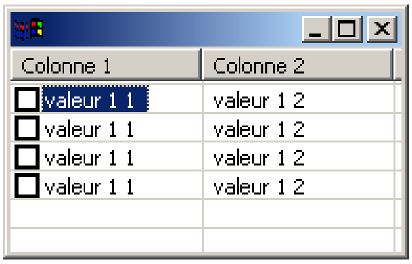
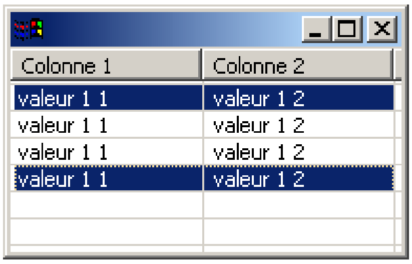
La classe Table possède plusieurs styles : BORDER, H_SCROLL, V_SCROLL, SINGLE, MULTI, CHECK, FULL_SELECTION, HIDE_SELECTION

BORDER :

```
Table table = new Table(shell, SWT.BORDER);
table.setSize(204,106);
TableColumn colonne1 = new TableColumn(table, SWT.LEFT);
colonne1.setText("Colonne 1");
colonne1.setWidth(100);
TableColumn colonne2 = new TableColumn(table, SWT.LEFT);
colonne2.setText("Colonne 2");
colonne2.setWidth(100);
table.setHeaderVisible(true);
table.setLinesVisible(true);
TableItem ligne1 = new TableItem(table,SWT.NONE);
ligne1.setText(new String[] {"valeur 1 1","valeur 1 2"});
TableItem ligne2 = new TableItem(table,SWT.NONE);
ligne2.setText(new String[] {"valeur 1 1","valeur 1 2"});
TableItem ligne3 = new TableItem(table,SWT.NONE);
ligne3.setText(new String[] {"valeur 1 1","valeur 1 2"});
TableItem ligne4 = new TableItem(table,SWT.NONE);
ligne4.setText(new String[] {"valeur 1 1","valeur 1 2"});
```



Colonne 1	Colonne 2
valeur 1 1	valeur 1 2
valeur 1 1	valeur 1 2
valeur 1 1	valeur 1 2
valeur 1 1	valeur 1 2

MULTI : permet la sélection de plusieurs éléments dans la table `Table table = new Table(shell, SWT.MULTI);`	 Colonne 1 Colonne 2 valeur 1 1 valeur 1 2 valeur 1 1 valeur 1 2 valeur 1 1 valeur 1 2 valeur 1 1 valeur 1 2
CHECK : une table avec une case à cocher pour chaque ligne `Table table = new Table(shell, SWT.CHECK);`	 Colonne 1 Colonne 2 □ valeur 1 1 valeur 1 2 □ valeur 1 1 valeur 1 2 □ valeur 1 1 valeur 1 2 □ valeur 1 1 valeur 1 2
FULL_SELECTION : la ou les lignes sélectionnées sont entièrement mises en valeur `Table table = new Table(shell, SWT.MULTI | SWT.FULL_SELECTION);`	 Colonne 1 Colonne 2 valeur 1 1 valeur 1 2 valeur 1 1 valeur 1 2 valeur 1 1 valeur 1 2 valeur 1 1 valeur 1 2
HIDE_SELECTION : seule la première colonne sélectionnée est mise en valeur	

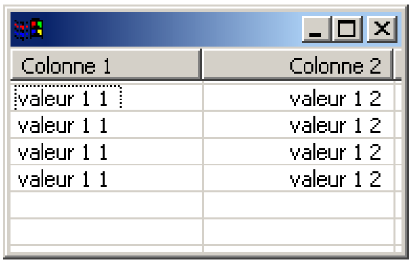
La méthode setHeaderVisible() permet de préciser si l'en-tête de la table doit être affiché ou non : par défaut il est non affiché (false).

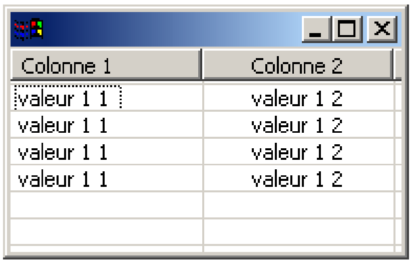
La méthode setLinesVisible() permet de préciser si les lignes de la table doivent être affichées ou non : par défaut elles ne sont pas affichées (false).

46.6.7.2. La classe TableColumn

Ce contrôle est une colonne d'un contrôle Table

La classe TableColumn possède trois styles : LEFT, RIGHT, CENTER

LEFT : alignement de la colonne sur la gauche (valeur par défaut)	
RIGHT : alignement de la colonne sur la droite `TableColumn colonne2 = new TableColumn(table, SWT.RIGHT);`	 Colonne 1 Colonne 2 valeur 1 1 valeur 1 2 valeur 1 1 valeur 1 2 valeur 1 1 valeur 1 2 valeur 1 1 valeur 1 2

CENTER : alignement centré de la colonne `TableColumn colonne2 = new TableColumn(table, SWT.CENTER);`	

Colonne 1	Colonne 2
valeur 1 1	valeur 1 2
valeur 1 1	valeur 1 2
valeur 1 1	valeur 1 2
valeur 1 1	valeur 1 2

Bizarrement, seul le style LEFT semble pouvoir s'appliquer à la première colonne de la table.

La méthode setWidth() permet de préciser la largeur de la colonne

La méthode setText() permet de préciser le libellé d'en-tête de la colonne

La méthode setResizable() permet de préciser si la colonne peut être redimensionnée ou non.

46.6.7.3. La classe TableItem

Ce contrôle est une ligne d'un contrôle Table

La classe TableItem ne possède aucun style.

Il existe plusieurs surcharges de la méthode setText() pour fournir à chaque ligne les données de ses colonnes.

Une surcharge de cette méthode permet de fournir les données sous la forme d'un tableau de chaînes de caractères.

Exemple :

```
ligne1.setText(new String[] {"valeur 1 1","valeur 1 2"});
```

Une autre surcharge de cette méthode permet de préciser le numéro de la colonne et le texte. La première colonne possède le numéro 0.

Exemple : modifier la valeur de la première cellule de la ligne

```
ligne4.setText(0 ,"valeur 2 2");
```

La méthode setCheck() permet de cocher ou non la case associée à la ligne si la table possède le style CHECK.

46.6.8. Les contrôles de type « arbre »

SWT permet la création d'un composant de type arbre en mettant en oeuvre les classes Tree et TreeItem.

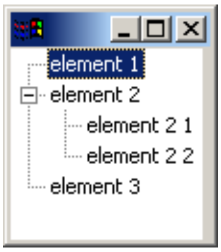
46.6.8.1. La classe Tree

Ce contrôle affiche et permet de sélectionner des données sous la forme d'une arborescence

La classe Tree possède plusieurs styles : BORDER, H_SCROLL, V_SCROLL, SINGLE, MULTI, CHECK

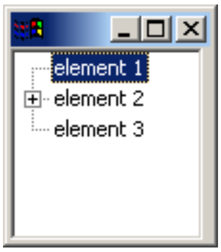
SINGLE : un arbre avec sélection unique

```
Tree tree = new Tree(shell, SWT.SINGLE);
TreeItem tree_1 = new TreeItem(tree, SWT.NONE);
tree_1.setText("element 1");
TreeItem tree_2 = new TreeItem(tree, SWT.NONE);
tree_2.setText("element 2");
TreeItem tree_2_1 = new TreeItem(tree_2, SWT.NONE);
tree_2_1.setText("element 2 1");
TreeItem tree_2_2 = new TreeItem(tree_2, SWT.NONE);
tree_2_2.setText("element 2 2");
TreeItem tree_3 = new TreeItem(tree, SWT.NONE);
tree_3.setText("element 3");
tree.setSize(100, 100);
```



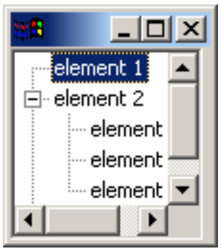
BORDER : arbre avec une bordure

```
Tree tree = new Tree(shell, SWT.SINGLE | SWT.BORDER);
```



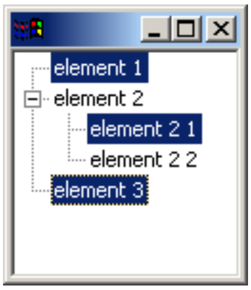
H_SCROLL et V_SCROLL : arbre avec si nécessaire une barre de défilement respectivement horizontal et vertical

```
Tree tree = new Tree(shell, SWT.SINGLE | SWT.BORDER |
  SWT.H_SCROLL | SWT.V_SCROLL);
```



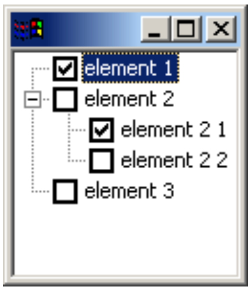
MULTI : un arbre avec sélection multiple possible

```
Tree tree = new Tree(shell, SWT.MULTI | SWT.BORDER);
```



CHECK : un arbre avec une case à cocher devant chaque élément

```
Tree tree = new Tree(shell, SWT.CHECK | SWT.BORDER);
```



46.6.8.2. La classe TreeItem

Ce contrôle est un élément d'une arborescence

Cette classe ne possède pas de style particulier.

Pour ajouter un élément racine à l'arbre, il suffit de passer l'arbre en tant qu'élément conteneur dans le constructeur.

Pour ajouter un élément fils à un élément, il suffit de passer l'élément père en tant qu'élément conteneur dans le constructeur.

Il existe un constructeur qui attend un troisième paramètre permettant de préciser la position de l'élément.

Exemple :

```
Tree tree = new Tree(shell, SWT.SINGLE);
TreeItem tree_1 = new TreeItem(tree, SWT.NONE);
tree_1.setText("element 1");
TreeItem tree_2 = new TreeItem(tree, SWT.NONE);
tree_2.setText("element 2");
TreeItem tree_2_1 = new TreeItem(tree_2, SWT.NONE);
tree_2_1.setText("element 2 1");
TreeItem tree_2_2 = new TreeItem(tree_2, SWT.NONE);
tree_2_2.setText("element 2 2");
TreeItem tree_3 = new TreeItem(tree, SWT.NONE);
tree_3.setText("element 3");
tree.setSize(100, 100);
```

46.6.9. La classe ScrollBar

Ce contrôle est une barre de défilement

La classe ScrollBar possède deux styles : HORIZONTAL, VERTICAL

46.6.10. Les contrôles pour le graphisme

SWT permet de dessiner des formes graphiques en mettant en oeuvre la classe GC et la classe Canvas.

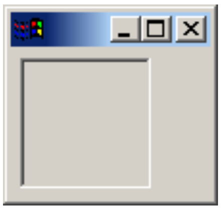
46.6.10.1. La classe Canvas

Ce contrôle est utilisé pour dessiner des formes graphiques

La classe Canvas définit plusieurs styles : BORDER, H_SCROLL, V_SCROLL, NO_BACKGROUND, NO_FOCUS, NO_MERGE_PAINTS, NO_REDRAW_RESIZE, NO_RADIO_GROUP

BORDER : une zone de dessin avec bordure

```
Canvas canvas = new Canvas(shell, SWT.BORDER);
canvas.setSize(200,200);
```



46.6.10.2. La classe GC

Cette classe encapsule un contexte graphique dans lequel il va être possible de dessiner des formes.

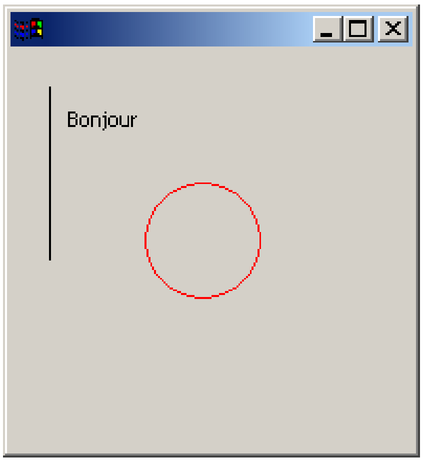
Pour réaliser ces opérations, la classe GC propose de nombreuses méthodes.

Attention : il est important d'appeler la méthode open() de la fenêtre avant de réaliser des opérations de dessin sur le contexte.

Ne pas oublier le libérer les ressources allouées à la classe GC en utilisant la méthode dispose() si l'objet de GC est explicitement instancié dans le code.

Exemple :

```
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.graphics.*;

public class TestSWT21 {
  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);
    shell.setSize(420,420);

    Canvas canvas = new Canvas(shell, SWT.NONE);
    canvas.setSize(200,200);
    canvas.setLocation(10,10);
    shell.pack();
    shell.open();

    GC gc = new GC(canvas);
    gc.drawText("Bonjour",20,20);
    gc.drawLine(10,10,10,100);
    gc.setForeground(display.getSystemColor(SWT.COLOR_RED));
    gc.drawOval(60,60,60,60);
    gc.dispose();

    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }
    display.dispose();
  }
}
```



Dans cette exemple, le dessin est réalisé une seule fois au démarrage, il n'est donc pas redessiné si nécessaire (fenêtre partiellement ou complètement masquée, redimensionnement, ...). Pour résoudre ce problème, il faut mettre les opérations de dessin en réponse à un événement de type PaintListener.

46.6.10.3. La classe Color

Cette classe encapsule une couleur définie dans le système graphique.

Elle possède deux constructeurs qui attendent en paramètre l'objet de type Display et soit un objet de type RGB, soit trois entiers représentant les valeurs des couleurs rouge, vert et bleu.

La classe RGB encapsule simplement les trois entiers représentant les valeurs des couleurs rouge, vert et bleu.

Exemple :

```
Color couleur = new Color(display,155,0,0);
Color couleur = new Color(display, new RGB(155,0,0));
```

Remarque : il ne faut pas oublier d'utiliser la méthode dispose() pour libérer les ressources du système allouées à cet objet une fois que celui-ci n'est plus utilisé.

Exemple :

```
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.graphics.*;

public class TestSWT22 {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);

    shell.setSize(200, 200);
    Color couleur = new Color(display,155,0,0);
    shell.setBackground(couleur);
    shell.open();

   while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }

    couleur.dispose();
    display.dispose();
  }
}
```

46.6.10.4. La classe Font

Cette classe encapsule une police de caractères définie dans le système graphique.

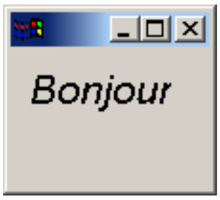
La classe Font peut utiliser plusieurs styles : NORMAL, BOLD et ITALIC

Il existe plusieurs constructeurs dont le plus simple à utiliser nécessite en paramètre l'objet display, le nom de la police (celle-ci doit être présente sur le système), la taille et le style.

Remarque : il ne faut pas oublier d'utiliser la méthode dispose() pour libérer les ressources du système allouées à cet objet une fois que celui-ci n'est plus utilisé.

Exemple :

```
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.*;

public class TestSWT23 {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);

    Font font = new Font(display, "Arial", 16, SWT.ITALIC);
    Label label = new Label(shell, SWT.NONE);
    label.setFont(font);
    label.setText("Bonjour");
    label.setLocation(10, 10);
    label.pack();
    shell.setSize(100, 100);
    shell.open();

    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }

    font.dispose();
    display.dispose();
  }
}
```



46.6.10.5. La classe Image

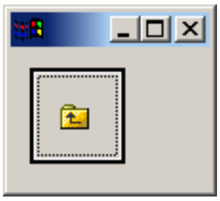
Cette classe encapsule une image au format BMP, ICO, GIF, JPEG ou PNG.

La classe Image possède plusieurs constructeurs dont le plus simple à utiliser est celui nécessitant en paramètres l'objet Display et une chaîne de caractères contenant le chemin vers le fichier de l'image

Remarque : il ne faut pas oublier d'utiliser la méthode dispose() pour libérer les ressources du système allouées à cet objet une fois que celui-ci n'est plus utilisé.

Exemple :

```
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.*;

public class TestSWT24 {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);

    Image image = new Image(display, "btn1.bmp");
    Button bouton = new Button(shell, SWT.FLAT);
    bouton.setImage(image);
    bouton.setBounds(10, 10, 50, 50);
    shell.setSize(100, 100);
    shell.open();

    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }

    image.dispose();
    display.dispose();
  }
}
```



Si l'application doit être packagée dans un fichier jar, incluant les images utiles, il faut utiliser la méthode getResourceAsStream() du classloader pour charger l'image.

Exemple :

```
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.*;
import java.io.*;

public class TestSWT25 {
  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);

    InputStream is = TestSWT25.class.getResourceAsStream("btn1.bmp");
    Image image = new Image(display, is);
    Button bouton = new Button(shell, SWT.FLAT);
    bouton.setImage(image);
    bouton.setBounds(10, 10, 50, 50);
    shell.setSize(100, 100);
    shell.open();

    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }

    image.dispose();
    display.dispose();
  }
}
```

46.7. Les conteneurs

Ce type de composant permet de contenir d'autres contrôles.

46.7.1. Les conteneurs de base

SWT propose deux contrôles de ce type : Composite et Group.

46.7.1.1. La classe Composite

Ce contrôle est un conteneur pour d'autres contrôles.

Ce contrôle possède les styles particuliers suivants : BORDER, H_SCROLL et V_SCROLL

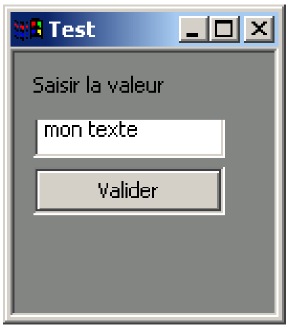
BORDER : permet la présence d'une bordure autour du composant `Composite composite = new Composite(shell, SWT.BORDER);`	
H_SCROLL : permet la présence d'une barre de défilement horizontal `Composite composite = new Composite(shell, SWT.H_SCROLL);`	
V_SCROLL : permet la présence d'une barre de défilement vertical `Composite composite = new Composite(shell, SWT.V_SCROLL);`	

Les contrôles sont ajoutés au contrôle Composite de la même façon que dans un objet de type Shell en précisant simplement que le conteneur est l'objet de type Composite.

La position indiquée pour les contrôles inclus dans le Composite est relative à l'objet Composite.

Exemple complet :

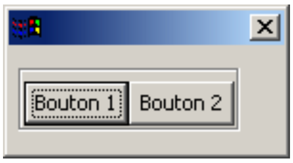
```
import org.eclipse.swt.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.widgets.*;

public class TestSWT2 {
  public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setText("Test");

    Composite composite = new Composite(shell, SWT.BORDER);
    Color couleur = new Color(display,131,133,131);
    composite.setBackground(couleur);
    Label label = new Label(composite, SWT.NONE);
    label.setBackground(couleur);
    label.setText("Saisir la valeur");
    label.setBounds(10, 10, 100, 25);
    Text text = new Text(composite, SWT.BORDER);
    text.setText("mon texte");
    text.setBounds(10, 30, 100, 25);
    Button button = new Button(composite, SWT.BORDER);
    button.setText("Valider");
    button.setBounds(10, 60, 100, 25);
    composite.setSize(140,140);

    shell.pack();
    shell.open();
    while (!shell.isDisposed())
      if (!display.readAndDispatch())
        display.sleep();

    couleur.dispose();
    display.dispose();
  }
}
```

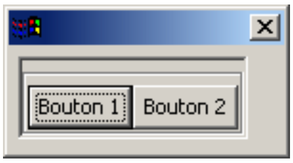


46.7.1.2. La classe Group

Ce contrôle permet de regrouper d'autres contrôles en les entourant d'une bordure et éventuellement d'un libellé.

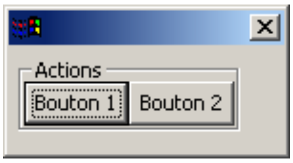
La classe Group possède plusieurs styles : BORDER, SHADOW_ETCHED_IN, SHADOW_ETCHED_OUT, SHADOW_IN, SHADOW_OUT, SHADOW_NONE

NONE : un cadre simple
`Group group = new Group(shell, SWT.NONE);`
`group.setLayout (new FillLayout ());`
`button button1 = new Button(group, SWT.NONE);`
`button1.setText("Bouton 1");`
`Button button2 = new Button(group, SWT.NONE);`
`button2.setText("Bouton 2");`
	

BORDER : un cadre simple avec une bordure `Group group = new Group(shell, SWT.BORDER);`	

La méthode setText() permet de préciser un titre affiché en haut à gauche du cadre.

```
Group group = new Group(shell, SWT.NONE);
group.setLayout (new FillLayout ());
group.setText ("Actions");
Button button1 = new Button(group, SWT.NONE);
button1.setText("Bouton 1");
Button button2 = new Button(group, SWT.NONE);
button2.setText("Bouton 2");
```



46.7.2. Les contrôles de type « barre d'outils »

SWT permet de créer des barres d'outils fixes ou flottantes.

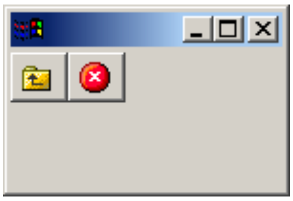
46.7.2.1. La classe ToolBar

Ce contrôle est une barre d'outils

La classe ToolBar possède plusieurs styles : BORDER, FLAT, WRAP, RIGHT, SHADOW_OUT HORIZONTAL, VERTICAL

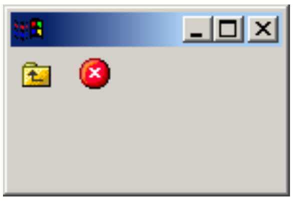
HORIZONTAL : une barre d'outils horizontale (style par défaut)

```
shell.setSize(150, 100);
ToolBar toolbar = new ToolBar(shell, SWT.HORIZONTAL);
toolbar.setSize(shell.getSize().x, 35);
toolbar.setLocation(0, 0);

Image imageBtn1 = new Image(display, "btn1.bmp");
ToolItem btn1 = new ToolItem(toolbar, SWT.PUSH);
btn1.setImage(imageBtn1);

Image imageBtn2 = new Image(display, "btn2.bmp");
ToolItem btn2 = new ToolItem(toolbar, SWT.PUSH);
btn2.setImage(imageBtn2);

shell.open();
while (!shell.isDisposed())
  if (!display.readAndDispatch())
    display.sleep();

imageBtn1.dispose();
imageBtn2.dispose();
display.dispose();
```



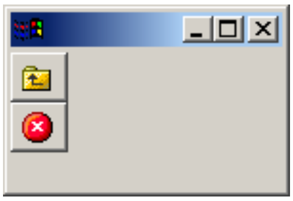
FLAT : une barre d'outils sans effet 3D

```
ToolBar toolbar = new ToolBar(shell, SWT.FLAT);
```



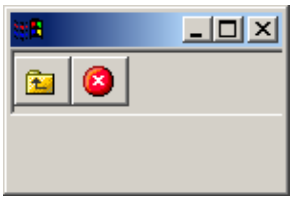
VERTICAL : une barre d'outils verticale

```
ToolBar toolbar = new ToolBar(shell, SWT.VERTICAL);
toolbar.setSize(35, shell.getSize().y);
```



BORDER : une barre d'outils avec une bordure

```
ToolBar toolbar = new ToolBar(shell, SWT.BORDER);
```



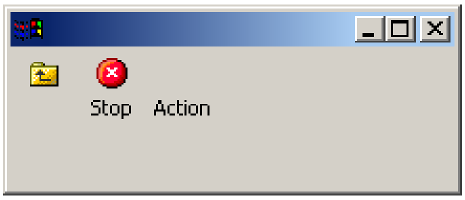
46.7.2.2. La classe ToolItem

Ce contrôle est un élément d'une barre d'outils

La classe ToolItem possède plusieurs styles : PUSH, CHECK, RADIO, SEPARATOR, DROP_DOWN

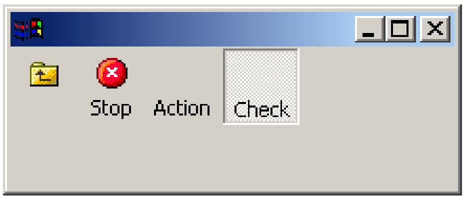
PUSH : un bouton simple

```
shell.setSize(240, 100);
ToolBar toolbar = new ToolBar(shell, SWT.FLAT);
toolbar.setSize(shell.getSize().x, 40);
toolbar.setLocation(0, 0);

Image imageBtn1 = new Image(display, "btn1.bmp");
ToolItem btn1 = new ToolItem(toolbar, SWT.PUSH);
btn1.setImage(imageBtn1);

Image imageBtn2 = new Image(display, "btn2.bmp");
ToolItem btn2 = new ToolItem(toolbar, SWT.PUSH);
btn2.setImage(imageBtn2);
btn2.setText("Stop");

ToolItem btn3= new ToolItem(toolbar, SWT.PUSH);
btn3.setText("Action");
shell.open();

while (!shell.isDisposed())
  if (!display.readAndDispatch())
    display.sleep();

imageBtn1.dispose();
imageBtn2.dispose();
display.dispose();
```



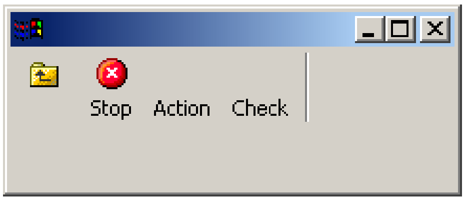
CHECK : un bouton qui peut conserver son état enfoncé

```
ToolItem btn4= new ToolItem(toolbar, SWT.CHECK);
btn4.setText("Check");
```

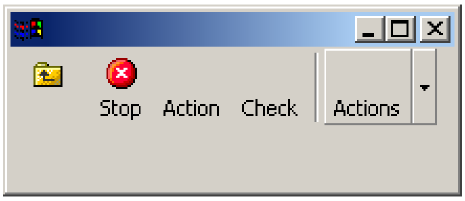


SEPARATOR : un séparateur

```
ToolItem btn5= new ToolItem(toolbar, SWT.SEPARATOR);
```

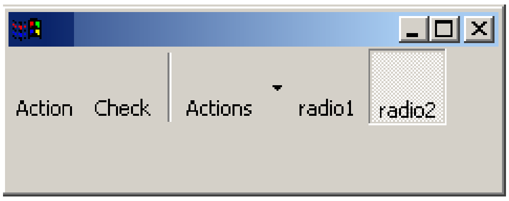


DROP_DOWN : un bouton avec une petite flèche vers le bas



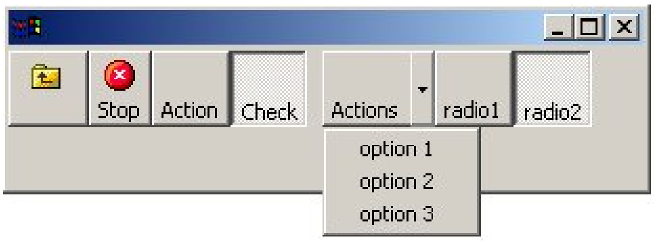
RADIO : un bouton dont un seul d'un même ensemble peut être sélectionné (un ensemble est défini par des boutons de type radio qui sont adjacents)

```
ToolItem btn7= new ToolItem(toolbar, SWT.RADIO);
btn7.setText("radio1");
ToolItem btn8= new ToolItem(toolbar, SWT.RADIO);
btn8.setText("radio2");
```



Exemple :

```
shell.setSize(340, 100);
final ToolBar toolbar = new ToolBar(shell, SWT.HORIZONTAL);
toolbar.setSize(shell.getSize().x, 45);
toolbar.setLocation(0, 0);

Image imageBtn1 = new Image(display, "btn1.bmp");
ToolItem btn1 = new ToolItem(toolbar, SWT.PUSH);
btn1.setImage(imageBtn1);

Image imageBtn2 = new Image(display, "btn2.bmp");
ToolItem btn2 = new ToolItem(toolbar, SWT.PUSH);
btn2.setImage(imageBtn2);
btn2.setText("Stop");

ToolItem btn3 = new ToolItem(toolbar, SWT.PUSH);
btn3.setText("Action");

ToolItem btn4 = new ToolItem(toolbar, SWT.CHECK);
btn4.setText("Check");

ToolItem btn5 = new ToolItem(toolbar, SWT.SEPARATOR);

final ToolItem btn6 = new ToolItem(toolbar, SWT.DROP_DOWN);
btn6.setText("Actions");

final Menu menu = new Menu(shell, SWT.POP_UP);
MenuItem menu1 = new MenuItem(menu, SWT.PUSH);
menu1.setText("option 1");
MenuItem menu2 = new MenuItem(menu, SWT.PUSH);
menu2.setText("option 2");
MenuItem menu3 = new MenuItem(menu, SWT.PUSH);
menu3.setText("option 3");

btn6.addListener(SWT.Selection, new Listener() {
  public void handleEvent(Event event) {
    if (event.detail == SWT.ARROW) {
      Rectangle rect = btn6.getBounds();
      Point pt = new Point(rect.x, rect.y + rect.height);
      pt = toolbar.toDisplay(pt);
      menu.setLocation(pt.x, pt.y);
      menu.setVisible(true);
    }
  }
});

ToolItem btn7 = new ToolItem(toolbar, SWT.RADIO);
btn7.setText("radio1");

ToolItem btn8 = new ToolItem(toolbar, SWT.RADIO);
btn8.setText("radio2");

shell.open();
while (!shell.isDisposed())
  if (!display.readAndDispatch())
    display.sleep();

imageBtn1.dispose();
imageBtn2.dispose();
display.dispose();
```



46.7.2.3. Les classes CoolBar et Cooltem

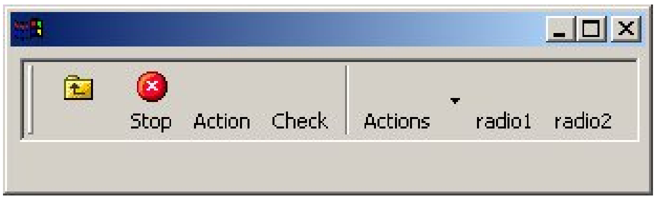
La classe CoolBar est une barre d'outils repositionnable. Ce contrôle doit être utilisé avec un ou plusieurs contrôles CoolItem qui représentent les éléments constitutifs de la CoolBar.

Le plus simple est d'associer une barre d'outils de type ToolBar à un de ses éléments en utilisant la méthode setControl() de la classe CoolItem.

Exemple : utilisation de la barre d'outils définie dans la section précédente

```
shell.setLayout(new GridLayout());

shell.setSize(340, 100);
CoolBar coolbar = new CoolBar(shell, SWT.BORDER);
final ToolBar toolbar = new ToolBar(coolbar, SWT.FLAT);

Image imageBtn1 = new Image(display, "btn1.bmp");
ToolItem btn1 = new ToolItem(toolbar, SWT.PUSH);
btn1.setImage(imageBtn1);

Image imageBtn2 = new Image(display, "btn2.bmp");
ToolItem btn2 = new ToolItem(toolbar, SWT.PUSH);
btn2.setImage(imageBtn2);
btn2.setText("Stop");

ToolItem btn3 = new ToolItem(toolbar,SWT.PUSH);
btn3.setText("Action");

ToolItem btn4 = new ToolItem(toolbar,SWT.CHECK);
btn4.setText("Check");

ToolItem btn5 = new ToolItem(toolbar, SWT.SEPARATOR);

final ToolItem btn6 = new ToolItem(toolbar, SWT.DROP_DOWN);
btn6.setText("Actions");

final Menu menu = new Menu(shell, SWT.POP_UP);
MenuItem menu1 = new MenuItem(menu, SWT.PUSH);
menu1.setText("option 1");
MenuItem menu2 = new MenuItem(menu, SWT.PUSH);
menu2.setText("option2");
MenuItem menu3 = new MenuItem(menu, SWT.PUSH);
menu3.setText("option3");

btn6.addListener(SWT.Selection, new Listener() {
  public void handleEvent(Event event) {
    if (event.detail == SWT.ARROW) {
      Rectangle rect = btn6.getBounds();
      Point pt = new Point(rect.x, rect.y + rect.height);
      pt = toolbar.toDisplay(pt);
      menu.setLocation(pt.x, pt.y);
      menu.setVisible(true);
    }
  }
});

ToolItem btn7 = new ToolItem(toolbar, SWT.RADIO);
btn7.setText("radio1");

ToolItem btn8 = new ToolItem(toolbar, SWT.RADIO);
btn8.setText("radio2");

CoolItem coolItem = new CoolItem(coolbar, SWT.NONE);
coolItem.setControl(toolbar);
Point size = toolbar.computeSize(SWT.DEFAULT, SWT.DEFAULT);
coolItem.setPreferredSize(coolItem.computeSize(size.x, size.y));

shell.open();

while (!shell.isDisposed())
  if (!display.readAndDispatch())
    display.sleep();
```

```
imageBtn1.dispose();
imageBtn2.dispose();
```



La classe CoolItem possède une méthode setLocked() qui attend un booléen en paramètre précisant si le contrôle peut être déplacé ou non. Cette méthode doit être appelée lors d'un clic sur un bouton de la barre pour empêcher le déplacement de celle-ci.

46.8. La gestion des erreurs

Lors de l'utilisation de l'API SWT, des exceptions de trois types peuvent être levées :

- IllegalArgumentException : un argument fourni à une méthode est invalide
- SWTException : cette exception est levée lors d'une erreur non fatale. Le code de l'erreur et le message de l'exception permettent d'obtenir des précisions sur l'exception
- SWTError : cette exception est levée lors d'une erreur fatale

46.9. Le positionnement des contrôles

46.9.1. Le positionnement absolu

Dans ce mode, il faut préciser pour chaque composant, sa position et sa taille. L'inconvénient de ce mode de positionnement est qu'il réagit très mal à un changement de la taille du conteneur des composants.

46.9.2. Le positionnement relatif avec les LayoutManager

SWT propose un certain nombre de gestionnaires de positionnement de contrôles (layout manager). Ceux-ci sont regroupés dans le package org.eclipse.swt.layout.

Le grand avantage de ce mode de positionnement est de laisser au LayoutManager utilisé le soin de positionner et de dimensionner chaque composant en fonction de ses règles et des paramètres qui lui sont fournis.

SWT définit quatre gestionnaires de positionnement :

- RowLayout pour un arrangement simple des composants au file de la page
- FillLayout pour les composants également répartis sur une colonne ou une rangée simple
- GridLayout pour une disposition des composants sur une grille rectangulaire
- FormLayout pour un positionnement plus précis mais aussi plus compliqué des composants.

Pour personnaliser finement l'arrangement des composants, des informations complémentaires peuvent être associées à chacun d'eux en utilisant un objet dédié du type RowData, GridData ou FormData respectivement pour les gestionnaires de positionnement RowLayout, GridLayout et FormLayout.

46.9.2.1. FillLayout

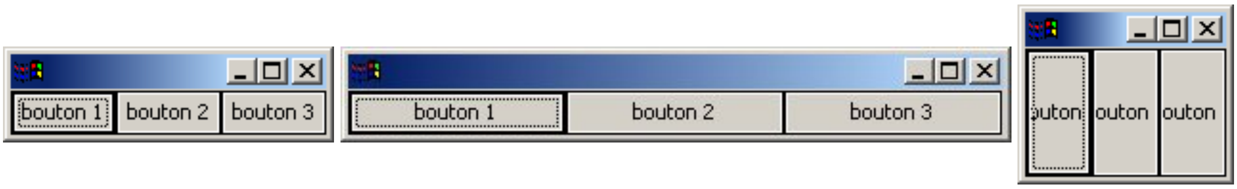
Le FillLayout est le gestionnaire de positionnement le plus simple : il organise les composants dans une colonne ou une rangée. L'espace entre les composants est calculé automatiquement par la classe FillLayout.

La classe FillLayout peut utiliser deux styles : SWT.HORIZONTAL (par défaut) et SWT.VERTICAL pour préciser le mode d'alignement

Exemple :

```
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.*;

public class TestSWT27 {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);

    shell.setLayout(new RowLayout());
    Button bouton1 = new Button(shell, SWT.FLAT);
    bouton1.setText("bouton 1");
    Button bouton2 = new Button(shell, SWT.FLAT);
    bouton2.setText("bouton 2");
    Button bouton3 = new Button(shell, SWT.FLAT);
    bouton3.setText("bouton 3");

    shell.pack();
    shell.open();

    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }

    display.dispose();
  }
}
```

Voici différents aperçus en cas de modification de la taille de la fenêtre.

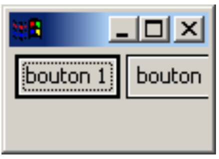
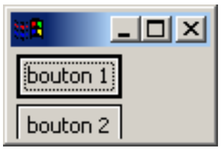
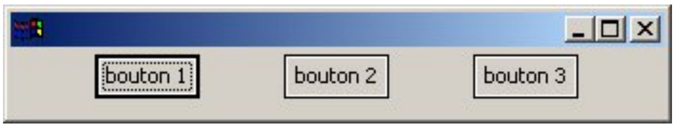
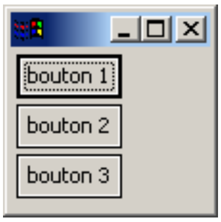


Il n'est pas possible de mettre un espace entre le bord du conteneur et les composants avec ce gestionnaire. Il n'est pas non plus possible pour ce gestionnaire de mettre des composants sur plusieurs colonnes ou rangées.

46.9.2.2. RowLayout

Ce gestionnaire propose d'arranger les composants en rangées ou en colonnes. Il possède des paramètres permettant de préciser une marge, un espace, une rupture et une compression.

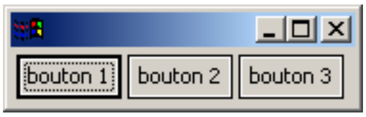
Propriété	Valeur par défaut	Rôle
wrap	true	demande de faire une rupture dans la rangée s'il n'y a plus de place

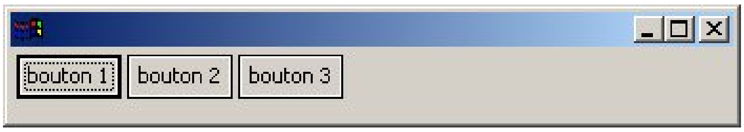
		false :  true : 
pack	true	demande à chaque composant de prendre sa taille préférée
justify	false	justification des composants 
type	SWT.HORIZONTAL	type de mise en forme SWT.HORIZONTAL ou SWT.VERTICAL SWT.VERTICAL : 
marginLeft	3	taille en pixels de la marge gauche
marginTop	3	taille en pixels de la marge haute
marginRight	3	taille en pixels de la marge droite
marginBottom	3	taille en pixels de la marge basse
spacing	3	taille en pixels entre deux cellules

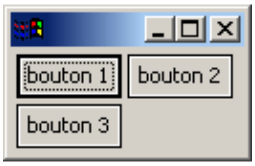
Exemple :

```
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.*;

public class TestSWT27 {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);

    RowLayout rowlayout = new RowLayout();
    shell.setLayout(rowlayout);
    Button bouton1 = new Button(shell, SWT.FLAT);
    bouton1.setText("bouton 1");

    Button bouton2 = new Button(shell, SWT.FLAT);
    bouton2.setText("bouton 2");

    Button bouton3 = new Button(shell, SWT.FLAT);
    bouton3.setText("bouton 3");

    shell.pack();
    shell.open();

    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }
    display.dispose();
  }
}
```

Voici différents aperçus en cas de modification de la taille de la fenêtre.







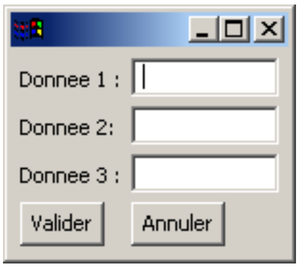
46.9.2.3. GridLayout

Ce gestionnaire permet d'arranger les composants dans une grille et possède plusieurs propriétés :

Propriété	Valeur par défaut	Rôle
horizontalSpacing	5	préciser l'espace horizontal entre chaque cellule
makeColumnsEqualWidth	false	donner à toutes les colonnes de la grille la même largeur
marginHeight	5	préciser la hauteur de la marge
marginWidth	5	préciser la largeur de la marge
numColumns	1	préciser le nombre de colonnes de la grille
verticalSpacing	5	préciser l'espace vertical entre cellules

Exemple :

```
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.*;

public class TestSWT28 {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);

    GridLayout gridLayout = new GridLayout();
    gridLayout.numColumns = 2;
    shell.setLayout(gridLayout);

    Label label1 = new Label(shell, SWT.NONE);
    label1.setText("Donnee 1 :");
    Text text1 = new Text(shell, SWT.BORDER);
    text1.setSize(200, 10);

    Label label2 = new Label(shell, SWT.NONE);
    label2.setText("Donnee 2 :");
    Text text2 = new Text(shell, SWT.BORDER);
    text2.setSize(200, 10);

    Label label3 = new Label(shell, SWT.NONE);
    label3.setText("Donnee 3 :");
    Text text3 = new Text(shell, SWT.BORDER);
    text3.setSize(200, 10);

    Button button1 = new Button(shell, SWT.NONE);
    button1.setText("Valider");

    Button button2 = new Button(shell, SWT.NONE);
    button2.setText("Annuler");

    shell.pack();
    shell.open();

    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }

    display.dispose();
  }
}
```



Les paramètres liés à un composant d'une cellule particulière de la grille peuvent être précisés grâce à un objet de type GridData. Ces paramètres précisent le comportement du composant en cas de redimensionnement.

Il existe deux façons de créer un objet de type GridData :

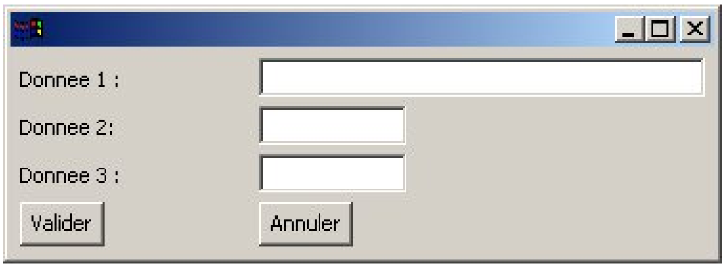
- instancier un objet de type GridData avec son constructeur sans paramètre et initialiser les propriétés en utilisant les setters appropriés.
- instancier un objet de type GridData avec son constructeur attendant un style en paramètre

La méthode setLayoutData() permet d'associer un objet GridData à un composant.

Attention : il ne faut pas appliquer le même objet GridData à plusieurs composants (la méthode setLayoutData() doit recevoir des objets GridData différents.).

Exemple :

```
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.*;

public class TestSWT28 {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);

    GridLayout gridLayout = new GridLayout();
    gridLayout.numColumns = 2;
    shell.setLayout(gridLayout);

    Label label1 = new Label(shell, SWT.NONE);
    label1.setText("Donnee 1 :");
    Text text1 = new Text(shell, SWT.BORDER);
    text1.setSize(200, 10);

    Label label2 = new Label(shell, SWT.NONE);
    label2.setText("Donnee 2:");
    Text text2 = new Text(shell, SWT.BORDER);
    text2.setSize(200, 10);

    Label label3 = new Label(shell, SWT.NONE);
    label3.setText("Donnee 3 :");
    Text text3 = new Text(shell, SWT.BORDER);
    text3.setSize(200, 10);

    Button button1 = new Button(shell, SWT.NONE);

    button1.setText("Valider");

    Button button2 = new Button(shell, SWT.NONE);
    button2.setText("Annuler");
    GridData data = new GridData();
    data.widthHint = 120;
    label1.setLayoutData(data);

    data = new GridData();
    data.widthHint = 220;
    text1.setLayoutData(data);

    shell.pack();
    shell.open();

    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }

    display.dispose();
  }
}
```



46.9.2.4. FormLayout

Ce gestionnaire possède deux propriétés :

Propriété	Valeur par défaut	Rôle
marginHeight	0	préciser la hauteur de la marge
marginWidth	0	préciser la largeur de la marge

Ce gestionnaire impose d'associer à chaque composant un objet de type FormData qui va préciser les informations de positionnement et de comportement du composant.

46.10. La gestion des événements

La gestion des événements avec SWT est très similaire à celle proposée par l'API Swing car elle repose sur les Listeners. Ces Listeners doivent être ajoutés au contrôle en fonction des événements qu'ils doivent traiter.

Dès lors, lorsque l'événement est émis suite à une action de l'utilisateur, la méthode correspondante du Listener enregistré est exécutée.

Dans la pratique, les Listeners sont des interfaces qu'il faut faire implémenter par une classe selon les besoins. Cette implémentation définira donc des méthodes qui contiennent les traitements à exécuter pour un événement précis. Un ou plusieurs paramètres fournis à ces méthodes permettent d'obtenir des informations plus précises sur l'événement.

Il suffit ensuite d'enregistrer le Listener auprès du contrôle en utilisant la méthode addXXXListener() du contrôle où XXX représente le type du Listener.

Exemple : pour le traitement d'un clic d'un bouton

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.widgets.*;

public class TestSWT3 {

  public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setText("Test");

    Button button = new Button(shell, SWT.NONE);
    button.setText("Valider");
    button.setBounds(1, 1, 100, 25);

    button.addSelectionListener(new SelectionListener() {
      public void widgetSelected(SelectionEvent arg0) {
                        System.out.println("Appui sur le bouton");
      }
      public void widgetDefaultSelected(SelectionEvent arg0) {
      }
    });

    shell.pack();
    shell.open();

    while (!shell.isDisposed())
      if (!display.readAndDispatch())
        display.sleep();

    display.dispose();
  }
}
```

Comme avec Swing, SWT propose un ensemble de classes de type Adapter qui sont des classes implémentant les interfaces Listener avec des méthodes vides. Pour les utiliser, il suffit de définir une classe fille qui hérite de la classe de type Adapter adéquate et de redéfinir la ou les méthodes utiles.

Exemple :

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.widgets.*;

public class TestSWT4 {

  public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setText("Test");

    Button button = new Button(shell, SWT.NONE);
    button.setText("Valider");
    button.setBounds(1, 1, 100, 25);

    button.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(SelectionEvent arg0) {
        System.out.println("Appui sur le bouton");
      }
    });

    shell.pack();
    shell.open();

    while (!shell.isDisposed())
      if (!display.readAndDispatch())
        display.sleep();

    display.dispose();
  }
}
```

SWT définit plusieurs Listeners :

- SelectionListener : événement lié à la sélection d'un élément du contrôle
- KeyListener : événement lié au clavier
- MouseListener : événement lié aux clics de la souris
- MouseMoveListener : événement lié au mouvement de la souris
- MouseTrackListener : événement lié à la souris en relation au contrôle (entrée, sortie, passage au dessus)
- ModifyListerner : événement lié à la modification du contenu d'un contrôle de saisie de texte
- VerifyListener : événement lié à la vérification avant modification du contenu d'un contrôle de saisie de texte
- FocusListener : événement lié à la prise ou à la perte du focus
- TraverseListener : événement lié à la traversée d'un contrôle au moyen de la touche tab ou des flèches
- PaintListener : événement lié à la nécessité de redessiner le composant

46.10.1. L'interface KeyListener

Cette interface définit deux méthodes keyPressed() et keyReleased() relatives à des événements émis par le clavier, respectivement lors de l'enfoncement d'une touche et la remontée d'une touche du clavier.

Ces deux méthodes possèdent un objet de type KeyEvent qui contient des informations sur l'événement grâce à trois attributs :

character	contient le caractère de la touche concernée
keyCode	contient le code de la touche concernée SWT défini des valeurs pour des touches particulières, par exemple SWT.ALT, SWT.ARROW_DOWN, SWT_ARROW_LEFT, SWT.CTRL, SWT.CR, SWT.F1, SWT.F2, ...
stateMask	contient l'état du clavier au moment de l'émission de l'événement, ce qui permet de savoir si l'une des touches Shift, Alt ou Ctrl était enfoncée au moment de l'événement. Il suffit pour cela de comparer la valeur avec SWT.SHIFT, SWT.ALT ou SWT.CTRL.

SWT définit une classe KeyAdapter qui implémente l'interface KeyListener avec des méthodes vides.

Exemple :

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.widgets.*;

public class TestSWT5 {

  public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setText("Test");

    shell.addKeyListener(new KeyAdapter() {
      public void keyReleased(KeyEvent e) {
        String res = "";
        switch (e.character) {
          case SWT.CR :
            res = "Touche Entree";
            break;
          case SWT.DEL :
            res = "Touche Supp";
            break;
          case SWT.ESC :
            res = "Touche Echap";
            break;
          default :
            res = res + e.character;
        }

        System.out.println(res);

      }
    });

    shell.pack();
    shell.open();

    while (!shell.isDisposed())
      if (!display.readAndDispatch())
        display.sleep();

    display.dispose();
  }
}
```

Exemple : utilisation de la propriété stateMask

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.widgets.*;

public class TestSWT6 {

  public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setText("Test");

    shell.addKeyListener(new KeyAdapter() {
      public void keyReleased(KeyEvent e) {
        String res = "";
        if (e.keyCode == SWT.SHIFT) {
            res = "touche shift";
        } else {
          if ((e.stateMask & SWT.SHIFT) != 0) {
            res = "" + e.character + " + touche shift";
          }
        }
        System.out.println(res);
      }
    });

    shell.pack();
    shell.open();

    while (!shell.isDisposed())
      if (!display.readAndDispatch())
        display.sleep();

    display.dispose();
  }
}
```

46.10.2. L'interface MouseListener

Cette interface définit trois méthodes mouseDown(), mouseUp() et mouseDoubleClick() relatives à des événements émis par un clic sur la souris, respectivement l'enfoncement d'un bouton et le relâchement d'un bouton ou le double clic sur un bouton de la souris.

Ces trois méthodes possèdent un objet de type MouseEvent qui contient des informations sur l'événement grâce à quatre attributs :

button	contient le numéro du bouton utilisé (de 1 à 3). Attention, la valeur est dépendante du système utilisé, par exemple sous Windows avec une souris à molette possédant deux boutons, l'appui sur le bouton de droite

	renvoie 3
stateMask	contient l'état du clavier au moment de l'émission de l'événement, ce qui permet de savoir par exemple si la touche Alt ou Shift ou Ctrl est enfoncée au moment de l'événement en effectuant un test sur la valeur avec SWT.ALT ou SWT.CTRL ou SWT.SHIFT
x	contient la coordonnée x du pointeur de la souris par rapport au contrôle lors de l'émission de l'événement
y	contient la coordonnée y du pointeur de la souris par rapport au contrôle lors de l'émission de l'événement

SWT définit une classe MouseAdapter qui implémente l'interface MouseListener avec des méthodes vides.

Exemple :

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.widgets.*;

public class TestSWT7 {

  public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setText("Test");


    shell.addMouseListener(new MouseAdapter() {
      public void mouseDown(MouseEvent e) {
        String res = "";
          res = "bouton " + e.button + ", x = " + e.x + ", y = " + e.y;
          if ((e.stateMask & SWT.SHIFT) != 0) {
            res = res + " + touche shift";
          }
        System.out.println(res);
      }
    });

    shell.pack();
    shell.open();

    while (!shell.isDisposed())
      if (!display.readAndDispatch())
        display.sleep();

    display.dispose();
  }
}
```

46.10.3. L'interface MouseMoveListener

Cette interface définit une seule méthode mouseMove() relative aux événements émis lors du déplacement de la souris au-dessus d'un contrôle.

Cette méthode possède un objet de type MouseEvent qui contient des informations sur l'événement grâce à quatre attributs.

La mise en oeuvre est similaire de l'interface MouseListener.

Exemple :

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.widgets.*;

public class TestSWT8 {

  public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setText("Test");

    shell.addMouseMoveListener(new MouseMoveListener() {
      public void mouseMove(MouseEvent e) {
        String res = "";
        if ((e.x < 20) & (e.y < 20)) {

          res = "x = " + e.x + ", y = " + e.y;
          if ((e.stateMask & SWT.SHIFT) != 0) {
            res = res + " + touche shift";
          }
          System.out.println(res);
        }
      }
    });

    shell.pack();
    shell.open();

    while (!shell.isDisposed())
      if (!display.readAndDispatch())
        display.sleep();

    display.dispose();
  }
}
```

46.10.4. L'interface MouseTrackListener

Cette interface définit trois méthodes mouseEnter(), mouseExit() et mouseHover() relatives à des événements émis respectivement par l'entrée de la souris sur la zone d'un composant, la sortie et le passage au-dessus de la zone d'un composant.

Ces trois méthodes possèdent un objet de type MouseEvent.

SWT définit une classe MouseTrackAdapter qui implémente l'interface MouseListener avec des méthodes vides.

Exemple : changement de la couleur de fond de la fenêtre

```
import org.eclipse.swt.events.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.graphics.*;

public class TestSWT9 {

  public static void main(String[] args) {
    Display display = new Display();
    final Shell shell = new Shell(display);

    final Color couleur1 = new Color(display,155,130,0);
    final Color couleur2 = new Color(display,130,130,130);
    shell.setText("Test");

    shell.addMouseTrackListener(new MouseTrackAdapter() {
      public void mouseEnter(MouseEvent e) {
            shell.setBackground(couleur1);
      }
      public void mouseExit(MouseEvent e) {
            shell.setBackground(couleur2);
      }
    });

    shell.pack();
    shell.open();

    while (!shell.isDisposed())
      if (!display.readAndDispatch())
        display.sleep();
    display.dispose();
  }
}
```

46.10.5. L'interface ModifyListener

Cette interface définit une seule méthode modifyText() relative à un événement émis lors de la modification du contenu d'un contrôle de saisie de texte.

Cette méthode possède un objet de type ModifyEvent.

Exemple :

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.widgets.*;

public class TestSWT10 {

  public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setText("Test");

    Text text = new Text(shell, SWT.BORDER);
    text.setText("mon texte");
    text.setSize(100, 25);

   text.addModifyListener(new ModifyListener() {
     public void modifyText(ModifyEvent e) {
       System.out.println("nouvelle valeur = " + ((Text)e.widget).getText());
      }
    });

    shell.pack();
    shell.open();

    while (!shell.isDisposed())
      if (!display.readAndDispatch())
        display.sleep();

    display.dispose();
  }
}
```

46.10.6. L'interface VerifyText()

Cette interface définit une seule méthode verifyText() relative à un événement émis lors de la vérification des données avant modification du contenu d'un contrôle de saisie de texte.

Cette méthode possède un objet de type VerifyEvent qui contient des informations sur l'événement grâce à quatre attributs :

doit	un drapeau qui indique si la modification doit être effectuée ou non
end	la position de fin de la modification
start	la position de début de la modification
text	la valeur de la modification

Exemple : n'autoriser la saisie que de chiffres

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.widgets.*;

public class TestSWT11 {

  public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setText("Test");

    Text text = new Text(shell, SWT.BORDER);
    text.setText("");
    text.setSize(100, 25);

    text.addVerifyListener(new VerifyListener() {
      public void verifyText(VerifyEvent e) {
        int valeur = 0;
        e.doit = true;
        if (e.text != "") {
          try {
            valeur = Integer.parseInt(e.text);
          } catch (NumberFormatException e1) {
            e.doit = false;
          }
        }

        System.out.println(
          "start = " + e.start + ", end = " + e.end + ", text = " + e.text);

      }
    });

    shell.pack();
    shell.open();

    while (!shell.isDisposed())
      if (!display.readAndDispatch())
        display.sleep();

    display.dispose();
  }
}
```

46.10.7. L'interface FocusListener

Cette interface définit deux méthodes focusGained() et focusLost() relatives à un événement émis respectivement lors de la prise et la perte du focus par un contrôle.

Ces méthodes possèdent un objet de type FocusEvent.

SWT définit une classe FocusAdapter qui implémente l'interface FocusListener avec des méthodes vides.

Exemple :

```
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.*;

public class TestSWT12 {

  public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);

    Text text = new Text(shell, SWT.BORDER);
    text.setText("mon texte");
    text.setBounds(10, 10, 100, 25);

    text.addFocusListener(new FocusListener() {
      public void focusGained(FocusEvent e) {
        System.out.println(e.widget + " obtient le focus");
      }
      public void focusLost(FocusEvent e) {
        System.out.println(e.widget + " perd le focus");
      }
    });

    Button button = new Button(shell, SWT.NONE);
    button.setText("Valider");
    button.setBounds(10, 40, 100, 25);

    shell.pack();
    shell.open();

    while (!shell.isDisposed())
      if (!display.readAndDispatch())
        display.sleep();

    display.dispose();
  }
}
```

46.10.8. L'interface TraverseListener

Cette interface définit une méthode keyTraversed() relative à un événement émis lors de la traversée d'un contrôle au moyen de la touche tab ou des flèches haut et bas.

Cette méthode possède un objet de type TraverseEvent qui contient des informations sur l'événement grâce à deux attributs :

Attribut	Rôle
doit	un drapeau qui indique si le composant peut être traversé ou non
detail	le type de l'opération qui génère la traversée

Exemple : empêcher le parcours des contrôles dans l'ordre inverse par la touche tab

```
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.*;

public class TestSWT13 {

  public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);

    TraverseListener tl = new TraverseListener() {
      public void keyTraversed(TraverseEvent e) {
        String res = "";
        res = e.widget + " est traverse grace à ";
        switch (e.detail) {
          case SWT.TRAVERSE_TAB_NEXT :
            res = res + " l'appui sur la touche tab";
            e.doit = true;
            break;
          case SWT.TRAVERSE_TAB_PREVIOUS :
            res = res + " l'appui sur la touche shift + tab";
            e.doit = false;
            break;
          default :
            res = res + " un autre moyen";
        }
        System.out.println(res);
      }
    };

    Text text = new Text(shell, SWT.BORDER);
    text.setText("mon texte");
    text.setBounds(10, 10, 100, 25);
    text.addTraverseListener(tl);

    Button button = new Button(shell, SWT.NONE);
    button.setText("Valider");
    button.setBounds(10, 40, 100, 25);
    button.addTraverseListener(tl);

    shell.pack();
    shell.open();

    while (!shell.isDisposed())
      if (!display.readAndDispatch())
        display.sleep();

    display.dispose();
  }
}
```

46.10.9. L'interface PaintListener

Cette interface définit une méthode paintControl() relative à un événement émis lors de la nécessité de redessiner le composant.

Cette méthode possède un objet de type PaintEvent qui contient des informations sur l'événement grâce à plusieurs attributs :

Attribut	Rôle
gc	un objet de type GC qui encapsule le contexte graphique
height	la hauteur de la zone à redessiner
width	la longueur de la zone à redessiner
x	l'abscisse de l'origine de la zone à redessiner
y	l'ordonnée de l'origine de la zone à redessiner

Exemple :

```
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.graphics.*;

public class TestSWT21 {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);
    shell.setSize(420, 420);

    Canvas canvas = new Canvas(shell, SWT.NONE);
    canvas.setSize(200, 200);
    canvas.setLocation(10, 10);
    canvas.addPaintListener(new PaintListener() {
      public void paintControl(PaintEvent e) {
        GC gc = e.gc;
        gc.drawText("Bonjour", 20, 20);
        gc.drawLine(10, 10, 10, 100);
        gc.setForeground(display.getSystemColor(SWT.COLOR_RED));
        gc.drawOval(60, 60, 60, 60);
      }
    });

    shell.pack();
    shell.open();

    GC gc = new GC(canvas);
    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }

    display.dispose();
  }
}
```

Remarque : il est important d'associer le listener avant d'ouvrir la fenêtre. Il n'est pas utile d'utiliser la méthode dispose() de l'objet de type GC car le code n'est pas responsable de son instanciation.

46.11. Les boîtes de dialogue

Comme dans toutes les interfaces graphiques, SWT permet l'utilisation de boîtes de dialogue soit prédéfinies soit personnalisées.

46.11.1. Les boîtes de dialogue prédéfinies

SWT propose plusieurs boîtes de dialogue prédéfinies.

46.11.1.1. La classe MessageBox

La classe MessageBox permet d'afficher un message à l'utilisateur et éventuellement de sélectionner une action standard via un bouton.

Les styles utilisables avec MessageBox sont :

- ICON_ERROR, ICON_INFORMATION, ICON_QUESTION, ICON_WARNING, ICON_WORKING pour sélectionner l'icône affichée dans la boîte de dialogue
- OK ou OK | CANCEL : pour une boîte avec des boutons de type « Ok » / « Annuler »
- YES | NO, YES | NO | CANCEL : pour une boîte avec des boutons de type « Oui » / « Non » / « Annuler »
- RETRY | CANCEL : pour une boîte avec des boutons de type « Réessayer » / « Annuler »
- ABORT | RETRY | IGNORE : pour une boîte avec des boutons de type « Abandon » / « Réessayer » / « Ignorer »

La méthode setMessage() permet de préciser le message qui va être affiché à l'utilisateur.

La méthode open permet d'ouvrir la boîte de dialogue et de connaître le bouton qui a été utilisé pour fermer la boîte de dialogue en comparant la valeur de retour avec la valeur de style du bouton correspondant.

Exemple :

```
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;

public class TestSWT18 {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);
    shell.setLayout(new GridLayout());
    shell.setSize(300, 300);

    Button btnOuvrir = new Button(shell, SWT.PUSH);
    btnOuvrir.setText("Afficher");
    btnOuvrir.addListener(SWT.Selection, new Listener() {
      public void handleEvent(Event e) {
        int reponse = 0;
        MessageBox mb = new MessageBox(shell,
          SWT.ICON_INFORMATION | SWT.ABORT | SWT.RETRY | SWT.IGNORE);
        mb.setMessage("Message d'information pour l'utilisateur");
        reponse = mb.open();
        if (reponse == SWT.ABORT) {
          System.out.println("Bouton abandonner selectionne");
        }
        if (reponse == SWT.RETRY) {
          System.out.println("Bouton reessayer selectionne");
        }
        if (reponse == SWT.IGNORE) {
          System.out.println("Bouton ignorer selectionne");
        }
      }
    });

    shell.pack();
    shell.open();
    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }
    display.dispose();
  }
}
```

46.11.1.2. La classe ColorDialog

Cette boîte de dialogue permet la sélection d'une couleur dans la palette des couleurs.

La méthode setRGB() permet de préciser la couleur qui est sélectionnée par défaut.

La méthode open() permet d'ouvrir la boîte de dialogue et de renvoyer la valeur de la couleur sélectionnée sous la forme d'un objet de type RGB. Si aucune couleur n'est sélectionnée (appui sur le bouton annuler dans la boîte de dialogue) alors l'objet renvoyé est null.

Exemple :

```
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.graphics.*;

public class TestSWT15 {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);
    shell.setLayout(new GridLayout());
    shell.setSize(300, 300);

    Button btnOuvrir = new Button(shell, SWT.PUSH);
    btnOuvrir.setText("Couleur");
    btnOuvrir.addListener(SWT.Selection, new Listener() {
      public void handleEvent(Event e) {
        Color couleurDeFond = shell.getBackground();

        ColorDialog colorDialog = new ColorDialog(shell);
        colorDialog.setRGB(couleurDeFond.getRGB());
        RGB couleur = colorDialog.open();

        if (couleur != null) {
          if (couleurDeFond != null)
            couleurDeFond.dispose();
          couleurDeFond = new Color(display, couleur);
          shell.setBackground(couleurDeFond);
        }
      }
    });

    shell.getBackground().dispose();
    shell.open();
    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }
    display.dispose();
  }
}
```

46.11.1.3. La classe FontDialog

Cette classe encapsule une boîte de dialogue permettant la sélection d'une police de caractère.

La méthode open() permet d'ouvrir la boîte de dialogue et renvoie un objet de type FontData qui encapsule les données de la police sélectionnée ou renvoie null si aucune n'a été sélectionnée.

Exemple :

```
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.layout.*;

public class TestSWT19 {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);
    shell.setLayout(new GridLayout());
    shell.setSize(300, 300);

    Label lblNomPolice = new Label(shell, SWT.NONE);
    lblNomPolice.setText("Nom de la police = ");
    final Text txtNomPolice = new Text(shell, SWT.BORDER | SWT.READ_ONLY);
    txtNomPolice.setText("");
    txtNomPolice.setSize(280, 40);

    Button btnOuvrir = new Button(shell, SWT.PUSH);
    btnOuvrir.setText("Police");
    btnOuvrir.addListener(SWT.Selection, new Listener() {
      public void handleEvent(Event e) {
        FontDialog dialog = new FontDialog(shell, SWT.OPEN);
        FontData fontData = dialog.open();
        if (fontData != null) {
          txtNomPolice.setText(fontData.getName());
          System.out.println("selection de la police " + fontData.getName());
          if (txtNomPolice.getFont() != null) {
            txtNomPolice.getFont().dispose();
          }
          Font font = new Font(display, fontData);
          txtNomPolice.setFont(font);

        }
      }
    });

    shell.pack();
    shell.open();
    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }
    display.dispose();
  }
}
```

46.11.1.4. La classe FileDialog

Cette boîte de dialogue permet de sélectionner un fichier.

La méthode open() ouvre la boîte de dialogue et renvoie le nom du fichier sélectionné. Si aucun fichier n'est sélectionné, alors elle renvoie null.

La méthode setFilterExtensions() permet de préciser sous la forme d'un tableau de chaînes la liste des extensions de fichiers acceptées par la sélection.

Exemple :

```
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;

public class TestSWT16 {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);
    shell.setLayout(new GridLayout());
    shell.setSize(300, 300);

    Label lblNomFichier = new Label(shell, SWT.NONE);
    lblNomFichier.setText("Nom du fichier = ");
    final Text txtNomFichier = new Text(shell, SWT.BORDER | SWT.READ_ONLY);
    txtNomFichier.setText("");
    txtNomFichier.setSize(280,40);

    Button btnOuvrir = new Button(shell, SWT.PUSH);
    btnOuvrir.setText("Ouvrir");
    btnOuvrir.addListener(SWT.Selection, new Listener() {
      public void handleEvent(Event e) {
        String nomFichier;
        FileDialog dialog = new FileDialog(shell, SWT.OPEN);
        dialog.setFilterExtensions(new String[] { "*.java", "*.*" });
        nomFichier = dialog.open();
        if ((nomFichier != null) && (nomFichier.length() != 0)){
          txtNomFichier.setText(nomFichier);
            System.out.println("selection du fichier "+nomFichier);
        }
      }
    });

    shell.pack();
    shell.open();
    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }

    display.dispose();
  }
}
```

46.11.1.5. La classe DirectoryDialog

Cette classe encapsule une boîte de dialogue qui permet la sélection d'un répertoire.

La méthode open() ouvre la boîte de dialogue et renvoie le nom du répertoire sélectionné. Si aucun répertoire n'est sélectionné, alors elle renvoie null.

La méthode setFilterPath() permet de préciser sous la forme d'une chaîne de caractères le répertoire sélectionné par défaut.

La méthode setMessage() permet de préciser un message qui sera affiché dans la boîte de dialogue.

Exemple :

```
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;

public class TestSWT17 {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);
    shell.setLayout(new GridLayout());
    shell.setSize(300, 300);

    Label lblNomFichier = new Label(shell, SWT.NONE);
    lblNomFichier.setText("Nom du fichier = ");
            final Text txtNomRepertoire = new Text(shell, SWT.BORDER | SWT.READ_ONLY);
            txtNomRepertoire.setText("");
            txtNomRepertoire.setSize(280,40);


    Button btnOuvrir = new Button(shell, SWT.PUSH);
    btnOuvrir.setText("Ouvrir");
    btnOuvrir.addListener(SWT.Selection, new Listener() {
      public void handleEvent(Event e) {
        String nomRepertoire;
        DirectoryDialog dialog = new DirectoryDialog(shell, SWT.OPEN);
        dialog.setFilterPath("d:/");
        dialog.setMessage("Test");
        nomRepertoire = dialog.open();
        if ((nomRepertoire != null) && (nomRepertoire.length() != 0)){
          txtNomRepertoire.setText(nomRepertoire);
          System.out.println("selection du repertoire "+nomRepertoire);
        }
      }
    });

    shell.pack();
    shell.open();
    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }
    display.dispose();
  }
}
```

46.11.1.6. La classe PrintDialog

La classe PrintDialog encapsule une boîte de dialogue permettant la sélection d'une imprimante configurée sur le système. Pour utiliser cette classe, il faut importer la package org.eclipse.swt.printing.

La méthode open() permet d'ouvrir la boîte de dialogue et renvoie un objet de type PrinterData qui encapsule les données de l'imprimante sélectionnée ou renvoie null si aucune n'a été sélectionnée.

Exemple :

```
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.printing.*;
import org.eclipse.swt.graphics.*;

public class TestSWT20 {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);
    shell.setLayout(new GridLayout());
    shell.setSize(300, 300);
    Button btnOuvrir = new Button(shell, SWT.PUSH);
    btnOuvrir.setText("Imprimer");
    btnOuvrir.addListener(SWT.Selection, new Listener() {
      public void handleEvent(Event e) {
        PrintDialog dialog = new PrintDialog(shell, SWT.OPEN);
        PrinterData printerData = dialog.open();
        if (printerData != null) {
          Printer printer = new Printer(printerData);
          if (printer.startJob("Test")) {
            printer.startPage();
            GC gc = new GC(printer);
            gc.drawString("Bonjour", 100, 100);
            printer.endPage();
            printer.endJob();
            gc.dispose();
            printer.dispose();
          }
        }
      }
    });

    shell.pack();
    shell.open();
    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }
    display.dispose();
  }
}
```

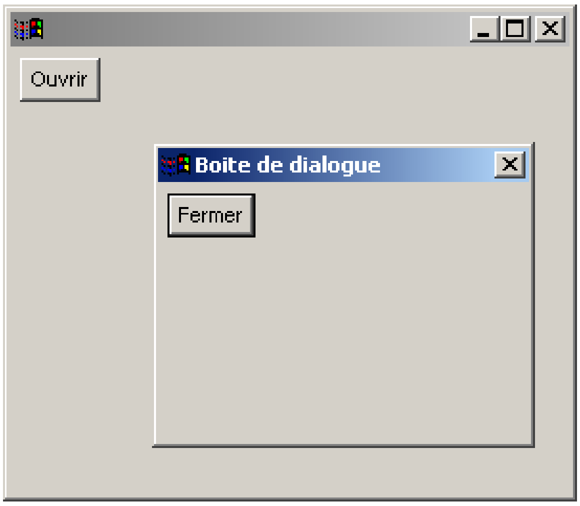
Remarque : cet exemple est très basic dans la mesure où il est préférable de lancer les tâches d'impression dans un thread pour ne pas bloquer l'interface utilisateur pendant ces traitements.

46.11.2. Les boîtes de dialogue personnalisées

Pour définir une fenêtre qui sera une boîte de dialogue, il suffit de définir un nouvel objet de type Shell qui sera lui-même rattaché à sa fenêtre mère.

Exemple :

```
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;

public class TestSWT14 {

  public static void main(String[] args) {
    Display display = new Display();
    final Shell shell = new Shell(display);
    shell.setLayout(new GridLayout());
    shell.setSize(300,300);

    Button btnOuvrir = new Button(shell, SWT.PUSH);
    btnOuvrir.setText("Ouvrir");
    btnOuvrir.addListener(SWT.Selection, new Listener() {
      public void handleEvent(Event e) {
        final Shell fenetreFille = new Shell(shell, SWT.TITLE | SWT.CLOSE);
        fenetreFille.setText("Boite de dialogue");
                        fenetreFille.setLayout(new GridLayout());

        fenetreFille.addListener(SWT.Close, new Listener() {
          public void handleEvent(Event e) {
            System.out.println("Fermeture de la boite de dialogue");
          }
        });

        Button btnFermer = new Button(fenetreFille, SWT.PUSH);
        btnFermer.setText("Fermer");
        btnFermer.addListener(SWT.Selection, new Listener() {
          public void handleEvent(Event e) {
            fenetreFille.close();
          }
        });
        fenetreFille.setSize(200, 200);
        fenetreFille.open();
      }
    });

    shell.open();
    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }
    display.dispose();
  }
}
```



47. JFace

Chapitre 47

Niveau : 🟩🟩🟨⬜⬜⬜ Intermédiaire

SWT est une API de bas niveau. Elle propose des objets qui permettent la création d'interfaces graphiques mais qui nécessitent aussi énormément de code.

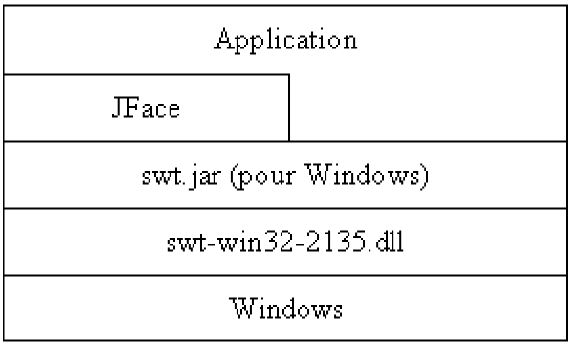
JFace propose d'encapsuler de nombreuses opérations de base et de faciliter ainsi le développement des interfaces graphiques reposant sur SWT.

Ce chapitre contient plusieurs sections :

- La présentation de JFace
- La structure générale d'une application
- Les boites de dialogue

47.1. La présentation de JFace

L'API de JFace est indépendante du système graphique utilisé : la dépendance est réalisée par SWT sur lequel JFace repose.



JFace est une bibliothèque qui facilite l'utilisation de SWT dans le développement d'applications standalones. Elle encapsule un certain nombre de traitements et réduit ainsi la quantité de code à produire.

L'utilisation de JFace n'est pas obligatoire mais sans celle-ci un certain nombre de fonctionnalités proposées par cette API seraient à redévelopper.

JFace n'est fournie en standard qu'avec Eclipse car la partie IHM d'Eclipse est développée avec elle. Cependant elle peut être utilisée dans une application standalone si toutes les bibliothèques requises sont copiées à partir d'Eclipse.

Ces bibliothèques sous la forme de fichiers .jar sont réparties dans plusieurs sous-répertoires du répertoire plug-in d'Eclipse :

Fichier .jar	Sous-répertoire
jface.jar	org.eclipse.jface_3.0.0
jfacetext.jar	org.eclipse.jface.text_3.0.0
osgi.jar	org.eclipse.osgi_3.0.0
runtime.jar	org.eclipse.core.runtime_3.0.0
text.jar	org.eclipse.text_3.0.0

Toutes les bibliothèques doivent être ajoutées dans le classpath de l'application.

Comme JFace repose sur SWT, il est aussi nécessaire d'ajouter la ou les bibliothèques requises par SWT notamment le fichier swt.jar et paramétrer l'application pour qu'elle puisse accéder à la bibliothèque native de SWT. Pour plus de détails, consultez le chapitre sur l'utilisation de SWT.

47.2. La structure générale d'une application

Une application utilisant JFace hérite de la classe ApplicationWindow. Cette classe encapsule un objet de type Shell de SWT.

Elle propose plusieurs méthodes :

Méthodes	Rôle
run()	traitements exécutés par l'application
createContents()	renvoie le composant qui sera affiché dans la fenêtre de l'application

La méthode run() est fréquemment la même :

1. appel à la méthode setBlockOnOpen(true)
2. appel à la méthode open()
3. libération du Display courant

L'appel de ces trois méthodes remplace la création d'un objet de Shell et l'écriture de la boucle de traitement des événements nécessaire en SWT.

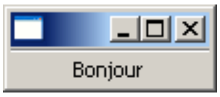
Le booléen passé en paramètre de la méthode setBlockOnOpen() permet simplement de préciser si la méthode doit utiliser ou non la boucle de traitement des événements.

La méthode open() assure l'initialisation et le traitement des événements

La dernière étape permettant la libération des ressources de l'objet Display courant est nécessaire car elle n'est pas réaliser par la méthode open().

Exemple :

```
import org.eclipse.jface.window.ApplicationWindow;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Control;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Label;

public class TestJFace1 extends ApplicationWindow {
  public TestJFace1() {
    super(null);
  }

  public void run() {
    setBlockOnOpen(true);
    open();
    Display.getCurrent().dispose();
  }

  protected Control createContents(Composite parent) {
    Label label = new Label(parent, SWT.CENTER);
    label.setText("Bonjour");
    return label;
  }

  public static void main(String[] args) {
    new TestJFace1().run();
  }
}
```


Bonjour

47.3. Les boites de dialogue

Les boîtes de dialogue proposées par JFace ne remplacent pas celles proposées en standard par SWT. Elles ajoutent d'autres fonctionnalités notamment pour répondre aux besoins particuliers d'Eclipse.

Toutes les classes de ces boîtes de dialogue sont regroupées dans le package org.eclipse.jface.dialogs.

47.3.1. L'affichage des messages d'erreur

JFace propose une boîte de dialogue dédiée à l'affichage de messages d'erreurs. Cette classe est spécifiquement étudiée pour les besoins d'Eclipse dans la mesure où elle utilise un objet de type IStatus.

L'interface IStatus définit les méthodes qui encapsulent une erreur ou une série d'erreurs.

Un status nécessite un code de sévérité. Plusieurs constantes sont définies dans l'interface IStatus

Pour instancier un status, il est nécessaire d'utiliser le seul et unique constructeur de la classe Status qui attend en paramètre :

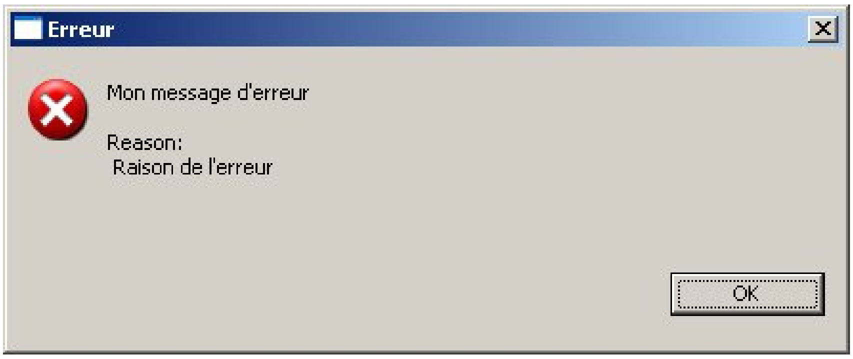
- un entier indiquant la sévérité (les valeurs possibles sont définies par des constantes)
- une chaîne précisant l'identifiant du plug-in
- un entier indiquant le code erreur du plug-in
- une chaîne précisant le message à afficher à l'utilisateur
- une exception

La classe ErrorDialog possède une méthode statique openError() qui attend en paramètres :

- le shell dans lequel la boîte de dialogue doit s'afficher
- le titre de la boîte de dialogue
- le message
- une instance de la classe Status qui encapsule l'erreur

Exemple :

```
import org.eclipse.jface.dialogs.*;
import org.eclipse.jface.window.ApplicationWindow;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.core.runtime.*;

public class TestJFace2 extends ApplicationWindow {

  public TestJFace2() {
    super(null);
  }

  public void run() {
    setBlockOnOpen(true);
    open();
    Display.getCurrent().dispose();
  }

  protected Control createContents(Composite parent) {

    Button boutonAfficher = new Button(parent, SWT.PUSH);
    boutonAfficher.setText("Afficher");
    boutonAfficher.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(SelectionEvent event) {

        Status status = new Status(IStatus.ERROR, "plugin", 0,
                   "Raison de l'erreur", null);
        ErrorDialog.openError(Display.getCurrent().getActiveShell(), "Erreur",
                   "Mon message d'erreur", status);
      }
    });
    return boutonAfficher;
  }

  public static void main(String[] args) {
    new TestJFace2().run();
  }
}
```



Mon message d'erreur

Reason:
 Raison de l'erreur

47.3.2. L'affichage des messages d'information à l'utilisateur

JFace propose une boîte de dialogue permettant d'afficher un message aux utilisateurs encapsulé dans la classe MessageDialog.

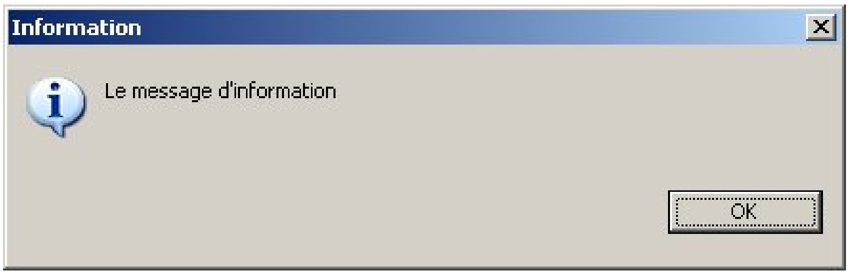
Cette classe encapsule dans différentes méthodes statiques les boîtes de dialogue équivalentes proposées par SWT. Ceci permet de les utiliser avec une seule ligne de code.

Le plus simple pour utiliser cette classe est de faire appel à ses méthodes statiques qui attendent trois paramètres :

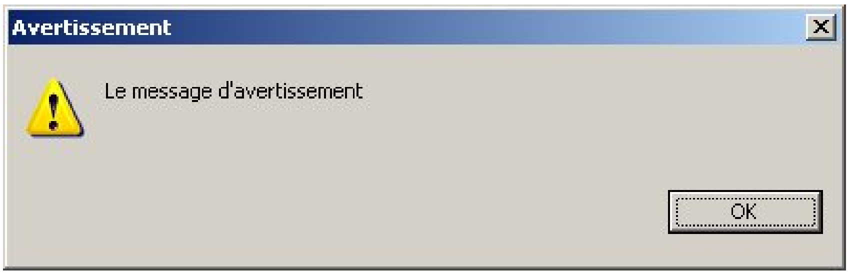
- le shell dans lequel la boîte de dialogue sera affichée
- le titre de la boîte de dialogue
- le message de la boîte de dialogue

Exemple :

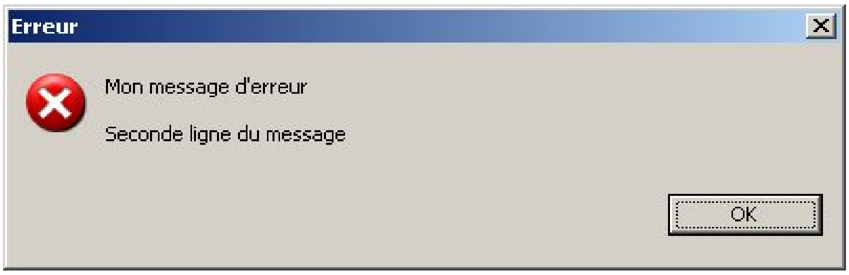
```
import org.eclipse.jface.dialogs.*;
import org.eclipse.jface.window.ApplicationWindow;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;

public class TestJFace3 extends ApplicationWindow {

  public TestJFace3() {
    super(null);
  }

  public void run() {
    setBlockOnOpen(true);
    open();
    Display.getCurrent().dispose();
  }

  protected Control createContents(Composite parent) {

    Button boutonAfficher = new Button(parent, SWT.PUSH);
    boutonAfficher.setText("Afficher");
    final Shell shell = parent.getShell();
    boutonAfficher.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(SelectionEvent event) {
        boolean reponse = false;
        MessageDialog.openInformation(shell, "Information", "Le message d'information");
        MessageDialog.openWarning(shell, "Avertissement", "Le message d'avertissement");
        MessageDialog.openError(shell, "Erreur",
          "Mon message d'erreur\n\nSeconde ligne du message");
        reponse = MessageDialog.openConfirm(shell, "Confirmation",
                    "Le message de la confirmation");
        System.out.println("reponse a la confirmation = " + reponse);
        reponse = MessageDialog.openQuestion(shell, "question",
                    "Le message de la question");
        System.out.println("reponse a la question = " + reponse);
      }
    });
    return boutonAfficher;
  }

  public static void main(String[] args) {
    new TestJFace3().run();
  }
}
```


Information

Le message d'information

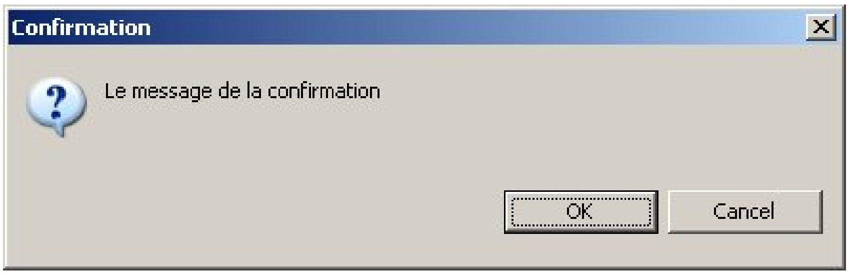

Avertissement

Le message d'avertissement

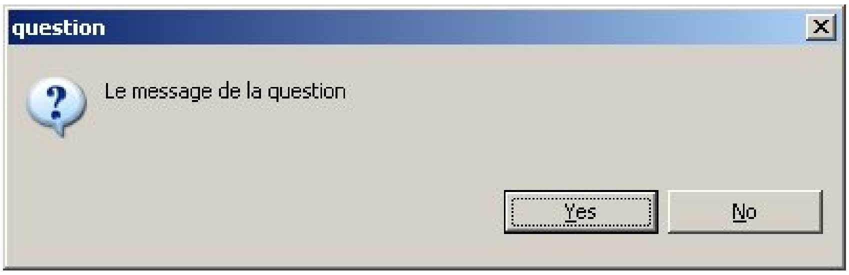


Mon message d'erreur

Seconde ligne du message



Le message de la confirmation



Le message de la question

47.3.3. La saisie d'une valeur par l'utilisateur

JFace propose une boîte de dialogue, encapsulée dans la classe InputDialog, qui permet de demander à l'utilisateur la saisie d'une donnée.

Cette classe possède un constructeur qui attend en paramètre :

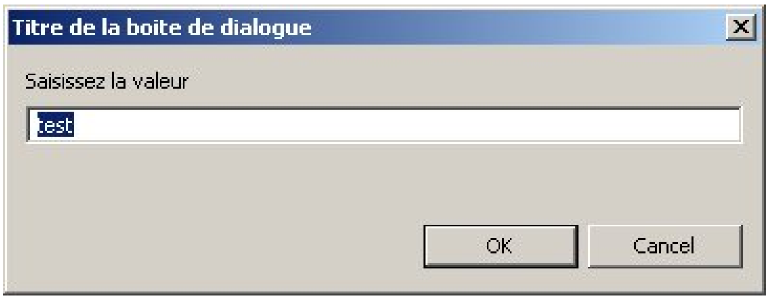
- le shell dans lequel la boîte de dialogue va être affichée
- le titre de la boîte de dialogue
- le texte de la boîte de dialogue
- la valeur des données par défaut à l'affichage de la boîte de dialogue
- un objet de type Validor permettant la validation des données saisies

L'appel à la méthode open() permet d'afficher la boîte de dialogue. La valeur retournée par cette méthode est soit Window.OK soit Window.CANCEL en fonction du bouton cliqué par l'utilisateur.

La méthode getValue() permet d'obtenir la valeur saisie par l'utilisateur si celui-ci a cliqué sur le bouton OK sinon elle renvoie null.

Exemple :

```
import org.eclipse.jface.dialogs.*;
import org.eclipse.jface.window.*;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;

public class TestJFace4 extends ApplicationWindow {

  public TestJFace4() {
    super(null);
  }

  public void run() {
    setBlockOnOpen(true);
    open();
    Display.getCurrent().dispose();
  }

  protected Control createContents(Composite parent) {

    Button boutonAfficher = new Button(parent, SWT.PUSH);
    boutonAfficher.setText("Afficher");
    final Shell shell = parent.getShell();

    boutonAfficher.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(SelectionEvent event) {
        int reponse = 0;

        InputDialog inputDialog = new InputDialog(Display.getCurrent().getActiveShell(),
                   "Titre de la boite de dialogue",
                   "Saisissez la valeur", "test", null);
        reponse = inputDialog.open();

        if (reponse == Window.OK) {
          System.out.println("Valeur saisie = " + inputDialog.getValue());
        } else {
          System.out.println("Operation annulée");
        }
      }
    });
    return boutonAfficher;
  }

  public static void main(String[] args) {
    new TestJFace4().run();
  }
}
```



Une particularité intéressante de cette boîte de dialogue est de pouvoir procéder à une validation des données au fur et à mesure de leur saisie.

Pour cela il faut définir un objet de type IInputValidator. Cette interface définit une unique méthode nommée isValid() qui possède en paramètre la valeur saisie courante et renvoie une chaîne de caractères qui contient un message d'erreur si la valeur n'est pas correcte. Si elle est correcte, il suffit de renvoyer null.

Une fois cette classe définie, il suffit de passer au dernier paramètre du constructeur de la classe InputDialog une instance de la classe réalisant la validation.

47.3.4. La boîte de dialogue pour afficher la progression d'un traitement

Exemple :

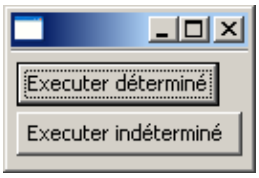
```
import java.lang.reflect.InvocationTargetException;

import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.jface.operation.IRunnableWithProgress;

public class MonTraitement implements IRunnableWithProgress {
  private static final int NB_ITERATION = 100;

  public void run(IProgressMonitor monitor) throws InvocationTargetException,
                                                   InterruptedException {
    monitor.beginTask("Exécution des traitements", NB_ITERATION);
    for (int nb = 0; nb < NB_ITERATION && !monitor.isCanceled(); nb++) {
      Thread.sleep(100);
      monitor.worked(1);
      monitor.subTask("Avancement : " + nb + " %");
    }
    monitor.done();
    if (monitor.isCanceled())
      throw new InterruptedException("Les traitements ont été interrompus");
  }
}
```

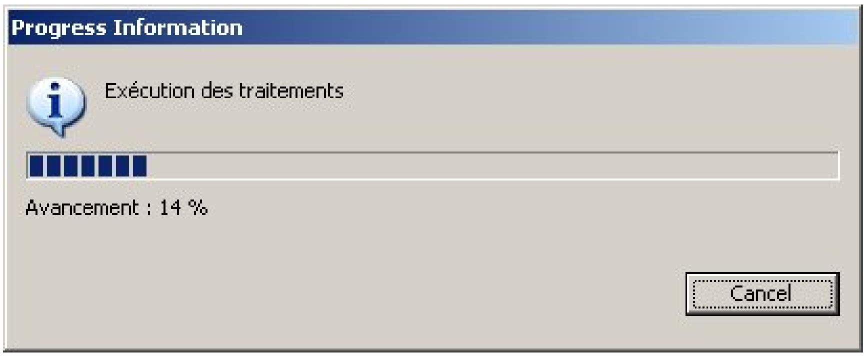
Exemple :

```
import java.lang.reflect.InvocationTargetException;

import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.jface.operation.IRunnableWithProgress;

public class MonTraitementInconnu implements IRunnableWithProgress {

  private static final int NB_ITERATION = 100;

  public void run(IProgressMonitor monitor) throws InvocationTargetException,
                                                   InterruptedException {
    monitor.beginTask("Lancement des traitements", IProgressMonitor.UNKNOWN);
    for (int nb = 0; nb < NB_ITERATION && !monitor.isCanceled(); nb++) {
      Thread.sleep(100);
    }
    monitor.done();
    if (monitor.isCanceled())
      throw new InterruptedException("Les traitements ont été interrompus");
  }
}
```

Exemple :

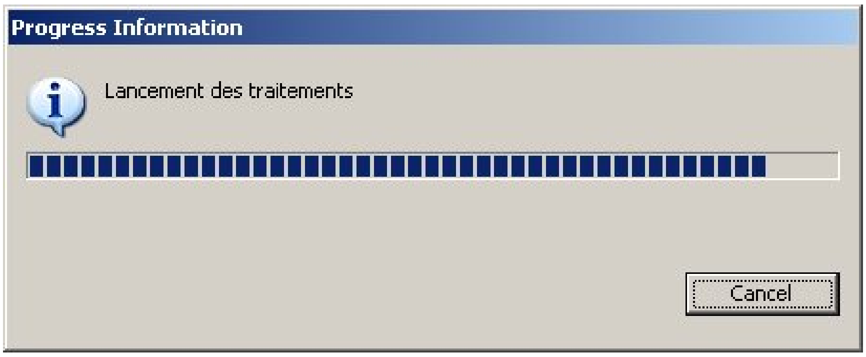
```
import java.lang.reflect.InvocationTargetException;

import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.jface.dialogs.ProgressMonitorDialog;
import org.eclipse.jface.window.ApplicationWindow;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.layout.RowLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Control;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;

public class TestJFace5 extends ApplicationWindow {

  public TestJFace5() {
    super(null);
  }

  public void run() {
    setBlockOnOpen(true);
    open();
    Display.getCurrent().dispose();
  }

  protected Control createContents(Composite parent) {

    Composite composite = new Composite(parent, SWT.NONE);
    composite.setLayout(new RowLayout(SWT.VERTICAL));
    Button boutonExecuterD = new Button(composite, SWT.PUSH);
    boutonExecuterD.setText("Exécuter déterminé");
    final Shell shell = parent.getShell();

    boutonExecuterD.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(SelectionEvent event) {
        try {
          new ProgressMonitorDialog(shell).run(true, true, new MonTraitement());
        } catch (InvocationTargetException e) {
          MessageDialog.openError(shell, "Erreur", e.getMessage());
        } catch (InterruptedException e) {
          MessageDialog.openInformation(shell, "Interruption", e.getMessage());
        }
      }
    });

    Button boutonExecuterU = new Button(composite, SWT.PUSH);
    boutonExecuterU.setText("Exécuter indéterminé");

    boutonExecuterU.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(SelectionEvent event) {
        try {
          new ProgressMonitorDialog(shell).run(true, true, new MonTraitementInconnu());
        } catch (InvocationTargetException e) {
          MessageDialog.openError(shell, "Erreur", e.getMessage());
        } catch (InterruptedException e) {
          MessageDialog.openInformation(shell, "Interruption", e.getMessage());
        }
      }
    });

    return composite;
  }

  public static void main(String[] args) {
    new TestJFace5().run();
  }
}
```





Exécution des traitements

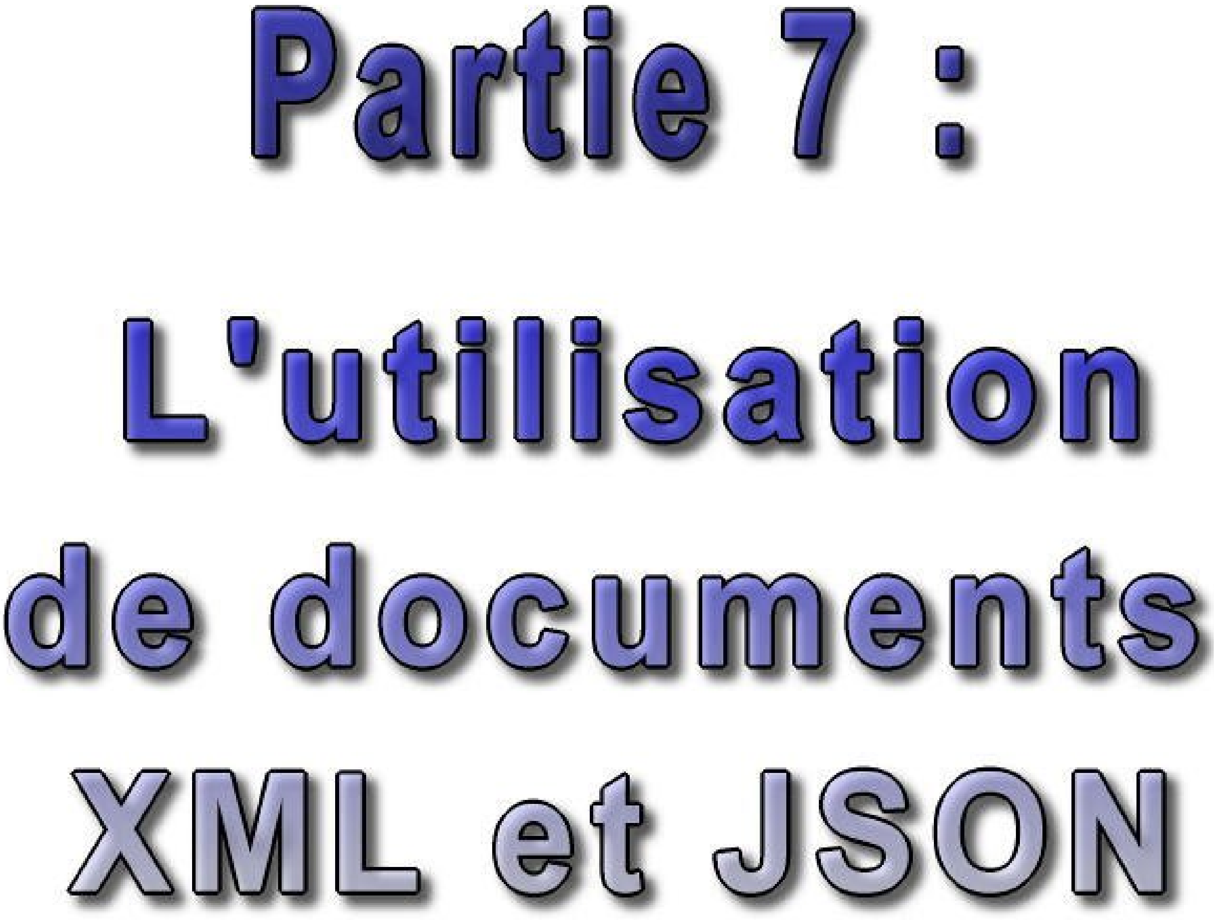
Avancement : 14 %



Lancement des traitements

La suite de ce chapitre sera développée dans une version future de ce document

Partie 7 : L'utilisation de documents XML et JSON



Cette partie traite de l'utilisation de documents XML et JSON avec Java. L'utilisation de documents XML et JSON peut se faire au travers de plusieurs API fournies en standard ou open source.

XML et JSON proposent de structurer des données dans des documents textuels pour permettre leur échange ou leur stockage. Les avantages qu'ils offrent leur permettent d'être largement utilisés.

Cette partie regroupe plusieurs chapitres :

- Java et XML : présente XML qui s'est imposée pour les échanges de données et explore les API Java pour utiliser XML
- SAX (Simple API for XML) : présente l'utilisation de l'API SAX avec Java. Cette API utilise des événements pour traiter un document XML
- DOM (Document Object Model) : présente l'utilisation avec Java de cette spécification du W3C pour proposer une API qui permet de modéliser, de parcourir et de manipuler un document XML
- XSLT (Extensible Stylesheet Language Transformations) : présente l'utilisation avec Java de cette recommandation du W3C pour transformer des documents XML
- Les modèles de documents : présente quelques API open source spécifiques à Java pour traiter un document XML : JDom et Dom4J
- JAXB (Java Architecture for XML Binding) : détaille l'utilisation de cette spécification qui permet de faire correspondre un document XML à un ensemble de classes et vice versa.
- StAX (Streaming Api for XML) : détaille l'utilisation de cette API qui permet de traiter un document XML de façon simple en consommant peu de mémoire tout en permettant de garder le contrôle sur les opérations d'analyse ou d'écriture

- JSON : présente le format JSON
- Gson : Présente l'API Gson de Google pour la lecture et la génération de documents JSON
- JSON-P (Java API for JSON Processing) : détaille l'utilisation de l'API JSON-P spécifiée dans la JSR 353
- JSON-B (Java API for JSON Binding) : détaille l'utilisation de l'API JSON-B qui propose une API standard pour permettre de convertir un document JSON en objet Java et vice versa

48. Java et XML

Chapitre 48

Niveau : 🟩🟩🟨⬜⬜⬜ Intermédiaire

L'utilisation de Java avec XML est facilitée par le fait qu'ils ont plusieurs points communs :

- indépendance de toute plate-forme
- conçus pour être utilisés sur un réseau
- prise en charge de la norme Unicode

Ce chapitre contient plusieurs sections :

- ♦ La présentation de XML
- ♦ Les règles pour formater un document XML
- ♦ La DTD (Document Type Definition)
- ♦ Les parseurs
- ♦ La génération de données au format XML
- ♦ JAXP : Java API for XML Parsing
- ♦ Jaxen

48.1. La présentation de XML

XML est l'acronyme de «eXtensible Markup Language».

XML permet d'échanger des données entres applications hétérogènes car il peut modéliser et stocker des données de façon portable.

XML est extensible dans la mesure où il n'utilise pas de tags prédéfinis comme HTML et il permet de définir de nouvelles balises : c'est un métalangage.

Le format HTML est utilisé pour formater et afficher les données qu'il contient : il est destiné à structurer, formater et échanger des documents d'une façon aussi standard que possible.

XML est utilisé pour modéliser et stocker des données. Il ne permet pas à lui seul d'afficher les données qu'il contient.

Pourtant, XML et HTML sont tous les deux des dérivés d'un langage nommé SGML (Standard Generalized Markup Language). La création d'XML est liée à la complexité de SGML. D'ailleurs, un fichier XML avec sa DTD correspondante peut être traité par un processeur SGML.

XML et Java ont en commun la portabilité réalisée grâce à une indépendance vis-à-vis du système et de son environnement.

48.2. Les règles pour formater un document XML

Un certain nombre de règles doivent être respectées pour définir un document XML valide et «bien formé». Pour pouvoir être analysé, un document XML doit avoir une syntaxe correcte. Les principales règles sont :

- le document doit contenir au moins une balise
- chaque balise d'ouverture (exemple <tag>) doit posséder une balise de fermeture (exemple </tag>). Si le tag est vide, c'est à dire qu'il ne possède aucune données (exemple <tag></tag>), un tag abrégé peut être utilisé (exemple correspondant : <tag/>)
- les balises ne peuvent pas être intercalées (exemple <liste><element></liste></element> n'est pas autorisé)
- toutes les balises du document doivent obligatoirement être contenues dans l'espace défini par une balise unique nommée élément racine
- les valeurs des attributs doivent obligatoirement être encadrées avec des quotes simples ou doubles
- les balises sont sensibles à la casse
- Les balises peuvent contenir des attributs même les balises vides
- les données incluses entre les balises ne doivent pas contenir de caractères < et & : il faut utiliser respectivement &lt ; et &amp ;
- La première ligne du document devrait normalement correspondre à la déclaration de document XML : le prologue.

48.3. La DTD (Document Type Definition)

Les balises d'un document XML sont libres. Pour pouvoir valider le document, il faut définir un document nommé DTD qui est optionnel. Sans sa présence, le document ne peut être validé : on peut simplement vérifier que la syntaxe du document est correcte.

Une DTD est un document qui contient la grammaire définissant le document XML. Elle précise notamment les balises autorisées et comment elles s'imbriquent.

La DTD peut être incluse dans l'en-tête du document XML ou être mise dans un fichier indépendant. Dans ce cas, la directive < !DOCTYPE> dans le document XML permet de préciser le fichier qui contient la DTD.

Il est possible d'utiliser une DTD publique ou de définir sa propre DTD si aucune ne correspond à ses besoins.

Pour être valide, un document XML doit avoir une syntaxe correcte et correspondre à la DTD.

48.4. Les parseurs

Il existe plusieurs types de parseurs. Les plus répandus sont ceux qui utilisent un arbre pour représenter et exploiter le document et ceux qui utilisent des événements. Le parseur peut en plus permettre de valider le document XML.

Ceux qui utilisent un arbre permettent de le parcourir pour obtenir les données et modifier le document.
Ceux qui utilisent des événements associent à des événements particuliers des méthodes pour traiter le document.

SAX (Simple API for XML) est une API libre créée par David Megginson qui utilise les événements pour analyser et exploiter les documents au format XML.

Les parseurs qui produisent des objets composant une arborescence pour représenter le document XML utilisent le modèle DOM (Document Object Model) défini par les recommandations du W3C.

Le choix d'utiliser SAX ou DOM doit tenir compte de leurs points forts et de leurs faiblesses :

	les avantages	les inconvénients
DOM	parcours libre de l'arbre	gourmand en mémoire

	possibilité de modifier la structure et le contenu de l'arbre	doit traiter tout le document avant d'exploiter les résultats
SAX	peu gourmand en ressources mémoire rapide principes faciles à mettre en oeuvre permet de ne traiter que les données utiles	traite les données séquentiellement un peu plus difficile à programmer, il est souvent nécessaire de sauvegarder des informations pour les traiter

SAX et DOM ne fournissent que des définitions : ils ne fournissent pas d'implémentation utilisable. L'implémentation est laissée aux différents éditeurs qui fournissent un parseur compatible avec SAX et/ou DOM. L'avantage d'utiliser l'un d'eux est que le code utilisé sera compatible avec les autres : le code nécessaire à l'instanciation du parseur est cependant spécifique à chaque fournisseur.

IBM fournit gratuitement un parseur XML : xml4j. Il est téléchargeable à l'adresse suivante : http://www.alphaworks.ibm.com/tech/xml4j

Le groupe Apache développe Xerces à partir de xml4j : il est possible de télécharger la dernière version à l'URL http://xml.apache.org

Sun a développé un projet dénommé Project X. Ce projet a été repris par le groupe Apache sous le nom de Crimson.

Ces trois projets apportent pour la plupart les mêmes fonctionnalités : ils se distinguent sur des points mineurs : performance, rapidité, facilité d'utilisation etc. ... Ces fonctionnalités évoluent très vite avec les versions de ces parseurs qui se succèdent très rapidement.

Pour les utiliser, il suffit de décompresser le fichier et d'ajouter les fichiers .jar dans la variable définissant le CLASSPATH.

Il existe plusieurs autres parseurs que l'on peut télécharger sur le web.

48.5. La génération de données au format XML

Il existe plusieurs façons de générer des données au format XML :

- coder cette génération à la main en écrivant dans un flux

Exemple :

```
public void service(HttpServletRequest request, HttpServletResponse response)
  throws IOException, ServletException {
  response.setContentType("text/xml");
  PrintWriter out = response.getWriter();

  out.println("<?xml version=\"1.0\"?>");
  out.println("<BIBLIOTHEQUE>");
  out.println(" <LIVRE>");
  out.println(" <TITRE>titre livre 1</TITRE>");
  out.println(" <AUTEUR>auteur 1</AUTEUR>");
  out.println(" <EDITEUR>editeur 1</EDITEUR>");
  out.println(" </LIVRE>");
  out.println(" <LIVRE>");
  out.println(" <TITRE>titre livre 2</TITRE>" );
  out.println(" <AUTEUR>auteur 2</AUTEUR>");
  out.println(" <EDITEUR>editeur 2</EDITEUR>" );
  out.println(" </LIVRE>");
  out.println(" <LIVRE>");
  out.println(" <TITRE>titre livre 3</TITRE>");
  out.println(" <AUTEUR>auteur 3</AUTEUR>");
  out.println(" <EDITEUR>editeur 3</EDITEUR>");
  out.println(" </LIVRE>");
  out.println("</BIBLIOTHEQUE>");
  }
}
```

- utiliser JDOM pour construire le document et le sauvegarder
- utiliser la classe javax.xml.stream.XmlStreamWriter
- utiliser la classe java.beans.XMLEncoder pour sérialiser un bean
- utiliser une API open source comme XStream

La suite de cette section sera développée dans une version future de ce document

48.6. JAXP : Java API for XML Parsing

JAXP est une API développée par Sun qui ne fournit pas une nouvelle méthode pour parser un document XML mais propose une interface commune pour appeler et paramétrer un parseur de façon indépendante de tout fournisseur et normaliser la source XML à traiter. En utilisant un code qui respecte JAXP, il est possible d'utiliser n'importe quel parseur qui répond à cette API tel que Crimson le parseur de Sun ou Xerces le parseur du groupe Apache.

JAXP supporte pour le moment les parseurs de type SAX et DOM.

	JAXP 1.0	JAXP 1.1
SAX	type 1	type 2
DOM	niveau 1	niveau 2

Par exemple, sans utiliser JAXP, il existe deux méthodes pour instancier un parseur de type SAX :

- créer une instance de la classe de type SaxParser
- utiliser la classe ParserFactory qui demande en paramètre le nom de la classe de type SaxParser

Ces deux possibilités nécessitent une recompilation d'une partie du code lors du changement du parseur.

JAXP propose de fournir le nom de la classe du parseur en paramètre à la JVM sous la forme d'une propriété système. Il n'est ainsi plus nécessaire de procéder à une recompilation car il suffit de mettre à jour cette propriété et le CLASSPATH pour qu'il référence les classes du nouveau parseur.

Le parseur de Sun et les principaux parseurs XML en Java implémentent cette API et il est très probable que tous les autres fournisseurs suivent cet exemple.

48.6.1. JAXP 1.1

JAXP version 1.1 contient une documentation au format javadoc, des exemples et trois fichiers jar :

- jaxp.jar : contient l'API JAXP
- crimson.jar : contient le parseur de Sun
- xalan.jar : contient l'outil du groupe Apache pour les transformations XSL

L'API JAXP est fournie avec une implémentation de référence de deux parseurs (un de type SAX et un de type DOM) dans le package org.apache.crimson et ses sous-packages.

JAXP se compose de plusieurs packages :

- javax.xml.parsers
- javax.xml.transform
- org.w3c.dom
- org.xml.sax

JAXP définit deux exceptions particulières :

- FactoryConfigurationError est levée si la classe du parseur précisée dans la variable System ne peut être instanciée
- ParserConfigurationException est levée lorsque les options précisées dans la factory ne sont pas supportées par le parseur

48.6.2. L'utilisation de JAXP avec un parseur de type SAX

L'API JAXP fournit la classe abstraite SAXParserFactory qui propose une méthode statique pour récupérer une instance d'un parseur de type SAX. Une classe fille instanciable de la classe SAXParserFactory est fournie par l'implémentation.

La propriété système javax.xml.parsers.SAXParserFActory permet de préciser cette classe fille qui hérite de la classe SAXParserFactory et qui sera instanciée.

Remarque : cette classe n'est pas thread safe.

La méthode statique newInstance() permet d'obtenir une instance de la classe SAXParserFactory et peut lever une exception de type FactoryConfigurationError.

Avant d'obtenir une instance du parseur, il est possible de fournir quelques paramètres à la Factory pour lui permettre de le configurer.

La méthode newSAXParser() permet d'obtenir une instance du parseur de type SAXParser : peut lever une exception de type ParserConfigurationException.

Les principales méthodes sont :

Méthode	Rôle
boolean isNamespaceAware()	indique si la factory est configurée pour instancier des parseurs qui prennent en charge les espaces de noms
boolean isValidating()	indique si la factory est configurée pour instancier des parseurs qui valident le document XML lors de son traitement
static SAXParserFactory newInstance()	permet d'obtenir une instance de la factory
SAXParser newSAXParser()	permet d'obtenir une nouvelle instance du parseur de type SAX configuré avec les options fournies à la factory
setNamespaceAware(boolean)	configure la factory pour instancier un parseur qui prend en charge les espaces de noms ou non selon le paramètre fourni
setValidating(boolean)	configure la factory pour instancier un parseur qui valide le document XML lors de son traitement ou non selon le paramètre fourni

Exemple :

```
SAXParserFActory factory = SAXParserFactory.newInstance();
SAXParser parser = factory.newSAXParser();
parser.parse(new File(args[0), new handler());
```

48.7. Jaxen

Jaxen est un moteur Xpath qui permet de retrouver des informations dans un document XML de type dom4j ou Jdom.

C'est un projet open source qui a été intégré dans dom4j pour permettre le support de Xpath dans ce framework.

49. SAX (Simple API for XML)

Chapitre 49

Niveau : 🟩🟩🟨⬜⬜⬜ Intermédiaire

SAX est l'acronyme de Simple API for XML. Cette API a été développée par David Megginson.

Ce type de parseur utilise des événements pour piloter le traitement d'un fichier XML. Un objet (nommé handler en anglais) doit implémenter des méthodes particulières définies dans une interface de l'API pour fournir les traitements à réaliser : selon les événements, le parseur appelle ces méthodes.

Les dernières informations concernant cette API sont disponible à l'URL : www.megginson.com/SAX/index.html

Les classes de l'API SAX sont regroupées dans le package org.xml.sax

Ce chapitre contient plusieurs sections :

- L'utilisation de SAX de type 1
- L'utilisation de SAX de type 2

49.1. L'utilisation de SAX de type 1

SAX type 1 est composé de deux packages :

- org.xml.sax :
- org.xml.sax.helpers :

SAX définit plusieurs classes et interfaces :

- les interfaces implémentées par le parseur : Parser, AttributeList et Locator
- les interfaces implémentées par le handler : DocumentHandler, ErrorHandler, DTDHandler et EntityHandler
- les classes de SAX :
- des utilitaires rassemblés dans le package org.xml.sax.helpers notamment la classe ParserFactory

Les exemples de cette section utilisent la version 2.0.15 du parseur xml4j d'IBM.

Pour parser un document XML avec un parseur XML SAX de type 1, il faut suivre les étapes suivantes :

- créer une classe qui implémente l'interface DocumentHandler ou hérite de la classe org.xml.sax.HandlerBase et qui se charge de répondre aux différents événements émis par le parseur
- créer une instance du parseur en utilisant la méthode makeParser() de la classe ParserFactory.
- associer le handler au parseur grâce à la méthode setDocumentHandler()
- exécuter la méthode parse() du parseur

Exemple : avec XML4J

```
import org.xml.sax.*;
import org.xml.sax.helpers.ParserFactory;
import com.ibm.xml.parsers.*;
import java.io.*;

public class MessageXML {
  static final String DONNEES_XML =
    "<?xml version=\"1.0\"?>\n"
    +"<BIBLIOTHEQUE>\n"
    +"  <LIVRE>\n"
    +"    <TITRE>titre livre 1</TITRE>\n"
    +"    <AUTEUR>auteur 1</AUTEUR>\n"
    +"    <EDITEUR>editeur 1</EDITEUR>\n"
    +"  </LIVRE>\n"
    +"  <LIVRE>\n"
    +"    <TITRE>titre livre 2</TITRE>\n"
    +"    <AUTEUR>auteur 2</AUTEUR>\n"
    +"    <EDITEUR>editeur 2</EDITEUR>\n"
    +"  </LIVRE>\n"
    +"  <LIVRE>\n"
    +"    <TITRE>titre livre 3</TITRE>\n"
    +"    <AUTEUR>auteur 3</AUTEUR>\n"
    +"    <EDITEUR>editeur 3</EDITEUR>\n"
    +"  </LIVRE>\n"
    +"</BIBLIOTHEQUE>\n";

  static final String CLASSE_PARSER = "com.ibm.xml.parsers.SAXParser";

  /**
   * Lance l'application.
   * @param args un tableau d'arguments de la ligne de commandes
   */
  public static void main(java.lang.String[] args) {

    MessageXML m = new MessageXML();
    m.parse();

    System.exit(0);
  }

  public MessageXML() {
    super();
  }

  public void parse() {
    TestXMLHandler handler = new TestXMLHandler();

    System.out.println("Lancement du parseur");

    try {
      Parser parser = ParserFactory.makeParser(CLASSE_PARSER);

      parser.setDocumentHandler(handler);
      parser.setErrorHandler((ErrorHandler) handler);

      parser.parse(new InputSource(new StringReader(DONNEES_XML)));

    } catch (Exception e) {
      System.out.println("Exception capturée : ");
      e.printStackTrace(System.out);
      return;
    }
  }
}
```

Il faut ensuite créer la classe du handler.

Exemple :

```
import java.util.*;

/**
```

```
 * Classe utilisée pour gérer les événements émis par SAX lors du traitement du fichier XML
 */
public class TestXMLHandler extends org.xml.sax.HandlerBase {
  public TestXMLHandler() {
    super();
  }

  /**
   * Actions à réaliser sur les données
   */
  public void characters(char[] caracteres, int debut, int longueur) {
    String donnees = new String(caracteres, debut, longueur);
    System.out.println("   valeur = *" + donnees + "*");
  }

  /**
   * Actions à réaliser lors de la fin du document XML.
   */
  public void endDocument() {
    System.out.println("Fin du document");
  }

  /**
   * Actions à réaliser lors de la détection de la fin d'un élément.
   */
  public void endElement(String name) {
    System.out.println("Fin tag " + name);
  }

  /**
   * Actions à réaliser au début du document.
   */
  public void startDocument() {
    System.out.println("Debut du document");
  }

  /**
   * Actions à réaliser lors de la détection d'un nouvel élément.
   */
  public void startElement(String name, org.xml.sax.AttributeList atts) {
    System.out.println("debut tag : " + name);
  }
}
```

Résultat :

```
Lancement du parser
Debut du document
debut tag : BIBLIOTHEQUE
   valeur = *
  *
debut tag : LIVRE
   valeur = *
    *
debut tag : TITRE
   valeur = *titre livre 1*
Fin tag TITRE
   valeur = *
    *
debut tag : AUTEUR
   valeur = *auteur 1*
Fin tag AUTEUR
   valeur = *
    *
debut tag : EDITEUR
   valeur = *editeur 1*
Fin tag EDITEUR
   valeur = *
  *
Fin tag LIVRE
   valeur = *
  *
debut tag : LIVRE
   valeur = *
```

```
    *
debut tag : TITRE
   valeur = *titre livre 2*
Fin tag TITRE
   valeur = *
    *
debut tag : AUTEUR
   valeur = *auteur 2*
Fin tag AUTEUR
   valeur = *
    *
debut tag : EDITEUR
   valeur = *editeur 2*
Fin tag EDITEUR
   valeur = *
  *
Fin tag LIVRE
   valeur = *
  *
debut tag : LIVRE
   valeur = *
    *
debut tag : TITRE
   valeur = *titre livre 3*
Fin tag TITRE
   valeur = *
    *
debut tag : AUTEUR
   valeur = *auteur 3*
Fin tag AUTEUR
   valeur = *
    *
debut tag : EDITEUR
   valeur = *editeur 3*
Fin tag EDITEUR
   valeur = *
  *
Fin tag LIVRE
   valeur = *
*
Fin tag BIBLIOTHEQUE
Fin du document
```

Un parseur SAX peut créer plusieurs types d'événements. Les principales méthodes pour y répondre sont :

Evénement	Rôle
startElement()	cette méthode est appelée lors de la détection d'un tag de début
endElement()	cette méthode est appelée lors de la détection d'un tag de fin
characters()	cette méthode est appelée lors de la détection de données entre deux tags
startDocument()	cette méthode est appelée lors du début du traitement du document XML
endDocument()	cette méthode est appelée lors de la fin du traitement du document XML

La classe handler doit redéfinir certaines de ces méthodes selon les besoins des traitements.

En règle générale :

- il faut sauvegarder dans une variable le tag courant détecté dans la méthode startElement()
- traiter les données en fonction du tag courant dans la méthode characters()

La sauvegarde du tag courant est obligatoire car la méthode characters() ne contient pas dans ses paramètres le nom du tag correspondant aux données.

Si les données contenues dans le document XML contiennent plusieurs occurrences qu'il faut gérer avec une collection qui contiendra des objets encapsulant les données, il faut :

- gérer la création d'un objet dans la méthode startElement() lors de la rencontre du tag de début d'un nouvel élément de la liste
- alimenter les attributs de l'objet avec les données de chaque tag utile dans la méthode characters()
- gérer l'ajout de l'objet à la collection dans la méthode endElement() lors de la rencontre du tag de fin d'élément de la liste

La méthode characters() est appelée lors de la détection de données entre un tag de début et un tag de fin mais aussi entre un tag de fin et le tag de début suivant lorsqu'il y a des caractères entre les deux. Ces caractères ne sont pas des données mais des espaces, des tabulations, des retour chariots et certains caractères non visibles.

Pour éviter de traiter les données de ces événements, il y a plusieurs solutions :

- supprimer tous les caractères entre les tags : tous les tags et les données sont rassemblés sur une seule et unique ligne. L'inconvénient de cette méthode est que le message est difficilement lisible par un être humain.
- une autre méthode consiste à remettre à vide la chaîne de caractères qui contient le tag courant (alimentée dans la méthode startElement()) dans la méthode endElement(). Il suffit alors d'effectuer les traitements dans la méthode characters() uniquement si le tag courant est différent de vide

Exemple :

```
import java.util.*;

/**
 * Classe utilisée pour gérer les événement émis par SAX lors du traitement du fichier XML
 */
public class TestXMLHandler extends org.xml.sax.HandlerBase {
  private String tagCourant = "";
  public TestXMLHandler() {
    super();
  }

  /**
   * Actions à réaliser sur les données
   */
  public void characters(char[] caracteres, int debut, int longueur) {
    String donnees = new String(caracteres, debut, longueur);
    if (!tagCourant.equals("") {
      System.out.println("  Element " + tagCourant
        + ", valeur = *" + donnees + "*");
    }
  }

  /**
   * Actions à réaliser lors de la fin du document XML.
   */
  public void endDocument() {
    System.out.println("Fin du document");
  }

  /**
   * Actions à réaliser lors de la détection de la fin d'un élément.
   */
  public void endElement(String name) {
    tagCourant = "";
    System.out.println("Fin tag " + name);
  }

  /**
   * Actions à réaliser au début du document.
   */
  public void startDocument() {
    System.out.println("Debut du document");
  }

  /**
```

```
   * Actions a réaliser lors de la détection d'un nouvel élément.
   */
  public void startElement(String name, org.xml.sax.AttributeList atts) {
    tagCourant = name;
    System.out.println("debut tag : " + name);
  }
}
```

Résultat :

```
Lancement du parser
Debut du document
debut tag : BIBLIOTHEQUE
   Element BIBLIOTHEQUE, valeur = *
  *
debut tag : LIVRE
   Element LIVRE, valeur = *
    *
debut tag : TITRE
   Element TITRE, valeur = *titre livre 1*
Fin tag TITRE
debut tag : AUTEUR
   Element AUTEUR, valeur = *auteur 1*
Fin tag AUTEUR
debut tag : EDITEUR
   Element EDITEUR, valeur = *editeur 1*
Fin tag EDITEUR
Fin tag LIVRE
debut tag : LIVRE
   Element LIVRE, valeur = *
    *
debut tag : TITRE
   Element TITRE, valeur = *titre livre 2*
Fin tag TITRE
debut tag : AUTEUR
   Element AUTEUR, valeur = *auteur 2*
Fin tag AUTEUR
debut tag : EDITEUR
   Element EDITEUR, valeur = *editeur 2*
Fin tag EDITEUR
Fin tag LIVRE
debut tag : LIVRE
   Element LIVRE, valeur = *
    *
debut tag : TITRE
   Element TITRE, valeur = *titre livre 3*
Fin tag TITRE
debut tag : AUTEUR
   Element AUTEUR, valeur = *auteur 3*
Fin tag AUTEUR
debut tag : EDITEUR
   Element EDITEUR, valeur = *editeur 3*
Fin tag EDITEUR
Fin tag LIVRE
Fin tag BIBLIOTHEQUE
Fin du document
```

- enfin il est possible de vérifier si le premier caractère des données contenues en paramètre de la méthode characters() est un caractère de contrôle ou non grâce à la méthode statique isISOControl() de la classe Character

Exemple :

```
...
/**
 * Actions à réaliser sur les données
 */
public void characters(char[] caracteres, int debut, int longueur) {
        String donnees = new String(caracteres, debut, longueur);

        if (!tagCourant.equals("")) {
                if(!Character.isISOControl(caracteres[debut])) {
                        System.out.println("   Element " + tagCourant
                 +", valeur = *" + donnees + "*");
                }
        }
}
...
```

Résultat :

```
Lancement du parser
Debut du document
debut tag : BIBLIOTHEQUE
debut tag : LIVRE
debut tag : TITRE
   Element TITRE, valeur = *titre livre 1*
Fin tag TITRE
debut tag : AUTEUR
   Element AUTEUR, valeur = *auteur 1*
Fin tag AUTEUR
debut tag : EDITEUR
   Element EDITEUR, valeur = *editeur 1*
Fin tag EDITEUR
Fin tag LIVRE
debut tag : LIVRE
debut tag : TITRE
   Element TITRE, valeur = *titre livre 2*
Fin tag TITRE
debut tag : AUTEUR
   Element AUTEUR, valeur = *auteur 2*
Fin tag AUTEUR
debut tag : EDITEUR
   Element EDITEUR, valeur = *editeur 2*
Fin tag EDITEUR
Fin tag LIVRE
debut tag : LIVRE
debut tag : TITRE
   Element TITRE, valeur = *titre livre 3*
Fin tag TITRE
debut tag : AUTEUR
   Element AUTEUR, valeur = *auteur 3*
Fin tag AUTEUR
debut tag : EDITEUR
   Element EDITEUR, valeur = *editeur 3*
Fin tag EDITEUR
Fin tag LIVRE
Fin tag BIBLIOTHEQUE
Fin du document
```

SAX définit une exception de type SAXParserException lorsque le parseur détecte une erreur dans le document en cours de traitement. Les méthodes getLineNumber() et getColumnNumber() permettent d'obtenir la ligne et la colonne où l'erreur a été détectée.

Exemple :

```
try {
...
}  catch (SAXParseException e) {
  System.out.println("Erreur lors du traitement du document XML");
  System.out.println(e.getMessage());
  System.out.println("ligne : "+e.getLineNumber());
  System.out.println("colonne : "+e.getColumnNumber());
  }
```

Pour les autres erreurs, SAX définit l'exception SAXException.

49.2. L'utilisation de SAX de type 2

SAX de type 2 apporte principalement le support des espaces de noms. Les classes et les interfaces sont toujours définies dans les packages org.xml.sax et ses sous-packages.

SAX de type 2 définit quatre interfaces que l'objet handler doit ou peut implémenter :

- ContentHandler : interface qui définit les méthodes appelées lors du traitement du document
- ErrorHandler : interface qui définit les méthodes appelées lors du traitement des warnings et des erreurs
- DTDHandler : interface qui définit les méthodes appelées lors du traitement de la DTD
- EntityResolver

Plusieurs classes et interfaces de SAX de type 1 sont deprecated :

	ancienne entité SAX 1	nouvelle entité SAX 2
Interface	org.xml.sax.Parser	XMLReader
	org.xml.sax.DocumentHandler	ContentHandler
	org.xml.sax.AttributeList	Attributes
Classes	org.xml.sax.helpers.ParserFactory	
	org.xml.sax.HandlerBase	DefaultHandler
	org.xml.sax.helpers.AttributeListImpl	AttributesImpl

Les principes de fonctionnement de SAX 2 sont très proches de SAX 1.

Exemple :

```
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class TestSAX2
{

  public static void main(String[] args)
  {
    try
    {
      Class c = Class.forName("org.apache.xerces.parsers.SAXParser");
      XMLReader reader = (XMLReader)c.newInstance();
      TestSAX2Handler handler = new TestSAX2Handler();
      reader.setContentHandler(handler);
      reader.parse("test.xml");
    }
    catch(Exception e){System.out.println(e);}
  }

}

class TestSAX2Handler extends DefaultHandler
{
  private String tagCourant = "";

  /**
   * Actions à réaliser lors de la détection d'un nouvel élément.
   */
  public void startElement(String nameSpace, String localName,
    String qName, Attributes attr) throws SAXException  {
    tagCourant = localName;
    System.out.println("debut tag : " + localName);
  }
  /**
   * Actions à réaliser lors de la détection de la fin d'un élément.
   */
  public void endElement(String nameSpace, String localName,
    String qName) throws SAXException {
    tagCourant = "";
    System.out.println("Fin tag " + localName);
  }

  /**
   * Actions à réaliser au début du document.
   */
  public void startDocument() {
    System.out.println("Debut du document");
  }

  /**
   * Actions à réaliser lors de la fin du document XML.
   */
  public void endDocument() {
    System.out.println("Fin du document");
  }

  /**
   * Actions à réaliser sur les données
   */
  public void characters(char[] caracteres, int debut,
    int longueur) throws SAXException {
    String donnees = new String(caracteres, debut, longueur);

    if (!tagCourant.equals("")) {
      if(!Character.isISOControl(caracteres[debut])) {
        System.out.println("   Element " + tagCourant +",
          valeur = *" + donnees + "*");
      }
    }
  }
}
```

50. DOM (Document Object Model)

Niveau : 🟩🟩🟨⬜⬜⬜ Intermédiaire

DOM est l'acronyme de Document Object Model. C'est une spécification du W3C pour proposer une API qui permet de modéliser, de parcourir et de manipuler un document XML.

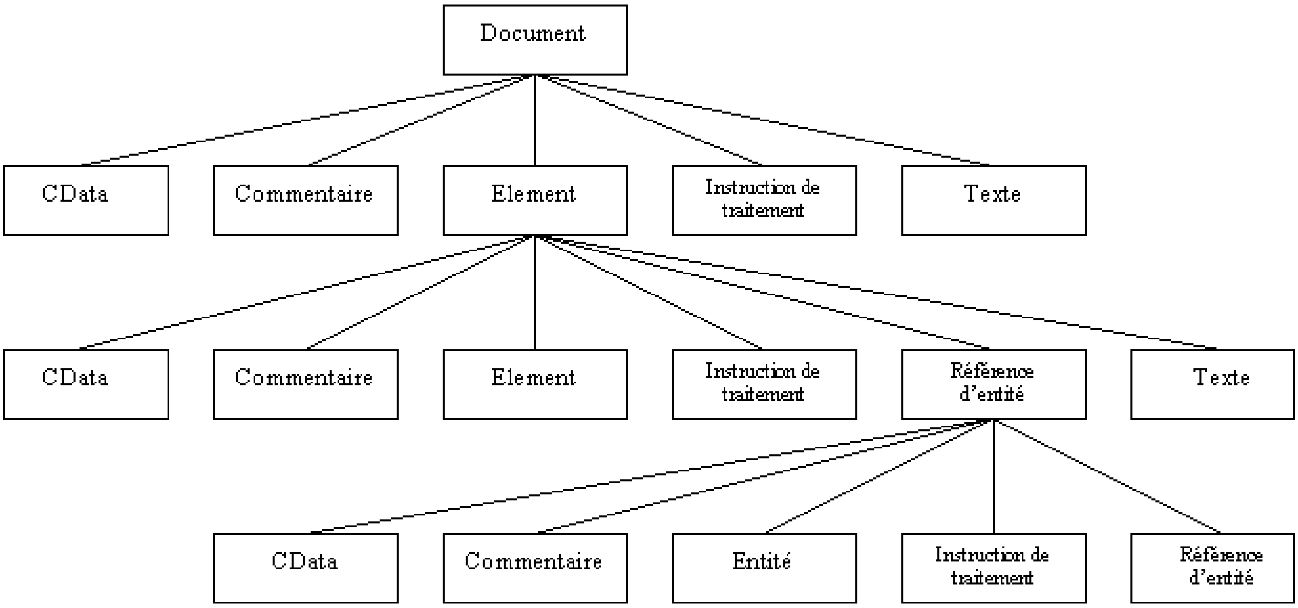
Le principal rôle de DOM est de fournir une représentation mémoire d'un document XML sous la forme d'un arbre d'objets et d'en permettre la manipulation (parcours, recherche et mise à jour)

A partir de cette représentation (le modèle), DOM propose de parcourir le document mais aussi de pouvoir le modifier. Ce dernier aspect est l'un des aspects les plus intéressants de DOM.

DOM est défini pour être indépendant du langage dans lequel il sera implémenté. DOM n'est qu'une spécification qui pour être utilisée doit être implémentée par un éditeur tiers.

Il existe plusieurs versions de DOM nommées «niveaux» :

- DOM Core Level 1 : publiée en 1998, cette spécification contient les bases pour manipuler un document XML (document, élément et noeud)
- DOM Level 2 : publiée en 2000, cette spécifications contient 6 parties (Core, HTML, Events, Style, View and Traversal et Range)
- DOM Level 3 : publiée en 2004



Chaque élément qui compose l'arbre possède un type. Selon ce type, l'élément peut avoir certains éléments fils comme le montre le schéma ci-dessus. Le premier élément est le document encapsulé dans l'interface Document

Toutes les classes et interfaces sont regroupées dans le package org.w3c.dom

Ce chapitre contient plusieurs sections :

- Les interfaces du DOM
- L'obtention d'un arbre DOM
- Le parcours d'un arbre DOM
- La modification d'un arbre DOM
- L'envoi d'un arbre DOM dans un flux

50.1. Les interfaces du DOM

Chaque type d'entité qui compose l'arbre est défini dans une interface. L'interface de base est l'interface Node dont plusieurs autres interfaces héritent.

50.1.1. L'interface Node

Chaque élément de l'arbre est un noeud encapsulé dans l'interface org.w3c.dom.Node ou dans une de ses interfaces filles.

L'interface définit plusieurs méthodes :

Méthode	Rôle
short getNodeType()	Renvoyer le type du noeud
String getNodeName()	Renvoyer le nom du noeud
String getNodeValue()	Renvoyer la valeur du noeud
NamedNodeList getAttributes()	Renvoyer la liste des attributs ou null
void setNodeValue(String)	Mettre à jour la valeur du noeud
boolean hasChildNodes()	Renvoyer un booléen qui indique si le noeud a au moins un noeud fils
Node getFirstChild()	Renvoyer le premier noeud fils du noeud ou null
Node getLastChild()	Renvoyer le dernier noeud fils du noeud ou null
NodeList getChildNodes()	Renvoyer une liste des noeuds fils du noeud ou null
Node getParentNode()	Renvoyer le noeud parent du noeud ou null
Node getPreviousSibling()	Renvoyer le noeud frère précédent
Node getNextSibling()	Renvoyer le noeud frère suivant
Document getOwnerDocument()	Renvoyer le document dans lequel le noeud est inclus
Node insertBefore(Node, Node)	Insèrer le premier noeud fourni en paramètre avant le second noeud
Node replaceNode(Node, Node)	Remplacer le second noeud fourni en paramètre par le premier
Node removeNode(Node)	Supprimer le noeud fourni en paramètre
Node appendChild(Node)	Ajouter le noeud fourni en paramètre aux noeuds enfants du noeud courant
Node cloneNode(boolean)	Renvoyer une copie du noeud. Le booléen fourni en paramètre indique si la copie doit inclure les noeuds enfants

Tous les différents noeuds qui composent l'arbre héritent de cette interface. La méthode getNodeType() permet de connaître le type du noeud. Le type est très important car il permet de savoir ce que contient le noeud.

Le type de noeud peut être :

Constante	Valeur	Rôle

ELEMENT_NODE	1	Elément
ATTRIBUTE_NODE	2	Attribut
TEXT_NODE	3	Texte
CDATA_SECTION_NODE	4	Section de type CData
ENTITY_REFERENCE_NODE	5	Réference d'entité
ENTITY_NODE	6	Entité
PROCESSING_INSTRUCTION_NODE	7	Instruction de traitement
COMMENT_NODE	8	Commentaire
DOCUMENT_NODE	9	Racine du document
DOCUMENT_TYPE_NODE	10	Document
DOCUMENT_FRAGMENT_NODE	11	Fragment de document
NOTATION_NODE	12	Notation

50.1.2. L'interface NodeList

Cette interface définit une liste ordonnée de noeuds suivant l'ordre du document XML. Elle définit deux méthodes :

Méthode	Rôle
int getLength()	Renvoie le nombre de noeuds contenus dans la liste
Node item(int)	Renvoie le noeud dont l'index est fourni en paramètre

50.1.3. L'interface Document

Cette interface définit les caractéristiques pour un objet qui sera la racine de l'arbre DOM. Elle hérite de l'interface Node. Un objet de type Document possède toujours un type de noeud DOCUMENT_NODE.

Méthode	Rôle
DocumentType getDocType()	Renvoyer les informations sur le type de document
Element getDocumentElement()	Renvoyer l'élément racine du document
NodeList getElementsByTagNames(String)	Renvoyer une liste des éléments dont le nom est fourni en paramètre
Attr createAttributes(String)	Créer un attribut dont le nom est fourni en paramètre
CDATASection createCDATASection(String)	Créer un noeud de type CDATA
Comment createComment(String)	Créer un noeud de type commentaire
Element createElement(string)	Créer un noeud de type élément dont le nom est fourni en paramètre

50.1.4. L'interface Element

Cette interface définit des méthodes pour manipuler un élément et en particulier les attributs d'un élément. Un élément dans un document XML correspondant à un tag. L'interface Element hérite de l'interface Node.

Un objet de type Element à toujours pour type de noeud ELEMENT_NODE

Méthode	Rôle
String getAttribute(String)	Renvoyer la valeur de l'attribut dont le nom est fourni en paramètre

removeAttribut(String)	Supprimer l'attribut dont le nom est fourni en paramètre
setAttribut(String, String)	Modifier ou créer un attribut dont le nom est fourni en premier paramètre et la valeur en second
String getTagName()	Renvoyer le nom du tag
Attr getAttributeNode(String)	Renvoyer un objet de type Attr qui encapsule l'attribut dont le nom est fourni en paramètre
Attr removeAttributeNode(Attr)	Supprimer l'attribut fourni en paramètre
Attr setAttributNode(Attr)	Modifier ou créer un attribut
NodeList getElementsByTagName(String)	Renvoyer une liste des noeuds enfants dont le nom correspond au paramètre fourni

50.1.5. L'interface CharacterData

Cette interface définit des méthodes pour manipuler les données de type PCDATA d'un noeud.

Méthode	Rôle
appendData()	Ajouter le texte fourni en paramètre aux données courantes
getData()	Renvoyer les données sous la forme d'une chaîne de caractères
setData()	Permettre d'initialiser les données avec la chaîne de caractères fournie en paramètre

50.1.6. L'interface Attr

Cette interface définit des méthodes pour manipuler les attributs d'un élément.

Les attributs ne sont pas des noeuds dans le modèle DOM. Pour pouvoir les manipuler, il faut utiliser un objet de type Element.

Méthode	Rôle
String getName()	Renvoyer le nom de l'attribut
String getValue()	Renvoyer la valeur de l'attribut
String setValue(String)	Mettre la valeur à celle fournie en paramètre

50.1.7. L'interface Comment

Cette interface permet de caractériser un noeud de type commentaire.

Cette interface étend simplement l'interface CharacterData. Un objet qui implémente cette interface générera un tag de la forme <!-- --> .

50.1.8. L'interface Text

Cette interface permet de caractériser un noeud de type Text. Un tel noeud représente les données d'un tag ou la valeur d'un attribut.

50.2. L'obtention d'un arbre DOM

Pour pouvoir utiliser un arbre DOM représentant un document, il faut utiliser un parseur qui implémente DOM. Ce dernier va parcourir le document XML et créer l'arbre DOM correspondant. Le but est d'obtenir un objet qui implémente l'interface Document car cet objet est le point d'entrée pour toutes les opérations sur l'arbre DOM.

Avant la définition de JAXP par Sun, l'instanciation d'un parseur était spécifique à chaque implémentation.

Exemple : utilisation de Xerces sans JAXP

```
package fr.jmdoudoux.dej.testdom;

import org.apache.xerces.parsers.*;
import org.w3c.dom.*;

public class TestDOM2 {

  public static void main(String[] args) {
    Document document = null;
    DOMParser parser = null;

    try {
      parser = new DOMParser();
      parser.parse("test.xml");
      document = parser.getDocument();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

JAXP permet, si le parseur respecte ses spécifications, de l'instancier de façon normalisée.

Exemple : utilisation de Xerces avec JAXP

```
package fr.jmdoudoux.dej.testdom;

import org.w3c.dom.*;
import javax.xml.parsers.*;

public class TestDOM1 {

  public static void main(String[] args) {
    Document document = null;
    DocumentBuilderFactory factory = null;

    try {
      factory = DocumentBuilderFactory.newInstance();
      DocumentBuilder builder = factory.newDocumentBuilder();
      document = builder.parse("test.xml");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

L'utilisation de JAXP est fortement recommandée.

Remarque : JAXP est détaillé dans une des sections suivantes de ce chapitre.

50.3. Le parcours d'un arbre DOM

Un document XML peut être représenté par une vue à plat ou arborescente. DOM Level 2 propose plusieurs interfaces pour permettre la navigation dans un document XML décrites dans la recommandation Traversal.

Le parcours peut être fait de deux manières différentes :

- une représentation horizontale du document en utilisant l'interface NodeIterator
- une représentation hiériarchique du document en utilisant l'interface TreeWalker

Il est possible d'appliquer un filtre lors des parcours.

Toutes les interfaces sont contenues dans le package org.w3c.dom.traversal.

50.3.1. L'interface DocumentTraversal

L'interface DocumentTraversal est le point de départ pour les objets qui vont permettre le parcours du document XML.

L'interface définit deux méthodes :

Méthode	Rôle
NodeIterator createNodeIterator(Node root, int whatToShow, NodeFilter filter, boolean entityReferenceExpansion)	Renvoyer une instance de type NodeIterator permettant le parcours de la sous-arborescence à partir du noeud fourni
TreeWalker createTreeWalker(Node root, int whatToShow, NodeFilter filter, boolean entityReferenceExpansion)	Renvoyer une instance de type TreeWalker permettant le parcours de la sous-arborescence à partir du noeud fourni

Les deux méthodes attendent les mêmes paramètres :

- Node root : le noeud à partir duquel le parcours commencera. Le NodeIterator est initialement positionné avant le noeud. Ce noeud est toujours inclus lors du parcours avec TreeWalker
- int whatToShow : permet de préciser le ou les types de noeuds qui seront obtenus lors du parcours. Les valeurs possibles sont les constantes dont le nom commence par SHOW_ dans l'interface NodeFilter. Ces valeurs peuvent être combinées avec des opérateurs OR.
- NodeFilter filter : une implémentation de type NodeFilter pour filtrer les noeuds obtenus lors du parcours. Si aucun filtre ne doit être utilisé, il faut passer la valeur null
- boolean entityReferenceExpansion : booléen qui précise si les références d'entités doivent être étendues lors du parcours

La spécification ne précise pas comment obtenir une instance de type DocumentTraversal : dans la plupart des implémentations, l'objet de type document implémente aussi l'interface DocumentTraversal.

50.3.2. L'interface NodeIterator

L'interface NodeIterator définit des méthodes pour parcourir les éléments d'un document XML.

Méthode	Rôle
void detach()	Détacher le NodeIterator de l'ensemble des noeuds qu'il parcourt. L'état du NodeIterator devient invalide
NodeFilter getFilter()	Obtenir le filtre de type NodeFilter
Node getRoot()	Obtenir le noeud initial du parcours
int getWhatToShow()	Obtenir une représentation du ou des types de noeuds qui seront obtenus lors du parcours
Node nextNode()	Renvoyer le noeud suivant dans le parcours
Node previousNode()	Renvoyer le noeud précédent dans le parcours

Le parcours se fait en utilisant un itérateur qui permet d'obtenir chacun des noeuds à tour de rôle.

Exemple :

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.traversal.DocumentTraversal;
import org.w3c.dom.traversal.NodeFilter;
import org.w3c.dom.traversal.NodeIterator;

public class TestNodeIterator {
  public static void main(String[] args) {
    try {
      DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
      DocumentBuilder loader = factory.newDocumentBuilder();
      Document document = loader.parse("livres.xml");
      DocumentTraversal traversal = (DocumentTraversal) document;
      NodeIterator iterator = traversal.createNodeIterator(
          document.getDocumentElement(), NodeFilter.SHOW_ELEMENT, null, true);
      for (Node n = iterator.nextNode(); n != null; n = iterator.nextNode()) {
        System.out.println("Element: " + ((Element) n).getTagName());
      }
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

50.3.3. L'interface TreeWalker

Une instance de type TreeWalker permet de parcourir l'arborescence ou une sous-arborescence d'un document XML. Les noeuds obtenus lors de ce parcours peuvent être filtrés soit sur leur type en utilisant la propriété whatToShow ou en utilisant un filtre personnalisé.

L'interface TreeWalker définit plusieurs méthodes :

Méthode	Rôle
Node firstChild()	Se déplacer sur le prochain noeud fils visible par le parcours et le renvoyer
Node getCurrentNode()	Obtenir le noeud courant dans le parcours
NodeFilter getFilter()	Obtenir le filtre associé
Node getRoot()	Obtenir le noeud initial du parcours
int getWhatToShow()	Obtenir une représentation du ou des types de noeuds qui seront obtenus lors du parcours
Node lastChild()	Se déplacer sur le dernier noeud visible par le parcours et le renvoyer
Node nextNode()	Se déplacer sur le prochain noeud visible par le parcours et le renvoyer
Node nextSibling()	Se déplacer sur le prochain noeud frère visible par le parcours et le renvoyer
Node parentNode()	Se déplacer sur le noeud père visible par le parcours et le renvoyer
Node previousNode()	Se déplacer sur le précédent noeud visible par le parcours et le renvoyer
Node previousSibling()	Se déplacer sur le précédent noeud frère visible par le parcours et le renvoyer
void setCurrentNode(Node currentNode)	Changer le noeud courant dans le parcours pour celui fourni en paramètre

Exemple :

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.traversal.DocumentTraversal;
import org.w3c.dom.traversal.NodeFilter;
import org.w3c.dom.traversal.TreeWalker;

public class TestTreeWalker {
  public static void main(String[] argv) throws Exception {
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    DocumentBuilder loader = factory.newDocumentBuilder();
    Document document = loader.parse("livres.xml");
    DocumentTraversal traversal = (DocumentTraversal) document;
    TreeWalker walker = traversal.createTreeWalker(
        document.getDocumentElement(), NodeFilter.SHOW_ALL, null, true);
    walker.getRoot();
    traverseLevel(walker, "");
  }

  private static final void traverseLevel(TreeWalker walker, String indent) {
    Node noeud = walker.getCurrentNode();
    if (noeud instanceof Element) {
      System.out.println(indent + "- " + ((Element) noeud).getTagName());
      for (Node n = walker.firstChild(); n != null; n = walker.nextSibling()) {
        traverseLevel(walker, indent + "  ");
      }
    }
    walker.setCurrentNode(noeud);
  }
}
```

50.3.4. Le filtrage lors du parcours

L'interface NodeFilter définit des fonctionnalités de filtre pour définir les noeuds qui doivent être obtenus lors du parcours.

L'interface NodeFilter définit une seule méthode : short acceptNode(Node n).

La méthode renvoie une valeur de type short qui permet de préciser si le noeud est filtré ou non. L'interface NodeFilter définit trois constantes :

- FILTER_ACCEPT : le noeud est inclus dans le vue logique fournie par le parcours
- FILTER_REJECT : le noeud est exclu ainsi que ses noeuds fils
- FILTER_SKIP : le noeud est exclu mais les noeuds fils seront inclus dans le parcours

Aucune implémentation standard n'est fournie par l'API DOM.

Une implémentation de l'interface NodeFilter ne connait pas la structure de données parcourue ni la façon dont ces données sont parcourues. La seule qu'elle peut utiliser c'est le noeud fourni en paramètre.

Exemple :

```
import org.w3c.dom.Node;
import org.w3c.dom.traversal.NodeFilter;

public class MonNodeFilter implements NodeFilter {
  @Override
  public short acceptNode(Node n) {
    if (n.getNodeName().contentEquals("auteur")) {
      return FILTER_ACCEPT;
    }
    return FILTER_SKIP;
  }
}
```

Si une instance de type NodeFilter est fournie à un NodeIterator ou à un TreeWalker alors ils appliquent le filtre pour savoir si un noeud doit être retourné ou non. Si le filtre renvoie FILTER_ACCEPT alors le noeud est renvoyé lors du parcours sinon le prochain noeud du parcours est recherché pour lui appliquer le filtre.

Exemple :

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.traversal.DocumentTraversal;
import org.w3c.dom.traversal.NodeFilter;
import org.w3c.dom.traversal.TreeWalker;

public class TestTreeWalkerAvecFiltre {
  public static void main(String[] argv) throws Exception {
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    DocumentBuilder loader = factory.newDocumentBuilder();
    Document document = loader.parse("livres.xml");
    DocumentTraversal traversal = (DocumentTraversal) document;
    TreeWalker walker = traversal.createTreeWalker(
        document.getDocumentElement(), NodeFilter.SHOW_ELEMENT,
        new MonNodeFilter(), true);
    Node noeud = null;
    noeud = walker.nextNode();
    while (noeud != null) {
      System.out.println(noeud.getNodeName() + " : "
          + noeud.getChildNodes().item(0).getNodeValue());
      noeud = walker.nextNode();
    }
  }
}
```

50.4. La modification d'un arbre DOM

Un des grands intérêts du DOM est sa faculté de créer ou modifier l'arbre qui représente un document XML.

50.4.1. La création d'un document

La méthode newDocument() de la classe DocumentBuilder renvoie une nouvelle instance d'un objet de type Document qui encapsule un arbre DOM vide.

Il faut a minima ajouter un tag racine au document XML. Pour cela, il faut appeler la méthode createElement() de l'objet Document en lui passant le nom du tag racine pour obtenir une référence sur le nouveau noeud. Il suffit ensuite d'utiliser la méthode appendChild() de l'objet Document en lui fournissant la référence sur le noeud en paramètre.

Exemple :

```
package fr.jmdoudoux.dej.testdom;

import org.w3c.dom.*;
import javax.xml.parsers.*;

public class TestDOM09 {

  public static void main(String[] args) {
    Document document = null;
    DocumentBuilderFactory fabrique = null;

    try {
      fabrique = DocumentBuilderFactory.newInstance();
      DocumentBuilder builder = fabrique.newDocumentBuilder();
      document = builder.newDocument();
      Element racine = (Element) document.createElement("bibliotheque");
      document.appendChild(racine);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

50.4.2. L'ajout d'un élément

L'interface Document propose plusieurs méthodes createXXX pour créer des instances de différents types d'éléments. Il suffit alors d'utiliser la méthode appendChild() d'un noeud pour lui attacher un noeud fils.

Exemple :

```
     Element monElement = document.createElement("monelement");
     Element monElementFils = document.createElement("monelementfils");
     monElement.appendChild(monElementFils);
```

Pour ajouter un texte à un noeud, il faut utiliser la méthode createTextNode() pour créer un noeud de type Text et l'ajouter au noeud concerné avec la méthode appendChild().

Exemple :

```
     Element monElementFils = document.createElement("monelementfils");
     monElementFils.appendChild(document.createTextNode("texte du tag fils");
     monElement.appendChild(monElementFils);
```

Pour ajouter un attribut à un élément, il existe deux méthodes : setAttributeNode() et setAtribute().

La méthode setAttributeNode() attend un objet de type Attr qu'il faut préalablement instancier.

Exemple :

```
     Attr monAttribut = document.createAttribute("attribut");
     monAttribut.setValue("valeur");
     monElement.setAttributeNode(monAttribut);
```

La méthode setAttribut permet d'associer directement un attribut et sa valeur grâce aux paramètres fournis.

Exemple :

```
    monElement.setAttribut("attribut","valeur");
```

La création d'un commentaire se fait en utilisant la méthode createComment() de la classe Document.

Toutes ces actions permettent la création complète d'un arbre DOM représentant un document XML.

Exemple : un exemple complet

```
package fr.jmdoudoux.dej.testdom;

import org.w3c.dom.*;
import javax.xml.parsers.*;

public class TestDOM11 {

  public static void main(String[] args) {
    Document document = null;
    DocumentBuilderFactory fabrique = null;
    try {
      fabrique = DocumentBuilderFactory.newInstance();
      DocumentBuilder builder = fabrique.newDocumentBuilder();
      document = builder.newDocument();
      Element racine = (Element) document.createElement("bibliotheque");
      document.appendChild(racine);
      Element livre = (Element) document.createElement("livre");
      livre.setAttribute("style", "1");
      Attr attribut = document.createAttribute("type");
      attribut.setValue("broche");
      livre.setAttributeNode(attribut);
      racine.appendChild(livre);
      livre.setAttribute("style", "1");
      Element titre = (Element) document.createElement("titre");
      titre.appendChild(document.createTextNode("Titre 1"));
      livre.appendChild(titre);
      racine.appendChild(document.createComment("mon commentaire"));
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
  <livre style="1" type="broche">
    <titre>Titre 1</titre>
  </livre>
  <!--mon commentaire-->
</bibliotheque>
```

50.5. L'envoi d'un arbre DOM dans un flux

Une fois un arbre DOM créé ou modifié, il est souvent utile de l'envoyer dans un flux (sauvegarde dans un fichier ou une base de données, envoi dans un message JMS ...).

Bizarrement, DOM Level 1 et 2 ne proposent rien pour réaliser cette tâche pourtant essentielle. Ainsi, chaque implémentation propose sa propre méthode en attendant des spécifications qui feront sûrement partie du DOM Level 3.

50.5.1. Un exemple avec Xerces

Xerces fournit la classe XMLSerializer qui permet de créer un document XML à partir d'un arbre DOM.

Xerces est téléchargeable sur le site web https://xerces.apache.org/xerces2-j/ sous la forme d'une archive de type zip qu'il faut décompresser dans un répertoire du système. Il suffit alors d'ajouter les fichiers xmlParserAPIs.jar et xercesImpl.jar dans le classpath.

Exemple :

```
package fr.jmdoudoux.dej.testdom;

import org.w3c.dom.*;
import javax.xml.parsers.*;
import org.apache.xml.serialize.*;

public class TestDOM10 {

  public static void main(String[] args) {
    Document document = null;
    DocumentBuilderFactory fabrique = null;

    try {
      fabrique = DocumentBuilderFactory.newInstance();
      DocumentBuilder builder = fabrique.newDocumentBuilder();
      document = builder.newDocument();
      Element racine = (Element) document.createElement("bibliotheque");
      document.appendChild(racine);

      for (int i = 1; i < 4; i++) {
        Element livre = (Element) document.createElement("livre");
        Element titre = (Element) document.createElement("titre");
        titre.appendChild(document.createTextNode("Titre "+i));
        livre.appendChild(titre);
        Element auteur = (Element) document.createElement("auteur");
        auteur.appendChild(document.createTextNode("Auteur "+i));
        livre.appendChild(auteur);
        Element editeur = (Element) document.createElement("editeur");
        editeur.appendChild(document.createTextNode("Editeur "+i));
        livre.appendChild(editeur);
        racine.appendChild(livre);
      }

      XMLSerializer ser = new XMLSerializer(System.out,
         new OutputFormat("xml", "UTF-8", true));
      ser.serialize(document);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
  <livre>
    <titre>Titre 1</titre>
    <auteur>Auteur 1</auteur>
    <editeur>Editeur 1</editeur>
  </livre>
  <livre>
    <titre>Titre 2</titre>
    <auteur>Auteur 2</auteur>
    <editeur>Editeur 2</editeur>
  </livre>
  <livre>
    <titre>Titre 3</titre>
    <auteur>Auteur 3</auteur>
    <editeur>Editeur 3</editeur>
  </livre>
</bibliotheque>
```

51. XSLT (Extensible Stylesheet Language Transformations)

Chapitre 5 1

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

XSLT est une recommandation du consortium W3C qui permet de transformer facilement des documents XML en d'autres documents standard sans programmation. Le principe est de définir une feuille de style qui indique comment transformer le document XML et de la fournir avec le document à un processeur XSLT.

On peut produire des documents de différents formats : XML, HTML, XHTML, WML, PDF, etc...

XSLT fait partie de XSL avec les recommandations :

- XSL-FO : flow object
- XPath : langage pour spécifier un élément dans un document. Ce langage est utilisé par XSL.

Une feuille de style XSLT est un fichier au format XML qui contient les informations nécessaires au processeur pour effectuer la transformation.

Le composant principal d'une feuille de style XSLT est le template qui définit le moyen de transformer un élément du document XML dans le nouveau document.

XSLT est relativement complet mais complexe : cette section n'est qu'une présentation rapide de quelques fonctionnalités de XSLT. XSLT possède plusieurs fonctionnalités avancées, telles que la sélection des éléments à traiter, le filtrage ou le tri de ces éléments.

Ce chapitre contient plusieurs sections :

♦ XPath
♦ La syntaxe de XSLT
♦ Un exemple avec Internet Explorer
♦ Un exemple avec Xalan 2

51.1. XPath

XML Path ou XPath est une spécification qui fournit une syntaxe pour permettre de sélectionner un ou plusieurs éléments dans un document XML. Il existe sept types d'éléments différents :

- racine (root)
- element
- text
- attribute (attribute)
- commentaire (comment)
- instruction de traitement (processing instruction)
- espace de nommage (name space)

Cette section ne présente que les fonctionnalités de base de XPath.

XPath est utilisé dans plusieurs technologies liées à XML telles que XPointer et XSLT.

Un document XML peut être représenté sous la forme d'un arbre composé de noeuds. XPath grâce à une notation particulière permet de localiser précisément un composant de l'arbre.

La notation reprend une partie de la notation utilisée pour naviguer dans un système d'exploitation, ainsi :

- le séparateur est le caractère slash /
- pour préciser un chemin à partir de la racine (chemin absolu), il faut qu'il commence par un /
- un double point .. permet de préciser l'élément père de l'élément courant
- un simple point . permet de préciser l'élément courant
- un arobase @ permet de préciser un attribut d'un élément
- pour préciser l'indice d'un élément il faut l'écrire entre crochets

XPath permet de filtrer les éléments sur différents critères en plus de leur nom

- @categorie="test" : recherche un attribut dont le nom est categorie et dont la valeur est "test"
- une barre verticale | permet de préciser deux valeurs

51.2. La syntaxe de XSLT

Une feuille de style XSLT est un document au format XML et doit donc respecter toutes les règles d'un tel document. Pour préciser les différentes instructions permettant de réaliser la transformation, un espace de nommage particulier est utilisé : xsl. Tous les tags de XSLT commencent donc par ce préfixe, ainsi le tag racine du document est xsl:stylesheet. Ce tag racine doit obligatoirement posséder un attribut version qui précise la version de XSLT utilisée.

Exemple : une feuille de style minimale qui ne fait rien

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

</xsl:stylesheet>
```

Le tag xsl:output permet de préciser le format de sortie. Ce tag possède plusieurs attributs :

- method : cet attribut permet de préciser le format. Les valeurs possibles sont : texte, xml ou html
- indent : cet attribut permet de définir si la sortie doit être indentée ou non. Les valeurs possibles sont : yes ou no
- encoding : cet attribut permet de préciser le jeu de caractères utilisé pour la sortie

Pour effectuer la transformation, le document doit contenir des règles. Ces règles suivent une syntaxe particulière et sont contenues dans des modèles (templates) associés à un ou plusieurs éléments désignés avec un motif au format XPath.

Un modèle est défini grâce au tag xsl:template. La valeur de l'attribut match permet de fournir le motif au format XPath qui sélectionnera le ou les éléments sur lesquels le modèle va agir.

Le tag xsl:apply-templates permet de demander le traitements des autres modèles définis pour chacun des noeuds fils du noeud courant.

Le tag xsl:value-of permet d'extraire la valeur de l'élément respectant le motif XPath fourni avec l'attribut select.

Il existe beaucoup d'autres tags notamment plusieurs qui permettent d'utiliser des structures de contrôles de types itératifs ou conditionnels.

Le tag xsl:for-each permet de parcourir un ensemble d'éléments sélectionnés par l'attribut select. Le modèle sera appliqué sur chacun des éléments de la liste.

Le tag xsl:if permet d'exécuter le modèle si la condition précisée par l'attribut test au format XPath est juste. XSLT ne définit pas de tag équivalent à la partie else : il faut définir un autre tag xsl:if avec une condition opposée.

Le tag xsl:choose permet de définir plusieurs conditions. Chaque condition est précisée grâce à l'attribut xsl:when avec l'attribut test. Le tag xsl:otherwise permet de définir un cas par défaut qui ne correspond pas aux autres cas définis dans le tag xsl:choose.

Le tag xsl:sort permet de trier un ensemble d'éléments. L'attribut select permet de préciser les éléments qui doivent être triés. L'attribut data-type permet de préciser le format des données (text ou number). l'attribut order permet de préciser l'ordre de tri (ascending ou descending).

51.3. Un exemple avec Internet Explorer

Le plus simple pour tester une feuille XSLT qui génère une page HTML est de la tester avec Internet Explorer version 6. Cette version est entièrement compatible avec XML et XSLT. Les versions 5 et 5.5 ne sont que partiellement compatibles. Les versions antérieures ne le sont pas du tout.

51.4. Un exemple avec Xalan 2

Xalan 2 utilise l'API JAXP.

Exemple : TestXSL2.java

```
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
import org.xml.sax.*;
import java.io.IOException;

public class TestXSL2
{
  public static void main(String[] args)
        throws TransformerException, TransformerConfigurationException,
         SAXException, IOException
        {

    TransformerFactory tFactory = TransformerFactory.newInstance();
    Transformer transformer = tFactory.newTransformer(new StreamSource("test.xsl"));

    transformer.transform(new StreamSource("test.xml"), new StreamResult("test.htm"));
  }
}
```

Exemple : la feuille de style XSL test.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns="http://www.w3.org/TR/REC-html40">

<xsl:output method="html" indent="no" />
<xsl:strip-space elements="*" />

<xsl:template match="/">
<HTML>
<HEAD>
<TITLE>Test avec XSL</TITLE>
</HEAD>
<xsl:apply-templates />
</HTML>
</xsl:template>

<xsl:template match="BIBLIOTHEQUE">
<BODY>
<H1>Liste des livres</H1>
<TABLE border="1" cellpadding="4">
<TR><TD>Titre</TD><TD>Auteur</TD><TD>Editeur</TD></TR>
<xsl:apply-templates />
</TABLE>
```

```
</BODY>
</xsl:template>

<xsl:template match="LIVRE">
<TR>
<TD><xsl:apply-templates select="TITRE"/></TD>
<TD><xsl:apply-templates select="AUTEUR"/></TD>
<TD><xsl:apply-templates select="EDITEUR"/></TD>
</TR>
</xsl:template>
</xsl:stylesheet>
```

Exemple : compilation et exécution avec Xalan

```
javac TestXSL2.java -classpath .;xalan2.jar
java -cp .;xalan2.jar TestXSL2
```

52. Les modèles de documents

Chapitre 52

Niveau : 🟩🟩🟨⬜⬜⬜ Intermédiaire

52.1. L'API JDOM

JDOM est une API open source Java dont le but est de représenter et manipuler un document XML de manière intuitive pour un développeur Java sans requérir une connaissance pointue de XML. Par exemple, JDOM utilise des classes plutôt que des interfaces. Ainsi pour créer un nouvel élément, il faut simplement instancier une classe.

Malgré la similitude de nom entre JDOM et DOM, ces deux API sont très différentes. JDOM est une API uniquement Java car elle s'appuie sur un ensemble de classes de l'API Java notamment celles de l'API Collection.

Le site officiel de l'API est à l'url http://www.jdom.org/

Ce chapitre contient plusieurs sections :

- L'API JDOM
- dom4j

52.1.1. L'historique de JDOM

En 2000, Brett McLaughlin et Jason Hunter développent une nouvelle API dédiée aux traitements de documents XML en Java. Le but est de fournir une API plus conviviale à utiliser en Java que SAX ou DOM.

L'historique de JDOM est marquée par plusieurs versions bêta et stables :

- La version bêta 3 est diffusée en avril 2000.
- La version 1.0 a été publiée en septembre 2004.
- La version 1.1 a été publiée en novembre 2007.

JDOM a fait l'objet d'une spécification sous la Java Specification Request numéro 102 (JSR-102) : malheureusement celle-ci n'a pas abouti.

52.1.2. La présentation de JDOM

Le but de JDOM n'est pas de définir un nouveau type de parseur mais de faciliter la manipulation au sens large de document XML : lecture d'un document, représentation sous forme d'arborescence, manipulation de cet arbre, définition d'un nouveau document, exportation vers plusieurs formats cibles ...

Dans le rôle de manipulation sous forme d'arbre, JDOM possède moins de fonctionnalités que DOM mais en contrepartie il offre une plus grande facilité pour répondre aux cas les plus classiques d'utilisation.

Cette facilité d'utilisation de JDOM lui permet d'être une API dont l'utilisation est assez répandue.

JDOM est donc un modèle de documents objets open source dédié à Java pour encapsuler un document XML. JDOM propose aussi une intégration de SAX, DOM, XSLT et XPath.

JDOM n'est pas un parseur : il a d'ailleurs besoin d'un parseur externe de type SAX ou DOM pour analyser un document et créer la hiérarchie d'objets relative à un document XML. L'utilisation d'un parseur de type SAX est recommandée car elle consomme moins de ressources que DOM pour cette opération. Par défaut, JDOM utilise le parseur défini par JAXP.

Un document XML est encapsulé dans un objet de type Document qui peut contenir des objets de type Comment, ProcessingInstruction et l'élément racine du document encapsulé dans un objet de type Element.

Les éléments d'un document sont encapsulés dans des classes dédiées : Element, Attribute, Text, ProcessingInstruction, Namespace, Comment, DocType, EntityRef, CDATA.

Un objet de type Element peut contenir des objets de type Comment, Text et d'autres objets de type Element.

A l'exception des objets de type Namespace, les éléments sont créés en utilisant leur constructeur.

JDOM vérifie que les données contenues dans les éléments respectent la norme XML : par exemple, il n'est pas possible de créer un commentaire contenant deux caractères moins qui se suivent.

Une fois un document XML encapsulé dans un arbre d'objets, il est possible de modifier cet arbre dans le respect des spécifications de XML.

JDOM permet d'exporter un arbre d'objets d'un document XML dans un flux, un arbre DOM ou un ensemble d'événements SAX.

JDOM interagit donc avec SAX et DOM pour créer un document en utilisant ces parseurs ou pour exporter un document vers ces API, ce qui permet de facilement intégrer JDOM dans des traitements existants. JDOM propose cependant sa propre API.

52.1.3. Les fonctionnalités et les caractéristiques

JDOM propose plusieurs fonctionnalités :

- Création de documents XML
- Encapsulation d'un document XML sous la forme d'objets Java de l'API
- Exportation d'un document dans un fichier, un flux SAX ou un arbre DOM
- Support de XSLT
- Support de XPath

Les points caractéristiques de l'API JDOM sont :

- elle est développée spécifiquement en et pour Java en utilisant les fonctionnalités de Java au niveau syntaxique et sémantique (utilisation des collections de Java 2, de l'opérateur new pour instancier des éléments, redéfinition des méthodes equals(), hashCode(), toString(), implémentation des interfaces Cloneable et Serializable, ...)
- elle se veut intuitive et productive notamment grâce à des classes dédiées à chaque élément instancié par son constructeur et l'utilisation de getter/setter
Exemple pour obtenir le texte d'un élément
DOM : String content = element.getFirstChild().getValue();
JDOM : String text = element.getText();
- elle se veut rapide et légère
- elle veut masquer la complexité de certains aspects de XML tout en respectant ses spécifications
- elle doit permettre les interactions entre SAX et DOM. JDOM peut encapsuler un document XML dans une hiérarchie d'objets à partir d'un flux, d'un arbre DOM ou d'événements SAX. Il est aussi capable d'exporter un document dans ces différents formats.

Il est légitime de se demander qu'elle est l'utilité de proposer une nouvelle API pour manipuler des documents XML en Java alors que plusieurs standards existent déjà. En fait le besoin est réel car JDOM propose des réponses à certaines faiblesses de SAX et DOM.

DOM est une API indépendante de tout langage : son implémentation en Java ne tient donc pas compte des spécificités et standards de Java ce qui rend sa mise en oeuvre peu aisée. JDOM est plus intuitif et facile à mettre en oeuvre que DOM.

Comme DOM, JDOM encapsule un document XML entier dans un arbre d'objets. Par contre chaque élément du document est encapsulé dans une classe dédiée selon son type et non sous la forme d'un objet de type Node.

JDOM peut être utilisé comme une alternative à DOM pour manipuler un document XML. JDOM ne remplace pas DOM puisque ce n'est pas un parseur, de plus il propose des interactions avec DOM en entrée et en sortie.

L'utilisation de DOM requiert de nombreuses ressources notamment à cause de son API qui de surcroît n'est pas intuitive en Java. DOM est développé de façon indépendante de tout langage et son organisation est proche de celle des spécifications XML (tous les éléments sont des Nodes par exemple).

SAX est particulièrement bien adapté à la lecture rapide avec peu de ressources d'un document XML mais son modèle de traitement par événements n'est pas intuitif et surtout SAX ne permet pas de modifier ni de naviguer dans un document.

JDOM propose d'apporter une solution à ces différents problèmes dans une seule et même API.

Afin de les rendre plus clairs, la plupart des exemples de ce chapitre héritent de la classe ci-dessous qui propose un moyen pour exporter un document vers la console.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.Document;
import org.jdom.output.Format;
import org.jdom.output.XMLOutputter;

public abstract class TestJDOM {

  protected static void afficher(Document document)
  {
    try
    {
      XMLOutputter sortie = new XMLOutputter(Format.getPrettyFormat());
      sortie.output(document, System.out);
    } catch (java.io.IOException e){}
  }
}
```

52.1.4. L'installation de JDOM

Avant la version 1.0 de JDOM, il était nécessaire de compiler l'API avec un script Ant fourni.

A partir de la version 1.0, l'archive de JDOM contient directement un binaire de la bibliothèque utilisable.

52.1.4.1. L'installation de JDOM version 1 bêta 7 sous Windows

Pour utiliser JDOM il faut construire la bibliothèque grâce à l'outil Ant. Ant doit donc être installé sur la machine.

Il faut aussi que la valeur de la variable JAVA_HOME soit définie avec le répertoire qui contient le JDK.

Exemple :

```
set JAVA_HOME=c:\j2sdk1.4.0-rc
```

Il suffit alors d'exécuter le fichier build.bat situé dans le répertoire d'installation de JDom.

Un message informe de la fin de la compilation :

Exemple :

```
package:

[jar] Building jar: C:\java\jdom-b7\build\jdom.jar
BUILD SUCCESSFUL
Total time: 1 minutes 33 seconds
```

Le fichier jdom.jar est créé dans le répertoire build.

Pour utiliser JDOM dans un projet, il faut obligatoirement avoir un parseur XML SAX et/ou DOM. Pour les exemples de cette section, lorsqu'aucun parser n'est fourni avec le JDK, c'est Xerces qui est utilisé. Il faut aussi avoir le fichier jaxp.jar.

Pour compiler et exécuter les exemples de cette section, j'ai utilisé le script suivant :

Exemple :

```
javac %1.java -classpath .;jdom.jar;xerces.jar;jaxp.jar
java -classpath .;jdom.jar;xerces.jar;jaxp.jar %1
```

52.1.4.2. L'installation de la version 1.x

L'archive contenant JDOM peut être téléchargée à l'url http://www.jdom.org/dist/binary/

Il suffit de décompresser le contenu de l'archive dans un répertoire du système et d'ajouter le fichier jdom.jar contenu dans le sous-répertoire build au classpath.

52.1.5. Les différentes entités de JDOM

Pour traiter un document XML, JDOM définit plusieurs entités qui peuvent être regroupées en trois groupes :

- les éléments de l'arbre
    - le document : la classe Document
    - les éléments : la classe Element
    - les commentaires : la classe Comment
    - les attributs : la classe Attribute
    - etc ...
- les entités pour obtenir un parseur :
    - les classes SAXBuilder et DOMBuilder
- les entités pour produire un document
    - les classes XMLOutputter, SAXOutputter, DOMOutputter

Ces classes sont regroupées dans cinq packages :

- org.jdom
- org.jdom.adapters
- org.jdom.input
- org.jdom.output
- org.jdom.transform

Attention : cette API a énormément évoluée jusqu'à sa version 1.0. Beaucoup de méthodes ont été déclarées deprecated au fur et à mesure des différentes versions bêta.

52.1.5.1. La classe Document

La classe org.jdom.Document encapsule l'arbre dans lequel JDOM stocke le document XML. Pour obtenir un objet Document, il y a deux possibilités :

- utiliser un objet XXXBuilder qui va parser un document XML existant et créer l'objet Document en utilisant un parseur
- instancier un nouvel objet Document pour créer un nouveau document XML

Pour créer un nouveau document, il suffit d'instancier un objet Document en utilisant un des constructeurs fournis dont les principaux sont :

Constructeur	Rôle
Document()	
Document(Element)	Création d'un document avec l'élément racine fourni
Document(Element, DocType)	Création d'un document avec l'élément racine et la déclaration doctype fournie
Document(List)	Création d'un document avec les entités fournies (élément racine, commentaires, instructions de traitement)
Document(List, DocType)	Création d'un document avec les entités et le type de document fournis

Exemple :

```
import org.jdom.*;

public class TestJDOM2 {

  public static void main(String[] args) {
    Element racine = new Element("bibliothèque");
    Document document = new Document(racine);
  }
}
```

La classe Document possède plusieurs méthodes dont les principales sont :

Méthode	Rôle
Document addContent(Comment)	Ajouter un commentaire au document
List getContent()	Renvoyer un objet List qui contient chaque élément du document
DocType getDocType()	Renvoyer un objet contenant les caractéristiques Doctype du document
Document setDocType()	Définir le DocType du document
Element getRootElement()	Renvoyer l'élément racine du document
Document setRootElement(Element)	Définir l'élément racine du document
boolean hasRootElement()	Renvoyer un booléen qui indique si le document possède un élément racine
Element detachRootElement()	Détache l'élément racine du document
Document addContent(ProcessingInstruction)	Ajouter une instruction de traitement
Document addContent(Comment)	Ajouter un commentaire
Document removeContent(ProcessingInstruction)	Supprimer une instruction de traitement
Document removeContent(Comment)	Supprimer un commentaire

Un objet de type Document possède :

- Un élément racine (RootElement)
- Un objet de type DocType facultatif
- Une collection des éléments rattachés au document : l'élément racine et éventuellement des instructions de traitement et des commentaires.

Un document peut ne pas avoir d'élément racine, lorsqu'il est créé avec le constructeur par défaut mais dans ce cas, le document n'est utilisable qu'à partir du moment où l'élément racine lui est ajouté.

Pour obtenir un document à partir d'un document XML existant, JDOM propose les classes SAXBuilder et DOMbuilder du package org.jdom.input.

52.1.5.2. La classe DocType

JDOM n'offre pas un modèle complet d'objets pour le support des DTD : seule la classe DocType est proposée pour encapsuler la déclaration d'un type document.

Pour instancier un objet de type DocType, il suffit d'utiliser un de ses constructeurs.

L'association de la DTD au document peut se faire en utilisant le constructeur de la classe Document qui attend un tel objet en paramètre ou en utilisant la méthode setDocType() de la classe Document.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.DocType;
import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM4 extends TestJDOM {

  public static void main(String[] args) {

    Element racine = new Element("bibliotheque");
    DocType docType = new DocType("bibliotheque", "bibliotheque.dtd");
    Document document = new Document(racine, docType);

    afficher(document);
  }
}
```

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE bibliotheque SYSTEM "bibliotheque.dtd">

<bibliotheque />
```

La classe DocType n'encapsule que des données informatives sur la DTD dans 4 propriétés :

- root element name
- internal DTD subset
- system ID
- public ID

Exemple de déclaration :

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

Dans cet exemple :

- root element name : html
- system ID : http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd
- public ID : -//W3C//DTD XHTML 1.0 Transitional//EN

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.DocType;
import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM6 extends TestJDOM {

  public static void main(String[] args) {

    Element racine = new Element("html");
    DocType docType = new DocType("html", "-//W3C//DTD XHTML 1.0 Transitional//EN",
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd");

    Document document = new Document(racine, docType);

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html />
```

JDOM permet de déclarer une DTD mais ne l'utilise en aucune façon pour assurer la validité du document. JDOM vérifie simplement que le document est bien formé.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.DocType;
import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM7 extends TestJDOM {

  public static void main(String[] args) {

    Element racine = new Element("test");
    DocType docType = new DocType("html", "-//W3C//DTD XHTML 1.0 Transitional//EN",
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd");

    Document document = new Document(racine, docType);

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<test />
```

Pour définir une DTD directement dans la déclaration, il faut utiliser la méthode setInternalSubset() en lui passant en paramètre la chaîne de caractères contenant la DTD.

Remarque : JDOM ne permet pas de valider un document en mémoire.

Pour vérifier la validité d'un document, il faut exporter le document et utiliser un parseur en activant l'option de validation.

52.1.5.3. La classe Element

La structure d'un document XML est composée d'éléments encapsulés dans la classe org.jdom.Element.

Un élément peut contenir du texte, des attributs, des commentaires et tous les autres éléments définis par la norme XML.

Cette classe possède plusieurs constructeurs dont les principaux sont :

Constructeur	Rôle
Element(String)	Créer un élément, en précisant son nom
Element(String, Namespace)	Créer un élément, en précisant son nom et son espace de nommage
Element(String, String)	Créer un objet, en précisant son nom et l'URI de son espace de nommage
Element(String, String, String)	Créer un objet, en précisant son nom, le préfixe et l'URI de son espace de nommage

La classe Element possède plusieurs propriétés :

- Name : le nom de l'élément
- Namespace : l'espace de nommage de l'élément. Il y a toujours un objet de type Namespace pour chaque élément : si l'élément ne possède pas d'espace de nommage alors la propriété vaut Namespace.NO_NAMESPACE
- Content : collection de type List des éléments fils de l'élément
- Parent : élément père de l'élément ; peut être null si l'élément est l'élément racine ou si l'élément n'est pas ajouté dans un document. Cette propriété ne peut être modifiée directement. Elle est modifiée par la méthode addContent() lors de l'association de l'élément à son père. Cette association n'est possible que pour un élément qui n'a pas déjà un père
- Document : document qui contient cet élément ; peut être null si l'élément n'est pas ajouté à un document
- Attributes : une collection de type List qui encapsule les attributs de l'élément

La classe Element possède de nombreuses méthodes pour obtenir, ajouter ou supprimer une entité de l'élément (un élément enfant, le texte, un attribut, un commentaire, ...) :

Méthode	Rôle
Element addContent(Comment)	Ajouter un commentaire à l'élément
Element addContent(Element)	Ajouter un élément fils à l'élément
Element addContent(String text)	Ajouter des données sous forme de texte à l'élément
Attribute getAttribute(String)	Renvoyer l'attribut dont le nom est fourni en paramètre

Attribute getAttribute(String, Namespace)	Renvoyer l'attribut dont le nom et l'espace de nommage sont fournis en paramètres
List getAttributes()	Renvoyer une collection qui contient tous les attributs
String getAttributeValue(String)	Renvoyer la valeur de l'attribut dont le nom est fourni en paramètres
String getAttributeValue(String, Namespace)	Renvoyer la valeur de l'attribut dont le nom et l'espace de nommage sont fournis en paramètres
Element getChild(String)	Renvoyer le premier élément enfant dont le nom est fourni en paramètre
Element getChild(String, Namespace)	Renvoyer le premier élément enfant dont le nom et l'espace de nommage sont fournis en paramètres
List getChildren()	Renvoyer une liste qui contient tous les éléments enfants directement rattachés à l'élément
List getChildren(String)	Renvoyer une liste qui contient tous les éléments enfants directement rattachés à l'élément dont le nom est fourni en paramètre
List getChildren(String, Namespace)	Renvoyer une liste qui contient tous les éléments enfants directement rattachés à l'élément dont le nom et l'espace de nommage sont fournis en paramètre
String getChildText(String name)	Renvoyer le texte du premier élément enfant dont le nom est fourni en paramètre
String getChildText(String, Namespace)	Renvoyer le texte du premier élément enfant dont le nom et l'espace de nommage sont fournis en paramètres
List getContent()	Renvoyer une liste qui contient toutes les entités de l'élément (texte, élément, commentaire ...)
Document getDocument()	Renvoyer l'objet Document qui contient l'élément
Element getParent()	Renvoyer l'élément père de l'élément
String getText()	Renvoyer les données au format texte contenues dans l'élément
boolean hasChildren()	Renvoyer un booléen qui indique si l'élément possède des éléments fils
boolean isRootElement()	Renvoyer un booléen qui indique si l'élément est l'élément racine du document
boolean removeAttribute(String)	Supprimer l'attribut dont le nom est fourni en paramètre
boolean removeAttribute(String, Namespace)	Supprimer l'attribut dont le nom et l'espace de nommage sont fournis en paramètres
boolean removeChild(String)	Supprimer l'élément enfant dont le nom est fourni en paramètre
boolean removeChild(String, Namespace)	Supprimer l'élément enfant dont le nom et l'espace de nommage sont fournis en paramètres
boolean removeChildren()	Supprimer tous les éléments enfants
boolean removeChildren(String)	Supprimer tous les éléments enfants dont le nom est fourni en paramètre
boolean removeChildren(String, Namespace)	Supprimer tous les éléments enfants dont le nom et l'espace de nommage sont fournis en paramètres
boolean removeContent(Comment)	Supprimer le commentaire fourni en paramètre
boolean removeContent(Element)	Supprimer l'élément fourni en paramètre
Element setAttribute(Attribute)	Ajouter un attribut
Element setAttribute(String, String)	Ajouter un attribut dont le nom et la valeur sont fournis en paramètres
Element setAttribute(String, String, Namespace)	Ajouter un attribut dont le nom, la valeur et l'espace de nommage sont fournis en paramètres

Element setText(String)	Mettre à jour les données au format texte de l'élément

Pour obtenir l'élément racine d'un document, il faut utiliser la méthode getRootElement() de la classe Document. Celle-ci renvoie un objet de type Element. Il est ainsi possible d'utiliser les méthodes ci-dessus pour parcourir et modifier le contenu du document.

L'utilisation de la classe Elément est très facile.

Pour créer un élément, il suffit d'instancier un objet de type Element.

Exemple :

```
          Element element = new Element("element1");
          element.setAttribute("attribut1","valeur1");
          element.setAttribute("attribut2","valeur2");
```

La classe possède plusieurs méthodes pour obtenir les entités de l'élément, un élément fils particulier ou une liste d'élément fils. Des appels successifs à ces méthodes permettent d'obtenir un élément précis du document.

Les méthodes de type setter, qui devraient par convention ne rien retourner (void), renvoient l'instance de type Element elle-même. Ceci permet de chaîner les appels aux différents setters.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM28 extends TestJDOM {

  public static void main(String[] args) {

    Element racine = new Element("bibliotheque");
    Document document = new Document(racine);
    Element livres = new Element("livres");
    racine.addContent(livres);
    livres.addContent(new Element("livre")
      .addContent(new Element("titre").setText("Titre livre 1"))
      .addContent(new Element("auteur").setText("Auteur 1"))
    );

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
  <livres>
    <livre>
      <titre>Titre livre 1</titre>
      <auteur>Auteur 1</auteur>
    </livre>
  </livres>
</bibliotheque>
```

52.1.5.4. La classe Attribut

La classe org.jdom.Attribut encapsule un attribut d'un élément.

La classe Attribut possède plusieurs propriétés :

- name : le nom de l'attribut
- namespace : l'espace de nommage
- value : la valeur de l'attribut
- parent : l'élément qui contient l'attribut
- Type : le type de l'attribut (par défaut Attribute.UNDECLARED_ATTRIBUTE)

La classe Element propose les méthodes getAttribute() et getAttributeValue() pour obtenir un attribut ou la valeur d'un attribut sous la forme d'un objet de type String. La méthode getAttribute() renvoie null si l'attribut n'existe pas.

La classe Attribute propose plusieurs méthodes getXXXValue() qui tentent de fournir la valeur de l'attribut dans le type primitif XXX. Si la conversion échoue, alors une exception de type org.jdom.DataConversionException est levée.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Attribute;
import org.jdom.DataConversionException;
import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM21 extends TestJDOM {

  public static void main(String[] args) {

    try {
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File("test.xml"));

      afficher(document);

      Element table = document.getRootElement().getChild("body").getChild("table");
      System.out.println("attribut width : " + table.getAttributeValue("width"));

      System.out.println("");
      Attribute border = table.getAttribute("border");
      System.out.println("attribut " + border.getName() + " : " + border.getValue());
      System.out.println("attribut " + border.getName() + " : " + border.getIntValue());

      System.out.println("");
      Attribute width = table.getAttribute("width");
      try {
        System.out.println("attribut " + width.getName()
          + " : " + width.getIntValue());
      } catch (DataConversionException dce) {
        System.out.println("attribut " + width.getName()
          + " : impossible d'obtenir une valeur entière");
      }

      System.out.println("");
      Attribute cellspacing = table.getAttribute("cellspacing");
      if (cellspacing == null) {
        System.out.println("l'attribut cellspacing n'existe pas");
      }
    } catch (JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<html>
  <head />
  <body>
    <table width="100%" border="0" />
  </body>
</html>

attribut width : 100%

attribut border : 0
attribut border : 0

attribut width : impossible d'obtenir une valeur entière

l'attribut cellspacing n'existe pas
```

L'association d'un attribut à un élément ou sa suppression se fait généralement en utilisant les méthodes setAttribute() et removeAttribute() de la classe Element plutôt qu'en manipulant des instances de la classe Attribute.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.Attribute;
import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM22 extends TestJDOM {

  public static void main(String[] args) {

    Element racine = new Element("html");
    Document document = new Document(racine);
    Element body = new Element("body");
    racine.addContent(body);
    Element table = new Element("table");
    body.addContent(table);
    table.setAttribute("width","100%");
    table.setAttribute(new Attribute("border","0"));
    table.setAttribute("cellspacing","10");
    table.removeAttribute("cellspacing");

    afficher(document);
  }
}
```

Résultat :

```
 <?xml version="1.0" encoding="UTF-8"?>
<html>
  <body>
    <table width="100%" border="0" />
  </body>
</html>
```

JDom vérifie les valeurs fournies pour les propriétés Name et Value.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM24 extends TestJDOM {

  public static void main(String[] args) {
    Element racine = new Element("html");
    Document document = new Document(racine);
    Element body = new Element("body");
    racine.addContent(body);
    Element table = new Element("table");
    body.addContent(table);
    table.setAttribute("@valeur","100");

    afficher(document);
  }
}
```

Résultat :

```
Exception in thread "main" org.jdom.IllegalNameException:
The name "@valeur" is not legal for JDOM/XML attributes:
XML names cannot begin with the character "@".
        at org.jdom.Attribute.setName(Attribute.java:363)
        at org.jdom.Attribute.<init>(Attribute.java:227)
        at org.jdom.Attribute.<init>(Attribute.java:251)
        at org.jdom.Element.setAttribute(Element.java:1128)
        at fr.jmdoudoux.dej.jdom.TestJDOM24.main(TestJDOM24.java:21)
```

La classe Element propose la méthode getAttributes() qui renvoie une collection d'objets de type Attribute contenant tous les attributs de l'élément.

Pour supprimer tous les attributs d'un élément, il suffit d'utiliser la méthode clear() sur la collection.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.util.List;
import java.util.ListIterator;

import org.jdom.Attribute;
import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM23 extends TestJDOM {

  public static void main(String[] args) {

    Element racine = new Element("html");
    Document document = new Document(racine);
    Element body = new Element("body");
    racine.addContent(body);
    Element table = new Element("table");
    body.addContent(table);
    table.setAttribute("width","100%");
    table.setAttribute(new Attribute("border","0"));
    table.setAttribute("cellspacing","10");

    List attributs = table.getAttributes();
    ListIterator iterator = attributs.listIterator();

    System.out.println("Liste des attributs");
    while (iterator.hasNext()) {
      Attribute attribut = (Attribute) iterator.next();
      System.out.println("attribut "+attribut.getName()+" : "+attribut.getValue());
    }

    System.out.println();
    // supprimer tous les attributs
    attributs.clear();

    afficher(document);
  }
}
```

Résultat :

```
Liste des attributs
attribut width : 100%
attribut border : 0
attribut cellspacing : 10

<?xml version="1.0" encoding="UTF-8"?>
<html>
  <body>
    <table />
  </body>
</html>
```

Un objet de type Attribut ne peut avoir qu'un seul élément parent.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.Attribute;
import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM25 extends TestJDOM {

  public static void main(String[] args) {

    Element racine = new Element("html");
    Document document = new Document(racine);
    Element body = new Element("body");
    racine.addContent(body);
    Element table = new Element("table");
    body.addContent(table);
    Attribute attribut = new Attribute("width","100%");
    table.setAttribute(attribut);
    body.setAttribute(attribut);

    afficher(document);
  }
}
```

Résultat :

```
Exception in thread "main" org.jdom.IllegalAddException:
The attribute already has an existing parent "table"
        at org.jdom.AttributeList.add(AttributeList.java:187)
        at org.jdom.AttributeList.add(AttributeList.java:131)
        at org.jdom.Element.setAttribute(Element.java:1181)
        at fr.jmdoudoux.dej.jdom.TestJDOM25.main(TestJDOM25.java:23)
```

Pour déplacer un attribut vers un autre élément, il faut au préalable utiliser la méthode detach().

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.Attribute;
import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM26 extends TestJDOM {
  public static void main(String[] args) {

    Element racine = new Element("html");
    Document document = new Document(racine);
    Element body = new Element("body");
    racine.addContent(body);
    Element table = new Element("table");
    body.addContent(table);
    Attribute attribut = new Attribute("width","100%");
    table.setAttribute(attribut);
    attribut.detach();
    body.setAttribute(attribut);

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<html>
  <body width="100%">
    <table />
  </body>
</html>
```

52.1.5.5. La classe Text

JDOM encapsule un noeud de type texte du document XML dans la classe Text.

Généralement, cette classe n'est pas utilisée directement car JDOM effectue directement les conversions vers String sauf lors du parcours des éléments fils retournés par la méthode getContent().

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.util.List;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.output.Format;
import org.jdom.output.XMLOutputter;

public class TestJDOM15 {

  public static void main(String[] args) {

    Element racine = new Element("bibliotheque");
    Document document = new Document(racine);
    Element adresse = new Element("adresse");
    adresse.addContent("mon adresse");
    racine.addContent(adresse);

    List elements = adresse.getContent();
    for (Object element : elements) {
      System.out.println(element.getClass().getName());
    }
  }
}
```

Résultat :

```
org.jdom.Text
```

La classe Texte stocke les caractères dans un champ value auquel il est possible d'accéder par la propriété Text (getText() et setText()).

Il n'est pas utile d'échapper les caractères utilisés par XML(<, >, &, ...). Il suffit de les fournir tels quels et JDOM les échappera lors de l'exportation du document.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM16 extends TestJDOM {

  public static void main(String[] args) {

    Element racine = new Element("bibliotheque");
    Document document = new Document(racine);
    Element adresse = new Element("adresse");
    adresse.addContent("mon adresse < 5 et > 10 & impaire ");
    racine.addContent(adresse);

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
  <adresse>mon adresse &lt; 5 et &gt; 10 &amp; impaire</adresse>
</bibliotheque>
```

La classe Text possède de nombreuses méthodes

Méthode	Rôle
String getText()	Obtenir la valeur du texte
String getTextTrim()	
String getTextNormalize()	
String normalizeString(String)	
Text setText(String)	Modifier la valeur du texte
void append(String)	Ajouter la chaîne de caractères à la valeur du texte
void append(Text)	Ajoute la valeur du texte de l'objet fourni à la valeur du texte
Element getParent();	Obtenir l'élément qui contient le texte
Document getDocument()	Obtenir le document qui contient le texte
Text setParent(Element)	Associer le texte à son élément parent
Text detach()	Détacher le texte de son élément père

La classe Text possède plusieurs propriétés :

- Value : la valeur du texte
- Parent : l'élément parent
- Document : le document qui contient l'objet

JDOM ne garantit pas que le contenu textuel d'un élément soit stocké dans un unique objet Text.

52.1.5.6. La classe Comment

La classe org.jdom.Comment encapsule un commentaire dans le document.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.Comment;
import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM18 extends TestJDOM {

  public static void main(String[] args) {

    Element racine = new Element("bibliotheque");
    Comment comment1 = new Comment("mon commentaire bibliotheque");
    racine.addContent(comment1);
    Document document = new Document(racine);
    Element adresse = new Element("adresse");
    Comment comment2 = new Comment("mon commentaire adresse");
    adresse.addContent(comment2);
    adresse.addContent("mon adresse");

    racine.addContent(adresse);

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
  <!--mon commentaire bibliotheque-->
  <adresse>
    <!--mon commentaire adresse-->
    mon adresse
  </adresse>
</bibliotheque>
```

Il est possible d'ajouter un commentaire directement au document :

- pour ajouter un commentaire à la fin, il suffit d'utiliser la méthode addContent de la classe Document
- pour ajouter un commentaire au début du document (entre le prologue et le tag racine), il faut obtenir la liste des éléments du document grâce à la méthode getContent() et ajouter le commentaire à la première position de la collection

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.util.List;

import org.jdom.Comment;
import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM19 extends TestJDOM {

  public static void main(String[] args) {

    Element racine = new Element("bibliotheque");
    Document document = new Document(racine);
    document.addContent(new Comment("mon commentaire de fin"));
    Element adresse = new Element("adresse");
    racine.addContent(adresse);

    Comment comment = new Comment("mon commentaire de debut");
    List content = document.getContent();
    content.add(0, comment);

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<!--mon commentaire de debut-->
<bibliotheque>
  <adresse />
</bibliotheque>
<!--mon commentaire de fin-->
```

Lors de l'instanciation d'un objet de type Comment, l'API vérifie que le contenu du commentaire respecte les spécifications XML et lève une exception de type IllegalDataException si celles-ci ne sont pas respectées.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.Comment;
import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM20 extends TestJDOM {

  public static void main(String[] args) {

    Element racine = new Element("bibliotheque");
    Comment comment1 = new Comment("mon commentaire -- bibliotheque");
    racine.addContent(comment1);
    Document document = new Document(racine);
    Element adresse = new Element("adresse");
    adresse.addContent("mon adresse");
    racine.addContent(adresse);

    afficher(document);
  }
}
```

Exemple :

```
Exception in thread "main" org.jdom.IllegalDataException:
The data "mon commentaire -- bibliotheque"
is not legal for a JDOM comment: Comments cannot contain double hyphens (--).
        at org.jdom.Comment.setText(Comment.java:120)
        at org.jdom.Comment.<init>(Comment.java:86)
        at fr.jmdoudoux.dej.jdom.TestJDOM20.main(TestJDOM20.java:13)
```

52.1.5.7. La classe Namespace

La classe org.jdom.Namespace encapsule un espace de nommage associé à un élément ou à un attribut. Chaque Namespace possède un URI. Si l'espace de nommage n'est pas celui par défaut, alors il doit avoir un préfixe sinon le préfixe est une chaîne vide.

Il peut y avoir autant d'espaces de nommages que nécessaire dans un même document.

Pour limiter l'occupation mémoire requise par l'espace de nommage de chaque éléments, il n'existe qu'une seule instance de la classe Namespace pour un même espace de nommage. Ceci est garantit par le fait que la classe Namespace ne possède pas de constructeur accessible et fait office de fabrique pour ces instances.

La méthode statique getNamespace() permet de retrouver ou de créer un espace de nom : elle attend en paramètre une URI et/ou un préfixe. Elle permet d'assurer que son appel avec les mêmes paramètres renvoie systématiquement la même instance de la classe Namespace.

Exemple :

```
      Namespace ns1 = Namespace.getNamespace("http://www.jmdoudoux.com");
      Namespace ns2 = Namespace.getNamespace("bibliotheque", "http://www.jmdoudoux.com");
```

Un objet de type Namespace peut être passé en paramètre du constructeur des classes Element et Attribute.

La surcharge de la méthode getNamespace() qui attend uniquement l'uri en paramètres permet de définir un espace de nommage par défaut.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.Namespace;

public class TestJDOM71 extends TestJDOM {

  public static void main(String[] args) {

    Namespace ns = Namespace.getNamespace("http://www.jmdoudoux.com");

    Element racine = new Element("bibliotheque", ns);
    Document document = new Document(racine);

    Element livre = new Element("livre", ns);
    racine.addContent(livre);

    livre.addContent(new Element("titre", ns).setText("titre1"));

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque xmlns="http://www.jmdoudoux.com">
  <livre>
    <titre>titre1</titre>
  </livre>
</bibliotheque>
```

La surcharge de la méthode getNamespace() qui attend le préfixe et l'uri en paramètre permet de définir un espace de nommage possédant un préfixe.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.Namespace;
```

```
public class TestJDOM72 extends TestJDOM {

  public static void main(String[] args) {

    Namespace ns = Namespace.getNamespace("bibliotheque","http://www.jmdoudoux.com");

    Element racine = new Element("bibliotheque", ns);
    Document document = new Document(racine);

    Element livre = new Element("livre", ns);
    racine.addContent(livre);

    livre.addContent(new Element("titre", ns).setText("titre1"));

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque:bibliotheque xmlns:bibliotheque="http://www.jmdoudoux.com">
  <bibliotheque:livre>
    <bibliotheque:titre>titre1</bibliotheque:titre>
  </bibliotheque:livre>
</bibliotheque:bibliotheque>
```

Il n'est pas possible de créer un élément dont le nom contient un caractère « : » : celui-ci est réservé par les spécifications de XML pour préciser un espace de nommage.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.Namespace;

public class TestJDOM73 extends TestJDOM {

  public static void main(String[] args) {

    Namespace ns = Namespace.getNamespace("bibliotheque","http://www.jmdoudoux.com");

    Element racine = new Element("bibliotheque", ns);
    Document document = new Document(racine);

    Element livre = new Element("bibliotheque:livre");
    racine.addContent(livre);

    livre.addContent(new Element("titre", ns).setText("titre1"));

    afficher(document);
  }
}
```

Résultat :

```
Exception in thread "main" org.jdom.IllegalNameException: The name "bibliotheque:livre"
is not legal for JDOM/XML elements: Element names cannot contain colons.
        at org.jdom.Element.setName(Element.java:207)
        at org.jdom.Element.<init>(Element.java:141)
        at org.jdom.Element.<init>(Element.java:153)
        at fr.jmdoudoux.dej.jdom.TestJDOM73.main(TestJDOM73.java:17)
```

Il faut utiliser une des surcharges du constructeur de la classe Element pour fournir l'espace de nommage.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.Namespace;

public class TestJDOM74 extends TestJDOM {

  public static void main(String[] args) {

    Namespace ns = Namespace.getNamespace("bibliotheque","http://www.jmdoudoux.com");

    Element racine = new Element("bibliotheque", ns);
    Document document = new Document(racine);

    Element livre = new Element("livre", "bibliotheque", "http://www.jmdoudoux.com");
    racine.addContent(livre);

    livre.addContent(new Element("titre", ns).setText("titre1"));

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque:bibliotheque xmlns:bibliotheque="http://www.jmdoudoux.com">
  <bibliotheque:livre>
    <bibliotheque:titre>titre1</bibliotheque:titre>
  </bibliotheque:livre>
</bibliotheque:bibliotheque>
```

Comme chaque objet de type Element et Attribute possède une référence sur un objet de type Namespace, il n'y a pas besoin de se préoccuper de l'espace de nommage lors du déplacement d'un élément.

Pour obtenir des éléments fils appartenant à un espace de nommage en utilisant les méthodes getChild() et getChildren() de la classe Element, il faut obligatoirement utiliser la surcharge de ces méthodes qui attend en paramètre un objet de type Namespace.

52.1.5.8. La classe CData

La classe CDATA est une sous-classe de la classe Text. La grande différence entre ces deux classes est la façon dont leurs données seront exportées par un objet de type XMLOutputter.

Ces données sont incluses dans une section CDATA et les caractères spéciaux qu'elles contiennent ne sont pas échappés.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.CDATA;
import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM17 extends TestJDOM {

  public static void main(String[] args) {

    Element racine = new Element("bibliotheque");
    Document document = new Document(racine);
    Element adresse = new Element("adresse");
    adresse.addContent("mon adresse < 5 et > 10 & impaire ");
    racine.addContent(adresse);
    CDATA cData = new CDATA("mon adresse < 5 et > 10 & impaire ");
    racine.addContent(cData);

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
  <adresse>mon adresse &lt; 5 et &gt; 10 &amp; impaire</adresse>
  <![CDATA[mon adresse < 5 et > 10 & impaire]]>
</bibliotheque>
```

52.1.5.9. La classe ProcessingInstruction

La classe ProcessingInstruction encapsule une instruction de traitement du document XML. Ces instructions permettent de fournir des informations aux outils qui traitent le document XML.

Une instruction est composée d'une cible (target) et de données (data).

La classe ProcessingInstruction possède plusieurs propriétés :

- target : le nom de la cible de l'instruction
- data : données de l'instruction
- parent : l'élément qui contient l'instruction
- document : le document qui contient l'instruction

Lors de l'instanciation d'un objet de type ProcessingInstruction, l'API vérifie que le nom de la cible respecte les spécifications XML et lève une exception de type IllegalTargetException si celles-ci ne sont pas respectées.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.ProcessingInstruction;

public class TestJDOM14 extends TestJDOM {

  public static void main(String[] args) {

    Element racine = new Element("html");
    Document document = new Document(racine);
    Element body = new Element("body");
    ProcessingInstruction pi = new ProcessingInstruction("php","echo 'bonjour';");
    body.addContent(pi);
    racine.addContent(body);

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<html>
  <body>
    <?php echo 'bonjour';?>
  </body>
```

```
</html>
```

Comme il est fréquent qu'une instruction de traitement possède des attributs, une surcharge du constructeur attend en paramètre le nom de la cible et un objet de type Map qui encapsule les attributs sous la forme clé/valeur.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.util.HashMap;
import java.util.Map;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.ProcessingInstruction;

public class TestJDOM45 extends TestJDOM {

  public static void main(String[] args) {

    Element racine = new Element("bibliotheque");
    Document document = new Document(racine);

    Map instructions = new HashMap();
    instructions.put("href", "bibliotheque.xsl");
    instructions.put("type", "text/xsl");
    ProcessingInstruction pi = new ProcessingInstruction("xml-stylesheet", instructions);
    document.getContent().add(0, pi);

    Element adresse = new Element("adresse");
    adresse.addContent("mon adresse");
    racine.addContent(adresse);

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="application/xml" href="bibliotheque.xsl"?>
<bibliotheque>
  <adresse>mon adresse</adresse>
</bibliotheque>
```

La méthode getParent() permet de connaître l'Element associé à l'instruction. Cette méthode renvoie null si l'instruction est rattachée directement au document ou si elle n'est pas encore attachée.

La classe ProcessingInstruction propose la méthode getData() qui renvoie toutes les données sous la forme d'une chaîne de caractères.

Fréquemment les données sont sous la forme d'attributs donc la classe ProcessingInstrution propose des méthodes pour faciliter leur manipulation :

- la méthode getPseudoAttributeValue() qui attend en paramètre le nom de l'attribut renvoie sa valeur
- la méthode getPseudoAttributeNames() renvoie une collection des noms d'attributs

Pour obtenir une instruction de traitement d'un élément, il faut utiliser la méthode getContent() de la classe Element ou Document et parcourir la collection pour obtenir celles de type ProcessingInstruction.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.IOException;
import java.io.StringReader;
import java.util.Iterator;
import java.util.List;

import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.ProcessingInstruction;
import org.jdom.input.SAXBuilder;

public class TestJDOM45 extends TestJDOM {

  public static void main(String[] args) {

    StringBuilder sb = new StringBuilder("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");
    sb.append("<?xml-stylesheet type=\"text/xsl\" href=\"bibliotheque.xsl\"?>");
    sb.append("<bibliotheque>");
    sb.append("  <adresse>mon adresse</adresse>");
    sb.append("</bibliotheque>");

    SAXBuilder builder = new SAXBuilder();
    Document document;
    try {
      document = builder.build(new StringReader(sb.toString()));

      List elements = document.getContent();
      Iterator iterator = elements.iterator();
      while (iterator.hasNext()) {
        Object o = iterator.next();
        if (o instanceof ProcessingInstruction) {
          ProcessingInstruction pi = (ProcessingInstruction) o;
          System.out.println("PI : "+pi.getTarget());

          List names = pi.getPseudoAttributeNames();
          Iterator itNames = names.iterator();
          while (itNames.hasNext()) {
            String name = itNames.next().toString();
            System.out.println("   "+name+ " = "+pi.getPseudoAttributeValue(name));
          }
        }
      }
    } catch (JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
PI : xml-stylesheet
   type = text/xsl
   href = bibliotheque.xsl
```

52.1.6. La création d'un document

JDOM encapsule un document XML dans une arborescence d'objets dont le point d'entrée est un objet de type org.jdom.Document.

La création d'un nouveau document se fait simplement en instanciant un objet de type Document grâce à l'opérateur new sur un des constructeurs de la classe.

Une instance de la classe Document peut aussi facilement être créée à partir d'un document XML existant en utilisant les classes SAXBuilder ou DOMBuilder du package org.jdom.input.

52.1.6.1. La création d'un nouveau document

Le plus simple pour créer un nouveau document est d'instancier un objet de type Element qui va encapsuler la racine du document et de passer cette instance au constructeur de la classe Document.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM48 {

  public static void main(String[] args) {

    Element racine = new Element("bibliotheque");
    Document document = new Document(racine);
  }
}
```

JDOM est beaucoup plus simple que DOM : la création d'un nouveau document avec Dom est plus verbeuse.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.DOMImplementation;
import org.w3c.dom.Document;
import org.w3c.dom.Element;

public class TestJDOM47 {

  public static void main(String[] args) {

    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    DocumentBuilder builder;
    try {
      builder = factory.newDocumentBuilder();
      DOMImplementation impl = builder.getDOMImplementation();
      Document doc = impl.createDocument(null, null, null);
      Element element = doc.createElement("bibliotheque");
    } catch (ParserConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

Il est aussi possible d'instancier un objet de type Document avec le constructeur par défaut et de lui associer l'élément racine en utilisant la méthode setRootElement().

Exemple :

```
    Document document = new Document();
    Element racine = new Element("bibliotheque");
    document.setRootElement(racine) ;
```

Dans ce cas, le document débute son existence dans un état illégal : il n'est pas possible de faire des opérations sur le document tant que sa racine n'est pas précisée sinon une exception de type IllegalStateException est levée.

52.1.6.2. L'obtention d'une instance de Document à partir d'un document XML

Pour obtenir un objet de type Document à partir d'un document XML existant, JDOM propose deux classes regroupées dans le package org.jdom.input qui implémentent l'interface Builder.

Cette interface définit la méthode build() qui renvoie un objet de type Document et qui est surchargée pour utiliser plusieurs sources différentes : flux (InputStream, Reader) , fichier (File) et URL (URL et String).

Les deux classes sont SAXBuilder et DOMBuilder.

JDOM ne fournit aucun parseur XML : il utilise celui précisé par JAXP ou celui explicitement demandé.

52.1.6.2.1. L'obtention d'une instance de Document à partir d'un document XML en utilisant SAX

La classe SAXBuilder permet d'analyser le document XML avec un parseur de type SAX compatible JAXP, de créer un arbre JDOM et de renvoyer un objet de type Document.

La mise en oeuvre de la classe SAXBuilder est très simple et ne nécessite que deux étapes :

- Instanciation d'un objet de type SAXBuilder en utilisant un de ses constructeurs
- Invocation de la méthode build() en lui passant en paramètre la ressource permettant l'accès au document

Si l'opération réussie, la méthode build() retourne un objet de type Document encapsulant le document sinon elle lève une exception de type JDOMException quand l'analyse du document échoue ou une exception de type IOException en cas de problème lors de la lecture du document.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.*;
import org.jdom.input.*;
import java.io.*;

public class TestJDOM3 {
  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File("test.xml"));
    } catch (JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

L'exception JDOMException est la classe mère de plusieurs exceptions notamment JDOMParseException qui permet de signifier un problème dans les traitements de JDOM.

Exemple avec le document XML :

```
<?xml version="1.0" encoding="UTF-8"?>
<html>
  <head></head>
  <body>
    <table width="100%" border="0"></table>
    <test>
  </body>
</html>
```

Résultat :

```
org.jdom.input.JDOMParseException: Error on line 7 of document
file:/C:/Documents%20and%20Settings/jmd/workspace/TestJDOM/test.xml:
The element type "test" must be terminated by the matching end-tag "</test>".
      at org.jdom.input.SAXBuilder.build(SAXBuilder.java:501)
      at org.jdom.input.SAXBuilder.build(SAXBuilder.java:847)
      at org.jdom.input.SAXBuilder.build(SAXBuilder.java:826)
      at fr.jmdoudoux.dej.jdom.TestJDOM3.main(TestJDOM3.java:10)
Caused by: org.xml.sax.SAXParseException: The element type "test"
must be terminated by the matching end-tag "</test>".
      at com.sun.org.apache.xerces.internal.util.ErrorHandlerWrapper.createSAXParseException
      (ErrorHandlerWrapper.java:195)
      at com.sun.org.apache.xerces.internal.util.ErrorHandlerWrapper.fatalError
      (ErrorHandlerWrapper.java:174)
      at com.sun.org.apache.xerces.internal.impl.XMLErrorReporter.reportError
      (XMLErrorReporter.java:388)
      at com.sun.org.apache.xerces.internal.impl.XMLScanner.reportFatalError
      (XMLScanner.java:1411)
      at com.sun.org.apache.xerces.internal.impl.XMLDocumentFragmentScannerImpl.scanEndElement
      (XMLDocumentFragmentScannerImpl.java:1739)
      at com.sun.org.apache.xerces.internal.impl.XMLDocumentFragmentScannerImpl
      $FragmentContentDriver.next(XMLDocumentFragmentScannerImpl.java:2923)
      at com.sun.org.apache.xerces.internal.impl.XMLDocumentScannerImpl.next
      (XMLDocumentScannerImpl.java:645)
      at com.sun.org.apache.xerces.internal.impl.XMLNSDocumentScannerImpl.next
      (XMLNSDocumentScannerImpl.java:140)
      at com.sun.org.apache.xerces.internal.impl.XMLDocumentFragmentScannerImpl.scanDocument
      (XMLDocumentFragmentScannerImpl.java:508)
      at com.sun.org.apache.xerces.internal.parsers.XML11Configuration.parse
      (XML11Configuration.java:807)
      at com.sun.org.apache.xerces.internal.parsers.XML11Configuration.parse
      (XML11Configuration.java:737)
      at com.sun.org.apache.xerces.internal.parsers.XMLParser.parse(XMLParser.java:107)
      at com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser.parse
      (AbstractSAXParser.java:1205)
      at com.sun.org.apache.xerces.internal.jaxp.SAXParserImpl$JAXPSAXParser.parse
      (SAXParserImpl.java:522)
      at org.jdom.input.SAXBuilder.build(SAXBuilder.java:489)
      ... 3 more
```

Par défaut, le parseur utilisé par JDOM est celui précisé par JAXP ou à défaut Xerces.

SAXBuilder possède plusieurs constructeurs qui permettent de préciser la classe du parseur (nom de la classe pleinement qualifiée de type XMLReader) à utiliser et/ou un booléen qui indique si le document doit être validé.

Par défaut, les vérifications faites par SAXBuilder sur le document XML ne concernent que le fait que le document soit bien formé. Pour valider le document explicitement, il faut demander la validation en passant la valeur true au paramètre validate du constructeur de SAXBuilder.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.*;
import org.jdom.input.*;
import java.io.*;

public class TestJDOM3 {
  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder(true);
      Document document = builder.build(new File("test.xml"));
    } catch(JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
org.jdom.input.JDOMParseException: Error on line 2 of document
file:/C:/Documents%20and%20Settings/jmd/workspace/TestJDOM/test.xml:
Document is invalid: no grammar found.
        at org.jdom.input.SAXBuilder.build(SAXBuilder.java:501)
        at org.jdom.input.SAXBuilder.build(SAXBuilder.java:847)
        at org.jdom.input.SAXBuilder.build(SAXBuilder.java:826)
        at fr.jmdoudoux.dej.jdom.TestJDOM3.main(TestJDOM3.java:10)
Caused by: org.xml.sax.SAXParseException: Document is invalid: no grammar found.
        at com.sun.org.apache.xerces.internal.util.ErrorHandlerWrapper.createSAXParseException
        (ErrorHandlerWrapper.java:195)
        at com.sun.org.apache.xerces.internal.util.ErrorHandlerWrapper.error
        (ErrorHandlerWrapper.java:131)
        at com.sun.org.apache.xerces.internal.impl.XMLErrorReporter.reportError
        (XMLErrorReporter.java:384)
        at com.sun.org.apache.xerces.internal.impl.XMLErrorReporter.reportError
        (XMLErrorReporter.java:318)
...
```

Dans l'exemple ci-dessus, la validation échoue car aucune DTD n'est précisée dans le document.

Il est possible de configurer le parseur SAX utilisé par SAXBuilder grâce à plusieurs méthodes :

- setErrorHandler(ErrorHandler)
- setEntityResolver(EntityResolver)
- setDTDHandler(DTDHandler)
- setIgnoringElementContentWhitespace(boolean)
- setFeature(String *name*, boolean *value*)
- setProperty(String *name*, Object *value*)

Il est possible de construire un arbre JDOM en utilisant la classe SAXBuilder à partir d'une chaîne de caractères contenant le document XML. Il suffit d'instancier un objet de type StringReader() en lui passant en paramètre la chaîne de caractères contenant le document.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.IOException;
import java.io.StringReader;

import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM49 {

  public static void main(String[] args) {
    try {
      StringBuilder sb = new StringBuilder("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");
      sb.append("<?xml-stylesheet type=\"text/xsl\" href=\"bibliotheque.xsl\"?>");
      sb.append("<bibliotheque>");
      sb.append("  <adresse>mon adresse complete</adresse>");
      sb.append("</bibliotheque>");

      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new StringReader(sb.toString()));
    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}
```

L'utilisation de SAXBuilder est particulièrement recommandée car elle consomme moins de ressources.

52.1.6.2.2. L'obtention d'une instance de Document à partir d'un arbre DOM

Bien qu'ayant des similitudes de nom, JDOM n'est pas compatible avec DOM. Il n'est pas possible d'utiliser des éléments d'une API dans l'autre directement.

JDOM propose cependant de convertir un arbre DOM en modèle JDOM et vice versa. Ceci est pratique pour intégrer JDOM dans du code existant manipulant des arbres DOM.

Pour créer un modèle JDOM à partir d'un arbre DOM, il faut utiliser la classe org.jdom.input.DomBuilder. Sa mise en oeuvre est similaire à celle de la classe SAXBuidler : instancier un objet de type DOMBuilder et invoquer sa méthode build().

La classe DOMBuilder propose deux surcharges de la méthode build() :

- org.jdom.Document build(org.w3c.dom.Document) : créer un document JDOM à partir d'un document DOM
- org.jdom.Element build(org.w3c.dom.Element) : créer un élément JDOM à partir d'un élément DOM

Les modifications faites dans le modèle JDOM n'ont aucun impact sur l'arbre DOM utilisé initialement pour créer l'arbre d'objets JDOM et vice versa.

52.1.6.3. La création d'éléments

Pour créer un nouvel élément, il suffit d'instancier un objet de la classe Element. Tous les constructeurs de cette classe attendent au moins en paramètre le nom de l'élément. Ce nom doit respecter les spécifications XML, sinon une exception de type IllegalNameException est levée.

L'exception IllegalNameException hérite de l'exception IllegalArgumentException qui est une exception de type runtime : il n'est donc pas obligatoire de traiter ce type d'exception mais elle peut tout de même être levée à l'exécution.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM29 extends TestJDOM {

  public static void main(String[] args) {

    Element racine = new Element("bibliotheque");
    Document document = new Document(racine);
    Element livres = new Element("*livres");
    racine.addContent(livres);

    afficher(document);
  }
}
```

Résultat :

```
Exception in thread "main" org.jdom.IllegalNameException:
The name "*livres" is not legal for JDOM/XML elements:
XML names cannot begin with the character "*".
        at org.jdom.Element.setName(Element.java:207)
        at org.jdom.Element.<init>(Element.java:141)
        at org.jdom.Element.<init>(Element.java:153)
        at fr.jmdoudoux.dej.jdom.TestJDOM29.main(TestJDOM29.java:13)
```

Chaque Element peut avoir autant d'objets de type Element fils que nécessaire pour représenter le document XML.

Il est possible de préciser le texte associé à l'élément en utilisant la méthode setText() de la classe Element.

52.1.6.4. L'ajout d'éléments fils

Pour ajouter un élément fils à un élément, il faut instancier l'élément fils et utiliser la méthode addContent() de l'instance de l'objet père.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM30 extends TestJDOM {

  public static void main(String[] args) {

    Element racine = new Element("bibliotheque");
    Document document = new Document(racine);
    Element livres = new Element("livres");
    racine.addContent(livres);

    Element livre = new Element("livre");
    livres.addContent(livre);

    Element titre = new Element("titre").setText("Titre livre 1");
    Element auteur = new Element("auteur").setText("Auteur 1");
    livre.addContent(titre);
    livre.addContent(auteur);

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
  <livres>
    <livre>
      <titre>Titre livre 1</titre>
      <auteur>Auteur 1</auteur>
    </livre>
  </livres>
</bibliotheque>
```

Comme la plupart des méthodes qui modifient un Element renvoient l'élément lui-même, il est possible de chaîner les différents appels aux méthodes.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM28 extends TestJDOM {

  public static void main(String[] args) {

    Element racine = new Element("bibliotheque");
    Document document = new Document(racine);
    Element livres = new Element("livres");
    racine.addContent(livres);
    livres.addContent(new Element("livre")
      .addContent(new Element("titre").setText("Titre livre 1"))
      .addContent(new Element("auteur").setText("Auteur 1"))
    );

    afficher(document);
  }
}
```

Attention : la lisibilité du code devient moins triviale.

Il est possible de définir sa propre classe qui hérite de la classe Element pour par exemple créer une portion de document.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.Element;

public class ElementLivre extends Element {

  private static final long serialVersionUID = 1L;

  public ElementLivre(String titre, String auteur) {
    super("Livre");
    addContent(new Element("titre").setText(titre));
    addContent(new Element("auteur").setText(auteur));
  }
}
```

Il suffit alors d'utiliser cette classe pour créer le document.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM31 extends TestJDOM {

  public static void main(String[] args) {

    Element racine = new Element("bibliotheque");
    Document document = new Document(racine);
    Element livres = new Element("livres");
    racine.addContent(livres);

    Element livre = new ElementLivre("Titre livre 1","Auteur 1");
    livres.addContent(livre);

    afficher(document);
  }
}
```

52.1.7. L'arborescence d'éléments

Un document XML possède une structure arborescente dans laquelle un Element peut avoir des Elements fils.

Cette section va utiliser le document xml suivant

Exemple : le fichier bibliotheque.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
  <livre>
    <!-- commentaires livre 1 -->
    <titre>titre1</titre>
    <auteur>auteur1</auteur>
    <editeur>editeur1</editeur>
  </livre>
  <livre>
    <!-- commentaires livre 2 -->
    <titre>titre2</titre>
    <auteur>auteur2</auteur>
    <editeur>editeur2</editeur>
  </livre>
  <livre>
    <!-- commentaires livre 3 -->
    <titre>titre3</titre>
    <auteur>auteur3</auteur>
    <editeur>editeur3</editeur>
  </livre>
</bibliotheque>
```

52.1.7.1. Le parcours des éléments

Le parcours des éléments avec JDOM utilise le framework Collection notamment les classes List et Iterator.

La méthode getChildren() de la classe Element renvoie une collection qui encapsule les éléments fils uniquement de premier niveau. Elle possède plusieurs surcharges :

Méthode	Rôle
List getChildren()	Renvoyer tous les éléments fils de premier niveau
List getChildren(String)	Renvoyer tous les éléments fils de premier niveau dont le nom est fourni en paramètre
List getChildren(String, Namescape)	Renvoyer tous les éléments fils de premier niveau dont le nom et l'espace de nommage sont fournis en paramètres

Il suffit alors de réaliser une itération sur la collection pour effectuer les traitements voulus.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.File;
import java.io.IOException;
import java.util.List;
import java.util.ListIterator;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM5 {
  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File("bibliotheque.xml"));

      Element element = document.getRootElement();
      List livres = element.getChildren("livre");
      ListIterator iterator = livres.listIterator();
      while (iterator.hasNext()) {
        Element el = (Element) iterator.next();
        System.out.println("titre = " + el.getChild("titre").getText());
      }
    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}
```

Résultat :

```
titre = titre1
titre = titre2
titre = titre3
```

L'inconvénient d'utiliser un nom de tag pour rechercher un élément est qu'il faut être sûr que l'élément existe sinon une exception de type NullPointerException est levée.

Le plus simple est d'avoir une DTD et d'activer la validation du document par le parseur.

Attention : le nom des éléments utilisés est sensible à la casse.

JDOM n'utilise pas encore les generics : il est donc nécessaire de réaliser des casts et des tests de type sur les éléments des collections.

52.1.7.2. L'accès direct à un élément fils

Plutôt que d'itérer sur les éléments fils, il est plus rapide d'utiliser la méthode getChild() de la classe Element surtout s'il n'y a qu'un seul élément fils ou que c'est le premier que l'on souhaite obtenir.

La méthode getChild() possède deux surcharges :

Méthode	Rôle
Element getChild(String)	Renvoyer le premier fils dont le nom est fourni en paramètre
Element getChild(String, Namespace)	Renvoyer le premier fils dont le nom et l'espace de nommage sont fournis en paramètres

Si aucun fils ne correspond alors la méthode renvoie null.

Si plusieurs fils correspondent alors la méthode getChild() renvoie uniquement le premier.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM41 {

  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document document = builder.build(new File("bibliotheque.xml"));

      Element elementRacine = document.getRootElement();
      Element elementLivre = elementRacine.getChild("livre");
      Element elementAuteur = elementLivre.getChild("auteur");
      System.out.println("auteur du premier livre = "+elementAuteur.getText());
    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}
```

Résultat :

```
auteur du premier livre = auteur1
```

Il est possible de chaîner les appels à la méthode getChild() puisqu'elle renvoie l'élément correspondant.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM42 {

  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document document = builder.build(new File("bibliotheque.xml"));

      Element elementAuteur = document.getRootElement().getChild("livre").getChild("auteur");
      System.out.println("auteur du premier livre = " + elementAuteur.getText());
    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}
```

La méthode getChild() ne renvoie que le premier fils correspondant au critère fourni : si plusieurs éléments fils répondent au critère, il faut utiliser la méthode getChildren() et itérer sur la collection.

Si aucun élément fils ne répond au critère alors la méthode getChild() renvoie null ce qui impliquera la levée d'une exception de type NullPointerException.

Pour éviter ceci, il est nécessaire de connaître la structure du document XML et que celui-ci soit validé grâce à une DTD ou un schéma.

Attention, si un élément père contient la définition d'un espace de nommage, il est nécessaire de le préciser.

Exemple : le document XML utilisé

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque xmlns="http://www.jmdoudoux.com">>
  <livre>
    <!-- commentaires livre 1 -->
    <titre>titre1</titre>
    <auteur>auteur1</auteur>
    <editeur>editeur1</editeur>
  </livre>
  <livre>
    <!-- commentaires livre 2 -->
    <titre>titre2</titre>
    <auteur>auteur2</auteur>
    <editeur>editeur2</editeur>
  </livre>
  <livre>
    <!-- commentaires livre 3 -->
    <titre>titre3</titre>
    <auteur>auteur3</auteur>
    <editeur>editeur3</editeur>
  </livre>
</bibliotheque>
```

Dans cette version du document XML, la balise racine définit un espace de nommage qui est donc propagé à tous ses éléments fils.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM44 {

  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document document = builder.build(new File("bibliotheque.xml"));

      Element elementRacine = document.getRootElement();
      Element elementLivre = elementRacine.getChild("livre");
      Element elementAuteur = elementLivre.getChild("auteur");
      System.out.println("auteur du premier livre = "+elementAuteur.getText());
    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}
```

Résultat :

```
Exception in thread "main" java.lang.NullPointerException
        at fr.jmdoudoux.dej.jdom.TestJDOM44.main(TestJDOM44.java:21)
```

Dans cet exemple, l'objet elementLivre est null car la méthode getChild() renvoie null : il n'existe pas dans le document d'élément fils de l'élément racine avec le nom « livre » et l'espace de nommage par défaut.

Pour que l'exemple précédent fonctionne, il est nécessaire de préciser l'espace de nommage en paramètre de la méthode getChild()

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.Namespace;
import org.jdom.input.SAXBuilder;
```

```
public class TestJDOM44 {

  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document document = builder.build(new File("bibliotheque.xml"));

      Namespace ns = Namespace.getNamespace("http://www.jmdoudoux.com");

      Element elementRacine = document.getRootElement();
      Element elementLivre = elementRacine.getChild("livre", ns);
      Element elementAuteur = elementLivre.getChild("auteur", ns);
      System.out.println("auteur du premier livre = "+elementAuteur.getText());
    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}
```

L'API Collection permet aussi d'accéder directement à un des éléments fils.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM43 {

  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document document = builder.build(new File("bibliotheque.xml"));

      Element elementLivre = (Element) document.getRootElement().getChildren("livre").get(2);
      Element elementAuteur = elementLivre.getChild("auteur");
      System.out.println("auteur du troisieme livre = " + elementAuteur.getText());
    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}
```

Résultat :

```
auteur du troisieme livre = auteur3
```

52.1.7.3. Le parcours de toute l'arborescence d'un document

JDOM ne fournit aucune API permettant de parcourir facilement tout le document comme peut le proposer DOM. Il faut utiliser une méthode récursive.

Exemple :

```
package fr.jmdoudoux.dej.jdom;
```

```
import java.io.File;
import java.io.IOException;
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM32 {

  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document document = builder.build(new File("bibliotheque.xml"));

      Element elementRacine = document.getRootElement();
      afficherFils(elementRacine, 0);
    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }

  /**
   * Methode recursive qui parcours un element et affiche ses fils
   * @param element
   * @param niveau
   */
  private static void afficherFils(Element element, int niveau) {
    String indentation = getIndentation(niveau);
    StringBuilder ligne = new StringBuilder(indentation);

    ligne.append(element.getName());
    if(element.getChildren().isEmpty()) {
      ligne.append(" = ");
      ligne.append(element.getText());
    }

    System.out.println(ligne.toString());

    List fils = element.getChildren();
    Iterator iterator = fils.iterator();
    while (iterator.hasNext()) {
      Element elementFils = (Element) iterator.next();
      afficherFils(elementFils, niveau+1);
    }
  }

  private static String getIndentation(int n) {
    char[] car = new char[n];
    Arrays.fill(car, 0, n, ' ');
    return new String(car);
  }
}
```

Résultat :

```
bibliotheque
 livre
  titre = titre1
  auteur = auteur1
  editeur = editeur1
 livre
  titre = titre2
  auteur = auteur2
  editeur = editeur2
 livre
  titre = titre3
  auteur = auteur3
  editeur = editeur3
```

La méthode getChildren() ne renvoie que des Elements. Pour obtenir les autres entités liées à l'élément, il faut utiliser la méthode getContent() qui renvoie toutes les entités (commentaires, texte, ...) y compris les éléments fils sous la forme d'une collection.

Pour traiter chaque élément de cette collection, il est nécessaire de faire un test de type sur l'occurrence de la collection en cours de traitement grâce à l'opérateur instanceof.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.File;
import java.io.IOException;
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;

import org.jdom.Comment;
import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM33 {

  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document document = builder.build(new File("bibliotheque.xml"));

      Element elementRacine = document.getRootElement();
      afficherFils(elementRacine, 0);
    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }

  /**
   * Methode recursive qui parcours un element et affiche ses fils
   * @param element
   * @param niveau
   */
  private static void afficherFils(Element element, int niveau) {
    String indentation = getIndentation(niveau);
    StringBuilder ligne = new StringBuilder(indentation);

    ligne.append(element.getName());
    if(element.getChildren().isEmpty()) {
      ligne.append(" = ");
      ligne.append(element.getText());
    }

    System.out.println(ligne.toString());

    List fils = element.getContent();
    Iterator iterator = fils.iterator();
    while (iterator.hasNext()) {
      Object objetFils = iterator.next();
      if (objetFils instanceof Element) {
        Element elementFils = (Element) objetFils;
        afficherFils(elementFils, niveau+1);
      } else {
        if (objetFils instanceof Comment) {
          Comment com = (Comment) objetFils;
          System.out.println(indentation+" -- "+com.getValue());
        }
      }
    }
  }

  private static String getIndentation(int n) {
    char[] car = new char[n];
    Arrays.fill(car, 0, n, ' ');
    return new String(car);
  }
}
```

Résultat :

```
bibliotheque
 livre
  --  commentaires livre 1
  titre = titre1
  auteur = auteur1
  editeur = editeur1
 livre
  --  commentaires livre 2
  titre = titre2
  auteur = auteur2
  editeur = editeur2
 livre
  --  commentaires livre 3
  titre = titre3
  auteur = auteur3
  editeur = editeur3
```

52.1.7.4. Les éléments parents

JDOM permet une navigation descendante de l'arbre des objets mais aussi ascendante.

Chaque élément à un unique élément père sauf l'élément racine qui n'en a pas.

La méthode getParent() renvoie l'élément parent de l'élément courant. Elle renvoie null pour l'élément racine du document.

Il est ainsi possible de remonter récursivement dans l'arborescence de niveau en niveau jusqu'à ce qu'il n'y ait plus d'élément parent.

Remarque : un élément qui n'est pas encore rattaché à un document ne possède pas non plus d'élément père : dans ce cas les méthodes getParent() et getDocument() renvoient null.

La classe Element propose la méthode isRootElement() qui renvoie true si l'élément est la racine du document.

La classe Element propose aussi la méthode isAncestor() qui renvoie un booléen indiquant si l'élément est un ancêtre de l'élément fourni en paramètre.

52.1.8. La modification d'un document

La classe Element propose de nombreuses méthodes pour modifier un élément :

Méthode	Rôle
addContent()	Ajouter l'entité fournie en paramètre en tant que fils de l'élément. Plusieurs surcharges existent pour des paramètres de type Content, Collection ou String et certaines permettent de définir l'index
removeAttribute()	Supprimer un attribut de l'élément
removeChild()	Supprimer un élément fils

removeChildren()	Supprimer tous les éléments fils dont les noms sont fournis en paramètres
removeContent()	Supprimer une entité fille
setAttribute()	Créer ou modifier un attribut
setContent()	Remplacer un noeud ou tous les noeuds de l'élément
setName()	Modifier le nom de l'élément
setText()	Modifier le texte de l'élément

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM66 extends TestJDOM {
  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File("bibliotheque.xml"));

      Element livre2 = (Element) document.getRootElement().getChildren().get(1);

      // Ajouter un nouvel élément
      livre2.addContent(new Element("publication").setText("1996"));
      // Supprimer tous les noeuds nommé editeur
      livre2.removeChildren("editeur");

      afficher(document);
    } catch(JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
  <livre>
    <!-- commentaires livre 1 -->
    <titre>titre1</titre>
    <auteur>auteur1</auteur>
    <editeur>editeur1</editeur>
  </livre>
  <livre>
    <!-- commentaires livre 2 -->
    <titre>titre2</titre>
    <auteur>auteur2</auteur>
    <publication>1996</publication>
  </livre>
  <livre>
    <!-- commentaires livre 3 -->
    <titre>titre3</titre>
    <auteur>auteur3</auteur>
    <editeur>editeur3</editeur>
  </livre>
</bibliotheque>
```

L'appel à la méthode getChildren() renvoie une collection des noeuds fils de l'élément courant. Cette collection est dynamique et peut être directement modifiée pour ajouter/enlever des noeuds, réordonner les noeuds simplement en utilisant les méthodes de l'API Collection.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.File;
import java.io.IOException;
import java.util.List;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM65 extends TestJDOM {
  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File("bibliotheque.xml"));

      Element livre2 = (Element) document.getRootElement().getChildren().get(1);

      List fils = livre2.getChildren();

      // Ajouter un nouvel élément
      fils.add(new Element("publication").setText("1996"));
      // Ajouter un nouveau noeud après le second noeud
      fils.add(1, new Element("isbn").setText("0000000000"));


      Element livre3 = (Element) document.getRootElement().getChildren().get(2);
      fils = livre3.getChildren();
      // supprimer le second noeud
      fils.remove(1);
      // Supprimer tous les neouds nommé editeur
      fils.removeAll(livre3.getChildren("editeur"));

      afficher(document);
    } catch(JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
  <livre>
    <!-- commentaires livre 1 -->
    <titre>titre1</titre>
    <auteur>auteur1</auteur>
    <editeur>editeur1</editeur>
  </livre>
  <livre>
    <!-- commentaires livre 2 -->
    <titre>titre2</titre>
    <isbn>0000000000</isbn>
    <auteur>auteur2</auteur>
    <editeur>editeur2</editeur>
    <publication>1996</publication>
  </livre>
  <livre>
    <!-- commentaires livre 3 -->
    <titre>titre3</titre>
  </livre>
</bibliotheque>
```

L'utilisation de l'API collection implique de tenir compte des contraintes qu'elle impose. Par exemple, il faut être vigilent quant aux modifications de la collection lors de son parcours avec un itérateur.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.File;
import java.io.IOException;
import java.util.Iterator;
import java.util.List;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM69 extends TestJDOM {
  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File("bibliotheque.xml"));

      Element racine = document.getRootElement();
      Element livre2 = (Element) racine.getChildren().get(1);

      List livre2Fils = livre2.getChildren();
      Iterator itr = livre2Fils.iterator();
      while (itr.hasNext()) {
        Element fils = (Element) itr.next();
        if ("auteur".equals(fils.getName())) {
          fils.detach();
        }
      }

      afficher(document);
    } catch(JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
Exception in thread "main" java.util.ConcurrentModificationException
   at org.jdom.ContentList$FilterListIterator.checkConcurrentModification(ContentList.java:940)
   at org.jdom.ContentList$FilterListIterator.nextIndex(ContentList.java:829)
   at org.jdom.ContentList$FilterListIterator.hasNext(ContentList.java:785)
   at fr.jmdoudoux.dej.jdom.TestJDOM69.main(TestJDOM69.java:23)
```

Une exception de type ConcurrentModificationException est levée car la méthode detach() modifie le contenu de la collection de façon concurrente au parcours fait par l'itérateur. Pour pallier ce problème, il ne faut pas utiliser la méthode detach() de la classe Element mais utiliser la méthode remove() de l'itérateur.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.File;
import java.io.IOException;
import java.util.Iterator;
import java.util.List;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM69 extends TestJDOM {
  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File("bibliotheque.xml"));

      Element racine = document.getRootElement();
      Element livre2 = (Element) racine.getChildren().get(1);

      List livre2Fils = livre2.getChildren();
      Iterator itr = livre2Fils.iterator();
      while (itr.hasNext()) {
        Element fils = (Element) itr.next();
        if ("auteur".equals(fils.getName())) {
          itr.remove();
        }
      }

      afficher(document);
    } catch(JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
  <livre>
    <!-- commentaires livre 1 -->
    <titre>titre1</titre>
    <auteur>auteur1</auteur>
    <editeur>editeur1</editeur>
  </livre>
  <livre>
    <!-- commentaires livre 2 -->
    <titre>titre2</titre>
    <editeur>editeur2</editeur>
  </livre>
  <livre>
    <!-- commentaires livre 3 -->
    <titre>titre3</titre>
    <auteur>auteur3</auteur>
    <editeur>editeur3</editeur>
  </livre>
</bibliotheque>
```

52.1.8.1. L'obtention du texte d'un élément

La classe Element propose plusieurs méthodes pour obtenir et modifier le texte d'un élément

Méthode	Rôle
String getText()	Renvoyer le texte de l'élément
String getTextTrim()	Renvoyer le texte de l'élément sans les espaces de début et de fin
String getTextNormalize()	Renvoyer le texte de l'élément sans les espaces de début et de fin, de plus tous les espaces consécutifs sont remplacés par un espace unique
Element setText(String)	Forcer la création d'un noeud unique de type texte

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.Element;

public class TestJDOM51 extends TestJDOM {

  public static void main(String[] args) {

    Element texte = new Element("texte");
    texte.setText("  mon   texte avec     des   espaces   ");
    System.out.println("getText()=*"+texte.getText()+"*");
    System.out.println("getTextTrim()=*"+texte.getTextTrim()+"*");
    System.out.println("getTextNormalize()=*"+texte.getTextNormalize()+"*");
  }
}
```

Résultat :

```
getText()=*  mon   texte avec     des   espaces   *
getTextTrim()=*mon   texte avec     des   espaces*
getTextNormalize()=*mon texte avec des espaces*
```

Dans le cas où un commentaire ou une instruction de traitement est inclus dans le texte, celui-ci est ignoré lors de l'appel à la méthode getText().

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.Element;
import java.io.IOException;
import java.io.StringReader;

import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM63 extends TestJDOM {

  public static void main(String[] args) {

    try {
      StringBuilder sb = new StringBuilder("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");
      sb.append("<?xml-stylesheet type=\"text/xsl\" href=\"bibliotheque.xsl\"?>");
      sb.append("<bibliotheque>");
      sb.append("  <adresse>mon adresse <!-- commentaire -->complete</adresse>");
      sb.append("</bibliotheque>");

      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new StringReader(sb.toString()));

      Element adresse = document.getRootElement().getChild("adresse");
      System.out.println("getText()=*" + adresse.getText() + "*");
      System.out.println("getTextTrim()=*" + adresse.getTextTrim() + "*");
      System.out.println("getTextNormalize()=*" + adresse.getTextNormalize() + "*");
    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}
```

Résultat :

```
getText()=*mon adresse complete*
getTextTrim()=*mon adresse complete*
getTextNormalize()=*mon adresse complete*
```

De plus, si le texte contient des éléments fils, le texte de ces derniers n'est pas repris par l'appel à la méthode getText().

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.Element;
import java.io.IOException;
import java.io.StringReader;

import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM64 extends TestJDOM {

  public static void main(String[] args) {

    try {
      StringBuilder sb = new StringBuilder("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");
      sb.append("<bibliotheque>");
      sb.append("  <adresse>mon adresse <b>complete</b> et integrale</adresse>");
      sb.append("</bibliotheque>");

      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new StringReader(sb.toString()));

      Element adresse = document.getRootElement().getChild("adresse");
      System.out.println("getText()=*" + adresse.getText() + "*");
      System.out.println("getTextTrim()=*" + adresse.getTextTrim() + "*");
      System.out.println("getTextNormalize()=*" + adresse.getTextNormalize() + "*");
    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}
```

Résultat :

```
getText()=*mon adresse  et integrale*
getTextTrim()=*mon adresse  et integrale*
getTextNormalize()=*mon adresse et integrale*
```

Pour obtenir l'intégralité du texte incluant le texte des éléments fils, il est nécessaire d'écrire un morceau de code qui va parcourir le noeud et ses éléments fils en concaténant le résultat des appels à la méthode getText().

52.1.8.2. La modification du texte d'un élément

La méthode setText() permet de modifier le texte d'un élément. Attention, son utilisation supprime tous les noeuds de l'élément déjà existant.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.Comment;
import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM52 extends TestJDOM {

  public static void main(String[] args) {

    Element racine = new Element("racine");
    Document document = new Document(racine);
    Element texte = new Element("test");
    racine.addContent(texte);
    Element fils = new Element("fils");
    texte.addContent(fils);
    Comment commentaire = new Comment("mon commentaire");
    texte.addContent(commentaire);
    texte.setText("mon texte");

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<racine>
  <test>mon texte</test>
</racine>
```

Pour éviter la suppression des noeuds fils, il faut utiliser la méthode addContent() plutôt que la méthode setText().

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.Comment;
import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM53 extends TestJDOM {

  public static void main(String[] args) {

    Element racine = new Element("racine");
    Document document = new Document(racine);
    Element texte = new Element("test");
    racine.addContent(texte);
    Element fils = new Element("fils");
    texte.addContent(fils);
    Comment commentaire = new Comment("mon commentaire");
    texte.addContent(commentaire);
    texte.addContent("mon texte");

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<racine>
  <test>
    <fils />
    <!--mon commentaire-->
    mon texte
  </test>
</racine>
```

Il ne faut pas utiliser de séquences d'échappement dans le texte d'un élément. JDOM prend le texte tel quel et les caractères seront échappés lors de l'exportation du document.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import org.jdom.Document;
import org.jdom.Element;
```

```
public class TestJDOM54 extends TestJDOM {

  public static void main(String[] args) {

    Element racine = new Element("racine");
    Document document = new Document(racine);
    Element texte = new Element("texte");
    racine.addContent(texte);
    Element test1 = new Element("test1");
    test1.setText("&#002A;");
    texte.addContent(test1);
    Element test2 = new Element("test2");
    test2.setText("\u002A");
    texte.addContent(test2);

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<racine>
  <texte>
    <test1>&amp;#002A;</test1>
    <test2>*</test2>
  </texte>
</racine>
```

52.1.8.3. L'obtention du texte d'un élément fils

Il est fréquent dans un document de vouloir obtenir le texte d'un élément fils.

La classe Element propose plusieurs méthodes pour obtenir facilement le texte d'un élément fils. Ces méthodes possèdent deux surcharges : une attendant en paramètre le nom du tag fils, l'autre le nom du tag fils et son espace de nommage.

Méthode	Rôle
String getChildText()	Renvoyer le texte du premier élément fils
String getChildTextTrim()	Renvoyer le texte du premier élément fils sans les espaces de début et de fin
String getChildTextNormalize()	Renvoyer le texte du premier élément fils sans les espaces de début et de fin, de plus tous les espaces consécutifs sont remplacés par un espace unique

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.Namespace;
import org.jdom.input.SAXBuilder;

public class TestJDOM55 {

  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document document = builder.build(new File("bibliotheque.xml"));

      Namespace ns = Namespace.getNamespace("http://www.jmdoudoux.com");

      Element elementRacine = document.getRootElement();
      Element elementLivre = elementRacine.getChild("livre", ns);
      String titre = elementLivre.getChildText("titre", ns);
      System.out.println("titre du premier livre = "+titre);
      String auteur = elementLivre.getChildText("auteur", ns);
      System.out.println("auteur du premier livre = "+auteur);
      String editeur = elementLivre.getChildText("editeur", ns);
      System.out.println("editeur du premier livre = "+editeur);
    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}
```

Résultat :

```
titre du premier livre = titre1
auteur du premier livre = auteur1
editeur du premier livre = editeur1
```

Il est préférable de valider le document XML utilisé avant d'utiliser ces méthodes :

- Elles ne renvoient que le premier élément fils répondant au critère
- Si aucun élément n'est trouvé alors elles renvoient null

Il ne faut les utiliser que pour des éléments fils uniques qui ne contiennent que des noeuds de type #PCDATA.

52.1.8.4. L'ajout et la suppression des fils

La classe Element possède plusieurs surcharges de la méthode addContent() pour ajouter des noeuds fils à l'élément.

Méthode	Rôle
addContent(Content)	Ajouter le noeud fourni à la suite des noeuds existants
addContent(int, Content)	Ajouter le noeud fourni à l'index fourni
addContent(Collection)	Ajouter la collection de noeuds fournie à la suite des noeuds existants
addContent(int, Collection)	Ajouter la collection de noeuds fournie à l'index indiqué
addContent(String)	Ajouter un noeud de type Text

Toutes ces méthodes peuvent lever une exception de type IllegalAddException et renvoient l'élément lui-même.

La classe Element possède plusieurs surcharges des méthodes removeContent(), removeChild() et removeChildren() pour supprimer des noeuds fils de l'élément.

Méthode	Rôle
removeContent()	Supprimer tous les noeuds fils
removeContent(Content)	Supprimer le noeud fils fourni
removeContent(int)	Supprimer le noeud dont l'index est fourni
removeContent(Filter)	Supprimer les noeuds fils correspondant au filtre fourni
removeChild(String)	Supprimer le premier noeud fils dont le nom est fourni
removeChild(String, Namespace)	Supprimer le premier noeud fils dont le nom et l'espace de nommage sont fournis
removeChildren(String)	Supprimer les noeuds fils dont le nom est fourni

removeChildren(String, Namespace)	Supprimer les noeuds fils dont le nom et l'espace de nommage sont fournis

La méthode removeChildren() ne supprime que les éléments fils de premier niveau répondant aux critères de nom et d'espace de nommage fournis.

Si aucun espace de nommage n'est fourni, seuls les éléments sans espace de nommage entrent dans les critères. Les autres noeuds fils tels que du texte, des commentaires ou des instructions de traitement ne sont pas supprimés.

La suppression d'un élément supprime l'élément mais aussi toute l'arborescence fille de l'élément.

Il est aussi possible d'obtenir une collection des noeuds fils en utilisant la méthode getContent() et de manipuler le contenu de cette collection en utilisant les méthodes add() et remove() de la collection.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.File;
import java.io.IOException;
import java.util.Iterator;
import java.util.List;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM70 extends TestJDOM {

  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File("bibliotheque.xml"));

      Element racine = document.getRootElement();
      supprimerNoeudsTousParNom(racine, "auteur");

      afficher(document);
    } catch(JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  /**
   * Supprime récursivement tous les éléments ayant pour nom celui fourni en paramètre
   * @param element element à traiter
   * @param nom nom des éléments à supprimer
   */
  public static void supprimerNoeudsTousParNom(Element element, String nom) {

    List elements = element.getChildren(nom);
    elements.removeAll(elements);

    // recherche de nouveau des fils puisque la collection a potentiellement été modifiée
    List fils = element.getChildren();
    Iterator iterator = fils.iterator();
    while (iterator.hasNext()) {
      supprimerNoeudsTousParNom((Element) iterator.next(), nom);
    }
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
```

```
  <livre>
    <!-- commentaires livre 1 -->
    <titre>titre1</titre>
    <editeur>editeur1</editeur>
  </livre>
  <livre>
    <!-- commentaires livre 2 -->
    <titre>titre2</titre>
    <editeur>editeur2</editeur>
  </livre>
  <livre>
    <!-- commentaires livre 3 -->
    <titre>titre3</titre>
    <editeur>editeur3</editeur>
  </livre>
</bibliotheque>
```

52.1.8.5. Le déplacement d'un ou des éléments

Lors de la création d'une instance de type Element, cet élément n'est pas associé à un document. Dans ce cas, la méthode getDocument() renvoie null.

JDOM interdit cependant qu'un Element soit inclus dans deux documents.

Pour déplacer un élément dans un même document ou dans un autre document, il est nécessaire d'invoquer sa méthode detach() au préalable.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.Namespace;
import org.jdom.input.SAXBuilder;

public class TestJDOM56 extends TestJDOM {

  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document documentSource = builder.build(new File("bibliotheque.xml"));

      Namespace ns = Namespace.getNamespace("http://www.jmdoudoux.com");

      Element elementLivre = documentSource.getRootElement().getChild("livre", ns);

      Element elementRacine = new Element("Bibliotheque");
      Document documentCible = new Document(elementRacine);

      elementLivre.detach();
      elementRacine.addContent(elementLivre);

      afficher(documentSource);
      afficher(documentCible);
    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque xmlns="http://www.jmdoudoux.com">
  <livre>
    <!-- commentaires livre 2 -->
    <titre>titre2</titre>
    <auteur>auteur2</auteur>
    <editeur>editeur2</editeur>
  </livre>
  <livre>
    <!-- commentaires livre 3 -->
    <titre>titre3</titre>
    <auteur>auteur3</auteur>
    <editeur>editeur3</editeur>
  </livre>
</bibliotheque>

<?xml version="1.0" encoding="UTF-8"?>
<Bibliotheque>
  <livre xmlns="http://www.jmdoudoux.com">
    <!-- commentaires livre 1 -->
    <titre>titre1</titre>
    <auteur>auteur1</auteur>
    <editeur>editeur1</editeur>
  </livre>
</Bibliotheque>
```

Le déplacement d'un élément implique le déplacement de ses noeuds fils. Les espaces de nommage sont aussi gérés lors de ce déplacement.

52.1.8.6. La duplication d'un élément

Il arrive parfois qu'il faille dupliquer un élément. Pour cela, il suffit simplement d'utiliser la méthode clone() sur l'instance de l'Element.

L'élément est dupliqué en incluant tous ses noeuds descendants.

Il ne reste plus qu'à ajouter le nouvel élément dans l'arborescence du document.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM67 extends TestJDOM {
  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File("bibliotheque.xml"));

      Element racine = document.getRootElement();

      Element livre2 = (Element) racine.getChildren().get(1);

      racine.addContent((Element) livre2.clone());

      afficher(document);
    } catch(JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
  <livre>
    <!-- commentaires livre 1 -->
    <titre>titre1</titre>
    <auteur>auteur1</auteur>
    <editeur>editeur1</editeur>
  </livre>
  <livre>
    <!-- commentaires livre 2 -->
    <titre>titre2</titre>
    <auteur>auteur2</auteur>
    <editeur>editeur2</editeur>
  </livre>
  <livre>
    <!-- commentaires livre 3 -->
    <titre>titre3</titre>
    <auteur>auteur3</auteur>
    <editeur>editeur3</editeur>
  </livre>
  <livre>
    <!-- commentaires livre 2 -->
    <titre>titre2</titre>
    <auteur>auteur2</auteur>
    <editeur>editeur2</editeur>
  </livre>
</bibliotheque>
```

La méthode cloneContent() renvoie une collection de noeuds fils dupliqués.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM68 extends TestJDOM {

  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File("bibliotheque.xml"));

      Element racine = document.getRootElement();

      Element livre2 = (Element) racine.getChildren().get(1);

      racine.addContent(livre2.cloneContent());

      afficher(document);
    } catch(JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
  <livre>
    <!-- commentaires livre 1 -->
    <titre>titre1</titre>
    <auteur>auteur1</auteur>
    <editeur>editeur1</editeur>
  </livre>
  <livre>
    <!-- commentaires livre 2 -->
    <titre>titre2</titre>
    <auteur>auteur2</auteur>
    <editeur>editeur2</editeur>
  </livre>
  <livre>
    <!-- commentaires livre 3 -->
    <titre>titre3</titre>
    <auteur>auteur3</auteur>
    <editeur>editeur3</editeur>
  </livre>
  <!-- commentaires livre 2 -->
  <titre>titre2</titre>
  <auteur>auteur2</auteur>
  <editeur>editeur2</editeur>
</bibliotheque>
```

52.1.9. L'utilisation de filtres

JDOM propose l'interface org.jdom.filter.Filter qui permet de définir un filtre.

L'interface Filter ne définit que la méthode matches() qui attend en paramètre un objet correspondant à un noeud et renvoie un booléen pour préciser si l'objet répond ou non aux critères du filtre.

Ce filtre peut être utilisé par plusieurs méthodes de certaines classes de JDOM pour restreindre leur action sur les entités qui répondent aux critères du filtre.

JDOM propose deux implémentations de l'interface Filter :

- ContentFilter : permet de filtrer sur le type de noeuds
- ElementFilter : permet de filtrer sur le nom et/ou l'espace de nommage des éléments

La classe ContentFilter possède plusieurs constructeurs :

Constructeur	Rôle
ContentFilter()	Filtre acceptant tous les types de noeuds
ContentFilter(int)	Filtre acceptant uniquement les noeuds précisés dans le masque (le masque utilise les constantes définies dans la classe ContentFilter)
ContentFilter(boolean)	Filtre acceptant ou non tous les types de noeuds selon la valeur du paramètre

La classe ContentFilter propose plusieurs méthodes setXXXVisible() qui attendent un paramètre de type booléen permettant d'inclure ou non les noeuds de type XXX.

Exemple : afficher tous les commentaires du document

```
package fr.jmdoudoux.dej.jdom;

import java.io.File;
import java.io.IOException;
import java.util.Iterator;
import java.util.List;

import org.jdom.Comment;
import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.filter.ContentFilter;
import org.jdom.input.SAXBuilder;

public class TestJDOM57 extends TestJDOM {

  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document documentSource = builder.build(new File("bibliotheque.xml"));

      Element elementRacine = documentSource.getRootElement();
      process(elementRacine);

    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }

  public static void process(Element element) {
    List children = element.getContent();
    Iterator iterator = children.iterator();
    while (iterator.hasNext()) {
      Object o = iterator.next();
      if (o instanceof Element) {
        Element child = (Element) o;
        afficherCommentaires(child);
        process(child);
      }
    }
  }

  public static void afficherCommentaires(Element element) {
    ContentFilter filtre = new ContentFilter(false);
    filtre.setCommentVisible(true);

    List children = element.getContent(filtre);
    Iterator iterator = children.iterator();
    while (iterator.hasNext()) {
      Comment comment = (Comment) iterator.next();
      System.out.println(comment.getText());
    }
  }
}
```

Résultat :

```
commentaires livre 1
commentaires livre 2
commentaires livre 3
```

Le classe ElementFilter possède plusieurs constructeurs :

Constructeur	Rôle
ElementFilter()	Filtre qui ne renvoie que les noeuds de type Element
ElementFilter(String)	Filtre qui ne renvoie que les éléments dont le nom est fourni
ElementFilter(Namespace)	Filtre qui ne renvoie que les éléments dont l'espace de nommage est fourni
ElementFilter(String, Namespace)	Filtre qui ne renvoie que les éléments dont le nom et l'espace de nommage sont fournis

Exemple : afficher le titre de tous les livres

```
package fr.jmdoudoux.dej.jdom;

import java.io.File;
import java.io.IOException;
import java.util.Iterator;
import java.util.List;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.filter.ElementFilter;
import org.jdom.input.SAXBuilder;

public class TestJDOM58 extends TestJDOM {

  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document documentSource = builder.build(new File("bibliotheque.xml"));

      Element elementRacine = documentSource.getRootElement();
      Iterator iterator = elementRacine.getContent().iterator();
      while (iterator.hasNext()) {
        Object o = iterator.next();
        if (o instanceof Element) {
          Element child = (Element) o;
          afficherTitre(child);
        }
      }
    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }

  public static void afficherTitre(Element element) {
    ElementFilter filtre = new ElementFilter("titre");

    List children = element.getContent(filtre);
    Iterator iterator = children.iterator();
    while (iterator.hasNext()) {
      Element fils = (Element) iterator.next();
      System.out.println(fils.getText());
    }
  }
}
```

Résultat :

```
titre1
titre2
titre3
```

Il est aussi possible de définir son propre filtre en créant une classe qui implémente l'interface Filter. Il suffit de définir la méthode matches() en lui faisant renvoyer un booléen indiquant le résultat de l'application des critères de filtre sur l'objet fourni en paramètres.

Exemple : afficher les auteurs de chaque livre

Exemple : afficher les auteurs de chaque livre

```
package fr.jmdoudoux.dej.jdom;

import java.io.File;
import java.io.IOException;
import java.util.Iterator;
import java.util.List;
import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.filter.Filter;
import org.jdom.input.SAXBuilder;

public class TestJDOM59 extends TestJDOM {

  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document documentSource = builder.build(new File("bibliotheque.xml"));

      Element elementRacine = documentSource.getRootElement();
      Iterator iterator = elementRacine.getContent().iterator();
      while (iterator.hasNext()) {
        Object o = iterator.next();
        if (o instanceof Element) {
          Element child = (Element) o;
          afficherTitre(child);
        }
      }
    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }

  public static void afficherTitre(Element element) {
    Filter filtre = new Filter() {
      private static final long serialVersionUID = 1L;

      public boolean matches(Object arg0) {
        boolean resultat = false;

        if(arg0 instanceof Element){
          Element element = (Element)arg0;
          resultat = "auteur".equals(element.getName());
        }
        return resultat;
      }
    };

    List children = element.getContent(filtre);
    Iterator iterator = children.iterator();
    while (iterator.hasNext()) {
      Element fils = (Element) iterator.next();
      System.out.println(fils.getText());
    }
  }
}
```

Résultat :

```
auteur1
auteur2
auteur3
```

Cet exemple n'a qu'un intérêt pédagogique puisque la même opération peut être réalisée en utilisant la méthode getChildText(). En pratique les filtres personnalisés sont plus complexes.

52.1.10. L'exportation d'un document

JDOM prévoit plusieurs classes pour permettre d'exporter le document contenu dans un objet de type Document. Cette exportation peut se faire :

- Sous la forme de flux : OutputStream ou Writer
- Vers d'autres API : Event Stream (SAX) ou Document (JDOM)

Les classes nécessaires à ces traitements sont regroupées dans le package org.jdom.output.

52.1.10.1. L'exportation dans un flux

La classe XMLOutputter permet d'envoyer le document XML dans un flux. Il est possible de fournir plusieurs paramètres pour formater la sortie du document.

Cette classe possède plusieurs constructeurs dont les principaux sont :

Constructeur	Rôle
XMLOutputter()	Créer un objet par défaut, sans paramètre de formatage
XMLOutputter(Format)	Créer un objet, en précisant en paramètre les options de formattage

La mise en oeuvre de cette classe est très simple : il suffit d'instancier un objet de type XMLOutputter et d'invoquer sa méthode output().

Exemple : lecture et exportation sur la console

```
package fr.jmdoudoux.dej.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;
import org.jdom.output.XMLOutputter;

public class TestJDOM34 {

  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File("test.xml"));
      XMLOutputter sortie = new XMLOutputter();
      sortie.output(document, System.out);

    } catch(JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

La classe XMLOuputter possède de nombreuses surcharges de la méthode output() permettant l'exportation dans un flux OutputStream ou Writer de différentes entités (Document, Element, Comment, Text, EntityRef, ProcessingInstruction, DocType, ...)

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.IOException;

import org.jdom.Element;
import org.jdom.output.XMLOutputter;

public class TestJDOM40 {
  public static void main(String[] args) {
    try {
      Element racine = new Element("html");
      Element body = new Element("body");
      racine.addContent(body);
      Element titre = new Element("H1");
      titre.setText("titre");
      body.addContent(titre);

      XMLOutputter sortie = new XMLOutputter();
      sortie.output(racine, System.out);
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
<html><body><H1>titre</H1></body></html>
```

Il est possible de configurer les options de formattage de l'exportation en utilisant un objet de type org.jdom.output.Format.

La classe Format propose trois formats prédéfinis que l'on peut obtenir en invoquant la méthode statique correspondante :

- CompactFormat
- PrettyFormat
- RawFormat

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;
import org.jdom.output.Format;
import org.jdom.output.XMLOutputter;

public class TestJDOM35 {

  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File("test.xml"));
      XMLOutputter sortie = new XMLOutputter(Format.getCompactFormat());
      sortie.output(document, System.out);

    } catch(JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat avec CompactFormat

```
<?xml version="1.0" encoding="UTF-8"?>
<html><head /><body><table width="100%" border="0" /></body></html>
```

Résultat avec PrettyFormat

```
<?xml
version="1.0" encoding="UTF-8"?>
<html>
  <head />
  <body>
    <table width="100%" border="0" />
  </body>
</html>
```

Résultat avec RawFormat

```
<?xml version="1.0" encoding="UTF-8"?>
<html>
<head />
<body>
<table width="100%" border="0" />
</body>
</html>
```

Pour des besoins plus spécifiques, il est possible d'instancier un objet Format et de le configurer en utilisant ses différentes méthodes.

Par défaut, XMLOutputter utilise l'encodage des caractères en UTF-8. Pour préciser un autre encodage des caractères, il faut utiliser la méthode setEncoding() de l'objet Format utilisé par XMLOutputter.

Le flux utilisé peut être de type OutputStream ou Writer. L'utilisation d'un OutputStream est plus facile car le Writer impose de préciser l'encodage de caractères correspondant à celui déclaré dans le prologue du document.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStreamWriter;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.output.Format;
import org.jdom.output.XMLOutputter;

public class TestJDOM38 extends TestJDOM {

  public static void main(String[] args) {

    Element racine = new Element("bibliotheque");
    Document document = new Document(racine);
    Element livres = new Element("livres");
    racine.addContent(livres);

    Element livre = new Element("livre");
    livres.addContent(livre);

    Element titre = new Element("titre").setText("Titre livre 1");
    Element auteur = new Element("auteur").setText("Auteur 1");
    livre.addContent(titre);
    livre.addContent(auteur);

    Format format = Format.getPrettyFormat();
    format.setEncoding("ISO-8859-1");
    XMLOutputter sortie = new XMLOutputter(format);

    FileOutputStream fos = null;
    try {
      fos = new FileOutputStream("c:/temp/test.xml");
      OutputStreamWriter out = new OutputStreamWriter(fos, "ISO-8859-1");
      sortie.output(document, out);
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      if (fos != null) {
        try {
          fos.close();
        } catch (IOException e) {
          e.printStackTrace();
        }
      }
    }
  }
}
```

La classe XMLOutputter propose aussi la méthode ouputString() qui exporte différentes entités selon la surcharge utilisée dans une chaîne de caractères.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;
import org.jdom.output.Format;
import org.jdom.output.XMLOutputter;

public class TestJDOM36 {

  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File("test.xml"));
      XMLOutputter sortie = new XMLOutputter(Format.getCompactFormat());
      String docXML = sortie.outputString(document);
      System.out.println(docXML);

    } catch(JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

La classe XMLOutputter se charge de convertir les caractères utilisés par XML en leurs entités respectives durant l'exportation.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.IOException;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.output.Format;
import org.jdom.output.XMLOutputter;

public class TestJDOM37  {

  public static void main(String[] args) {
    Element racine = new Element("personne");
    Document document = new Document(racine);
    Element adresse = new Element("adresse");
    adresse.addContent("mon adresse < 5 et > 10 & impaire ");
    racine.addContent(adresse);
    XMLOutputter sortie = new XMLOutputter(Format.getRawFormat());
    try {
      sortie.output(document, System.out);
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<personne><adresse>mon adresse &lt; 5 et &gt; 10 &amp; impaire </adresse></personne>
```

52.1.10.2. L'exportation dans un arbre DOM

La classe org.jdom.ouput.DOMOutputter permet d'exporter un document JDOM dans un arbre DOM.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;
import org.jdom.output.DOMOutputter;

public class TestJDOM39 {

  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File("test.xml"));

      DOMOutputter domOutputter = new DOMOutputter();
      org.w3c.dom.Document documentDOM = domOutputter.output(document);
    } catch(JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Cette exportation est plutôt une conversion de l'arbre d'objets du modèle JDOM vers le modèle DOM.

Les deux arbres d'objets sont indépendants l'un de l'autre : une modification dans l'arbre JDOM après l'exportation doit être faite aussi dans l'arbre DOM ou il est nécessaire de refaire une exportation pour refléter la modification dans l'arbre DOM

52.1.10.3. L'exportation en SAX

La classe SAXOutputter permet de générer des événements SAX à partir d'un document JDOM.

La classe SAXOutputter possède plusieurs constructeurs qui attendent tous un objet de type ContentHandler et certains des objets de type ErrorHandler, DTDHandler, EntityResolver et LexicalHandler.

La méthode output() parcourt l'arbre JDOM et émet les événements SAX correspondants qui seront traités par les handlers.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;
import org.jdom.output.SAXOutputter;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class TestJDOM50 {

  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document document = builder.build(new File("bibliotheque.xml"));

      MonSAXHandler handler = new MonSAXHandler();
      SAXOutputter outputter = new SAXOutputter(handler);
      outputter.setErrorHandler(handler);
      outputter.setDTDHandler(handler);
      outputter.output(document);

    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}

class MonSAXHandler extends DefaultHandler {
  private String tagCourant = "";

  /**
   * Actions à réaliser lors de la détection d'un nouvel élément.
   */
  public void startElement(String nameSpace,String localName,String qName,
    Attributes attr) throws SAXException {
    tagCourant = localName;
    System.out.println("debut tag : " + localName);
  }

  /**
   * Actions à réaliser lors de la détection de la fin d'un élément.
   */
  public void endElement(String nameSpace, String localName, String qName)
    throws SAXException {
    tagCourant = "";
    System.out.println("Fin tag " + localName);
  }

  /**
   * Actions à réaliser au début du document.
   */
  public void startDocument() {
    System.out.println("Debut du document");
  }

  /**
   * Actions à réaliser lors de la fin du document XML.
   */
  public void endDocument() {
    System.out.println("Fin du document");
  }

  /**
   * Actions à réaliser sur les données
   */
  public void characters(char[] caracteres, int debut, int longueur) throws SAXException {
    String donnees = new String(caracteres, debut, longueur);
    if (!tagCourant.equals("")) {
      if (!Character.isISOControl(caracteres[debut])) {
        System.out.println(" Element " + tagCourant + ", valeur = *" + donnees + "*");
      }
    }
  }
}
```

Résultat :

```
Debut du document
debut tag : bibliotheque
debut tag : livre
debut tag : titre
 Element titre, valeur = *titre1*
Fin tag titre
debut tag : auteur
 Element auteur, valeur = *auteur1*
Fin tag auteur
debut tag : editeur
 Element editeur, valeur = *editeur1*
Fin tag editeur
Fin tag livre
debut tag : livre
debut tag : titre
 Element titre, valeur = *titre2*
Fin tag titre
debut tag : auteur
 Element auteur, valeur = *auteur2*
Fin tag auteur
debut tag : editeur
 Element editeur, valeur = *editeur2*
Fin tag editeur
Fin tag livre
debut tag : livre
debut tag : titre
 Element titre, valeur = *titre3*
Fin tag titre
debut tag : auteur
 Element auteur, valeur = *auteur3*
Fin tag auteur
debut tag : editeur
 Element editeur, valeur = *editeur3*
Fin tag editeur
Fin tag livre
Fin tag bibliotheque
Fin du document
```

52.1.11. L'utilisation de XSLT

La classe org.jdom.transform.XSLTransformer est un helper qui facilite la mise en oeuvre de transformations simples. Cette classe utilise l'API TrAX de JAXP. Le moteur de transformation utilisé doit être paramétré en utilisant JAXP.

La classe XSLTransformer possède plusieurs constructeurs qui attendent en paramètre une feuille de style XSL.

Elle possède plusieurs surcharges de la méthode transform() qui appliquent la feuille de style à un document ou un ensemble de noeuds.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;
import org.jdom.transform.XSLTransformer;

public class TestJDOM62 extends TestJDOM {

  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document documentSource = builder.build(new File("bibliotheque.xml"));

      XSLTransformer transformer = new XSLTransformer("bibliotheque.xsl");
      Document documentCible = transformer.transform(documentSource);

      afficher(documentCible);

    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}
```

Exemple : la feuille de style bibliotheque.xsl

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
    <body>
      <h2>bibliotheque</h2>
      <table border="1">
        <tr bgcolor="lightblue">
          <th align="left">Titre</th>
          <th align="left">Auteur</th>
        </tr>
        <xsl:for-each select="bibliotheque/livre">
          <tr>
            <td>
              <xsl:value-of select="titre" />
            </td>
            <td>
              <xsl:value-of select="auteur" />
            </td>
          </tr>
        </xsl:for-each>
      </table>
    </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<html>
  <body>
    <h2>bibliotheque</h2>
    <table border="1">
      <tr bgcolor="lightblue">
        <th align="left">Titre</th>
        <th align="left">Auteur</th>
      </tr>
      <tr>
        <td>titre1</td>
        <td>auteur1</td>
      </tr>
      <tr>
        <td>titre2</td>
        <td>auteur2</td>
      </tr>
      <tr>
        <td>titre3</td>
        <td>auteur3</td>
      </tr>
    </table>
  </body>
</html>
```

Les méthodes transform() qui attendent en paramètre un objet de type document retournent un objet JDOM de type Document encapsulant le résultat de la transformation. Ceci est particulièrement adapté lorsqu'un document XML est transformé en un autre document XML.

Pour des besoins plus spécifiques, il est possible d'obtenir une instance de la classe Transformer de JAXP et d'utiliser les classes org.jdom.transform.JDOMSource et JDOMResult comme wrapper respectivement en entrée et en sortie de la transformation.

Ceci est nécessaire par exemple lorsque des paramètres doivent être passés à la feuille de style.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.File;
import java.io.IOException;

import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamSource;

import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;
import org.jdom.transform.JDOMResult;
import org.jdom.transform.JDOMSource;

public class TestJDOM75 extends TestJDOM {

  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document documentSource = builder.build(new File("bibliotheque.xml"));

      TransformerFactory factory = TransformerFactory.newInstance();
      Transformer transformer = factory.newTransformer(new StreamSource("biblio.xsl"));

      JDOMSource source = new JDOMSource(documentSource);
      JDOMResult resultat = new JDOMResult();

      transformer.setParameter("libelle", "mon libelle");
      transformer.transform(source, resultat);

      afficher(resultat.getDocument());
    } catch (JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    } catch (TransformerException e) {
      e.printStackTrace();
    }
  }
}
```

Exemple : la feuille de style qui déclare et utilise un paramètre nommé libelle

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:param name="libelle" select=""/>
<xsl:template match="/">
  <html>
    <body>
      <h2>bibliotheque : <xsl:value-of select="$libelle"/> </h2>
      <table border="1">
        <tr bgcolor="lightblue">
          <th align="left">Titre</th>
          <th align="left">Auteur</th>
        </tr>
        <xsl:for-each select="bibliotheque/livre">
          <tr>
            <td>
              <xsl:value-of select="titre" />
            </td>
            <td>
              <xsl:value-of select="auteur" />
            </td>
          </tr>
        </xsl:for-each>
      </table>
    </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<html>
  <body>
    <h2>bibliotheque : mon libelle</h2>
    <table border="1">
      <tr bgcolor="lightblue">
        <th align="left">Titre</th>
        <th align="left">Auteur</th>
      </tr>
      <tr>
        <td>titre1</td>
        <td>auteur1</td>
      </tr>
      <tr>
        <td>titre2</td>
        <td>auteur2</td>
      </tr>
      <tr>
        <td>titre3</td>
        <td>auteur3</td>
      </tr>
    </table>
  </body>
</html>
```

52.1.12. L'utilisation de XPath

JDOM propose un support de XPath en utilisant Jaxen depuis sa version bêta 9.

Les bibliothèques de Jaxen doivent être ajoutées au classpath : elles sont fournies dans le sous-répertoire lib de l'archive binaire de JDOM.

Elles peuvent aussi être téléchargées sur le site https://jaxen.org/

Il faut ajouter au classpath les bibliothèques : jaxen-core.jar, jaxen-jdom.jar et saxpath.jar.

Il faut instancier un objet de type org.jdom.xpath.XPath en utilisant sa méthode statique newInstance() qui attend en paramètre l'expression XPath.

L'invocation de la méthode selectNodes() sur cette instance en lui passant en paramètre le document renvoie une collection des noeuds qui répond à l'expression XPath.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.File;
import java.io.IOException;
import java.util.Iterator;
import java.util.List;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;
import org.jdom.xpath.XPath;

public class TestJDOM60 extends TestJDOM {

  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document document = builder.build(new File("bibliotheque.xml"));

      XPath x      = XPath.newInstance("/bibliotheque/livre");
      List list    = x.selectNodes(document);

      System.out.println("nb livres="+list.size());

      Iterator iterator = list.iterator();
      while (iterator.hasNext()) {
        Object o = iterator.next();
        if (o instanceof Element) {
          Element livre = (Element) o;
          System.out.println("livre : "+livre.getChildText("titre"));
        }
      }
    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}
```

Résultat :

```
nb livres=3
livre : titre1
livre : titre2
livre : titre3
```

Le type des objets contenus dans la collection retournée par la méthode dépend de l'expression XPath fournie.

Exemple :

```
package fr.jmdoudoux.dej.jdom;

import java.io.File;
import java.io.IOException;
import java.util.Iterator;
import java.util.List;

import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.Text;
import org.jdom.input.SAXBuilder;
import org.jdom.xpath.XPath;

public class TestJDOM61 {

  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document document = builder.build(new File("bibliotheque.xml"));

      XPath x      = XPath.newInstance("/bibliotheque/livre/titre/text()");
      List list    = x.selectNodes(document);

      System.out.println("nb titres="+list.size());

      Iterator iterator = list.iterator();
      while (iterator.hasNext()) {
        Text texte  = (Text) iterator.next();
        System.out.println("livre : "+texte.getValue());
      }
    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}
```

Résultat :

```
nb titres=3
livre : titre1
livre : titre2
livre : titre3
```

La méthode selectSingleNode() peut être utilisée à la place de la méthode selectNodes() lorsque l'expression ne renvoie qu'un seul noeud.

52.1.13. L'intégration à Java

JDOM est une API développée en Java pour Java : elle repose sur des API de Java telles que l'API Collection et met en oeuvre certaines fonctionnalités de Java notamment le clonage ou la sérialisation.

JDOM a délibérément été voulue non thread safe puisque cela devrait être la plus large utilisation de l'API. Pour une utilisation des objets dans un contexte multithread, il faut procéder manuellement à une synchronisation des portions de code critiques.

Les méthodes equals() sont redéfinies pour ne retourner l'égalité que si les deux objets sont exactement les mêmes (test sur les références effectué par l'opérateur ==).

Ce test d'égalité permet de garantir que l'élément cible est bien celui concerné notamment en tenant compte de sa position dans le document. Il peut, par exemple, y avoir plusieurs éléments avec le même nom et la même valeur dans un même document. Le fait d'avoir des instances distinctes garantit de manipuler l'élément concerné.

De plus, les méthodes equals() et hashCode() sont déclarées final pour ne pas permettre de déroger à cette règle de comparaison.

Les classes qui encapsulent des données du document implémentent l'interface Serializable (sauf la classe Namespace) ainsi ces objets peuvent être sérialisés pour les rendre persistants ou permettre leur échange à travers le réseau.

Les classes qui encapsulent des données du document redéfinissent la méthode toString() pour retourner une représentation textuelle des données qu'elles encapsulent entre crochets en précisant le type de l'entité.

Attention : la méthode toString() ne renvoie pas de représentation au format XML de l'entité. Pour obtenir cette représentation, il faut utiliser un objet de type XMLOutputter.

Les classes qui encapsulent des données du document implémentent l'interface Cloneable sauf la classe Namespace qui encapsule un objet immuable. Le clonage d'un élément se fait de façon récursive : seul le parent de l'objet original n'est pas repris dans la copie.

L'utilisation de l'API Collection pour encapsuler un ensemble d'entités implique que toutes les modifications sur une collection affecte le document.

De nombreuses méthodes de l'API JDOM peuvent lever une exception qui hérite de la classe JDOMException. Ceci permet de faire un traitement générique sur ces exceptions ou de faire un traitement sur une exception en particulier.

52.1.14. Les contraintes de la mise en oeuvre de JDOM

La mise en oeuvre de JDOM présente plusieurs contraintes dont il faut tenir compte.

L'utilisation de JDOM implique la création en mémoire de l'arbre d'objets encapsulant le document ce qui peut être difficile dans un environnement ayant peu de ressources, notamment mémoire, ou pour traiter de gros documents.

JDOM ne propose pas de support complet pour les DTD ou les schémas : il n'est pas possible de s'assurer que le document est valide lors d'une modification.

52.2. dom4j

dom4j est un framework open source pour manipuler des données XML, XSL et Xpath. Il est entièrement développé en Java et pour Java.

Dom4j n'est pas un parser mais propose un modèle de représentation d'un document XML et une API pour en faciliter l'utilisation. Pour obtenir une telle représentation, dom4j utilise soit SAX, soit DOM. Comme il est compatible JAXP, il est possible d'utiliser tout parser qui implémente cette API.

La version de dom4j utilisée dans cette section est la 1.3

52.2.1. L'installation de dom4j

Il faut télécharger la dernière version à l'url https://www.dom4j.org/download.html

Il suffit de décompresser le fichier téléchargé. Celui-ci contient de nombreuses bibliothèques (ant, xalan, xerces, crimson, junit, ...), le code source du projet et la documentation.

Le plus simple pour utiliser rapidement dom4j est de copier les fichiers jar contenus dans le répertoire lib dans le répertoire ext du répertoire %JAVA_HOME%/jre/lib/ext ainsi que les fichiers dom4j.jar et dom4j-full.jar.

52.2.2. La création d'un document

Dom4j encapsule un document dans un objet de type org.dom4j.Document. Dom4j propose des API pour facilement créer un tel objet qui va être le point d'entrée de la représentation d'un document XML .

Exemple utilisant SAX :

```
import org.dom4j.*;
import org.dom4j.io.*;

public class Testdom4j_1 {

  public static void main(String args[]) {
    DOCUMENT DOCUMENT;
    try {
      SAXReader xmlReader = new SAXReader();
      document = xmlReader.read("test.xml");
    } catch (Exception e){
      e.printStackTrace();
    }
  }
}
```

Pour exécuter ce code, il suffit d'exécuter

java -cp .;%JAVA_HOME%/jre/lib/ext/dom4j-full.jar Testdom4j_1

Exemple à partir d'une chaîne de caractères :

```
import org.dom4j.*;

public class Testdom4j_8 {

  public static void main(String args[]) {
    Document document = null;
    String texte = "<bibliotheque><livre><titre>titre 1</titre><auteur>auteur 1</auteur>"
         +"<editeur>editeur 1</editeur></livre></bibliotheque>";
    try {
      document = DocumentHelper.parseText(texte);
    } catch (Exception e){
     e.printStackTrace();
    }
  }
}
```

52.2.3. Le parcours d'un document

Le parcours du document construit peut se faire de plusieurs façons :

- utilisation de l'API collection
- utilisation de XPath
- utilisation du pattern Visitor

Le parcours peut se faire en utilisant l'API collection de Java.

Exemple : obtenir tous les noeuds fils du noeud racine

```
import org.dom4j.*;
import org.dom4j.io.*;
import java.util.*;

public class Testdom4j_2 {

  public static void main(String args[]) {
    Document document;
    org.dom4j.Element racine;
    try {
      SAXReader xmlReader = new SAXReader();
      document = xmlReader.read("test.xml");
      racine = document.getRootElement();
      Iterator it = racine.elementIterator();
      while(it.hasNext()){
        Element element = (Element)it.next();
        System.out.println(element.getName());
      }
    } catch (Exception e){
      e.printStackTrace();
    }
  }
}
```

Un des grands intérêts de dom4j est de proposer une recherche dans le document en utilisant la technologie Xpath.

Exemple : obtenir tous les noeuds fils du noeud racine

```
import org.dom4j.*;
import org.dom4j.io.*;
import java.util.*;

public class Testdom4j_3 {

  public static void main(String args[]) {
    Document document;
    try {
      SAXReader xmlReader = new SAXReader();
      document = xmlReader.read("test.xml");
      XPath xpathSelector = DocumentHelper.createXPath("/bibliotheque/livre/auteur");
      List liste = xpathSelector.selectNodes(document);
      for ( Iterator it = liste.iterator(); it.hasNext(); ) {
        Element element = (Element) it.next();
        System.out.println(element.getName()+" : "+element.getText());
      }
    } catch (Exception e){
      e.printStackTrace();
    }
  }
}
```

52.2.4. La modification d'un document XML

L'interface Document propose plusieurs méthodes pour modifier la structure du document.

Méthode	Rôle
Document addComment(String)	Ajouter un commentaire
void setDocType(DocumentType)	Modifier les caractéristiques du type de document
void setRootElement(Element)	Modifier l'élément racine du document

L'interface Element propose plusieurs méthodes pour modifier un élément du document.

Méthode	Rôle
void add(...)	Méthode surchargée qui permet d'ajouter un attribut, une entité, un espace nommage ou du texte à l'élément
Element addAttribute(String, String)	Ajouter un attribut à l'élément
Element addComment(String)	Ajouter un commentaire à l'élément
Element addEntity(String, String)	Ajouter une entité à l'élément
Element addNameSpace(String, String)	Ajouter un espace de nommage à l'élément
Element addText(String)	Ajouter un text à l'élément

52.2.5. La création d'un nouveau document XML

Il est très facile de créer un document XML

La classe DocumentHelper propose une méthode createDocument() qui renvoie une nouvelle instance de la classe Document. Il suffit alors d'ajouter chacun des noeuds de l'arbre du nouveau document XML en utilisant la méthode addElement() de la classe Document.

Exemple :

```
import org.dom4j.*;

public class Testdom4j_4 {

  public static void main(String args[]) {
    Document document = DocumentHelper.createDocument();
    Element root = document.addElement( "bibliotheque" );
    Element livre = null;
    try {
    livre = root.addElement("livre");
    livre.addElement("titre").addText("titre 1");
    livre.addElement("auteur").addText("auteur 1");
    livre.addElement("editeur").addText("editeur 1");
    } catch (Exception e){
      e.printStackTrace();
    }
  }
}
```

52.2.6. L'exportation d'un document

Pour écrire le document XML dans un fichier, une méthode de la classe Document permet de réaliser cette action très simplement

Exemple :

```
import org.dom4j.*;
import java.io.*;

public class Testdom4j_5 {

  public static void main(String args[]) {
    Document document = DocumentHelper.createDocument();
    Element root = document.addElement( "bibliotheque" );
    Element livre = null;
    try {
      livre = root.addElement("livre");
      livre.addElement("titre").addText("titre 1");
      livre.addElement("auteur").addText("auteur 1");
      livre.addElement("editeur").addText("editeur 1");
      livre = root.addElement("livre");
      livre.addElement("titre").addText("titre 2");
      livre.addElement("auteur").addText("auteur 2");
      livre.addElement("editeur").addText("editeur 2");
      livre = root.addElement("livre");
      livre.addElement("titre").addText("titre 3");
      livre.addElement("auteur").addText("auteur 3");
      livre.addElement("editeur").addText("editeur 3");
      FileWriter out = new FileWriter( "test2.xml" );
      document.write( out );
      out.close();
    } catch (Exception e){
      e.printStackTrace();
    }
  }
}
```

Pour pouvoir agir sur le formatage du document ou pour utiliser un flux différent, il faut utiliser la classe XMLWriter

Exemple :

```
import org.dom4j.*;
import org.dom4j.io.*;
import java.io.*;

public class Testdom4j_6 {
  public static void main(String args[]) {
    Document document = DocumentHelper.createDocument();
    Element root = document.addElement( "bibliotheque" );
    Element livre = null;
    try {
      livre = root.addElement("livre");
      livre.addElement("titre").addText("titre 1");
      livre.addElement("auteur").addText("auteur 1");
      livre.addElement("editeur").addText("editeur 1");
      livre = root.addElement("livre");
      livre.addElement("titre").addText("titre 2");
      livre.addElement("auteur").addText("auteur 2");
      livre.addElement("editeur").addText("editeur 2");
      livre = root.addElement("livre");
      livre.addElement("titre").addText("titre 3");
      livre.addElement("auteur").addText("auteur 3");
      livre.addElement("editeur").addText("editeur 3");
      OutputFormat format = OutputFormat.createPrettyPrint();
      XMLWriter writer = new XMLWriter( System.out, format );
      writer.write( document );
    } catch (Exception e){
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
C:\test_dom4j>java -cp .;c:\j2sdk1.4.0_01/jre/lib/ext/dom4j
-full.jar Testdom4j_6
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
  <livre>
    <titre>titre 1</titre>
    <auteur>auteur 1</auteur>
    <editeur>editeur 1</editeur>
  </livre>
  <livre>
    <titre>titre 2</titre>
    <auteur>auteur 2</auteur>
    <editeur>editeur 2</editeur>
  </livre>
  <livre>
    <titre>titre 3</titre>
    <auteur>auteur 3</auteur>
    <editeur>editeur 3</editeur>
  </livre>
</bibliotheque>
```

La classe OutputFormat possède une méthode createPrettyPrint() qui renvoie un objet de type OutputFormat contenant des paramètres par défaut.

Il est possible d'obtenir une chaîne de caractères à partir de tout ou partie d'un document.

Exemple :

```
import org.dom4j.*;
import org.dom4j.io.*;

public class Testdom4j_7 {

  public static void main(String args[]) {
    Document document = DocumentHelper.createDocument();
    Element root = document.addElement( "bibliotheque" );
    Element livre = null;
    String texte = "";
    try {
      livre = root.addElement("livre");
      livre.addElement("titre").addText("titre 1");
      livre.addElement("auteur").addText("auteur 1");
      livre.addElement("editeur").addText("editeur 1");
      texte = document.asXML();
      System.out.println(texte);
    } catch (Exception e){
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
C:\test_dom4j>java -cp .;c:\j2sdk1.4.0_01/jre/lib/ext/dom4j
-full.jar Testdom4j_7
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque><livre><titre>titre 1</titre><auteur>auteur 1</auteur><editeur>edi
teur 1</editeur></livre></bibliotheque>
```

53. JAXB (Java Architecture for XML Binding)

Chapitre 53

Niveau : 🟩🟩🟨⬜⬜⬜ Intermédiaire

JAXB est une spécification qui permet de faire correspondre un document XML à un ensemble de classes et vice versa au moyen d'opérations de sérialisation/désérialisation nommées marshalling/unmarshalling.

JAXB permet aux développeurs de manipuler un document XML sans avoir à connaître XML ou la façon dont un document XML est traité comme cela est le cas avec SAX, DOM ou StAX. La manipulation du document XML se fait en utilisant des objets précédemment générés à partir d'une DTD pour JAXB 1.0 et d'un schéma XML du document à traiter pour JAXB 2.0.

Le page officiel de JAXB est à l'url : https://www.oracle.com/technical-resources/articles/javase/jaxb.html

Ce chapitre contient plusieurs sections :

- ♦ JAXB 1.0
- ♦ JAXB 2.0

53.1. JAXB 1.0

JAXB est l'acronyme de Java Architecture for XML Binding.

Le but de l'API et des spécifications JAXB est de faciliter la manipulation d'un document XML en générant un ensemble de classes qui fournissent un niveau d'abstraction plus élevé que l'utilisation de JAXP (SAX ou DOM). Avec ces deux API, toute la logique de traitements des données contenues dans le document est à écrire.

JAXB au contraire fournit un outil qui analyse un schéma XML et génère à partir de ce dernier un ensemble de classes qui vont encapsuler les traitements de manipulation du document.

Le grand avantage est de fournir au développeur un moyen de manipuler un document XML sans connaître XML ou les technologies d'analyse. Toutes les manipulations se font au travers d'objets Java.

Ces classes sont utilisées pour faire correspondre le document XML avec des instances de ces classes et vice versa : ces opérations se nomment respectivement unmarshalling et marshalling.

L'implémentation de référence de JAXB v1.0 est fournie avec le JSWDK 1.1.

Les exemples de ce chapitre utilisent cette implémentation de référence et le fichier XML suivant :

Exemple :

```
<bibliotheque>
  <livre>
    <titre>titre 1</titre>
    <auteur>auteur 1</auteur>
    <editeur>editeur 1</editeur>
  </livre>
  <livre>
    <titre>titre 2</titre>
    <auteur>auteur 2</auteur>
    <editeur>editeur 2</editeur>
  </livre>
  <livre>
    <titre>titre 3</titre>
    <auteur>auteur 3</auteur>
    <editeur>editeur 3</editeur>
  </livre>
</bibliotheque>
```

Le schéma XML correspondant à ce fichier XML est le suivant :

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
<xs:element name="bibliotheque">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="livre" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  </xs:element>
  <xs:element name="livre">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="titre"/>
        <xs:element ref="auteur"/>
        <xs:element ref="editeur"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="titre" type="xs:string"/>
  <xs:element name="auteur" type="xs:string"/>
  <xs:element name="editeur" type="xs:string"/>
</xs:schema>
```

53.1.1. La génération des classes

L'outil xjc permet d'analyser un schéma XML et de générer les interfaces et les classes qui vont permettre la manipulation d'un document XML qui respecte ce schéma. Cette opération se nomme binding.

La syntaxe de cet outil est très simple :

xjc [options] schema

schema est le nom d'un fichier contenant le schéma XML.

Les principales options sont les suivantes :

- -nv : ne pas réaliser une validation stricte du schéma fourni
- -d répertoire : permet de préciser le nom du répertoire qui va contenir les classes générées
- -p package : permet de préciser le nom du package qui va contenir les classes générées

Exemple :

```
C:\java\jaxb>xjc test.xsd
parsing a schema...
compiling a schema...
generated\impl\AuteurImpl.java
generated\impl\BibliothequeImpl.java
generated\impl\BibliothequeTypeImpl.java
generated\impl\EditeurImpl.java
generated\impl\LivreImpl.java
```

```
generated\impl\LivreTypeImpl.java
generated\impl\TitreImpl.java
generated\Auteur.java
generated\Bibliotheque.java
generated\BibliothequeType.java
generated\Editeur.java
generated\Livre.java
generated\LivreType.java
generated\ObjectFactory.java
generated\Titre.java
generated\bgm.ser
generated\jaxb.properties
```

L'exécution de la commande de l'exemple génère les fichiers suivants :

```
Generated
    Auteur.java
    bgm.ser
    Bibliotheque.java
    BibliothequeType.java
    Editeur.java
    jaxb.properties
    Livre.java
    LivreType.java
    ObjectFactory.java
    Titre.java
    generated\impl
        AuteurImpl.java
        BibliothequeImpl.java
        BibliothequeTypeImpl.java
        EditeurImpl.java
        LivreImpl.java
        LivreTypeImpl.java
        TitreImpl.java
```

Sans précision, les fichiers générés le sont dans le répertoire "Generated".

Pour préciser un autre répertoire, il faut utiliser l'option -d :

```
xjc -d sources test.xsd
```

Les classes et interfaces sont générées dans le répertoire "sources/generated"

Si le répertoire précisé n'existe pas, une exception est levée.

Exemple :

```
C:\java\jaxb>xjc -d sources test.xsd
parsing a schema...
compiling a schema...
java.lang.reflect.InvocationTargetException
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.
java:39)
```

Pour éviter l'utilisation du répertoire "generated", il faut préciser un package pour les entités générées en utilisant l'option -p.

Exemple :

```
C:\java\jaxb>xjc -d sources -p fr.jmdoudoux.dej.jaxb test.xsd
parsing a schema...
compiling a schema...
com\moi\test\jaxb\Auteur.java
com\moi\test\jaxb\Bibliotheque.java
com\moi\test\jaxb\BibliothequeType.java
com\moi\test\jaxb\Editeur.java
```

```
com\moi\test\jaxb\Livre.java
com\moi\test\jaxb\LivreType.java
com\moi\test\jaxb\ObjectFactory.java
com\moi\test\jaxb\Titre.java
com\moi\test\jaxb\jaxb.properties
com\moi\test\jaxb\bgm.ser
com\moi\test\jaxb\impl\AuteurImpl.java
com\moi\test\jaxb\impl\BibliothequeImpl.java
com\moi\test\jaxb\impl\BibliothequeTypeImpl.java
com\moi\test\jaxb\impl\EditeurImpl.java
com\moi\test\jaxb\impl\LivreImpl.java
com\moi\test\jaxb\impl\LivreTypeImpl.java
com\moi\test\jaxb\impl\TitreImpl.java
```

Un objet de type factory et des interfaces pour chacun des éléments qui composent le document sont définis.

Pour chaque élément qui peut contenir d'autres éléments, des interfaces de XXXType sont créées (BibliothequeType et LivreType dans l'exemple).

L'interface BibliothequeType définit simplement un getter sur une collection qui contiendra tous les livres.

L'interface LivreType définit des getters et des setters sur les éléments auteur, editeur et titre.

Les interfaces Titre, Editeur et Auteur définissent un getter et un setter sur les valeurs des données que ces éléments contiennent.

La classe ObjectFactory permet de créer des instances des différentes entités définies.

Le répertoire impl contient les classes qui implémentent ces interfaces. Ces classes sont spécifiques à l'implémentation des spécifications JAXB utilisée.

Pour pouvoir utiliser ces interfaces et ces classes, il faut les compiler en incluant au classpath, tous les fichiers .jar contenus dans le répertoire lib de l'implémentation de JAXB.

53.1.2. L'API JAXB

L'API JAXB propose un framework composé de classes regroupées dans trois packages :

- javax.xml.bind : contient les interfaces principales et la classe JAXBContext
- javax.xml.bind.util : contient des utilitaires
- javax.xml.bind.helper : contient une implémentation partielle de certaines interfaces pour faciliter le développement d'une implémentation des spécifications de JAXB

53.1.3. L'utilisation des classes générées et de l'API

Pour pouvoir utiliser JAXP, il faut tout d'abord obtenir un objet de type JAXBContext qui est le point d'entrée pour utiliser l'API. Il faut ensuite utiliser la méthode newInstance() qui attend en paramètre le nom du package contennant les interfaces générées (celui fournit au paramètre -p de la commande xjc).

Pour pouvoir créer en mémoire les objets qui représentent le document XML, il faut à partir de l'instance du type JAXBContext, appeler la méthode createUnmarshaller() qui renvoie un objet de type Unmarshaller.

L'appel de la méthode unmarshal() permet de créer les différents objets.

Pour parcourir le document, il suffit d'utiliser les différents objets instanciés.

Exemple :

```
package fr.jmdoudoux.dej.jaxb;
import javax.xml.bind.*;
import java.io.*;
import java.util.*;
public class TestJAXB {

  public static void main(String[] args) {

    try {
      JAXBContext jc = JAXBContext.newInstance("fr.jmdoudoux.dej.jaxb");
      Unmarshaller unmarshaller = jc.createUnmarshaller();
      unmarshaller.setValidating(true);

      Bibliotheque bibliotheque = (Bibliotheque) unmarshaller.unmarshal(new File("test.xml"));

      List livres = bibliotheque.getLivre();
      for (int i = 0; i < livres.size(); i++) {
        LivreType livre = (LivreType) livres.get(i);
        System.out.println("Livre ");
        System.out.println("Titre   : " + livre.getTitre());
        System.out.println("Auteur  : " + livre.getAuteur());
        System.out.println("Editeur : " + livre.getEditeur());
        System.out.println();
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
Livre
Titre   : titre 1
Auteur  : auteur 1
Editeur : editeur 1
Livre
Titre   : titre 2
Auteur  : auteur 2
Editeur : editeur 2
Livre
Titre   : titre 3
Auteur  : auteur 3
Editeur : editeur 3
```

53.1.4. La création d'un nouveau document XML

Parmi les classes générées à partir du schéma XML, il y a la classe ObjectFactory qui permet de créer des instances des autres classes générées.

Exemple :

```
import javax.xml.bind.*;
import java.io.*;
import java.util.*;
public class TestJAXB2 {

  public static void main(String[] args) {
    try {
      ObjectFactory objFactory = new ObjectFactory();

      Bibliotheque bibliotheque = (Bibliotheque) objFactory.createBibliotheque();
      List livres = bibliotheque.getLivre();
      for (int i = 1; i < 4; i++) {
        LivreType livreType = objFactory.createLivreType();
        //    LivreType livre = objFactory.createLivreType();
        livreType.setAuteur("Auteur" + i);
        livreType.setEditeur("Editeur" + i);
        livreType.setTitre("Titre" + i);
        livres.add(livreType);

      }

    } catch (Exception e) {

    }
  }
}
```

53.1.5. La génération d'un document XML

Une fois la représentation en mémoire du document XML créée ou modifiée, il est fréquent de devoir l'envoyer dans un flux tel qu'un fichier pour conserver les modifications. Cette opération se nomme marshalling.

Pour réaliser cette opération, il faut tout d'abord obtenir un objet du type JAXBContext en utilisant la méthode newInstance(). Cette méthode demande en paramètre une chaîne de caractères indiquant le package des interfaces générées à partir du schéma.

La méthode createMarshaller() permet d'obtenir un objet de type Marshaller. C'est cet objet qui va formater le document XML.

Il est possible de lui préciser des propriétés pour effectuer sa tâche en utilisant la méthode setProperty(). Des constantes sont définies pour ces propriétés dont les principales sont :

- JAXB_ENCODING : permet de préciser le jeu de caractères d'encodage du document XML sous la forme d'une chaîne de caractères
- JAXB_FORMATTED_OUTPUT : booléen qui indique si le document XML doit être formaté

Les propriétés doivent être des objets.

L'appel de la méthode marshal() formate le document dont l'objet racine est fourni en premier paramètre. Il existe plusieurs surcharges de cette méthode pour préciser où est envoyé le résultat de la génération. Le second paramètre permet de préciser cette cible : un flux en sortie, un arbre DOM, des événements SAX.

Exemple :

```
package fr.jmdoudoux.dej.jaxb;
import javax.xml.bind.*;
import java.io.*;
import java.util.*;
public class TestJAXB3 {

  public static void main(String[] args) {
    try {

      ObjectFactory objFactory = new ObjectFactory();

      Bibliotheque bibliotheque = (Bibliotheque) objFactory.createBibliotheque();
      List livres = bibliotheque.getLivre();
      for (int i = 1; i < 4; i++) {
        LivreType livreType = objFactory.createLivreType();
        //    LivreType livre = objFactory.createLivreType();
        livreType.setAuteur("Auteur" + i);
        livreType.setEditeur("Editeur" + i);
        livreType.setTitre("Titre" + i);
        livres.add(livreType);
      }
      JAXBContext jaxbContext = JAXBContext.newInstance("fr.jmdoudoux.dej.jaxb");
      Marshaller marshaller = jaxbContext.createMarshaller();
      marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, new Boolean(true));
      Validator validator = jaxbContext.createValidator();

      marshaller.marshal(bibliotheque, System.out);
    } catch (Exception e) {
    }
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<bibliotheque>
<livre>
<titre>Titre1</titre>
<auteur>Auteur1</auteur>
<editeur>Editeur1</editeur>
</livre>
<livre>
<titre>Titre2</titre>
<auteur>Auteur2</auteur>
<editeur>Editeur2</editeur>
</livre>
<livre>
<titre>Titre3</titre>
<auteur>Auteur3</auteur>
<editeur>Editeur3</editeur>
</livre>
</bibliotheque>
```

53.2. JAXB 2.0

JAXB 2.0 a été développé sous la JSR 222 et est incorporée dans Java EE 5 et dans Java SE 6.

Les fonctionnalités de JAXB 2.0 par rapport à JAXB 1.0 sont :

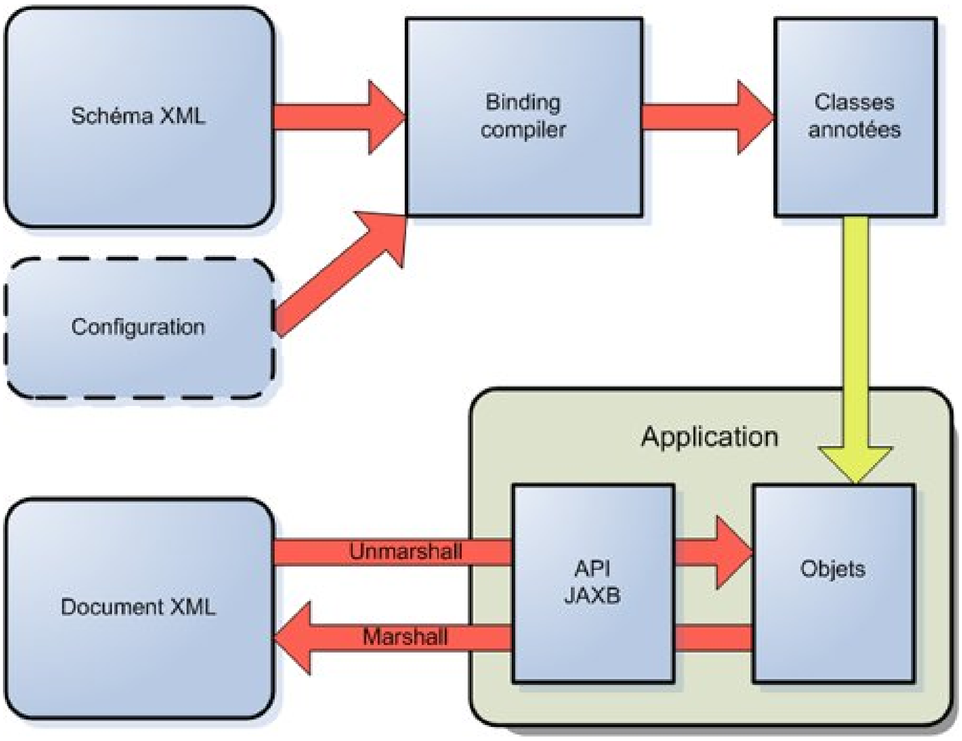
- support uniquement des schémas XML (les DTD ne sont plus supportées)
- mise en oeuvre des annotations
- assure la correspondance bidirectionnelle entre un schéma XML et le bean correspondant.
- l'utilisation de fonctionnalités proposées par Java 5 notamment les generics et les énumérations
- le nombre d'entités générées est moins important : JAXB 2.0 génère une classe pour chaque complexType du schéma alors que JAXB 1.0 génère une interface et une classe qui implémente cette interface. Une méthode de la classe ObjectFactory est générée pour renvoyer une instance de cette classe.

En plus de son utilité principale, JAXB 2.0 propose d'atteindre plusieurs objectifs :

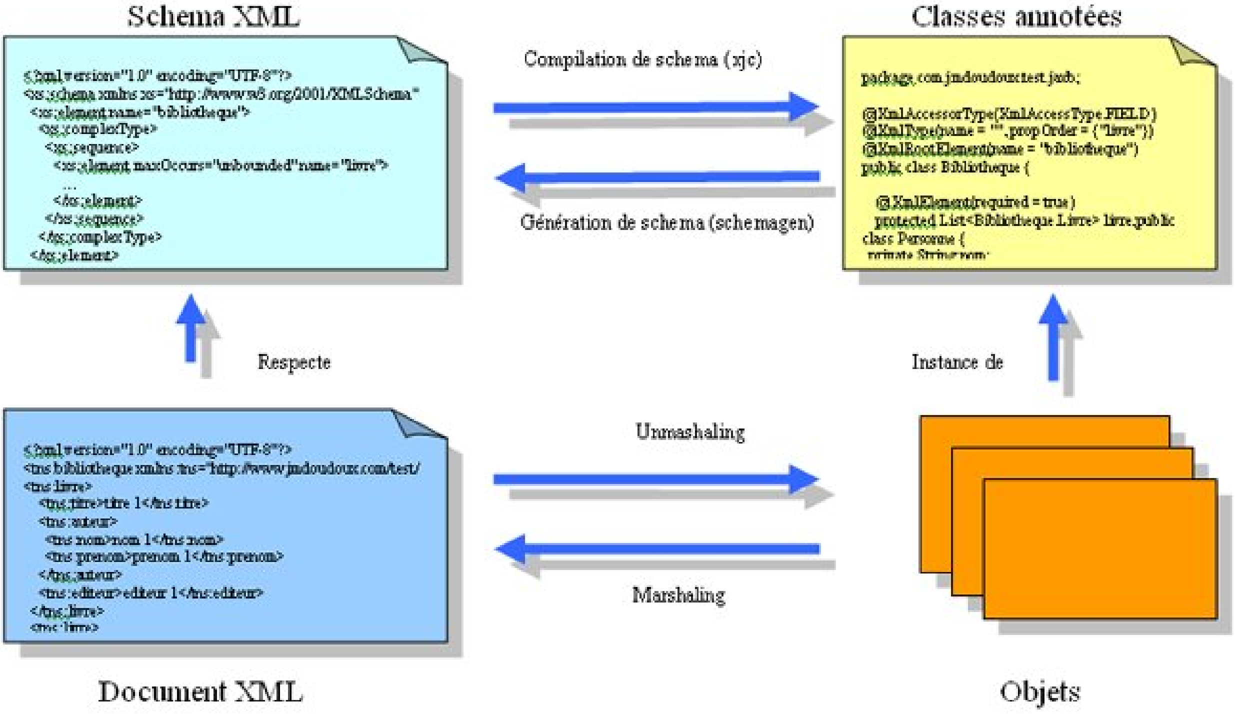
- Être facile à utiliser pour consulter et modifier un document XML sans connaissance ni de XML ni de techniques de traitement de documents XML
- Être configurable : JAXB met en oeuvre des fonctionnalités par défaut qu'il est possible de modifier par configuration pour répondre à ses propres besoins
- S'assurer que la création d'un document XML à partir d'objets et retransformer ce document en objets donne le même ensemble d'objets
- Pouvoir valider un document XML ou les objets qui encapsulent un document sans avoir à écrire le document correspondant
- Être portable : chaque implémentation doit au minimum mettre en oeuvre les spécifications de JAXB

L'utilisation de JAXB implique généralement deux étapes :

- Génération des classes et interfaces à partir du schéma XML
- Utilisation des classes générées et de l'API JAXB pour transformer un document XML en graphe d'objets et vice versa, pour manipuler les données dans le graphe d'objets et pour valider le document



JAXB 2.0 permet toujours de mapper des objets Java dans un document XML et vice versa. Il permet aussi de générer des classes Java à partir un schéma XML et inversement.



```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
 <xs:element name="bibliotheque">
  <xs:complexType>
   <xs:sequence>
    <xs:element maxOccurs="unbounded" name="livre">
     ...
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
```

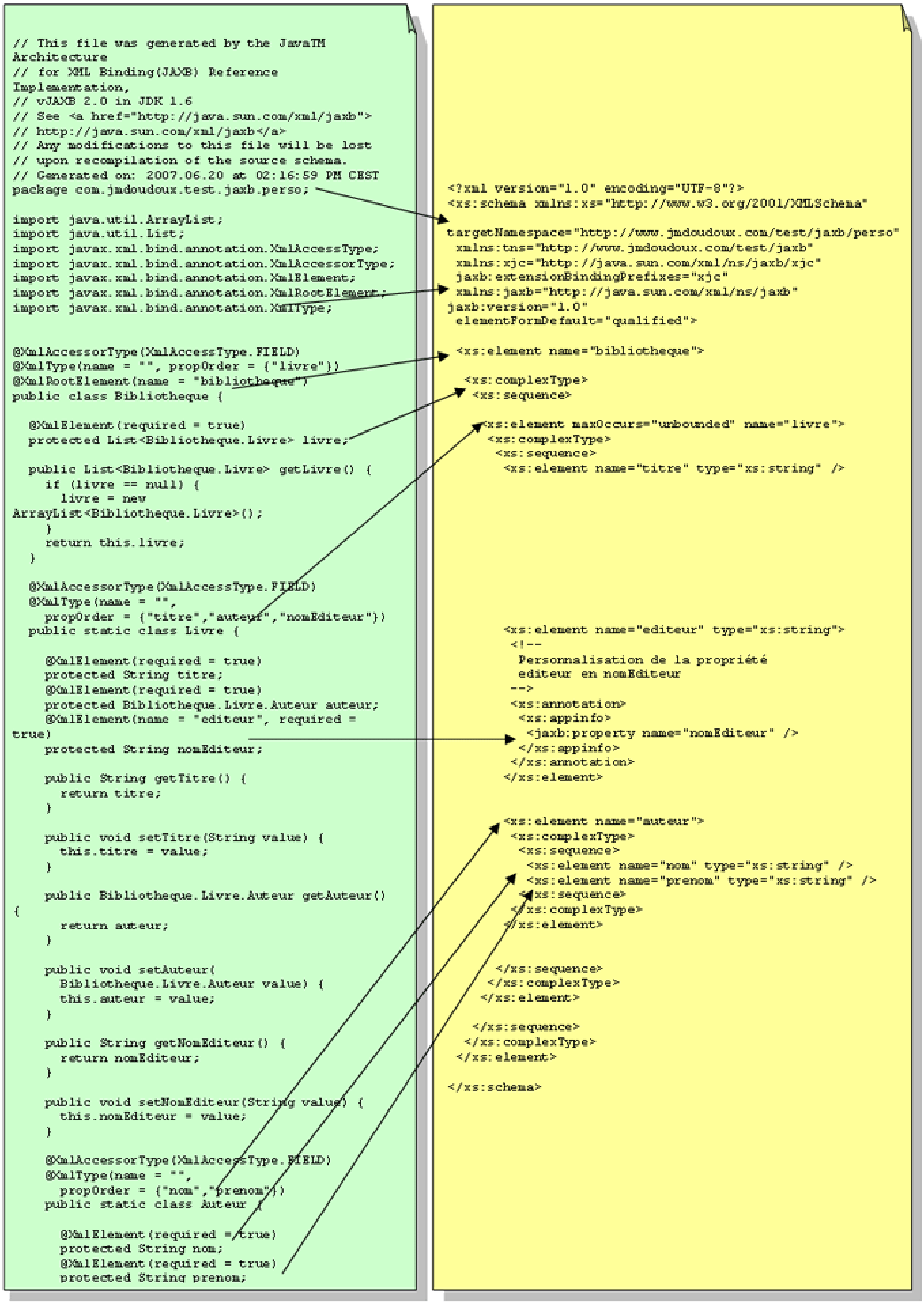
```
package com.jmdoudoux.test.jaxb;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {"livre"})
@XmlRootElement(name = "bibliotheque")
public class Bibliotheque {

  @XmlElement(required = true)
  protected List<Bibliotheque.Livre> livre;public
class Personne {
 private String nom;
```

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:bibliotheque xmlns:tns="http://www.jmdoudoux.com/test/
<tns:livre>
  <tns:titre>titre 1</tns:titre>
  <tns:auteur>
   <tns:nom>nom 1</tns:nom>
   <tns:prenom>prenom 1</tns:prenom>
  </tns:auteur>
  <tns:editeur>editeur 1</tns:editeur>
 </tns:livre>
 <tns:livre>
```

La sérialisation d'un graphe d'objets Java est effectuée par une opération de mashalling. L'opération inverse est dite unmashalling. Lors de ces deux opérations, le document XML peut être validé.

JAXB 2.0 utilise de nombreuses annotations définies dans le package javax.xml.bin.annotation essentiellement pour préciser le mode de fonctionnement lors des opérations de marshalling/unmarshalling.

Ces annotations précisent le mapping entre les classes Java et le document XML. La plupart de ces annotations ont des valeurs par défaut ce qui réduit l'obligation de leur utilisation si la valeur par défaut correspond au besoin.



```
// This file was generated by the JavaTM
Architecture
// for XML Binding(JAXB) Reference
Implementation,
// vJAXB 2.0 in JDK 1.6
// See <a href="http://java.sun.com/xml/jaxb">
// http://java.sun.com/xml/jaxb</a>
// Any modifications to this file will be lost
// upon recompilation of the source schema.
// Generated on: 2007.06.20 at 02:16:59 PM CEST
package com.jmdoudoux.test.jaxb.perso;

import java.util.ArrayList;
import java.util.List;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;


@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {"livre"})
@XmlRootElement(name = "bibliotheque")
public class Bibliotheque {

  @XmlElement(required = true)
  protected List<Bibliotheque.Livre> livre;

  public List<Bibliotheque.Livre> getLivre() {
    if (livre == null) {
      livre = new
ArrayList<Bibliotheque.Livre>();
    }
    return this.livre;
  }

  @XmlAccessorType(XmlAccessType.FIELD)
  @XmlType(name = "",
    propOrder = {"titre","auteur","nomEditeur"})
  public static class Livre {

    @XmlElement(required = true)
    protected String titre;
    @XmlElement(required = true)
    protected Bibliotheque.Livre.Auteur auteur;
    @XmlElement(name = "editeur", required =
true)
    protected String nomEditeur;

    public String getTitre() {
      return titre;
    }

    public void setTitre(String value) {
      this.titre = value;
    }

    public Bibliotheque.Livre.Auteur getAuteur()
{
      return auteur;
    }

    public void setAuteur(
      Bibliotheque.Livre.Auteur value) {
      this.auteur = value;
    }

    public String getNomEditeur() {
      return nomEditeur;
    }

    public void setNomEditeur(String value) {
      this.nomEditeur = value;
    }

    @XmlAccessorType(XmlAccessType.FIELD)
    @XmlType(name = "",
      propOrder = {"nom","prenom"})
    public static class Auteur {

      @XmlElement(required = true)
      protected String nom;
      @XmlElement(required = true)
      protected String prenom;
```

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"

targetNamespace="http://www.jmdoudoux.com/test/jaxb/perso"
 xmlns:tns="http://www.jmdoudoux.com/test/jaxb"
 xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
 jaxb:extensionBindingPrefixes="xjc"
 xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
jaxb:version="1.0"
 elementFormDefault="qualified">

 <xs:element name="bibliotheque">

  <xs:complexType>
   <xs:sequence>

    <xs:element maxOccurs="unbounded" name="livre">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="titre" type="xs:string" />






       <xs:element name="editeur" type="xs:string">
        <!--
         Personnalisation de la propriété
         editeur en nomEditeur
        -->
        <xs:annotation>
         <xs:appinfo>
          <jaxb:property name="nomEditeur" />
         </xs:appinfo>
        </xs:annotation>
       </xs:element>


       <xs:element name="auteur">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="nom" type="xs:string" />
          <xs:element name="prenom" type="xs:string" />
         </xs:sequence>
        </xs:complexType>
       </xs:element>


      </xs:sequence>
     </xs:complexType>
    </xs:element>

   </xs:sequence>
  </xs:complexType>
 </xs:element>

</xs:schema>
```

JAXB 2.0 permet aussi de réaliser dynamiquement à l'exécution une transformation d'un graphe d'objets en document XML et vice versa. C'est cette fonctionnalité qui est largement utilisée dans les services web grâce à l'API JAX-WS 2.0.

La classe abstraite JAXBContext fournie par l'API JAXB permet de gérer la transformation d'objets Java en XML et vice versa.

JAXB 2.0 propose plusieurs outils pour faciliter sa mise en oeuvre :

- un générateur de classes Java (schema compiler) à partir d'un schéma XML. L'outil est nommé xjc dans l'implémentation de référence. Les classes générées mettent en oeuvre les annotations de JAXB.
- un générateur de schéma XML (schema generator) à partir d'un graphe d'objets nommé schemagen dans l'implémentation de référence.

L'API JAXB est contenue dans la package javax.xml.bind

53.2.1. L'obtention de JAXB 2.0

JAXB 2.0 est incorporée dans Java EE 5 et dans Java SE 6.

Avec la version fournie avec JWSDP 2.0, les bibliothèques suivantes doivent être ajoutées au classpath :
jaxb\lib\jaxb-api.jar,
jaxb\lib\jaxb-impl.jar,
jaxb\lib\jaxb-xjc.jar,
jwsdp-shared\lib\activation.jar,
sjsxp\lib\jsr173_api.jar,
sjsxp\lib\sjsxp.jar

Attention JAXB 2.0 requiert un JDK 5.0 minimum pour être utilisé.

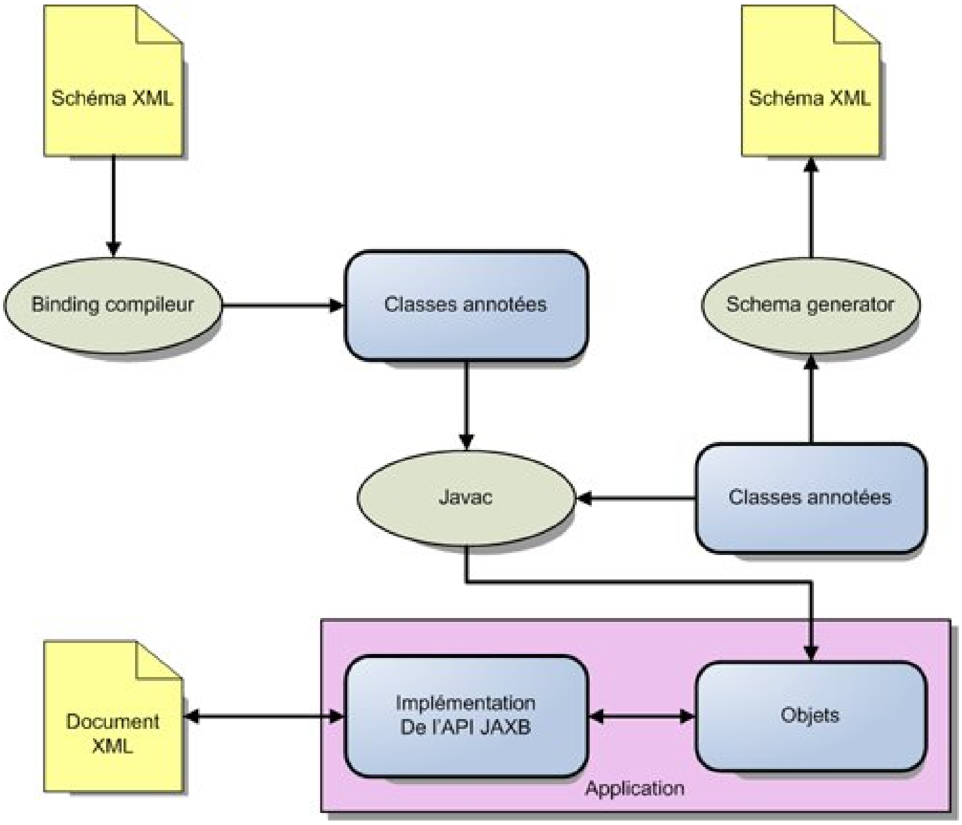
53.2.2. La mise en oeuvre de JAXB 2.0

La mise en oeuvre de JAXB requiert pour un usage standard plusieurs étapes :

- La génération des classes en utilisant l'outil xjc de JAXB à partir d'un schéma du document XML
- Ecriture de code utilisant les classes générées et l'API JAXB pour transformer un document XML en objets Java, modifier des données encapsulées dans le graphe d'objets ou transformer le graphe d'objets en un document XML avec une validation optionnelle du document
- La compilation du code généré et écrit puis l'exécution de l'application

L'utilisation de JAXB se fait donc en deux phases :

1. générer des classes à partir d'un schéma XML et les utiliser dans le code de l'application
2. à l'exécution de l'application, le document XML est transformé en graphe d'objets, les données de ces objets peuvent être modifiées puis le document XML peut être regénéré à partir des objets.



L'utilisation de JAXB met en oeuvre plusieurs éléments :

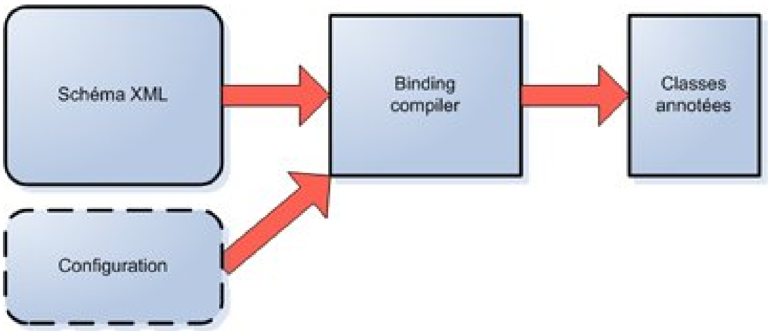
- Une ou plusieurs classes annotées qui vont encapsuler des données du document XML
- (optionnel) un schéma XML : c'est un document XML qui décrit la structure des éléments, attributs et entités d'un document XML. Le but d'un schéma XML est similaire à celui d'une DTD mais le schéma propose une description plus riche et plus fine. Ce schéma peut éventuellement être enrichi de données de configurations concernant les classes à générer
- (optionnel) un outil qui génère les classes annotées à partir d'un schéma avec éventuellement un fichier de configuration pour les classes à générer
- Une API utilisée à l'exécution pour transformer un document XML en un ensemble d'objets du type des classes annotées ou vice versa et permettre des validations
- Un document XML qui sera lu et/ou écrit en fonction des traitements à réaliser

Les avantages d'utiliser JAXB sont nombreux :

- Facilite la manipulation de documents XML dans une application Java
- Manipulation du document XML au travers d'objets : aucune connaissance de XML ou de la manière de traiter un document n'est requise
- Un document XML peut être créé en utilisant les classes générées
- Les traitements de JAXB peuvent être configurés
- Les ressources requises par le graphe d'objets utilisé par JAXB sont moins importantes qu'avec DOM

53.2.3. La génération des classes à partir d'un schéma

Pour permettre l'utilisation et la manipulation d'un document XML, JAXB propose de générer un ensemble de classes à partir du schéma XML du document.



Chaque implémentation de JAXB doit fournir un outil (binding compiler) qui permet la génération de classes et interfaces à partir d'un schema (binding a schema).

53.2.4. La commande xjc

L'implémentation de référence fournit l'outil xjc pour générer les classes à partir d'un schéma XML.

L'utilisation la plus simple de l'outil xjc est de lui fournir le fichier qui contient le schéma XML du document à utiliser.

Exemple :

```
xjc biblio.xsd
```

L'outil xjc possède plusieurs options dont voici les principales :

Option	Rôle
-p nom_package	Précise le package qui va contenir les classes générées
-d répertoire	Précise le répertoire qui va contenir les classes générées
-nv	Inhibe la validation du schéma
-b fichier	Précise un fichier de configuration
-classpath chemin	Précise le classpath

53.2.5. Les classes générées

Le compilateur génère des classes en correspondance avec le schéma XML fourni à l'outil.

Exemple : biblio.xsd

```
<?xml version="1.0"  encoding="UTF-8"?>
<xs:schema  xmlns:xs="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.jmdoudoux.com/test/jaxb"
            xmlns:tns="http://www.jmdoudoux.com/test/jaxb"
            elementFormDefault="qualified">
  <xs:element  name="bibliotheque">
    <xs:complexType>
      <xs:sequence>
        <xs:element  maxOccurs="unbounded" name="livre">
          <xs:complexType>
            <xs:sequence>
              <xs:element  name="titre" type="xs:string" />
              <xs:element  name="auteur">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="nom"  type="xs:string" />
                    <xs:element  name="prenom" type="xs:string" />
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element  name="editeur" type="xs:string" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

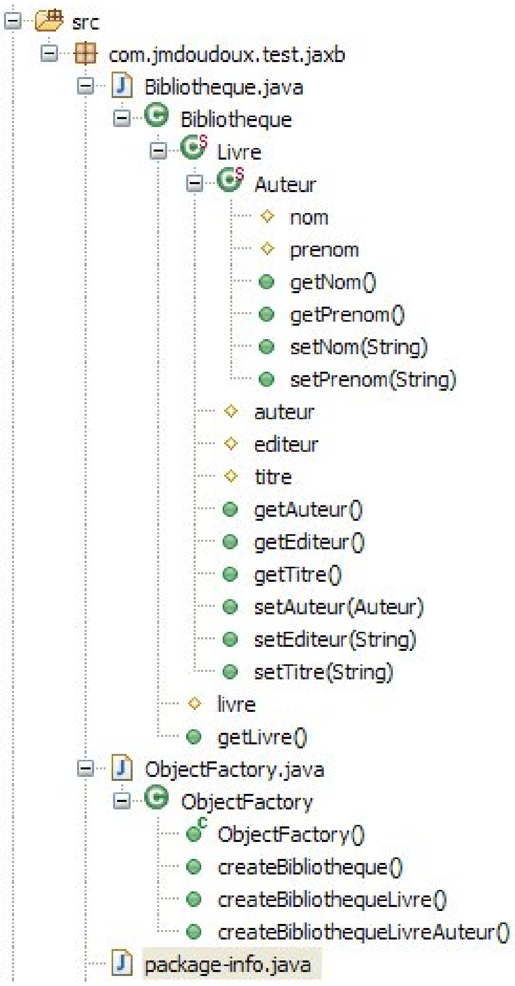
Exemple : exécution de la commande xjc

```
C:\Documents and Settings\jmd\workspace\TestJAXB>xjc -d src biblio.xsd
parsing a schema...
compiling a schema...
com\jmdoudoux\test\jaxb\Bibliotheque.java
com\jmdoudoux\test\jaxb\ObjectFactory.java
com\jmdoudoux\test\jaxb\package-info.java
```

Trois entités sont générées dans le répertoire src :

- fr.jmdoudoux.dej.jaxb.Bibliotheque.java : classes qui encapsulent le document XML
- fr.jmdoudoux.dej.jaxb.package-info.java : permet de conserver les espaces de nommages utilisés dans le package
- fr.jmdoudoux.dej.jaxb.ObjectFactory.java : fabrique qui permet d'instancier des objets utilisés lors du mapping

Par défaut, le package utilisé est déduit de l'espace de nommage défini dans le schéma.



Chaque classe qui encapsule un type complexe du schéma possède des getters et setters sur les éléments du schéma.

La fabrique permet de créer des instances de chacun des types d'objets correspondant à un type complexe du schéma. Cette fabrique est particulièrement utile lors de la création d'un nouveau document XML : le graphe d'objets est créé en ajoutant des instances des objets retournés par cette fabrique.

Les classes générées sont dépendantes de l'implémentation de JAXB utilisée : il est préférable d'utiliser les classes générées par une implémentation avec cette implémentation.

Par défaut, JAXB utilise des règles pour définir chaque entité incluse dans le schéma (element et complexType définis dans le schéma).

53.2.6. L'utilisation de l'API JAXB 2.0

JAXB fournit une API qui permet à l'exécution d'effectuer les opérations de transformation d'un document XML en un graphe d'objets utilisant les classes générées et vice versa (unmashalling/marshalling) ainsi que des opérations de validation.

L'objet principal pour les opérations de transformation est l'objet JAXBContext : il permet d'utiliser l'API JAXB. Pour obtenir une instance de cet objet, il faut utiliser la méthode statique newInstance() en lui passant en paramètre le ou les packages contenant les classes générées à utiliser. Dans le cas où plusieurs packages doivent être précisés, il faut les séparer par une virgule.

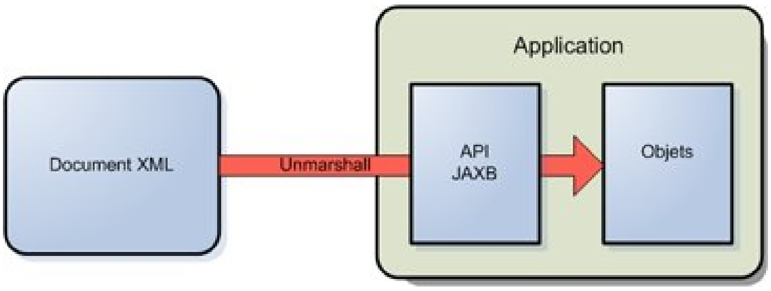
Une autre surcharge de la méthode newInstance() attend en paramètre la classe qui encapsule la racine du document.

Exemple :

```
JAXBContext jc = JAXBContext.newInstance("fr.jmdoudoux.dej.jaxb");
```

53.2.6.1. Le mapping d'un document XML à des objets (unmarshalling)

L'API JAXB propose de transformer un document XML en un ensemble d'objets qui vont encapsuler les données et la hiérarchie du document. Ces objets sont des instances des classes générées à partir du schéma XML.



La création des objets nécessite la création d'un objet de type JAXBContext en utilisant la méthode statique newInstance().

Il faut ensuite instancier un objet de type Unmarshaller qui va permettre de transformer un document XML en un ensemble d'objets. Un telle instance est obtenue en utilisant la méthode createUnmarshaller() de la classe JAXBContext.

Exemple :

```
  Unmarshaller unmarshaller = jc.createUnmarshaller();
```

La méthode unmarshal() de la classe Unmarshaller se charge de traiter un document XML et retourne un objet du type complexe qui encapsule la racine du document XML. Elle possède de nombreuses surcharges à utiliser en fonction des besoins.

Exemple :

```
  Bibliotheque bibliotheque = (Bibliotheque) unmarshaller.unmarshal(new File("biblio.xml"));
```

A partir de cet objet, il est possible d'obtenir et de modifier des données encapsulées dans les différents objets créés à partir des classes générées et du contenu du document. Chacun de ces objets possède un getter et un setter sur leur noeud direct.

Exemple :

```
List<Livre> livres = bibliotheque.getLivre();
      for (int i = 0; i < livres.size(); i++) {
      Livre livre = livres.get(i);
      System.out.println("Livre ");
      System.out.println("Titre : " + livre.getTitre());
      System.out.println("Auteur : " + livre.getAuteur().getNom()
        + " " +livre.getAuteur().getPrenom());
      System.out.println("Editeur : " + livre.getEditeur());
      System.out.println();
```

Il est possible de demander la validation du document avec le schéma en utilisant la méthode setValidating() de la classe Unmarshaller.

Exemple :

```
      unmarshaller.setValidating(true);
```

Exemple : mise en oeuvre des entités générées à partir du schéma biblio.xsd

```
package fr.jmdoudoux.dej.jaxb;

import java.io.File;
import java.util.List;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.Unmarshaller;

import fr.jmdoudoux.dej.jaxb.Bibliotheque.Livre;

public class TestJAXB2 {

  public static void main(String[] args) {
    try {
      JAXBContext jc = JAXBContext.newInstance("fr.jmdoudoux.dej.jaxb");
      Unmarshaller unmarshaller = jc.createUnmarshaller();
      Bibliotheque bibliotheque = (Bibliotheque) unmarshaller.unmarshal(
             new File("biblio.xml"));
      List<Livre> livres = bibliotheque.getLivre();
      for (int i = 0; i < livres.size(); i++) {
      Livre livre = livres.get(i);
      System.out.println("Livre ");
      System.out.println("Titre : " + livre.getTitre());
      System.out.println("Auteur : " + livre.getAuteur().getNom()
        + " " +livre.getAuteur().getPrenom());
      System.out.println("Editeur : " + livre.getEditeur());
      System.out.println();
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Exemple : le document XML utilisé

```
<?xml version="1.0"  encoding="UTF-8"?>
<tns:bibliotheque  xmlns:tns="http://www.jmdoudoux.com/test/jaxb"
                   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                   xsi:schemaLocation="http://www.jmdoudoux.com/test/jaxb biblio.xsd  ">
```

```
   <tns:livre>
     <tns:titre>titre 1</tns:titre>
     <tns:auteur>
       <tns:nom>nom 1</tns:nom>
       <tns:prenom>prenom 1</tns:prenom>
     </tns:auteur>
     <tns:editeur>editeur 1</tns:editeur>
   </tns:livre>
   <tns:livre>
     <tns:titre>titre 2</tns:titre>
     <tns:auteur>
       <tns:nom>nom 2</tns:nom>
       <tns:prenom>prenom 2</tns:prenom>
     </tns:auteur>
     <tns:editeur>editeur 2</tns:editeur>
   </tns:livre>
   <tns:livre>
     <tns:titre>titre 3</tns:titre>
     <tns:auteur>
       <tns:nom>nom 3</tns:nom>
       <tns:prenom>prenom 3</tns:prenom>
     </tns:auteur>
     <tns:editeur>editeur 3</tns:editeur>
   </tns:livre>
</tns:bibliotheque>
```

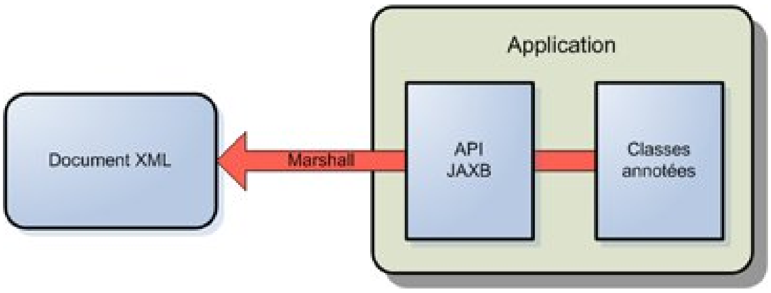
Résultat :

```
Livre
Titre : titre 1
Auteur : nom 1 prenom 1
Editeur : editeur 1

Livre
Titre : titre 2
Auteur : nom 2 prenom 2
Editeur : editeur 2

Livre
Titre : titre 3
Auteur : nom 3 prenom 3
Editeur : editeur 3
```

53.2.6.2. La création d'un document XML à partir d'objets

JAXB permet de créer un document XML à partir d'un graphe d'objets : cette opération est nommée marshalling. Une opération de mashalling est l'opération inverse de l'opération d'unmarshalling.

Ce graphe d'objets peut être issu d'une opération de type unmarshalling (construction à partir d'un document XML existant) ou issu d'une création de toutes pièces de l'ensemble des objets. Dans le premier cas cela correspond à une modification du document et dans le second cas à une création de document.



La création des objets nécessite la création d'un objet de type JAXBContext en utilisant la méthode statique newInstance().

Il faut ensuite instancier un objet de type Marshaller qui va permettre de transformerun ensemble d'objets en un document XML. Une telle instance est obtenue en utilisant la méthode createMarshaller() de la classe JAXBContext.

Exemple :

```
      Marshaller marshaller = jc.createMarshaller();
```

La méthode marshal() de la classe Marshaller se charge de créer un document XML à partir d'un graphe d'objets dont l'objet racine lui est fourni en paramètre.

La méthode marshal() possède plusieurs surcharges qui permettent de préciser la forme du document XML généré :

- un fichier (File),
- un flux (outputStream),
- un flux de caractères (Writer),
- un document DOM (Document),
- un gestionnaire d'événements SAX (ContentHandler),
- un objet de type javax.xml.transform.SAXResult,
- un objet de type javax.xml.transform.DOMResult,
- un objet de type javax.xml.transform.StreamResult,
- un objet de type javax.xml.stream.XMLStreamWriter,
- ou un objet de type javax.xml.stream.XMLEventWriter.

L'objet Mashaller possède des propriétés qu'il est possible de valoriser en utilisant la méthode setProperty(). Les spécifications de JAXB proposent des propriétés qui doivent être obligatoirement supportées par l'implémentation. Chaque implémentation est libre de proposer des propriétés supplémentaires.

Exemple : demander le formattage du document créé

```
      Marshaller m = context.createMarshaller();
      m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
```

Il est possible de demander la validation du graphe d'objets. La validation n'est pas intégrée à l'opération de mashalling mais elle est effectuée à la demande.

La validation s'effectue en utilisant la classe Validator. Une instance de cette classe est obtenue en utilisation la méthode createValidator() de la classe JAXBContext.

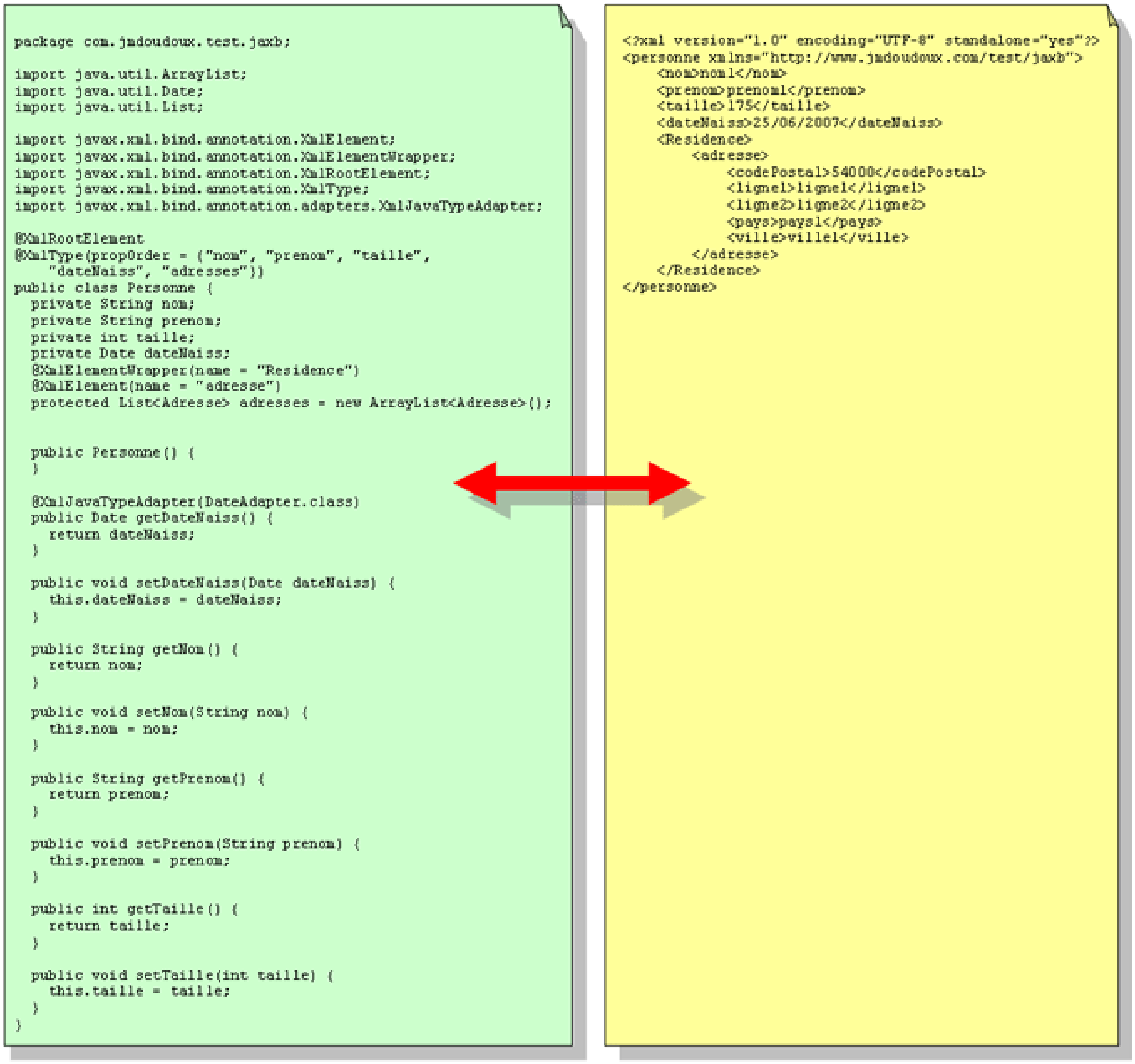
Exemple :

```
Validator validator = jaxbContext.createValidator();
```

Pour valider le graphe d'objets vis-à-vis du schéma, il faut utiliser la méthode validate() de la classe Validator en lui passant en paramètre l'objet qui encapsule la racine du document.

53.2.6.3. La création d'un document en utilisant des classes annotées

JAXB permet de mapper une ou plusieurs classes annotées vers un document XML sans être obligé d'utiliser un schéma XML. Dans ce cas, le développeur a la charge d'écrire la ou les classes annotées requises.



```
package com.jmdoudoux.test.jaxb;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlElementWrapper;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;
import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter;

@XmlRootElement
@XmlType(propOrder = {"nom", "prenom", "taille",
    "dateNaiss", "adresses"})
public class Personne {
  private String nom;
  private String prenom;
  private int taille;
  private Date dateNaiss;
  @XmlElementWrapper(name = "Residence")
  @XmlElement(name = "adresse")
  protected List<Adresse> adresses = new ArrayList<Adresse>();


  public Personne() {
  }

  @XmlJavaTypeAdapter(DateAdapter.class)
  public Date getDateNaiss() {
    return dateNaiss;
  }

  public void setDateNaiss(Date dateNaiss) {
    this.dateNaiss = dateNaiss;
  }

  public String getNom() {
    return nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

  public String getPrenom() {
    return prenom;
  }

  public void setPrenom(String prenom) {
    this.prenom = prenom;
  }

  public int getTaille() {
    return taille;
  }

  public void setTaille(int taille) {
    this.taille = taille;
  }
}
```

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<personne xmlns="http://www.jmdoudoux.com/test/jaxb">
    <nom>nom1</nom>
    <prenom>prenom1</prenom>
    <taille>175</taille>
    <dateNaiss>25/06/2007</dateNaiss>
    <Residence>
        <adresse>
            <codePostal>54000</codePostal>
            <ligne1>ligne1</ligne1>
            <ligne2>ligne2</ligne2>
            <pays>pays1</pays>
            <ville>ville1</ville>
        </adresse>
    </Residence>
</personne>
```

Il n'est donc pas nécessaire d'utiliser des classes générées par le compilateur de schéma mais dans ce cas, la ou les classes doivent être créées manuellement en utilisant les annotations adéquates.

La classe qui encapsule la racine du document doit être annotée avec l'annotation @XmlRootElement. Une exception de type javax.xml.bind.MarshalException est levée par JAXB si la classe racine ne possède pas cette annotation.

Exemple :

```
javax.xml.bind.MarshalException
 - with linked exception:
[com.sun.istack.internal.SAXException2: unable to marshal type "fr.jmdoudoux.dej.jaxb.
Personne" as an element because it is missing an @XmlRootElement annotation]
```

JAXB utilise des comportements par défaut qui ne nécessitent de la part du développeur que de définir des comportements particuliers si les valeurs par défaut ne conviennent pas.

Exemple : un bean utilisé avec JAXB

```
package fr.jmdoudoux.dej.jaxb;

import java.util.Date;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
```

```
public class Personne {
  private String nom;
  private String prenom;
  private int taille;
  private Date dateNaiss;

  public Personne() {
  }

  public Personne(String nom, String prenom, int taille, Date dateNaiss) {
    super();
    this.nom = nom;
    this.prenom = prenom;
    this.taille = taille;
    this.dateNaiss = dateNaiss;
  }

  public Date getDateNaiss() {
    return dateNaiss;
  }

  public void setDateNaiss(Date dateNaiss) {
    this.dateNaiss = dateNaiss;
  }

  public String getNom() {
    return nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

  public String getPrenom() {
    return prenom;
  }

  public void setPrenom(String prenom) {
    this.prenom = prenom;
  }

  public int getTaille() {
    return taille;
  }

  public void setTaille(int taille) {
    this.taille = taille;
  }
}
```

La création du document nécessite la création d'un objet de type JAXBContext en utilisant la méthode statique newInstance().

Il faut ensuite créer un objet de type Marshaller à partir du contexte et appeler sa méthode marshal() pour générer le document.

Exemple : marshalling de l'objet

```
package fr.jmdoudoux.dej.jaxb;

import java.util.Date;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;

public class TestJAXB1 {

  public static void main(String[] args) {
    try {
      JAXBContext context = JAXBContext.newInstance(Personne.class);
      Marshaller m = context.createMarshaller();
      m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
      Personne p = new Personne("nom1", "prenom1", 175, new Date());
      m.marshal(p, System.out);
    } catch (JAXBException ex) {
      ex.printStackTrace();
    }
  }
}
```

Résultat :

```
<?xml version="1.0"  encoding="UTF-8" standalone="yes"?>
<personne>
  <dateNaiss>2007-01-16T17:03:31.213+01:00</dateNaiss>
  <nom>nom1</nom>
  <prenom>prenom1</prenom>
  <taille>175</taille>
</personne>
```

53.2.6.4. La création d'un document en utilisant les classes générées à partir d'un schéma

Une des classes générées à partir du schéma se nomme ObjectFactory : c'est une fabrique d'objets pour les classes générées qui encapsulent des données d'un document respectant le schéma.

Pour créer un document XML en utilisant ces classes, il faut mettre en oeuvre plusieurs étapes.

La création du document nécessite la création d'un objet de type JAXBContext en utilisant la méthode statique newInstance().

Il faut ensuite créer le graphe d'objets en utilisant la classe ObjectFactory pour instancier les différents objets et valoriser les données de ces objets en utilisant les setters.

Il faut créer un objet de type Marshaller à partir du contexte et appeler sa méthode marshal() pour générer le document.

Exemple :

```
package fr.jmdoudoux.dej.jaxb;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;

public class TestJAXB3 {
  public static void main(String[] args) {
    try {
      JAXBContext context = JAXBContext.newInstance(Bibliotheque.class);
      Marshaller m = context.createMarshaller();
      m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);

      ObjectFactory fabrique = new ObjectFactory();
      Bibliotheque bibliotheque = fabrique.createBibliotheque();

      Bibliotheque.Livre livre = fabrique.createBibliothequeLivre();
      livre.setEditeur("editeur 1");
      livre.setTitre("titre 1");

      Bibliotheque.Livre.Auteur auteur = fabrique
          .createBibliothequeLivreAuteur();
      auteur.setNom("nom 1");
      auteur.setPrenom("prenom 1");
      livre.setAuteur(auteur);

      bibliotheque.getLivre().add(livre);

      m.marshal(bibliotheque, System.out);
    } catch (JAXBException ex) {
      ex.printStackTrace();
    }
  }
}
```

53.2.7. La configuration de la liaison XML / Objets

JAXB propose des fonctionnalités pour configurer finement les traitements qu'il propose.

53.2.7.1. L'annotation du schéma XML

JAXB utilise des traitements par défaut qu'il est possible de configurer différemment en utilisant soit les annotations soit un fichier de configuration qui sera fourni au compilateur.

Les classes générées peuvent aussi être configurées, notamment le nom du package et des classes utilisées.

Par défaut, le générateur de classes à partir du schéma utilise des conventions de nommage des différentes entités générées à partir des noms utilisés dans le schéma. Il est possible de configurer de façon différente les noms utilisés.

Les spécifications JAXB décrivent de façon précise comment les éléments d'un schéma sont transformés en classes Java. Il est possible de préciser des informations particulières dans le schéma pour modifier ce comportement par défaut.

Ces informations peuvent être incluses directement dans le schéma ou fournies dans un fichier dédié. Dans le schéma, ces informations sont fournies dans un tag <annotation> qui contient un tag fils <appinfo>.

Exemple : biblio2.xsd

```
<?xml  version="1.0" encoding="UTF-8"?>
<xs:schema  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.jmdoudoux.com/test/jaxb/perso"
   xmlns:tns="http://www.jmdoudoux.com/test/jaxb"
   xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
  jaxb:extensionBindingPrefixes="xjc"
   xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"  jaxb:version="1.0"
  elementFormDefault="qualified">
  <xs:element  name="bibliotheque">
    <xs:complexType>
      <xs:sequence>
        <xs:element  maxOccurs="unbounded" name="livre">
          <xs:complexType>
            <xs:sequence>
              <xs:element  name="titre" type="xs:string" />
               <xs:element  name="auteur">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element  name="nom"
                       type="xs:string" />
                    <xs:element  name="prenom"
                      type="xs:string" />
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element  name="editeur" type="xs:string">
                <!--
                  Personnalisation de la  propriété editeur en nomEditeur
                -->
                <xs:annotation>
                  <xs:appinfo>
                    <jaxb:property  name="nomEditeur" />
                  </xs:appinfo>
                </xs:annotation>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Exemple : génération des classes à partir du schéma annoté

```
C:\Documents and Settings\jmd\workspace\TestJAXB>xjc -d src -extension biblio2.x
sd
parsing a schema...
compiling a schema...
com\jmdoudoux\test\jaxb\perso\Bibliotheque.java
com\jmdoudoux\test\jaxb\perso\ObjectFactory.java
com\jmdoudoux\test\jaxb\perso\package-info.java
```

L'option -extension du générateur de classes autorise l'outil à utiliser des extensions proposées par l'implémentation de JAXB. Sans cette option, l'outil utilise le mode strict et une exception est levée si une extension est utilisée.

Exemple : la classe Livre générée à partir du schéma

```
...
    @XmlAccessorType(XmlAccessType.FIELD)
    @XmlType(name = "", propOrder = {
        "titre",
        "auteur",
        "nomEditeur"
    })
    public static class Livre {

        @XmlElement(required = true)
        protected String titre;
        @XmlElement(required = true)
        protected Bibliotheque.Livre.Auteur auteur;
        @XmlElement(name = "editeur", required = true)
        protected String nomEditeur;

...

        /**
         * Gets the value of the nomEditeur property.
         *
         * @return
         *     possible object is
         *     {@link String }
         *
         */
        public String getNomEditeur() {
            return nomEditeur;
        }

        /**
         * Sets the value of the nomEditeur property.
         *
         * @param value
         *     allowed object is
         *     {@link String }
         *
         */
        public void setNomEditeur(String value) {
            this.nomEditeur = value;
        }
...
```

JAXB propose de nombreuses fonctionnalités de configuration : consultez les spécifications pour de plus amples détails.

53.2.7.2. Les annotations de JAXB

La configuration de la transformation d'un document XML en objets Java est réalisée grâce à l'utilisation d'annotations dédiées dans les classes Java.

De nombreuses annotations sont définies par JAXB 2.0 dont voici les principales :

XmlAccessorOrder	Ordonner les champs et propriétés dans la classe
XmlAccessorType	Préciser comment un champ ou une propriété est accédé par JAXB. Par défaut, tous les champs public ou annotés sont pris en compte sauf ceux marqués avec @XmlTransient. Sa valeur est de type XmlAccessType qui est une énumération qui peut prendre les valeurs NONE, FIELD, PROPERTY, PUBLIC_MEMBER
XmlAnyAttribute	
XmlAnyElement	Convertir une collections d'éléments en une collection de type List<Element>
XmlAttachmentRef	
XmlAttribute	Convertir une propriété en un attribut dans le document XML
XmlElement	Convertir une propriété en un élément dans le document XML
XmlElementDecl	Associer une fabrique à un élément XML
XmlElementRef	
XmlElementRefs	
XmlElements	Contenir plusieurs annotations @XmlElement
XmlElementWrapper	Créer un élément père dans le document XML pour des collections d'éléments
XmlEnum	Définir la façon dont une énumération est convertie dans le document XML
XmlEnumValue	Définir la valeur numérique d'un élément d'une énumération
XmlID	Convertir une propriété en un XML ID dans le document XML. Ne peut être utilisé que sur une seule propriété
XmlIDREF	Convertir une propriété en un XML IDREF dans le document XML
XmlInlineBinaryData	
XmlList	Préciser que les éléments de liste sont représentés sous la forme d'une chaîne de caractères dans laquelle chaque valeur est séparée par un espace
XmlMimeType	
XmlMixed	
XmlNs	Associer un préfixe d'un espace de nommage à un URI
XmlRegistry	Marquer une classe comme possédant une ou des méthodes annotées avec @XmlElementDecl
XmlRootElement	Associer une classe ou une énumération à un élément racine XML
XmlSchema	Associer un ou plusieurs espaces de nommage à un package
XmlSchemaType	Associer un type Java ou une énumération à un type défini dans un schéma
XmlSchemaTypes	
XmlTransient	Marquer une entité ne devant pas être mappée dans le document XML
XmlType	Associer une classe ou une énumération à un type d'un schéma XML. L'attribut propOrder permet de définir l'ordre des champs dans le document XML
XmlValue	Convertir une classe en une valeur simple

Ces annotations sont définies dans le package javax.xml.bind.annotation.

L'annotation @XmlRootElement peut être utilisée sur une classe pour préciser que cette classe sera le tag racine du document XML. Chaque attribut de la classe sera un tag fils dans le document XML. L'attribut namespace de l'annotation @XmlRootElement permet de préciser l'espace de nommage.

L'annotation XmlTransient permet d'ignorer une entité dans le mapping.

Exemple :

```
  @XmlTransient
  public String getNom() {
    return nom;
  }
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<personne xmlns="http://www.jmdoudoux.com/test/jaxb">
    <dateNaiss>2007-06-21T15:18:28.809+02:00</dateNaiss>
    <prenom>prenom1</prenom>
    <taille>175</taille>
</personne>
```

L'annotation XmlAttribute permet de mapper une propriété sous la forme d'un attribut et fournir des précisions sur la configuration de cet attribut.

Exemple :

```
@XmlAttribute (name="nom")
public String getNom() {
  return nom;
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<personne  xmlns="http://www.jmdoudoux.com/test/jaxb" nom="nom1">
  <dateNaiss>2007-06-21T15:20:34.377+02:00</dateNaiss>
  <prenom>prenom1</prenom>
  <taille>175</taille>
</personne>
```

Pour les collections, il est possible d'utiliser l'annotation @XmlElementWrapper pour définir un élément père qui encapsule les occurrences de la collection et d'utiliser l'annotation XmlElement pour préciser le nom de chaque élément de la collection.

Exemple :

```
  @XmlElementWrapper(name = "adresses")
  @XmlElement(name = "adresse")
  protected List<Adresse> adresses = new ArrayList<Adresse>();
```

Résultat :

```
<?xml version="1.0"  encoding="UTF-8" standalone="yes"?>
<personne  xmlns="http://www.jmdoudoux.com/test/jaxb">
  <adresses>
    <adresse>
      <codePostal>54000</codePostal>
      <ligne1>ligne1</ligne1>
      <ligne2>ligne2</ligne2>
      <pays>pays1</pays>
      <ville>ville1</ville>
    </adresse>
  </adresses>
  <dateNaiss>2007-06-21T15:34:35.547+02:00</dateNaiss>
  <nom>nom1</nom>
  <prenom>prenom1</prenom>
  <taille>175</taille>
</personne>
```

Il est possible de configurer l'ordre des éléments.

Exemple :

```
@XmlRootElement
@XmlType(propOrder = {"nom", "prenom", "taille", "dateNaiss", "adresses"})
public class Personne {
  private String nom;
  private String prenom;
  private int    taille;
  private Date   dateNaiss;

  @XmlElementWrapper(name = "adresses")
  @XmlElement(name = "adresse")
  protected List<Adresse> adresses = new ArrayList<Adresse>();

  // getters et setters ...
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<personne  xmlns="http://www.jmdoudoux.com/test/jaxb">
  <nom>nom1</nom>
  <prenom>prenom1</prenom>
  <taille>175</taille>
  <dateNaiss>2007-06-21T15:44:12.815+02:00</dateNaiss>
  <adresses>
    <adresse>
      <codePostal>54000</codePostal>
      <ligne1>ligne1</ligne1>
      <ligne2>ligne2</ligne2>
      <pays>pays1</pays>
      <ville>ville1</ville>
    </adresse>
  </adresses>
</personne>
```

Il est possible de définir des classes de type Adapter qui permettent de personnaliser la façon dont un objet est sérialisé/désérialisé dans le document XML.

Ces adapters héritent de la classe javax.xml.bind.annotation.adapters.XmlAdapter. Il suffit de redéfinir les méthodes marshal() et unmashal(). Ces méthodes seront utilisées à l'exécution par l'API JAXB lors des transformations.

Exemple : la classe DateAdapter

```
package fr.jmdoudoux.dej.jaxb;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;

import javax.xml.bind.annotation.adapters.XmlAdapter;

public class DateAdapter extends XmlAdapter<String, Date> {

  DateFormat df = new SimpleDateFormat("dd/MM/yyyy");

  public Date unmarshal(String date) throws Exception {
    return df.parse(date);
  }

  public String marshal(Date date) throws Exception {
    return df.format(date);
  }
}
```

L'annotation @XmlJavaTypeAdapter permet de préciser la classe de type Adapter qui sera à utiliser à la place de la conversion par défaut.

Exemple :

```
package fr.jmdoudoux.dej.jaxb;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlElementWrapper;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;
import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter;

@XmlRootElement
@XmlType(propOrder = {"nom", "prenom", "taille", "dateNaiss", "adresses"})
public class Personne {
  private String nom;
  private String prenom;
  private int    taille;
  private Date   dateNaiss;

  @XmlElementWrapper(name = "adresses")
  @XmlElement(name = "adresse")
  protected List<Adresse> adresses = new ArrayList<Adresse>();

  public Personne() {
  }

  public Personne(String nom, String prenom, int taille, Date dateNaiss) {
    super();
    this.nom = nom;
    this.prenom = prenom;
    this.taille = taille;
    this.dateNaiss = dateNaiss;
  }

  @XmlJavaTypeAdapter(DateAdapter.class)
  public Date getDateNaiss() {
    return dateNaiss;
  }

  public void setDateNaiss(Date dateNaiss) {
    this.dateNaiss = dateNaiss;
  }

  // getters et setters ...
}
```

Résultat :

```
<?xml version="1.0"  encoding="UTF-8" standalone="yes"?>
<personne  xmlns="http://www.jmdoudoux.com/test/jaxb">
  <nom>nom1</nom>
  <prenom>prenom1</prenom>
  <taille>175</taille>
  <dateNaiss>22/06/2007</dateNaiss>
  <adresses>
    <adresse>
      <codePostal>54000</codePostal>
      <ligne1>ligne1</ligne1>
      <ligne2>ligne2</ligne2>
      <pays>pays1</pays>
      <ville>ville1</ville>
    </adresse>
  </adresses>
</personne>
```

Ceci ne présente qu'une toute petite partie des fonctionnalités proposées par JAXB. Pour de plus amples informations, consultez les spécifications de JAXB.

53.2.7.3. La génération d'un schéma à partir de classes compilées

L'outil schemagen fourni avec l'implémentation par défaut permet de générer un schéma XML à partir de classes annotées compilées.

Exemple :

```
C:\Documents and Settings\jmd\workspace\TestJAXB>schemagen -cp ./bin fr.jmdoudo
ux.test.jaxb.Personne
Note: Writing schema1.xsd
```

Résultat :

```
<?xml  version="1.0" encoding="UTF-8"  standalone="yes"?>
<xs:schema  version="1.0"  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="personne" type="personne"/>
  <xs:complexType  name="personne">
      <xs:sequence>
        <xs:element name="nom"  type="xs:string" minOccurs="0"/>
        <xs:element name="prenom"  type="xs:string" minOccurs="0"/>
        <xs:element name="taille"  type="xs:int"/>
        <xs:element name="dateNaiss"  type="xs:string" minOccurs="0"/>
        <xs:element name="adresses"  minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="adresse" type="adresse" maxOccurs="unbounded" minOccurs="0"/>
            </xs:sequence>
           </xs:complexType>
         </xs:element>
       </xs:sequence>
     </xs:complexType>
     <xs:complexType name="adresse">
       <xs:sequence>
         <xs:element name="codePostal" type="xs:string"  minOccurs="0"/>
         <xs:element name="ligne1" type="xs:string"  minOccurs="0"/>
         <xs:element name="ligne2" type="xs:string"  minOccurs="0"/>
         <xs:element name="pays" type="xs:string"  minOccurs="0"/>
         <xs:element name="ville" type="xs:string"  minOccurs="0"/>
       </xs:sequence>
     </xs:complexType>
</xs:schema>
```

54. StAX (Streaming Api for XML)

Chapitre 54

Niveau : 🟩🟩🟨⬜⬜⬜ Intermédiaire

StAX est l'acronyme de Streaming Api for XML : c'est une API qui permet de traiter un document XML de façon simple en consommant peu de mémoire tout en permettant de garder le contrôle sur les opérations d'analyse ou d'écriture.

StAX a été développée sous la JSR-173 et est incorporée dans Java SE 6.0.

StAX propose des fonctionnalités pour parcourir et écrire un document XML mais ne permet pas de manipuler le contenu d'un document.

Le but de StAX n'est pas de remplacer SAX ou DOM mais de proposer une nouvelle façon d'analyser un document XML : StAX vient en complément des API DOM et SAX.

Sa mise en oeuvre par rapport aux deux API existantes peut être dans certains cas plus simple et donc plus facile que SAX et plus efficace et performante que DOM. StAX permet de traiter un document XML de manière rapide, facile et consommant peu de ressources : le modèle d'événements utilisé est plus simple que celui de SAX et les ressources requises sont moins importantes que pour un traitement grâce à DOM.

Ce chapitre contient plusieurs sections :

- ♦ La présentation de StAX
- ♦ Les deux API de StAX
- ♦ Les fabriques
- ♦ Le traitement d'un document XML avec l'API du type curseur
- ♦ Le traitement d'un document XML avec l'API du type itérateur
- ♦ La mise en oeuvre des filtres
- ♦ L'écriture un document XML avec l'API de type curseur
- ♦ L'écriture un document XML avec l'API de type itérateur
- ♦ La comparaison entre SAX, DOM et StAX

54.1. La présentation de StAX

Avant StAX, l'analyse d'un document XML pouvait se faire principalement par deux API standard (DOM et SAX) ou une API non standard (JDOM).

L'analyse d'un document XML peut se faire grâce à deux grandes catégories de parseurs :

- Parseur basé sur un arbre : DOM implémente cette technique qui consiste à représenter et stocker le document dans un arbre d'objets. Il est ainsi possible, une fois cet arbre créé, de parcourir librement les noeuds de l'arbre et de le modifier. Cette représentation est généralement plus gourmande en ressource que le document lui-même ce qui la rend particulièrement inadaptée à des documents de grandes tailles.
- Parseur basé sur un flux (document streaming) : des événements sont émis lors de la lecture séquentielle du flux pour notifier chaque changement lors de l'analyse du document. SAX implémente cette technique qui nécessite moins de ressources mais offre cependant moins de souplesse dans la manipulation du document.

Il existe deux sortes de traitement par flux :

- Push : le parser génère des événements à chaque noeud du document que le client en ait besoin ou non
- Pull : le client demande explicitement au parser de lui donner l'événement suivant ce qui permet au client de conserver la main sur les traitements du parser et ainsi de piloter l'analyse

	Push parsing	Pull parsing
Implémentation	SAX	StAX
Contrôle des traitements	Par le parseur	Par le client
Complexité de mise en oeuvre	Moyenne	Faible
Parcours de tout le document	Oui	Non (interruption possible par le client)

Avec le traitement par flux, seul l'élément courant durant le parcours séquentiel du document est accessible. Ceci limite les traitements possibles sur le document et impose généralement de conserver un contexte.

L'utilisation d'un traitement par flux est particulièrement utile lors de la manipulation de gros documents, de l'utilisation dans un environnement possédant des ressources limitées (exemple utilisation avec Java ME) ou lors de traitements en parallèle de documents (exemple dans un serveur d'applications ou un moteur de services web).

StAX propose un modèle de traitement du document qui repose sur une lecture séquentielle du document sous le contrôle de l'application (ce n'est pas le parseur qui pilote le parcours mais l'application qui pilote le parseur). StAX représente un document sous la forme d'un ensemble d'événements qui sont fournis à la demande de l'application dans l'ordre du parcours séquentiel du document.

StAX repose sur le modèle de conception Iterator : chaque élément du document est parcouru séquentiellement à la demande du code pour émettre un événement. Ce parcours se fait à l'aide d'un curseur virtuel.

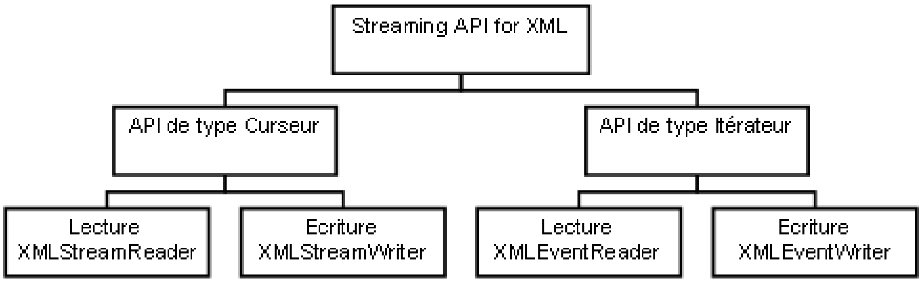
54.2. Les deux API de StAX

StAX est donc une API qui propose de mettre en oeuvre une troisième méthode pour traiter un document XML : le pull parsing. Son but est de fournir un parser qui puisse traiter de gros documents XML, avec une faible quantité de ressources requises et dont les traitements sont contrôlés par le code.

StAX propose une API pour un traitement d'un document XML sous la forme d'une itération sur des événements émis par le parser à la demande du client. Elle propose deux formes d'API :

- une API du type curseur (Cursor) : permet le parcours du document en générant un événement codé sur un entier pour chaque type d'élément rencontré
- une API de type itérateur (Event Iterator) : permet le parcours du document en produisant un événement de type XMLEvent à chaque élément rencontré

Elles permettent toutes les deux la lecture et l'écriture d'un document XML.



La définition de deux API permet de les conserver séparément avec une faible complexité plutôt que d'avoir une seule API plus complexe.

L'API de type curseur parcourt le flux du document et émet des événements sous la forme d'un entier codant chaque événement.

L'API de type curseur est plus efficace dans la mesure où elle n'a pas besoin d'instancier un objet pour chaque événement comme le fait l'API de type itérateur. L'API de type itérateur est plus facile à utiliser puisque toutes les données utiles sont déjà présentes dans l'objet de type XMLEvent.

L'interface XMLStreamReader définit un contrat pour un objet qui va analyser un document XML avec une API de type curseur.

L'interface XMLStreamReader propose des méthodes pour obtenir des informations sur l'élément courant représenté par l'événement courant du curseur. Ces méthodes retournent des chaînes de caractères ce qui limite les ressources à une transformation en chaîne de caractères quels que soient les contenus des éléments retournés.

L'interface XMLStreamWriter définit un contrat pour un objet qui va générer un document XML.

L'API de type itérateur parcourt le flux du document et émet des événements sous la forme d'objets de type XMLEvent qui encapsulent les informations de l'événement.

L'interface XMLEventReader définit un contrat pour un objet qui va analyser un document XML avec une API de type itérateur sur des événements. Elle hérite de l'interface Iterator : elle propose donc la méthode nextEvent() qui retourne le prochain événement et la méthode hasNext() qui permet de savoir s'il y a encore un événement à traiter.

L'interface XMLEvent encapsule les données d'un événement lié au parcours du document XML : ces événements sont émis à la demande du client dans l'ordre de leur apparition lors du parcours du document.

La définition de deux API permet de laisser au développeur le choix d'utiliser l'API de type curseur pour limiter l'instanciation d'objets durant l'analyse du document ou d'utiliser l'API de type itérateur pour bénéficier directement des événements sous la forme d'objets. Le développeur peut ainsi choisir en fonction de son contexte de mettre en oeuvre l'une ou l'autre des API selon des critères de consommation de ressources ou de simplicité et de fiabilité du code.

L'API de type curseur est moins verbeuse et moins puissante que celle de type itérateur d'événements. Elle est cependant plus efficace car elle instancie moins d'objets. Le code à produire avec l'API de type curseur est plus petit et généralement plus efficace. L'API de type itérateur est plus flexible et évolutive que l'API de type curseur.

Elles permettent toutes les deux uniquement la lecture vers l'avant du document mais l'API de type itérateur propose en plus la méthode peek() qui permet de connaître le prochain événement.

StAX permet aussi de construire un document XML en utilisant les flux. Les API de type curseur et itérateur proposent leur propre interface pour permettre l'écriture de documents.

Les API de StAX sont contenues dans les packages javax.xml.stream et javax.xml.transform.stream

Comme SAX, StAX est un parseur dont les spécifications sont écrites pour Java. Il existe une implémentation de référence mais l'implémentation de ses spécifications peut être réalisée par un tiers.

54.3. Les fabriques

StAX propose des fabriques pour les différents types d'objets : XMLInputFactory, XMLOutputFactory et XMLEventFactory. Des paramètres propres à une implémentation peuvent être manipulés en utilisant les méthodes getProperty() et setProperty() de ces fabriques.

La classe XMLInputFactory est une fabrique qui permet d'obtenir et de configurer une instance du parseur pour une lecture d'un document.

Il faut utiliser la méthode statique newInstance() pour obtenir une instance de la fabrique : elle détermine la classe à instancier en regardant dans l'ordre :

- Utilisation de la propriété système javax.xml.stream.XMLInputFactory
- Utilisation du fichier lib/xml.stream.properties dans le répertoire d'installation du JRE
- Utilisation de l'API Services avec le fichier META-INF/services/javax.stream.XMLInputFactory du jar
- Utilisation de l'instance par défaut

L'instance de la classe XMLInputFactory permet de configurer et d'instancier un parseur. La configuration se fait en utilisant des propriétés de la fabrique :

Propriété	Rôle
javax.xml.stream.isValidating	Permettre d'activer la validation du document en utilisant sa DTD (optionnelle, false par défaut)
javax.xml.stream.isCoalescing	Permettre d'indiquer si tous les événements de type characters contigus doivent être regroupés en un seul événement (false par défaut)
javax.xml.stream.isNamespaceAware	Supprimer le support des espaces de nommages (optionnelle, true par défaut)
javax.xml.stream.isReplacingEntityReferences	Permettre de demander le remplacement des entités de référence internes par leur valeur et ainsi émettre un événement de type Characters (true par défaut)
javax.xml.stream.isSupportingExternalEntities	
javax.xml.stream.reporter	
javax.xml.stream.resolver	Permettre de préciser une implémentation de type XMLResolver utilisée pour résoudre les entités externes
javax.xml.stream.allocator	
javax.xml.stream.supportDTD	Permettre de préciser si le support des DTD est activé (true par défaut)

La classe XMLOutputFactory est une fabrique qui permet de créer des objets pour écrire un document.

Il faut utiliser la méthode statique newInstance() pour obtenir une instance de la fabrique : elle détermine la classe à instancier en regardant dans l'ordre :

- Utilisation de la propriété système javax.xml.stream.XMLOutputFactory
- Utilisation du fichier lib/xml.stream.properties dans le répertoire d'installation du JRE
- Utilisation de l'API Services avec le fichier META-INF/services/javax.stream.XMLOutputFactory du jar
- Utilisation de l'instance par défaut

La classe XMLOuputFactory ne propose qu'une seule propriété :

Propriété	Rôle
javax.xml.stream.isRepairingNamespaces	Préciser si un préfixe par défaut doit être créé pour les espaces de nommages

La classe XMLEventFactory est une fabrique qui permet de créer des objets qui héritent de XMLEvent.

Il faut utiliser la méthode statique newInstance() pour obtenir une instance de la fabrique : elle détermine la classe à instancier en regardant dans l'ordre :

- Utilisation de la propriété système javax.xml.stream.XMLEventFactory
- Utilisation du fichier lib/xml.stream.properties dans le répertoire d'installation du JRE
- Utilisation de l'API Services avec le fichier META-INF/services/javax.stream.XMLEventFactory du jar
- Utilisation de l'instance par défaut

La classe XMLEventFactory ne possède pas de propriétés.

Pour modifier les propriétés de la fabrique, il faut utiliser la méthode setProperty() qui attend en paramètres le nom de la propriété et sa valeur.

Exemple :

```
    XMLInputFactory xmlif = XMLInputFactory.newInstance();

    xmlif.setProperty("javax.xml.stream.isCoalescing",Boolean.TRUE);
    xmlif.setProperty("javax.xml.stream.isReplacingEntityReferences", Boolean.TRUE);

    XMLStreamReader xmlsr = xmlif.createXMLStreamReader(new FileReader(
        "biblio.xml"));
```

Il est important de valoriser les propriétés avant de créer une instance d'un parseur. Une fois cette instance créée, il n'est plus possible de modifier ses propriétés.

Certains paramètres de configuration sont optionnels et ne sont donc pas obligatoirement supportés par une implémentation donnée. La méthode isPropertySupported() des fabriques XMLInputFactory et XMLOuputFactory permet de vérifier le support d'une propriété dont le nom est fourni en paramètre.

Exemple :

```
...
      XMLInputFactory xmlif =  XMLInputFactory.newInstance();
      if  (xmlif.isPropertySupported("javax.xml.stream.isReplacingEntityReferences"))  {
        System.out.println("javax.xml.stream.isReplacingEntityReferences  supporte");
        xmlif.setProperty("javax.xml.stream.isReplacingEntityReferences",  Boolean.TRUE);
      }
      XMLStreamReader xmlsr =  xmlif.createXMLStreamReader(new FileReader(
          "biblio.xml"));
  ...
```

XMLStreamReader et XMLEventReader possèdent la méthode getProperty() qui permet d'obtenir la valeur d'une propriété.

54.4. Le traitement d'un document XML avec l'API du type curseur

L'API de type curseur ne permet un parcours du document que vers l'avant : l'analyseur StAX parcourt le flux de caractères du document et émet des événements à la demande.

L'interface principale de l'API de type curseur est XMLStreamReader : elle propose des méthodes pour le parcours du document et de nombreuses méthodes qu'il ne faut utiliser que dans le contexte de l'événement en cours de traitement. Les informations retournées par ces méthodes le sont sous la forme de chaînes de caractères directement extraites du document : ceci rend les traitements d'analyse peu consommateurs en ressources.

Lors de l'analyse d'un document, l'instance de l'interface XMLStreamReader permet de se déplacer dans les différents éléments qui composent le document XML en cours de traitement. Ce déplacement ne peut se faire que vers l'avant. Un événement est émis par le parseur à la demande de l'application : celui-ci correspond au type de l'élément courant dans le document.

Il est nécessaire de créer une instance de la classe XMLStreamReader en utilisant la fabrique XMLInputFactory. Il faut obtenir une instance de la fabrique XMLInputFactory en utilisant sa méthode newInstance().

Exemple :

```
XMLInputFactory xmlif = XMLInputFactory.newInstance();
```

Il faut instancier un objet de type XMLStreamReader en utilisant la méthode createXMLStreamReader() de la fabrique. Cette méthode possède plusieurs surcharges acceptant en paramètre un objet de type Reader ou InputStream.

Exemple :

```
    XMLStreamReader xmlsr = xmlif.createXMLStreamReader(new FileReader(
        "biblio.xml"));
```

Le parseur permet donc d'avancer dans la lecture du document grâce aux méthodes hasNext() et next() de la classe XMLStreamReader qui parcourent séquentiellement les événements émis.

La méthode hasNext() renvoie un booléen qui précise si au moins un événement est encore disponible pour traitement. La méthode next() permet d'obtenir un identifiant sur l'événement suivant dans le flux de lecture du document.

Ceci permet d'itérer sur les événements jusqu'à ce qu'il n'y en ait plus à traiter. Il suffit pour cela d'appeler la méthode next() tant que hasNext() renvoie true.

Remarque : bien que les méthodes hasNext() et next() soient définies dans l'interface Iterator, l'interface XMLStreamReader n'hérite pas de cette interface.

La mise en oeuvre classique consiste donc à réaliser une itération sur les événements et à exécuter les traitements en fonction de ceux-ci.

Exemple :

```
    int eventType;
    while (xmlsr.hasNext()) {
      eventType = xmlsr.next();
      ...
    }
```

La méthode next() renvoie un code sous la forme d'un entier qui précise le type d'événement qui a été rencontré lors de la lecture d'un élément du document. Ce code correspond à un type défini sous la forme d'une constante dans l'interface XMLStreamConstants.

Lors du parcours des éléments qui composent le document en cours de traitement, un événement particulier permet de déterminer le type d'élément qui est en cours de traitement. Les événements qui peuvent être retournés par la méthode next() sont :

Evénement	Rôle
START_DOCUMENT	Le début du document (le prologue)
START_ELEMENT	Une balise ouvrante
ATTRIBUTE	Un attribut
NAMESPACE	La déclaration d'un espace de nommage
CHARACTERS	Du texte entre deux balises (pas forcément une balise ouvrante et sa balise fermante)
COMMENT	Un commentaire
SPACE	Un séparateur
PROCESSING_INSTRUCTION	Une instruction de traitement
DTD	Une DTD
ENTITY_REFERENCE	Une entité de référence
CDATA	Une section CData
END_ELEMENT	Une balise fermante
END_DOCUMENT	La fin du document
ENTITY_DECLARATION	La déclaration d'une entité
NOTATION_DECLARATION	La déclaration d'une notation

La méthode getEventType() permet de connaître le type de l'événement courant.

Il est nécessaire de traiter chaque événement en fonction des besoins : généralement un opérateur switch est utilisé pour définir les traitements de chaque événement utile.

Exemple :

```
      eventType = xmlsr.next();
      switch (eventType) {
      case XMLEvent.START_ELEMENT:
        System.out.println(xmlsr.getName());
        break;
      case XMLEvent.CHARACTERS:
        String chaine = xmlsr.getText();
        if (!xmlsr.isWhiteSpace()) {
          System.out.println("\t->\"" + chaine + "\"");
        }
        break;
      default:
        break;
      }
```

Le premier événement émis lors de l'analyse du document est de type START_DOCUMENT.

A chaque itération, les traitements peuvent prendre en compte ou ignorer l'événement en fonction des besoins. Ceci permet d'avoir une grande liberté sur l'analyse du document.

Exemple :

```
package fr.jmdoudoux.dej.stax;

import java.io.FileReader;

import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamReader;
import javax.xml.stream.events.XMLEvent;

public class TestStax1 {
  public static void main(String args[]) throws Exception {
    XMLInputFactory xmlif = XMLInputFactory.newInstance();
    XMLStreamReader xmlsr = xmlif.createXMLStreamReader(new FileReader(
        "biblio.xml"));
    int eventType;
    while (xmlsr.hasNext()) {
      eventType = xmlsr.next();
      switch (eventType) {
      case XMLEvent.START_ELEMENT:
        System.out.println(xmlsr.getName());
        break;
      case XMLEvent.CHARACTERS:
        String chaine = xmlsr.getText();
        if (!xmlsr.isWhiteSpace()) {
          System.out.println("\t->\""+chaine+"\"" );
        }
        break;
      default:
        break;
      }
    }
  }
}
```

Résultat :

```
{http://www.jmdoudoux.com/test/jaxb}bibliotheque
{http://www.jmdoudoux.com/test/jaxb}livre
{http://www.jmdoudoux.com/test/jaxb}titre
```

```
        ->"titre 1"
{http://www.jmdoudoux.com/test/jaxb}auteur
{http://www.jmdoudoux.com/test/jaxb}nom
        ->"nom 1"
{http://www.jmdoudoux.com/test/jaxb}prenom
        ->"prenom 1"
{http://www.jmdoudoux.com/test/jaxb}editeur
        ->"editeur 1"
{http://www.jmdoudoux.com/test/jaxb}livre
{http://www.jmdoudoux.com/test/jaxb}titre
        ->"titre 2"
{http://www.jmdoudoux.com/test/jaxb}auteur
{http://www.jmdoudoux.com/test/jaxb}nom
        ->"nom 2"
{http://www.jmdoudoux.com/test/jaxb}prenom
        ->"prenom 2"
{http://www.jmdoudoux.com/test/jaxb}editeur
        ->"editeur 2"
{http://www.jmdoudoux.com/test/jaxb}livre
{http://www.jmdoudoux.com/test/jaxb}titre
        ->"titre 3"
{http://www.jmdoudoux.com/test/jaxb}auteur
{http://www.jmdoudoux.com/test/jaxb}nom
        ->"nom 3"
{http://www.jmdoudoux.com/test/jaxb}prenom
        ->"prenom 3"
{http://www.jmdoudoux.com/test/jaxb}editeur
        ->"editeur 3"
```

L'interface XMLStreamReader propose des méthodes pour obtenir des données sur l'élément courant en fonction de l'événement lié à cet élément.

Plusieurs méthodes permettent d'obtenir des informations sur l'élément courant du curseur :
String getName();
String getLocalName();
String getNamespaceURI();
String getText();
String getElementText();
int getEventType();
Location getLocation();
int getAttributeCount();
QName getAttributeName(int);
String getAttributeValue(String, String);

Elle propose plusieurs méthodes pour obtenir des informations sur les attributs :
int getAttributeCount();
String getAttributeNamespace(int index);
String getAttributeLocalName(int index);
String getAttributePrefix(int index);
String getAttributeType(int index);
String getAttributeValue(int index);
String getAttributeValue(String namespaceUri,String localName);
boolean isAttributeSpecified(int index);

Elle propose plusieurs méthodes pour obtenir des informations sur les espaces de nommages :
int getNamespaceCount();
String getNamespacePrefix(int index);
String getNamespaceURI(int index);

Certaines méthodes sont utilisables selon l'événement pour obtenir un complément d'information sur l'entité courante. Ces méthodes ne sont utilisables que dans un contexte précis. Par exemple, la méthode getAttributeValue() n'est utilisable que sur un événement de type START_ELEMENT.

Le tableau ci-dessous précise les méthodes utilisables pour chaque événement :

Evénement	Méthodes
Tous	getProperty(), hasNext(), require(), close(), getNamespaceURI(), isStartElement(), isEndElement(), isCharacters(), isWhiteSpace(), getNamespaceContext(), getEventType(),getLocation(), hasText(), hasName()
START_ELEMENT	next(), getName(), getLocalName(), hasName(), getPrefix(), getAttributeXXX(), isAttributeSpecified(), getNamespaceXXX(), getElementText(), nextTag()
ATTRIBUTE	next(), nextTag(), getAttributeXXX(), isAttributeSpecified()
NAMESPACE	next(), nextTag(), getNamespaceXXX()
END_ELEMENT	next(), getName(), getLocalName(), hasName(), getPrefix(), getNamespaceXXX(), nextTag()
CHARACTERS	next(), getTextXXX(), nextTag()
CDATA	next(), getTextXXX(), nextTag()
COMMENT	next(), getTextXXX(), nextTag()
SPACE	next(), getTextXXX(), nextTag()
START_DOCUMENT	next(), getEncoding(), getVersion(), isStandalone(), standaloneSet(), getCharacterEncodingScheme(), nextTag()
END_DOCUMENT	close()
PROCESSING_INSTRUCTION	next(), getPITarget(), getPIData(), nextTag()
ENTITY_REFERENCE	next(), getLocalName(), getText(), nextTag()
DTD	next(), getText(), nextTag()

Il est préférable d'utiliser la méthode close() de la classe XMLStreamReader à la fin des traitements pour libérer les ressources.

Exemple :

```
    xmlsr.close();
```

Ci-dessous un exemple complet :

Exemple : le document à traiter

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:bibliotheque  xmlns:tns="http://www.jmdoudoux.com/test/stax" *
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.jmdoudoux.com/test/stax biblio2.xsd  ">
  <?MonTraitement?>
  <tns:livre>
      <!-- mon commentaire -->
    <tns:titre>titre 1</tns:titre>
    <tns:auteur>
      <tns:nom>nom 1</tns:nom>
      <tns:prenom>prenom  1</tns:prenom>
    </tns:auteur>
    <tns:editeur>editeur  1</tns:editeur>
  </tns:livre>
</tns:bibliotheque>
```

Exemple : parcours du document

```
package fr.jmdoudoux.dej.stax;

import java.io.FileReader;
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamReader;
import javax.xml.stream.events.XMLEvent;

public class TestStax6 {
  public static void main(String args[]) throws Exception {
    XMLInputFactory xmlif = XMLInputFactory.newInstance();
    XMLStreamReader xmlsr = xmlif.createXMLStreamReader(new FileReader(
        "biblio2.xml"));
    int eventType;
    while (xmlsr.hasNext()) {
      eventType = xmlsr.next();
      switch (eventType) {
      case XMLEvent.START_ELEMENT:
        System.out.println("START_ELEMENT : " + xmlsr.getName());
        break;
      case XMLEvent.START_DOCUMENT:
        System.out.println("START_DOCUMENT : " + xmlsr.getName());
        break;
      case XMLEvent.END_ELEMENT:
        System.out.println("END_ELEMENT : " + xmlsr.getName());
        break;
      case XMLEvent.END_DOCUMENT:
        System.out.println("END_DOCUMENT : ");
        break;
      case XMLEvent.COMMENT:
        System.out.println("COMMENT : "+ xmlsr.getText());
        break;
      case XMLEvent.CHARACTERS:
        System.out.println("CHARACTERS : ");
        break;
      case XMLEvent.PROCESSING_INSTRUCTION:
        System.out.println("PROCESSING_INSTRUCTION : "+ xmlsr.getPITarget());
        break;
      default:
        break;
      }
    }
  }
}
```

Résultat d'exécution :

```
START_ELEMENT : {http://www.jmdoudoux.com/test/stax}bibliotheque
CHARACTERS :
PROCESSING_INSTRUCTION : MonTraitement
CHARACTERS :
START_ELEMENT : {http://www.jmdoudoux.com/test/stax}livre
CHARACTERS :
COMMENT :  mon commentaire
CHARACTERS :
START_ELEMENT : {http://www.jmdoudoux.com/test/stax}titre
CHARACTERS :
END_ELEMENT : {http://www.jmdoudoux.com/test/stax}titre
CHARACTERS :
START_ELEMENT : {http://www.jmdoudoux.com/test/stax}auteur
CHARACTERS :
START_ELEMENT : {http://www.jmdoudoux.com/test/stax}nom
CHARACTERS :
END_ELEMENT : {http://www.jmdoudoux.com/test/stax}nom
CHARACTERS :
START_ELEMENT : {http://www.jmdoudoux.com/test/stax}prenom
CHARACTERS :
END_ELEMENT : {http://www.jmdoudoux.com/test/stax}prenom
CHARACTERS :
END_ELEMENT : {http://www.jmdoudoux.com/test/stax}auteur
CHARACTERS :
START_ELEMENT : {http://www.jmdoudoux.com/test/stax}editeur
CHARACTERS :
END_ELEMENT : {http://www.jmdoudoux.com/test/stax}editeur
CHARACTERS :
END_ELEMENT : {http://www.jmdoudoux.com/test/stax}livre
CHARACTERS :
END_ELEMENT : {http://www.jmdoudoux.com/test/stax}bibliotheque
```

```
END_DOCUMENT :
```

Chaque événement de type StartElement possède un événement correspondant de type EndElement même si le tag est sous sa forme réduite (<exemple/>)

Par défaut, les attributs n'émettent pas d'événement mais sont accessibles par une collection à partir de l'événement StartElement. Il en est de même avec les espaces de nommages.

Remarque : une partie texte du document peut émettre plusieurs événements de type Characters.

Durant le traitement du document, l'analyseur maintient une pile des espaces de nommages qui sont utilisés. Il est tout à fait possible d'interrompre le parcours du document : c'est un grand avantage de StAX de fournir le contrôle de la progression de l'analyse à l'application.

Par exemple, s'il n'est nécessaire de traiter qu'un seul tag, il suffit de tester la valeur du tag sur un événement de type START_ELEMENT, de réaliser les traitements sur le tag puis d'arrêter le parcours du document.

Exemple :

```
package fr.jmdoudoux.dej.stax;

import java.io.FileReader;

import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamReader;
import javax.xml.stream.events.XMLEvent;

public class TestStax7 {
  public static void main(String args[]) throws Exception {
    XMLInputFactory xmlif = XMLInputFactory.newInstance();
    XMLStreamReader xmlsr = xmlif.createXMLStreamReader(new FileReader(
        "biblio.xml"));
    int eventType;
    boolean encore = xmlsr.hasNext();

    while (encore) {

      eventType = xmlsr.next();
      if (eventType == XMLEvent.START_ELEMENT) {
        System.out.println("element=" + xmlsr.getLocalName());
        if (xmlsr.getLocalName().equals("editeur")) {
          xmlsr.next();
          System.out.println("Premier editeur : " + xmlsr.getText());
          encore = false;
        }
      }
      if (!xmlsr.hasNext()) {
        encore = false;
      }
    }
  }
}
```

Résultat :

```
element=bibliotheque
element=livre
element=titre
element=auteur
element=nom
element=prenom
element=editeur
Premier editeur : editeur 1
```

54.5. Le traitement d'un document XML avec l'API du type itérateur

L'API de type itérateur repose sur l'interface XMLEventReader qui représente le parseur et sur l'interface XMLEvent qui représente un événement. Ces événements sont réutilisables et peuvent être enrichis avec des événements personnalisés.

L'interface XMLEventReader propose plusieurs méthodes pour itérer sur le document XML et obtenir l'événement courant.

Les événements émis lors de l'analyse du document sont encapsulés dans un objet de type XMLEvent qui possède pour chaque événement une classe fille : Attribute, Characters, Comment, StartDocument, EndDocument, StartElement, EndElement, Namespace, DTD, EntityDeclaration, EntityReference, NotationDeclaration, et ProcessingInstruction. Chacune de ces classes possède des propriétés dédiées.

L'API de type itérateur propose plusieurs types d'événements qui implémentent l'interface XMLEvent :

Evénement	Rôle
StartDocument	Concerne le début du document (le prologue)
StartElement	Concerne le début d'un élément
EndElement	Concerne la fin d'un élément
Characters	Concerne une section de type CData ou une entité de type CharacterEntities. Les séparateurs sont également représentés par cet événement
EntityReference	Concerne une entité de référence
ProcessingInstruction	Concerne une instruction de traitement
Comment	Concerne un tag de commentaires
EndDocument	Concerne la fin du document
DTD	Concerne les informations sur la DTD
Attribut	Normalement les attributs sont représentés par un événement StartElement mais ils peuvent être représentés par cet événement
Namespace	Normalement les espaces de nommages sont représentés par un événement StartElement mais ils peuvent être représentés par cet événement

Remarque : les événements DTD, EntityDeclaration, EntityReférence, NotationDeclaration et ProcessingInstruction ne sont levés que si une DTD est associée au document en cours de traitement.

L'interface StartElement qui hérite de l'interface XMLEvent propose plusieurs méthodes pour obtenir des informations sur les attributs et les espaces de nommages :

- Attribute getAttributeByName() : renvoie un attribut à partir de son nom
- Iterator getAttributes() : permet de parcourir tous les attributs de l'élément
- NamespaceContext getNamespaceContext : renvoie le contexte de l'espace de nommage
- Iterator getNamespaces : permet de parcourir tous les espaces de nommages de l'élément
- String getNamespaceURI : retourne la valeur d'un préfixe dans le contexte de l'élément

L'interface XMLEventReader analyse le document XML et émet des événements sous la forme d'objets de type XMLEvent.

L'utilisation de XMLEventReader est similaire à celle de XMLStreamReader. XMLEventReader permet en plus de connaître le prochain événement grâce à la méthode peek() sans consommer l'événement ce qui permet d'anticiper sur les traitements.

L'interface XMLEventReader définit plusieurs méthodes :

Méthode	Rôle

XMLEvent nextEvent()	obtenir et consommer l'événement suivant (avance dans l'itération)
boolean hasNext()	préciser s'il y a encore un événement à traiter
XMLEvent peek()	obtenir le prochain événement sans le consommer (l'itération ne bouge pas)
next()	avancer dans l'itération et renvoie le prochain événement
XMLEvent nextTag()	obtenir le prochain événement dans l'itération en ignorant les séparateurs pour renvoyer le prochain événement de type START_ELEMENT ou END_ELEMENT

Pour utiliser l'API de type itérateur, plusieurs étapes sont nécessaires.

Il faut obtenir une instance de la fabrique XMLInputFactory en utilisant sa méthode statique newInstance().

Exemple :

```
    XMLInputFactory xmlif = XMLInputFactory.newInstance();
```

Il faut instancier un objet de type XMLEventReader en utilisant la méthode createXMLEventReader() de la fabrique qui possède plusieurs surcharges acceptant en paramètre un objet de type Reader ou InputStream.

Exemple :

```
    XMLEventReader xmler = xmlif.createXMLEventReader(new FileReader(
        "biblio.xml"));
```

La méthode hasNext() renvoie un booléen qui précise si au moins un événement est encore disponible. La méthode nextEvent() permet d'obtenir l'événement suivant. Il suffit de faire une itération tant que la méthode hasNext() renvoie true et d'appeler dans cette itération la méthode nextEvent() pour parcourir tout le document.

Exemple :

```
    XMLEvent event;
    while (xmler.hasNext()) {
      event = xmler.nextEvent();
      ...
    }
```

L'interface XMLEvent propose la méthode getEventType() pour connaître le type de l'événement. Elle propose aussi plusieurs méthodes :

- isXXX() pour chaque événement qui renvoie un booléen indiquant si l'événement est du type XXX.
- asXXX() pour chaque événement qui renvoie une instance de XXX correspondant à l'événement qui est du type XXX.

Exemple complet :

```
package fr.jmdoudoux.dej.stax;

import java.io.*;
import javax.xml.stream.*;
import javax.xml.stream.events.*;

public class TestStax2 {

  public static void main(String args[]) throws Exception {

    XMLInputFactory xmlif = XMLInputFactory.newInstance();
    XMLEventReader xmler = xmlif.createXMLEventReader(new FileReader(
        "biblio.xml"));
    XMLEvent event;
    while (xmler.hasNext()) {
      event = xmler.nextEvent();
      if (event.isStartElement()) {
        System.out.println(event.asStartElement().getName());
      } else if (event.isCharacters()) {
        if (!event.asCharacters().isWhiteSpace()) {
          System.out.println("\t>" + event.asCharacters().getData());
        }
      }
    }
  }
}
```

Résultat de l'éxécution :

```
{http://www.jmdoudoux.com/test/jaxb}bibliotheque
{http://www.jmdoudoux.com/test/jaxb}livre
{http://www.jmdoudoux.com/test/jaxb}titre
        >titre 1
{http://www.jmdoudoux.com/test/jaxb}auteur
{http://www.jmdoudoux.com/test/jaxb}nom
        >nom 1
{http://www.jmdoudoux.com/test/jaxb}prenom
        >prenom 1
{http://www.jmdoudoux.com/test/jaxb}editeur
        >editeur 1
{http://www.jmdoudoux.com/test/jaxb}livre
{http://www.jmdoudoux.com/test/jaxb}titre
        >titre 2
{http://www.jmdoudoux.com/test/jaxb}auteur
{http://www.jmdoudoux.com/test/jaxb}nom
        >nom 2
{http://www.jmdoudoux.com/test/jaxb}prenom
        >prenom 2
{http://www.jmdoudoux.com/test/jaxb}editeur
        >editeur 2
{http://www.jmdoudoux.com/test/jaxb}livre
{http://www.jmdoudoux.com/test/jaxb}titre
        >titre 3
{http://www.jmdoudoux.com/test/jaxb}auteur
{http://www.jmdoudoux.com/test/jaxb}nom
        >nom 3
{http://www.jmdoudoux.com/test/jaxb}prenom
        >prenom 3
{http://www.jmdoudoux.com/test/jaxb}editeur
        >editeur 3
```

Comme avec l'API de type curseur, il est possible avec l'API de type itérateur d'interrompre le traitement du document.

Exemple :

```
package fr.jmdoudoux.dej.stax;

import java.io.*;
import javax.xml.stream.*;
import javax.xml.stream.events.*;

public class TestStax3 {

  public static void main(String args[]) throws Exception {
    boolean termine = false;
    XMLInputFactory xmlif = XMLInputFactory.newInstance();
    FileReader fr = new FileReader("biblio.xml");
    XMLEventReader xmler = xmlif.createXMLEventReader(fr);
    XMLEvent event;
    termine = !xmler.hasNext();

    while (!termine) {
      event = xmler.nextEvent();
      if (event.isStartElement()) {
        if (event.asStartElement().getName().getLocalPart() == "editeur") {
          event = xmler.nextEvent();
          System.out.println("Premier editeur = "+event.asCharacters().getData());
          termine = true;
        }
      }
      if (!termine && !xmler.hasNext()) {
        termine = true;
      }
    }
      fr.close();
      xmler.close();
  }
}
```

Les événements sont émis dans l'ordre de rencontre des éléments lors du parcours du document par le parseur.

Si le document XML est syntaxiquement correct, alors chaque événement de type StartElement possède un événement de type EndElement correspondant.

54.6. La mise en oeuvre des filtres

StAX propose la mise en oeuvre de filtres pour n'obtenir que les événements désirés.

Pour l'API de type itérateur, l'interface EventFilter définit la méthode accept() qui attend en paramètre un objet de type XMLEvent et renvoie un booléen qui précise si cet événement doit être traité.

Exemple :

```
package fr.jmdoudoux.dej.stax;

import javax.xml.stream.EventFilter;
import javax.xml.stream.events.XMLEvent;

public class MonEventFilter implements EventFilter {

  public boolean accept(XMLEvent event) {

    if (event.isStartElement() || event.isEndElement())
      return true;
    else
      return false;
  }

}
```

Pour utiliser le filtre, il faut créer une instance de la classe XMLStreamReader en utilisant la méthode createFilteredReader() de la fabrique XMLInputFactory. Elle attend en paramètre l'instance de XMLStreamReader pour le traitement du document et le filtre.

Exemple :

```
package fr.jmdoudoux.dej.stax;

import java.io.FileReader;

import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamReader;
import javax.xml.stream.events.XMLEvent;

public class TestStax11 {
  public static void main(String args[]) throws Exception {
    XMLInputFactory xmlif = XMLInputFactory.newInstance();

    XMLStreamReader xmlr = xmlif.createXMLStreamReader(
        new FileReader("biblio.xml"));
    XMLStreamReader xmlsr =
      xmlif.createFilteredReader(xmlr, new MonStreamFilter());

    while (xmlsr.hasNext()) {
      int eventType = xmlsr.next();
      switch (eventType) {
      case XMLEvent.START_ELEMENT:
        System.out.println("START_ELEMENT "+xmlsr.getName());
        break;
      case XMLEvent.END_ELEMENT:
        System.out.println("END_ELEMENT "+xmlsr.getName());
        break;
      default:
        System.out.println("AUTRE "+xmlsr.getName());
        break;
      }
    }
  }
}
```

Résultat :

```
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}livre
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}titre
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}titre
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}auteur
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}nom
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}nom
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}prenom
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}prenom
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}auteur
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}editeur
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}editeur
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}livre
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}livre
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}titre
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}titre
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}auteur
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}nom
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}nom
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}prenom
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}prenom
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}auteur
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}editeur
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}editeur
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}livre
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}livre
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}titre
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}titre
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}auteur
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}nom
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}nom
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}prenom
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}prenom
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}auteur
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}editeur
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}editeur
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}livre
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}bibliotheque
```

Pour l'API de type curseur, l'interface StreamFilter définit la méthode accept() qui attend en paramètre un objet de type XMLStreamReader et renvoie un booléen qui précise si l'événement courant doit être traité.

Exemple :

```
package fr.jmdoudoux.dej.stax;

import javax.xml.stream.StreamFilter;
import javax.xml.stream.XMLStreamReader;
```

```
public class MonStreamfilter implements StreamFilter {

  public boolean accept(XMLStreamReader reader) {
    if(reader.isStartElement() || reader.isEndElement())
      return true;
    else
      return false;
  }

}
```

Pour utiliser le filtre, il faut créer une instance de la classe XMLEventReader en utilisant la méthode createFilteredReader de la fabrique XMLInputFactory. Elle attend en paramètre l'instance de XMLEventReader pour le traitement du document et le filtre.

Exemple :

```
package fr.jmdoudoux.dej.stax;

import java.io.*;
import javax.xml.stream.*;
import javax.xml.stream.events.*;

public class TestStAX12 {

  public static void main(String args[]) throws Exception {

    XMLInputFactory xmlif = XMLInputFactory.newInstance();
    XMLEventReader xmlr = xmlif.createXMLEventReader(new FileReader(
        "biblio.xml"));
    XMLEventReader xmler = xmlif.createFilteredReader(xmlr,
        new MonEventFilter());

    XMLEvent event;
    while (xmler.hasNext()) {
      event = xmler.nextEvent();
      if (event.isStartElement()) {
        System.out.println("StartElement=" + event.asStartElement().getName());
      } else if (event.isEndElement()) {
        System.out.println("EndElement=" + event.asEndElement().getName());
      } else {
        System.out.println("Autre");
      }
    }
  }
}
```

Résultat :

```
StartElement={http://www.jmdoudoux.com/test/jaxb}bibliotheque
StartElement={http://www.jmdoudoux.com/test/jaxb}livre
StartElement={http://www.jmdoudoux.com/test/jaxb}titre
EndElement={http://www.jmdoudoux.com/test/jaxb}titre
StartElement={http://www.jmdoudoux.com/test/jaxb}auteur
StartElement={http://www.jmdoudoux.com/test/jaxb}nom
EndElement={http://www.jmdoudoux.com/test/jaxb}nom
StartElement={http://www.jmdoudoux.com/test/jaxb}prenom
EndElement={http://www.jmdoudoux.com/test/jaxb}prenom
EndElement={http://www.jmdoudoux.com/test/jaxb}auteur
StartElement={http://www.jmdoudoux.com/test/jaxb}editeur
EndElement={http://www.jmdoudoux.com/test/jaxb}editeur
EndElement={http://www.jmdoudoux.com/test/jaxb}livre
StartElement={http://www.jmdoudoux.com/test/jaxb}livre
StartElement={http://www.jmdoudoux.com/test/jaxb}titre
EndElement={http://www.jmdoudoux.com/test/jaxb}titre
StartElement={http://www.jmdoudoux.com/test/jaxb}auteur
StartElement={http://www.jmdoudoux.com/test/jaxb}nom
EndElement={http://www.jmdoudoux.com/test/jaxb}nom
StartElement={http://www.jmdoudoux.com/test/jaxb}prenom
EndElement={http://www.jmdoudoux.com/test/jaxb}prenom
EndElement={http://www.jmdoudoux.com/test/jaxb}auteur
```

```
StartElement={http://www.jmdoudoux.com/test/jaxb}editeur
EndElement={http://www.jmdoudoux.com/test/jaxb}editeur
EndElement={http://www.jmdoudoux.com/test/jaxb}livre
StartElement={http://www.jmdoudoux.com/test/jaxb}livre
StartElement={http://www.jmdoudoux.com/test/jaxb}titre
EndElement={http://www.jmdoudoux.com/test/jaxb}titre
StartElement={http://www.jmdoudoux.com/test/jaxb}auteur
StartElement={http://www.jmdoudoux.com/test/jaxb}nom
EndElement={http://www.jmdoudoux.com/test/jaxb}nom
StartElement={http://www.jmdoudoux.com/test/jaxb}prenom
EndElement={http://www.jmdoudoux.com/test/jaxb}prenom
EndElement={http://www.jmdoudoux.com/test/jaxb}auteur
StartElement={http://www.jmdoudoux.com/test/jaxb}editeur
EndElement={http://www.jmdoudoux.com/test/jaxb}editeur
EndElement={http://www.jmdoudoux.com/test/jaxb}livre
EndElement={http://www.jmdoudoux.com/test/jaxb}bibliotheque
```

54.7. L'écriture un document XML avec l'API de type curseur

L'interface XMLStreamWriter propose des fonctionnalités simples et de bas niveau pour écrire un document.

L'interface XMLStreamWriter définit les méthodes pour un objet capable de réécrire un document en cours de parcours ou d'écrire un nouveau document.

Une instance d'un tel objet est obtenue en utilisant la fabrique XMLOutputFactory.

Exemple :

```
    XMLStreamWriter writer = XMLOutputFactory.newInstance().
        createXMLStreamWriter(outStream);
```

L'interface XMLStreamWriter propose de nombreuses méthodes pour ajouter des noeuds de différents types au document en cours de rédaction :

Méthode	Rôle
writeStartDocument()	ajouter le prologue du document
writeEndDocument()	ajouter tous les éléments de type fin requis pour terminer le document
writeStartElement()	ajouter un élément de type début
writeEndElement()	ajouter un élément de type fin
writeComment()	ajouter un élément de type commentaire
writeNamespace()	ajouter un espace de nommage
writeCharacters()	ajouter un élément de type texte
writeProcessingInstruction()	ajouter une instruction de traitement

Remarque : chaque méthode writeStartXxx() doit avoir un appel à la méthode writeEndXxx() correspondante dans les traitements.

Il faut obtenir une instance de la fabrique XMLOuputFactory en utilisant sa méthode newInstance().

Exemple :

```
    XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();
```

Il faut instancier un objet de type FileWriter qui va encapsuler le fichier où sera stocké le document XML

Exemple :

```
FileWriter output = new FileWriter(new File("test.xml"));
```

Il faut obtenir une instance de l'interface XMLStreamWriter en utilisant la méthode createXMLStreamWriter() de la fabrique.

Exemple :

```
XMLStreamWriter xmlsw = outputFactory.createXMLStreamWriter(output);
```

Il faut créer le prologue du document en utilisant la méthode writeStartDocument() qui attend en paramètre le nom du jeu de caractères d'encodage et la version de xml. Ces deux informations ne sont utilisées que comme valeurs des attributs encoding et version du prologue.

Exemple :

```
xmlsw.writeStartDocument("Cp1252", "1.0");
```

Pour préciser le jeu de caractères utilisé pour encoder le document XML, il est nécessaire d'utiliser une version surchargée de la méthode createXMLStreamWriter().

Exemple :

```
FileOutputStream output = new FileOutputStream("test.xml");
XMLStreamWriter xmlsw = outputFactory.createXMLStreamWriter(output, "UTF-8");
xmlsw.writeStartDocument("UTF-8", "1.0");
```

La création d'une balise dans le document à la position courante se fait en utilisant la méthode writeStartElement(). Cette méthode possède trois surcharges qui permettent de préciser le nom de la balise, son préfixe et l'URI de son espace de nommage.

La méthode writeNamespace() qui attend en paramètres un préfixe et une uri permet de définir un espace de nommage pour la balise courante.

Exemple :

```
xmlsw.writeNamespace("tns", "http://www.jmdoudoux.com/test/stax");
```

La méthode writeAttribut() permet de définir un attribut pour la balise courante. Elle possède plusieurs surcharges qui attendent en paramètres le nom de l'attribut, sa valeur, un préfixe et l'uri de l'espace de nommage

Exemple :

```
xmlsw.writeAttribut("xsi", "http://www.w3.org/2001/XMLSchema-instance");
```

La méthode writeCharacters() qui peut être utilisée avec une chaîne de caractères ou un tableau de caractères permet d'écrire un noeud de type texte dans la balise courante.

Exemple :

```
  xmlsw.writeCharacters("titre "+i);
```

La méthode writeCharacters() permet d'ajouter du texte dans le document en échappant les caractères utilisés par XML (<, >, &, ...).

La méthode writeEndElement() permet de créer un balise fermante à la balise courante. Elle détermine automatiquement le nom de la balise courante pour créer la balise nécessaire. Son appel est obligatoire pour chaque balise ouverte.

Exemple :

```
      xmlsw.writeEndElement();
```

Une balise de commentaires peut être créée en utilisant la méthode writeComment().

Exemple :

```
    xmlsw.writeComment("Fichier de test XMLStreamWriter");
```

La méthode writeProcessingInstruction() permet d'ajouter une balise de type instruction de traitement.

La méthode writeEndDocument() permet de créer toutes les balises fermantes requises à partir de la balise courante jusqu'à la balise racine.

Une fois le document complet, il est nécessaire d'utiliser les méthodes flush() et close() de la classe XMLStreamWriter pour enregistrer le document XML dans le fichier.

Exemple :

```
    xmlsw.flush();
    xmlsw.close();
```

Exemple complet :

```
package fr.jmdoudoux.dej.stax;

import java.io.StringWriter;

import javax.xml.stream.XMLOutputFactory;
import javax.xml.stream.XMLStreamWriter;

public class TestStax5 {

  public static void main(String args[]) throws Exception {

    String ns = "http://www.jmdoudoux.com/test/stax";

    StringWriter strw = new StringWriter();
    XMLOutputFactory output = XMLOutputFactory.newInstance();
    XMLStreamWriter writer = output.createXMLStreamWriter(strw);
    writer.writeStartDocument();
    writer.setPrefix("tns",ns);
    writer.setDefaultNamespace(ns);
    writer.writeStartElement(ns,"bibliotheque");
      writer.writeNamespace("tns",ns);
      writer.writeStartElement(ns,"livre");
        writer.writeAttribute("id","1");
        writer.writeStartElement(ns,"titre");
          writer.writeCharacters("titre1");
        writer.writeEndElement();
        writer.writeStartElement(ns,"auteur");
          writer.writeStartElement(ns, "nom");
          writer.writeCharacters("nom1");
          writer.writeEndElement();
          writer.writeStartElement(ns,"prenom");
          writer.writeCharacters("prenom1");
          writer.writeEndElement();
        writer.writeEndElement();
        writer.writeStartElement(ns,"editeur");
        writer.writeCharacters("editeur1");
        writer.writeEndElement();
      writer.writeEndElement();
    writer.writeEndElement();
    writer.flush();

    System.out.println(strw.toString());
  }
}
```

Remarque : l'indentation des méthodes writeXxx() permet de vérifier qu'aucun appel de méthode n'a été oublié.

Résultat :

```
  <?xml version="1.0" ?>
  <bibliotheque  xmlns:tns="http://www.jmdoudoux.com/test/stax">
     <tns:livre id="1">
       <tns:titre>titre1</tns:titre>
       <tns:auteur>
         <tns:nom>nom1</nom>
         <tns:prenom>prenom1</tns:prenom>
       </tns:auteur>
       <tns:editeur>editeur1</tns:editeur>
    </tns:livre>
  </bibliotheque>
```

Attention : une implémentation de l'interface XMLStreamWriter n'a pas l'obligation de vérifier que le document créé soit bien formé. Par exemple, l'oubli d'un appel à la méthode writeEndElement() pour un tag provoque un décalage dans la balise de fin, il en résulte l'absence de la balise de fermeture du tag racine.

Exemple complet :

```
package fr.jmdoudoux.dej.stax;

import java.io.File;
import java.io.FileWriter;

import javax.xml.stream.XMLOutputFactory;
import javax.xml.stream.XMLStreamWriter;

public class TestStax4 {

  public static void main(String args[]) throws Exception {

    XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();
    FileWriter output = new FileWriter(new File("test.xml"));
    XMLStreamWriter xmlsw = outputFactory.createXMLStreamWriter(output);
    xmlsw.writeStartDocument("Cp1252", "1.0");
    xmlsw.writeComment("Fichier de test XMLStreamWriter");
    xmlsw.writeStartElement("tns", "bibliotheque",
        "http://www.jmdoudoux.com/test/stax");
    xmlsw.writeNamespace("tns", "http://www.jmdoudoux.com/test/stax");
    xmlsw.writeNamespace("xsi", "http://www.w3.org/2001/XMLSchema-instance");
    xmlsw.writeAttribute("xsi:schemaLocation",
        "http://www.jmdoudoux.com/test/stax/biblio.xsd");

    for (int i = 1; i > 4; i++) {

      xmlsw.writeStartElement("tns", "livre",
          "http://www.jmdoudoux.com/test/stax");

      xmlsw.writeStartElement("tns", "titre",
          "http://www.jmdoudoux.com/test/stax");
      xmlsw.writeCharacters("titre "+i);
      xmlsw.writeEndElement();

      xmlsw.writeStartElement("tns", "auteur",
          "http://www.jmdoudoux.com/test/stax");
      xmlsw.writeStartElement("tns", "nom",
          "http://www.jmdoudoux.com/test/stax");
      xmlsw.writeCharacters("nom "+i);
      xmlsw.writeEndElement();
      xmlsw.writeStartElement("tns", "prenom",
          "http://www.jmdoudoux.com/test/stax");
      xmlsw.writeCharacters("prenom "+i);
      xmlsw.writeEndElement();
      xmlsw.writeEndElement();

      xmlsw.writeStartElement("tns", "editeur",
          "http://www.jmdoudoux.com/test/stax");
      xmlsw.writeCharacters("editeur "+i);
      xmlsw.writeEndElement();

      xmlsw.writeEndElement();
    }

    xmlsw.writeEndElement();
    xmlsw.flush();
    xmlsw.close();

  }
}
```

Résultat :

```
  <?xml  version="1.0" encoding="Cp1252"?><!--Fichier de test  XMLStreamWriter-->
  <tns:bibliotheque  xmlns:tns="http://www.jmdoudoux.com/test/stax"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.jmdoudoux.com/test/stax/biblio.xsd">
    <tns:livre>
       <tns:titre>titre 1</tns:titre>
       <tns:auteur>
         <tns:nom>nom 1</tns:nom>
         <tns:prenom>prenom 1</tns:prenom>
       </tns:auteur>
       <tns:editeur>editeur 1</tns:editeur>
     </tns:livre>
     <tns:livre>
       <tns:titre>titre 2</tns:titre>
       <tns:auteur>
         <tns:nom>nom 2</tns:nom>
         <tns:prenom>prenom 2</tns:prenom>
       </tns:auteur>
       <tns:editeur>editeur 2</tns:editeur>
     </tns:livre>
     <tns:livre>
       <tns:titre>titre 3</tns:titre>
       <tns:auteur>
         <tns:nom>nom 3</tns:nom>
         <tns:prenom>prenom 3</tns:prenom>
       </tns:auteur>
       <tns:editeur>editeur 3</tns:editeur>
     </tns:livre>
  </tns:bibliotheque>
```

54.8. L'écriture un document XML avec l'API de type itérateur

L'interface XMLEventWriter propose des fonctionnalités à l'API de type itérateur pour écrire un document XML : celle-ci est particulièrement adaptée à la réécriture d'un document en cours de traitement par l'API de type itérateur mais elle peut aussi être utilisée pour créer un nouveau document. Elle propose des méthodes pour créer un document XML à partir d'objets de type XMLEvent.

Une instance de type XMLEventWriter est obtenue en utilisant la fabrique XMLOutPutFactory.

Elle possède plusieurs méthodes dont les principales sont :

Méthode	Rôle
void flush()	Permettre de vider le cache et d'écrire les données qu'il contient
void close()	Fermer le flux d'écriture

void add(XMLEvent)	Ajouter un élément dans le document

Les événements sont ajoutés au fur et à mesure et ne peuvent plus être modifiés une fois ajoutés. L'ajout d'attributs ou d'espaces de nommages se fait toujours sur le dernier élément de type StartElement ajouté dans le flux.

La méthode setPrefix() permet d'associer un préfixe à un espace de nommage.

Il faut instancier une occurrence de l'interface XMLEventWriter à partir d'une fabrique de type XMLOutputFactory.

Exemple :

```
    XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();
    XMLEventWriter writer = outputFactory.createXMLEventWriter(new FileWriter(
        "test2.xml"));
```

Il faut obtenir une instance de la fabrique XMLEventFactory en utilisant sa méthode newInstance().

Exemple :

```
    XMLEventFactory eventFactory = XMLEventFactory.newInstance();
```

Cette fabrique permet de créer des instances des événements qui seront ajoutés dans le document.

Exemple :

```
    writer.add(eventFactory.createStartDocument());
```

Une fois le document terminé, il suffit d'appeler les méthodes flush() et close().

Exemple :

```
package fr.jmdoudoux.dej.stax;

import java.io.FileWriter;

import javax.xml.stream.XMLEventFactory;
import javax.xml.stream.XMLEventWriter;
import javax.xml.stream.XMLOutputFactory;

public class TestStax8 {

  private static final String NS_TNS = "http://www.jmdoudoux.com/test/stax";

  private static final String PREFIX_TNS = "tns";

  public static void main(String args[]) throws Exception {
    XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();
    XMLEventWriter writer = outputFactory.createXMLEventWriter(new FileWriter(
        "test2.xml"));
    XMLEventFactory eventFactory = XMLEventFactory.newInstance();

    writer.setPrefix(PREFIX_TNS, NS_TNS);
    writer.add(eventFactory.createStartDocument());
    writer.add(eventFactory.createStartElement(PREFIX_TNS, NS_TNS,
        "bibliotheque"));
    writer.add(eventFactory.createNamespace(PREFIX_TNS, NS_TNS));
    writer.add(eventFactory.createProcessingInstruction("MonTraitement", ""));
    writer.add(eventFactory.createStartElement(PREFIX_TNS, NS_TNS, "livre"));
    writer.add(eventFactory.createComment("mon commentaire"));
    writer.add(eventFactory.createStartElement(PREFIX_TNS, NS_TNS, "titre"));
    writer.add(eventFactory.createCharacters("titre 1"));
    writer.add(eventFactory.createEndElement(PREFIX_TNS, NS_TNS, "titre"));
    writer.add(eventFactory.createStartElement(PREFIX_TNS, NS_TNS, "auteur"));
    writer.add(eventFactory.createStartElement(PREFIX_TNS, NS_TNS, "nom"));
    writer.add(eventFactory.createCharacters("nom 1"));
    writer.add(eventFactory.createEndElement(PREFIX_TNS, NS_TNS, "nom"));
    writer.add(eventFactory.createStartElement(PREFIX_TNS, NS_TNS, "prenom"));
    writer.add(eventFactory.createCharacters("prenom 1"));
    writer.add(eventFactory.createEndElement(PREFIX_TNS, NS_TNS, "prenom"));
    writer.add(eventFactory.createEndElement(PREFIX_TNS, NS_TNS, "auteur"));
    writer.add(eventFactory.createStartElement(PREFIX_TNS, NS_TNS, "editeur"));
    writer.add(eventFactory.createCharacters("editeur 1"));
    writer.add(eventFactory.createEndElement(PREFIX_TNS, NS_TNS, "editeur"));
    writer.add(eventFactory.createEndElement(PREFIX_TNS, NS_TNS, "livre"));
    writer.add(eventFactory
        .createEndElement(PREFIX_TNS, NS_TNS, "bibliotheque"));

    writer.add(eventFactory.createEndDocument());
    writer.flush();
    writer.close();

  }
}
```

Résultat :

```
<?xml version="1.0"?>
<tns:bibliotheque  xmlns:tns="http://www.jmdoudoux.com/test/stax">
  <?MonTraitement  ?>
    <tns:livre>
      <!--mon commentaire-->
      <tns:titre>titre  1</tns:titre>
      <tns:auteur>
        <tns:nom>nom  1</tns:nom>
        <tns:prenom>prenom  1</tns:prenom>
      </tns:auteur>
    <tns:editeur>editeur  1</tns:editeur>
  </tns:livre>>/p<
</tns:bibliotheque>
```

Il est aussi possible d'utiliser l'API de type itérateur en lecture et en écriture simultanément.

Exemple :

```
package fr.jmdoudoux.dej.stax;

import java.io.FileReader;
import java.io.FileWriter;

import javax.xml.stream.XMLEventFactory;
import javax.xml.stream.XMLEventReader;
import javax.xml.stream.XMLEventWriter;
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLOutputFactory;
import javax.xml.stream.events.Characters;
import javax.xml.stream.events.XMLEvent;

public class TestStax9 {

  public static void main(String args[]) throws Exception {
    XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();

    XMLInputFactory xmlif = XMLInputFactory.newInstance();
    FileReader fr = new FileReader("biblio.xml");
    XMLEventReader reader = xmlif.createXMLEventReader(fr);

    XMLEventFactory eventFactory = XMLEventFactory.newInstance();
    XMLEventWriter writer = outputFactory.createXMLEventWriter(new FileWriter(
        "test3.xml"));

    while (reader.hasNext()) {
      XMLEvent event = (XMLEvent) reader.next();
      if (event.getEventType() == XMLEvent.CHARACTERS) {
        Characters characters = event.asCharacters();
        if (!characters.isWhiteSpace()) {
          writer.add(eventFactory.createCharacters(characters.getData() + " modif"));
        }
      } else {
        writer.add(event);
      }
    }
    writer.flush();

    writer.close();

  }
}
```

Résultat :

```
<?xml  version="1.0"?>
  <tns:bibliotheque  xmlns:tns="http://www.jmdoudoux.com/test/jaxb"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.jmdoudoux.com/test/jaxb  biblio.xsd ">
     <tns:livre>
       <tns:titre>titre 1 modif</tns:titre>
       <tns:auteur>
         <tns:nom>nom 1 modif</tns:nom>
         <tns:prenom>prenom 1 modif</tns:prenom>
       </tns:auteur>
       <tns:editeur>editeur 1 modif</tns:editeur>
     </tns:livre>
     <tns:livre>
       <tns:titre>titre 2 modif</tns:titre>
       <tns:auteur>
         <tns:nom>nom 2 modif</tns:nom>
         <tns:prenom>prenom 2 modif</tns:prenom>
       </tns:auteur>
       <tns:editeur>editeur 2 modif</tns:editeur>
     </tns:livre>
     <tns:livre>
       <tns:titre>titre 3 modif</tns:titre>
       <tns:auteur>
         <tns:nom>nom 3 modif</tns:nom>
         <tns:prenom>prenom 3 modif</tns:prenom>
       </tns:auteur>
       <tns:editeur>editeur 3 modif</tns:editeur>
     </tns:livre>
</tns:bibliotheque>
```

54.9. La comparaison entre SAX, DOM et StAX

Les parseurs avant l'arrivée de StAX utilisent deux méthodes principales pour traiter un document XML :

- ceux basés sur un modèle événementiel utilisé par SAX notamment
- ceux basés sur un modèle reposant sur un arbre d'objets utilisé par DOM notamment

Ces deux modèles ont chacun leurs avantages et leurs inconvénients.

	Avantages	Inconvénients
SAX	• grande efficacité • faible ressource nécessaire • API simple • traitement au fur et à mesure de la lecture	• l'état du document doit être conservé « manuellement » • le document tout entier doit être parcouru • ne peut être utilisé que pour lire un document • traitements séquentiels
DOM	• lecture aléatoire dans l'arbre du document	

	• permet la mise à jour d'un document	• ressources nécessaires importantes proportionnelles à la taille du document • API plus complexe car non développée spécifiquement pour Java

Les trois API de JAXP permettant d'analyser un document XML ont chacune des points forts et des points faibles dont il faut tenir compte pour déterminer quelle API sera la mieux adaptée en fonction des besoins.

	SAX	StAX	DOM
Type de traitements	Événement de type push	Événement de type pull	Arbre d'objets en mémoire
Facilité de mise en oeuvre	Moyenne	Elevée	Moyenne
Support XPath	Non	Non	Oui
Consommation en ressources	Faible	Faible	Dépendante de la taille du document
Sens de parcours	Vers l'avant uniquement	Vers l'avant uniquement	Libre
Lecture	Oui	Oui	Oui
Ecriture	Non	Oui	Oui
Modification	Non	Non	Oui

Même si l'API StAX est basée sur des événements, ses fonctionnalités la placent entre les deux autres types de parsers.

SAX et StAX reposent tous les deux sur un traitement par flux : le document est parcouru et traité au fur et à mesure. Ce type de traitement est efficace et peu consommateur en ressources : il est donc particulièrement adapté au traitement de gros documents.

L'avantage de StAX par rapport à SAX est de donner la possibilité au développeur de demander le prochain événement et de le traiter si nécessaire plutôt que de fournir des traitements dans des fonctions de type « callback » appelées par le parseur. Ceci donne au développeur un meilleur contrôle sur les traitements en facilitant leur mise en oeuvre et permet à tout moment d'interrompre le traitement du parseur sans attendre le traitement de tout le document.

SAX lit et analyse le document au fur et à mesure et émet des événements à destination d'un handler définit dans l'application qui est composée de méthodes de type callback. Ces méthodes sont automatiquement exécutées par le parseur en fonction des événements émis par ce dernier lors de la lecture du document. C'est donc le parseur qui a le contrôle sur les traitements d'analyse du document : ce type de traitement est dit push (c'est le parseur qui émet des événements à son initiative vers l'application). Il nécessite le parcours de tout le document. SAX ne permet pas d'écrire un document XML.

SAX n'est pas aussi simple à mettre en oeuvre que StAX puisqu'il faut développer un handler qui va traiter les événements émis sous la forme de callback : le code des traitements pour sa mise en oeuvre peut être rapidement complexe. Stax est plus simple que SAX : c'est une forme de traitement de type pull (les événements sont émis par le parseur à la demande de l'application) ce qui permet de donner le contrôle de l'analyse au développeur grâce à un parcours d'un ensemble d'événements.

Avec StAX, c'est donc l'application qui possède le contrôle sur le traitement du document ce qui rend plus intuitif le code à écrire pour traiter le document : l'application peut ignorer un élément, appliquer un filtre ou arrêter le traitement du document à tout moment.

Les fonctionnalités de StAX sont proches de celles de SAX. Cependant StAX propose des fonctionnalités supplémentaires :

- une mise en oeuvre des traitements sous une forme itérative qui la rend plus naturelle que la forme de type callback de SAX
- StAX permet l'écriture de documents
- StAX peut être plus efficace car il n'oblige pas à traiter tout le document

StAX est donc aussi efficace que SAX en proposant un modèle de mise en oeuvre plus facile et extensible. StAX pourrait remplacer SAX mais StAX est une API récente qui ne possède pas d'implémentation dans d'autres langages pour le moment. SAX est un standard de fait implémenté dans de nombreuses solutions de parsing dans différentes plates-formes et langages.

DOM est basé sur un arbre d'objets en mémoire qui représente l'ensemble des éléments d'un document XML. Ceci est très pratique pour permettre de se déplacer librement dans le document et de le parcourir à son gré d'autant que DOM supporte l'utilisation des expressions XPath.

DOM est la seule API qui permet de modifier le document. La contre-partie est que DOM consomme beaucoup de ressources et notamment de mémoire puisque tout le document est représenté par un arbre d'objets en mémoire : cela exclut de fait son utilisation pour des documents XML volumineux.

DOM est donc l'API la plus puissante puisqu'elle permet un parcours du document dans n'importe quel ordre et quel sens, de modifier le document (création, modification et suppression de noeuds dans le document) et de l'écrire.

DOM et StAX ont en commun de pouvoir écrire un document XML.

L'existence de trois API pour traiter un document XML entraîne logiquement des interrogations sur le choix de l'API à utiliser en fonction du besoin. Il n'existe pas de règles immuables concernant ces choix mais voici quelques cas d'utilisation particuliers :

- La transformation d'un document XML en un autre document XML : il est fréquent de devoir transformer un document XML en un autre pour modifier sa structure ou l'appauvrir. La solution la plus adaptée pour modifier la structure ou appauvrir le document semble être XSLT puisque c'est son rôle principal. StAX ou DOM peuvent être aussi utilisés notamment dans le cas d'enrichissement du document : ces deux API nécessitent l'écriture de code mais cela permet aussi un accès à toutes les API de Java.
- Data Binding : l'utilisation de plus en plus fréquente de XML nécessite de pouvoir mapper un objet à un document ou une portion de document XML et vice versa. Le plus simple est d'utiliser une API dédiée telle que JAXB mais il est aussi possible de réaliser ce traitement à la main. SAX ne peut être utilisé que pour mapper un document XML dans un objet (unmarshalling). StAX et DOM pouvant écrire un document, ces deux API peuvent être utilisés pour des opérations dans les deux sens.
- Un document XML comme source de données : les données à utiliser et éventuellement à mettre à jour sont stockées dans un document XML. Dans ce cas de figure, seul DOM peut répondre au besoin grâce à son support de XPath pour accéder directement à une donnée et sa possibilité de modifier le contenu du document pour réaliser des mises à jour.

StAX peut donc être utilisé dans de nombreux cas de traitements de documents XML.

55. JSON

Chapitre 55

Niveau : 🟩🟩⬜⬜⬜⬜ Elémentaire

JSON est un format de données, basé sur du texte. JSON est l'acronyme de JavaScript Object Notation car c'est un dérivé de la représentation littérale d'un objet en JavaScript définit par l'ECMAScript Programming Language Standard.

Le format JSON est spécifié dans la RFC 4627. Il permet de sérialiser une structure de données au format texte. JSON est largement utilisé pour stocker des données ou échanger des données notamment sur Internet mais aussi entre les couches IHM et applicative/service d'une application.

JSON connait un fort engouement car il possède quelques points forts :

- standard ouvert
- syntaxe simple et compacte
- facile à parser et à écrire
- format offrant une structuration des données compacte

La syntaxe de JSON est très simple ce qui explique une partie de son succès. Elle ne définit que deux types de structure : un objet qui est un ensemble de paires clé/valeur et un tableau.

JSON définit 6 types de données : string, number, object, array, true, false et null.

Exemple :

```
{ "nom": "Dupont",
  "prenom": "Jean",
  "ville": "Paris",
  "pays": "France",
  "telephone": [ { "mobile": "0612345678" },
  { "fax": "0312345678" } ]
}
```

JSON est couramment utilisé pour échanger (sérialiser/désérialiser) des données entre applications ou modules d'une application en utilisant le réseau. Ces applications peuvent être développées avec différents langages et exécutées sur différentes plates-formes.

JSON est utilisable avec de nombreux langages notamment Java grâce à plusieurs API open source.

Ce chapitre contient plusieurs sections :

- ♦ La syntaxe de JSON
- ♦ L'utilisation de JSON
- ♦ JSON ou XML

55.1. La syntaxe de JSON

JSON ne définit que deux structures de données :

- objet (object) est composé de paires nom/valeur
- tableau (array) est une liste de valeurs

JSON supporte plusieurs types de données :

- Numérique : nombre entier ou flottant
- Chaîne de caractères : ensemble de caractères Unicode (sauf une double quote et un antislash) entouré par des doubles guillemets
- Booléen : true ou false
- Tableau : un ensemble ordonné de valeurs entouré par des crochets. Chaque valeur est séparée par un caractère virgule. Les types des valeurs d'un tableau peuvent être différents
- Object : est composé de paires clé/valeur, chacune étant séparé par une virgule, entourées par des accolades. Une clé est obligatoirement une chaîne de caractères. Une valeur peut être : littérale (chaîne de caractères, nombre, true, false, null), un objet ou un tableau. Une clé est séparée de sa valeur par un caractère deux points
- La valeur null

Les valeurs d'un objet ou d'un tableau peuvent être d'un de ces types.

Dans une chaîne de caractères, le caractère d'échappement est le caractère antislash qui permet notamment de représenter dans la chaîne de caractères :

- \" : une double quote
- \\ : un antislash
- \/ : un slash
- \b : un backspace
- \f : un formfeed
- \n : une nouvelle ligne
- \r : un retour chariot
- \t : une tabulation
- \unnnn : le caractère Unicode dont le numéro est nnnn

Exemple :

```
{ "prenom":"Jean-Michel" ,
"ville":"Paris" }
```

Le séparateur d'un nombre flottant est le caractère point.

JSON utilise deux types de structures :

- objet : un objet est défini entre accolades. Il est composé de paires nom/valeur, chacune étant séparée par une virgule. Dans une paire, le nom est séparé de sa valeur par un caractère deux points. Le nom est obligatoirement une chaîne de caractères

Exemple :

```
{ "prenom":"Jean-Michel" ,
"ville":"Paris" }
```

- tableau : est un ensemble ordonné de valeurs. Un tableau est défini entre crochet. Chaque valeur est séparée par une virgule

Exemple :

```
[{ "prenom":"prenom1", "nom":"nom1" },
 { "prenom":"prenom2" , "nom":"nom2" },
 { "prenom":"prenom3" , "nom":"nom3" }]
```

Une valeur peut être d'un des types supportés par JSON : ces structures peuvent donc être imbriquées.

Exemple :

```
{ "groupe":
  [{ "prenom":"prenom1" , "nom":"nom1" },
   { "prenom":"prenom2" , "nom":"nom2" },
   { "prenom":"prenom3" , "nom":"nom3" }]
}
```

Le tableau ci-dessous résume la syntaxe de JSON.

objet **{}** **{** *membres* **}** *membres* *paire* *paire* **,** *membres* *paire* *string* **:** *valeur* *tableau* **[]** **[** *elements* **]** *elements* *value* *value* **,** *elements* *value* *string* *nombre* *objet* *tableau* **true** **false** **null**	*string* **""** **"** *chars* **"** *chars* *char* *char chars* *char* *caractère Unicode sauf* **" et \\** \\" \\\\ \\/ **\\b** **\\f** **\\n** **\\r** **\\t** **\\u** *nnnn*	*nombre* *int* *int frac* *int exp* *int frac exp* *int* *digit* *digit1-9 digits* **-** *digit* **-** *digit1-9 digits* *frac* **.** *digits* *exp* *e digits* *digits* *digit* *digit digits* *e* **e** **e+** **e-** **E** **E+** **E-**

Les contenus binaires peuvent être intégrés dans une représentation JSON en les encodant en Base 64.

55.2. L'utilisation de JSON

JSON est essentiellement utilisé pour échanger ou stocker des données structurées.

Le type MIME d'un document JSON est application/json

JSON est fréquemment utilisé dans les requêtes AJAX car son exploitation par JavaScript est plus simple et plus rapide que XML.

JSON est indépendant de tout langage : de nombreuses bibliothèques existent pour de nombreux langages dont Java.

Il existe plusieurs API pour manipuler des documents au format JSON en Java :

json.org	Jackson JSON Processor	google-gson
Jettison	Boon	Stringtree
SOJO	Json-lib	json-taglib
XStream	Flexjson	JON tools
Argo	jsonij	fastjson
mjson	jjson	json-simple
json-io	FOSS Nova JSON	Corn CONVERTER

55.3. JSON ou XML

JSON possède plusieurs points communs avec XML :

- le format est textuel
- la structure est hiérarchique
- lisible facilement par un humain

Contrairement à XML, JSON ne propose pas de standard pour décrire la structure des données, ce qui pourrait permettre de valider un document JSON.

Moins verbeux que XML, l'utilisation de JSON tent à se répendre notamment pour une utilisation combinée avec REST pour proposer des API facile à utiliser.

JSON est plus compact et facile à parser qu'un document XML équivalent. Un document JSON est cependant légèrement moins lisible.

	Avantages	Inconvénient
XML	XML est extensible XML permet de définir une structure particulière pour chaque document avec une unicité des tags en utilisant des namespaces Possibilité de définir une description du document (DTD ou schéma) permettant de le valider Utilisation possible d'attributs dans les tags	La verbosité liée à l'utilisation de nom de tags et de leur redondance fréquente dans les tags Le coût de parsing des documents qui augmente avec la taille des documents
JSON	La vitesse de traitement La simplicité de mise en oeuvre Plus compacte Exploitable directement avec JavaScript	Support d'un nombre limité de type de données Ne prend pas non plus en charge les métadonnées qui pourraint fournir des informations supplémentaires sur les données Difficultés de validation : JSON ne dispose pas de mécanisme intégré pour vérifier la structure, le format ou le contenu des données Faible sécurisation en tant que format de données pour les applications Web. JSON est vulnérable à

		diverses attaques, telles que le cross-site scripting (XSS), la falsification de requêtes intersites (CSRF) Pas de support des commentaires Encoding uniquement en UTF-8

Remarque : XML et JSON ne conviennent pas pour stocker directement des données binaires de taille importante.

L'exemple ci-dessous est la définition d'un menu : il s'agit d'un objet composé de membres qui sont un attribut et un tableau lequel contient d'autres objets, les éléments du menu.

Exemple :

```
{
  "menu": "Fichier",
  "commandes": [
      {
          "title": "Nouveau",
          "action":"NouveauDoc"
      },
      {
          "title": "Ouvrir",
          "action": "OuvrirDoc"
      },
      {
          "title": "Fermer",
          "action": "FermerDoc"
      }
   ]
}
```

L'équivalent XML est le document ci-dessous :

Exemple :

```
<?xml version="1.0" ?>
<root>
  <menu>Fichier</menu>
  <commands>
     <item>
         <title>Nouveau</value>
         <action>NouveauDoc</action>
     </item>
     <item>
         <title>Ouvrir</value>
         <action>OuvrirDoc</action>
     </item>
     <item>
         <title>Fermer</value>
         <action>FermerDoc</action>
     </item>
  </commands>
</root>
```

Comme pour XML, il existe deux grands modèles de programmation pour générer ou parser des données aux formats Json :

- le modèle objet (object model) : il repose sur la création d'un graphe d'objets qui encapsule en mémoire la représentation des données JSON. Cette approche est très flexible car elle permet de naviguer et de modifier le modèle mais elle requiert de représenter tous les éléments en mémoire
- le modèle en stream (streaming model) : avec cette approche, le parser émet des événements qu'il faut intercepter et traiter. La génération d'une représentation JSON se fait en ajoutant les éléments un par un.

56. Gson

Chapitre 56

Niveau : 🟩🟩🟨⬜⬜⬜ Intermédiaire

Gson est une bibliothèque open source développée par Google pour convertir un objet Java dans sa représentation JSON et vice versa.

Le site officiel est à l'url : https://github.com/google/gson

Pour utiliser Gson dans une application, il faut ajouter le fichier gson-version.jar au classpath.

Pour utiliser Gson dans un projet Maven, il faut ajouter la dépendance :

Exemple :

```
<dependency>
  <groupId>com.google.code.gson</groupId>
  <artifactId>gson</artifactId>
  <version>2.2.4</version>
</dependency>
```

Gson peut lire ou écrire des documents JSON en utilisant deux modèles de programmation :

- object model : ce modèle est similaire à DOM pour XML
- streaming : ce modèle est similaire au parser de type pull

Plusieurs versions ont été diffusées :

- 1.1 : juillet 2008
- 1.2 : août 2008, amélioration des performances, support pour des documents de plus de 15Mo
- 1.2.3 : octobre 2008, support des maps, Gson est thread-safe
- 1.3 : avril 2009, la méthode Gson.toJson() attend un paramètre de type Appendable, support de la désérialisation des valeurs double NaN et infinity, nouvelle API Parser, ajout de la méthode generateNonExecutableJson() à la classe GsonBuilder, support de FieldNamingStrategy
- 1.4 : octobre 2009, support des stratégies d'exclusions, naming policy LOWER_CASE_WITH_DASHES
- 1.5 : août 2010, naming policy UPPER_CAMEL_CASE_WITH_SPACES, amélioration des performances
- 1.6 : octobre 2010, nouveau parser qui améliore les performances, API Streaming
- 1.7 janvier 2011, amélioration des performances,
- 2.0 : novembre 2011, plus rapide, null est sérialisé en "null",
- 2.1 : décembre 2011
- 2.2 : mai 2012
- 2.2.2 : juillet 2012
- 2.2.3 : avril 2013
- 2.2.4 : mai 2013
- 2.3 : oaût 2014
- 2.3.1 : novembre 2014
- 2.4 : octobre 2015

- 2.5 : novembre 2015
- 2.6 : février 2016
- 2.7 : juin 2016
- 2.8.0 : octobre 2016
- 2.8.1 : mai 2017
- 2.8.2 : septembre 2017
- 2.8.3 : avril 2018
- 2.8.4 : mai 2018
- 2.8.5 : mai 2018

Ce chapitre contient plusieurs sections :

- ♦ La classe Gson
- ♦ La sérialisation
- ♦ La désérialisation
- ♦ La personnalisation de la sérialisation/désérialisation
- ♦ Les annotations de Gson
- ♦ L'API Streaming
- ♦ Mixer l'utilisation du model objets et de l'API Streaming
- ♦ Les concepts avancés

56.1. La classe Gson

La classe Gson est la classe principale de l'API Object Model.

Une instance de la classe Gson peut être obtenue :

- en invoquant son constructeur pour obtenir une instance avec la configuration par défaut qui permet de facilement répondre aux cas simples
- en utilisant une fabrique du type GsonBuilder qui permet de configurer précisément l'instance créée

Une instance de type Gson ne maintient aucun état lors de l'invocation de ses méthodes.

Ces deux méthodes principales possèdent plusieurs surcharges :

- fromJson() : permet de créer un objet qui encapsule les données d'un document JSON. Ces surcharges attendent deux paramètres qui sont le document JSON (JsonElement, JsonReader, Reader, String) et le type d'objet à créer (Class ou Type)
- toJson() : permet de créer une représentation JSON d'un objet (Objet) ou d'un élément (JsonElement)

56.2. La sérialisation

La sérialisation de données dans leur représentation JSON se fait en utilisant une des surcharges de la méthode toJson() de la classe Gson.

L'objet passé en paramètre de la méthode toJson() peut être une valeur d'un type commun (type primitif, wrapper, chaîne de caractères, tableaux).

Exemple :

```
package fr.jmdoudoux.dej.gson;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class TestSerialisation {

  public static void main(final String[] args) {
    final GsonBuilder builder = new GsonBuilder();
    final Gson gson = builder.create();

    System.out.println("1 -> " + gson.toJson(1));
    System.out.println("abcde -> " + gson.toJson("abcde"));
    System.out.println("Long(10) -> " + gson.toJson(new Long(10)));
    final int[] values = { 1, 2, 3 };
    System.out.println("{1,2,3} -> " + gson.toJson(values));
  }
}
```

Résultat :

```
1 -> 1
abcde -> "abcd"
Long(10) -> 10
{1,2,3} -> [1,2,3]
```

Il est possible de passer un tableau en paramètre de la méthode toJson().

Exemple :

```
package fr.jmdoudoux.dej.gson;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class TestSerialisation {

  public static void main(final String[] args) {
    final GsonBuilder builder = new GsonBuilder();
    final Gson gson = builder.create();

    final int[] entiers = { 1, 2, 3, 4 };
    System.out.println("entiers -> " + gson.toJson(entiers));
    final int[][] valeurs = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
    System.out.println("valeurs -> " + gson.toJson(valeurs));
    final String[] chaines = { "ab", "cd", "ef" };
    System.out.println("chaines -> " + gson.toJson(chaines));
  }
}
```

Résultat :

```
entiers -> [1,2,3,4]
valeurs -> [[1,2,3],[4,5,6],[7,8,9]]
chaines -> ["ab","cd","ef"]
```

Il est possible de passer une instance d'une classe qui encapsule les données à sérialiser en paramètre de la méthode toJson().

Exemple :

```
package fr.jmdoudoux.dej.gson;

import com.google.gson.annotations.SerializedName;

public class Coordonnees {

  private final int abscisse;
  private final int ordonnee;

  public Coordonnees(final int abscisse, final int ordonnee) {
    super();
    this.abscisse = abscisse;
    this.ordonnee = ordonnee;
  }

  public int getAbscisse() {
    return this.abscisse;
  }

  public int getOrdonnee() {
    return this.ordonnee;
  }

  @Override
  public String toString() {
    return "Coordonnees [abscisse=" + this.abscisse + ", ordonnee=" + this.ordonnee + "]";
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.gson;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class TestSerialisation {

  public static void main(final String[] args) {
    final GsonBuilder builder = new GsonBuilder();
    final Gson gson = builder.create();

    final Coordonnees coordonnees = new Coordonnees(120, 450);

    final String json = gson.toJson(coordonnees);
    System.out.println("Resultat = " + json);
  }
}
```

Résultat :

```
Resultat = {"abscisse":120,"ordonnee":450}
```

Par défaut, tous les champs de la classe et des classes mères sont utilisés lors de la sérialisation même ceux déclarés private.

Tous les champs marqués avec le mot clé transient sont ignorés lors de la sérialisation. Les champs null sont aussi par défaut ignorés durant la sérialisation.

Les champs d'une classe interne qui font référence à la classe englobante sont ignorés (champ synthetic).

Attention : il ne faut pas sérialiser un objet qui contient une référence circulaire.

Il est possible de fournir une collection en paramètre de la méthode toJson().

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.util.ArrayList;
import java.util.List;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class TestSerialisation {

  public static void main(final String[] args) {
    final GsonBuilder builder = new GsonBuilder();
    final Gson gson = builder.create();

    final List<Integer> listeEntier = new ArrayList<Integer>();
    listeEntier.add(1);
    listeEntier.add(2);
    listeEntier.add(3);
    System.out.println("liste -> " + gson.toJson(listeEntier));

    final List listeObjet = new ArrayList();
    listeObjet.add("chaine");
    listeObjet.add(123);
    final Coordonnees coordonnees = new Coordonnees(120, 450);
    listeObjet.add(coordonnees);
    final String json = gson.toJson(listeObjet);
    System.out.println("Resultat = " + json);
  }
}
```

Résultat :

```
liste -> [1,2,3]
Resultat = ["chaine",123,{"abscisse":120,"ordonnee":450}]
```

Il est possible de sérialiser une collection qui contient des objets de types différents mais il ne sera pas possible de les désérialiser car rien ne permet de préciser le type de chacune des valeurs.

Les collections de type Map sont sérialisées de manière particulière.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.util.HashMap;
import java.util.Map;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class TestSerialisation {

  public static void main(final String[] args) {
    final GsonBuilder builder = new GsonBuilder();
    final Gson gson = builder.create();
    final Map<Integer, String> valeurs = new HashMap<Integer, String>();
    valeurs.put(1, "Valeur1");
    valeurs.put(2, "Valeur2");
    valeurs.put(3, "Valeur3");
    final String json = gson.toJson(valeurs);
    System.out.println("Resultat = " + json);
  }
}
```

Résultat :

```
Resultat = {"2":"Valeur2","1":"Valeur1","3":"Valeur3"}
```

56.3. La désérialisation

La désérialisation de données à partir de leur représentation JSON se fait en utilisant une des surcharges de la méthode fromJson() de la classe Gson.

La méthode fromJson() possède plusieurs surcharges :

Méthode	Rôle
String toJson(JsonElement jsonElement)	Convertir un JsonElement dans sa représentation JSON
void toJson(JsonElement jsonElement, Appendable writer)	Ecrire la représentation JSON du JsonElement dans l'instance de type Appendable fournie en paramètre

void toJson(JsonElement jsonElement, JsonWriter writer)	Ecrire la représentation JSON du JsonElement dans l'instance de type Writer fournie en paramètre
String toJson(Object src)	Convertir un objet dans sa représentation JSON
void toJson(Object src, Appendable writer)	Ecrire la représentation JSON de l'objet dans l'instance de type Appendable fournie en paramètre
String toJson(Object src, Type typeOfSrc)	Convertir un objet typé avec un generic dans sa représentation JSON
void toJson(Object src, Type typeOfSrc, Appendable writer)	Ecrire la représentation JSON de l'objet typé avec un generic dans l'instance de type Appendable fournie en paramètre
void toJson(Object src, Type typeOfSrc, JsonWriter writer)	Ecrire la représentation JSON de l'objet typé avec un generic dans l'instance de type JsonWriter fournie en paramètre

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.util.Arrays;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class TestDeserialisation {

  public static void main(final String[] args) {
    final Gson gson = new GsonBuilder().setPrettyPrinting().create();

    final int unInt = gson.fromJson("1", int.class);
    System.out.println(unInt);
    final Integer unInteger = gson.fromJson("1", Integer.class);
    System.out.println(unInteger);
    final Long unLong = gson.fromJson("1", Long.class);
    System.out.println(unLong);
    final Boolean booleen = gson.fromJson("false", Boolean.class);
    System.out.println(booleen);
    final String chaine = gson.fromJson("\"abc\"", String.class);
    System.out.println(chaine);
    final String[] chaine2 = gson.fromJson("[\"abc\"]", String[].class);
    System.out.println(Arrays.deepToString(chaine2));
  }
}
```

Résultat :

```
1
1
1
false
abc
[abc]
```

Il est possible désérialiser la représentation JSON d'un objet en la passant en paramètre de la méthode fromJson() avec le type de la classe à produire.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class TestDeserialisation {

  public static void main(final String[] args) {
    final Gson gson = new GsonBuilder().create();
    final String json = "{\"id\":1,\"nom\":\"nom1\",\"prenom\":\"prenom1\"}";
    final Personne personne = gson.fromJson(json, Personne.class);
    System.out.println(personne);
  }
}
```

Résultat :

```
Personne [id=1, nom=nom1, prenom=prenom1]
```

Lors de la désérialisation, les valeurs qui ne sont pas contenues dans le document JSON sont initialisées avec null.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class TestDeserialisation {

  public static void main(final String[] args) {
    final Gson gson = new GsonBuilder().create();

    final String json = "{\"id\":1,\"nom\":\"nom1\"}";
    final Personne personne = gson.fromJson(json, Personne.class);
    System.out.println(personne);
  }
}
```

Résultat :

```
Personne [id=1, nom=nom1, prenom=null]
```

La classe doit posséder un constructeur par défaut. Le nom de la classe n'a pas d'importance, par contre la casse du nom des champs doit correspondre aux clés dans la représentation JSON. Gson utilise l'introspection pour alimenter les champs, donc il n'est pas obligatoire de disposer de setter dans la classe.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class TestDeserialisation {

  public static void main(final String[] args) {
    final Gson gson = new GsonBuilder().setPrettyPrinting().create();

    final String json = "{\"ID\":1,\"NOM\":\"nom1\"}";
    final Personne personne = gson.fromJson(json, Personne.class);
    System.out.println(personne);
  }
}
```

Résultat :

```
Personne [id=0, nom=null, prenom=null]
```

Il est possible de désérialiser des données stockées dans un tableau.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.util.Arrays;

public class Groupe {

  private final String     nom;
  private final Personne[] personnes;
  private int              nbPersonnes = 0;

  public Groupe(final String nom) {
    super();
    this.nom = nom;
    this.personnes = new Personne[10];
  }

  public String getNom() {
    return this.nom;
  }

  public Personne[] getPersonnes() {
    return this.personnes;
  }

  public void ajouter(final Personne personne) {
    if (this.nbPersonnes < 10) {
      this.personnes[this.nbPersonnes] = personne;
      this.nbPersonnes++;
    }
  }

  @Override
  public String toString() {
    return "Groupe [nom=" + this.nom + ", personnes="
        + Arrays.toString(this.personnes) + ", nbPersonnes="
        + this.nbPersonnes + "]";
  }

}
```

Exemple :

```
package fr.jmdoudoux.dej.gson;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class TestDeserialisation {

  public static void main(final String[] args) {
    final Gson gson = new GsonBuilder().create();

    final String json = "{\"nom\":\"Groupe1\",\"personnes\":"
      +"[{\"id\":1,\"nom\":\"nom1\",\"prenom\":\"prenom1\"},"
      +"{\"id\":2,\"nom\":\"nom2\",\"prenom\":\"prenom2\"},"
      +"null,null,null,null,null,null,null,null],\"nbPersonnes\":2}";
    final Groupe groupe = gson.fromJson(json, Groupe.class);

    System.out.println(groupe);
  }
}
```

Résultat :

```
Groupe [nom=Groupe1, personnes=[Personne
[id=1, nom=nom1, prenom=prenom1], Personne [id=2, nom=nom2, prenom=prenom2],
null, null, null, null, null, null, null, null], nbPersonnes=2]
```

Il est aussi possible de désérialiser des collections. Gson permet de sérialiser une collection contenant des objets de type divers mais ne permet pas de désérialiser une telle collection car il n'est pas possible de préciser le type de chacun des éléments. Pour être désérialisée, une collection doit contenir des objets d'un même type précisé sous la forme d'un generic.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.util.ArrayList;
import java.util.List;

public class Groupe {
  private final String         nom;
  private final List<Personne> personnes = new ArrayList();

  public Groupe(final String nom) {
    super();
    this.nom = nom;
  }

  public String getNom() {
    return this.nom;
  }

  public List<Personne> getPersonnes() {
    return this.personnes;
  }

  public void ajouter(final Personne personne) {
    this.personnes.add(personne);
  }

  @Override
  public String toString() {
    return "Groupe [nom=" + this.nom + ", personnes=" + this.personnes  + "]";
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.gson;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class TestDeserialisation {
  public static void main(final String[] args) {
    final Gson gson = new GsonBuilder().create();
    final String json = "{\"nom\":\"Groupe1\",\"personnes\":[{\"id\":1,\"nom\":\"nom1\","
      + "\"prenom\":\"prenom1\"},{\"id\":2,\"nom\":\"nom2\",\"prenom\":\"prenom2\"}]}";
    final Groupe groupe = gson.fromJson(json, Groupe.class);
    System.out.println(groupe);
  }
}
```

Résultat :

```
Groupe [nom=Groupe1, personnes=[Personne [id=1, nom=nom1,
prenom=prenom1], Personne [id=2, nom=nom2, prenom=prenom2]]]
```

56.4. La personnalisation de la sérialisation/désérialisation

Gson propose plusieurs fonctionnalités pour personnaliser les opérations de sérialisation et de désérialisation.

56.4.1. La classe GsonBuilder

La classe GsonBuilder est une fabrique qui permet de créer une instance de type Gson qui soit configurable.

Cette classe met en oeuvre le motif de conception Builder : chaque méthode qui permet de configurer une fonctionnalité de l'instance à créer renvoie l'instance du builder elle-même, ce qui permet de chaîner les invocations. La méthode create() permet d'obtenir l'instance.

La classe GsonBuilder propose plusieurs méthodes qui permettent de configurer les instances qui seront créées :

Méthode	Rôle
GsonBuilder addDeserializationExclusionStrategy(ExclusionStrategy strategy)	Ajouter une stratégie d'exclusion des éléments lors de la désérialisation
GsonBuilder addSerializationExclusionStrategy(ExclusionStrategy strategy)	Ajouter une stratégie d'exclusion des éléments lors de la sérialisation
Gson create()	Créer une instance avec la configuration courante
GsonBuilder disableHtmlEscaping()	Désactiver l'échappement des caractères HTML (activé par défaut)
GsonBuilder disableInnerClassSerialization()	Désactiver la sérialisation des classes internes
GsonBuilder enableComplexMapKeySerialization()	
GsonBuilder excludeFieldsWithModifiers(int... modifiers)	Exclure les champs qui possèdent le ou les modificateurs fournis en paramètres
GsonBuilder excludeFieldsWithoutExposeAnnotation()	Exclure tous les champs qui ne sont pas annotés avec @Expose et prendre en compte ceux qui le sont selon les paramètres de l'annotation
GsonBuilder generateNonExecutableJson()	Rendre le document JSON généré non exécutable par JavaScript en le préfixant par un texte
GsonBuilder registerTypeAdapter(Type type, Object typeAdapter)	
GsonBuilder registerTypeAdapterFactory(TypeAdapterFactory factory)	
GsonBuilder registerTypeHierarchyAdapter(Class<?> baseType, Object typeAdapter)	
GsonBuilder serializeNulls()	Demander à Gson de sérialiser les champs null
GsonBuilder serializeSpecialFloatingPointValues()	Demander à Gson de sérialiser les valeurs spéciales pour les champs de type double (NaN, Infinity, -Infinity).
GsonBuilder setDateFormat(int style)	Préciser le style de format des dates
GsonBuilder setDateFormat(int dateStyle, int timeStyle)	Préciser le style de format des dates et des heures
GsonBuilder setDateFormat(String pattern)	Préciser le format des dates
GsonBuilder setExclusionStrategies(ExclusionStrategy... strategies)	Configurer Gson pour appliquer des stratégies d'exclusion pendant les sérialisations et les désérialisations
GsonBuilder setFieldNamingPolicy(FieldNamingPolicy namingConvention)	Configurer Gson pour appliquer une politique de nommage des champs
GsonBuilder setFieldNamingStrategy(FieldNamingStrategy fieldNamingStrategy)	Configurer Gson pour appliquer une stratégie de nommage de champs lors des sérialisations et des déserializations

GsonBuilder setLongSerializationPolicy(LongSerializationPolicy serializationPolicy)	Configurer Gson pour utiliser une stratégie de sérialisation des données de type long ou Long
GsonBuilder setPrettyPrinting()	Configurer Gson pour formater les documents générés
GsonBuilder setVersion(double ignoreVersionsAfter)	Préciser le numéro de version courant et ainsi activer l'utilisation des annotations @Since et @Until

L'ordre d'invocation des méthodes de configuration n'a pas d'importance.

Exemple :

```
 Gson gson = new GsonBuilder()
     .setFieldNamingPolicy(FieldNamingPolicy.UPPER_CAMEL_CASE)
     .setDateFormat(DateFormat.LONG)
     .serializeNulls()
     .setPrettyPrinting()
     .setVersion(1.2)
     .create();
```

56.4.2. L'utilisation de Serializer et TypeAdapter

La sérialisation et désérialisation par défaut ne répond pas toujours aux besoins. Gson permet d'enregistrer et d'utiliser des classes qui permettront de réaliser une sérialisation ou une désérialisation personnalisée.

Gson définit deux interfaces qu'il faut implémenter selon les besoins :

- JsonSerializer<T>
- JsonDeserializer<T>

Les Serializer et les Deserializer doivent être enregistrés auprès d'une instance de type GsonBuilder en invoquant la méthode registerTypeAdapter(). Elle attend en paramètre la classe de l'instance à sérialiser et une instance du type de l'implémentation du Serializer.

Si les Serializer ne proposent pas de constructeur par défaut, il est nécessaire de proposer une implémentation de l'interface InstanceCreator<T>. Celle-ci doit aussi être enregistrée dans l'instance de type GsonBuilder en invoquant la méthode registerTypeAdapter(). Elle ne définit qu'une méthode createInstance() qui attend en paramètre le type de l'instance à créer.

Les exemples de cette section vont développer une personnalisation de la sérialisation/désérialisation pour une classe qui encapsule une chaîne de caractères.

Exemple :

```
package fr.jmdoudoux.dej.gson;

public class MaChaine {

  private String valeur;

  public String getValeur() {
    return this.valeur;
  }

  public void setValeur(final String valeur) {
    this.valeur = valeur;
  }

  @Override
  public String toString() {
    return "MaChaine [valeur=" + this.valeur + "]";
  }
}
```

56.4.2.1. L'interface JsonSerializer

L'interface JsonSerializer<T> définit les fonctionnalités d'un Serialiser personnalisé. Le type T permet d'indiquer le type de la classe à sérialiser.

Un Serializer permet de définir précisément comment une classe est sérialisée : ceci est particulièrement utile lorsque le mécanisme de sérialisation par défaut de Gson ne répond pas au besoin.

L'interface JsonSerializer ne définit qu'une seule méthode :

Méthode	Rôle
JsonElement serialize(T src, Type typeOfSrc, JsonSerializationContext context)	Méthode invoquée par Gson lorsqu'il faut sérialiser un objet de type T

Il faut définir une classe qui implémente l'interface JsonSerializer et redéfinir la méthode serialize(). Celle-ci renvoie une instance de type JsonElement qui encapsule l'arborescence des éléments du document JSON.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.lang.reflect.Type;

import org.apache.commons.codec.binary.Base64;

import com.google.gson.JsonElement;
import com.google.gson.JsonNull;
import com.google.gson.JsonPrimitive;
import com.google.gson.JsonSerializationContext;
import com.google.gson.JsonSerializer;

public class MaChaineSerializer implements JsonSerializer<MaChaine> {
  @Override
  public JsonElement serialize(final MaChaine maChaine,
    final Type typeOfSrc, final JsonSerializationContext context) {
    JsonElement resultat = null;

    if (maChaine == null) {
      resultat = JsonNull.INSTANCE;
    } else {
      if (maChaine.getValeur() == null) {
        resultat = JsonNull.INSTANCE;
      } else {
        resultat = new JsonPrimitive(new String(
          Base64.encodeBase64(maChaine.getValeur().getBytes())));
      }
    }
    return resultat;
  }
}
```

Il est nécessaire d'enregistrer un Serializer en invoquant la méthode registerTypeAdapter() de la classe GsonBuilder

Exemple :

```
package fr.jmdoudoux.dej.gson;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class TestSerializer {
  public static void main(final String[] args) {
    MaChaine maChaine = new MaChaine();
    maChaine.setValeur("ma valeur");

    Gson gson = new GsonBuilder().create();
    String json = gson.toJson(maChaine);
    System.out.println("Serialisation sans serializer = " + json);

    gson = new GsonBuilder().registerTypeAdapter(MaChaine.class,
      new MaChaineSerializer()).create();
    json = gson.toJson(maChaine);
    System.out.println("Serialisation avec serializer = " + json);

    maChaine = new MaChaine();
    json = gson.toJson(maChaine);
    System.out.println("Serialisation null avec serializer = " + json);
  }
}
```

Résultat :

```
Serialisation sans serializer = {"valeur":"ma valeur"}
Serialisation avec serializer = "bWEgdmFsZXVy"
Serialisation null avec serializer = null
```

56.4.2.2. L'interface JsonDeserializer

L'interface JsonDeserializer<T> définit les fonctionnalités d'un Deserialiser personnalisé. Le type T permet d'indiquer le type de la classe à désérialiser.

Un Deserializer permet de définir précisément comment une classe est désérialisée : ceci est particulièrement utile lorsque le mécanisme de désérialisation par défaut de Gson ne répond pas au besoin.

L'interface JsonDeserializer ne définit qu'une seule méthode :

Méthode	Rôle
T deserialize(JsonElement json, Type typeOfT, JsonDeserializationContext context)	Méthode invoquée par Gson lorsqu'il faut désérialiser un objet de type T

Il faut définir une classe qui implémente l'interface JsonDeserializer et redéfinir la méthode deserialize(). Celle-ci renvoie une instance de type T encapsulant le résultat de la désérialisation du JsonElement qui encapsule l'arborescence des éléments du fragment JSON.

Exemple :

```
package fr.jmdoudoux.dej.gson;

 import java.lang.reflect.Type;

 import org.apache.commons.codec.binary.Base64;

 import com.google.gson.JsonDeserializationContext;
 import com.google.gson.JsonDeserializer;
 import com.google.gson.JsonElement;
 import com.google.gson.JsonNull;
 import com.google.gson.JsonParseException;

 public class MaChaineDeserializer implements JsonDeserializer<MaChaine> {
   @Override
   public MaChaine deserialize(final JsonElement json, final Type typeOfT,
       final JsonDeserializationContext context) throws JsonParseException {
     final MaChaine resultat = new MaChaine();

     if (json != JsonNull.INSTANCE) {
       final String valeurBase64 = json.getAsJsonPrimitive().getAsString();
       final String valeur = new String(Base64.decodeBase64(valeurBase64));
       resultat.setValeur(valeur);
     }
     return resultat;
   }
 }
```

Il est nécessaire d'enregistrer un Deserializer en invoquant la méthode registerTypeAdapter() de la classe GsonBuilder

Exemple :

```
package fr.jmdoudoux.dej.gson;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class TestDeserializer {

  public static void main(final String[] args) {
    MaChaine maChaine = new MaChaine();
    maChaine.setValeur("ma valeur");

    Gson gson = new GsonBuilder().create();
    String json = gson.toJson(maChaine);
    System.out.println("Serialisation sans serializer = " + json);

    gson = new GsonBuilder()
        .registerTypeAdapter(MaChaine.class, new MaChaineSerializer())
        .registerTypeAdapter(MaChaine.class, new MaChaineDeserializer())
        .create();
    json = gson.toJson(maChaine);
    System.out.println("Serialisation avec serializer = " + json);

    final MaChaine maChaineDes = gson.fromJson(json, MaChaine.class);
    System.out.println("Deserialisation avec serializer = " + maChaineDes);

    maChaine = new MaChaine();
    json = gson.toJson(maChaine);
    System.out.println("Serialisation null avec serializer = " + json);
  }
}
```

Résultat :

```
Serialisation sans serializer = {"valeur":"ma valeur"}
Serialisation avec serializer = "bWEgdmFsZXVy"
Deserialisation avec serializer = MaChaine [valeur=ma valeur]
Serialisation null avec serializer = null
```

56.4.2.3. La classe TypeAdapter

Depuis la version 2.1 de Gson, il est préférable d'utiliser une instance de type com.google.gson.TypeAdapter car elle utilise l'API de streaming qui est plus efficace.

La classe TypeAdapter possède plusieurs méthodes :

Méthode	Rôle
T fromJson(Reader in)	Convertir le document Json passé en paramètre en un objet Java de type T
T fromJson(String json)	Convertir le document Json passé en paramètre en un objet Java de type T
T fromJsonTree(JsonElement jsonTree)	Convertir le document Json passé en paramètre en un objet Java de type T
TypeAdapter <T> nullSafe()	
Abstract T read(JsonReader in)	

	Lire un élément JSON (valeur, objet ou tableau) et le convertir en objet Java
String toJson(T value)	Convertir un objet en document JSON
void toJson(Writer out, T, value)	Convertir un objet en document JSON et l'envoyer dans le flux fourni en paramètre
JsonElement toJsonTree(T value)	Convertir un objet en un arbre d'éléments Json
abstract void write(JsonWriter out, T value)	Convertir un objet en un élément JSON (valeur, objet ou tableau)

Il faut définir une classe fille qui hérite de la classe TypeAdapter et redéfinir les méthodes abstraites read() et write().

Il est nécessaire d'enregistrer un adapter auprès de l'instance de type Gson en invoquant la méthode registerTypeAdapter de la classe GsonBuilder.

Dans l'exemple ci-dessous, l'adapter va sérialiser l'objet uniquement en encodant sa valeur en base 64 et vice versa lors de la désérialisation.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.io.IOException;

import org.apache.commons.codec.binary.Base64;

import com.google.gson.TypeAdapter;
import com.google.gson.stream.JsonReader;
import com.google.gson.stream.JsonToken;
import com.google.gson.stream.JsonWriter;

public class MaChaineAdapter extends TypeAdapter<MaChaine> {

  @Override
  public MaChaine read(final JsonReader reader) throws IOException {
    final MaChaine resultat = new MaChaine();
    if (reader.peek() == JsonToken.NULL) {
      reader.nextNull();
    } else {
      final String valeurBase64 = reader.nextString();
      final String valeur = new String(Base64.decodeBase64(valeurBase64));
      resultat.setValeur(valeur);
    }
    return resultat;
  }

  @Override
  public void write(final JsonWriter writer, final MaChaine maChaine) throws IOException {
    if (maChaine == null) {
      writer.nullValue();
    } else {
      if (maChaine.getValeur() == null) {
        writer.nullValue();
      } else {
        writer.value(new String(Base64.encodeBase64(maChaine.getValeur().getBytes())));
      }
    }
  }
}
```

Résultat :

```
Serialisation sans adapter = {"valeur":"ma valeur"}
Serialisation avec adapter = "bWEgdmFsZXVy"
Deserialisation avec adapter = MaChaine [valeur=ma valeur]
Serialisation null avec adapter = null
```

```
Deserialisation null avec adapter = MaChaine [valeur=null]
```

56.4.3. L'interface InstanceCreator

Lors d'une opération de désérialisation, Gson doit pouvoir créer une instance de chaque classe dont il va avoir besoin. Ces classes devraient avoir un constructeur sans argument public ou private.

Avant la version 1.7, lorsqu'une classe qui devait être instanciée par Gson ne possèdaient pas de constructeur par défaut, il fallait définir une classe de type InstanceCreator<T>

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.lang.reflect.Type;

import com.google.gson.InstanceCreator;

public class MonChampInstanceCreator implements InstanceCreator<MonChamp<?>> {
  @SuppressWarnings("rawtypes")
  @Override
  public MonChamp createInstance(final Type type) {
    return new MonChamp(123);
  }
}
```

Une instance de type InstanceCreator devait être enregistrée dans le GsonBuilder en utilisant la méthode MonChampInstanceCreator.

Exemple :

```
    final GsonBuilder builder = new GsonBuilder();
    builder.registerTypeAdapter(MonChamp.class, new MonChampInstanceCreator()).create();
```

56.4.4. Le formatage de la représentation JSON

Par défaut, la représentation JSON faite par Gson est compacte : elle ne contient aucun élément de formatage.

Il est possible de configurer l'instance de type Gson créée en utilisant un GsonBuilder pour qu'elle applique un formatage de la représentation JSON en invoquant la méthode setPrettyPrinting().

Exemple :

```
package fr.jmdoudoux.dej.gson;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class TestSerialisation {
  public static void main(final String[] args) {
    final Gson gson = new GsonBuilder().setPrettyPrinting().create();
    final Personne personne = new Personne(1, "nom1", "prenom1");
    final String json = gson.toJson(personne);
    System.out.println("Resultat = " + json);
  }
}
```

Résultat :

```
Resultat = {
  "id": 1,
  "nom": "nom1",
  "prenom": "prenom1"
}
```

Par défaut, la classe Gson utilise la classe JsonCompactFormatter. Elle utilise la classe JsonPrintFormatter si le formatage a été demandé dans sa configuration.

56.4.5. Le support des valeurs null

Par défaut, les objets null sont ignorés par Gson, ce qui permet d'avoir des représentations JSON plus compacte.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class TestSerialisation {
  public static void main(final String[] args) {
    final Gson gson = new GsonBuilder().create();
    final Personne personne = new Personne(1, "nom1", null);
    final String json = gson.toJson(personne);
    System.out.println("Resultat = " + json);
  }
}
```

Résultat :

```
Resultat =
{"id":1,"nom":"nom1"}
```

Il est possible de configurer l'instance de type Gson créée en utilisant un GsonBuilder pour qu'elle tienne compte des objets null en invoquant la méthode serializeNulls().

Exemple :

```
package fr.jmdoudoux.dej.gson;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class TestSerialisation {
  public static void main(final String[] args) {
    final Gson gson = new GsonBuilder().serializeNulls().create();
    final Personne personne = new Personne(1, "nom1", null);
    final String json = gson.toJson(personne);
    System.out.println("Resultat = " + json);
  }
}
```

Résultat :

```
Resultat =
{"id":1,"nom":"nom1","prenom":null}
```

56.4.6. L'exclusion de champs

Gson propose plusieurs solutions pour exclure certains champs des opérations de sérialisation. Si ces solutions ne sont pas suffisantes, il est toujours possible de créer ses propres Serializer et Deserializer.

56.4.6.1. L'exclusion de champs sur la base de modificateurs

Par défaut, les champs qui sont définis avec le mot clé transient ou static sont ignorés par Gson.

Il est possible de configurer l'instance de type Gson pour qu'elle ignore les champs possédant certains modificateurs en invoquant la méthode excludeFieldsWithModifier() de la classe GsonBuilder. L'invocation de la méthode excludeFieldsWithModifiers() permet de modifier le comportement par défaut.

Exemple :

```
Gson gson = gsonBuilder.excludeFieldsWithModifiers(Modifier.STATIC,
  Modifier.TRANSIENT, Modifier.VOLATILE).create();
```

Il est possible d'utiliser toutes les constantes définies dans la classe java.lang.reflect.Modifier en paramètre de la méthode excludeFieldsWithModifier().

56.4.6.2. Les stratégies d'exclusion personnalisées

Pour des besoins plus particuliers, il est possible de définir sa propre stratégie d'exclusion et de la faire appliquer à l'instance de type Gson. Cette stratégie permet de décider si une classe ou un champ doit être pris en compte lors des opérations de Gson.

Il faut définir une classe qui implémente l'interface ExclusionStrategy. Cette interface définit deux méthodes :

Méthode	Rôle
boolean shouldSkipClass(Class<?> clazz)	Renvoyer un booléen qui précise si la classe fournie en paramètre doit être ignorée
boolean shouldSkipField(FieldAttributes f)	Renvoyer un booléen qui précise si le champ fourni en paramètre doit être ignoré

L'exemple ci-dessous va définir une annotation qui est un simple marqueur et qui sera utilisée par une stratégie d'exclusion personnalisée pour ignorer les champs marqués avec celle-ci.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.FIELD, ElementType.TYPE })
public @interface ExclureDeGson {
}
```

Il faut utiliser l'annotation sur les champs ou les classes concernées

Exemple :

```
package fr.jmdoudoux.dej.gson;

public class MonBean {

  @ExclureDeGson
  private String champ10;
  private String champ11;
  private String champ12;

  public MonBean() {
    super();
  }

  // ...
}
```

La stratégie personnalisée va ignorer toutes les classes et les champs annotés avec l'annotation ExclureDeGson.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import com.google.gson.ExclusionStrategy;
import com.google.gson.FieldAttributes;

public class MonExclusionStrategy implements ExclusionStrategy {

  public MonExclusionStrategy() {
  }

  @Override
  public boolean shouldSkipClass(final Class<?> clazz) {
    return clazz.getAnnotation(ExclureDeGson.class) != null;
  }

  @Override
  public boolean shouldSkipField(final FieldAttributes f) {
    return f.getAnnotation(ExclureDeGson.class) != null;
  }
}
```

Pour demander à Gson d'utiliser la stratégie d'exclusion, il faut en passer une instance en paramètre de la méthode setExclusionsStrategies() de la classe GsonBuilder.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class TestSerialisation {

  public static void main(final String[] args) {
    final MonBean monBean = new MonBean();
    monBean.setChamp10("champ10");
    monBean.setChamp11("champ11");
    monBean.setChamp12("champ12");

    final GsonBuilder builder = new GsonBuilder();
    builder.setExclusionStrategies(new MonExclusionStrategy());
    final Gson gson = builder.create();
    final String json = gson.toJson(monBean);
    System.out.println("Resultat = " + json);
  }
}
```

Résultat :

```
Resultat = {"champ11":"champ11","champ12":"champ12"}
```

56.4.7. Le nommage des éléments

Il est possible de configurer l'instance de type Gson créée en utilisant un GsonBuilder pour qu'elle applique une règle de formatage lors du nommage d'un élément pour sa représentation JSON en invoquant sa méthode setFieldNamingPolicy().

L'énumération FieldNamingPolicy définit plusieurs valeurs qui correspondent à des conventions de nommage classiques :

Valeur	Rôle
IDENTITY	La politique de nommage est de reprendre le nom du champ tel quel
LOWER_CASE_WITH_DASHES	La politique de nommage est de tout mettre en minuscule, chaque mot séparé par un caractère tiret
LOWER_CASE_WITH_UNDERSCORES	La politique de nommage est de tout mettre en minuscule, chaque mot séparé par un caractère souligné
UPPER_CAMEL_CASE	La politique de nommage est de mettre la première lettre en majuscule
UPPER_CAMEL_CASE_WITH_SPACES	La politique de nommage est de mettre la première lettre de chaque mot en majuscule en séparant les mots par des espaces

Exemple :

```
package fr.jmdoudoux.dej.gson;

import com.google.gson.FieldNamingPolicy;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class TestSerialisation {

  public static void main(final String[] args) {
    final Personne personne = new Personne(1, "nom1", "prenom1");
    Gson gson = new GsonBuilder()
      .setFieldNamingPolicy(FieldNamingPolicy.UPPER_CAMEL_CASE).create();
    String json = gson.toJson(personne);
    System.out.println("Resultat = " + json);

    gson = new GsonBuilder()
      .setFieldNamingPolicy(FieldNamingPolicy.LOWER_CASE_WITH_DASHES).create();
    json = gson.toJson(personne);
    System.out.println("Resultat = " + json);
  }
}
```

Résultat :

```
Resultat = {"Id":1,"Nom":"nom1","Prenom":"prenom1"}
Resultat = {"id":1,"nom":"nom1","prenom":"prenom1"}
```

Il est aussi possible de définir sa propre stratégie de nommage en créant une classe qui implémente l'interface FieldNamingStrategy. Elle ne définit qu'une seule méthode :

Méthode	Rôle
String translateName (Field field)	Renvoyer le nom du champ JSON pour le champ de la classe fournie en paramètre

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.lang.reflect.Field;

import com.google.gson.FieldNamingStrategy;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class TestSerialisation {

  public static void main(final String[] args) {
    final MonBean monBean = new MonBean();
    monBean.setChamp10("champ10");
    monBean.setChamp11("champ11");
    monBean.setChamp12("champ12");

    final GsonBuilder builder = new GsonBuilder();
    builder.setFieldNamingStrategy(new FieldNamingStrategy() {
      @Override
      public String translateName(final Field field) {
        return "monapp_" + field.getName().toLowerCase();
      }
    });

    final Gson gson = builder.create();
    final String json = gson.toJson(monBean);
    System.out.println("Resultat = " + json);
  }
}
```

Résultat :

```
Resultat = {"monapp_champ10":"champ10","monapp_champ11":"champ11","monapp_champ12":"champ12"}
```

Cette fonctionnalité permet de personnaliser le nom des champs dans le document JSON. Ceci peut être particulièrement utile lorsque le document JSON contient des noms de champs qui ne correspondent pas à des noms de variables légales en Java. C'est la cas, par exemple, si le nom du champ contient un caractère '-'.

L'API propose aussi la possibilité de forcer le nom d'un élément en utilisant l'annotation @SerializedName (cette possibilité est détaillée dans une des sections suivantes).

56.5. Les annotations de Gson

Gson propose plusieurs annotations pour faciliter la configuration des opérations de sérialisation/désérialisation :

- @Expose : permet de préciser si un champ doit être utilisé ou non lors des opérations de sérialisation et de désérialisation
- @SerializedName : permet de préciser le nom du champ qui sera utilisé lors des opérations de sérialisation/désérialisation
- @Since : permet de définir à partir de quelle version le champ ou la classe doit être pris en compte
- @Until : permet de définir jusqu'à quelle version le champ ou la classe doit être pris en compte

56.5.1. La personnalisation forcée des noms des champs

L'annotation @SerializedName permet de préciser le nom du champ qui sera utilisé lors des opérations de sérialisation/désérialisation. Le nom est précisé comme valeur de l'annotation : il doit respecter les règles applicables à un nom de champs JSON.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class TestSerializedName {
  public static void main(final String[] args) {
    final GsonBuilder builder = new GsonBuilder();
    final Gson gson = builder.create();

    Coordonnees coordonnees = new Coordonnees(120, 450);

    final String json = gson.toJson(coordonnees);
    System.out.println("Resutlat = " + json);

    coordonnees = gson.fromJson("{\"abscisse\":120,\"ordonnee\":450}", Coordonnees.class);
    System.out.println(coordonnees);

    coordonnees = gson.fromJson("{\"x\":120,\"y\":450}", Coordonnees.class);
    System.out.println(coordonnees);
  }
}
```

Résultat :

```
Resutlat = {"x":120,"y":450}
Coordonnees [abscisse=0, ordonnee=0]
Coordonnees [abscisse=120, ordonnee=450]
```

La sérialisation et la désérialisation de l'objet utilisent les noms précisés par l'annotation.

L'utilisation de cette annotation ne nécessite aucune configuration particulière. Elle outrepasse n'importe quel FieldNamingPolicy qui pourrait être précisé dans la configuration.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import com.google.gson.FieldNamingPolicy;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class TestSerializedName {

  public static void main(final String[] args) {
    final GsonBuilder builder = new GsonBuilder();
    builder.setFieldNamingPolicy(FieldNamingPolicy.UPPER_CAMEL_CASE);
    final Gson gson = builder.create();

    final Coordonnees coordonnees = new Coordonnees(120, 450);

    final String json = gson.toJson(coordonnees);
    System.out.println("Resultat  = " + json);
  }
}
```

Résultat :

```
Resultat  = {"x":120,"y":450}
```

56.5.2. La désignation des champs à prendre en compte

En plus de l'utilisation du mot clé transient qui permet d'exclure complètement un champ lors des opérations de Gson et d'utiliser des classes filles de type JsonSerializer ou JsonDeserializer pour avoir un contrôle très fin sur ses opérations, Gson propose l'annotation @Expose. Elle ne s'utilise que sur des champs.

L'annotation @Expose possède deux attributs optionnels :

Attribut	Rôle
deserialize	Booléen qui précise si le champ doit être utilisé lors des opérations de désérialisation

serialize	Booléen qui précise si le champ doit être utilisé lors des opérations de sérialisation

Selon les valeurs fournies à ces deux attributs, le champ concerné sera pris en compte lors des opérations réalisées par Gson. Leur valeur par défaut est true.

Exemple d'utilisation	Sérialisation	Desérialisation
`private String monChamp;`	Non	Non

`@Expose`
`private String monChamp;`
	Oui	Oui

`@Expose(deserialize = false)`
`private String monChamp;`
	Oui	Non

`@Expose(serialize = false)`
`private String monChamp;`
	Non	Oui

`@Expose(serialize = false, deserialize = false)`
`private String monChamp;`
	Non	Non

Exemple :

```
package fr.jmdoudoux.dej.gson;

import com.google.gson.annotations.Expose;

public class MonObjet {

  private final String champ1;
  @Expose(serialize = false, deserialize = false)
  private final String champ2;
  @Expose(serialize = false)
  private final String champ3;
  @Expose(deserialize = false)
  private final String champ4;

  public MonObjet(final String champ1, final String champ2,
    final String champ3, final String champ4) {
    super();
    this.champ1 = champ1;
    this.champ2 = champ2;
    this.champ3 = champ3;
    this.champ4 = champ4;
  }

  public String getChamp1() {
    return this.champ1;
  }

  public String getChamp2() {
    return this.champ2;
  }

  public String getChamp3() {
    return this.champ3;
  }

  public String getChamp4() {
    return this.champ4;
  }

  @Override
  public String toString() {
    return "MonObjet [champ1=" + this.champ1 + ", champ2=" + this.champ2 + ",
      champ3=" + this.champ3 + ", champ4=" + this.champ4 + "]";
  }
}
```

Pour permettre la prise en compte de l'annotation @Expose, il est nécessaire de configurer l'instance de type Gson : pour cela, il faut invoquer la méthode excludeFieldsWithoutExposeAnnotation() de la classe GsonBuilder.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class TestSerialisation {

  public static void main(final String[] args) {
    final MonObjet monObjet = new MonObjet("valeur1", "valeur2", "valeur3", "valeur4");

    final GsonBuilder builder = new GsonBuilder();
    Gson gson = builder.create();
    String json = gson.toJson(monObjet);
    System.out.println("Resultat = " + json);

    builder.excludeFieldsWithoutExposeAnnotation();
    gson = builder.create();
    json = gson.toJson(monObjet);
    System.out.println("Resultat = " + json);
  }
}
```

Résultat :

```
Resultat = {"champ1":"valeur1","champ2":"valeur2","champ3":"valeur3","champ4":"valeur4"}
Resultat = {"champ4":"valeur4"}
```

L'utilisation de l'annotation @Expose(serialize = false, deserialize = false) est équivalente à déclarer le champ avec le modificateur transient.

56.5.3. La gestion des versions

GSON propose deux annotations utilisables sur des classes ou des champs qui permettent de déterminer ceux qui doivent être sérialisés et désérialisés en fonction d'un numéro de version :

- @Since : permet de définir à partir de quelle version le champ ou la classe doit être pris en compte
- @Until (à partir de la version 1.3) : permet de définir jusqu'à quelle version le champ ou la classe doit être pris en compte

Exemple :

```
package fr.jmdoudoux.dej.gson;

import com.google.gson.annotations.Since;
import com.google.gson.annotations.Until;

public class MonBean {

  @Until(1.1)
  private String champ10;
  private String champ11;
  @Since(1.2)
  private String champ12;

  public MonBean() {
    super();
  }

  public String getChamp10() {
    return this.champ10;
  }
  public void setChamp10(final String champ10) {
    this.champ10 = champ10;
  }

  public String getChamp11() {
    return this.champ11;
  }

  public void setChamp11(final String champ11) {
    this.champ11 = champ11;
  }

  public String getChamp12() {
    return this.champ12;
  }

  public void setChamp12(final String champ12) {
    this.champ12 = champ12;
  }
}
```

Le numéro de version courant doit être précisé à l'instance de type Gson en invoquant la méthode setVersion() de la classe GsonBuilder. Elle attend en paramètre la valeur de la version courante.

Si la version courante n'est pas précisée à l'instance de type Gson, alors les annotations @Since et @Until sont simplement ignorées.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class TestSerialisation {

  public static void main(final String[] args) {
    final MonBean monBean = new MonBean();
    monBean.setChamp10("champ10");
    monBean.setChamp11("champ11");
    monBean.setChamp12("champ12");

    final GsonBuilder builder = new GsonBuilder();
    Gson gson = builder.create();
    String json = gson.toJson(monBean);
    System.out.println("Resultat = " + json);

    builder.setVersion(1.0);
    gson = builder.create();
    json = gson.toJson(monBean);
    System.out.println("Resultat version 1.0 = " + json);

    builder.setVersion(1.1);
    gson = builder.create();
    json = gson.toJson(monBean);
    System.out.println("Resultat version 1.1 = " + json);

    builder.setVersion(1.2);
    gson = builder.create();
    json = gson.toJson(monBean);
    System.out.println("Resultat version 1.2 = " + json);
  }
}
```

Résultat :

```
Resultat = {"champ10":"champ10","champ11":"champ11","champ12":"champ12"}
Resultat version 1.0 = {"champ10":"champ10","champ11":"champ11"}
Resultat version 1.1 = {"champ11":"champ11"}
Resultat version 1.2 = {"champ11":"champ11","champ12":"champ12"}
```

56.6. L'API Streaming

Depuis la version 1.6, en plus du modèle objets, Gson propose une API de streaming pour lire et écrire un document JSON. Cette API traite un document JSON sous la forme d'un ensemble d'éléments qui doivent être utilisés dans l'ordre.

Le modèle objets est accessible en utilisant la classe JsonElement. Elle est le point d'entrée qui permet de naviguer dans les éléments du modèle. Le binding objet/Json de Gson utilise cette API.

L'API streaming repose sur deux classes principales :

- JsonReader pour lire et analyser un document JSON
- JsonWriter pour créer un document JSON

L'API de Streaming est performante mais le code à produire pour l'utiliser est plus complexe car il est nécessaire de traiter chacun des éléments lus durant l'analyse ou à écrire.

Même si l'utilisation de l'API Streaming peut sembler lourde à mettre à oeuvre car elle nécessite beaucoup de code à écrire, elle est le moyen le plus rapide, le plus efficace et le plus puissant pour traiter des documents JSON. Elle permet aussi d'avoir le contrôle sur la manière dont le document est analysé et traité.

L'utilisation de l'API Streaming est particulièrement utile dans plusieurs cas :

- si le document JSON est important, impliquant que son modèle objets correspondant soit intégralement monté en mémoire. Ceci est particulièrement vrai dans des environnements à ressources limitées
- si le document doit être lu ou écrit avant qu'il ne soit intégralement disponible

Durant une analyse de type streaming, chaque élément du document JSON est représenté par un token.

Exemple : { "nom":"nom1" }

Cet exemple contient quatre tokens :

- début d'objet : {
- nom de propriété : "nom"
- valeur de la propriété : "nom1"
- fin d'objet : }

56.6.1. L'énumération JsonToken

Lors de l'analyse d'un document JSON par la classe JsonReader, un token est émis pour chaque élément du document.

Les différents tokens sont définis dans l'énumération com.google.gson.stream.JsonToken

Valeur	Rôle
BEGIN_ARRAY	Le début d'un tableau
BEGIN_OBJECT	Le début d'un objet
BOOLEAN	La valeur true ou false
END_ARRAY	La fermeture d'un tableau
END_DOCUMENT	La fin du document
END_OBJECT	La fermeture d'un objet
NAME	Le nom d'une propriété

NULL	La valeur null
NUMBER	Une valeur numérique qui peut correspondre à une valeur Java de type int, long ou double
STRING	Une valeur sous la forme d'une chaîne de caractères

56.6.2. Le parcours d'un document JSON avec la classe JsonReader

La classe com.google.gson.stream.JsonReader permet de lire et analyser un document JSON en mode stream : à chaque élément du document, elle émet un token correspondant à l'élément courant.

Un token est émis pour les différents éléments qui composent le document JSON tels que les valeurs littérales, les délimiteurs de début et de fin d'objets ou de tableau, les paires nom/valeur d'un objet, ... Les tokens sont émis au fur et à mesure de la progression dans le document JSON.

Elle ne possède qu'un seul constructeur qui attend en paramètre un objet de type Reader.

Le pilotage du parcours, qui est obligatoirement séquentiel, se fait en utilisant les différentes méthodes de la classe JsonReader :

Méthode	Rôle
void beginArray()	Consommer le prochain token en présumant qu'il correspond à un début de tableau
void beginObject()	Consommer le prochain token en présumant qu'il correspond à un début d'un objet
void close()	Fermer le Reader encapsulant la lecture du document
void endArray()	Consommer le prochain token en présumant qu'il correspond à une fin de tableau
void endObject()	Consommer le prochain token en présumant qu'il correspond à la fin d'un objet
boolean hasNext()	Renvoyer un booléen qui précise si le tableau ou l'objet courant possède encore un élément
boolean isLenient()	Renvoyer un booléen qui indique si le parseur est permissif
boolean nextBoolean()	Consommer et renvoyer la valeur du prochain token en présumant que c'est une valeur booléenne
double nextDouble()	Consommer et renvoyer la valeur du prochain token en présumant que c'est une valeur numérique de type double
int nextInt()	Consommer et renvoyer la valeur du prochain token en présumant que c'est une valeur numérique de type int
long nextLong()	Consommer et renvoyer la valeur du prochain token en présumant que c'est une valeur numérique de type long
String nextName()	Consommer et renvoyer la valeur du prochain token en présumant que c'est le nom d'une propriété
void nextNull()	Consommer et renvoyer la valeur du prochain token en présumant que c'est une valeur null
String nextString()	Consommer et renvoyer la valeur du prochain token en présumant que c'est une valeur de type chaîne de caractères
JsonToken peek()	Renvoyer le type du prochain token sans le consommer
void setLenient(boolean lenient)	Indiquer au parseur s'il doit être permissif (true) ou non (false)
void skipValue()	Consommer le prochain token en l'ignorant et en présumant que c'est une valeur

La méthode skipValue() permet d'ignorer une valeur dans le document.

Pour gérer les valeurs null, il est nécessaire d'utiliser la méthode peek() pour déterminer la nature du prochaine token : si le token est JsonToken.NULL alors la valeur est null.

La classe JsonReader n'est pas thread-safe.

Il est nécessaire d'écrire du code qui va gérer les tokens de chaque sous-structure du document Json.

56.6.2.1. Le traitement d'un objet

Le traitement d'un objet contient généralement les opérations suivantes :

- invoquer la méthode beginObject()
- itérer tant que hasNext() pour obtenir le nom du champ avec la méthode nextName() et obtenir la valeur correspondante en invoquant une des méthodes nextXXX()
- invoquer la méthode endObject()

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.io.IOException;
import java.io.StringReader;

import com.google.gson.stream.JsonReader;

public class TestJsonReaderObjet {

  public static void main(final String[] args) {

    final String json = "{\"id\":1,\"nom\":\"nom1\",\"prenom\":\"prenom1\"}";

    try {
      final Personne personne = new Personne(0, "", "");
      final JsonReader reader = new JsonReader(new StringReader(json));

      reader.beginObject();

      while (reader.hasNext()) {
        final String name = reader.nextName();
        if (name.equals("id")) {
          personne.setId(reader.nextLong());
        } else if (name.equals("nom")) {
          personne.setNom(reader.nextString());
        } else if (name.equals("prenom")) {
          personne.setPrenom(reader.nextString());
        } else {
          reader.skipValue();
        }
      }

      reader.endObject();
      reader.close();
      System.out.println("Personne = " + personne);
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

Cet exemple est très simple car il ne gère pas certains cas particuliers comme les valeurs null.

Exemple :

```
// ...
    final String json = "{\"id\":1,\"nom\":\"nom1\",\"prenom\":null}";
// ...
```

Résultat :

```
Exception in thread "main" java.lang.IllegalStateException: Expected a string
but was NULL at line 1 column 35
        at com.google.gson.stream.JsonReader.nextString(JsonReader.java:821)
        at fr.jmdoudoux.dej.gson.TestJsonReaderObjet.main(TestJsonReaderObjet.java:27)
```

Il est possible d'invoquer la méthode peek() et de tester le type de token retourné pour déterminer si une valeur est null.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.io.IOException;
import java.io.StringReader;

import com.google.gson.stream.JsonReader;
import com.google.gson.stream.JsonToken;

public class TestJsonReaderObjet {

  public static void main(final String[] args) {

    final String json = "{\"id\":1,\"nom\":\"nom1\",\"prenom\":null}";

    try {
      final Personne personne = new Personne(0, "", "");
      final JsonReader reader = new JsonReader(new StringReader(json));

      reader.beginObject();

      while (reader.hasNext()) {
        final String name = reader.nextName();
        if (name.equals("id")) {
          personne.setId(reader.nextLong());
        } else if (name.equals("nom")) {
          personne.setNom(reader.nextString());
        } else if (name.equals("prenom")) {

          if (reader.peek() == JsonToken.NULL) {
            reader.nextNull(); // ou skipValue()
            personne.setPrenom(null);
          } else {
            personne.setPrenom(reader.nextString());
          }
        } else {
          reader.skipValue();
        }
      }

      reader.endObject();
      reader.close();
      System.out.println("Personne = " + personne);
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

56.6.2.2. Le traitement d'un tableau

Le traitement d'un objet contient généralement les opérations suivantes :

- invoquer la méthode beginArray()
- itérer tant que hasNext() pour obtenir chaque élément en invoquant une des méthodes nextXXX().
- invoquer la méthode endArray()

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.io.IOException;
import java.io.StringReader;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import com.google.gson.stream.JsonReader;

public class TestJsonReaderTableau {

  public static void main(final String[] args) {
    final String json = "[\"element1\",\"element2\",\"element3\",\"element4\"]";

    try {
      final List<String> liste = new ArrayList<String>();
      final JsonReader reader = new JsonReader(new StringReader(json));

      reader.beginArray();

      while (reader.hasNext()) {
        liste.add(reader.nextString());
      }
      reader.endArray();
      reader.close();
      System.out.println("Liste = " + Arrays.deepToString(liste.toArray()));
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

56.6.2.3. Le traitement d'éléments composites

Lorsque le document JSON est composé des différentes structures imbriquées, il est nécessaire de les traiter séquentiellement une par une.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.io.IOException;
import java.io.StringReader;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import com.google.gson.stream.JsonReader;
import com.google.gson.stream.JsonToken;

public class TestJsonReader {

  public static void main(final String[] args) {
    final TestJsonReader app = new TestJsonReader();
    final String json = "[{\"id\":1,\"nom\":\"nom1\",\"prenom\":null},"
      +"{\"id\":2,\"nom\":\"nom2\",\"prenom\":\"prenom2\"}]";

    try {
      List<Personne> liste = null;
      final JsonReader reader = new JsonReader(new StringReader(json));
      liste = app.lirePersonnes(reader);
      reader.close();
      System.out.println("Liste = " + Arrays.deepToString(liste.toArray()));
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
  public List<Personne> lirePersonnes(final JsonReader reader) throws IOException {
    final List<Personne> resultat = new ArrayList<Personne>();
    reader.beginArray();
    while (reader.hasNext()) {
      resultat.add(lirePersonne(reader));
    }
    reader.endArray();
    return resultat;
  }

  public Personne lirePersonne(final JsonReader reader) throws IOException {
    final Personne personne = new Personne(0, "", "");
    reader.beginObject();
    while (reader.hasNext()) {
      final String name = reader.nextName();
      if (name.equals("id")) {
        personne.setId(reader.nextLong());
      } else if (name.equals("nom")) {
        personne.setNom(reader.nextString());
      } else if (name.equals("prenom")) {

        if (reader.peek() == JsonToken.NULL) {
          reader.nextNull(); // ou skipValue()
          personne.setPrenom(null);
        } else {
          personne.setPrenom(reader.nextString());
        }
      } else {
        reader.skipValue();
      }
    }
    reader.endObject();
    return personne;
  }
}
```

Un JsonReader permet de lire les valeurs numériques indifféremment qu'elles soient sous la forme d'un nombre ou d'une chaîne de caractères dans le document JSON. Dans ce cas, il est possible d'obtenir la valeur en utilisant les méthodes nextInt(), nextLong() ou nextString().

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.io.IOException;
import java.io.StringReader;

import com.google.gson.stream.JsonReader;

public class TestJsonReaderNumerique {

  public static void main(final String[] args) {
    new TestJsonReader();
    final String json = "[1,\"1\"]";

    try {
      final JsonReader reader = new JsonReader(new StringReader(json));
      reader.beginArray();
      final String valeur1 = reader.nextString();
      System.out.println("valeur1=" + valeur1);
      final long valeur2 = reader.nextLong();
      System.out.println("valeur2=" + valeur2);
      reader.endArray();
      reader.close();
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
valeur1=1
valeur2=1
```

Ce comportement permet de contourner la façon dont JavaScript gère certaines valeurs numériques notamment les entiers longs puisque ceux-ci doivent être représentés sous la forme de chaînes de caractères.

Parfois un document JSON qui débute par un préfixe empêche son exécution direct en JavaScript. C'est notamment le cas des documents créés à partir d'une instance de type Gson produite par un GsonBuilder dont la méthode generateNonExecutableJson() a été invoquée.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.io.IOException;
import java.io.StringReader;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import com.google.gson.stream.JsonReader;

public class TestJsonReaderLenient {

  public static void main(final String[] args) {
    final String json = ")]}'\n[\"element1\",\"element2\"]";

    try {
      final List<String> liste = new ArrayList<String>();
      final JsonReader reader = new JsonReader(new StringReader(json));
      reader.beginArray();
      while (reader.hasNext()) {
        liste.add(reader.nextString());
      }
      reader.endArray();
      reader.close();
      System.out.println("Liste = " + Arrays.deepToString(liste.toArray()));
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
com.google.gson.stream.MalformedJsonException: Use JsonReader.setLenient(true)
to accept malformed JSON at line 1 column 1
        at com.google.gson.stream.JsonReader.syntaxError(JsonReader.java:1505)
        at com.google.gson.stream.JsonReader.checkLenient(JsonReader.java:1386)
        at com.google.gson.stream.JsonReader.doPeek(JsonReader.java:572)
        at com.google.gson.stream.JsonReader.beginArray(JsonReader.java:332)
        at fr.jmdoudoux.dej.gson.TestJsonReaderLenient.main(TestJsonReaderLenient.java:19)
```

Dans ce cas, il peut être utile d'invoquer la méthode setLenient() avec en paramètre la valeur true pour permettre d'analyser le document sans erreur.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.io.IOException;
import java.io.StringReader;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import com.google.gson.stream.JsonReader;

public class TestJsonReaderLenient {

  public static void main(final String[] args) {
    final String json = ")]}'\n[\"element1\",\"element2\"]";

    try {
      final List<String> liste = new ArrayList<String>();
      final JsonReader reader = new JsonReader(new StringReader(json));
      reader.setLenient(true);
      reader.beginArray();
      while (reader.hasNext()) {
        liste.add(reader.nextString());
      }
      reader.endArray();
      reader.close();
      System.out.println("Liste = " + Arrays.deepToString(liste.toArray()));
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

56.6.3. La création d'un document JSON avec la classe JsonWriter

La classe JsonWriter permet de créer un document JSON en lui ajoutant chacun des éléments un par un.

Il faut créer une instance de type JsonWriter. Elle ne possède qu'un seul constructeur qui attend en paramètre une instance de type Writer qui encapsule le flux vers le document.

Elle propose plusieurs méthodes pour ajouter les différents éléments et configurer la création du document :

Méthode	Rôle
JsonWriter beginArray()	Ajouter le début d'un nouveau tableau
JsonWriter beginObject()	Ajouter le début d'un nouvel objet
void close()	Fermer le Writer avec une invocation préalable de la méthode flush()
JsonWriter endArray()	Ajouter une fin de tableau
JsonWriter endObject()	Ajouter une fin d'objet
void flush()	Demander à ce que toutes les données en cours soient envoyées au Writer
boolean getSerializeNulls()	Renvoyer un booléen qui précise si un membre d'un objet doit être ajouté au document quand sa valeur est null
boolean isHtmlSafe()	Renvoyer un booléen qui précise si l'instance est configurée pour que le document généré puisse être inclus dans un document HTML ou XML
boolean isLenient()	Renvoyer un booléen qui précise si l'objet est configuré pour être permissif sur la syntaxe
JsonWriter name(String name)	Ajouter le nom d'une propriété
JsonWriter nullValue()	Ajouter une valeur null
void setHtmlSafe(boolean htmlSafe)	Configurer l'instance pour que le document généré puisse être inclus dans un document HTML ou XML
void setIndent(String indent)	Préciser la chaîne de caractères qui sera utilisée pour l'indentation des éléments du document
void setLenient(boolean lenient)	Configurer l'instance pour être permissive sur la syntaxe du document généré
void setSerializeNulls(boolean serializeNulls)	Préciser si les valeurs null doivent être incluses dans le document

JsonWriter value(boolean value)	Ajouter une valeur de type booléen
JsonWriter value(double value)	Ajouter une valeur de type double
JsonWriter value(long value)	Ajouter une valeur de type long
JsonWriter value(Number value)	Ajouter une valeur de type numérique
JsonWriter value(String value)	Ajouter une valeur de type chaîne de caractères

La racine d'un document Json doit être soit un objet soit un tableau.

Par défaut, l'invocation d'une méthode qui va engendrer la génération d'un document JSON malformé lève une exception de type IllegalStateException.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.io.IOException;
import java.io.StringWriter;

import com.google.gson.stream.JsonWriter;

public class TestJsonWriterErreur {

  public static void main(final String[] args) {
    try {
      final StringWriter stringWriter = new StringWriter();
      final JsonWriter jsonWriter = new JsonWriter(stringWriter);
      jsonWriter.beginArray();
      jsonWriter.value("element 1");
      jsonWriter.endArray();
      jsonWriter.endArray();
      jsonWriter.close();
      System.out.println("json=" + stringWriter.toString());
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
Exception in thread "main" java.lang.IllegalStateException: Nesting problem.
        at com.google.gson.stream.JsonWriter.close(JsonWriter.java:339)
        at com.google.gson.stream.JsonWriter.endArray(JsonWriter.java:297)
        at fr.jmdoudoux.dej.gson.TestJsonWriterErreur.main(TestJsonWriterErreur.java:17)
```

56.6.3.1. L'ajout d'un objet

L'ajout d'un objet implique généralement les opérations suivantes :

- beginObject()
- pour chaque champ, invoquer les méthodes name() pour préciser le nom et value() pour préciser la valeur
- endObject() ;

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.io.IOException;
import java.io.StringWriter;

import com.google.gson.stream.JsonWriter;

public class TestJsonWriterObjet {

  public static void main(final String[] args) {
    try {
      final StringWriter stringWriter = new StringWriter();
      final JsonWriter jsonWriter = new JsonWriter(stringWriter);
      jsonWriter.beginObject();
      jsonWriter.name("id").value(1);
      jsonWriter.name("nom").value("nom1");
      jsonWriter.name("prenom").value("prenom1");
      jsonWriter.endObject();
      jsonWriter.close();
      System.out.println("json=" + stringWriter.toString());
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
json={"id":1,"nom":"nom1","prenom":"prenom1"}
```

Pour demander un échappement des caractères contenus dans le document JSON, ce qui permettra d'intégrer le texte dans un fichier HTML ou XML, il faut passer la valeur true à la méthode setHtmlSafe().

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.io.IOException;
import java.io.StringWriter;

import com.google.gson.stream.JsonWriter;

public class TestJsonWriterObjet {

  public static void main(final String[] args) {
    try {
      final StringWriter stringWriter = new StringWriter();
      final JsonWriter jsonWriter = new JsonWriter(stringWriter);
      jsonWriter.setHtmlSafe(true);
      jsonWriter.beginObject();
      jsonWriter.name("id").value(1);
      jsonWriter.name("nom").value("nom1 & 2");
      jsonWriter.name("prenom").value("prenom1");
      jsonWriter.endObject();
      jsonWriter.close();
      System.out.println("json=" + stringWriter.toString());
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
json={"id":1,"nom":"nom1 \u0026 2","prenom":"prenom1"}
```

Pour mettre une valeur null, il faut utiliser la méthode nullValue().

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.io.IOException;
import java.io.StringWriter;

import com.google.gson.stream.JsonWriter;
```

```
public class TestJsonWriterObjet {

  public static void main(final String[] args) {
    try {
      final StringWriter stringWriter = new StringWriter();
      final JsonWriter jsonWriter = new JsonWriter(stringWriter);
      jsonWriter.beginObject();
      jsonWriter.name("id").value(1);
      jsonWriter.name("nom").value("nom1");
      jsonWriter.name("prenom").nullValue();
      jsonWriter.endObject();
      jsonWriter.close();
      System.out.println("json=" + stringWriter.toString());
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
json={"id":1,"nom":"nom1 & 2","prenom":null}
```

56.6.3.2. L'ajout d'un tableau

L'ajout d'un objet implique généralement les opérations suivantes :

- beginArray()
- pour chaque occurrence, invoquer la méthode value() pour préciser la valeur
- endArray() ;

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.io.IOException;
import java.io.StringWriter;

import com.google.gson.stream.JsonWriter;

public class TestJsonWriterTableau {

  public static void main(final String[] args) {
    try {
      final StringWriter stringWriter = new StringWriter();
      final JsonWriter jsonWriter = new JsonWriter(stringWriter);
      jsonWriter.beginArray();
      jsonWriter.value("element 1");
      jsonWriter.value("element 2");
      jsonWriter.nullValue();
      jsonWriter.endArray();
      jsonWriter.close();
      System.out.println("json=" + stringWriter.toString());
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
json=["element 1","element 2",null]
```

56.6.3.3. L'ajout d'éléments composites

Lorsque le document JSON est composé des différentes structures imbriquées, il est nécessaire de les traiter séquentiellement une par une.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.io.IOException;
import java.io.StringWriter;

import com.google.gson.stream.JsonWriter;

public class TestJsonWriter {

  public static void main(final String[] args) {
    try {
      final StringWriter stringWriter = new StringWriter();
      final JsonWriter jsonWriter = new JsonWriter(stringWriter);
      jsonWriter.beginArray();
      jsonWriter.beginObject();
      jsonWriter.name("id").value(1);
      jsonWriter.name("nom").value("nom1");
      jsonWriter.name("prenom").nullValue();
      jsonWriter.endObject();
      jsonWriter.beginObject();
      jsonWriter.name("id").value(2);
      jsonWriter.name("nom").value("nom2");
      jsonWriter.name("prenom").value("prenom2");
      jsonWriter.endObject();
      jsonWriter.endArray();
      jsonWriter.close();
      System.out.println("json=" + stringWriter.toString());
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
json=[{"id":1,"nom":"nom1","prenom":null},{"id":2,"nom":"nom2","prenom":"prenom2"}]
```

56.7. Mixer l'utilisation du model objets et de l'API Streaming

Gson permet aussi de combiner les deux modèles selon les besoins. L'utilisation du modèle objets peut être mixée avec celle de l'API Streaming pour bénéficier des points forts de chacune des approches :

- productivité du modèle objets
- efficacité du parsing de l'API streaming

Dans ce cas, il faut utiliser l'API Streaming pour parcourir ou générer un document et utiliser les méthodes fromJson() ou toJson() de la classe Gson pour créer ou lire un élément.

Les deux API offrent des passerelles pour faciliter leurs utilisations simultanées.

56.7.1. Mixer l'utilisation pour analyser un document

Il est possible d'employer un JsonReader pour parcourir la structure du document et d'utiliser le modèle objets pour créer directement des objets. Ceci évite d'avoir à analyser et créer manuellement les objets tout en gardant la main sur le parcours du document.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.io.IOException;
import java.io.StringReader;
import java.io.StringWriter;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import com.google.gson.stream.JsonReader;

public class TestGsonStreamObjetParse {

  public static void main(final String[] args) {
    final Gson gson = new GsonBuilder().create();
    final String jsonIn = "[{\"id\":1,\"nom\":\"nom1\",\"prenom\":null},"
      +"{\"id\":2,\"nom\":\"nom2\",\"prenom\":\"prenom2\"}]";

    try {
      new StringWriter();
      final JsonReader reader = new JsonReader(new StringReader(jsonIn));

      final List<Personne> personnes = new ArrayList<Personne>();
      reader.beginArray();
      while (reader.hasNext()) {
        final Personne personne = gson.fromJson(reader, Personne.class);
        personnes.add(personne);
      }
      reader.endArray();
      reader.close();

      System.out.println("Resultat = " + Arrays.deepToString(personnes.toArray()));
      reader.close();
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
Resultat = [Personne [id=1, nom=nom1, prenom=null], Personne [id=2, nom=nom2,
prenom=prenom2]]
```

Une autre approche possible est de lire le document avec un JsonReader et de passer ce JsonReader en paramètre de la méthode parse() d'un JsonParser qui renvoie une instance de type JsonElement. Il suffit alors de traiter ce JsonElement.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.io.IOException;
import java.io.StringReader;
import java.io.StringWriter;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import com.google.gson.JsonArray;
import com.google.gson.JsonElement;
import com.google.gson.JsonParser;
import com.google.gson.stream.JsonReader;

public class TestGsonStreamObjetParse2 {
  public static void main(final String[] args) {
    final Gson gson = new GsonBuilder().create();
    final String jsonIn = "[{\"id\":1,\"nom\":\"nom1\",\"prenom\":null},"
      +"{\"id\":2,\"nom\":\"nom2\",\"prenom\":\"prenom2\"}]";

    try {
      new StringWriter();
      final JsonReader jsonReader = new JsonReader(new StringReader(jsonIn));
      final JsonParser jsonParser = new JsonParser();
      final JsonArray jsonArray = jsonParser.parse(jsonReader).getAsJsonArray();

      final List<Personne> personnes = new ArrayList<Personne>();
      for (final JsonElement element : jsonArray) {
        final Personne personne = gson.fromJson(element, Personne.class);
        personnes.add(personne);
      }

      System.out.println("Resultat = " + Arrays.deepToString(personnes.toArray()));
      jsonReader.close();
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
Resultat = [Personne [id=1, nom=nom1, prenom=null], Personne
[id=2, nom=nom2, prenom=prenom2]]
```

56.7.2. Mixer l'utilisation pour générer un document

Il est possible d'utiliser un JsonWriter pour créer les éléments relatifs à la structure du document et de créer les différents objets en utilisant la méthode toJson() de la classe Gson.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.io.IOException;
import java.io.StringWriter;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import com.google.gson.stream.JsonWriter;

public class TestGsonStreamObjetWrite {

  public static void main(final String[] args) {
    final Gson gson = new GsonBuilder().create();
    final Personne personne1 = new Personne(1, "nom1", "prenom1");
    final Personne personne2 = new Personne(2, "nom2", "prenom2");

    try {
      final StringWriter stringWriter = new StringWriter();
      final JsonWriter jsonWriter = new JsonWriter(stringWriter);
      jsonWriter.beginArray();
      gson.toJson(personne1, Personne.class, jsonWriter);
      gson.toJson(personne2, Personne.class, jsonWriter);
      jsonWriter.endArray();
      jsonWriter.close();
      System.out.println("json=" + stringWriter.toString());
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
json=[{"id":1,"nom":"nom1","prenom":"prenom1"},{"id":2,"nom":"nom2","prenom":"prenom2"}]
```

56.8. Les concepts avancés

Généralement les opérations de sérialisation/désérialisation ne concernent pas simplement des éléments simples comme des types primitifs, des objets ou des collections mais utilisent des concepts avancés comme les generics, les collections avec des types différents ou des graphes d'objets.

56.8.1. La sérialisation/désérialisation de types generic

La sérialisation/désérialisation par défaut de Gson fonctionne bien si les classes ne sont pas typées avec des generics. Si la classe est définie en utilisant un generic alors le type sera perdu à cause de l'implémentation des generics qui repose sur le Type Erasure. Le type generic est perdu après la phase de compilation : il n'y a alors aucun moyen de le déterminer à l'exécution.

Exemple :

```
package fr.jmdoudoux.dej.gson;

public class MaClasse<T> {
  private final T      monBean;
  private final String nom;

  public MaClasse(final String nom, final T monBean) {
    super();
    this.nom = nom;
    this.monBean = monBean;
  }

  @Override
  public String toString() {
    return "MaClasse [monBean=" + this.monBean + ", nom=" + this.nom + "]";
  }

  public T getMonBean() {
    return this.monBean;
  }
}
```

Le problème va survenir lors de la désérialisation car Gson ne peut pas retrouver le type generic.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class TestSerialisation {

  public static void main(final String[] args) {
    final Gson gson = new GsonBuilder().create();
    final MonBean monBean = new MonBean();
    monBean.setChamp10("champ 10");
    monBean.setChamp11("champ 11");
    monBean.setChamp12("champ 12");
    final MaClasse monObjet = new MaClasse<MonBean>("ma classe", monBean);
    final String json = gson.toJson(monObjet);

    System.out.println("Resultat=" + json);
    System.out.println("classe monObjet=" + monObjet.getClass().getName());

    final MaClasse monObjetLu = gson.fromJson(json, monObjet.getClass());
    System.out.println("classe monObjetLu=" + monObjetLu.getClass().getName());
    System.out.println(monObjetLu.getMonBean().toString());
    System.out.println("classe monObjetLu.getBean="
      + monObjetLu.getMonBean().getClass().getName());
  }
}
```

Résultat :

```
Resultat={"monBean":{"champ10":"champ 10", "champ11":"champ 11",
      "champ12":"champ 12"},"nom":"ma classe"}
classe monObjet=fr.jmdoudoux.dej.gson.MaClasse
classe monObjetLu=fr.jmdoudoux.dej.gson.MaClasse
{champ10=champ 10, champ11=champ 11, champ12=champ 12}
classe monObjetLu.getBean=com.google.gson.internal.LinkedTreeMap
```

Pour permettre à Gson d'instancier le bon type, ce dernier doit lui être précisé en utilisant une instance de la classe TypeToken.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.lang.reflect.Type;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import com.google.gson.reflect.TypeToken;

public class TestSerialisation {

  public static void main(final String[] args) {
    final Gson gson = new GsonBuilder().create();
    final MonBean monBean = new MonBean();
    monBean.setChamp10("champ 10");
    monBean.setChamp11("champ 11");
    monBean.setChamp12("champ 12");
    final MaClasse<MonBean> monObjet = new MaClasse<MonBean>("ma classe", monBean);
    final String json = gson.toJson(monObjet);

    System.out.println("Resultat=" + json);
    System.out.println("classe monObjet=" + monObjet.getClass().getName());

    final Type maClasseType = new TypeToken<MaClasse<MonBean>>() {
    }.getType();
    final MaClasse<MonBean> monObjetLu = gson.fromJson(json, maClasseType);
    System.out.println("classe monObjetLu=" + monObjetLu.getClass().getName());
    System.out.println(monObjetLu.getMonBean().toString());
    System.out.println("classe monObjetLu.getBean="
      + monObjetLu.getMonBean().getClass().getName());
  }
}
```

Résultat :

```
Resultat={"monBean":{"champ10":"champ 10",
      "champ11":"champ 11",
      "champ12":"champ 12"},"nom":"ma classe"}
classe monObjet=fr.jmdoudoux.dej.gson.MaClasse
classe monObjetLu=fr.jmdoudoux.dej.gson.MaClasse
MonBean [champ10=champ 10, champ11=champ 11, champ12=champ 12]
classe monObjetLu.getBean=fr.jmdoudoux.dej.gson.MonBean
```

La syntaxe utilisée repose sur l'instanciation d'une classe anonyme interne dont la méthode getType() est invoquée. L'instance de type java.lang.reflect.Type retournée doit être passée en paramètre de la méthode fromJson().

56.8.2. La sérialisation/désérialisation d'une collection contenant différents types

Même si ce n'est pas conseillé, la sérialisation d'une collection contenant différents types ne pose aucun problème particulier.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.util.ArrayList;
import java.util.List;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class TestSerialisation {

  public static void main(final String[] args) {
    final MonBean monBean = new MonBean("valeur10", "valeur11", "valeur12");

    final GsonBuilder builder = new GsonBuilder();
    final Gson gson = builder.create();
    final List maListe = new ArrayList();
    maListe.add(12345);
    maListe.add("test");
    maListe.add(monBean);

    final String json = gson.toJson(maListe);
    System.out.println("Resultat=" + json);
  }
}
```

Résultat :

```
Resultat=[12345,"test",{"champ10":"valeur10","champ11":"valeur11","champ12":"valeur12"}]
```

Par contre la méthode fromJson() ne va pas pouvoir désérialiser le document Json puisqu'elle n'a aucun moyen de déterminer le type de chaque élément. Normalement, il faudrait typer la collection avec un generic qui précise le type à utiliser : cependant cela oblige à n'avoir qu'un seul type d'éléments dans la collection.

Le plus simple dans ce cas est d'utiliser l'API Streaming ou d'utiliser la classe JsonParser pour parcourir chacun des éléments du tableau et de les traiter individuellement en invoquant la méthode fromJson() de la classe Gson.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import com.google.gson.JsonArray;
import com.google.gson.JsonParser;

public class TestSerialisation {

  public static void main(final String[] args) {
    final MonBean monBean = new MonBean("valeur10", "valeur11", "valeur12");

    final GsonBuilder builder = new GsonBuilder();
    final Gson gson = builder.create();
    final List maListe = new ArrayList();
    maListe.add(12345);
    maListe.add("test");
    maListe.add(monBean);

    final String json = gson.toJson(maListe);
    System.out.println("Resultat serialisation=" + json);

    final List liste = new ArrayList();
    final JsonParser parser = new JsonParser();
    final JsonArray array = parser.parse(json).getAsJsonArray();
    final int element1 = gson.fromJson(array.get(0), int.class);
    liste.add(element1);
    final String element2 = gson.fromJson(array.get(1), String.class);
    liste.add(element2);
    final MonBean element3 = gson.fromJson(array.get(2), MonBean.class);
    liste.add(element3);
    System.out.println("Resultat deserialisation=" + Arrays.deepToString(liste.toArray()));
  }
}
```

Résultat :

```
Resultat serialisation=[12345,"test",{"champ10":"valeur10", "champ11":"valeur11","champ12"
:"valeur12"}]
Resultat deserialisation=[12345, test, MonBean [champ10=valeur10, champ11=valeur11, champ12=val
eur12]]
```

56.8.3. La désérialisation quand un même objet est référencé plusieurs fois

Lorsque l'on sérialise un graphe d'objets, il est possible qu'une même instance soit référencée plusieurs fois dans le graphe.

Dans l'exemple de cette section, un groupe est composé d'une ou plusieurs personnes.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Groupe {

  private final long           id;
  private final String         nom;
  private final List<Personne> personnes = new ArrayList<Personne>();

  public Groupe(final long id, final String nom) {
    super();
    this.nom = nom;
    this.id = id;
  }

  public String getNom() {
    return this.nom;
  }

  public long getId() {
    return this.id;
  }

  public List<Personne> getPersonnes() {
    return this.personnes;
  }

  public void ajouter(final Personne personne) {
    this.personnes.add(personne);
  }

  @Override
  public String toString() {
    return "Groupe [id=" + this.id + ", nom=" + this.nom + ", personnes="
        + Arrays.deepToString(this.personnes.toArray()) + "]";
  }
}
```

Une personne peut appartenir à plusieurs groupes.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class TestSerialisation {

  public static void main(final String[] args) {
    final Personne[] personnes = new Personne[5];
    for (int i = 0; i < 5; i++) {
      personnes[i] = new Personne(i, "nom" + i, "prenom" + i);
    }

    final Groupe[] groupe = new Groupe[3];
    for (int i = 0; i < 3; i++) {
      groupe[i] = new Groupe(i, "groupe" + i);
    }

    final Groupes groupes = new Groupes();
    groupes.ajouterGroupe(groupe[0], personnes[0], personnes[2], personnes[4]);
    groupes.ajouterGroupe(groupe[1], personnes[1], personnes[4]);
    groupes.ajouterGroupe(groupe[2], personnes[1], personnes[2], personnes[3]);

    final GsonBuilder builder = new GsonBuilder();
    builder.setPrettyPrinting();
    builder.registerTypeAdapter(Groupes.class, new GroupesDeserializer());
    final Gson gson = builder.create();

    final String json = gson.toJson(groupes.getGroupes());
    System.out.println("Resultat serialisation=" + json);

  }
}
```

Résultat :

```
Resultat serialisation=[
  { "id": 0,
    "nom": "groupe0",
    "personnes": [
      { "id": 0, "nom": "nom0", "prenom": "prenom0" },
      { "id": 2, "nom": "nom2", "prenom": "prenom2" },
      { "id": 4, "nom": "nom4","prenom": "prenom4" } ] },
  { "id": 1,
    "nom": "groupe1",
    "personnes": [
      { "id": 1, "nom": "nom1", "prenom": "prenom1" },
      { "id": 4, "nom": "nom4", "prenom": "prenom4" } ] },
  { "id": 2,
    "nom": "groupe2",
    "personnes": [
      { "id": 1, "nom": "nom1", "prenom": "prenom1" },
      { "id": 2, "nom": "nom2", "prenom": "prenom2" },
      { "id": 3, "nom": "nom3", "prenom": "prenom3" } ] }
]
```

Comme une personne peut appartenir à plusieurs groupes, il y a deux solutions lors de la désérialisation :

- créer une nouvelle instance pour chaque personne : c'est le mode de fonctionnement pas défaut de Gson
- ne créer qu'une seule instance pour chaque personne distincte

Cela dépend du contexte mais généralement, la seconde solution est préférée. Elle nécessite de développer sa propre solution de désérialisation.

Il faut développer une classe qui va gérer les différentes instances.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class Groupes {

  private final Map<Long, Personne> personnes = new HashMap<Long, Personne>();
  private final List<Groupe>        groupes   = new ArrayList<Groupe>();

  protected void ajouterPersonne(final Personne personne) {
    this.personnes.put(personne.getId(), personne);
  }

  protected void ajouterGroupe(final Groupe groupe) {
    this.groupes.add(groupe);
  }

  public void ajouterGroupe(final Groupe groupe, final Personne... personnes) {
    for (final Personne personne : personnes) {
      Personne pers = this.personnes.get(personne.getId());
      if (pers == null) {
        ajouterPersonne(personne);
        pers = personne;
      }
      groupe.ajouter(pers);
    }
    ajouterGroupe(groupe);
  }

  public void afficherStats() {
    System.out.println("Nb personnes=" + this.personnes.size());
    System.out.println("Nb groupes=" + this.groupes.size());
  }

  public List<Groupe> getGroupes() {
    return this.groupes;
  }

  @Override
  public String toString() {
    return "Groupes [" + Arrays.deepToString(this.groupes.toArray()) + "]";
  }
}
```

La classe Groupes encapsule les instances de type Groupe dans une collection et les instances de types Personnes dans une Map.

Lors de l'ajout d'un groupe, les personnes à associer au groupe sont d'abord recherchées dans la Map pour obtenir des instances éventuellement existantes sinon elles sont ajoutées à la Map.

Il faut ensuite définir un Deserializer personnalisé qui va extraire les données du document json, créer les différentes instances et invoquer la méthode ajouterGroupe() de la classe Groupes pour réaliser les associations.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import java.lang.reflect.Type;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import com.google.gson.JsonArray;
import com.google.gson.JsonDeserializationContext;
import com.google.gson.JsonDeserializer;
import com.google.gson.JsonElement;
import com.google.gson.JsonNull;
import com.google.gson.JsonObject;
import com.google.gson.JsonParseException;

public class GroupesDeserializer implements JsonDeserializer<Groupes> {
  @Override
  public Groupes deserialize(final JsonElement json, final Type typeOfT,
      final JsonDeserializationContext context) throws JsonParseException {
    final Groupes resultat = new Groupes();

    if (json != JsonNull.INSTANCE) {
      final GsonBuilder builder = new GsonBuilder();
      final Gson gson = builder.create();
      final JsonArray jsonArray = json.getAsJsonArray();

      for (int i = 0; i < jsonArray.size(); i++) {
        final JsonObject jsonObject = jsonArray.get(i).getAsJsonObject();
        final long id = jsonObject.get("id").getAsLong();
        final String nom = jsonObject.get("nom").getAsString();
        final Groupe groupe = new Groupe(id, nom);

        final JsonArray personnesArray = jsonObject.getAsJsonArray("personnes");
        final Personne[] personnes = new Personne[personnesArray.size()];
        for (int j = 0; j < personnesArray.size(); j++) {
          final JsonObject jsonPersonne = personnesArray.get(j).getAsJsonObject();
          final Personne personne = gson.fromJson(jsonPersonne, Personne.class);
          personnes[j] = personne;
        }
        resultat.ajouterGroupe(groupe, personnes);
      }
    }
    return resultat;
  }
}
```

Il suffit alors d'associer le Deserializer à l'instance de type Gson et de l'utiliser pour désérialiser le document json.

Exemple :

```
package fr.jmdoudoux.dej.gson;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class TestSerialisation {

  public static void main(final String[] args) {
    final Personne[] personnes = new Personne[5];
    for (int i = 0; i < 5; i++) {
      personnes[i] = new Personne(i, "nom" + i, "prenom" + i);
    }

    final Groupe[] groupe = new Groupe[3];
    for (int i = 0; i < 3; i++) {
      groupe[i] = new Groupe(i, "groupe" + i);
    }

    final Groupes groupes = new Groupes();
    groupes.ajouterGroupe(groupe[0], personnes[0], personnes[2], personnes[4]);
    groupes.ajouterGroupe(groupe[1], personnes[1], personnes[4]);
    groupes.ajouterGroupe(groupe[2], personnes[1], personnes[2], personnes[3]);

    final GsonBuilder builder = new GsonBuilder();
    builder.setPrettyPrinting();
    builder.registerTypeAdapter(Groupes.class, new GroupesDeserializer());
    final Gson gson = builder.create();
    final String json = gson.toJson(groupes.getGroupes());

    final Groupes groupesDes = gson.fromJson(json, Groupes.class);
    System.out.println("Deserialisation avec serializer = " + groupesDes);
    groupesDes.afficherStats();
  }
}
```

Résultat :

```
Deserialisation avec serializer = Groupes [[Groupe [id=0, nom=groupe0,
      personnes=[Personne [id=0, nom=nom0, prenom=prenom0],
      Personne [id=2, nom=nom2, prenom=prenom2],
      Personne [id=4, nom=nom4, prenom=prenom4]]],
      Groupe [id=1, nom=groupe1, personnes=[Personne [id=1, nom=nom1, prenom=prenom1],
      Personne [id=4, nom=nom4, prenom=prenom4]]],
      Groupe [id=2, nom=groupe2, personnes=[Personne [id=1, nom=nom1, prenom=prenom1],
      Personne [id=2, nom=nom2, prenom=prenom2],
      Personne [id=3, nom=nom3, prenom=prenom3]]]]]
Nb personnes=5
Nb groupes=3
```

57. JSON-P (Java API for JSON Processing)

Chapitre 57

Niveau : 🟩🟩🟨⬜⬜⬜ Intermédiaire

La JSR 353 définit une API standardisée qui permet de parser et générer un document JSON : JSON-P 1.0. Elle ne propose rien concernant le binding entre un document JSON et un objet Java.

Deux API sont proposées :

- Streaming API : API de bas niveau qui permet la consommation et la production d'un document JSON en utilisant un flux d'événements, de manière similaire à l'API StAX
- Object Model API : API de plus haut niveau qui utilise des objets, de manière similaire à DOM. Cette API utilise l'API Streaming.

Les classes et interfaces de l'API JSON-P sont contenues dans deux packages :

- javax.json : contient les classes et interfaces de l'Object Model API
- javax.json.stream : contient les classes et interfaces de la Streaming API

Les principales classes et interfaces du package javax.json sont :

Classe ou interface	Description
Json	Fabrique qui permet de créer des instances de certains types ou fabriques de l'API (Parser, Builder, Generator, Writer, Reader)
JsonReader	Créer un modèle objets à partir d'une représentation Json des données
JsonObjectBuilder JsonArrayBuilder	Créer un modèle objets ou un tableau en lui ajoutant des éléments
JsonWriter	Envoyer dans un flux une représentation Json d'un modèle objet
JsonValue JsonStructure JsonObject JsonArray JsonString JsonNumber	Encapsule les types de données d'un élément JSON JsonStructure, JsonObject, JsonArray, JsonString et JsonNumber sont des sous-types de JsonValue JsonObject et JsonArray sont des sous-types de JsonStructure
JsonException	Exception pouvant être levée lors du traitement de la représentation JSON

Les principales classes et interfaces du package javax.json.stream sont :

Classe ou interface	Rôle
JsonParser	Parser un document JSON en émettant des événements
JsonGenerator	Ecrire un document JSON : chaque élément est écrit un par un

Ce chapitre contient plusieurs sections :

- La classe Json
- L'API Streaming
- L'API Object Model

57.1. La classe Json

La classe Json est une fabrique qui permet de créer des instances des différents types d'objets à utiliser pour mettre en oeuvre l'API.

Méthode	Rôle
static JsonArrayBuilder createArrayBuilder()	Créer une instance de type JsonArrayBuilder
static JsonBuilderFactory createBuilderFactory(Map<String,?> config)	Créer une instance de type JsonBuilderFactory
static JsonGenerator createGenerator(OutputStream out)	Créer une instance de type JSonGenerator pour écrire le document JSON dans un flux d'octets
static JsonGenerator createGenerator(Writer writer)	Créer une instance de type JSonGenerator pour écrire le document JSON dans un flux de caractères
static JsonGeneratorFactory createGeneratorFactory(Map<String,?> config)	Créer une instance de type JsonGeneratorFactory
static JsonObjectBuilder createObjectBuilder()	Créer une instance de type JsonObjectBuilder
static JsonParser createParser(InputStream in)	Créer une instance de type JsonParser pour parser un document JSON à partir d'un flux d'octets
static JsonParser createParser(Reader reader)	Créer une instance de type JsonParser pour parser un document JSON à partir d'un flux de caractères
static JsonParserFactory createParserFactory(Map<String,?> config)	Créer une instance de type JsonParserFactory
static JsonReader createReader(InputStream in)	Créer une instance de type JsonReader pour lire un document JSON à partir d'un flux d'octets
static JsonReader createReader(Reader reader)	Créer une instance de type JsonReader pour lire un document JSON à partir d'un flux de caractères
static JsonReaderFactory createReaderFactory(Map<String,?> config)	Créer une instance de type JsonReaderFactory
static JsonWriter createWriter(OutputStream out)	Créer une instance de type JsonWriter pour écrire un document JSON dans d'un flux d'octets
static JsonWriter createWriter(Writer writer)	Créer une instance de type JsonWriter pour écrire un document JSON dans d'un flux de caractères
static JsonWriterFactory createWriterFactory(Map<String,?> config)	Créer une instance de type JsonWriterFactory

57.2. L'API Streaming

L'API Streaming permet de parcourir et de générer un document JSON sous la forme d'un flux.

Elle définit plusieurs interfaces :

- JsonParser : parser un document JSON selon des événements
- JsonParserFactory : fabrique d'instances de type JsonParser configurable

- JsonGenerator : écrire chacun des éléments d'un document JSON
- JsonGeneratorFactory : fabrique d'instances de type JsonGenerator configurable

57.2.1. L'interface JsonParser

L'interface javax.json.stream.JsonParser définit des méthodes pour parser un document JSON et émettre des événements durant le parcours d'une manière similaire à un XMLStreamReader de l'API StaX.

Les différents événements sont définis dans l'énumération JsonParser.Event

Constante	Rôle
END_ARRAY	Fin d'un tableau JSON
END_OBJECT	Fin d'un objet JSON
KEY_NAME	Nom d'une clé dans une paire nom/valeur d'un objet JSON
START_ARRAY	Début d'un tableau JSON
START_OBJECT	Début d'un objet JSON
VALUE_FALSE	La valeur false (dans un objet ou un tableau JSON)
VALUE_NULL	La valeur null (dans un objet ou un tableau JSON)
VALUE_NUMBER	Une valeur numérique (dans un objet ou un tableau JSON)
VALUE_STRING	La valeur alphanumérique (dans un objet ou un tableau JSON)
VALUE_TRUE	La valeur true (dans un objet ou un tableau JSON)

L'interface JsonParser définit plusieurs méthodes :

Méthode	Rôle
void close()	Fermer le parser et libérer les éventuelles ressources associées
BigDecimal getBigDecimal()	Renvoyer la valeur numérique courante sous la forme d'un BigDecimal
int getInt()	Renvoyer la valeur numérique courante sous la forme d'un entier
JsonLocation getLocation()	Renvoyer un objet qui encapsule la localisation courante du parser dans le document
long getLong()	Renvoyer la valeur numérique de la valeur courante
String getString()	Renvoyer sous forme d'une chaîne de caractères la valeur courante
boolean hasNext()	Renvoyer un booléen qui précise si le parcours n'est pas encore terminé en renvoyant true sinon renvoie false
boolean isIntegralNumber()	Renvoyer un booléen qui précise si la valeur numérique courante est un entier
JsonParser.Event next()	Renvoyer l'événement suivant

L'utilisation de l'API Streaming se fait en plusieurs étapes :

- obtenir une instance de la classe JsonParser
- itérer sur chaque événement en utilisant les méthodes hasNext() et next()
- traiter chaque événement au besoin selon son type en utilisant les méthodes de l'instance du parser pour obtenir les informations utiles
- invoquer explicitement la méthode close() ou implicitement en utilisant une instruction try avec ressources à partir de Java 7

Pour obtenir une instance de type JsonParser, il y a deux solutions :

- invoquer une des surcharges de la méthode createParser() de la classe Json qui attend en paramètre un objet de type InputStream ou Reader
- obtenir une instance de type JsonParserFactory en invoquant la méthode Json.createParserFactory() et invoquer la méthode createParser() sur cette instance. Cette façon de faire permet de configurer l'instance

Différentes informations peuvent être obtenues lors du parcours en utilisant des méthodes d'une instance de type JsonParser selon le type d'événement en cours de traitement :

- la valeur de l'élément courant en invoquant la méthode getString() pour les événements KEY_NAME, VALUE_STRING et VALUE_NUMBER
- la valeur de l'élément courant en invoquant les méthodes getNumberType(), getIntValue(), getLongValue(), getBigDecimalValue() et isIntegralNumber() pour l'événement VALUE_NUMBER

Exemple (code Java 7) :

```
      String document = "[{\n" + "\"nom\":\"nom1\", \"prenom\": \"prenom1\", \"taille\": 175\n"
              + "},\n"
              + "{\n"
              + "\"nom\": \"nom2\", \"prenom\": \"prenom2\",\"taille\": 183\n" + "}\n"
              + "]";
      try (JsonParser parser = Json.createParser(new StringReader(document))) {
        Event event = null;
        while (parser.hasNext()) {
          event = parser.next();
          System.out.print("event=" + event);
          switch (event) {
            case KEY_NAME:
              System.out.print(" cle=" + parser.getString());
              break;
            case VALUE_STRING:
              System.out.print(" valeur=" + parser.getString());
              break;
            case VALUE_NUMBER:
              if (parser.isIntegralNumber()) {
                System.out.println(" valeur=" + parser.getInt());
              } else {
                System.out.println(" valeur=" + parser.getBigDecimal());
              }
              break;
            case VALUE_NULL:
              System.out.print(" valeur=null");
              break;
          }
          System.out.println("");
        }
      } catch (Exception e) {
        e.printStackTrace();
      }
```

Résultat :

```
event=START_ARRAY
event=START_OBJECT
event=KEY_NAME
cle=nom
event=VALUE_STRING
valeur=nom1
event=KEY_NAME
cle=prenom
event=VALUE_STRING
valeur=prenom1
event=KEY_NAME
cle=taille
event=VALUE_NUMBER
valeur=175
event=END_OBJECT
event=START_OBJECT
event=KEY_NAME
cle=nom
```

```
event=VALUE_STRING
valeur=nom2
event=KEY_NAME
cle=prenom
event=VALUE_STRING
valeur=prenom2
event=KEY_NAME
cle=taille
event=VALUE_NUMBER
valeur=183
event=END_OBJECT
event=END_ARRAY
```

L'utilisation de la Streaming API est très efficace pour parser un document JSON par contre elle implique un surplus de code car il faut gérer chacun des événements.

57.2.2. L'interface JsonGenerator

L'interface javax.json.stream.JsonGenerator définit des méthodes pour faciliter l'ajout d'éléments JSON dans un flux d'octets ou de caractères.

Méthode	Rôle
void close()	Libérer les ressources associées au générateur
void flush()	Flush les données dans le flux associé
JsonGenerator write(BigDecimal value)	Ecrire la valeur numérique dans le tableau
JsonGenerator write(BigInteger value)	Ecrire la valeur numérique dans le tableau
JsonGenerator write(boolean value)	Ecrire la valeur booléenne dans le tableau
JsonGenerator write(double value)	Ecrire la valeur numérique dans le tableau
JsonGenerator write(int value)	Ecrire la valeur numérique dans le tableau
JsonGenerator write(JsonValue value)	Ecrire la valeur dans le tableau
JsonGenerator write(long value)	Ecrire la valeur numérique dans le tableau
JsonGenerator write(String value)	Ecrire la chaîne de caractères dans le tableau
JsonGenerator write(String name, BigDecimal value)	Ecrire la valeur numérique associée au nom fourni en paramètres
JsonGenerator write(String name, BigInteger value)	Ecrire la valeur numérique associée au nom fourni en paramètres
JsonGenerator write(String name, boolean value)	Ecrire la valeur booléenne associée au nom fourni en paramètres
JsonGenerator write(String name, double value)	Ecrire la valeur numérique associée au nom fourni en paramètres
JsonGenerator write(String name, int value)	Ecrire la valeur numérique associée au nom fourni en paramètres
JsonGenerator write(String name, JsonValue value)	Ecrire la valeur associée au nom fourni en paramètres
JsonGenerator write(String name, long value)	Ecrire la valeur numérique associée au nom fourni en paramètres
JsonGenerator write(String name, String value)	Ecrire la chaîne de caractères associée au nom fourni en paramètres
JsonGenerator writeEnd()	Ecrire un marqueur de fin pour l'élément courant
JsonGenerator writeNull()	Ecrire la valeur JSON null dans le tableau
JsonGenerator writeNull(String name)	Ecrire la valeur JSON null associée au nom fourni en paramètres
JsonGenerator writeStartArray()	Ecrire un marqueur de début de tableau

JsonGenerator writeStartArray(String name)	Ecrire un marqueur de début de tableau associé au nom fourni en paramètres
JsonGenerator writeStartObject()	Ecrire un marqueur de début d'objet dans le tableau
JsonGenerator writeStartObject(String name)	Ecrire un marqueur de début d'objet associé au nom fourni en paramètres

Pour obtenir une instance de type JsonGenerator, il y a deux solutions :

- Invoquer une des surcharges de la méthode createGenerator() de la classe Json qui attend en paramètre un objet de type InputStream ou Reader
- Obtenir une instance de type JsonParserFactory en invoquant la méthode Json.createGeneratorFactory() et invoquer la méthode createGenerator() sur cette instance. Cette façon de faire permet de configurer l'instance

Les méthodes write() mettent en oeuvre le principe de fluent API ce qui permet de chaîner leurs invocations.

Exemple :

```
    StringWriter sw = new StringWriter();
    JsonGenerator jsonGen = Json.createGenerator(sw);
    jsonGen.writeStartArray()
           .writeStartObject()
           .write("nom", "nom1")
           .write("prenom", "prenom1")
           .write("taille", "175")
           .writeEnd()
           .writeStartObject()
           .write("nom", "nom2")
           .write("prenom", "prenom2")
           .write("taille", "183")
           .writeEnd()
           .writeEnd()
           .close();
    String doc = sw.toString();
    System.out.println(doc);
```

Résultat :

```
[{"nom":"nom1","prenom":"prenom1","taille":"175"},
{"nom":"nom2","prenom":"prenom2","taille":"183"}]
```

L'utilisation de cette interface est similaire à celle de l'interface XMLStreamWriter de l'API Stax.

Il est important d'invoquer explicitement la méthode close() ou implicitement en utilisant une instruction try with resources à partir de Java 7.

57.3. L'API Object Model

L'API Object Model est de plus haut niveau, en permettant notamment :

- de créer un objet ou un graphe d'objets à partir de leur représentation JSON
- de naviguer dans le graphe d'objets et de le modifier
- de sérialiser les objets du graphe dans leur représentation JSON

Plusieurs classes permettent de faciliter la manipulation de ces objets :

- JsonObjectBuilder : permet de créer des instances de type JsonObject et JsonArray
- JsonReader : permet de lire un document JSON
- JsonWriter : permet d'écrire un document JSON

57.3.1. Les classes qui encapsulent un élément d'un document Json

L'API Object Model propose plusieurs interfaces qui définissent les différentes fonctionnalités des éléments d'un document JSON :

- JsonObject : un objet JSON
- JsonArray : un tableau JSON
- JsonString : une valeur de type chaîne de caractères
- JsonNumber : une valeur de type numérique

57.3.1.1. L'interface JsonValue

L'interface JsonValue définit les fonctionnalités d'une classe qui encapsule de manière immuable une valeur d'un document JSON.

Elle possède plusieurs interfaces filles : JsonStructure, JsonArray, JsonObject, JsonString et JsonNumber.

Elle définit trois valeurs particulières :

- JsonValue.TRUE
- JsonValue.FALSE
- JsonValue.NULL

Elle possède la classe interne JsonValue.ValueType qui est une énumération des différents types de valeur :

Enumération	Rôle
ARRAY	la valeur est un tableau
FALSE	la valeur false
NULL	la valeur null
NUMBER	la valeur est un numérique
OBJECT	la valeur est un objet
STRING	la valeur est une chaîne de caractères
TRUE	la valeur true

Elle définit plusieurs méthodes :

Méthode	Rôle
JsonValue.ValueType getValueType()	Renvoyer le type de la valeur
String toString()	Renvoyer une représentation sous forme d'une chaîne de caractères de la valeur

57.3.1.2. L'interface JsonNumber

L'interface JsonNumber définit les méthodes pour une classe qui encapsule de manière immuable une valeur numérique d'un document JSON.

Méthode	Rôle
BigDecimal bigDecimalValue()	Renvoyer la valeur sous la forme d'un BigDecimal

BigInteger bigIntegerValue()	Renvoyer la valeur sous la forme d'un BigInteger
BigInteger bigIntegerValueExact()	Renvoyer la valeur sous la forme d'un BigInteger
double doubleValue()	Renvoyer la valeur sous la forme d'un double
boolean equals(Object obj)	Comparer l'égalité entre l'instance courante et celle fournie en paramètre
int intValue()	Renvoyer la valeur sous la forme d'un entier de type double
int intValueExact()	Renvoyer la valeur sous la forme d'un entier de type double
boolean isIntegral()	Renvoyer un boolean qui précise si la valeur est entière
long longValue()	Renvoyer la valeur sous la forme d'un entier de type long
long longValueExact()	Renvoyer la valeur sous la forme d'un entier de type long
String toString()	Renvoyer une représentation sous forme d'une chaîne de caractères de la valeur

57.3.1.3. L'interface JsonString

L'interface JsonString définit les méthodes pour une classe qui encapsule de manière immuable une valeur d'un document JSON qui est une chaîne de caractères.

Méthode	Rôle
boolean equals(Object obj)	Comparer l'égalité entre l'instance courante et celle fournie en paramètre
CharSequence getChars()	Renvoyer une séquence de caractères de la valeur
String getString()	Renvoyer la valeur

57.3.1.4. L'interface JsonStructure

L'interface JsonStructure est l'interface mère des interfaces JsonObject et JsonArray. Elle hérite de l'interface JsonValue.

57.3.1.5. L'interface JsonObject

L'interface javax.json.JsonObjet définit les méthodes d'une classe qui encapsule un objet JSON immuable. Un objet JSON est composé de paires clé/valeur.

Les valeurs encapsulées dans un JsonObject peuvent être :

- une instance de type JsonString, JsonNumber, JsonObjet, JsonArray
- une des constantes : JsonValue.TRUE, JsonValue.FALSE, JsonValue.NULL

Elle hérite des interfaces JsonStructure et Map<String, JsonValue>.

Pour obtenir une instance de type JsonReader, il faut soit :

- invoquer la méthode readObject() d'une instance de type JsonReader
- invoquer la méthode build() d'une instance de type JsonObjectBuilder

L'interface propose plusieurs méthodes pour obtenir les différentes valeurs encapsulées dans l'instance :

Méthode	Rôle
boolean getBoolean(String name)	Renvoyer la valeur booléenne associée à la clé
boolean getBoolean(String name, boolean defaultValue)	Renvoyer la valeur booléenne associée à la clé avec une valeur par défaut si la clé n'est pas trouvée

int getInt(String name)	Renvoyer la valeur entière associée à la clé : elle invoque la méthode getJsonNumber(name).intValue()
int getInt(String name, int defaultValue)	Renvoyer la valeur entière associée à la clé avec une valeur par défaut si la clé n'est pas trouvée
JsonArray getJsonArray(String name)	Renvoyer l'instance de type JsonArray encapsulant les données du tableau associées à la clé
JsonNumber getJsonNumber(String name)	Renvoyer l'instance de type JsonNumber encapsulant la valeur numérique associée à la clé
JsonObject getJsonObject(String name)	Renvoyer l'instance de type JsonObject encapsulant les données de l'objet associées à la clé
JsonString getJsonString(String name)	Renvoyer la chaîne de caractères associée à la clé
String getString(String name)	Renvoyer la chaîne de caractères associée à la clé : elle invoque la méthode getJsonString(name).getString()
String getString(String name, String defaultValue)	Renvoyer la chaîne de caractères associée à la clé avec la valeur par défaut si la clé n'est pas trouvée
boolean isNull(String name)	Renvoyer un booléen qui précise si la valeur associée à la clé est JsonValue.NULL

L'interface JsonObject hérite de l'interface Map mais les données qu'elle encapsule sont immuables : toute tentative de modification de la collection lèvera une exception de type UnsupportedException.

L'itération sur les éléments contenus dans le JsonObjet se fait dans l'ordre dans lequel les éléments ont été ajoutés.

57.3.1.6. L'interface JsonArray

L'interface javax.json.JsonArray définit les méthodes pour une classe qui encapsule un tableau immuable de valeurs JSON. Elle hérite des interfaces JsonStructure et List<JsonValue>.

Pour obtenir une instance de type JsonReader, il faut soit :

- invoquer la méthode readArray() d'une instance de type JsonReader
- invoquer la méthode build() d'une instance de type JsonArrayBuilder

Les valeurs encapsulées dans un JsonObject peuvent être :

- une instance de type JsonString, JsonNumber, JsonObjet, JsonArray
- une des constantes : JsonValue.TRUE, JsonValue.FALSE, JsonValue.NULL

L'interface JsonArray hérite de l'interface List mais les données qu'elle encapsule sont immuables : oute tentative de modification de la collection lèvera une exception de type UnsupportedException.

L'interface propose plusieurs méthodes pour obtenir les différentes valeurs encapsulées dans l'instance :

Méthode	Rôle
boolean getBoolean(int index)	Renvoyer la valeur booléenne de l'index fourni
boolean getBoolean(int index, boolean defaultValue)	Renvoyer la valeur booléenne de l'index fourni avec une valeur par défaut si la clé n'est pas trouvée
int getInt(int index)	Renvoyer la valeur entière de l'index fourni : elle invoque la méthode getJsonNumber(index).intValue()
int getInt(int index, int defaultValue)	Renvoyer la valeur entière de l'index fourni avec une valeur par défaut si la clé n'est pas trouvée

JsonArray getJsonArray(int index)	Renvoyer l'instance de type JsonArray encapsulant les données du tableau pour l'index fourni
JsonNumber getJsonNumber(int index)	Renvoyer l'instance de type JsonNumber encapsulant la valeur numérique pour l'index fourni
JsonObject getJsonObject(int index)	Renvoyer l'instance de type JsonObject encapsulant les données de l'objet pour l'index fourni
JsonString getJsonString(int index)	Renvoyer la chaîne de caractères pour l'index fourni
String getString(int index)	Renvoyer la chaîne de caractères pour l'index fourni : elle invoque la méthode getJsonString(index).getString()
String getString(int index, String defaultValue)	Renvoyer la chaîne de caractères pour l'index fourni avec la valeur par défaut si la clé n'est pas trouvée
<T extends JsonValue> List<T> getValuesAs(Class<T> clazz)	Renvoyer une vue sous la forme d'une List typée avec la classe fournie en paramètre
boolean isNull(int index)	Renvoyer un booléen qui précise si la valeur à l'index fourni en paramètre est JsonValue.NULL

57.3.2. L'interface JsonReader

L'interface JsonReader permet de lire un document JSON.

Plusieurs méthodes permettent d'obtenir le premier élément du document JSON lu ou de le fermer :

Méthode	Rôle
void close()	Terminer les traitements et libérer les éventuelles ressources
JsonStructure read()	Renvoyer un objet de type JsonArray ou JsonObject selon le contenu du document
JsonObject readObject()	Renvoyer un objet de type JsonObject qui encapsule l'objet du document
JsonArray readArray()	Renvoyer un objet de type JsonArray qui encapsule le tableau du document

Ces méthodes ne doivent être invoquées qu'une seule fois pour une même instance.

La méthode close() doit être invoquée explicitement ou implicitement grâce à une instruction try with ressources de Java 7 pour libérer d'éventuelles ressources.

Pour créer une instance de type JsonReader, il y a deux possibilités :

- invoquer la méthode createReader() de la classe Json qui est une fabrique pour créer une instance de la classe JsonReader. Elle attend en paramètre le document source à lire sous la forme d'une instance de type InputStream ou Reader.
- obtenir une instance de type JsonParserFactory en invoquant la méthode Json.createGeneratorFactory() et invoquer la méthode createGenerator() sur cette instance. Cette façon de faire permet de configurer l'instance.

Exemple (code Java 7) :

```
            String document = "[{\n"
                    + "\"nom\":\"nom1\", \"prenom\": \"prenom1\", \"taille\":175\n"
                    + "},\n"
                    + "{\n"
                    + "\"nom\":\"nom2\", \"prenom\": \"prenom2\",\"taille\": 183\n"
                    + "}\n"
                    + "]";
            try (JsonReader reader = Json.createReader(new StringReader(document))) {
                JsonArray array = reader.readArray();
                JsonObject obj = array.getJsonObject(1);
                String nom = obj.getJsonString("nom").getString();
            }
```

L'objet de type JsonObject, JsonArray ou JsonStructure encapsule l'élément racine du graphe d'objets créé suite à la lecture. Cet objet peut être utilisé pour parcourir le graphe ou pour écrire sa représentation JSON.

57.3.3. Le parcours du modèle objet

Il est possible de parcourir ou d'obtenir un élément particulier du graphe d'objets en utilisant les différentes méthodes des objets qui le compose.

La méthode getValueType() de la classe JsonValue permet de déterminer le type de l'élément en cours de traitement. Il suffit alors de faire un cast vers le type concerné pour avoir accès aux méthodes dédiées de ce type.

Exemple (code Java 7) :

```
//
...
      String document = "[{\n"
              + "\"nom\":\"nom1\", \"prenom\": \"prenom1\",\"taille\": 175\n"
              + "},\n"
              + "{\n"
              + "\"nom\":\"nom2\", \"prenom\": \"prenom2\",\"taille\": 183\n"
              + "}\n"
              + "]";
      try (JsonReader reader = Json.createReader(new StringReader(document))) {
        JsonArray array = reader.readArray();
        System.out.println("Debut du parcours du modele");
        parcourirModele(array, null, 0);
        System.out.println("Fin du parcours du modele");
      } catch (Exception e) {
        e.printStackTrace();
      }
//
...
  public static void parcourirModele(final JsonValue element,
    final String cle, final int niveau) {
    String indentation = Strings.repeat("..", niveau);
    int niveauSuivant = niveau+1;
    if (cle != null) {
      System.out.print(indentation+"Key " + cle + ": ");
    }
    switch (element.getValueType()) {
      case OBJECT:
        System.out.println(indentation+"Objet");
        JsonObject object = (JsonObject) element;
        for (String nom : object.keySet()) {
          parcourirModele(object.get(nom), nom, niveauSuivant);
        }
        break;
      case ARRAY:
        System.out.println(indentation+"Tableau");
        JsonArray array = (JsonArray) element;
        for (JsonValue val : array) {
          parcourirModele(val, null, niveauSuivant);
        }
        break;
      case STRING:
        JsonString st = (JsonString) element;
        System.out.println(" String " + st.getString());
        break;
      case NUMBER:
        JsonNumber num = (JsonNumber) element;
        System.out.println(" Nombre " + num.toString());
        break;
      case TRUE:
      case FALSE:
      case NULL:
        System.out.println(" " +element.getValueType().toString());
        break;
    }
  }
```

Résultat :

```
Debut du parcours du modele
Tableau
..Objet
....Key
nom: String nom1
....Key
prenom: String prenom1
....Key
taille: Nombre 175
..Objet
....Key
nom: String nom2
....Key
prenom: String prenom2
....Key
taille: Nombre 183
Fin
du parcours du modele
```

La méthode s'appelle récursivement si l'élément est un JsonArray indépendant ou contenu dans un JsonObject permettant ainsi le parcours de tous les éléments.

La méthode keySet() de la classe JsonObject renvoie une collection des clés qui composent l'objet. La méthode get() de la classe JsonObject permet d'obtenir la valeur pour la clé passée en paramètre.

57.3.4. L'interface JsonArrayBuilder

L'interface JsonArrayBuilder définit des méthodes pour faciliter la création d'objets de type JsonArray qui encapsulent un tableau JSON.

Cette interface repose sur le motif de conception builder : elle définit donc plusieurs méthodes pour ajouter des éléments au tableau et pour obtenir l'objet :

Méthode	Rôle
JsonArrayBuilder add(BigDecimal value)	Ajouter une nouvelle valeur au tableau
JsonArrayBuilder add(BigInteger value)	Ajouter une nouvelle valeur au tableau
JsonArrayBuilder add(String name, boolean value)	Ajouter une nouvelle valeur au tableau
JsonArrayBuilder add(double value)	Ajouter une nouvelle valeur au tableau
JsonArrayBuilder add(int value)	Ajouter une nouvelle valeur au tableau
JsonArrayBuilder add(JsonArrayBuilder builder)	Ajouter un autre tableau JSON (encapsulé dans une instance de type JsonArrayBuilder) au tableau
JsonArrayBuilder add(JsonObjectBuilder builder)	Ajouter un nouvel objet JSON (encapsulé dans une instance de type JsonObjectBuilder) au tableau
JsonArrayBuilder add(JsonValue value)	Ajouter une nouvelle valeur au tableau
JsonArrayBuilder add(long value)	Ajouter une nouvelle valeur au tableau
JsonArrayBuilder add(String value)	Ajouter une nouvelle valeur au tableau
JsonArrayBuilder addNull(String name)	Ajouter une valeur null au tableau

JsonObject build()	Retourner l'instance encapsulant le tableau contenant toutes les valeurs ajoutées

Toutes les surcharges de la méthode add() utilisent le modèle fluent : elles renvoient l'instance elle-même du JsonArrayBuilder ce qui permet de chaîner leurs invocations.

Pour obtenir une instance de type JsonArrayBuilder, il faut utiliser la méthode createArrayBuilder() de la classe Json qui est une fabrique. Par défaut, elle crée un tableau JSON vide et propose des surcharges de la méthode add() pour ajouter des valeurs au document et la méthode build() pour obtenir l'instance de type JsonArrayBuilder correspondante.

Exemple :

```
        JsonArray jsonArray = Json.createArrayBuilder()
                .add("valeur1")
                .add("valeur2")
                .add("valeur3")
                .build();
```

Résultat :

```
["valeur1","valeur2","valeur3"]
```

57.3.5. L'interface JsonObjectBuilder

L'interface JsonObjectBuilder définit des méthodes pour faciliter la création d'un objet de type JsonObject qui encapsule un objet JSON.

Cette interface repose sur le motif de conception builder : elle définit donc plusieurs méthodes pour ajouter des éléments à l'objet et pour obtenir l'objet:

Méthode	Rôle
JsonObjectBuilder add(String name, BigDecimal value)	Ajouter un nouvel élément dont le nom et la valeur sont fournis en paramètres
JsonObjectBuilder add(String name, BigInteger value)	Ajouter un nouvel élément dont le nom et la valeur sont fournis en paramètres
JsonObjectBuilder add(String name, boolean value)	Ajouter un nouvel élément dont le nom et la valeur sont fournis en paramètres
JsonObjectBuilder add(String name, double value)	Ajouter un nouvel élément dont le nom et la valeur sont fournis en paramètres
JsonObjectBuilder add(String name, int value)	Ajouter un nouvel élément dont le nom et la valeur sont fournis en paramètres
JsonObjectBuilder add(String name, JsonArrayBuilder builder)	Ajouter un nouvel élément dont le nom et le tableau JSON (encapsulé dans une instance de type JsonArrayBuilder) associé sont fournis en paramètres
JsonObjectBuilder add(String name, JsonObjectBuilder builder)	Ajouter un nouvel élément dont le nom et l'objet JSON (encapsulé dans une instance de type JsonObjectBuilder) associé sont fournis en paramètres
JsonObjectBuilder add(String name, JsonValue value)	Ajouter un nouvel élément dont le nom et la valeur sont fournis en paramètres
JsonObjectBuilder add(String name, long value)	Ajouter un nouvel élément dont le nom et la valeur sont fournis en paramètres
JsonObjectBuilder add(String name, String value)	Ajouter un nouvel élément dont le nom et la valeur sont fournis en paramètres
JsonObjectBuilder addNull(String name)	Ajouter un nouvel élément dont le nom est fourni en paramètre avec la valeur null

JsonObject build()	Retourner l'instance contenant tous les éléments ajoutés

Toutes les surcharges de la méthode add() utilisent le modèle fluent : elles renvoient l'instance elle-même du JsonObjectBuilder ce qui permet de chaîner leurs invocations.

Pour obtenir une instance de type JsonObjectBuilder, il faut utiliser la méthode createObjectBuilder() de la classe Json qui est une fabrique. Par défaut, elle crée un objet JSON vide et propose des surcharges de la méthode add() pour ajouter des éléments et la méthode build() pour obtenir l'instance de type JsonObjectBuilder correspondante.

Exemple :

```
        JsonObject jsonObj = Json.createObjectBuilder()
                .add("nom", "nom1")
                .add("prenom", "prenom1")
                .add("taille", "175")
                .build();
```

Résultat :

```
{"nom":"nom1","prenom":"prenom1","taille":"175"}
```

L'objet JSON peut être plus complexe en imbriquant différents objets et tableaux.

Exemple :

```
        JsonObject jsonObj = Json.createObjectBuilder()
                .add("nom", "groupe1")
                .add("personnes", Json.createArrayBuilder()
                  .add(Json.createObjectBuilder()
                    .add("nom", "nom1")
                    .add("prenom", "prenom1")
                    .add("taille", "175"))
                  .add(Json.createObjectBuilder()
                    .add("nom", "nom1")
                    .add("prenom", "prenom1")
                    .add("taille", "175")))
                .build();
```

Résultat :

```
{"nom":"groupe1",
"personnes":[{"nom":"nom1","prenom":"prenom1","taille":"175"},
      {"nom":"nom1","prenom":"prenom1","taille":"175"}]}
```

57.3.6. L'interface JsonWriter

L'interface javax.json.JsonWriter définit des méthodes pour permettre l'envoi dans un flux de caractères d'instances de types JsonObject, JsonArray ou JsonStructure.

Méthode	Rôle
void close()	Fermer le writer et libérer les éventuelles ressources associées
void write(JsonStructure value)	Ecrire l'objet ou le tableau JSON passé en paramètre
void writeArray(JsonArray array)	Ecrire le tableau JSON passé en paramètre
void writeObject(JsonObject object)	Ecrire l'objet JSON passé en paramètre

Pour obtenir une instance de type JsonWriter, il faut soit :

- invoquer la méthode createWriter() de la classe Json qui est une fabrique pour créer une instance de la classe JsonWriter. Elle attend en paramètre une instance de type InputStream ou Reader
- obtenir une instance de type JsonWriterFactory en invoquant la méthode Json.createWriterFactory() et invoquer la méthode createWriter() sur cette instance. Cette façon de faire permet de configurer l'instance.

Exemple (code Java 7) :

```
                JsonObject jsonObj = Json.createObjectBuilder()
                        .add("nom", "nom1")
                        .add("prenom", "prenom1")
                        .add("taille", "175")
                        .build();
                StringWriter stringWriter = new StringWriter();
                try (JsonWriter writer = Json.createWriter(stringWriter)) {
                    writer.writeObject(jsonObj);
                    System.out.println(stringWriter.toString());
                } catch (Exception e) {
                    e.printStackTrace();
                }
```

La méthode close() doit être invoquée explicitement ou implicitement grâce à une instruction try with ressources de Java 7 pour fermer le flux associé.

57.3.7. Les interfaces JsonXXXFactory

L'API définit plusieurs interfaces qui définissent des fonctionnalités pour des fabriques d'instances :

- JsonBuilderFactory : pour créer des instances de type JsonObjectbuilder ou JsonArrayBuilder
- JsonReaderFactory : pour créer des instances de type JsonReader
- JsonWriterFactory : pour créer des instances de type JsonWriter

Toutes leurs méthodes sont threads safe. La classe Json propose aussi des méthodes pour créer des instances mais si plusieurs instances d'un même type doivent être créées alors il est préférable d'utiliser la fabrique correspondante.

L'interface JsonBuilderFactory définit les méthodes d'une fabrique permettant de créer des instances de type JsonObjectBuilder et JsonArrayBuilder.

Elle définit plusieurs méthodes :

Méthode	Rôle
JsonArrayBuilder createArrayBuilder()	Créer une instance de type JsonArrayBuilder
JsonObjectBuilder createObjectBuilder()	Créer une instance de type JsonObjectBuilder
Map<String,?> getConfigInUse()	Renvoyer une collection immuable de type Map qui contient les propriétés de configuration utilisées pour créer les instances

Pour obtenir une instance de type JsonBuilderFactory, il faut utiliser la méthode statique createBuilderFactory() de la classe Json.

L'interface JsonReaderFactory définit les méthodes d'une fabrique permettant de créer des instances de type JsonReader.

Elle définit plusieurs méthodes :

Méthode	Rôle
JsonReader createReader(InputStream in)	Créer une instance de type JsonReader sur un flux d'octets

JsonReader createReader(InputStream in, Charset charset)	Créer une instance de type JsonReader sur un flux d'octets en utilisant le jeu de caractères fourni en paramètre
JsonReader createReader(Reader reader)	Créer une instance de type JsonReader sur un flux de type caractères
Map<String,?> getConfigInUse()	Renvoyer une collection immuable de type Map qui contient les propriétés de configuration utilisées pour créer les instances

Pour obtenir une instance de type JsonReaderFactory, il faut utiliser la méthode statique createReaderFactory() de la classe Json.

L'interface JsonReaderFactory définit les méthodes d'une fabrique permettant de créer des instances de type JsonReader.

Elle définit plusieurs méthodes :

Méthode	Rôle
JsonWriter createWriter(OutputStream out)	Créer une instance de type JsonWriter sur un flux d'octets
JsonWriter createWriter(OutputStream out, Charset charset)	Créer une instance de type JsonWriter sur un flux d'octets en utilisant le jeu de caractères fourni en paramètre
JsonWriter createWriter(Writer writer)	Créer une instance de type JsonWriter sur un flux de type caractères
Map<String,?> getConfigInUse()	Renvoyer une collection immuable de type Map qui contient les propriétés de configuration utilisées pour créer les instances

Pour obtenir une instance de type JsonWriterFactory, il faut utiliser la méthode statique createWriterFactory() de la classe Json.

58. JSON-B (Java API for JSON Binding)

Chapitre 58

Niveau :	🟩🟩🟨⬜⬜⬜ Intermédiaire
Version utilisée :	1.0

Il existe différentes solutions open source (Jackson, Genson, Gson, ...) pour réaliser le binding entre des documents JSON et des objets Java mais JSON-B propose une API standard pour permettre ses opérations.

Java API for JSON Binding (JSON-B) est spécifié dans la JSR 367.

Le site web officiel est à l'url https://javaee.github.io/jsonb-spec/

L'API JSON-B permet de réaliser des opérations de sérialisation et de désérialisation entre des documents JSON et des objets Java :

- La désérialisation est un traitement qui lit un document JSON pour construire un objet ou un graphe d'objets qui encapsule les données du document
- La sérialisation est le processus inverse qui permet de produire un document JSON à partir d'un objet

L'implémentation de référence de JSON-B 1.0 est Yasson

Les conversions de JSON-B se font avec un comportement par défaut mais il est possible de les personnaliser.

Ce chapitre contient plusieurs sections :

- ♦ La sérialisation/désérialisation d'un objet
- ♦ Le moteur JSON-B
- ♦ Le mapping par défaut
- ♦ La configuration du moteur JSON-B
- ♦ La personnalisation du mapping

58.1. La sérialisation/désérialisation d'un objet

La mise en oeuvre de JSON-B pour le binding de documents JSON est similaire à celle de JAX-B pour le binding de documents XML.

Des règles de conversions par défaut sont utilisées lors des opérations exécutées par JSON-B. Il est possible de personnaliser ces règles en utilisant des annotations ou un objet de type JsonbConfig pour des règles globales.

Les classes et interfaces de l'API JSON-B sont dans le package javax.json.bind et ses sous-packages.

Pour utiliser Yasson, l'implémentation de référence de JSON-B 1.0, en dehors d'un serveur d'applications Java EE 8, il faut ajouter plusieurs dépendances :

Exemple avec Maven :

```
  <dependencies>
```

```
    <!-- JSON-P API -->
    <dependency>
      <groupId>javax.json</groupId>
      <artifactId>javax.json-api</artifactId>
      <version>1.1</version>
    </dependency>

    <!-- JSON-B API -->
    <dependency>
      <groupId>javax.json.bind</groupId>
      <artifactId>javax.json.bind-api</artifactId>
      <version>1.0</version>
    </dependency>

    <!-- JSON-B RI -->
    <dependency>
      <groupId>org.eclipse</groupId>
      <artifactId>yasson</artifactId>
      <version>1.0</version>
      <scope>runtime</scope>
    </dependency>

    <!-- JSON-P RI -->
    <dependency>
      <groupId>org.glassfish</groupId>
      <artifactId>javax.json</artifactId>
      <version>1.1</version>
      <scope>runtime</scope>
    </dependency>
  </dependencies>
```

Il est ainsi possible de sérialiser/désérialiser de simples POJO.

Exemple (code Java 8) :

```
import java.time.LocalDate;

public class Personne {

  private String    nom;
  private String    prenom;
  private int       taille;
  private boolean   adulte;
  private LocalDate dateNaissance;

  // getters et setters
}
```

Pour sérialiser une instance, il suffit de la passer en paramètre de la méthode toJson() d'une instance de type Jsonb préalablement obtenue.

Exemple (code Java 8) :

```
import java.time.LocalDate;
import java.time.Month;

import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;

public class TestJsonB {

  public static void main(String[] args) {

    Personne pers = new Personne();
    pers.nom = "nom";
    pers.prenom = "prenom";
    pers.taille = 175;
    pers.adulte = true;
    pers.dateNaissance = LocalDate.of(1985, Month.AUGUST, 11);
    Jsonb jsonb = JsonbBuilder.create();
    String result = jsonb.toJson(pers);

    System.out.println(result);
  }
}
```

Résultat :

```
{"adulte":true,"dateNaissance":"1985-08-11","nom":"nom","prenom":"prenom","taille":175}
```

La désérialisation se fait aussi de manière très simple : il suffit de passer le document JSON en paramètre de la méthode fromJson() d'une instance de type Jsonb préalablement obtenue.

Exemple (code Java 8) :

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;

public class TestJsonB {

  public static void main(String[] args) {

    Personne pers = new Personne();
    Jsonb jsonb = JsonbBuilder.create();

    pers = jsonb.fromJson("{\"nom\":\"nom1\",\"prenom\":\"prenom1\",\"taille\":183}",
        Personne.class);
    System.out.println("nom=" + pers.nom);
    System.out.println("prenom=" + pers.prenom);
    System.out.println("taille=" + pers.taille);
  }
}
```

Résultat :

```
nom=nom1
prenom=prenom1
taille=183
```

58.2. Le moteur JSON-B

L'interface JsonbBuilder est le point d'entrée pour utiliser l'API JSON-B.

Elle définit plusieurs méthodes mettant en oeuvre le design pattern builder dont le but est de créer une instance de type Jsonb en fonction des paramètres et de la configuration fournie avant l'invocation de la méthode build() :

Méthode	Rôle
Jsonb build()	Obtenir l'instance à partir du builder
static Jsonb create()	Obtenir une nouvelle instance de Jsonb obtenue à partir du JsonbBuilder de l'implémentation par défaut
static Jsonb create(JsonbConfig config)	Créer une nouvelle instance en utilisant l'implémentation par défaut de JsonbBuilder et la configuration passée en paramètre
static JsonbBuilder newBuilder()	Obtenir une instance de type JsonbBuilder obtenue grâce au fournisseur par défaut
static JsonbBuilder newBuilder(String providerName)	Obtenir une instance de type JsonbBuilder obtenue grâce au fournisseur dont le nom est passé en paramètre

static JsonbBuilder newBuilder(JsonbProvider provider)	Obtenir une instance de type JsonbBuilder obtenue grâce au fournisseur passé en paramètre
JsonbBuilder withConfig(JsonbConfig config)	Préciser la configuration de l'instance qui sera obtenue
JsonbBuilder withProvider(javax.json.spi.JsonProvider jsonpProvider)	Préciser le fournisseur de l'implémentation de JSON-B à utiliser

Si le comportement par défaut du moteur JSON-B respecte les besoins, il n'y a aucune configuration particulière à utiliser. L'obtention d'une instance configurée par défaut peut facilement être réalisée en invoquant la méthode create() de la classe JsonbBuilder.

Exemple (code Java 8) :

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;

public class TestJsonB {

  public static void main(String[] args) {

    Jsonb jsonb = JsonbBuilder.create();
  }
}
```

Il est recommandé de n'avoir qu'une seule instance de type Jsonb par configuration de mapping d'autant que les instances de type Jsonb doivent être thread safe.

L'interface Jsonb permet de demander des sérialisations et des désérialisations au moteur JSON-B. Elle définit plusieurs surcharges de deux méthodes :

Méthode	Rôle

<T> T fromJson(String str, Class<T> type);
<T> T fromJson(String str, Type runtimeType);
<T> T fromJson(Reader reader, Class<T> type);
<T> T fromJson(Reader reader, Type runtimeType);
<T> T fromJson(InputStream stream, Class<T> type);
<T> T fromJson(InputStream stream, Type runtimeType);
	Convertir un document JSON en un objet Java

String toJson(Object object);
String toJson(Object object, Type runtimeType);
void toJson(Object object, Writer writer);
void toJson(Object object, Type runtimeType, Writer writer);
void toJson(Object object, OutputStream stream);
void toJson(Object object, Type runtimeType, OutputStream stream);
	Convertir un objet Java en un document JSON

L'interface Jsonb hérite de l'interface AutoCloseable.

Le moteur JSON-B utilise un ensemble de règles pour réaliser les mapping.

58.3. Le mapping par défaut

Les implémentations de JSON-B doivent assurer la correspondance pour des documents JSON qui respectent la RFC 7159 (The JavaScript Object Notation (JSON) Data Interchange Format).

Le document JSON issue d'une sérialisation doit donc respecter la RFC 7159 et être encodé en UTF-8.

Le mapping par défaut ne requiert ni configuration ni annotation particulière. Ce mapping par défaut comporte des règles pour la sérialisation/désérialisation pour les principaux types :

- Les types primitifs au travers de leur wrapper
- Certains types du JDK
- Les dates
- Les tableaux
- Les collections
- Les énumérations
- Des classes de JSON-P
- Les classes qui encapsulent des données des types précédents

58.3.1. Le mapping par défaut des types communs

Une implémentation de JSON-B doit prendre en charge les types de base et les types spécifiques du JDK ci-dessous :

Type de base	Type spécifique du JDK
java.lang.String	java.math.BigInteger
java.lang.Character	java.math.BigDecimal
java.lang.Byte (byte)	java.net.URL
java.lang.Short (short)	java.net.URI
java.lang.Integer (int)	java.util.Optional
java.lang.Long (long)	java.util.OptionalInt
java.lang.Float (float)	java.util.OptionalLong
java.lang.Double (double)	java.util.OptionalDouble
java.lang.Boolean (boolean)	
java.lang.Number	

Exemple (code Java 8) :

```
public class Primitives {

  public String  unString  = "string";
  public char    unChar    = 'a';
  public byte    unByte    = 1;
  public short   unShort   = 2;
  public int     unInt     = 3;
  public long    unLong    = 4L;
  public float   unFloat   = 5.1f;
  public double  unDouble  = 6.2;
  public boolean unBoolean = true;
}
```

Exemple (code Java 8) :

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import javax.json.bind.JsonbConfig;

public class TestJsonB {

  public static void main(String[] args) {

    JsonbConfig config = (new JsonbConfig()).withFormatting(true);
    Jsonb jsonb = JsonbBuilder.create(config);
    Primitives primitives = new Primitives();
    String result = jsonb.toJson(primitives);
    System.out.println(result);
  }
}
```

Résultat :

```
{
    "unString": "string",
    "unBoolean": true,
    "unByte": 1,
    "unChar": "a",
    "unDouble": 6.2,
    "unFloat": 5.1,
    "unInt": 3,
    "unLong": 4,
    "unShort": 2
}
```

Lors de la sérialisation, la valeur d'une instance de type BigInteger, BigDecimal, URL et URI est obtenue en invoquant leur méthode toString(). La désérialisation utilise leur constructeur qui attend une chaîne de caractères en paramètre.

La valeur sérialisée pour les Optionalxxx est la valeur encapsulée si elle est présente ou null si elle est vide.

Exemple (code Java 8) :

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import javax.json.bind.JsonbConfig;

public class TestJsonB {

  public static void main(String[] args) {

    JsonbConfig config = (new JsonbConfig()).withFormatting(true);
    Jsonb jsonb = JsonbBuilder.create(config);
    Objets objets = new Objets();
    String result = jsonb.toJson(objets);
    System.out.println(result);
  }
}
```

Exemple (code Java 8) :

```
import java.math.BigDecimal;
import java.math.BigInteger;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.OptionalInt;

public class Objets {

  public BigInteger  bigInteger    = BigInteger.valueOf(1000L);
  public BigDecimal  bigDecimal    = new BigDecimal("1.2");
  public OptionalInt optionalInt   = OptionalInt.of(1);
  public OptionalInt optionalEmpty = OptionalInt.empty();
  public URL         url           = null;

  public Objets() {
    try {
      url = new URL("file://c:/temp");
    } catch (MalformedURLException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
{
    "bigDecimal": 1.2,
    "bigInteger": 1000,
    "optionalInt": 1,
    "url": "file://c:/temp"
}
```

Une implémentation de JSON-B doit prendre en charge les types temporels en sérialisant leur valeur selon le tableau ci-dessous :

java.util.Date, java.util.Calendar, java.util.GregorianCalendar	ISO_DATE ou ISO_DATE_TIME selon la présence d'informations sur l'heure
java.util.TimeZone, java.util.SimpleTimeZone	NormalizedCustomId (cf Javadoc de la classe TimeZone)
java.time.Instant	ISO_INSTANT
java.time.LocalDate	ISO_LOCAL_DATE
java.time.LocalTime	ISO_LOCAL_TIME
java.time.LocalDateTime	ISO_LOCAL_DATE_TIME
java.time.ZonedDateTime	ISO_ZONED_DATE_TIME
java.time.OffsetDateTime	ISO_OFFSET_DATE_TIME
java.time.OffsetTime	ISO_OFFSET_TIME
java.time.ZoneId	NormalizedZoneId (cf Javadoc de la classe ZoneId)
java.time.ZoneOffset	NormalizedZoneId (cf Javadoc de la classe ZoneOffset)
java.time.Duration	Représentation en secondes définie par l'ISO 8601
java.time.Period	Représentation période définie par l'ISO 8601

Exemple (code Java 8) :

```java
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.time.Duration;
import java.time.Instant;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.Month;
import java.time.OffsetDateTime;
import java.time.Period;
import java.time.ZonedDateTime;
import java.util.Calendar;
import java.util.Date;

public class Dates {

  SimpleDateFormat      sdf            = new SimpleDateFormat("dd.MM.yyyy");
  public Date           date           = null;

  public Calendar       calendar       = Calendar.getInstance();

  public LocalDate      localDate      = LocalDate.now();
  public LocalTime      localTime      = LocalTime.now();
  public LocalDateTime  localDateTime  = LocalDateTime.now();

  public ZonedDateTime  zonedDatetime  = ZonedDateTime.now();
  public OffsetDateTime offsetDateTime = OffsetDateTime.now();

  public Duration       duree          = Duration.ofHours(1).plusMinutes(30);
  public Instant        instant        = Instant.parse("2017-08-27T12:00:00Z");

  public Period         periode        = Period.between(LocalDate.of(2017,
      Month.DECEMBER, 25), LocalDate.of(2016, Month.DECEMBER, 25));

  public Dates() {
    calendar.clear();
    calendar.set(2017, 7, 26);

    try {
      date = sdf.parse("15.11.2016");
    } catch (ParseException e) {
      e.printStackTrace();
    }
  }
}
```

Exemple (code Java 8) :

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import javax.json.bind.JsonbConfig;

public class TestJsonB {

  public static void main(String[] args) {
    JsonbConfig config = (new JsonbConfig()).withFormatting(true);
    Jsonb jsonb = JsonbBuilder.create(config);
    Dates dates = new Dates();
    String result = jsonb.toJson(dates);
    System.out.println(result);
  }
}
```

Résultat :

```
{
    "calendar": "2017-08-26+02:00",
    "date": "2016-11-14T23:00:00Z[UTC]",
    "duree": "PT1H30M",
    "instant": "2017-08-27T12:00:00Z",
    "localDate": "2017-08-27",
    "localDateTime": "2017-08-27T15:24:48.241",
    "localTime": "15:24:48.241",
    "offsetDateTime": "2017-08-27T15:24:48.242+02:00",
    "periode": "P-1Y",
    "zonedDatetime": "2017-08-27T15:24:48.241+02:00[Europe/Paris]"
}
```

Le mapping par défaut d'une énumération suit plusieurs règles :

- La sérialisation utilise la méthode name() pour obtenir la valeur
- La désérialisation utilise la méthode valueOf()

Exemple (code Java 8) :

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;

public class TestJsonB {

  enum Taille { PETIT, MOYEN, GRAND };

  public static void main(String[] args) {

    Jsonb jsonb = JsonbBuilder.create();
    String result = jsonb.toJson(Taille.GRAND);
    System.out.println(result);
  }
}
```

Résultat :

```
"GRAND"
```

58.3.2. Le mapping par défaut d'un type quelconque

Le mapping par défaut d'un type quelconque suit plusieurs règles :

- Les classes imbriquées public et protected sont supportées
- Les classes anonymes sont uniquement sérialisées
- Pour la désérialisation, les classes doivent avoir un constructeur par défaut (sans paramètre)
- L'héritage est supporté

Le mapping par défaut d'un champ suit plusieurs règles :

- Les champs finaux sont sérialisés
- Les champs static ne sont pas sérialisés
- Les champs transient ne sont pas sérialisés
- Les champs null ne sont pas sérialisés
- L'ordre de sérialisation des champs est l'ordre lexicographique. Les champs de la classe parente sont sérialisés avant ceux de la classe

Une implémentation de JSON-B doit respecter un certain ordre pour accéder ou alimenter des valeurs lors des opérations.

Pour la sérialisation :

- Le getter public d'un champ
- Un champ public sans getter
- Un getter sans champ

Pour la désérialisation :

- Le setter public d'un champ
- Un champ public sans setter
- Un setter sans champ

Exemple (code Java 8) :

```
public class Donnees {

  public final int     champPublicFinal             = 1;
  private final int    champPublicPrivate           = 2;
  public static int    champPublicStatic            = 3;
  public int           champPublicSansGetter        = 4;
  public int           champPublicAvecGetterPrivate = 5;
  private int          champPrivateSansGetter       = 6;
  private int          champPrivateAvecGetter       = 7;
  public Integer       champPublicNull              = null;
  public transient int champPublicTransient         = 8;

  public int getSansChamp() {
    return 9;
  };

  public int getChampPrivateAvecGetter() {
    return 10;
  };

  private int getChampPublicAvecGetterPrivate() {
    return 11;
  }
}
```

Exemple (code Java 8) :

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;

public class TestJsonB {

  public static void main(String[] args) {
    Jsonb jsonb = JsonbBuilder.create();
    Donnees donnees = new Donnees();
    String result = jsonb.toJson(donnees);
    System.out.println(result);
  }
}
```

Résultat :

```
{"champPrivateAvecGetter":10,"champPublicFinal":1,"champPublicSansGetter":4,"sansChamp":9}
```

58.3.3. Le mapping par défaut des tableaux et des collections

Le mapping par défaut prend en compte les tableaux mono ou multi-dimensions et les principales classes de bases de l'API Collection : Collection, Map, Set, HashSet, NavigableSet, SortedSet, TreeSet, LinkedHashSet, HashMap, NavigableMap, SortedMap, TreeMap, LinkedHashMap, List, ArrayList, LinkedList, Deque, ArrayDeque, Queue, PriorityQueue.

Exemple (code Java 8) :

```
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Deque;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;

public class Valeurs {

  public int[]                tableau = new int[5];
  public int[][]              tab2d   = { { 2, 4, 1 }, { 6, 8 }, { 7, 3, 6, 5 } };
  public List<Integer>        list    = new ArrayList<>();
  public Map<Integer, String> map     = new HashMap<>();
  public Set<Integer>         set     = new HashSet<>();
  public Deque<Integer>       deque   = new ArrayDeque<>();
}
```

Exemple (code Java 8) :

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;

public class TestJsonB {

  public static void main(String[] args) {
    Jsonb jsonb = JsonbBuilder.create();

    Valeurs valeurs = new Valeurs();
    valeurs.tableau[0] = 1;
    valeurs.tableau[1] = 2;
    valeurs.list.add(3);
    valeurs.list.add(null);
    valeurs.list.add(4);
    valeurs.map.put(1, "un");
    valeurs.map.put(2, "deux");
    valeurs.set.add(5);
    valeurs.set.add(5);
    valeurs.set.add(6);
    valeurs.deque.push(7);
    valeurs.deque.push(8);
    String result = jsonb.toJson(valeurs);
    System.out.println(result);
  }
}
```

Résultat :

```
{"deque":[8,7],"list":[3,null,4],"map":{"1":"un",
"2":"deux"},"set":[5,6],"tab2d":[[2,4,1],[6,8],[7,3,6,5]],"tableau":[1,2,0,0,0]}
```

Lors de la désérialisation d'un tableau JSON, il est nécessaire de préciser le type de l'instance retournée par la méthode fromJson(). Ce type peut être un tableau.

Exemple (code Java 8) :

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;

public class TestJsonB {

  public static void main(String[] args) {

    Jsonb jsonb = JsonbBuilder.create();
    String[] valeurs = jsonb.fromJson("[\"valeur1\",\"valeur2\",\"valeur3\"]",
        String[].class);
    for (String valeur : valeurs) {
      System.out.println(valeur);
    }
  }
}
```

Résultat :

```
valeur1
valeur2
valeur3
```

Ce type peut aussi être une collection.

Exemple (code Java 8) :

```
import java.util.ArrayList;
import java.util.List;

import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;

public class TestJsonB {

  public static void main(String[] args) {

    Jsonb jsonb = JsonbBuilder.create();
    List valeurs = jsonb.fromJson("[\"valeur1\",\"valeur2\",\"valeur3\"]",
        ArrayList.class);
    for (Object object : valeurs) {
      System.out.println(object);
    }
  }
}
```

JSON-B supporte aussi les collections génériques. Pour que le moteur puisse réaliser la désérialisation, il faut passer en second paramètre de la méthode fromJson() le type de l'objet renvoyé.

Exemple (code Java 8) :

```
import java.util.ArrayList;
import java.util.List;

import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;

public class TestJsonB {

  public static void main(String[] args) {

    Jsonb jsonb = JsonbBuilder.create();
    List<String> valeurs = jsonb.fromJson("[\"valeur1\",\"valeur2\",\"valeur3\"]",
        new ArrayList<String>() {}.getClass().getGenericSuperclass());
    for (String valeur : valeurs) {
      System.out.println(valeur);
    }
  }
}
```

58.3.4. Le mapping par défaut des classes de JSON-P

Le mapping par défaut peut utiliser des classes de l'API JSON-P :

- javax.json.JsonArray
- javax.json.JsonStructure
- javax.json.JsonValue
- javax.json.JsonPointer
- javax.json.JsonString
- javax.json.JsonNumber
- javax.json.JsonObject

Le mapping par défaut des classes de l'API JSON-P suit plusieurs règles :

- Les types JSON-P supportés sont : javax.json.JsonArray, javax.json.JsonStructure, javax.json.JsonValue, javax.json.JsonPointer, javax.json.JsonString, javax.json.JsonNumber
- La sérialisation utilise la classe javax.json.JsonWriter
- La désérialisation utilise la classe javax.json.JsonReader

Exemple (code Java 8) :

```
import javax.json.Json;
import javax.json.JsonBuilderFactory;
import javax.json.JsonObject;
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;

public class TestJsonB {

  public static void main(String[] args) {
    Jsonb jsonb = JsonbBuilder.create();
    JsonBuilderFactory factory = Json.createBuilderFactory(null);

    JsonObject jsonObject = factory.createObjectBuilder()
        .add("nom", "nom1")
        .add("prenom", "prenom1")
        .add("dateNaissance", "1985-08-11")
        .add("taille", 175)
        .build();

    String result = jsonb.toJson(jsonObject);
    System.out.println(result);
  }
}
```

Résultat :

```
{"nom":"nom1","prenom":"prenom1","dateNaissance":"1985-08-11","taille":175}
```

58.4. La configuration du moteur JSON-B

Il est possible de modifier le comportement par défaut du moteur JSON-B, pour la sérialisation et la désérialisation, en utilisant une configuration particulière, encapsulée dans un objet de type javax.json.bind.JsonbConfig.

58.4.1. La classe JsonbConfig

La classe JsonbConfig met en oeuvre le design pattern builder pour permettre de construire un objet qui encapsule les options de configuration particulière du moteur JSON-B.

Elle définit plusieurs méthodes permettant de définir la configuration qui sera utilisée par le moteur JSON-B :

Méthode	Rôle
Map<String,Object> getAsMap()	Renvoyer une Map non modifiable de toutes les propriétés de configuration
Optional<Object> getProperty(String name)	Renvoyer la valeur de la propriété de configuration dont le nom est fourni en paramètre
JsonbConfig setProperty(String name, Object value)	Modifier la valeur de la propriété dont le nom est fourni en paramètre
JsonbConfig withAdapters(JsonbAdapter... adapters)	Enregistrer les adaptateurs fournis en paramètre
JsonbConfig withBinaryDataStrategy(String binaryDataStrategy)	Préciser la stratégie de conversion des données binaires
JsonbConfig withDateFormat(String dateFormat, Locale locale)	Préciser le format de conversion des dates
JsonbConfig withDeserializers(JsonbDeserializer... deserializers)	Enregistrer les Deserializers fournis en paramètres
JsonbConfig withEncoding(String encoding)	Préciser le jeu d'encodage des caractères des documents JSON
JsonbConfig withFormatting(Boolean formatted)	Préciser si le document JSON produit doit être formaté ou non
JsonbConfig withLocale(Locale locale)	Préciser la Locale à utiliser
JsonbConfig withNullValues(Boolean serializeNullValues)	Préciser si les propriétés ayant une valeur null sont sérialisées ou non
JsonbConfig withPropertyNamingStrategy(String propertyNamingStrategy)	Préciser la stratégie de nommage des propriétés parmi celles proposées en standard
JsonbConfig withPropertyNamingStrategy(PropertyNamingStrategy propertyNamingStrategy)	Préciser une stratégie personnalisée de nommage des propriétés
JsonbConfig withPropertyOrderStrategy(String propertyOrderStrategy)	Préciser la stratégie de gestion de l'ordre des propriétés
JsonbConfig withPropertyVisibilityStrategy(PropertyVisibilityStrategy propertyVisibilityStrategy)	Préciser une stratégie de gestion de la visibilité des propriétés
JsonbConfig withSerializers(JsonbSerializer... serializers)	Enregistrer des Serializers
JsonbConfig withStrictIJSON(Boolean enabled)	Préciser si le document JSON produit doit respecter la spécification I-JSON ou non

Pour utiliser la configuration, il faut passer une instance de JsonbConfig en paramètre de la méthode withConfig() de la classe JsonbBuilder.

Exemple (code Java 8) :

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import javax.json.bind.JsonbConfig;

public class TestJsonB {

  public static void main(String[] args) {

     JsonbConfig config = (new JsonbConfig()).withFormatting(true);
     Jsonb jsonb = JsonbBuilder.create(config);
  }
}
```

58.4.2. Le formatage du résultat de la sérialisation

Pour demander le formatage du résultat de la sérialisation, ce qui augmente la taille du document mais le rend plus facile à lire par un humain, il suffit de passer la valeur true à la méthode withFormatting() de la classe JsonbConfig.

Exemple (code Java 8) :

```
    JsonbConfig config = new JsonbConfig().withFormatting(true);
    Jsonb jsonb = JsonbBuilder.create(config);
    String result = jsonb.toJson(pers);
    System.out.println(result);
```

Résultat :

```
{
    "adulte": true,
    "dateNaissance": "1985-08-11",
    "nom": "nom",
    "prenom": "prenom",
    "taille": 175
}
```

58.5. La personnalisation du mapping

De nombreux aspects du mapping peuvent être configurés :

- Le nom des propriétés
- L'ordre des propriétés
- Ignorer des propriétés
- La gestion des valeurs null
- La création des instances
- La visibilité des champs
- Le format des dates et des nombres
- L'encodage des données binaires
- Les adaptateurs
- Les Serializers/Deserializers

Cette configuration peut se faire soit :

- En utilisant certaines annotations
- En utilisant une instance de type JsonbConfig pour configurer l'instance de type Jsonb

58.5.1. Le nom d'une propriété

Par défaut, le nom de la propriété dans le document est le nom de propriété dans l'objet Java. Il est possible de personnaliser le nom d'une propriété en utilisant l'annotation @JsonbProperty pour changer son nom.

L'annotation @JsonbProperty peut être utilisée sur :

- Sur un champ : le nom de la propriété est utilisé lors des sérialisations et des désérialisations
- Sur un getter : le nom de la propriété indiqué est uniquement utilisé lors des sérialisations
- Sur un setter : le nom de la propriété indiqué est uniquement utilisé lors des désérialisations

Pour modifier le nom de la propriété d'un champ lors la sérialisation ou la désérialisation, il faut utiliser l'annotation @JsonbProperty sur un champ en lui passant comme valeur le nom de la propriété.

Exemple (code Java 8) :

```
import java.time.LocalDate;

import javax.json.bind.annotation.JsonbProperty;

public class Personne {

  @JsonbProperty("nom-special")
  private String    nom;

  // ...
}
```

Résultat :

```
{"adulte":true,"dateNaissance":"1985-08-11","nom-special":"nom","prenom":"prenom","taille":175}
```

Il est possible d'utiliser l'annotation @JsonbProperty sur les getters/setters pour permettre d'utiliser un nom de propriété différent lors de la sérialisation et de la désérialisation.

Exemple (code Java 8) :

```
import java.time.LocalDate;

import javax.json.bind.annotation.JsonbProperty;

public class Personne {

  // ...

  @JsonbProperty("nom-serialise")
  public String getNom() {
    return nom;
  }

  @JsonbProperty("nom-deserialise")
  public void setNom(String nom) {
    this.nom = nom;
  }

  //...
}
```

Exemple (code Java 8) :

```
import java.time.LocalDate;
import java.time.Month;

import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;

public class TestJsonB {

  public static void main(String[] args) {

    Personne pers = new Personne();
    pers.setNom("nom");
    pers.setPrenom("prenom");
    pers.setTaille(175);
    pers.setAdulte(true);
    pers.setDateNaissance(LocalDate.of(1985, Month.AUGUST, 11));

    Jsonb jsonb = JsonbBuilder.create();
    String result = jsonb.toJson(pers);
    System.out.println(result);

    pers = jsonb.fromJson(
        "{\"nom-deserialise\":\"nom1\",\"prenom\":\"prenom1\",\"taille\":183}",
        Personne.class);
    System.out.println("nom=" + pers.getNom());
  }
}
```

Résultat :

```
{"adulte":true,"dateNaissance":"1985-08-11","nom-serialise":"nom","prenom":"prenom","taille":
175}
nom=nom1
```

58.5.2. La stratégie de nommage des propriétés

Une stratégie de nommage de propriétés permet de définir d'une manière globale comment sont gérés les noms des propriétés.

Plusieurs stratégies de nommage sont proposées grâce à des constantes de l'interface PropertyNamingStrategy :

Stratégie de nommage	Exemple
IDENTITY	nomDeMaPropriete
LOWER_CASE_WITH_DASHES	nom-de-ma-propriete
LOWER_CASE_WITH_UNDERSCORES	nom_de_ma_propriete
UPPER_CAMEL_CASE	NomDeMaPropriete
UPPER_CAMEL_CASE_WITH_SPACES	Nom De Ma Propriete
CASE_INSENSITIVE	nOmDeMaPrOpRiEtE

Il est possible de définir et d'utiliser sa propre stratégie de nommage en implémentant l'interface JsonbNamingInterface.

La stratégie par défaut est IDENTITY.

Pour demander l'application d'une stratégie particulière, il faut utiliser une des deux surcharges de la méthode de la classe JsonbConfig :

- JsonbConfig withPropertyNamingStrategy(String propertyNamingStrategy) : pour appliquer une des stratégies standard
- JsonbConfig withPropertyNamingStrategy(PropertyNamingStrategy propertyNamingStrategy) : pour appliquer sa propre implémentation

Exemple (code Java 8) :

```
    JsonbConfig config = new JsonbConfig().withPropertyNamingStrategy(
        PropertyNamingStrategy.LOWER_CASE_WITH_DASHES);
    Jsonb jsonb = JsonbBuilder.create(config);
    String result = jsonb.toJson(pers);
    System.out.println(result);
```

Résultat :

```
{"adulte":true,"date-naissance":"1985-08-11","nom":"nom","prenom":"prenom","taille":175}
```

58.5.3. Les stratégies de gestion de l'ordre des propriétés

Il est possible de personnaliser l'ordre dans lequel les propriétés sont sérialisées en utilisant la classe PropertyOrderStrategy. Elle définit trois constantes pour désigner les stratégies supportées :

- LEXICOGRAPHICAL : l'ordre lexicographique
- ANY : aucun ordre particulier
- REVERSE : l'ordre lexicographique inverse

La stratégie de gestion de l'ordre de sérialisation des propriétés par défaut du moteur JSON-B est LEXICOGRAPHICAL. Il est possible de modifier ce comportement de deux manières :

- Globalement en utilisant la méthode withPropertyOrderStrategy() de la classe JsonbConfig
- Utiliser l'annotation @JsonbPropertyOrder sur une classe

Il est possible de modifier la configuration pour qu'elle utilise une stratégie d'ordonnancement des propriétés lors de la sérialisation en utilisant la méthode withPropertyOrderStrategy() de la classe JsonbConfig en lui passant en paramètre une des constantes définies dans la classe PropertyOrderStrategy.

Exemple (code Java 8) :

```
   JsonbConfig config = new JsonbConfig().withPropertyOrderStrategy(PropertyOrderStrategy.ANY);
   Jsonb jsonb = JsonbBuilder.create(config);
   String result = jsonb.toJson(pers);
   System.out.println(result);
```

Résultat :

```
{"taille":175,"adulte":true,"dateNaissance":"1985-08-11","nom":"nom","prenom":"prenom"}
```

Pour appliquer une stratégie particulière à une classe, il faut lui appliquer l'annotation @JsonbPropertyOrder en lui passant comme valeur une des constantes définies dans la classe PropertyOrderStrategy

Exemple (code Java 8) :

```
import java.time.LocalDate;

import javax.json.bind.annotation.JsonbPropertyOrder;
import javax.json.bind.config.PropertyOrderStrategy;

@JsonbPropertyOrder(PropertyOrderStrategy.ANY)
public class Personne {

  // ...
}
```

58.5.4. Ignorer une propriété

L'annotation @javax.json.bind.annotation.JsonbTransient permet de demander au moteur JSON-B d'ignorer une propriété lors de ses opérations de sérialisation et/ou de désérialisation selon l'élément sur lequel l'annotation est placée :

- Sur un champ : la propriété est ignorée lors des sérialisations et des désérialisations
- Sur un getter : la propriété est uniquement ignorée lors des sérialisations
- Sur un setter : la propriété est uniquement ignorée lors des désérialisations

Exemple (code Java 8) :

```
import java.time.LocalDate;
import javax.json.bind.annotation.JsonbTransient;

public class Personne {

  private String    nom;
  private String    prenom;
  private int       taille;
  private boolean   adulte;

  @JsonbTransient
  private LocalDate dateNaissance;

  // ...

  public LocalDate getDateNaissance() {
    return dateNaissance;
  }

  public void setDateNaissance(LocalDate dateNaissance) {
    this.dateNaissance = dateNaissance;
  }
}
```

Dans l'exemple ci-dessus, la propriété dateNaissance sera ignorée lors des opérations de sérialisation mais sera utilisée lors de désérialisations avec JSON-B.

Exemple (code Java 8) :

```
  // ...

  private LocalDate dateNaissance;

  // ...

  public LocalDate getDateNaissance() {
    return dateNaissance;
  }

  @JsonbTransient
  public void setDateNaissance(LocalDate dateNaissance) {
    this.dateNaissance = dateNaissance;
  }
}
```

Dans l'exemple ci-dessus, la propriété dateNaissance sera ignorée lors des opérations de sérialisation et de désérialisation avec JSON-B.

Exemple (code Java 8) :

```
  // ...

  private LocalDate dateNaissance;

  // ...

  public LocalDate getDateNaissance() {
    return dateNaissance;
  }

  public void setDateNaissance(LocalDate dateNaissance) {
    this.dateNaissance = dateNaissance;
  }
}
```

Dans l'exemple ci-dessus, la propriété dateNaissance sera utilisée lors des opérations de sérialisation et sera ignorée lors de désérialisations avec JSON-B.

58.5.5. La visibilité des propriétés

Il est possible de définir une stratégie personnalisée de visibilité des propriétés. Cela peut, par exemple, permettre de sérialiser des champs private sans getter.

Exemple (code Java 8) :

```
public class MaClasse {

  private String description;
  private String nom;
  private int    longueur;

  public MaClasse() {
  }

  public MaClasse(String description, String nom, int longueur) {
    this.description = description;
    this.nom = nom;
    this.longueur = longueur;
  }
}
```

Il faut définir une classe qui implémente l'interface PropertyVisibilityStrategy. Cette interface définit deux méthodes :

Méthode	Rôle
boolean isVisible(Field field)	Renvoyer un booléen qui précise si le champ passé en paramètre doit être pris en compte comme étant une JsonbProperty
boolean isVisible(Method method)	Renvoyer un booléen qui précise si la méthode passée en paramètre doit être prise en compte comme étant une JsonbProperty

Exemple (code Java 8) :

```
import java.lang.reflect.Field;
import java.lang.reflect.Method;

import javax.json.bind.config.PropertyVisibilityStrategy;

public class MonPropertyVisibilityStrategy implements PropertyVisibilityStrategy {

  @Override
  public boolean isVisible(Field field) {
    return true;
  }

  @Override
  public boolean isVisible(Method method) {
    return true;
  }
}
```

L'implémentation ci-dessus est très basique car elle permet d'accéder à toutes les propriétés.

Pour pouvoir utiliser la stratégie de manière globale, il faut utiliser la méthode withPropertyVisibilityStrategy() de la classe JsonbConfig utilisée pour obtenir l'instance du moteur JSON-B. Elle attend en paramètre une instance de la classe qui implémente l'interface PropertyVisibilityStrategy.

Exemple (code Java 8) :

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import javax.json.bind.JsonbConfig;
```

```
public class TestJsonB {

  public static void main(String[] args) {

    MaClasse obj = new MaClasse("la description", "le nom", 250);

    JsonbConfig config = new JsonbConfig()
        .withPropertyVisibilityStrategy(new MonPropertyVisibilityStrategy());
    Jsonb jsonb = JsonbBuilder.create(config);
    String result = jsonb.toJson(obj);
    System.out.println(result);
  }
}
```

Résultat :

```
{"description":"la description","longueur":250,"nom":"le nom"}
```

Pour demander l'application de cette stratégie sur une seule classe, il faut lui appliquer l'annotation @JsonbVisibility en lui passant comme valeur de son attribut par le défaut la classe d'implémentation de l'interface PropertyVisibilityStrategy.

Exemple (code Java 8) :

```
import javax.json.bind.annotation.JsonbVisibility;

@JsonbVisibility(MonPropertyVisibilityStrategy.class)
public class MaClasse {

  private String description;
  private String nom;
  private int    longueur;

  // ...
}
```

Exemple (code Java 8) :

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;

public class TestJsonB {

  public static void main(String[] args) {

    MaClasse obj = new MaClasse("la description", "le nom", 250);

    Jsonb jsonb = JsonbBuilder.create();
    String result = jsonb.toJson(obj);
    System.out.println(result);
  }
}
```

Résultat :

```
{"description":"la description","longueur":250,"nom":"le nom"}
```

58.5.6. La gestion des valeurs null

La configuration par défaut du moteur JSON-B ne sérialise pas les champs dont la valeur est null.

Exemple (code Java 8) :

```
import java.time.LocalDate;
import java.time.Month;
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import javax.json.bind.JsonbConfig;

public class TestJsonB {

  public static void main(String[] args) {

    Personne pers = new Personne();
    pers.setNom(null);
    pers.setPrenom(null);
    pers.setTaille(175);
    pers.setAdulte(true);
    pers.setDateNaissance(LocalDate.of(1985, Month.AUGUST, 11));

    Jsonb jsonb = JsonbBuilder.create();
    String result = jsonb.toJson(pers);
    System.out.println(result);
  }
}
```

Résultat :

```
{"adulte":true,"dateNaissance":"1985-08-11","taille":175}
```

Il y a trois possibilités pour modifier ce comportement par défaut :

- Utiliser l'annotation @JsonbNillable sur une classe ou un package
- Utiliser l'annotation @JsonbProperty sur une propriété avec l'attribut nillable=true
- Invoquer withNullValues(true) sur l'instance de JsonbConfig utilisée par le moteur pour changer globalement le comportement de la gestion des valeurs null

Il est possible d'utiliser l'annotation @JsonbNillable sur une classe ou un package pour demander la sérialisation des valeurs des propriétés de la ou des classes concernées.

Exemple (code Java 8) :

```
import java.time.LocalDate;

import javax.json.bind.annotation.JsonbNillable;

@JsonbNillable
public class Personne {

  // ...
}
```

Résultat :

```
{"adulte":true,"dateNaissance":"1985-08-11","nom":null,"prenom":null,"taille":175}
```

Il est possible d'utiliser l'annotation @JsonbProperty avec son attribut nillable=true sur un champ pour demander de sérialiser sa valeur si elle est null.

Exemple (code Java 8) :

```
import java.time.LocalDate;

import javax.json.bind.annotation.JsonbProperty;

public class Personne {

  private String    nom;
  @JsonbProperty(nillable = true)
  private String    prenom;
  // ...
}
```

Résultat :

```
{"adulte":true,"dateNaissance":"1985-08-11","prenom":null,"taille":175}
```

Il est possible de modifier globalement la configuration pour que toutes les valeurs null soient sérialisées en invoquant la méthode withNullValues(true) de l'instance de JsonbConfig utilisée pour obtenir l'instance du moteur.

Exemple (code Java 8) :

```
    JsonbConfig config = new JsonbConfig().withNullValues(true);
    Jsonb jsonb = JsonbBuilder.create(config);
    String result = jsonb.toJson(pers);
    System.out.println(result);
```

Résultat :

```
{"adulte":true,"dateNaissance":"1985-08-11","nom":null,"prenom":null,"taille":175}
```

58.5.7. La personnalisation de la création d'instanciation

Par défaut, les classes utilisées lors de la désérialisation doivent avoir un constructeur par défaut qui sera invoqué par le moteur JSON-B pour créer une instance. Cette contrainte est parfois trop contraignante : c'est par exemple le cas pour un objet immuable.

L'annotation @javax.json.bind.annotation.JsonbCreator peut être utilisée sur un constructeur ou une fabrique statique pour indiquer au moteur JSON comment créer une instance.

L'annotation @JsonbCreator ne possède aucun attribut et ne peut être utilisée que sur une seule méthode statique qui est une fabrique ou un seul constructeur avec paramètre(s) dans la classe. Si ce n'est pas le cas alors une exception de type JsonbException est levée.

Pour assurer la bonne liaison entre les données lues du document JSON et les paramètres du constructeur ou de la fabrique, il faut annoter chaque paramètre avec l'annotation @JsonbProperty() en lui précisant comme valeur le nom de l'attribut. Le nom de l'attribut précisé doit exister dans le document JSON sinon une exception de type JsonbException est levée.

Si l'annotation @JsonbProperty n'est pas utilisée sur les paramètres, le moteur sa tenter de faire une correspondance avec le nom du paramètre, sous réserve que celui-ci soit accessible.

Il est possible d'utiliser l'annotation @JsonbCreator sur un constructeur.

Exemple (code Java 8) :

```
import javax.json.bind.annotation.JsonbCreator;
import javax.json.bind.annotation.JsonbProperty;

public class Produit {
  private long   id;
  private String libelle;
  private String reference;

  @JsonbCreator
  public Produit(@JsonbProperty("id") long id,
      @JsonbProperty("libelle") String lib,
      @JsonbProperty("reference") String ref) {
    this.id = id;
    libelle = lib;
    reference = ref;
    System.out.println("Invocation constructeur Produit");
  }

  public long getId() {
    return id;
  }

  public String getLibelle() {
    return libelle;
  }

  public String getReference() {
    return reference;
  }
}
```

Exemple (code Java 8) :

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import javax.json.bind.JsonbConfig;

public class TestJsonB {

  public static void main(String[] args) {

    JsonbConfig config = new JsonbConfig().withStrictIJSON(true);
    Jsonb jsonb = JsonbBuilder.create(config);

    Produit produit = new Produit(1, "test", "ref1");
    String result = jsonb.toJson(produit);
    System.out.println(result);

    Produit prod = jsonb.fromJson(result, Produit.class);
    System.out.println("id       =" + prod.getId());
    System.out.println("libelle  =" + prod.getLibelle());
    System.out.println("reference=" + prod.getReference());
  }
}
```

Résultat :

```
Invocation constructeur Produit
{"id":1,"libelle":"test","reference":"ref1"}
Invocation constructeur Produit
id       =1
libelle  =test
reference=ref1
```

Il est possible d'utiliser l'annotation @JsonbCreator sur une méthode statique qui est une fabrique pour obtenir une instance : elle doit donc renvoyer une instance du type de la classe sinon une exception de type JsonbException est levée.

Exemple (code Java 8) :

```
import javax.json.bind.annotation.JsonbCreator;
import javax.json.bind.annotation.JsonbProperty;

public class Produit {
  private long   id;
  private String libelle;
  private String reference;

  @JsonbCreator
  public static Produit creerInstance(@JsonbProperty("id") long id,
      @JsonbProperty("libelle") String lib,
      @JsonbProperty("reference") String ref) {
    Produit result = new Produit();
    result.id = id;
    result.libelle = lib;
    result.reference = ref;
    System.out.println("Invocation fabrique Produit");
    return result;
  }

  public long getId() {
    return id;
  }

  public String getLibelle() {
    return libelle;
  }

  public String getReference() {
    return reference;
  }
}
```

Exemple (code Java 8) :

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import javax.json.bind.JsonbConfig;

public class TestJsonB {

  public static void main(String[] args) {
    JsonbConfig config = new JsonbConfig().withStrictIJSON(true);
    Jsonb jsonb = JsonbBuilder.create(config);

    Produit produit = Produit.creerInstance(1, "test", "ref1");
    String result = jsonb.toJson(produit);
    System.out.println(result);

    Produit prod = jsonb.fromJson(result, Produit.class);
    System.out.println("id       =" + prod.getId());
    System.out.println("libelle  =" + prod.getLibelle());
    System.out.println("reference=" + prod.getReference());
  }
}
```

Résultat :

```
Invocation fabrique Produit
{"id":1,"libelle":"test","reference":"ref1"}
Invocation fabrique Produit
id       =1
libelle  =test
reference=ref1
```

58.5.8. Le format des dates et des nombres

Par défaut, JSON-B utilise les formats ISO pour sérialiser et désérialiser des données temporelles et numériques.

Exemple (code Java 8) :

```
import java.math.BigDecimal;
import java.util.Date;

public class ResultatEpreuve {
  public long       idEpreuve;
  public long       idCandidat;
  public Date       date;
  public BigDecimal note;
}
```

Exemple (code Java 8) :

```
import java.math.BigDecimal;
import java.util.Date;

import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import javax.json.bind.JsonbConfig;

public class TestJsonB {

  public static void main(String[] args) {
    JsonbConfig config = (new JsonbConfig()).withFormatting(true);
    Jsonb jsonb = JsonbBuilder.create(config);
    ResultatEpreuve re = new ResultatEpreuve();
    re.date = new Date();
    re.idCandidat = 123;
    re.idEpreuve = 456;
    re.note = new BigDecimal("15.5");
    String result = jsonb.toJson(re);
    System.out.println(result);
  }
}
```

Résultat :

```
{
    "date": "2017-08-31T07:53:37.753Z[UTC]",
    "idCandidat": 123,
    "idEpreuve": 456,
    "note": 15.5
}
```

Pour utiliser un format particulier pour un champ, il est possible d'utiliser les annotations @JsonbDateFormat et @JsonbNumberFormat.

Exemple (code Java 8) :

```
import java.math.BigDecimal;
import java.util.Date;

import javax.json.bind.annotation.JsonbDateFormat;
import javax.json.bind.annotation.JsonbNumberFormat;

public class ResultatEpreuve {
  public long       idEpreuve;
  public long       idCandidat;
  @JsonbDateFormat("dd/MMM/yyyy")
  public Date       date;
  @JsonbNumberFormat("#0.00")
  public BigDecimal note;
}
```

Résultat :

```
{
    "date": "31/Aug/2017",
    "idCandidat": 123,
    "idEpreuve": 456,
    "note": "15,50"
}
```

Les annotations @JsonbDateFormat et @JsonbNumberFormat peuvent s'utiliser sur un champ, un getter/setter, un type, un paramètre ou un package.

L'annotation @JsonbDateFormat possède deux attributs optionnels :

Attribut	Rôle
String locale	La Locale à utiliser
String value	Le motif de formatage à utiliser exprimé avec le format supporté par la classe DateTimeFormatter

Exemple (code Java 8) :

```
  // ...
  @JsonbDateFormat(value = "dd/MMM/yyyy", locale = "FR")
  public Date       date;
  // ...
```

Résultat :

```
{
    "date": "31/août/2017",
    "idCandidat": 123,
    "idEpreuve": 456,
    "note": "15,50"
}
```

L'annotation @JsonbNumberFormat possède deux attributs optionnels :

Attribut	Rôle
String locale	La Locale à utiliser
String value	Le motif de formatage à utiliser en utilisant le format supporté par la classe DecimalFormat

Il est aussi possible de configurer globalement le format des dates utilisant une instance de type JsonbConfig qui propose deux méthodes :

Méthode	Rôle
JsonbConfig withDateFormat(String dateFormat, Locale locale)	Préciser le format de date et la Locale à utiliser
JsonbConfig withLocale(Locale locale)	Préciser la Locale à utiliser

Exemple (code Java 8) :

```
import java.math.BigDecimal;
import java.util.Date;
import java.util.Locale;

import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import javax.json.bind.JsonbConfig;

public class TestJsonB {

  public static void main(String[] args) {

    JsonbConfig config = (new JsonbConfig()).withFormatting(true)
        .withDateFormat("dd/MMM/yyyy", Locale.ITALIAN);
    Jsonb jsonb = JsonbBuilder.create(config);
    ResultatEpreuve re = new ResultatEpreuve();
    re.date = new Date();
    re.idCandidat = 123;
    re.idEpreuve = 456;
    re.note = new BigDecimal("15.5");
    String result = jsonb.toJson(re);
    System.out.println(result);
  }
}
```

Résultat :

```
{
    "date": "31/ago/2017",
    "idCandidat": 123,
    "idEpreuve": 456,
    "note": 15.5
}
```

58.5.9. Binary Data Encoding

JSON-B supporte les données binaires. Les trois types d'encodage supportés sont définis sous la forme de constantes dans la classe BinaryDataStrategy:

- BYTE : avec cette stratégie, les données sont encodées comme un tableau d'octets. C'est la stratégie par défaut
- BASE_64 : avec cette stratégie, les données sont encodées en Base64 selon les RFC 4648 et RFC 2045
- BASE_64_URL : avec cette stratégie, les données sont encodées en utilisant "URL and Filename safe Base64 Alphabet" défini dans la Table 2 de la RFC 4648

Pour définir la stratégie globale de binding des données binaires, il faut passer une de ces constantes à la méthode withBinaryDataStrategy() de la classe JsonbConfig.

Exemple (code Java 8) :

```
import java.util.Random;

import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import javax.json.bind.JsonbConfig;
import javax.json.bind.config.BinaryDataStrategy;

public class TestJsonB {

  public static void main(String[] args) {
    byte[] donnees = new byte[10];
    new Random().nextBytes(donnees);

    System.out.println("BYTE        " + serialiserBinaire(donnees,
        BinaryDataStrategy.BYTE));
    System.out.println("BASE_64     " + serialiserBinaire(donnees,
        BinaryDataStrategy.BASE_64));
    System.out.println("BASE_64_URL " + serialiserBinaire(donnees,
        BinaryDataStrategy.BASE_64_URL));
  }

  private static String serialiserBinaire(byte[] donnees, String strategie) {
    JsonbConfig config = new JsonbConfig().withBinaryDataStrategy(strategie);
    Jsonb jsonb = JsonbBuilder.create(config);
    String result = jsonb.toJson(donnees);
    return result;
  }
}
```

Résultat :

```
BYTE        [108,47,87,-83,77,-1,3,-90,-111,95]
BASE_64     "bC9XrU3/A6aRXw=="
BASE_64_URL "bC9XrU3_A6aRXw=="
```

58.5.10. Strict I-JSON support

Internet JSON ou I-JSON est une spécification pour définir un profile sur l'utilisation de JSON afin de maximiser l'interopérabilité de l'exploitation des documents par des applications.

JSON-B propose un support de I-JSON par défaut à l'exception de trois recommandations :

- JSON-B ne limite pas la sérialisation en un document JSON dont le niveau supérieur soit un objet ou un tableau
- JSON-B ne sérialise pas les données binaires avec l'encodage base 64 url par défaut
- JSON Binding ne met pas en oeuvre pas les restrictions supplémentaires recommandées par I-JSON sur les dates, les heures ou une durée

Pour activer le support complet, il faut utiliser la méthode withStrictIJSON() avec la valeur true en paramètre de l'instance de type JsonbConfig utilisée pour configurer le moteur.

Exemple (code Java 8) :

```
    // ...
    JsonbConfig config = new JsonbConfig().withStrictIJSON(true);
    Jsonb jsonb = JsonbBuilder.create(config);
    // ...
```

58.5.11. Les adaptateurs

Parfois, il n'est pas toujours possible d'utiliser des annotations pour personnaliser le binding, par exemple parce que le code source n'est pas disponible ou parce que les annotations proposées ne permettent pas de répondre aux besoins. Dans ces cas, il est possible d'utiliser un adaptateur pour tenter de répondre aux besoins.

Les adaptateurs de JSON-B fonctionnent de manière similaire à ceux proposés par JAX-B.

Un adaptateur est une classe qui implémente l'interface javax.json.bind.adapter.JsonbAdapter. L'interface JsonbAdapter<Original, Adapted> possède deux types génériques :

- Original : le type qui n'est pas pris en charge par JSON-B
- Adapted : le type que JSON-B sait prendre en compte

Elle définit deux méthodes :

Méthode	Rôle
Original adaptFromJson(Adapted obj)	Cette méthode est invoquée uniquement lors de la désérialisation
Adapted adaptToJson(Original obj)	Cette méthode est invoquée uniquement lors de la sérialisation

L'implémentation doit contenir les traitements pour créer une instance de type Adapted à partir d'une instance de type Original et vice versa. Cela permettra au moteur JSON-B de sérialiser/désérialiser le type Adapted à la place du type Original.

Exemple (code Java 8) :

```
import javax.json.Json;
import javax.json.JsonObject;
import javax.json.bind.adapter.JsonbAdapter;

public class PersonneAdapter implements JsonbAdapter<Personne, JsonObject> {

  @Override
  public JsonObject adaptToJson(Personne pers) throws Exception {
    return Json.createObjectBuilder()
        .add("n", pers.getNom())
        .add("p", pers.getPrenom())
        .build();
  }

  @Override
  public Personne adaptFromJson(JsonObject obj) throws Exception {
    Personne p = new Personne();
    p.setNom(obj.getString("n"));
    p.setPrenom(obj.getString("p"));
    return p;
  }
}
```

Lors d'une sérialisation d'un objet de type Original, le moteur JSON-B invoque la méthode adaptToJson() de l'adaptateur pour obtenir une instance de type Adapted et la sérialiser.

Lors d'une désérialisation d'un objet de type Adapted, le moteur JSON-B invoque la méthode adaptFromJson() de l'adaptateur pour obtenir une instance de type Original et la retourner.

Il y a deux manières d'enregistrer un adaptateur pour qu'il soit pris en compte par le moteur :

- Utiliser la méthode withAdapters() de la classe JsonbConfig
- Utiliser l'annotation @JsonbTypeAdapter

Il est possible d'enregistrer globalement un adaptateur en utilisant la méthode withAdapters() de la classe JsonbConfig en lui passant une nouvelle instance de l'adaptateur en paramètre. L'adaptateur sera alors utilisé à chaque sérialisation/désérialisation d'un objet de type Original.

Exemple (code Java 8) :

```
import java.time.LocalDate;
import java.time.Month;

import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import javax.json.bind.JsonbConfig;

public class TestJsonB {

  public static void main(String[] args) {
    JsonbConfig config = new JsonbConfig().withAdapters(new PersonneAdapter());
    Jsonb jsonb = JsonbBuilder.create(config);
    Personne pers = new Personne();
    pers.setNom("nom");
    pers.setPrenom("prenom");
    pers.setTaille(175);
    pers.setAdulte(true);
    pers.setDateNaissance(LocalDate.of(1985, Month.AUGUST, 11));

    String result = jsonb.toJson(pers);
    System.out.println("result=" + result);

    pers = jsonb.fromJson(result, Personne.class);
    System.out.println("nom=" + pers.getNom());
    System.out.println("dateNaissance=" + pers.getDateNaissance());
  }
}
```

Pour utiliser l'adaptateur uniquement pour un champ, il est possible de l'annoter avec @JsonbTypeAdapter en lui précisant comme attribut le type de l'adaptateur.

Exemple (code Java 8) :

```
import javax.json.bind.annotation.JsonbTypeAdapter;

public class MonBean {
  public String   id = "1";
  @JsonbTypeAdapter(PersonneAdapter.class)
  public Personne personne;
}
```

Exemple (code Java 8) :

```
import java.time.LocalDate;
import java.time.Month;

import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;

public class TestJsonB {

  public static void main(String[] args) {
    Jsonb jsonb = JsonbBuilder.create();
    Personne pers = new Personne();
    pers.setNom("nom");
    pers.setPrenom("prenom");
    pers.setTaille(175);
    pers.setAdulte(true);
    pers.setDateNaissance(LocalDate.of(1985, Month.AUGUST, 11));

    MonBean monBean = new MonBean();
    monBean.personne = pers;

    String result = jsonb.toJson(monBean);
    System.out.println("result=" + result);

    monBean = jsonb.fromJson(result, MonBean.class);
    System.out.println("nom=" + monBean.personne.getNom());
    System.out.println("dateNaissance=" + monBean.personne.getDateNaissance());
  }
}
```

Résultat :

```
{
    "date": "31/ago/2017",
    "idCandidat": 123,
    "idEpreuve": 456,
    "note": 15.5
}
```

58.5.12. Les Serializers/Deserializers

Les Serializers/Deserializers permettent d'avoir un contrôle de bas niveau sur les opérations de sérialisation/désérialisation grâce à une utilisation de JSON-P. Ils sont par exemple utiles lorsque les adaptateurs ne suffisent pas.

Un Serializer est une classe qui implémente l'interface javax.json.bind.serializers.JsonbSerializer.

Un Deserializer est une classe qui implémente l'interface javax.json.bind.serializers.JsonbDeserializer.

Il y a deux possibilités pour enregistrer un Serializer ou Deserializer :

- De manière globale en utilisant les méthodes withSerializer() et/ou withDeserializer() de la classe JsonbConfig
- Utiliser l'annotation @JsonbSerializer et/ou @JsonbDeserializer sur un type

L'interface JsonbSerializer<T> permet de personnaliser la sérialisation d'un objet de type T. Elle permet un contrôle très précis de la sérialisation en utilisant un objet de type JsonGenerator de l'API de type Stream de JSON-P.

L'interface JsonbSerializer<T> définit une seule méthode :

Méthode	Rôle
void serialize(T obj, javax.json.stream.JsonGenerator generator, SerializationContext ctx)	Sérialiser l'objet passé en paramètre en utilisant l'API Stream de JSON-P

L'implémentation de la méthode serialize() doit utiliser l'objet de type JsonGenerator pour créer le document JSON à partir de l'objet passé en paramètre.

Exemple (code Java 8) :

```
import javax.json.bind.serializer.JsonbSerializer;
import javax.json.bind.serializer.SerializationContext;
import javax.json.stream.JsonGenerator;

public class PersonneSerializer implements JsonbSerializer<Personne> {
  @Override
  public void serialize(Personne personne, JsonGenerator generator,
      SerializationContext ctx) {
```

```
    generator.writeStartObject();
    generator.write("nom_s", personne.getNom());
    generator.write("prenom_s", personne.getPrenom());
    generator.writeEnd();
  }
}
```

Dans les traitements de l'implémentation de la méthode serialize(), il est possible d'utiliser l'instance de type SerializationContext pour sérialiser un objet.

L'interface SerialisationContext propose deux méthodes pour faciliter la sérialisation d'un objet :

Méthode	Rôle
<T> void serialize(String key, T object, javax.json.stream.JsonGenerator generator)	Sérialiser l'objet passé en paramètre en utilisant l'instance de type JsonGenerator sous la forme d'un attribut dont le nom est passé en paramètre
<T> void serialize(T object, javax.json.stream.JsonGenerator generator)	Sérialiser l'objet passé en paramètre en utilisant l'instance de type JsonGenerator sous la forme d'un tableau

Pour utiliser le Serializer de manière globale, il faut en passer une instance en paramètre de la méthode withSerializer() de la classe JsonbConfig.

Exemple (code Java 8) :

```
import java.time.LocalDate;
import java.time.Month;

import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import javax.json.bind.JsonbConfig;

public class TestJsonB {

  public static void main(String[] args) {

    JsonbConfig config = new JsonbConfig()
        .withSerializers(new PersonneSerializer());
    Jsonb jsonb = JsonbBuilder.create(config);

    Personne pers = new Personne();
    pers.setNom("nom");
    pers.setPrenom("prenom");
    pers.setTaille(175);
    pers.setAdulte(true);
    pers.setDateNaissance(LocalDate.of(1985, Month.AUGUST, 11));
    String result = jsonb.toJson(pers);
    System.out.println(result);
  }
}
```

Résultat :

```
{"nom_s":"nom","prenom_s":"prenom"}
```

Il est aussi possible d'utiliser l'annotation @JsonbTypeSerializer en lui passant comme valeur la classe de l'implémentation du JsonbSerializer.

Exemple (code Java 8) :

```
import javax.json.bind.annotation.JsonbTypeSerializer;

public class MonBean {
  public String   id = "1";
  @JsonbTypeSerializer(PersonneSerializer.class)
  public Personne personne;
}
```

Exemple (code Java 8) :

```
import java.time.LocalDate;
import java.time.Month;

import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;

public class TestJsonB {

  public static void main(String[] args) {

    Jsonb jsonb = JsonbBuilder.create();

    MonBean monBean = new MonBean();
    Personne pers = new Personne();
    pers.setNom("nom");
    pers.setPrenom("prenom");
    pers.setTaille(175);
    pers.setAdulte(true);
    pers.setDateNaissance(LocalDate.of(1985, Month.AUGUST, 11));

    monBean.id = "1";
    monBean.personne = pers;

    String result = jsonb.toJson(monBean);
    System.out.println(result);
  }
}
```

Résultat :

```
{"id":"1","personne":{"nom_s":"nom","prenom_s":"prenom"}}
```

L'interface JsonbDeserializer<T> permet de personnaliser la désérialisation d'un objet de type T. Elle permet un contrôle très précis de la désérialisation en utilisant un objet de type JsonParser de l'API de type Stream de JSON-P.

L'interface JsonbDeserializer<T> définit une seule méthode :

Méthode	Rôle
T deserialize(javax.json.stream.JsonParser parser, DeserializationContext ctx, Type rtType)	Désérialiser l'objet passé en paramètre en utilisant l'API Stream de JSON-P

L'implémentation de la méthode deserialize() doit utiliser l'objet de type JsonParser pour extraire les données du document JSON et créer l'instance à retourner.

Exemple (code Java 8) :

```
import java.lang.reflect.Type;

import javax.json.bind.serializer.DeserializationContext;
import javax.json.bind.serializer.JsonbDeserializer;
import javax.json.stream.JsonParser;
import javax.json.stream.JsonParser.Event;

public class PersonneDeserializer implements JsonbDeserializer<Personne> {

  @Override
  public Personne deserialize(JsonParser parser, DeserializationContext ctx,
      Type rtType) {
    Personne result = new Personne();
    while (parser.hasNext()) {
      Event event = parser.next();
      if (event == JsonParser.Event.KEY_NAME) {
        String nomAttr = parser.getString();
        parser.next();
        if (nomAttr.equals("nom_s")) {
          result.setNom(parser.getString());

        } else if (nomAttr.equals("prenom_s")) {
          result.setPrenom(parser.getString());
        }
      }
    }
    return result;
  }
}
```

Dans les traitements de l'implémentation de la méthode deserialize(), il est possible d'utiliser l'instance de type DeserializationContext pour désérialiser un objet.

L'interface DeserializationContext propose deux méthodes pour faciliter la désérialisation d'un objet :

Méthode	Rôle
<T> T deserialize(Class<T> clazz, javax.json.stream.JsonParser parser)	Désérialiser l'objet passé en paramètre en utilisant l'instance de type JsonParser
<T> T deserialize(Type type, javax.json.stream.JsonParser parser)	Désérialiser l'objet passé en paramètre en utilisant l'instance de type JsonParser

Pour utiliser le Deserializer de manière globale, il faut en passer une instance en paramètre de la méthode withDeserializer() de la classe JsonbConfig.

Exemple (code Java 8) :

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import javax.json.bind.JsonbConfig;

public class TestJsonB {

  public static void main(String[] args) {

    JsonbConfig config = new JsonbConfig()
        .withSerializers(new PersonneSerializer())
        .withDeserializers(new PersonneDeserializer());
    Jsonb jsonb = JsonbBuilder.create(config);
    Personne pers = jsonb.fromJson("{\"nom_s\":\"nom1\",\"prenom_s\":\"prenom1\"}",
        Personne.class);
    System.out.println("nom   =" + pers.getNom());
    System.out.println("prenom=" + pers.getPrenom());
  }
}
```

Résultat :

```
nom   =nom1
prenom=prenom1
```

Il est aussi possible d'utiliser l'annotation @JsonbTypeDeserializer en lui passant comme valeur la classe de l'implémentation du JsonbDeserializer.

Exemple (code Java 8) :

```
import javax.json.bind.annotation.JsonbTypeDeserializer;
import javax.json.bind.annotation.JsonbTypeSerializer;

public class MonBean {
  public String   id = "1";
  @JsonbTypeSerializer(PersonneSerializer.class)
  @JsonbTypeDeserializer(PersonneDeserializer.class)
  public Personne personne;
}
```

Exemple (code Java 8) :

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;

public class TestJsonB {

  public static void main(String[] args) {

    Jsonb jsonb = JsonbBuilder.create();
    MonBean monBean = jsonb.fromJson(
        "{\"id\":\"1\",\"personne\":{\"nom_s\":\"nom\",\"prenom_s\":\"prenom\"}}",
        MonBean.class);
    System.out.println("nom   =" + monBean.personne.getNom());
    System.out.println("prenom=" + monBean.personne.getPrenom());
  }
}
```

Résultat :

```
nom   =nom
prenom=prenom
```

Partie 8 : L'accès aux bases de données

Partie 8 :
L'accès aux
bases de
données

Cette partie concerne l'accès aux bases de données à partir d'applications Java. Il existe pour ce besoin de nombreuses solutions sous différentes formes dont plusieurs API standards ou frameworks open source.

Cette partie regroupe plusieurs chapitres :

- La persistance des objets : expose les difficultés liées à la persistance des objets vis à vis du modèle relationnel et présente rapidement des solutions architecturales et techniques (API standards et open source)
- JDBC : indique comment utiliser cette API historique pour accéder aux bases de données
- JDO (Java Data Object) : API qui standardise et automatise le mapping entre des objets Java et un système de gestion de données
- Hibernate : présente Hibernate, un framework de mapping Objets/Relationnel open source
- JPA (Java Persistence API) : JPA est la spécification de l'API standard dans le domaine du mapping O/R utilisable avec Java EE mais aussi avec Java SE à partir de la version 5.

59. La persistance des objets

Chapitre 59

Niveau : 🟩🟩⬜⬜⬜⬜ Elémentaire

La quasi-totalité des applications de gestion traitent des données dans des volumes plus ou moins importants. Dès que ce volume devient assez important, les données sont stockées dans une base de données.

Il existe plusieurs types de base de données

- Hiérarchique : historiquement le type le plus ancien, ces bases de données étaient largement utilisées sur les gros systèmes de type mainframe. Les données sont organisées de façon hiérarchique grâce à des pointeurs. Exemple DL1, IMS, Adabas
- Relationnelle (RDBMS / SGBDR) : c'est le modèle le plus répandu actuellement. Ce type de base de données repose sur les théories ensemblistes et l'algèbre relationnel. Les données sont organisées en tables possédant des relations entre elles grâce à des clés primaires et étrangères. Les opérations sur la base sont réalisées grâce à des requêtes SQL. Exemple : MySQL, PosgreSQL, HSQLDB, Derby
- Objet (ODBMS / SGBDO) : Exemple db4objects
- XML (XDBMS) : Exemple : Xindice

La seconde catégorie est historiquement la plus répandue mais aussi une des moins compatibles avec la programmation orientée objet.

Ce chapitre contient plusieurs sections :

- ♦ La correspondance entre les modèles relationnel et objet
- ♦ L'évolution des solutions de persistance avec Java
- ♦ Le mapping O/R (objet/relationnel)
- ♦ L'architecture et la persistance de données
- ♦ Les différentes solutions
- ♦ Les API standards
- ♦ Les frameworks open source
- ♦ L'utilisation de procédures stockées

59.1. La correspondance entre les modèles relationnel et objet

La correspondance des données entre le modèle relationnel et le modèle objet doit faire face à plusieurs problèmes :

- le modèle objet propose plus de fonctionnalités : héritage, polymorphisme, ...
- les relations entre les entités des deux modèles sont différentes
- un objet ne possède pas d'identifiant en standard (hormis son adresse mémoire qui varie d'une exécution à l'autre). Dans le modèle relationnel, chaque occurrence devrait posséder un identifiant unique

La persistance des objets en Java présente de surcroît quelques inconvénients supplémentaires :

- de multiples choix dans les solutions et les outils (standard, commerciaux, open source)
- de multiples choix dans les API et leurs implémentations
- de nombreuses évolutions dans les API standard et les frameworks open source

59.2. L'évolution des solutions de persistance avec Java

La première approche pour faire une correspondance entre ces deux modèles a été d'utiliser l'API JDBC fournie en standard avec le JDK. Cependant cette approche présente plusieurs inconvénients majeurs :

- nécessite l'écriture de nombreuses lignes de codes, souvent répétitives
- le mapping entre les tables et les objets est un travail de bas niveau
- ...

Tous ces facteurs réduisent la productivité mais aussi les possibilités d'évolutions et de maintenance. De plus, une grande partie de ce travail peut être automatisée.

Face à ce constat, différentes solutions sont apparues :

- des frameworks open source : le plus populaire est Hibernate qui utilise des POJOs.
- des frameworks commerciaux dont Toplink était le leader avant que sa base passe en open source
- des API Standards : JDO, EJB entity, JPA

59.3. Le mapping O/R (objet/relationnel)

Le mapping Objet/Relationnel (mapping O/R) consiste à réaliser la correspondance entre le modèle de données relationnel et le modèle objets de la façon la plus facile possible.

Un outil de mapping O/R doit cependant proposer un certain nombre de fonctionnalités parmi lesquelles :

- Assurer le mapping des tables avec les classes, des champs avec les attributs, des relations et des cardinalités
- Proposer une interface qui permette de facilement mettre en oeuvre des actions de type CRUD
- Eventuellement permettre l'héritage des mappings
- Proposer un langage de requêtes indépendant de la base de données cible et assurer une traduction en SQL natif selon la base utilisée
- Supporter différentes formes d'identifiants générés automatiquement par les bases de données (identity, sequence, ...)
- Proposer un support des transactions
- Assurer une gestion des accès concurrents (verrou, dead lock, ...)
- Fournir des fonctionnalités pour améliorer les performances (cache, lazy loading, ...)

Les solutions de mapping sont donc riches en fonctionnalités ce qui peut rendre leur mise en oeuvre plus ou moins complexe. Cette complexité est cependant différente d'un développement de toute pièce avec JDBC.

Les solutions de mapping O/R permettent de réduire la quantité de code à produire mais impliquent une partie configuration (généralement sous la forme d'un ou plusieurs fichiers XML ou d'annotations pour les solutions reposant sur Java 5).

Depuis quelques années, les principales solutions mettent en oeuvre des POJO (Plain Old Java Object).

59.3.1. Le mapping de l'héritage de classes

L'héritage est un des concepts clé de la programmation orientée objet mais il ne possède pas d'équivalent dans le modèle relationnel.

Le modèle objet supporte des relations de type « est un » ou « possède un ».

Le modèle relationnel supporte seulement les relations de types « possède un »

Il existe cependant plusieurs stratégies standard pour permettre de mapper une hiérarchie de classes dans le modèle relationnel. Chacune de ces stratégies présente des inconvénients.

Le choix de la stratégie doit tenir de deux points particuliers :

- la performance
- la maintenabilité du schéma et des données de la base de données

Le changement de la stratégie de mapping impose une migration plus ou moins lourde des données.

59.3.1.1. Une table par hiérarchie de classes

Dans la stratégie une table par hiérarchie de classes (Table Per Hierarchy : TPH), une seule table est utilisée pour mapper l'ensemble de la hiérarchie de classes.

Cette table contient toutes les propriétés de chaque classe de la hiérarchie. Les propriétés qui n'appartiennent pas à la classe précisée par le discriminant ont pour valeur null. Toutes les colonnes de la table correspondant à une de ces propriétés doivent être nullable. Si la hiérarchie comporte de nombreuses classes ou si le nombre de leurs propriétés est important alors la table est composée de toutes ces propriétés dont la plupart seront null.

Il est important que toutes les propriétés des classes qui sont mappées aient toutes un nom de colonne unique.

Cette table requiert une colonne technique supplémentaire qui sert de discriminant en identifiant la classe concernée pour les données de la ligne. La colonne qui sert de discriminant permet de préciser selon sa valeur à quelle classe correspondent les données de la ligne : elle permet à Hibernate de déterminer le type de la classe qu'il devra instancier lors de la lecture de la ligne dans la base de données.

Il n'est pas possible de définir le type d'une classe fille comme clé étrangère sur l'identifiant de la table puisque toutes les classes sont mappées dans une seule table.

Avantages	Inconvénients
Facile à implémenter Les performances de cette stratégie sont bonnes grâce à une seule instruction select permettant de retrouver les données (aucune jointure ni sous-select ne sont requis) Une seule table à gérer	Chaque changement d'une propriété dans une des classes nécessite de modifier la colonne correspondante dans la table (ajout, suppression, modification). Elle n'est donc pas adaptée si le graphe d'objets est modifié fréquemment. La contrainte NOT NULL ne peut pas être utilisée sur toutes les colonnes qui correspondent à une propriété dans les classes filles La table n'est pas normalisée : elle ne répond pas à la troisième forme normale

Si des requêtes sont faites sur les types des classes filles, il peut être intéressant de définir un index sur la colonne qui sert de discriminant.

59.3.1.2. Une table par sous-classe

La stratégie une table par sous-classe (Table Per Subclass : TPS) propose de convertir la relation "est un" du modèle objet en une relation « possède un » du modèle relationnel en utilisant des clés étrangères. Toutes les sous-classes et la mère si elle est abstraite sont mappées sur leurs propres tables.

Les tables des sous-classes possèdent une clé étrangère qui permet de faire référence à la clé primaire de la table de la classe mère mettant en oeuvre une relation de type 1-1. La clé primaire de la table de la classe mère est utilisée comme clé primaire et clé étrangère dans les tables des classes filles. Hibernate utilise alors des jointures ouvertes dans les requêtes SQL pour obtenir les données.

Cette stratégie de mapping est celle qui est la plus proche du modèle objet. Chaque classe abstraite ou concrète est mappée sur une table dans la base de données. Seule la colonne servant d'identifiant est partagée par toutes les tables : elle permet de faire les jointures requises entre les tables.

Avantages	Inconvénients
Le modèle relationnel est normalisé et l'intégrité des données peut être maintenue Les changements dans la base de données sont simples lorsqu'une modification a lieu dans une classe	Les performances se dégradent lorsque la hiérarchie de classes grossit car le nombre de jointures entre tables augmente dans les requêtes

59.3.1.3. Une table par classe concrète

Dans la stratégie une table par classe concrète (Table Per Concret Class), chaque classe concrète de la hiérarchie possède sa propre table. Les colonnes des classes mères sont dupliquées dans les tables des classes fille. Les classes abstraites ne possèdent pas de table.

Chaque table de la hiérarchie possède des identifiants dédiés : il n'y a pas d'identifiants dupliqués dans ces tables. Ceci permet de garantir l'unicité des identifiants lors de requêtes polymorphiques.

Il n'y a pas de relation entre les tables de la hiérarchie. Il n'est pas nécessaire d'avoir un champ de type discriminant puisque chaque classe possède sa propre table.

Avantages	Inconvénients
C'est la stratégie de mapping la plus simple à implémenter	Les tables ne sont pas normalisées Les colonnes correspondant à des données de classes mères abstraites sont dupliquées dans les tables des classes filles. La modification d'une colonne (ajout / modification / suppression) de la table de la classe mère doit donc être reportée dans toutes les tables des classes filles Le support du polymorphisme est délicat car il nécessite plusieurs requêtes ou l'utilisation d'une clause UNION. Pour effectuer une requête sur le type de la classe mère, il sera nécessaire de faire des requêtes sur chacune des tables Les performances se dégradent si la hiérarchie comporte de nombreuses classes : les requêtes polymorphiques sont coûteuses car elles requièrent l'utilisation d'unions sur les tables de la hiérarchie Il n'est pas possible d'utiliser la colonne identifiant comme clé étrangère car elle est répartie sur plusieurs tables Il n'est pas possible de définir des contraintes d'unicité sur des colonnes d'une classe mère car elles sont dupliquées sur toutes les tables Il n'est pas possible d'utiliser une valeur auto-générée pour les colonnes identifiants car chaque ligne dans toutes les classes de la hiérarchie doivent avoir un identifiant unique

Cette stratégie n'est pas fréquemment utilisée.

59.3.2. Le choix d'une solution de mapping O/R

Comme pour tout choix d'une solution, des critères standard doivent entrer en ligne de compte (prix, complexité de prise en main, performance, maturité, pérennité, support, ...)

Dans le cas d'une solution de mapping O/R, il faut aussi prendre en compte des critères plus spécifiques à ce type de technologie.

La solution doit proposer des fonctionnalités de base :

- gestion de tous les types de relations (1-1, 1-n, n-n)
- langage de requêtes supportant des fonctions avancées de SQL (jointure, groupage, agrégat, ...)
- support des transactions
- support de l'héritage et du polymorphisme
- support de nombreuses bases de données
- gestion des accès concurrents
- support des différents types de clés et des clés composées
- support des mises à jour en cascade
- support du mapping de type une classe contenant des données de plusieurs tables ou l'inverse
- configuration par fichiers ou annotations
- ...

La prise en compte des performances et des optimisations proposées par la solution est très importante :

- chargement différé (lazy loading) au niveau d'un champ
- gestion de caches au niveau des données et des requêtes
- optimisation des requêtes
- ...

Il est aussi nécessaire de tenir compte des outils proposés par la solution pour faciliter sa mise en oeuvre. Ces outils peuvent par exemple :

- permettre l'automatisation de la génération des classes ou des schémas de la base de données
- faciliter la rédaction et la maintenance des fichiers de configuration
- ...

Certaines fonctionnalités avancées peuvent être utiles voire requises en fonction des besoins :

- mise en oeuvre de POJO
- support du mode déconnecté
- support des procédures stockées
- ...

59.3.3. Les difficultés lors de la mise en place d'un outil de mapping O/R

De nombreuses difficultés peuvent survenir lors de la mise en oeuvre d'un outil de mapping O/R

- Difficultés à mapper le modèle relationnel à cause de la complexité du modèle ou de sa mauvaise conception
- Temps d'apprentissage de l'outil plus ou moins important
- Difficultés pour mettre en oeuvre les transactions
- Parfois des problèmes de performance peuvent nécessiter un paramétrage plus fin notamment en se qui concerne la mise en oeuvre de caches ou du chargement tardif (lazy loading)
- Difficultés pour maintenir les fichiers de configuration généralement au format XML et à les synchroniser avec les évolutions du modèle de données. Des outils existent pour certaines solutions afin de faciliter cette tâche.

59.4. L'architecture et la persistance de données

Dans une architecture en couches, il est important de prévoir une couche dédiée aux accès aux données.

Il est assez fréquent dans cette couche de parler de la notion de CRUD qui représente un ensemble des 4 opérations de bases réalisables sur des données.

Il est aussi de bon usage de mettre en oeuvre le design pattern DAO (Data Access Object) proposé par Sun.

59.4.1. La couche de persistance

La partie du code responsable de l'accès aux données dans une application multiniveau doit être encapsulée dans une couche dédiée aux interactions avec la base de données de l'architecture généralement appelée couche de persistance. Celle-ci permet notamment :

- d'ajouter un niveau d'abstraction entre la base de données et l'utilisation qui en est faite.
- de simplifier la couche métier qui utilise les traitements de cette couche
- de masquer les traitements réalisés pour mapper les objets dans la base de données et vice versa
- de faciliter le remplacement de la base de données utilisée

La couche métier qui va utiliser la couche de persistance reste indépendante du code dédié à l'accès à la base de données. Ainsi la couche métier ne contient aucune requête SQL, ni code de connexion ou d'accès à la base de données. La couche métier utilise les classes de la couche persistance qui encapsulent ces traitements. Ainsi la couche métier manipule uniquement des objets pour les accès à la base de données.

Le choix des API ou des outils dépend du contexte : certaines solutions ne sont utilisables qu'avec la plate-forme Enterprise Edition (exemple : les EJB) ou sont utilisables indifféremment avec les plates-formes Standard et Enterprise Edition.

L'utilisation d'une API standard permet de garantir la pérennité et de choisir l'implémentation à mettre en oeuvre.

Les solutions open source et commerciales ont les avantages et inconvénients inhérents à leur typologie respective.

59.4.2. Les opérations de type CRUD

L'acronyme CRUD (Create, Read, Update and Delete) désigne les quatre opérations réalisables sur des données (création, lecture, mise à jour et suppression).

Exemple : une interface qui propose des opérations de type CRUD pour un objet de type Entite

```
public interface EntiteCrud {
  public Entite obtenir(Integer id);
  public void creer(Entite entite);
  public void modifier(Entite entite);
  public Collection obtenirTous();
  public void supprimer(Entite entite);
}
```

59.4.3. Le modèle de conception DAO (Data Access Object)

DAO est l'acronyme de Data Access Object. C'est un modèle de conception qui propose de découpler l'accès à une source de données.

L'accès aux données dépend fortement de la source de données. Par exemple, l'utilisation d'une base de données est spécifique pour chaque fournisseur. Même si SQL et JDBC assurent une partie de l'indépendance vis-à-vis de la base de données utilisée, certaines contraintes imposent une mise en oeuvre spécifique de certaines fonctionnalités.

Par exemple, la gestion des champs de type identifiants est proposée selon diverses formes par les bases de données : champ auto-incrémenté, identity, séquence, ...

Le motif de conception DAO proposé dans le blue print de Sun propose de séparer les traitements d'accès physique à une source de données de leur utilisation dans les objets métiers. Cette séparation permet de modifier une source de données sans avoir à modifier les traitements qui l'utilisent.

Le DAO peut aussi proposer un mécanisme pour rendre l'accès aux bases de données indépendant de la base de données utilisée et même rendre celle-ci paramétrable.

Les classes métier utilisent le DAO par son interface et sont donc indépendantes de son implémentation. Si cette implémentation change (par exemple un changement de base de données), seul l'implémentation du DAO est modifiée mais les classes qui l'utilisent via son interface ne sont pas impactées.

Le DAO définit donc une interface qui va exposer les fonctionnalités utilisables. Ces fonctionnalités doivent être indépendantes de l'implémentation sous-jacente. Par exemple, aucune méthode ne doit avoir de requêtes SQL en paramètre. Pour les même raisons, le DAO doit proposer sa propre hiérarchie d'exceptions.

Une implémentation concrète de cette interface doit être proposée. Cette implémentation peut être plus ou moins complexe en fonction de critères de simplicité ou de flexibilité.

Fréquemment les DAO ne mettent pas en oeuvre certaines fonctionnalités comme la mise en oeuvre d'un cache ou la gestion des accès concurrents.

59.5. Les différentes solutions

Différentes solutions peuvent être utilisées pour la persistance des objets en Java :

- Sérialisation
- JDBC
- Génération automatisée de code source
- SQL/J
- Enrichissement du code source (enhancement)
- Génération de bytecode
- framework de mapping O/R (Object Relational Mapping)
- Base de données objet (ODBMS)

La plupart de ces solutions offre de surcroît un choix plus ou moins important d'implémentations.

59.6. Les API standards

Les différentes évolutions de Java ont apporté plusieurs solutions pour assurer la persistance des données vers une base de données.

59.6.1. JDBC

JDBC est l'acronyme de Java DataBase Connectivity. C'est l'API standard pour permettre un accès à une base de données. Son but est de permettre de coder des accès à une base de données en laissant le code le plus indépendant de la base de données utilisée.

C'est une spécification qui définit des interfaces pour se connecter et interagir avec la base de données (exécution de requêtes ou de procédures stockées, parcours des résultats des requêtes de sélection, ...)

L'implémentation de ces spécifications est fournie par des tiers, et en particulier les fournisseurs de base de données, sous la forme de Driver.

La mise en oeuvre de JDBC est détaillée dans le chapitre JDBC

59.6.2. JDO 1.0

JDO est l'acronyme de Java Data Object : le but de cette API est de rendre transparente la persistance d'un objet. Elle repose sur l'enrichissement du bytecode à la compilation.

Cette API a été spécifiée sous la JSR-012.

Il existe plusieurs implémentations dont :

- Apache JDO (https://db.apache.org/jdo/)
- JPOX
- Xcalia LiDO
- Kodo (http://www.solarmetric.com/) - BEA
- Speedo (http://speedo.objectweb.org/)

La mise en oeuvre de JDO est détaillée dans le chapitre JDO

59.6.3. JD0 2.0

La version 2 de JDO a été diffusée en mars 2006.

Elle a été spécifiée sous la JSR-243.

Il existe plusieurs implémentations dont :

- Apache JDO (http://db.apache.org/jdo/)
- JPOX : c'est l'implémentation de référence (RI) pour JDO 2.0
- Kodo

59.6.4. EJB 2.0

Les EJB (Enterprise Java Bean) proposent des beans de type Entités pour assurer la persistance des objets.

Les EJB de type Entité peuvent être de deux types :

- CMP (Container Managed Persistance) : la persistance est assurée par le conteneur d'EJB en fonction du paramétrage fourni
- BMP (Bean Managed Persistance) :

Les EJB bénéficient des services proposés par le conteneur cependant cela les rend dépendants de ce conteneur pour l'exécution : ils sont difficilement utilisables en dehors du conteneur (par exemple pour les tester).

Il existe de nombreuses implémentations puisque chaque serveur d'applications certifié J2EE doit implémenter les EJB ce qui inclut entre autres JBoss de RedHat, JonAS, Geronimo d'Apache, GlassFish de Sun/Oracle, Websphere d'IBM, Weblogic de BEA, ...

Les Entity Beans CMP 2.x ont été déclarés pruned dans Java EE 6.

59.6.5. Java Persistence API et EJB 3.0

JPA (Java Persistence API) est issu des travaux de la JSR-220 concernant la version 3.0 des EJB : elle remplace d'ailleurs les EJB Entités version 2. C'est une synthèse standardisée des meilleurs outils du sujet (Hibernate, Toplink/EclipseLink, ...)

L'API repose sur

- l'utilisation d'entités persistantes sous la forme de POJOs
- un gestionnaire de persistance (EntityManager) qui assure la gestion des entités persistantes
- l'utilisation d'annotations
- la configuration via des fichiers xml

JPA peut être utilisé avec Java EE dans un serveur d'applications mais aussi avec Java SE (avec quelques fonctionnalités proposées par le conteneur en moins).

JPA est une spécification : il est nécessaire d'utiliser une implémentation pour la mettre en oeuvre. L'implémentation de référence est la partie open source d'Oracle Toplink : Toplink essential. La version 3.2 d'Hibernate implémente aussi JPA.

JPA ne peut être utilisé qu'avec des bases de données relationnelles.

La version 3.0 des EJB utilise JPA pour la persistance des données.

La mise en oeuvre de JPA est détaillée dans le chapitre JPA

59.7. Les frameworks open source

Pour palier certaines faiblesses des API standards, la communauté open source a développé de nombreux frameworks concernant la persistance de données dont le plus utilisé est Hibernate. Cette section va rapidement présenter quelques-uns d'entre-eux.

59.7.1. iBatis

Le site officiel du projet est à l'url : https://ibatis.apache.org/

59.7.2. MyBatis

Le site officiel du projet est à l'url : https://blog.mybatis.org/

59.7.3. Hibernate

Hibernate est le framework open source de mapping O/R le plus populaire. Cette popularité est liée à la richesse des fonctionnalités proposées et à ses performances.

Hibernate propose son propre langage d'interrogation HQL et a largement inspiré les concepteurs de l'API JPA. Hibernate est un projet open source de mapping O/R qui fait référence en la matière car il possède plusieurs avantages :

- manipulation de données d'une base de données relationnelles à partir d'objets Java
- facile à mettre en oeuvre, efficace et fiable
- open source

Le site officiel du projet est à l'url : https://hibernate.org/

L'utilisation d'Hibernate est détaillée dans le chapitre Hibernate

59.7.4. EclipseLink

Le site officiel du projet est à l'url : https://www.eclipse.org/eclipselink/

EclipseLink est un projet open source de la fondation Eclipse qui propose un framework extensible permettant de se connecter à une source de données : base de données (JPA), Moxy pour XML (JAXB), Java Connector Architecture (JCA), Service Data Object (SDO).

EclipseLink est l'implémentation de référence de JPA 2.0.

59.7.5. Castor

Castor permet de mapper des données relationnelles ou XML avec des objets Java.

Castor propose une solution riche mais qui n'implémente aucun standard.

Le site officiel du projet est à l'url : https://castor.exolab.org/xml-framework.html

59.7.6. Apache Torque

Torque est un framework initialement développé pour le projet Jakarta Turbine : il est développé depuis sous la forme d'un projet autonome.

Torque se compose d'un générateur qui va générer automatiquement les classes requises pour accéder à la base de données et d'un environnement d'exécution qui va permettre la mise en oeuvre des classes générées.

Torque utilise un fichier XML contenant une description de la base de données pour générer des classes permettant des opérations sur la base de données grâce à des outils dédiés. Ces classes reposent sur un environnement d'exécution (runtime) fourni par Torque.

Le site officiel du projet est à l'url https://db.apache.org/torque/

59.7.7. TopLink

TopLink a été racheté par Oracle et intégré dans ses solutions J2EE.

Le site officiel du produit est à l'url : https://www.oracle.com/middleware/technologies/top-link.html

La partie qui compose la base de TopLink est en open source.

59.7.8. Apache OJB

Le site officiel du projet est à l'url https://db.apache.org/ojb/

59.7.9. Apache Cayenne

Le site officiel du projet est à l'url https://cayenne.apache.org/

Cayenne est distribué avec un outil de modélisation nommé CayenneModeler.

59.8. L'utilisation de procédures stockées

L'utilisation de procédures stockées peut apporter des améliorations notamment en termes de performance de certains traitements.

Cependant, les procédures stockées possèdent plusieurs inconvénients :

- peu portable
- peu flexible
- maintenance généralement difficile liée au manque d'outillage
- ...

60. JDBC

Niveau : 🟩🟩🟨⬜⬜⬜ Intermédiaire

JDBC est une marque déposée de Sun/Oracle, souvent considéré comme étant l'acronyme de Java DataBase Connectivity et désigne une API de bas niveau de Java SE pour permettre un accès aux bases de données avec Java.

Elle permet de se connecter à une base de données et d'interagir avec notamment en exécutant des requêtes SQL.

L'architecture de JDBC permet d'utiliser la même API pour accéder aux différentes bases de données grâce à l'utilisation de pilotes (drivers) qui fournissent une implémentation spécifique à la base de données à utiliser. Chaque base de données a alors la responsabilité de fournir un pilote qui assure l'interface entre l'API et les actions exécutées de manière propriétaire sur la base de données.

JDBC a connu plusieurs versions livrées dans différentes versions du JDK.

Version de JDBC	Spécification	Version du JDK
1.0		1.1
2.0 (JDBC Core 2.1 et JDBC Optional 2.0)		1.2
3.0	JSR 54	1.4
4.0	JSR 221	1.6
4.1	JSR 221	1.7
4.2	JSR 221	1.8
4.3	JSR 221	9

Initialement l'API JDBC est contenue dans le package java.sql.

A partir de JDBC 2.0, l'API est contenue dans deux packages :

- java.sql : il contient l'API de base JDBC nommée JDBC Core
- javax.sql : il contient l'API facultative de JDBC. Ce package étend les fonctionnalités de l'API JDBC d'une API côté client à une API côté serveur, et il constitue un élément de la plateforme Java EE

A partir de Java 9, l'API JDBC est dans le module java.sql.

Ce chapitre présente dans plusieurs sections l'utilisation de cette API :

♦ Les outils nécessaires pour utiliser JDBC
♦ Les types de pilotes JDBC
♦ La présentation des classes et interfaces de base de l'API JDBC
♦ La connexion à une base de données
♦ L'accès à la base de données
♦ L'obtention d'informations sur la base de données

- L'utilisation d'un objet de type PreparedStatement
- L'utilisation des transactions
- Les procédures stockées
- Le traitement des erreurs JDBC
- Les évolutions de l'API JDBC
- MySQL et Java
- L'amélioration des performances avec JDBC

60.1. Les outils nécessaires pour utiliser JDBC

Les classes de JDBC version 1.0 sont regroupées dans le package java.sql et sont incluses dans le JDK à partir de sa version 1.1.

Pour pouvoir utiliser JDBC, il faut un pilote qui est spécifique à la base de données à laquelle on veut accéder. Ce pilote permet de réaliser l'indépendance de JDBC vis à vis des bases de données.

60.2. Les types de pilotes JDBC

Il existe quatre types de pilote JDBC :

- Type 1 (JDBC-ODBC bridge) : le pont JDBC-ODBC qui s'utilise avec ODBC et un pilote ODBC spécifique pour la base à accéder.

  Cette solution fonctionnait très bien sous Windows. C'était une solution pour des développements avec exécution sous Windows d'une application locale qui avait le mérite d'être universelle car il existait des pilotes ODBC pour la quasi totalité des bases de données. Cette solution "simple" pour le développement possède cependant plusieurs inconvénients :

  - la multiplication du nombre de couches rend complexe l'architecture (bien que transparent pour le développeur) et détériore un peu les performances
  - lors du déploiement, ODBC et son pilote doivent être installés sur tous les postes où l'application va fonctionner
  - la partie native (ODBC et son pilote) rend l'application moins portable et dépendante d'une plate-forme

  Pour utiliser le pont JDBC-ODBC sous Window 9x, il faut utiliser ODBC en version 32 bits.

  A partir de Java 8, l'implémentation du pilote JDBC-ODBC bridge n'est plus fournie dans le JDK.
- Type 2 : un driver écrit en Java qui appelle l'API native de la base de données

  Ce type de driver convertit les ordres JDBC pour appeler directement les API de la base de données via un pilote natif sur le client. Ce type de driver nécessite aussi l'utilisation de code natif sur le client.
- Type 3 : un driver écrit en Java utilisant un middleware

  Ce type de driver utilise un protocole réseau propriétaire spécifique à une base de données. Un serveur dédié reçoit les messages par ce protocole et dialogue directement avec la base de données. Ce type de driver peut être facilement utilisé par une applet mais dans ce cas le serveur intermédiaire doit obligatoirement être installé sur la machine contenant le serveur web.
- Type 4 : un driver Java utilisant le protocole natif de la base de données

  Ce type de driver, écrit en Java, appelle directement le SGBD par le réseau. Il est fourni par l'éditeur de la base de données.

Les drivers se présentent souvent sous forme de fichiers jar dont le chemin doit être ajouté au classpath pour permettre à la JVM de pouvoir en charger les classes à utiliser.

60.2.1. L'enregistrement d'une base de données dans ODBC sous Windows 9x ou XP

Pour utiliser un pilote de type 1 (pont ODBC-JDBC) sous Windows 9x, il est nécessaire d'enregistrer la base de données dans ODBC avant de pouvoir l'utiliser.

Attention : ODBC n'est pas fourni en standard avec Windows 9x.

Pour enregistrer une nouvelle base de données, il faut utiliser l'administrateur de source de données ODBC.

Pour lancer cette application sous Windows 9x, il faut double-cliquer sur l'icône "ODBC 32bits" dans le panneau de configuration.

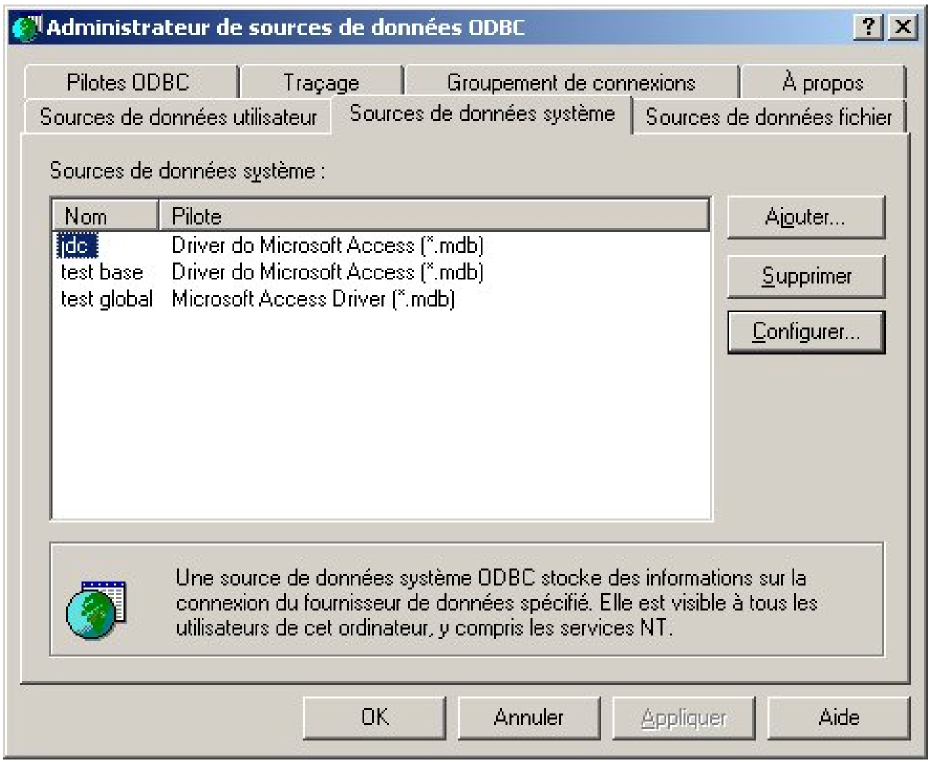
ODBC Data Sources (32bit)

Sous Windows XP, il faut double cliquer sur l'icône "Source de données (ODBC)" dans le répertoire "Outils d'administration" du panneau de configuration.

Sources de données (ODBC)

L'outil se compose de plusieurs onglets :

- L'onglet "Pilote ODBC" liste l'ensemble des pilotes qui sont installés sur la machine.
- L'onglet "Source de données utilisateur" liste l'ensemble des sources de données pour l'utilisateur couramment connecté sous Windows.
- L'onglet "Source de données système" liste l'ensemble des sources de données accessibles par tous les utilisateurs.

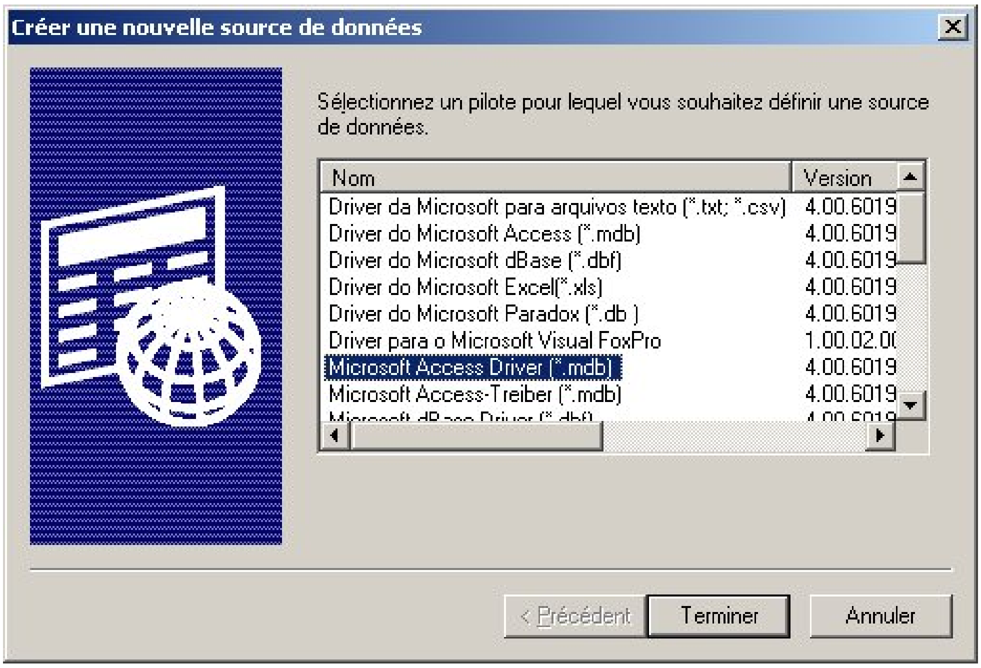

Administrateur de sources de données ODBC
? X
Pilotes ODBC
Traçage
Groupement de connexions
À propos
Sources de données utilisateur
Sources de données système
Sources de données fichier
Ajouter...
Supprimer
Configurer...
OK
Annuler
Appliquer
Aide

Sources de données système :

Nom	Pilote
jdc	Driver do Microsoft Access (*.mdb)
test base	Driver do Microsoft Access (*.mdb)
test global	Microsoft Access Driver (*.mdb)

Une source de données système ODBC stocke des informations sur la connexion du fournisseur de données spécifié. Elle est visible à tous les utilisateurs de cet ordinateur, y compris les services NT.

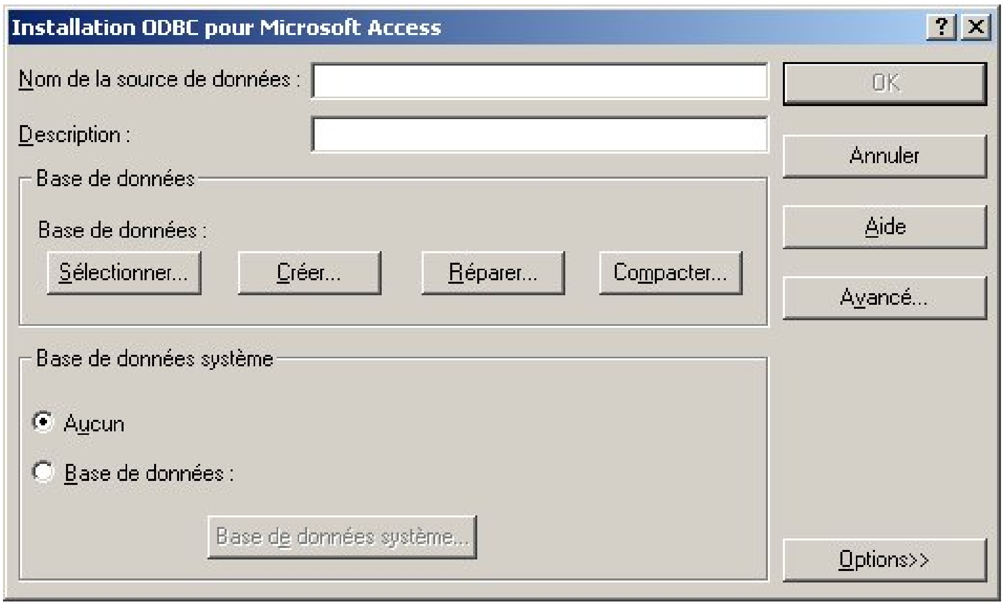
Le plus simple est de créer une telle source de données en cliquant sur le bouton "Ajouter". Une boîte de dialogue permet de sélectionner le pilote qui sera utilisé par la source de données.


Créer une nouvelle source de données

Sélectionnez un pilote pour lequel vous souhaitez définir une source de données.

Nom	Version
Driver da Microsoft para arquivos texto (*.txt; *.csv)	4.00.6019
Driver do Microsoft Access (*.mdb)	4.00.6019
Driver do Microsoft dBase (*.dbf)	4.00.6019
Driver do Microsoft Excel(*.xls)	4.00.6019
Driver do Microsoft Paradox (*.db)	4.00.6019
Driver para o Microsoft Visual FoxPro	1.00.02.0(
Microsoft Access Driver (*.mdb)	4.00.6019
Microsoft Access-Treiber (*.mdb)	4.00.6019

Il suffit de sélectionner le pilote et de cliquer sur "Terminer". Dans l'exemple ci-dessous, le pilote sélectionné concerne une base Microsoft Access.


Installation ODBC pour Microsoft Access

Nom de la source de données :

Description :

Base de données

Base de données :

Base de données système

● Aucun

○ Base de données :

Il suffit de saisir les informations nécessaires notamment le nom de la source de données et de sélectionner la base. Un clic sur le bouton "Ok" crée la source de données qui pourra alors être utilisée.

60.3. La présentation des classes et interfaces de base de l'API JDBC

Les classes et interface de base de l'API JDBC sont dans le package java.sql.

Les 4 types de base de JDBC sont : DriverManager, Connection, Statement, et ResultSet, chacune correspondant à une étape de l'accès aux données.

Classe/interface	Rôle
DriverManager	Charger et configurer le driver de la base de données
Connection	Réaliser la connexion et l'authentification à la base de données

Statement (et ses interfaces filles PreparedStatement et CallableStatement)	Encapsuler la requête SQL et la transmettre pour exécution à la base de données
ResultSet	Parcourir les informations retournées par la base de données dans le cas d'une sélection de données

Chacune de ces classes dépend de l'instanciation d'un objet de la précédente classe car l'utilisation de JDBC pour interagir avec une base de données requière plusieurs étapes :

- Obtenir une instance de Connection qui se connecte à la base de données
- Obtenir une instance de Statement à partir de la connexion
- Configurer et exécuter une requête SQL via le Statement
- Exploiter les résultats retournés par la base de données
- Fermer les différentes instances utilisées

60.4. La connexion à une base de données

La connexion à une base de données requiert au préalable le chargement du pilote JDBC qui sera utilisé pour communiquer avec la base de données. Il faut ajouter au classpath le fichier jar du pilote JDBC pour la base de données à utiliser, pour que le classe d'implémentation de l'interface java.sql.Driver puisse être trouvée et chargée.

Avant Java 6, la classe d'implémentation du pilote doit être chargée explicitement avant toute utilisation.

A partir de Java 6, si le pilote est compatible avec JDBC 4.0, alors la classe d'implémentation du pilote sera trouvée et chargée automatiquement.

Une fabrique permet alors de créer une instance de type Connection qui va encapsuler la connexion à la base de données.

60.4.1. Le chargement explicite du pilote avant Java 6

Avant Java 6, il faut obligatoirement, avant toute utilisation de l'API JDBC, charger explicitement la classe d'implémentation du pilote. Cela peut se faire de plusieurs manières :

- utiliser la méthode static forName() de la classe Class en lui passant en paramètre le nom pleinement qualifié de la classe
- utiliser la méthode static registerDriver() de classe DriverManager qui attend en paramètre une instance de type java.sql.Driver

La classe à charger est spécifique à chaque fournisseur. Pour se connecter à une base en utilisant un driver spécifique, la documentation du driver fournit le nom de la classe d'implémentation du pilote à utiliser. Par exemple, si le nom de la classe est jdbc.DriverXXX, le chargement du driver se fera avec le code suivant :

```
Class.forName("jdbc.DriverXXX");
```

Exemple de classes d'implémentation de pilotes pour différentes bases de données

Base de données	Classe d'implémentation
Derby	org.apache.derby.jdbc.EmbeddedDriver
HSQLDB	org.hsqldb.jdbcDriver
H2	org.h2.Driver
IBM DB2 UDB Type 4	com.ibm.db2.jcc.DB2Driver
MariaDB Connector/J	org.mariadb.jdbc.Driver

Microsoft SQL Server	com.microsoft.sqlserver.jdbc.SQLServerDriver
MySQL Connector/J 5.1	com.mysql.jdbc.Driver
MySQL Connector/J 8.0	com.mysql.cj.jdbc.Driver
Oracle Thin Client	oracle.jdbc.driver.OracleDriver
Oracle OCI	oracle.jdbc.driver.OracleDriver
PostgreSQL	org.postgresql.Driver
Sybase jConnect 6.0	com.sybase.jdbc3.jdbc.SybDriver
Sybase jConnect 7.0	com.sybase.jdbc4.jdbc.SybDriver

Exemple : Chargement du pilote pour une base PostgreSQL

```
    Class.forName("org.postgresql.Driver");
```

Autre exemple, pour se connecter à une base de données via ODBC, il faut tout d'abord charger le pilote JDBC-ODBC qui fait le lien entre les deux.

Exemple (code Java 1.1) :

```
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Il n'est pas nécessaire de créer une instance de cette classe et de l'enregistrer avec le DriverManager car le chargement de la classe avec Class.forName() le fait automatiquement : ce traitement charge la classe du pilote et exécute une méthode statique de la classe qui enregistre le pilote auprès du DriverManager.

La méthode static forName() de la classe Class peut lever une exception de type java.lang.ClassNotFoundException si le nom de la classe fournie en paramètre ne peut pas être trouvée dans le classpath..

60.4.2. Le chargement du pilote par le DriverManager

Il est possible d'utiliser la méthode la static registerDriver() de classe DriverManager qui attend en paramètre une instance de type java.sql.Driver.

Il est aussi possible d'utiliser la propriété de la JVM jdbc.drivers :

- soit en paramètre de la ligne de commande de la JVM, exemple : java -Djdbc.drivers=org.postgresql.Driver Monapp
- soit par programmation, exemple : System.setProperty("jdbc.drivers", "org.postgresql.Driver");

Durant son initialisation, la classe DriverManager tentera de charger les pilotes JDBC disponibles précisés :

- par la propriété système jdbc.drivers. La valeur de cette propriété contient une liste de noms de classes pleinement qualifiés de pilotes JDBC. Chaque pilote est chargé à l'aide du système classloader
- à partir de JDBC 4.0, par les implémentations de la classe java.sql.Driver fournies en tant que services et qui sont chargés via le mécanisme de chargement des fournisseurs de services. Les pilotes JDBC 4.0 doivent prendre en charge le mécanisme de fourniture de services pour les implémentations de java.sql.Driver

60.4.3. L'établissement de la connexion

Pour se connecter à une base de données, il faut obtenir une instance de type Connection en invoquant la méthode getconnection() de la classe DriverManager en lui précisant sous forme d'une URL la base de données à accéder. Lorsque la méthode getConnection() est invoquée, le DriverManager tente de trouver un pilote approprié parmi ceux qui ont été chargés à l'initialisation et ceux qui ont été chargés explicitement en utilisant le même classloader que l'application courante.

La syntaxe de l'URL peut varier d'un type de base de données à l'autre mais elle est généralement de la forme :

```
jdbc:<subprotocol>:<subname>
```

- subprotocol correspond au sous-protocole qui définit le type de mécanisme de connectivité à la base de données qui peut être pris en charge par un ou plusieurs pilotes
- le contenu et la syntaxe du subname dépendent du sous-protocole

Exemple (code Java 1.1) : Etablir une connexion sur la base testDB via ODBC

```
    String urlDB = "jdbc:odbc:testDB";

    con = DriverManager.getConnection(urlDB);
```

Dans l'URL de l'exemple ci-dessus :

- "jdbc" désigne le protocole et vaut toujours "jdbc"
- "odbc" désigne le sous protocole qui définit le mécanisme de connexion pour un type de base de données
- "testDB" est le nom de la base de données

Il faut impérativement consulter la documentation du pilote JDBC pour connaître la syntaxe exacte de l'URL selon le pilote utilisé : elle devra notamment indiquer le sous-protocole à utiliser (celui à mettre derrière jdbc dans l'URL).

Exemple d'URL de connexion à différentes bases de données exécutées en local

Base de données	Exemple d'URL
Apache Derby embarqué	`jdbc:derby:appdb;create=true`
Apache Derby	`jdbc:derby://localhost:1527/appdb;create=true`
H2 embarqué	`jdbc:h2:mem:appdb`
H2	`jdbc:h2:tcp://localhost/~/appdb`
HSQLDB	`jdbc:hsqldb:hsql://localhost:9001/appdb`
MariaDB	`jdbc:mariadb://localhost:3306/appdb`
MySQL	`jdbc:mysql://localhost:3306/appdb`
Oracle thin client	`jdbc:oracle:thin:@localhost:1521/appservice`
PostgreSQL	`jdbc:postgresql://localhost:5432/appdb`
SQL Server	`jdbc:sqlserver://localhost:1433/appdb`

La méthode getConnection() peut lever une exception de type java.sql.SQLException notamment si aucun pilote correspond au sous-protocole de l'URL n'est trouvé.

Exemple :

```
java.sql.SQLException: No suitable driver found for jdbc:derby:appdb;create=true
        at java.sql/java.sql.DriverManager.getConnection(DriverManager.java:706)
        at java.sql/java.sql.DriverManager.getConnection(DriverManager.java:252)
        at fr.jmdoudoux.dej.jdbc.MonApp.main(MonApp.java:11)
```

Une seconde surcharge de la méthode getConnection() attend en paramètres en plus de l'url, l'utilisateur et le mot de passe à utiliser pour se connecter à la base de données.

Exemple (code Java 1.1) :

```
    Connection con = DriverManager.getConnection(url, "admin", "motdepasse");
```

Dans l'exemple ci-dessus, l'utilisateur utilisé est "admin"avec le mot de passe "motdepasse".

Exemple : Connection à la base PostgreSQL nommée test avec le user adm et le mot de passe 12345 sur la machine locale

```
Connection con=DriverManager.getConnection("jdbc:postgresql://localhost/test","adm","12345");
```

Important : lorsque la connexion n'est plus utile, il faut explicitement invoquer sa méthode close() afin de libérer toutes les autres ressources de la base de données que la connexion peut conserver.

Exemple : Connexion à une base de données Derby

```
package fr.jmdoudoux.dej.jdbc;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class MonApp {

  public static void main(String[] args) {
    String urlDB = "jdbc:derby:appdb;create=true";
    Connection con = null;
    try {
      con = DriverManager.getConnection(urlDB);
    } catch (SQLException e) {
      e.printStackTrace();
    } finally {
      if (con != null) {
        try {
          con.close();
        } catch (SQLException e) {
          e.printStackTrace();
        }
      }
    }
  }
}
```

A partir de JDBC 4.1, l'interface Connection implémente l'interface java.lang.AutoClosable ce qui permet son utilisation dans une instruction try-with-resources qui va faire généer par le compilateur le code requis pour l'invocation de la méthode. Cela simplifie le code et le rend plus sûr notamment par ce qu'il n'y a pas de risque d'oublier l'invocation de la méthode close().

Exemple : Connexion à une base de données Derby

```
package fr.jmdoudoux.dej.jdbc;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class MonApp {

  public static void main(String[] args) {
    String urlDB = "jdbc:derby:appdb;create=true";
    try ( Connection con = DriverManager.getConnection(urlDB)) {
      // ...
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
}
```

60.5. L'accès à la base de données

Une fois la connexion établie, il est possible d'interagir avec une base de données, notamment pour exécuter des requêtes SQL ou obtenir des méta-donnés.

60.5.1. L'exécution de requêtes SQL

Les requêtes SQL sont exécutées avec les méthodes d'un objet de type Statement que l'on obtient à partir d'un objet Connection.

Une instance de type Statement est utilisée pour exécuter une requête SQL statique et renvoyer les résultats qu'elle produit.

Une instance de l'interface Statement permet de soumettre des requêtes SQL, sans support de paramètres d'entrée, à la base de données. Pour obtenir une instance de type Statement, il faut invoquer la méthode createStatement() sur un objet de type Connection :

Exemple (code Java 1.1) :

```
    Statement stmt = con.createStatement();
```

Il est possible de créer plusieurs instances de type Statement à partir d'une même instance de de type Connection.

L'invocation de la méthode close() d'une instance de type Statement permet d'indiquer que son exploitation est terminée.

Important : lorsque l'instance de Statement n'est plus utile, il faut explicitement invoquer sa méthode close() afin de libérer toutes les autres ressources.

A partir de JDBC 4.1, l'interface Statement implémente l'interface java.lang.AutoClosable ce qui permet son utilisation dans une instruction try-with-resources.

La méthode à utiliser pour exécuter un objet de type Statement dépend du type de requête SQL qu'il encapsule.

Si l'objet Statement encapsule une requête SQL avec une instruction SELECT alors il faut invoquer la méthode executeQuery() qui renvoie un objet de type ResultSet. Par défaut, un seul objet ResultSet par objet Statement peut être ouvert en même temps. Par conséquent, si la lecture d'un objet ResultSet est entrecoupée de la lecture d'un autre, chacun doit avoir été obtenu par des objets de type Statement différents. Toutes les méthodes d'exécution de l'interface Statement ferment implicitement un objet ResultSet s'il en existe un ouvert.

Si l'objet Statement encapsule une requête SQL qui modifie des données alors il faut invoquer la méthode executeUpdate() qui retourne un entier de type int dont la valeur indique le nombre d'enregistrements impactés.

Si l'objet Statement encapsule une requête SQL donc le type n'est pas connu alors il faut invoquer la méthode execute() qui renvoient un booléen indiquant selon sa valeur la forme du premier résultat :

- true si le premier résultat est un objet ResultSet
- false si c'est un entier qui indique le nombre de mises à jour ou s'il n'y a pas de résultats

Selon la valeur retournée, il faut invoquer les méthodes getResultSet() ou getUpdateCount() pour récupérer le résultat, et getMoreResults() pour passer aux résultats suivants.

La méthode à utiliser pour soumettre la requête à la base de données dépend du type de la requête soumise :

- Pour une requête de type interrogation (SELECT), il faut utiliser la méthode executeQuery() de l'interface Statement
- Pour une requête de mise à jour, il faut utiliser la méthode executeUpdate()

Lors de l'appel à la méthode d'exécution, il est nécessaire de lui fournir en paramètre la requête SQL sous forme de chaîne de caractères.

Exemple (code Java 1.1) :

```
    ResultSet resultats = null;
    String requete = "SELECT * FROM client";

    try {
      Statement stmt = con.createStatement();
      resultats = stmt.executeQuery(requete);
    } catch (SQLException e) {
      //traitement de l'exception
    }
```

Le résultat d'une requête d'interrogation est renvoyé dans un objet de la classe ResultSet par la méthode executeQuery().

Exemple (code Java 1.1) :

```
    ResultSet rs = stmt.executeQuery("SELECT * FROM employe");
```

La méthode executeUpdate() retourne le nombre d'enregistrements qui ont été mis à jour

Exemple (code Java 1.1) :

```
...

//insertion d'un enregistrement dans la table client

requete = "INSERT INTO client VALUES (3,'client 3','prenom 3')";
try {
   Statement stmt = con.createStatement();
   int nbMaj = stmt.executeUpdate(requete);
   affiche("nb mise a jour = "+nbMaj);
} catch (SQLException e) {
   e.printStackTrace();
}

...
```

Lorsque la méthode executeUpdate() est utilisée pour exécuter un traitement de type DDL (Data Definition Langage : définition de données) comme la création d'un table, elle retourne 0. Si la méthode retourne 0, cela peut signifier deux choses : le traitement de mise à jour n'a affecté aucun enregistrement ou le traitement concernait un traitement de type DDL.

Si l'on utilise executeQuery() pour exécuter une requête SQL ne contenant pas d'ordre SELECT, alors une exception de type SQLException est levée.

Exemple (code Java 1.1) :

```
...

    requete = "INSERT INTO client VALUES (4,'client 4','prenom 4')";
    try {
       Statement stmt = con.createStatement();
       ResultSet resultats = stmt.executeQuery(requete);
    } catch (SQLException e) {
       e.printStackTrace();
    }

...
```

Résultat :

```
java.sql.SQLException: No ResultSet was produced
java.lang.Throwable(java.lang.String)
java.lang.Exception(java.lang.String)
java.sql.SQLException(java.lang.String)
java.sql.ResultSet sun.jdbc.odbc.JdbcOdbcStatement.executeQuery(java.lang.String)
void testjdbc.TestJDBC1.main(java.lang.String [])
```

Attention : dans ce cas la requête est quand même effectuée. Dans l'exemple, un nouvel enregistrement est créé dans la table.

Il n'est pas nécessaire de définir un objet Statement pour chaque ordre SQL : il est possible d'en définir un et de le réutiliser

60.5.2. Le parcours des enregistrements obtenus en retour

La classe ResultSet représente une abstraction des résultats de l'exécution d'une requête SQL qui se compose de plusieurs enregistrements constitués de colonnes qui contiennent les données.

Les principales méthodes pour obtenir des données sont :

Méthode	Rôle
getInt(int)	Retourner sous forme d'entier le contenu de la colonne dont le numéro est passé en paramètre
getInt(String)	Retourner sous forme d'entier le contenu de la colonne dont le nom est passé en paramètre
getFloat(int)	Retourner sous forme d'un nombre flottant le contenu de la colonne dont le numéro est passé en paramètre
getFloat(String)	Retourner sous forme d'un nombre flottant le contenu de la colonne dont le nom est passé en paramètre
getDate(int)	Retourner sous forme de date le contenu de la colonne dont le numéro est passé en paramètre
getDate(String)	Retourner sous forme de date le contenu de la colonne dont le nom est passé en paramètre
next()	Se déplacer sur le prochain enregistrement : retourne false si la fin est atteinte
close()	Fermer le ResultSet
getMetaData()	Retourner un objet de type ResultSetMetaData associé au ResultSet

La méthode getMetaData() retourne un objet de la classe ResultSetMetaData qui permet d'obtenir des informations sur le résultat de la requête. Ainsi, le nombre de colonnes peut être obtenu grâce à la méthode getColumnCount() de cet objet.

Exemple :

```
    ResultSetMetaData rsmd;
    rsmd = results.getMetaData();
    nbCols = rsmd.getColumnCount();
```

La méthode next() déplace le curseur sur le prochain enregistrement. Le curseur pointe initialement juste avant le premier enregistrement : il est donc nécessaire de faire un premier appel à la méthode next() pour se placer sur le premier enregistrement.

Des appels successifs à la m&thode next() permettent de parcourir l'ensemble des enregistrements. Elle retourne false lorsqu'il n'y a plus d'enregistrement. Il faut toujours protéger le parcours d'une table dans un bloc try.

Exemple (code Java 1.1) :

```
    //parcours des données retournées

    try {
      ResultSetMetaData rsmd = resultats.getMetaData();
      int nbCols = rsmd.getColumnCount();
      while (resultats.next()) {
        for (int i = 1; i <= nbCols; i++)
          System.out.print(resultats.getString(i) + " ");
          System.out.println();
      }
      resultats.close();
    } catch (SQLException e) {
      //traitement de l'exception
    }
```

Les méthodes getXXX() permettent d'extraire les données selon leur type spécifié par XXX tel que getString(), getDouble(), getInteger(), ... Il existe deux formes de ces méthodes : une pour indiquer le numéro de la colonne en paramètre (en commençant par 1) et une pour indiquer le nom de la colonne en paramètre. La première méthode est plus efficace mais peut générer plus d'erreurs à l'exécution notamment si la structure de la table évolue.

Attention : il est important de noter que ce numéro de colonne fourni en paramètre fait référence au numéro de colonne de l'objet ResultSet (celui correspondant dans l'ordre SELECT) et non au numéro de colonne de la table.

La méthode getString() permet d'obtenir la valeur d'un champ de n'importe quel type sous la forme d'une chaîne de caractères.

60.5.3. Un exemple complet de mise à jour et de sélection sur une table

Exemple (code Java 1.1) :

```
import java.sql.*;

public class TestJDBC1 {

  private static void affiche(String message) {
    System.out.println(message);
  }

  private static void arret(String message) {
    System.err.println(message);
    System.exit(99);
  }

  public static void main(java.lang.String[] args) {
    Connection con = null;
    ResultSet resultats = null;
    String requete = "";

    // chargement du pilote
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    } catch (ClassNotFoundException e) {
      arret("Impossible de charger le pilote jdbc:odbc");
    }

    //connection a la base de données

    affiche("connexion a la base de données");
    try {

      String DBurl = "jdbc:odbc:testDB";
      con = DriverManager.getConnection(DBurl);
    } catch (SQLException e) {
      arret("Connection à la base de données impossible");
    }

    //insertion d'un enregistrement dans la table client
    affiche("Creation enregistrement");

    requete = "INSERT INTO client VALUES (3,'client 3','prenom 3')";
    try {
      Statement stmt = con.createStatement();
      int nbMaj = stmt.executeUpdate(requete);
      affiche("Nb mise a jour = "+nbMaj);
    } catch (SQLException e) {
      e.printStackTrace();
    }

    //creation et execution de la requete
    affiche("Creation et execution de la requête");
    requete = "SELECT * FROM client";

    try {
      Statement stmt = con.createStatement();
      resultats = stmt.executeQuery(requete);
    } catch (SQLException e) {
      arret("Anomalie lors de l'execution de la requête");
    }

    //parcours des données retournées
    affiche("Parcours des données retournées");
    try {
      ResultSetMetaData rsmd = resultats.getMetaData();
      int nbCols = rsmd.getColumnCount();
      boolean encore = resultats.next();

      while (encore) {

        for (int i = 1; i <= nbCols; i++)
          System.out.print(resultats.getString(i) + " ");
        System.out.println();
        encore = resultats.next();
      }

        resultats.close();
    } catch (SQLException e) {
      arret(e.getMessage());
    }
  }
}
```

Résultat :

```
connexion a la base de données
Creation enregistrement
Nb mise a jour = 1
Creation et execution de la requête
Parcours des données retournées
1.0 client 1 prenom 1
2.0 client 2 prenom 2
3.0 client 3 prenom 3
```

60.6. L'obtention d'informations sur la base de données

L'API JDBC propose plusieurs interfaces pour permettre d'obtenir dynamiquement des informations concernant les métadonnées sur la base de données ou sur un ResultSet.

Les objets qui peuvent être utilisés pour obtenir des informations sur la base de données sont :

Classe	Rôle

DatabaseMetaData	Informations à propos de la base de données : nom des tables, index, version, ...
ResultSetMetaData	Informations sur les colonnes (nom et type) d'un ResultSet

60.6.1. L'interface ResultSetMetaData

La méthode getMetaData() d'un objet ResultSet retourne un objet de type ResultSetMetaData. Cet objet permet de connaître le nombre, le nom et le type des colonnes.

Méthode	Rôle
int getColumnCount()	Retourner le nombre de colonnes du ResultSet
String getColumnName(int)	Retourner le nom de la colonne dont le numéro est donné
String getColumnLabel(int)	Retourner le libellé de la colonne donnée
boolean isCurrency(int)	Retourner true si la colonne contient un nombre au format monétaire
boolean isReadOnly(int)	Retourner true si la colonne est en lecture seule
boolean isAutoIncrement(int)	Retourner true si la colonne est auto incrémentée
int getColumnType(int)	Retourner le type de données SQL de la colonne

60.6.2. L'interface DatabaseMetaData

Un objet de la classe DatabaseMetaData permet d'obtenir des informations sur la base de données dans son ensemble : nom des tables, nom des colonnes dans une table, méthodes SQL supportées

Méthode	Rôle
ResultSet getCatalogs()	Retourner la liste du catalogue d'informations (Avec le pont JDBC-ODBC, on obtient la liste des bases de données enregistrées dans ODBC)
ResultSet getTables(catalog, schema, tableNames, columnNames)	Retourner une description de toutes les tables correspondant au TableNames donné et à toutes les colonnes correspondantes à columnNames
ResultSet getColumns(catalog, schema, tableNames, columnNames)	Retourner une description de toutes les colonnes correspondant au TableNames donné et à toutes les colonnes correspondantes à columnNames
String getURL()	Retourner l'URL de la base à laquelle on est connecté
String getDriverName()	Retourner le nom du driver utilisé

La méthode getTables(catalog, schema, tablemask, types[]) de l'objet DataBaseMetaData possède quatre arguments :

- catalog : le nom du catalogue dans lequel les tables doivent être recherchées. Pour une base de données JDBC-ODBC, il peut être mis à null
- schema : le schéma de la base de données à inclure pour les bases les supportant. Il peut être mis à null
- tablemask : un masque décrivant les noms des tables à retrouver. Pour les retrouver toutes, il faut initialiser la chaîne de caractères avec le caractère '%'
- types[] : tableau de chaînes décrivant le type de tables à retrouver. La valeur null permet de retrouver toutes les tables

Exemple (code Java 1.1) :

```
    con = DriverManager.getConnection(url);
    dma =con.getMetaData();
    String[] types = new String[1];
    types[0] = "TABLE"; //set table type mask

    results = dma.getTables(null, null, "%", types);

    while (results.next()) {
      for (i = 1; i <= numCols; i++)
        System.out.print(results.getString(i)+" ");
      System.out.println();
    }
```

60.7. L'utilisation d'un objet de type PreparedStatement

L'interface PreparedStatement définit les méthodes pour un objet qui va encapsuler une requête précompilée à laquelle il est possible de définir des paramètres. Ce type de requête est particulièrement adapté pour une exécution répétée d'une même requête avec des paramètres différents. Cette interface hérite de l'interface Statement.

Lors de l'utilisation d'un objet de type PreparedStatement, la requête est envoyée au moteur de la base de données pour que celui-ci prépare son exécution.

Un objet qui implémente l'interface PreparedStatement est obtenu en utilisant la méthode preparedStatement() d'un objet de type Connection. Cette méthode attend en paramètre une chaîne de caractères contenant la requête SQL. Dans cette chaîne, chaque paramètre est représenté par un caractère «?».

Un ensemble de méthodes setXXX() (où XXX représente un type primitif ou certains types tels que String, Date, Object, ...) permet de fournir les valeurs de chaque paramètre défini dans la requête. Le premier paramètre de ces méthodes précise le numéro du paramètre dont la méthode va fournir la valeur : la valeur du premier paramètre est 1. Le second paramètre précise cette valeur.

La méthode setNull() qui attend en paramètre le numéro du paramètre et le type JDBC du paramètre permet de mettre à NULL un paramètre.

Exemple (code Java 1.1) :

```
package fr.jmdoudoux.dej;

import java.sql.*;

public class TestJDBC2 {

  private static void affiche(String message) {
    System.out.println(message);
  }

  private static void arret(String message) {
    System.err.println(message);
    System.exit(99);
  }

  public static void main(java.lang.String[] args) {
    Connection con = null;
    ResultSet resultats = null;
    String requete = "";

    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    } catch (ClassNotFoundException e) {
      arret("Impossible de charger le pilote jdbc:odbc");
    }

    affiche("connexion a la base de données");
    try {
      String DBurl = "jdbc:odbc:testDB";
      con = DriverManager.getConnection(DBurl);
      PreparedStatement recherchePersonne =
        con.prepareStatement("SELECT * FROM personnes WHERE nom_personne = ?");

      recherchePersonne.setString(1, "nom3");
      resultats = recherchePersonne.executeQuery();

      affiche("parcours des données retournées");

      boolean encore = resultats.next();

      while (encore) {
        System.out.print(resultats.getInt(1) + " :  "+resultats.getString(2)+" "+
          resultats.getString(3)+"("+resultats.getDate(4)+")");
        System.out.println();
        encore = resultats.next();
      }

      resultats.close();
    } catch (SQLException e) {
      arret(e.getMessage());
    }
  }
}
```

Pour exécuter la requête, l'interface PreparedStatement propose deux méthodes :

- executeQuery() : cette méthode permet d'exécuter une requête de type interrogation et renvoie un objet de type ResultSet qui contient les données issues de l'exécution de la requête
- executeUpdate() : cette méthode permet d'exécuter une requête de type mise à jour et renvoie un entier qui contient le nombre d'occurrences impactées par la mise à jour

60.8. L'utilisation des transactions

Une transaction permet de ne valider un ensemble de traitements sur la base de données que s'ils se sont tous effectués correctement.

Par exemple, une opération bancaire de transfert de fond d'un compte vers un autre oblige à la réalisation de l'opération de débit sur un compte et de l'opération de crédit sur l'autre compte. La réalisation d'une seule de ces opérations laisserait les données de la base dans un état inconsistant.

Une transaction est un mécanisme qui permet donc de s'assurer que toutes les opérations qui la compose seront réellement effectuées ou annulées.

Une transaction est gérée à partir de l'objet Connection. Par défaut, une connexion est en mode auto-commit. Dans ce mode, chaque opération est validée unitairement, chacune dans sa propre transaction.

Pour pouvoir rassembler plusieurs traitements dans une transaction, il faut tout d'abord désactiver le mode auto-commit. La classe Connection possède la méthode setAutoCommit() qui attend un booléen qui précise le mode de fonctionnement.

Exemple (code Java 1.1) :

```
    connection.setAutoCommit(false);
```

Une fois le mode auto-commit désactivé, un appel à la méthode commit() de la classe Connection permet de valider la transaction courante. L'appel à cette méthode valide la transaction courante et créé implicitement une nouvelle transaction.

Si une anomalie intervient durant la transaction, il est possible de faire un retour en arrière pour revenir à la situation de la base de données au début de la transaction en appelant la méthode rollback() de la classe Connection.

60.9. Les procédures stockées

L'interface CallableStatement définit les méthodes pour un objet qui va permettre d'appeler une procédure stockée.

Cette interface hérite de l'interface PreparedStatement.

Un objet qui implémente l'interface CallableStatement est obtenu en utilisant la méthode prepareCall() d'un objet de type Connection. Cette méthode attend en paramètre une chaîne de caractères contenant la chaîne d'appel de la procédure stockée.

L'appel d'une procédure étant particulier à chaque base de données supportant une telle fonctionnalité, JDBC propose une syntaxe unifiée qui sera transcrite par le pilote en un appel natif à la base de données. Cette syntaxe peut prendre plusieurs formes :

- {call nom_procedure_stockees} : cette forme la plus simple permet l'appel d'une procédure stockée sans paramètre ni valeur de retour
- {call nom_procedure_stockees(?, ?, ...)} : cette forme permet l'appel d'une procédure stockée avec des paramètres
- {? = call nom_procedure_stockees(?, ?, ...)} : cette forme permet l'appel d'une procédure stockée avec des paramètres et une valeur de retour

Un ensemble de méthode setXXX() (où XXX représente un type primitif ou certains types tels que String, Date, Object, ...) permet de fournir les valeurs de chaque paramètre défini dans la requête. Le premier paramètre de ces méthodes précise le numéro du paramètre dont la méthode va fournir la valeur. Le second paramètre précise cette valeur.

Un ensemble de méthode getXXX() (où XXX représente un type primitif ou certains types tels que String, Date, Object, ...) permet d'obtenir la valeur du paramètre de retour en fournissant la valeur 0 comme index de départ et un autre index pour les paramètres définis en entrée/sortie dans la procédure stockée.

Pour exécuter la requête, l'interface PreparedStatement propose deux méthodes :

- executeQuery() : cette méthode permet d'exécuter une requête de type interrogation et renvoie un objet de type ResultSet qui contient les données issues de l'exécution de la requête
- executeUpdate() : cette méthode permet d'exécuter une requête de type mise à jour et renvoie un entier qui contient le nombre d'occurrences impactées par la mise à jour

60.10. Le traitement des erreurs JDBC

JDBC permet de connaitre les avertissements et les erreurs générées par la base de données lors de l'exécution de requête.

La classe SQLException représente les erreurs émises par la base de données. Elle contient trois attributs qui permettent de préciser l'erreur :

- message : contient une description de l'erreur
- SQLState : code défini par les normes X/Open et SQL99
- errorCode : le code d'erreur du fournisseur du pilote

La classe SQLException possède une méthode getNextException() qui permet d'obtenir les autres exceptions levées durant la requête. La méthode renvoie null une fois la dernière exception renvoyée.

Exemple (code Java 1.1) :

```
package fr.jmdoudoux.dej;

import java.sql.*;

public class TestJDBC3 {

  private static void affiche(String message) {
    System.out.println(message);
  }
  private static void arret(String message) {
    System.err.println(message);
    System.exit(99);
  }

  public static void main(java.lang.String[] args) {
    Connection con = null;
    ResultSet resultats = null;
    String requete = "";

    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    } catch (ClassNotFoundException e) {
      arret("Impossible de charger le pilote jdbc:odbc");
    }

    affiche("Connexion à la base de données");
    try {

      String DBurl = "jdbc:odbc:testDB";
      con = DriverManager.getConnection(DBurl);

      requete = "SELECT * FROM tableinexistante";

      Statement stmt = con.createStatement();
      resultats = stmt.executeQuery(requete);

      affiche("Parcours des données retournées");

      boolean encore = resultats.next();

      while (encore) {
        System.out.print(resultats.getInt(1) + " :  " + resultats.getString(2) +
          " " + resultats.getString(3) + "(" + resultats.getDate(4) + ")");
        System.out.println();
        encore = resultats.next();
      }

      resultats.close();
    } catch (SQLException e) {
      System.out.println("SQLException");
      do {
        System.out.println("SQLState : " + e.getSQLState());
        System.out.println("Description :  " + e.getMessage());
        System.out.println("code erreur :   " + e.getErrorCode());
        System.out.println("");
        e = e.getNextException();
      } while (e != null);
      arret("");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

60.11. Les évolutions de l'API JDBC

60.11.1. JDBC 2.0

La version 2.0 de l'API JDBC a été intégrée au JDK 1.2. Cette nouvelle version apporte plusieurs fonctionnalités dont les principales sont :

- support du parcours dans les deux sens des résultats
- support de la mise à jour des résultats
- possibilité de faire des mises à jour de masse (Batch Updates)
- prise en compte des champs définis par SQL-3 dont BLOB et CLOB

L'API JDBC 2.0 est séparée en deux parties :

- la partie principale (core API) contient les classes et interfaces nécessaires à l'utilisation de bases de données : elles sont regroupées dans le package java.sql
- la seconde partie est une extension qui permet de gérer les transactions distribuées, les pools de connexions, la connexion avec un objet DataSource ... Les classes et interfaces sont regroupées dans le package javax.sql

60.11.1.1. Les fonctionnalités de l'objet ResultSet

Les possibilités de l'objet ResultSet dans la version 1.0 de JDBC sont très limitées : uniquement le parcours séquentiel de chaque occurrence de la table retournée.

La version 2.0 apporte des améliorations à l'objet ResultSet : le parcours des occurrences dans les deux sens et la possibilité de faire des mises à jour sur une occurrence.

Concernant le parcours, il est possible de préciser trois modes de fonctionnement :

- forward-only : parcours séquentiel de chaque occurrence (java.sql.ResultSet.TYPE_FORWARD_ONLY)
- scroll-insensitive : les occurrences ne reflètent pas les mises à jour qui peuvent intervenir durant le parcours (java.sql.ResultSet.TYPE_SCROLL_INSENSITIVE)
- scroll-sensitive : les occurrences reflètent les mises à jour qui peuvent intervenir durant le parcours (java.sql.ResultSet.TYPE_SCROLL_SENSITIVE)

Il est aussi possible de préciser si le ResultSet peut être mise à jour ou non :

- java.sql.ResultSet.CONCUR_READ_ONLY : lecture seule
- java.sql.resultSet.CONCUR_UPDATABLE : mises à jour possibles

C'est à la création d'un objet de type Statement qu'il faut préciser ces deux modes. Si ces deux modes ne sont pas précisés, ce sont les caractéristiques de la version 1.0 de JDBC qui sont utilisées (TYPE_FORWARD_ONLY et CONCUR_READ_ONLY).

Exemple (code jdbc 2.0) :

```
Statement statement = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
  ResultSet.CONCUR_READ_ONLY);
ResultSet resultSet = statement.executeQuery("SELECT nom, prenom FROM employes");
```

Le support de ces fonctionnalités est optionnel pour un pilote. L'objet DatabaseMetadata possède la méthode supportsResultSetType() qui attend en paramètre une constante qui représente une caractéristique : la méthode renvoie un booléen qui indique si la caractéristique est supportée ou non.

A la création du ResultSet, le curseur est positionné avant la première occurrence à traiter. Pour se déplacer dans l'ensemble des occurrences, il y a toujours la méthode next() pour se déplacer sur le suivant mais aussi plusieurs autres méthodes pour permettre le parcours des occurrences en fonctions du mode utilisé dont les principales sont :

Méthode	Rôle
boolean isBeforeFirst()	Renvoyer un booléen qui indique si la position courante du curseur se trouve avant la première ligne
boolean isAfterLast()	Renvoyer un booléen qui indique si la position courante du curseur se trouve après la dernière ligne
boolean isFirst()	Renvoyer un booléen qui indique si le curseur est positionné sur la première ligne
boolean isLast()	Renvoyer un booléen qui indique si le curseur est positionné sur la dernière ligne
boolean first()	Déplacer le curseur sur la première ligne

boolean last()	Déplacer le curseur sur la dernière ligne
boolean absolute(int)	Déplacer le curseur sur la ligne dont le numéro est fourni en paramètre à partir du début s'il est positif et à partir de la fin s'il est négatif. 1 déplace sur la première ligne, -1 sur la dernière, -2 sur l'avant dernière ...
boolean relative(int)	Déplacer le curseur du nombre de lignes fourni en paramètre par rapport à la position courante du curseur. Le paramètre doit être négatif pour se déplacer vers le début et positif pour se déplacer vers la fin. Avant l'appel de cette méthode, il faut obligatoirement que le curseur soit positionné sur une ligne.
boolean previous()	Déplacer le curseur sur la ligne précédente. Le boolen indique si la première occurrence est dépassée.
void afterLast()	Déplacer le curseur après la dernière ligne
void beforeFirst()	Déplacer le curseur avant la première ligne
int getRow()	Renvoyer le numéro de la ligne courante

Exemple (code jdbc 2.0) :

```
    Statement statement = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
      ResultSet.CONCUR_READ_ONLY);
    ResultSet resultSet = statement.executeQuery(
      "SELECT nom, prenom FROM employes ORDER BY nom");
    resultSet.afterLast();
    while (resultSet.previous()) {
      System.out.println(resultSet.getString("nom")+
        " "+resultSet.getString("prenom"));
    }
```

Durant le parcours d'un ResultSet, il est possible d'effectuer des mises à jour sur la ligne courante du curseur. Pour cela, il faut déclarer l'objet ResultSet comme acceptant les mises à jour. Avec les versions précédentes de JDBC, il fallait utiliser la méthode executeUpdate() avec une requête SQL.

Maintenant pour réaliser ces mises à jour, JDBC 2.0 propose de les réaliser via des appels de méthodes plutôt que d'utiliser des requêtes SQL.

Méthode	Rôle
updateXXX(String, XXX)	Permettre de mettre à jour la colonne dont le nom est fourni en paramètre. Le type Java de cette colonne est XXX
updateXXX(int, XXX)	Permettre de mettre à jour la colonne dont l'index est fourni en paramètre. Le type Java de cette colonne est XXX
updateRow()	Permettre d'actualiser les modifications réalisées avec des appels à updateXXX()
boolean rowsUpdated()	Indiquer si la ligne courante a été modifiée
deleteRow()	Supprimer la ligne courante
rowDeleted()	Indiquer si la ligne courante est supprimée
moveToInsertRow()	Permettre de créer une nouvelle ligne dans l'ensemble de résultat
insertRow()	Permettre de valider la création de la ligne

Pour réaliser une mise à jour dans la ligne courante désignée par le curseur, il faut utiliser une des méthodes updateXXX() sur chacun des champs à modifier. Une fois toutes les modifications faites dans une ligne, il faut appeler la méthode updateRow() pour reporter ces modifications dans la base de données car les méthodes updateXXX() ne font des mises à jour que dans le jeu de résultats. Les mises à jour sont perdues si un changement de ligne intervient avant l'appel à la méthode updateRow().

La méthode cancelRowUpdates() permet d'annuler toutes les modifications faites dans la ligne. L'appel à cette méthode doit être effectué avant l'appel à la méthode updateRow().

Pour insérer une nouvelle ligne dans le jeu de résultat, il faut tout d'abord appeler la méthode moveToInsertRow(). Cette méthode déplace le curseur vers un buffer dédié à la création d'une nouvelle ligne. Il faut alimenter chacun des champs nécessaires dans cette nouvelle ligne. Pour valider la création de cette nouvelle ligne, il faut appeler la méthode insertRow().

Pour supprimer la ligne courante, il faut appeler la méthode deleteRow(). Cette méthode agit sur le jeu de résultats et sur la base de données.

60.11.1.2. Les mises à jour de masse (Batch Updates)

JDBC 2.0 permet de réaliser des mises à jour de masse en regroupant plusieurs traitements pour les envoyer en une seule fois au SGBD. Ceci permet d'améliorer les performances surtout si le nombre de traitements est important.

Cette fonctionnalité n'est pas obligatoirement supportée par le pilote. La méthode supportsBatchUpdates() de la classe DatabaseMetaData permet de savoir si elle est utilisable avec le pilote.

Plusieurs méthodes ont été ajoutées à l'interface Statement pour pouvoir utiliser les mises à jour de masse :

Méthode	Rôle
void addBatch(String)	Permettre d'ajouter une chaîne contenant une requête SQL
int[] executeBatch()	Permettre d'exécuter toutes les requêtes. Elle renvoie un tableau d'entiers qui contient pour chaque requête, le nombre de mises à jour effectuées.
void clearBatch()	Supprimer toutes les requêtes stockées

Lors de l'utilisation de batchupdate, il est préférable de positionner l'attribut autocommit à false afin de faciliter la gestion des transactions et le traitement d'une erreur dans l'exécution d'un ou plusieurs traitements.

Exemple (code jdbc 2.0) :

```
    connection.setAutoCommit(false);
    Statement statement = connection.createStatement();

    for(int i=0; i<10 ; i++) {
      statement.addBatch("INSERT INTO personne VALUES('nom"+i+"','prenom"+i+"')");
    }
    statement.executeBatch();
```

Une exception particulière peut être levée en plus de l'exception SQLException lors de l'exécution d'une mise à jour de masse. L'exception SQLException est levée si une requête SQL d'interrogation doit être exécutée (requête de type SELECT). L'exception BatchUpdateException est levée si une des requêtes de mise à jour échoue.

L'exception BatchUpdateException possède une méthode getUpdateCounts() qui renvoie un tableau d'entiers contenant le nombre d'occurrences impactées par chaque requête réussie.

60.11.1.3. Le package javax.sql

Ce package est une extension à l'API JDBC qui propose des fonctionnalités pour les développements d'applications d'entreprise. C'est pour cette raison que cette extension est intégrée à J2EE/Java EE.

Les principales fonctionnalités proposées sont :

- une nouvelle interface pour assurer la connexion : l'interface DataSource

- les pools de connexions
- les transactions distribuées
- l'API Rowset

DataSource et Rowset peuvent être utilisées directement. Les pools de connexions et les transactions distribuées sont utilisés par une implémentation dans les serveurs d'applications pour fournir ces services.

60.11.1.4. L'interface DataSource

A partir de JDBC version 3.0 fournie avec Java 1.4, l'interface javax.sql.DataSource propose de fournir une meilleure alternative à la classe DriverManager pour assurer la création d'instance de connexions à une base de données.

Une implémentation de l'interface DataSource est une fabrique pour créer des connexions vers une source de données. Il existe plusieurs types d'implémentations de l'interface DataSource :

- implémentation basique (basic implementation) : créer des instance de type Connection
- implémentation utilisant un pool de connections (connection pooling implementation) : obtenir des instances préalablement créées et stockées dans un pool
- implémentation pour transactions distribuées (distributed transaction implementation) : obtenir des instances pouvant participer à une transaction distribuée

Les fournisseurs de pilotes doivent proposer au moins une implémentation de l'interface DataSource.

Une fois créé, un objet de type DataSource peut être enregistré dans un service de nommage. Il suffit alors d'utiliser JNDI pour obtenir une instance de classe DataSource.

Exemple :

```
    // ...
    Context ctx = new InitialContext();
    DataSource ds = (DataSource) ctx.lookup("jdbc/applicationDB");
    Connection con = ds.getConnection("admin", "mpadmin");
    // ...
```

Si aucun service de nommage n'est utilisable, il est possible de créer une instance de la classe implémentant DataSource proposée par le fournisseur du pilote JDBC.

Exemple :

```
package fr.jmdoudoux.dej.jdbc;

import java.sql.Connection;
import java.sql.SQLException;

import com.mysql.jdbc.jdbc2.optional.MysqlDataSource;

public class TestDataSource {

  public static void main(String[] args) {
    MysqlDataSource dataSource = new MysqlDataSource();
    dataSource.setUser("root");
    dataSource.setPassword("password");
    dataSource.setServerName("localhost");
    dataSource.setPort(3306);
    dataSource.setDatabaseName("mabdd");

    try {
      Connection connection = dataSource.getConnection();

      // utilisation de la connexion

      connection.close();
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
}
```

60.11.1.5. Les pools de connexions

Un pool de connexions permet de maintenir et réutiliser un ensemble de connexions établies vers une base de données. L'établissement d'une connexion est très coûteux en ressources. L'intérêt du pool de connexions est de limiter le nombre de ces créations et ainsi d'améliorer les performances surtout si le nombre de connexions est important.

La suite de cette section sera développée dans une version future de ce document

60.11.1.6. Les transactions distribuées

Les connexions obtenues à partir d'un objet DataSource peuvent participer à une transaction distribuée.

La suite de cette section sera développée dans une version future de ce document

60.11.1.7. L'API RowSet

L'interface javax.sql.Rowset définit des objets qui permettent de manipuler les données d'une base de données.

Pour utiliser l'interface RowSet, il est nécessaire d'avoir une implémentation : l'implémentation de référence, une implémentation d'un tiers (par exemple le fournisseur du pilote JDBC) ou développée par soi-même.

L'implémentation d'un RowSet peut être de deux types :

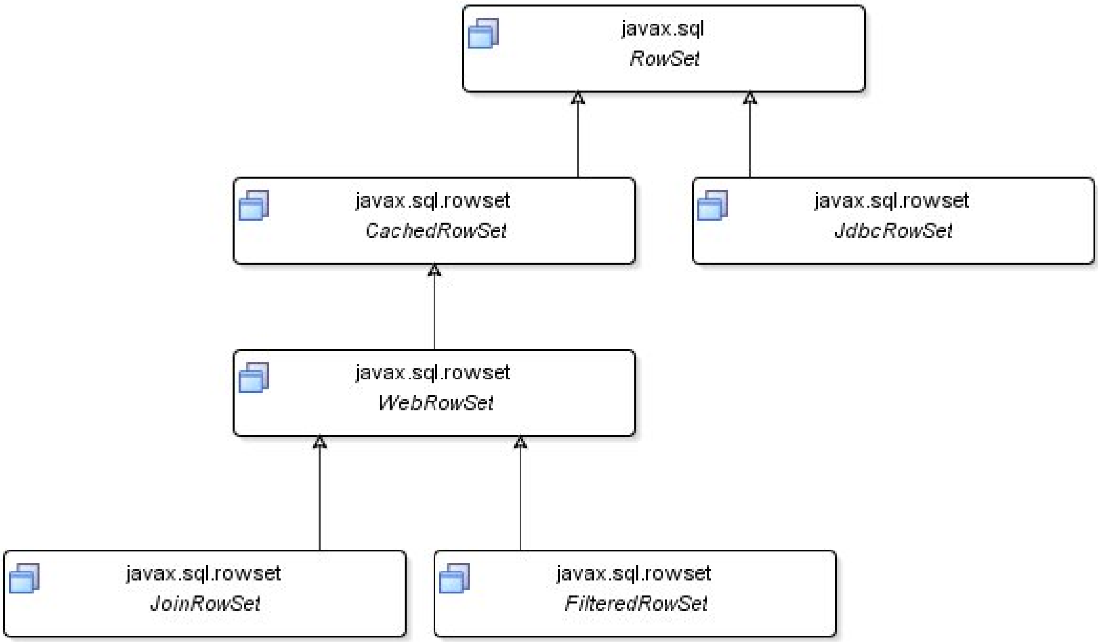
- connectée à la base de données durant toute sa durée de vie
- déconnectée de la base après avoir récupéré des données dans la base pour permettre une manipulation des données en mode déconnecté. Les modifications peuvent alors être reportées dans la base lors d'une reconnexion ultérieure.

Un RowSet de type déconnecté doit posséder un objet de type RowSetReader pour permettre la lecture des données et un objet de type RowSetWriter pour permettre l'enregistrement des données.

Avant Java 5, l'implémentation de référence de Rowset était téléchargeable séparément.

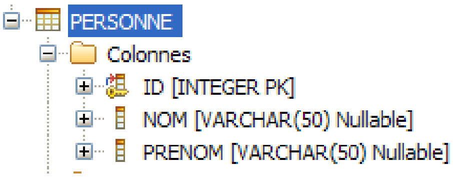
Java 5 fournit en standard une implémentation de référence des interfaces filles de l'interface RowSet définies dans la JSR 114 :

- JDBCRowSet : manipuler les données en mode connecté
- CachedRowSet : manipuler les données d'une source de données en mode déconnecté (les données sont stockées dans l'objet)
- WebRowSet : permet la lecture et l'écriture des données au format XML (hérite de CachedRowSet)
- FilteredRowSet : permet de faire des filtres (hérite de WebRowSet)
- JoinRowSet : permet de faire des jointures (hérite de WebRowSet) avec des objets implémentant l'interface Joinable



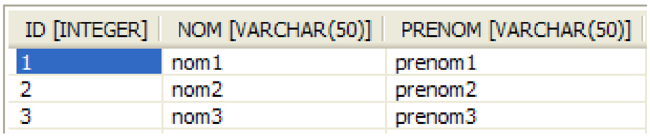
Ces interfaces filles sont définies dans le package javax.sql. Les implémentations sont nommées du nom de l'interface suivi de impl : elles sont regroupées dans le package javax.sql.rowset.

Les exemples de cette section utilisent une base de données JavaDB en mode embeded ou client/server selon les besoins. La table utilisée est composée de 3 champs :



```
PERSONNE
  Colonnes
    ID [INTEGER PK]
    NOM [VARCHAR(50) Nullable]
    PRENOM [VARCHAR(50) Nullable]
```

La table personne contient 3 occurrences



ID [INTEGER]	NOM [VARCHAR(50)]	PRENOM [VARCHAR(50)]
1	nom1	prenom1
2	nom2	prenom2
3	nom3	prenom3

60.11.1.7.1. L'interface RowSet

Un RowSet est un objet qui encapsule les données d'une source de données. L'implémentation d'un RowSet est un Javabean. Un RowSet peut obtenir lui-même ses données en se connectant à la base de données.

L'interface RowSet est définie depuis la version 2.0 de l'API JDBC. Elle hérite de l'interface ResultSet : elle encapsule donc des données tabulaires dont l'utilisation générale est similaire.

L'intérêt des objets de type RowSet est que ce sont des javabeans : ils gèrent donc des propriétés, sont sérialisables et peuvent mettre en oeuvre un mécanisme d'événements. Cela permet la mise en oeuvre de JDBC au travers d'un javabean.

Le fait que les RowSet soient des JavaBeans permet de les sérialiser (pour des échanges à travers le réseau par exemple) ou de les utiliser directement avec d'autres Java Beans (avec les composants Swing dans une interface graphique par exemple).

Les implémentations de l'interface RowSet sont sérialisables ce qui facilite leur utilisation par rapport aux objets de type ResultSet qui ne le sont pas. Ils peuvent par exemple être utilisés par des EJB.

Cette interface propose un ensemble de propriétés pour permettre la connexion à une source de données. La propriété command contient la requête SQL qui permet d'obtenir les données. Ceci permet d'éviter la mise en oeuvre des différents objets de l'API JDBC (Connection et Statment notamment).

La méthode setURL() permet de préciser l'url JDBC utilisée lors de la connexion. Les méthodes setUsername() et setPassword() permettent de fournir le nom du user et son mot de passe pour la connexion.

La méthode setCommand() permet de préciser la requête qui sera exécutée pour obtenir les données.

La méthode execute() permet de réaliser les traitements pour charger les données (connexion à la base de données, exécution de la requête, parcours des données et éventuellement fermeture de la connexion selon l'implémentation du RowSet).

Le parcours des données se fait de la même façon que pour un ResultSet sachant qu'il peut toujours se faire dans les deux sens selon le paramétrage du RowSet (utilisation des méthodes first(), last(), next() et previous()).

Un RowSet peut être rempli de deux façons :

- En lui fournissant les informations de connexion et la requête à exécuter
- En fournissant à la méthode populate() un objet de type ResultSet qui contient déjà les données issues de l'exécution d'une requête

Une fois rempli, le RowSet peut toujours être parcouru dans les deux sens même si le pilote JDBC utilisé pour remplir les données ne permet pas cette fonctionnalité. La méthode size() permet de connaître le nombre d'occurrences contenues dans le RowSet.

Attention : lorsque le RowSet est rempli grâce à un ResultSet, il est nécessaire pour faire des modifications dans la table de la base de données de fournir au Rowset les informations de connexion et même la table concernée en utilisant la méthode setTableName().

Il est possible de préciser le niveau d'isolation de la transaction utilisée avec la connexion.

Exemple :

```
  rs.setTransactionIsolation(
          Connection.TRANSACTION_READ_COMMITTED);
```

Les interfaces des spécifications de RowSet sont contenues dans le package javax.sql.rowset.
L'implémentation fournie avec le JDK est contenue dans le package com.sun.rowset : elle a été spécifiée par la JSR 114.
Elle propose 5 RowSets standards : JdbcRowSet, CachedRowSet, WebRowSet, FilteredRowSet et JoinRowSet

Le JdbcRowSet fonctionne en mode connecté alors que CachedRowSet, WebRowSet, FilteredRowSet et JoinRowSet fonctionnent en mode déconnecté.

60.11.1.7.2. L'interface JdbcRowSet

JdbcRowSet est un Rowset connecté qui encapsule un ResultSet.

Contrairement au ResultSet, JdbcRowSet permet d'encapsuler un ensemble de données et de proposer un parcours des données dans les deux sens même si l'implémentation du ResultSet utilisé pour le remplir ne le permet pas.

JdbcRowSet peut donc être parcouru dans les deux sens et peut être mis à jour.

Java 5 fournit une implémentation de cette interface avec la classe com.sun.rowset.JdbcRowSetImpl

La classe JdbcRowSetImpl possède deux constructeurs :

- sans paramètre
- avec un objet de type ResultSet en paramètre

En utilisant le constructeur sans paramètre, il est nécessaire d'utiliser les méthodes utiles à la configuration de la connexion et de la requête à exécuter.

Exemple (Java 5) :

```
package fr.jmdoudoux.dej.rowset;

import java.sql.ResultSet;

import javax.sql.rowset.JdbcRowSet;

import com.sun.rowset.JdbcRowSetImpl;

public class TestJdbcRowSet {

  public static void main(String[] args) {
    JdbcRowSet rs;

    try {
      Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

      rs = new JdbcRowSetImpl();
      rs.setUrl("jdbc:derby:C:/Program Files/Java/jdk1.6.0/db/MaBaseDeTest");
      rs.setUsername("APP");
      rs.setPassword("");
      rs.setCommand("SELECT * FROM PERSONNE");
      rs.setConcurrency(ResultSet.CONCUR_READ_ONLY);
      rs.execute();

      while (rs.next()) {
        System.out.println("nom : "
            + rs.getString("nom")
            + ", prenom : "
            + rs.getString("prenom"));
      }

      rs.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Il est possible d'utiliser des paramètres dans la requête passée en paramètre de la méthode setCommand(). Chacun des paramètres est défini avec le caractère « ? ». La valeur de chaque paramètre est fournie en utilisant une des méthodes setXXX() qui attend en paramètre l'index du paramètre et sa valeur.

Exemple (Java 5) :

```
package fr.jmdoudoux.dej.rowset;

import java.sql.ResultSet;

import javax.sql.rowset.JdbcRowSet;

import com.sun.rowset.JdbcRowSetImpl;

public class TestJdbcRowSet2 {

  public static void main(String[] args) {
    JdbcRowSet rs;

    try {
      Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

      rs = new JdbcRowSetImpl();
      rs.setUrl("jdbc:derby:C:/Program Files/Java/jdk1.6.0/db/MaBaseDeTest");
      rs.setUsername("APP");
      rs.setPassword("");
      rs.setCommand("SELECT * FROM PERSONNE where id > ?");
      rs.setInt(1, 2);
      rs.setConcurrency(ResultSet.CONCUR_READ_ONLY);
      rs.execute();

      while (rs.next()) {
        System.out.println("nom : "
            + rs.getString("nom")
            + ", prenom : "
            + rs.getString("prenom"));
      }

      rs.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

En utilisant le constructeur attendant en paramètre un objet de type ResultSet, l'instance obtenue encapsule les données du ResultSet. Ces données peuvent être parcourues dans les deux sens et sont modifiables.

Exemple (Java 5) :

```
package fr.jmdoudoux.dej.rowset;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

import javax.sql.rowset.JdbcRowSet;

import com.sun.rowset.JdbcRowSetImpl;

public class TestJdbcRowSet3 {

  public static void main(String[] args) {
    JdbcRowSet rs;

    try {
      Connection conn = null;
      Statement stmt = null;

      Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

      conn = DriverManager.getConnection(
            "jdbc:derby:C:/Program Files/Java/jdk1.6.0/db/MaBaseDeTest;user=APP");
      stmt = conn.createStatement();
      ResultSet resultSet = stmt.executeQuery("select * from personne");

      rs = new JdbcRowSetImpl(resultSet);

      while (rs.next()) {
        System.out.println("nom : "
            + rs.getString("nom")
            + ", prenom : "
            + rs.getString("prenom"));
      }

      rs.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Les données encapsulées dans le RowSet peuvent être mises à jour en fournissant la valeur ResultSet.CONCUR_UPDATABLE à la méthode setConcurrency(). Des méthodes updateXXX() héritées de la classe ResultSet permettent de mettre à jour une donnée en fonction de son type.

La méthode updateRow() permet de demander la mise à jour des données dans le RowSet.

La méthode commit() permet de demander la répercussion des modifications dans la base de données.

Exemple (Java 5) :

```
package fr.jmdoudoux.dej.rowset;

import java.sql.ResultSet;

import javax.sql.rowset.JdbcRowSet;

import com.sun.rowset.JdbcRowSetImpl;

public class TestJdbcRowSet4 {

  public static void main(String[] args) {
    JdbcRowSet rs;

    try {
      Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

      rs = new JdbcRowSetImpl();
      rs.setUrl("jdbc:derby:C:/Program Files/Java/jdk1.6.0/db/MaBaseDeTest");
      rs.setUsername("APP");
      rs.setPassword("");
      rs.setCommand("SELECT * FROM PERSONNE");
      rs.setConcurrency(ResultSet.CONCUR_UPDATABLE);
      rs.execute();

      rs.absolute(2);
      rs.updateString("nom", "nom2 modifie");
      rs.updateRow();
      rs.commit();

      rs.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

60.11.1.7.3. L'interface CachedRowSet

L'interface CachedRowSet définit un RowSet déconnecté : la connexion à la base de données n'est maintenue que pour récupérer toutes les données. Toutes ces données sont stockées dans l'objet et la connexion est fermée. Il est alors possible de manipuler ces données (consultation et mise à jour). Les modifications peuvent alors être rendues persistantes en utilisant une nouvelle connexion dédiée à cette tâche.

Ceci peut permettre de réduire les ressources réseaux et serveurs mais introduit généralement des problématiques de synchronisation des mises à jour.

L'implémentation standard de l'interface CachedRowSet est proposée par la classe com.sun.rowset.CachedRowSetImpl. Cet objet maintient l'état des données qu'il encapsule en mémoire. Il a simplement besoin de la connexion pour remplir les données et plus tard au moment de rendre les modifications sur ces données persistantes.

Exemple (Java 5) :

```
package fr.jmdoudoux.dej.rowset;

import java.sql.ResultSet;

import javax.sql.rowset.CachedRowSet;

import com.sun.rowset.CachedRowSetImpl;

public class TestCachedRowSet {

  public static void main(String[] args) {
    CachedRowSet rs;

    try {
      Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

      rs = new CachedRowSetImpl();
      rs.setUrl("jdbc:derby:C:/Program Files/Java/jdk1.6.0/db/MaBaseDeTest");
      rs.setCommand("SELECT * FROM PERSONNE");
      rs.setUsername("APP");
      rs.setPassword("");
      rs.setConcurrency(ResultSet.CONCUR_READ_ONLY);
      rs.execute();

      while (rs.next()) {
        System.out.println("nom : " + rs.getString("nom"));
      }

      rs.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

La méthode populate() permet de remplir le rowSet avec les données d'un ResultSet.

Ce premier exemple n'est pas pertinent car il aurait été plus efficace d'utiliser directement le ResultSet. Par contre, le CachedRowSet devient intéressant dès qu'il faut faire des mises à jour sans être connecté à la base de données

Les mises à jour sont faites uniquement dans l'objet CachedRowSet. Pour reporter ces modifications dans la base de données, il faut utiliser la méthode acceptChanges(). Lors de l'appel à cette méthode, l'objet CachedRowSet se reconnecte à la base de données et effectue les mises à jour.

Exemple (Java 5) :

```
package fr.jmdoudoux.dej.rowset;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

import javax.sql.rowset.CachedRowSet;

import com.sun.rowset.CachedRowSetImpl;

public class TestCachedRowSet3 {

  public static void main(String[] args) {
    CachedRowSet rs;

    try {

      Connection conn = null;
      Statement stmt = null;

      Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

      conn = DriverManager.getConnection(
            "jdbc:derby:C:/Program Files/Java/jdk1.6.0/db/MaBaseDeTest;user=APP");
      stmt = conn.createStatement();
      ResultSet resultSet = stmt.executeQuery"select * from personne");

      rs = new CachedRowSetImpl();
      rs.populate(resultSet);

      rs.absolute(2);
      rs.updateString("nom", "nom2");
      rs.updateRow();

      rs.acceptChanges(conn);
      rs.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

La propriété COMMIT_ON_ACCEPT_CHANGES est un booléen qui permet de préciser si un commit est réalisé automatiquement à la fin de la méthode acceptChanges(). La valeur par défaut est true. Si sa valeur est false, il faut explicitement faire appel à la méthode commit() pour valider la transaction.

Il est tout à fait possible que les données dans la base soient modifiées entre la récupération des données et leur mise à jour dans la base de données. Avant chaque mise à jour, CachedRowSet vérifie les données courantes dans la base avec leur valeur initiale lors du remplissage des données. Si une différence est détectée alors une exception de type SyncProviderException est levée.

Exemple (Java 5) :

```
package fr.jmdoudoux.dej.rowset;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

import javax.sql.rowset.CachedRowSet;

import com.sun.rowset.CachedRowSetImpl;

public class TestCachedRowSet3 {

  public static void main(String[] args) {
    CachedRowSet rs;

    try {

      Connection conn = null;
      Statement stmt = null;

      Class.forName("org.apache.derby.jdbc.ClientDriver");

      java.util.Properties props = new java.util.Properties();
      props.put("user","APP");
      props.put("password","APP");
      conn = DriverManager.getConnection("jdbc:derby://localhost:1527/MaBaseDeTest", props);

      stmt = conn.createStatement();
      ResultSet resultSet = stmt.executeQuery("select * from personne");

      rs = new CachedRowSetImpl();
      rs.populate(resultSet);

      System.out.println("debut attente");
      Thread.sleep(60000);
      // mise à jour de l'occurrence dans la base de données par un outil externe
      System.out.println("fin attente");

      rs.absolute(2);
      rs.updateString("nom", "nom2");
      rs.updateRow();

      rs.acceptChanges(conn);

      rs.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
debut attente
fin attente
javax.sql.rowset.spi.SyncProviderException: 3 conflicts while synchronizing
        at com.sun.rowset.internal.CachedRowSetWriter.writeData(CachedRowSetWriter.java:373)
        at com.sun.rowset.CachedRowSetImpl.acceptChanges(CachedRowSetImpl.java:862)
        at com.sun.rowset.CachedRowSetImpl.acceptChanges(CachedRowSetImpl.java:921)
        at fr.jmdoudoux.dej.rowset.TestCachedRowSet3.main(TestCachedRowSet3.java:43)
```

Le CachedRowSet propose un mécanisme pour gérer ce cas de figure. Ce mécanisme impose de préciser au CachedRowSet la ou les colonnes qui représentent la clé ceci afin de lui permettre de faire correspondre ces occurrences avec celles de la base de données : c'est la méthode setKeyColumns() qui attend en paramètre un tableau entier contenant les index des colonnes.

Remarque : l'index des colonnes utilisées dans un CachedRowSet commence à 1 à non à 0.

Le traitement des conflits est à faire dans le traitement de l'exception de type SyncProviderException. Cette exception propose la méthode getSyncResolver() qui renvoie un objet de type SyncResolver.

L'objet de type SyncResolver permet d'obtenir les conflits détectés et de les résoudre en fonction des besoins. L'interface SyncResolver définit plusieurs méthodes :

Méthode	Rôle
Object getConflictValue()	Retourne la valeur dans la base de données de l'occurrence courante du SyncResolver pour la colonne fournie en paramètre (index ou nom selon la surcharge utilisée). La valeur retournée est null pour une colonne qui n'est pas en conflit.
int getStatus()	Renvoie un entier qui précise le type d'opération tentée sur la base de données : DELETE_ROW_CONFLICT, INSERT_ROW_CONFLICT, UPDATE_ROW_CONFLICT ou NO_ROW_CONFLICT
boolean nextConflict()	Se déplace sur le prochain conflit s'il existe et renvoie true si le déplacement a eu lieu
boolean previousConflict()	Se déplace sur le conflit précédent s'il existe et renvoie true si le déplacement a eu lieu
void setResolvedValue()	Permet de définir la valeur dans la base de données de l'occurrence courante du SyncResolver pour la colonne fournie en paramètre (index ou nom selon la surcharge utilisée)

Chaque fournisseur propose sa propre implémentation de SyncProvider. Les exemples de cette section utilisent l'implémentation de référence fournie avec le JDK à partir de la version 5.0. Cette implémentation propose un mode de gestion optimiste des accès concurrents (aucun verrou n'est posé sur les occurrences dans la base de données).

Il faut réaliser une itération sur les conflits en utilisant la méthode nextConflict().

La méthode getStatus() permet de connaître le type de mise jour tentée sur la base de données

La méthode getRow() héritée de l'interface ResultSet permet de connaître l'index de l'occurrence concernée par le conflit. Ceci permet de se déplacer dans le RowSet pour obtenir les nouvelles valeurs.

La méthode getConflictValue() est utilisée dans une itération sur les colonnes pour déterminer celles qui sont en conflit : dans ce cas la valeur retournée est différente de null.

A partir de la nouvelle valeur, de la valeur courante dans la base de données et du type de mises à jour, les traitements doivent déterminer la valeur à mettre dans la base de données.

Cette valeur est fournie en utilisant la méthode setResolvedValue().

Exemple (Java 5) :

```
package fr.jmdoudoux.dej.rowset;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import javax.sql.rowset.CachedRowSet;

import com.sun.rowset.CachedRowSetImpl;
import javax.sql.rowset.spi.SyncProviderException;
import javax.sql.rowset.spi.SyncResolver;

public class TestCachedRowSet4 {

  public static void main(String[] args) {
    CachedRowSet rs=null;

    try {

      Connection conn = null;
      Statement stmt = null;

      Class.forName("org.apache.derby.jdbc.ClientDriver");

      java.util.Properties props = new java.util.Properties();
      props.put("user", "APP");
      props.put("password", "APP");
      conn = DriverManager.getConnection(
          "jdbc:derby://localhost:1527/MaBaseDeTest", props);

      stmt = conn.createStatement();
      ResultSet resultSet = stmt.executeQuery("select * from personne");

      rs = new CachedRowSetImpl();
      rs.populate(resultSet);
      rs.setTableName("PERSONNE");
      // la première colonne compose la clé
      rs.setKeyColumns(new int[] { 1 });

      System.out.println("debut attente");
      Thread.sleep(60000);
      // mise à jour de l'occurrence dans la
          // base de données par un outil externe
      System.out.println("fin attente");

      rs.absolute(2);
      rs.updateString("nom", "nom2");
      rs.updateRow();

      rs.acceptChanges(conn);

      rs.close();
    } catch (SyncProviderException spe) {
      SyncResolver resolver = spe.getSyncResolver();

      try {
        while (resolver.nextConflict()) {
          if (resolver.getStatus() == SyncResolver.UPDATE_ROW_CONFLICT) {
            int row = resolver.getRow();
            rs.absolute(row);
            int nbColonne = rs.getMetaData().getColumnCount();
            for (int i = 1; i <= nbColonne; i++) {
              if (resolver.getConflictValue(i) != null) {
                Object valeur = rs.getObject(i);
                Object valeurResolver = resolver.getConflictValue(i);
                System.out.println("champ = "
                                  + rs.getMetaData().getColumnName(i)
                                  +" , Valeur = "+valeur+"
                                  , valeur dans la base="+valeurResolver);
                // Determiner la valeur à mettre dans la base
        // dans ce cas simplement la nouvelle valeur
        resolver.setResolvedValue(i,  valeur);
       }
      }
     }
    }
   } catch (SQLException e) {
    e.printStackTrace();
   }
  } catch (Exception e) {
   e.printStackTrace();
  }
 }
}
```

Résultat :

```
debut attente
fin attente
champ = NOM , Valeur = nom2 , valeur dans la base=nom2 mod
```

L'interface CachedRowSet propose plusieurs méthodes pour annuler des mises à jour faites dans les données encapsulées (avant l'appel à la méthode acceptChanges()) :

Méthode	Rôle
void undoDelete()	Annule l'opération de suppression de l'occurrence courante
void undoInsert()	Annule l'opération d'insertion de l'occurrence courante
void undoUpdate()	Annule l'opération de modification de l'occurrence
void restoreOriginal()	Remettre l'ensemble des données à leur valeur originale (toutes les modifications sont perdues) et remet le curseur avant la première occurrence

La méthode getOriginal() renvoie un ResultSet qui contient toutes les valeurs originales des données du RowSet.

Le stockage des données en mémoire rend le CachedRowSet inapproprié à une utilisation avec de gros volume de données. Dans ce cas, le CachedRowSet peut travailler en paginant sur des portions de données : l'ensemble des données est traité par page (une page contenant un certain nombre d'occurrences). La méthode setPageSize() permet de préciser le nombre maximum d'occurrences dans une page. La méthode nextPage() permet d'obtenir la page suivante. Ce mécanisme est particulièrement utile pour traiter de grosses quantités de données.

La méthode release() permet de supprimer toutes les données contenues dans le RowSet : attention son appel fait perdre toutes les modifications dans les données qui n'ont pas été reportées dans la base de données .

60.11.1.7.4. L'interface WebRowSet

WebRowSet possède la capacité de lire ou d'écrire le contenu du RowSet au format XML. Cette faculté lui permet d'être utilisé pour échanger des données non pas sous une forme sérialisée mais sous la forme d'un document XML (par exemple dans une requête HTTP ou SOAP).

Dans l'implémentation standard, le document XML respecte le schéma :
http://java.sun.com/xml/ns/jdbc/webrowset.xsd

Le contenu au format XML d'un WebRowSet peut être exporté dans un flux quelconque : par exemple, l'envoi du contenu XML d'un WebRowSet dans une réponse d'une servlet.

Le document XML issu d'un WebRowSet possède un noeud racine <webRowSet> qui possède trois noeuds fils :

- <properties> : contient les propriétés du WebRowSet notamment les paramètres de connexion sauf le user et mot de passe

- <metadata> : contient les méta-données du WebRowSet (configuration de chaque colonne)
- <data> : contient les données du WebRowSet

Chaque occurrence de données est stockée dans un tag <currentRow>. La valeur de chaque colonne est stockée dans un tag <columnValue>.

Les occurrences ajoutées sont stockées dans un tag <insertRow>.
Les occurrences modifiées sont stockées dans un tag <updateRow>. La valeur de chaque colonne modifiée est stockée dans un tag fils <updateValue>
Les occurrences supprimées sont stockées dans un tag <deleteRow>.

Exemple (Java 5) :

```
package fr.jmdoudoux.dej.rowset;

import java.sql.ResultSet;

import javax.sql.rowset.WebRowSet;
import com.sun.rowset.WebRowSetImpl;


public class TestWebRowSet {

  public static void main(String[] args) {
    WebRowSet rs;

    try {
      Class.forName("org.apache.derby.jdbc.ClientDriver");

      rs = new WebRowSetImpl();
      rs.setUrl("jdbc:derby://localhost:1527/MaBaseDeTest");
      rs.setCommand("SELECT * FROM PERSONNE");
      rs.setUsername("APP");
      rs.setPassword("APP");
      rs.setConcurrency(ResultSet.CONCUR_READ_ONLY);
      rs.execute();
      rs.writeXml(System.out);

      rs.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
<?xml  version="1.0"?>
<webRowSet
      xmlns="http://java.sun.com/xml/ns/jdbc"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/jdbc
          http://java.sun.com/xml/ns/jdbc/webrowset.xsd">
  <properties>
    <command>SELECT * FROM  PERSONNE</command>
    <concurrency>1007</concurrency>
     <datasource><null/></datasource>
     <escape-processing>true</escape-processing>
     <fetch-direction>1000</fetch-direction>
    <fetch-size>0</fetch-size>
     <isolation-level>2</isolation-level>
    <key-columns>
    </key-columns>
    <map>
    </map>
     <max-field-size>0</max-field-size>
    <max-rows>0</max-rows>
     <query-timeout>0</query-timeout>
    <read-only>true</read-only>
     <rowset-type>ResultSet.TYPE_SCROLL_INSENSITIVE</rowset-type>
    <show-deleted>false</show-deleted>
```

```
   <table-name>PERSONNE</table-name>
   <url>jdbc:derby://localhost:1527/MaBaseDeTest</url>
  <sync-provider>
     <sync-provider-name>com.sun.rowset.providers.RIOptimisticProvider</sync-provider-name>
    <sync-provider-vendor>Sun  Microsystems Inc.</sync-provider-vendor>
     <sync-provider-version>1.0</sync-provider-version>
     <sync-provider-grade>2</sync-provider-grade>
     <data-source-lock>1</data-source-lock>
  </sync-provider>
</properties>
<metadata>
  <column-count>3</column-count>
  <column-definition>
     <column-index>1</column-index>
     <auto-increment>false</auto-increment>
     <case-sensitive>false</case-sensitive>
    <currency>false</currency>
    <nullable>0</nullable>
    <signed>true</signed>
    <searchable>true</searchable>
     <column-display-size>11</column-display-size>
     <column-label>ID</column-label>
    <column-name>ID</column-name>
     <schema-name>APP</schema-name>
     <column-precision>10</column-precision>
     <column-scale>0</column-scale>
     <table-name>PERSONNE</table-name>
    <catalog-name></catalog-name>
    <column-type>4</column-type>
     <column-type-name>INTEGER</column-type-name>
  </column-definition>
  <column-definition>
    <column-index>2</column-index>
     <auto-increment>false</auto-increment>
     <case-sensitive>true</case-sensitive>
    <currency>false</currency>
    <nullable>1</nullable>
    <signed>false</signed>
    <searchable>true</searchable>
    <column-display-size>50</column-display-size>
     <column-label>NOM</column-label>
     <column-name>NOM</column-name>
     <schema-name>APP</schema-name>
     <column-precision>50</column-precision>
     <column-scale>0</column-scale>
    <table-name>PERSONNE</table-name>
    <catalog-name></catalog-name>
    <column-type>12</column-type>
     <column-type-name>VARCHAR</column-type-name>
  </column-definition>
  <column-definition>
     <column-index>3</column-index>
    <auto-increment>false</auto-increment>
     <case-sensitive>true</case-sensitive>
    <currency>false</currency>
    <nullable>1</nullable>
    <signed>false</signed>
    <searchable>true</searchable>
     <column-display-size>50</column-display-size>
    <column-label>PRENOM</column-label>
     <column-name>PRENOM</column-name>
     <schema-name>APP</schema-name>
     <column-precision>50</column-precision>
     <column-scale>0</column-scale>
     <table-name>PERSONNE</table-name>
    <catalog-name></catalog-name>
    <column-type>12</column-type>
     <column-type-name>VARCHAR</column-type-name>
  </column-definition>
</metadata>
<data>
  <currentRow>
    <columnValue>1</columnValue>
     <columnValue>nom1</columnValue>
```

```
      <columnValue>prenom1</columnValue>
    </currentRow>
    <currentRow>
      <columnValue>2</columnValue>
       <columnValue>nom2</columnValue>
       <columnValue>prenom2</columnValue>
    </currentRow>
    <currentRow>
      <columnValue>3</columnValue>
      <columnValue>nom3</columnValue>
       <columnValue>prenom3</columnValue>
    </currentRow>
  </data>
</webRowSet>
```

La méthode readXml() permet de remplir l'objet WebRowSet avec un fichier XML par exemple précédemment créé grâce à la méthode writeXml().

60.11.1.7.5. L'interface FilteredRowSet

L'interface FilteredRowSet qui hérite de l'interface WebRowSet permet de mettre en oeuvre un filtre par programmation sans utiliser SQL.

FilteredRowSet est particulièrement utile car il permet de filtrer un ensemble de données sans avoir à effectuer une requête sur la base de données avec le filtre.

Le filtre est encapsulé dans une classe qui implémente l'interface Predicate. Dans cette classe, il faut redéfinir les méthodes evaluate() qui renvoie un booléen précisant si l'occurrence est conservée ou non par le filtre.

La méthode evaluate() acceptant en paramètre un objet de type RowSet est utilisée par l'objet FilteredRowSet lors du parcours de ses occurrences.

Les surcharges de la méthode evaluate() acceptant un objet et une colonne (par index ou par nom) sont utilisées par l'objet FilteredRowSet pour déterminer si une valeur d'une colonne correspond au filtre.

Exemple (Java 5) : ne conserver que les personnes dont le nom se termine par 2

```
package fr.jmdoudoux.dej.rowset;

import java.sql.SQLException;

import javax.sql.RowSet;
import javax.sql.rowset.Predicate;

public class PersonnePredicate implements Predicate {

  public boolean evaluate(Object value, int column) throws SQLException {
    // inutilisé dans cet exemple
    return false;
  }

  public boolean evaluate(Object value, String columnName) throws SQLException {
    // inutilisé dans cet exemple
    return false;
  }

  public boolean evaluate(RowSet rowset) {
    try {
      String nom = rowset.getString("nom");
      if (nom.endsWith("2")) {
        return true;
      } else {
        return false;
      }
    } catch (SQLException sqle) {
      return false;
    }
  }
}
```

Le filtre est précisé au FilteredRowSet en utilisant la méthode setFilter() qui attend en paramètre une instance de la classe Predicate.

Exemple (Java 5) :

```
package fr.jmdoudoux.dej.rowset;

import java.sql.ResultSet;

import javax.sql.rowset.FilteredRowSet;

import com.sun.rowset.FilteredRowSetImpl;

public class TestFilteredRowSet {

  public static void main(String[] args) {
    FilteredRowSet rs;

    try {
      Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

      rs = new FilteredRowSetImpl();
      rs.setUrl("jdbc:derby:C:/Program Files/Java/jdk1.6.0/db/MaBaseDeTest");
      rs.setCommand("SELECT * FROM PERSONNE");
      rs.setUsername("APP");
      rs.setPassword("");
      rs.setConcurrency(ResultSet.CONCUR_READ_ONLY);
      rs.setFilter(new PersonnePredicate());
      rs.execute();

      while (rs.next()) {
        System.out.println("nom : " + rs.getString("nom"));
      }

      rs.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
nom : nom2
```

60.11.1.7.6. L'interface JoinRowSet

L'interface JoinRowSet qui hérite de l'interface WebRowSet permet de faire des jointures entre plusieurs instances de l'interface Joinable. Les interfaces qui héritent de Joinable sont : CachedRowSet, FilteredRowSet, JdbcRowSet, JoinRowSet, WebRowSet.

JoinRowSet peut être particulièrement utile si les données des RowSet qu'il encapsule appartiennent à des sources de données différentes

Pour utiliser un JoinRowSet, il faut en créer une instance et utiliser la méthode addRowSet() pour ajouter les instances de l'interface Joinable à utiliser dans la jointure. La méthode adddRowSet() possède plusieurs surcharges qui permettent de préciser l'instance de Joinable et la ou les clés utilisées lors de la jointure.

Exemple (Java 5) :

```
package fr.jmdoudoux.dej.rowset;
```

```
import java.sql.ResultSet;

import javax.sql.rowset.CachedRowSet;
import javax.sql.rowset.JoinRowSet;

import com.sun.rowset.CachedRowSetImpl;
import com.sun.rowset.JoinRowSetImpl;

public class TestJoinRowSetRowSet {

  public static void main(String[] args) {
    CachedRowSet rs1;
    CachedRowSet rs2;
    JoinRowSet rs;

    try {
      Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

      rs1 = new CachedRowSetImpl();
      rs1.setUrl("jdbc:derby:C:/Program Files/Java/jdk1.6.0/db/MaBaseDeTest");
      rs1.setCommand("SELECT * FROM PERSONNE");
      rs1.setUsername("APP");
      rs1.setPassword("");
      rs1.setConcurrency(ResultSet.CONCUR_READ_ONLY);
      rs1.execute();

      rs2 = new CachedRowSetImpl();
      rs2.setUrl("jdbc:derby:C:/Program Files/Java/jdk1.6.0/db/MaBaseDeTest");
      rs2.setCommand("SELECT * FROM ADRESSE");
      rs2.setUsername("APP");
      rs2.setPassword("");
      rs2.setConcurrency(ResultSet.CONCUR_READ_ONLY);
      rs2.execute();

      rs = new JoinRowSetImpl();
      rs.addRowSet(rs1,1);
      rs.addRowSet(rs2,1);

      while (rs.next()) {
        System.out.println("nom : " + rs.getString("nom")+", rue : " + rs.getString("rue"));
      }

      rs.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
nom : nom3, rue : rue3
nom : nom2, rue : rue2
nom : nom1, rue : rue1
```

La méthode setJoinType() permet de préciser le type de jointure à effectuer en utilisant les constantes définies dans l'interface JoinRowSet : CROSS_JOIN, FULL_JOIN, INNER_JOIN (par défaut), LEFT_OUTER_JOIN et RIGHT_OUTER_JOIN. Les implémentations n'ont pas d'obligation à supporter tous les types de jointures : l'utilisation d'un type de jointure non supporté par l'implémentation lève une exception de type SQLException.

60.11.1.7.7. L'utilisation des événements

L'interface RowSetListener permet de gérer certains événements d'un RowSet. Le modèle d'événement des Javabeans est mis en oeuvre au travers de ce listener de type RowSetListener et d'un événement de type RowSetEvent.

Les méthodes addRowSetListener() et removeRowSetListener() de l'interface RowSet permettent respectivement d'enregistrer et de supprimer un listener

L'interface RowSetListener définit trois méthodes :

- cursorMoved() : appelée lorsque le curseur de parcours des données change
- rowChanged() : appelée lorsqu'une donnée est modifiée
- rowSetChanged() : appelée lorsque l'ensemble des données est modifié

Exemple (Java 5) :

```
package fr.jmdoudoux.dej.rowset;

import java.sql.ResultSet;

import javax.sql.RowSetEvent;
import javax.sql.RowSetListener;
import javax.sql.rowset.JdbcRowSet;

import com.sun.rowset.JdbcRowSetImpl;

public class TestRowSetListener {

  public static void main(String[] args) {
    JdbcRowSet rs;

    try {
      Class.forName("org.apache.derby.jdbc.ClientDriver");

      rs = new JdbcRowSetImpl();
      rs.setUrl("jdbc:derby://localhost:1527/MaBaseDeTest");
      rs.setCommand("SELECT * FROM PERSONNE");
      rs.setUsername("APP");
      rs.setPassword("APP");
      rs.setConcurrency(ResultSet.CONCUR_READ_ONLY);
      rs.addRowSetListener(new RowSetListener() {
        public void cursorMoved(RowSetEvent event) {
          System.out.println("L'evenement cursorMoved est emis");
        }

        public void rowChanged(RowSetEvent event) {
          System.out.println("L'evenement rowChanged est emi"");
        }

        public void rowSetChanged(RowSetEvent event) {
          System.out.println("L'evenement rowSetChanged est emis");
        }

      });
      rs.execute();

      while (rs.next())
        System.out.println("nom : " + rs.getString("nom"));

      rs.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public TestRowSetListener() {
    JdbcRowSet rs;

    try {
      Class.forName("oracle.jdbc.driver.OracleDriver");

      rs = new JdbcRowSetImpl();
      rs.setUrl("jdbc:oracle:thin:@localhost:1521:test");
      rs.setCommand("SELECT * FROM article");
      rs.setUsername("test");
      rs.setPassword("test");
      rs.setConcurrency(ResultSet.CONCUR_READ_ONLY);
      rs.addRowSetListener(new RowSetListener() {
        public void cursorMoved(RowSetEvent event) {
          System.out.println("L'evenement cursorMoved est emis");
        }
        public void rowChanged(RowSetEvent event) {
          System.out.println("L'evenement rowChanged est emis");
        }

        public void rowSetChanged(RowSetEvent event) {
          System.out.println("L'evenement rowSetChanged est emis");
        }

      });
      rs.execute();

      while (rs.next())
        System.out.println("libelle : " + rs.getString("libelle"));

      rs.close();

    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
L'evenement rowSetChanged est emis
L'evenement cursorMoved est emis
nom : nom1
L'evenement cursorMoved est emis
nom : nom2
L'evenement cursorMoved est emis
nom : nom3
L'evenement cursorMoved est emis
```

60.11.2. JDBC 3.0

Les spécifications de l'API JDBC version 3.0, disponible depuis mai 2002, sont issues des travaux de la JSR 54 et sont directement intégrées dans la plate-forme J2SE 1.4.

Ces spécifications ont été développées en tenant compte de plusieurs points : conserver une compatibilité avec la version précédente de l'API, assurer une meilleure interaction avec la technologie JCA, le support de SQL 99, ...

Cette version propose plusieurs améliorations dont les savepoints, le support de SQL 99, la récupération des identifiants générés détaillées dans les sections suivantes.

JDBC n'est qu'une spécification : l'implémentation réalisée au travers des pilotes peut proposer tout ou uniquement une partie de ces fonctionnalités.

60.11.2.1. Le nommage des paramètres d'un objet de type CallableStatement

Avant la version 3.0, lors de l'utilisation d'une instance de l'interface CallableStatement, pour assigner une valeur à un paramètre, il fallait obligatoirement utiliser son index. Il est dorénavant possible d'utiliser un nom pour un paramètre et d'utiliser ce nom pour mettre à jour sa valeur.

L'interface CallableStatement s'est vu rajouter des surcharges des méthodes getXXX() et setXXX() attendant en premier paramètre une chaîne de caractères qui va contenir le nom du paramètre.

Cette fonctionnalité est intéressante notamment pour l'appel de procédures stockées qui possèdent des valeurs par défaut pour certains paramètres. Il est ainsi possible de ne fournir que les valeurs voulues lors de l'appel.

60.11.2.2. Les types java.sql.Types.DATALINK et java.sql.Types.BOOLEAN

Deux nouveaux types sont supportés : java.sql.Types.DATALINK pour des url vers des ressources externes et java.sql.Types.BOOLEAN pour les booléens. Les valeurs d'une donnée de ces types sont obtenues en utilisant respectivement les méthodes getURL() et getBoolean() de la classe ResultSet.

60.11.2.3. L'obtention des valeurs générées automatiquement lors d'une insertion

La plupart des bases de données relationnelles proposent des fonctionnalités pour permettre la génération d'une valeur, généralement auto incrémentée dans un champ d'une base de données, permettant la génération d'un identifiant unique. Ceci est très pratique pour définir un champ qui sera la clé primaire d'une table. Cependant avant la version 3.0 de JDBC, il était nécessaire d'effectuer une lecture après l'insertion des données.

Ceci pose souvent des problèmes notamment pour arriver à utiliser une clause where dans la requête d'interrogation qui soit sûre de renvoyer les données de la ligne insérée. De plus, cela impose de réaliser une opération supplémentaire sur la base de données.

Il est maintenant possible d'obtenir facilement la valeur d'un identifiant générée&nbsp par la base de données lors de l'insertion d'une nouvelle occurrence dans une table. Attention, le support de cette fonctionnalité par le pilote est optionnel.

Il suffit de préciser lors de l'appel à la méthode executeUpdate() de l'interface Statement la valeur Statement.RETURN_GENERATED_KEYS au paramètre autoGeneratedKeys de type int.

Pour obtenir la valeur de la clé ou des clés générées, il suffit d'appeler la méthode getGeneratedKeys() de l'instance de l'interface Statement utilisée pour exécuter la mise à jour : le ResultSet retourné par cette méthode contient un champ pour chaque champ généré par la base de données.

Exemple :

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;

public class TestJdbc101 {

  public static void main(java.lang.String[] args) {
    Connection con = null;
    Statement stmt = null;
    ResultSet resultats = null;
    String requete = "";

    // chargement du pilote
    try {
      Class.forName("com.mysql.jdbc.Driver").newInstance();
    } catch (Exception e) {
      e.printStackTrace();
      System.exit(99);
    }

    try {
      String DBurl = "jdbc:mysql://localhost:3306/testjava";
      con = DriverManager.getConnection(DBurl);
      stmt = con.createStatement();

      stmt.executeUpdate(
          "INSERT INTO personne (nom, prenom, taille) "
              + "values ('nom1', 'prenom1', 174)",
            Statement.RETURN_GENERATED_KEYS);
      int idGenere = -1;
      resultats = stmt.getGeneratedKeys();
      if (resultats.next()) {
        idGenere = resultats.getInt(1);
      } else {
        System.out.println("Impossible d'obtenir la valeur generee");
      }
      resultats.close();
      resultats = null;
      System.out.println("valeur id genere = " + idGenere);

    } catch (SQLException e) {
      e.printStackTrace();
    } finally {
      if (resultats != null) {
        try {
          resultats.close();
        } catch (SQLException ex) {
        }
      }
      if (stmt != null) {
        try {
          stmt.close();
        } catch (SQLException ex) {
        }
      }
      if (con != null) {
        try {
          con.close();
        } catch (SQLException ex) {
        }
      }
    }
  }
}
```

Résultat :

```
valeur id genere = 1
```

60.11.2.4. Le support des points de sauvegarde (savepoint)

Pour utiliser les transactions, il est nécessaire de demander la désactivation du mode auto-commit de la connexion. Il faut appeler la méthode setAutoCommit() avec le paramètre false de l'instance de la classe Connection qui encapsule la connexion à la base de données.

La transaction peut alors être validée ou annulée en totalité avec respectivement les méthodes commit() et rollback().

Avant la version 3.0 de JDBC, il n'était possible que de valider toutes les opérations ou d'annuler toutes les opérations de la transaction. Il n'était pas possible de réaliser des validations ou des annulations d'un sous-ensemble d'opérations de la transaction.

Avec la version 3.0 de JDBC, les savepoints permettent de définir des points nommés entre l'exécution de deux opérations de la transaction. Ce savepoint peut être considéré comme un marqueur. Toutes les opérations réalisées depuis la définition de ce marqueur peuvent être annulées sans que les opérations réalisées avant le marqueur ne soient annulées.

Pour définir un savepoint, il suffit d'appeler la méthode setSavePoint() de la classe Connection. Cette méthode renvoie un objet de type Savepoint qu'il faut passer en paramètre de la méthode rollback() pour annuler les opérations réalisées depuis la définition du savepoint.

Exemple :

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Savepoint;
import java.sql.Statement;

public class TestJdbc102 {
  public static void main(java.lang.String[] args) {
    Connection con = null;
    Statement stmt = null;
    ResultSet resultats = null;
    String requete = "";

    // chargement du pilote
    try {
      Class.forName("com.mysql.jdbc.Driver").newInstance();
    } catch (Exception e) {
      e.printStackTrace();
      System.exit(99);
    }

    try {
      String DBurl = "jdbc:mysql://localhost:3306/test";
      con = DriverManager.getConnection(DBurl);
      stmt = con.createStatement();

      con.setAutoCommit(false);
      con.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);

      stmt.executeUpdate("UPDATE personne SET nom = 'nom1 modif1' WHERE id=1");
      Savepoint svpt = con.setSavepoint("savepoint_1");
      stmt.executeUpdate("UPDATE personne SET nom = 'nom1 modif2' WHERE id=1");
      con.rollback(svpt);
      con.commit();

      // creation et execution de la requête
      requete = "SELECT * FROM personne";
      stmt = con.createStatement();
      resultats = stmt.executeQuery(requete);

      ResultSetMetaData rsmd = resultats.getMetaData();
      int nbCols = rsmd.getColumnCount();
      boolean encore = resultats.next();
      while (encore) {
        for (int i = 1; i <= nbCols; i++)
          System.out.print(resultats.getString(i) + " ");
        System.out.println();
        encore = resultats.next();
      }

      resultats.close();
      resultats = null;

    } catch (SQLException e) {
      e.printStackTrace();
      if (con != null) {
        try {
        con.rollback();
        } catch (SQLException ex) {
        }
      }
    } finally {
      if (resultats != null) {
        try {
          resultats.close();
        } catch (SQLException ex) {
        }
      }
      if (stmt != null) {
        try {
          stmt.close();
        } catch (SQLException ex) {
        }
      }
      if (con != null) {
        try {
          con.close();
        } catch (SQLException ex) {
        }
      }
    }
  }
}
```

Dans l'exemple ci-dessus, seule la première mise à jour est effective suite au commit de la transaction.

60.11.2.5. Le pool d'objets PreparedStatements

Il est maintenant possible d'utiliser un pool d'objets de type PreparedStatement. Cette mise en pool est transparente pour le développeur car elle est gérée par le pool de connexions.

Lors de la fermeture d'un objet PreparedStatement, celui-ci est mis dans le pool pour permettre une réutilisation et ainsi évite une recompilation d'un nouvel objet de ce type.

Ceci permet d'éviter la répétition des traitements coûteux effectués lors de la création d'un objet PreparedStatement (vérification et optimisation de la requête par la base de données).

Un pilote compatible avec la version 3.0 de JDBC va ainsi mettre en place un pool pour ces objets : à leur fermeture, les objets sont remis dans le pool. Lorsque le PreparedStatement est réutilisé, l'objet est repris du pool plutôt que recréé.

60.11.2.6. La définition de propriétés pour les pools de connexions

La version 3.0 de JDBC propose un contrôle plus précis sur les paramètres du pool de connexions tel que la taille du pool, le nombre minimum et maximum de connexions qu'il contient, ...

L'utilisation de ces propriétés peut améliorer les performances sans modification dans le code qui met en oeuvre l'API JDBC. En effet, elles affectent des mécanismes transparents pour le développeur et il n'est pas recommandé de modifier ces paramètres via l'API (il est préférable de les configurer au travers du serveur d'applications).

Ceci permet aussi de standardiser ces propriétés et de rendre la configuration moins dépendante des fournisseurs de pilotes.

Propriété	Description
maxStatements	Préciser le nombre maximum de statements gérés par le pool La valeur 0 indique une désactivation du mécanisme de mise en pool
initialPoolSize	Préciser le nombre de connexions créées par le pool à sa création
minPoolSize	Préciser le nombre de connexions minimum gérées par le pool. La valeur 0 précise que les connexions seront créées en fonction des besoins.
maxPoolSize	Préciser le nombre maximum de connexions gérées par le pool. La valeur 0 indique qu'il n'y a pas de maximum.
maxIdleTime	Préciser la durée en secondes avant qu'une connexion inutilisée du pool ne soit fermée. La valeur 0 indique qu'il n'y aura pas de cloture des connexions.

60.11.2.7. L'ajout de metadata pour obtenir la liste des types de données supportés

La méthode getType Info() permet d'obtenir un ResultSet qui contient la liste des types de données supportés par la base de données et le pilote.

60.11.2.8. L'utilisation de plusieurs ResultSets retournés par un CallableStatement

La version 2 de l'API JDBC ne permet à un objet Statement de n'avoir qu'un seul ResultSet ouvert à un instant donné.

La version 3 de l'API propose une fonctionnalité pour outrepasser cette limitation. Par défaut, la méthode execute() ferme le ResultSet retourné par sa précédente exécution. L'interface Statement a été enrichie d'une nouvelle méthode nommée getMoreResults(). Cette méthode attend un paramètre qui peut prendre les valeurs :

CLOSE_ALL_RESULTS	Les ResultSets précédemment ouverts sont fermés à l'appel de la méthode
CLOSE_CURRENT_RESULT	L'objet ResultSet courant est fermé lors de l'appel à la méthode
KEEP_CURRENT_RESULT	L'objet ResultSet courant reste ouvert lors de l'appel à la méthode

Elle retourne un booléen qui vaut true s'il y a encore au moins un ResultSet à traiter.

Cette fonctionnalité peut être pratique notamment pour utiliser des procédures stockées qui renvoient plusieurs curseurs de données.

60.11.2.9. Préciser si un ResultSet doit être maintenu ouvert ou fermé à la fin d'une transaction

Un ResultSet est automatiquement fermé à la fin d'une transaction. JDBC 3.0 propose une fonctionnalité qui permet de préciser si dans ce cas le ResultSet doit être maintenu ouvert ou fermé.

Une version surchargée des méthodes createStatement(), prepareCall() et prepareStatement() de la classe Connection attend en paramètre un entier nommé resultSetHoldability qui peut prendre les valeurs :

ResultSet.HOLD_CURSORS_OVER_COMMIT	Maintient l'objet ouvert après l'exécution d'un commit d'une transaction
ResultSet.CLOSE_CURSORS_AT_COMMIT	Ferme l'objet après l'exécution d'un commit d'une transaction

60.11.2.10. La mise à jour des données de type BLOB, CLOB, REF et ARRAY

La norme SQL99 propose les types de données BLOB (Binary Large OBject) et CLOB (Character Large OBject) pour permettre la gestion des données de grandes tailles respectivement de type binaire ou chaîne de caractères.

JDBC 2.0 ne proposait que des fonctionnalités pour lire des données de ces types. Chaque pilote souhaitant proposer des fonctionnalités pour les mettre à jour le faisait de façon particulière : ceci rend le code dépendant du fournisseur du pilote.

JDBC 3.0 propose en standard un mécanisme pour mettre à jour les champs de type BLOB et CLOB.

L'API propose dans l'interface java.sql.Blob une nouvelle méthode setBinaryStream() qui renvoie un objet de type OutputStream.

L'API propose dans l'interface java.sql.Clob plusieurs méthodes pour modifier le contenu du champ :

- setString() qui modifie le contenu avec la chaîne de caractères à partir de la position fournie en paramètre
- setAsciiStream() qui renvoie un objet de type Writer pour traiter un flux au format Ascii
- setCharacterStream() qui renvoie un objet de type Writer pour traiter un flux au format Unicode

L'exemple ci-dessous utilise la table suivante :

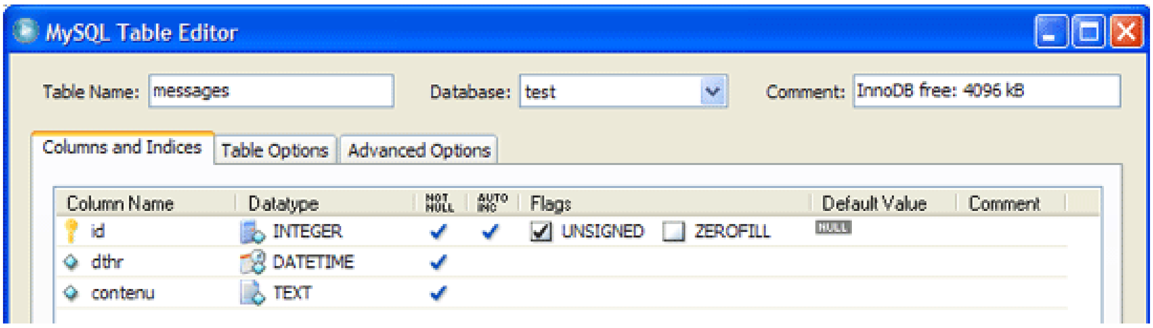


Table Name: messages Database: test Comment: InnoDB free: 4096 kB

Column Name	Datatype	NOT NULL	AUTO INC	Flags	Default Value	Comment
id	INTEGER	✓	✓	☑ UNSIGNED □ ZEROFILL	NULL	
dthr	DATETIME	✓				
contenu	TEXT	✓				

Exemple :

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.Writer;
import java.sql.Clob;
import java.sql.Connection;
import java.sql.Date;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class TestJdbc103 {

  public static void main(java.lang.String[] args) {
    Connection con = null;
    Statement stmt = null;
    PreparedStatement pstmt = null;
    ResultSet resultats = null;
    String requete = "";

    // chargement du pilote
    try {
      Class.forName("com.mysql.jdbc.Driver").newInstance();
    } catch (Exception e) {
      e.printStackTrace();
      System.exit(99);
    }

    try {
      String DBurl = "jdbc:mysql://localhost:3306/test";
      con = DriverManager.getConnection(DBurl);
      pstmt = con
          .prepareStatement("insert into messages (dthr, contenu)  Values (?, ?) ");

      pstmt.setDate(1, new Date(new java.util.Date().getTime()));

      Clob contenu = con.createClob();
      Writer writer = contenu.setCharacterStream(1);
      writer.write("contenu du message 1");
      writer.close();

      pstmt.setClob(2, contenu);
      pstmt.executeUpdate();

      // creation et execution de la requête
      requete = "SELECT id, dthr, contenu FROM messages where id=1";
      stmt = con.createStatement();
      resultats = stmt.executeQuery(requete);
      resultats.next();
      contenu = resultats.getClob(3);
      System.out.println("contenu=" + ClobToString(contenu));
    } catch (SQLException e) {
      e.printStackTrace();
      if (con != null) {
        try {
          con.rollback();
        } catch (SQLException ex) {
        }
      }
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      if (resultats != null) {
        try {
          resultats.close();
        } catch (SQLException ex) {
        }
      }
      if (stmt != null) {
        try {
          stmt.close();
        } catch (SQLException ex) {
        }
      }
      if (pstmt != null) {
        try {
          pstmt.close();
        } catch (SQLException ex) {
        }
      }
      if (con != null) {
        try {
          con.close();
        } catch (SQLException ex) {
        }
      }
    }
  }

  public static String ClobToString(Clob cl) throws IOException, SQLException {
    StringBuffer resultat = new StringBuffer("");
    if (cl != null) {
      String ligne = null;

      BufferedReader br = new BufferedReader(cl.getCharacterStream());

      while ((ligne = br.readLine()) != null)
        resultat.append(ligne);
    }
    return resultat.toString();
  }
}
```

60.11.3. JDBC 4.0

JDBC 4.0 est inclus dans Java SE 6. JDBC 4.0 propose plusieurs améliorations dans l'API :

- Le chargement automatique des implémentations de java.sql.Driver via l'API ServiceLoader
- Le support du type ROWID de SQL
- Le support de la conversion pour les types National Character
- Les améliorations dans la gestion des exceptions
- Le support du type XML de SQL:2003
- L'amélioration du support pour les objets larges (BLOB et CLOB)

60.11.3.1. Le chargement automatique des implémentations de Driver

Si le pilote est compatible JDBC 4.0, il n'est plus nécessaire de charger la classe du Driver avec l'invocation de la classe forName() de la classe Class. Cela est possible si l'implémentation du driver JDBC fournit une implémentation de l'interface java.sql.Driver en utilisant l'API ServiceLoader.

Au chargement de la classe DriverManager, un bloc d'initialisation statique recherche via l'API ServiceLoader toutes les implémentations de l'interface java.sql.Driver présente dns le le classpath.

Lors de l'invocation de la méthode getConnection(), le DriverManager cherche le pilote approprié parmi les pilotes JDBC qui ont été chargés lors de l'initialisation et ceux chargés explicitement à l'aide du même classloader de l'application actuelle.

60.11.3.2. Le support du type ROWID de SQL

L'interface RowID est ajoutée pour supporter le type de données ROWID de SQL qui est pris en charge par des bases de données telles qu'Oracle ou DB2. RowID est utile dans les cas où il y a plusieurs enregistrements qui n'ont pas de colonne d'identifiant unique et où vous devez stocker la sortie de la requête dans une collection qui n'autorise pas les doublons. La méthode getRowId() de la classe ResultSet permet d'obtenir un RowId et la méthode setRowId() d'un PreparedStatement pour utiliser le RowId dans une requête.

La valeur d'un objet de type RowId n'est pas portable entre les sources de données et doit être considérée comme spécifique à la source de données lors de l'utilisation des méthodes set() ou update() respectivement dans des PreparedStatement et des ResultSet. Elle ne doit donc pas être partagée entre différents objets de type Connection et ResultSet.

La méthode getRowIdLifetime() de la classe DatabaseMetaData permet d'obtenir la validité de la durée de vie de l'objet RowId sous la forme d'une des valeurs de l'énumération java.sql.RowIdLifeTime :

Valeur	Rôle
ROWID_UNSUPPORTED	Indiquer que la base de données ne supporte pas le type ROWID
ROWID_VALID_FOREVER	Indiquer que la durée de vie d'un RowId de cette source de données est illimitée. La durée de vie du RowID est illimitée tant que la ligne dans la table de la base de données n'est pas supprimée.
ROWID_VALID_OTHER	Indiquer que la durée de vie d'un RowId de cette source de données est indéterminée : elle ne correspond pas à ROWID_VALID_TRANSACTION, ROWID_VALID_SESSION ou ROWID_VALID_FOREVER. La durée de vie du RowID dépend de l'implémentation du fournisseur de la base de données.
ROWID_VALID_SESSION	Indiquer que la durée de vie d'un RowId de cette source de données est au moins la session qui le contient. La durée de vie du RowID est la durée de la session actuelle tant que la ligne dans la table de la base de données n'est pas supprimée.
ROWID_VALID_TRANSACTION	Indiquer que la durée de vie d'un RowId de cette source de données est au moins celle de la transaction qui le contient. La durée de vie du RowID est comprise dans la transaction actuelle tant que la ligne dans la table de la base de données n'est pas supprimée.

60.11.3.3. Les améliorations dans la gestion des exceptions

JDBC 4.0 propose plusieurs améliorations dans la gestion des exceptions.

Plusieurs exceptions filles de SQLException ont été ajoutées :

- java.sql.SQLClientInfoException
- java.sql.SQLDataException
- java.sql.SQLFeatureNotSupportedException
- java.sql.SQLIntegrityConstraintViolationException
- java.sql.SQLInvalidAuthorizationSpecException
- java.sql.SQLSyntaxErrorException
- java.sql.SQLTransactionRollbackException
- java.sql.SQLTransientConnectionException

La classe SQLException implémente l'interface Iterable<Throwable> permettant un parcours des exceptions chaînées.

Exemple (code jdbc 4.0) :

```
    try {
      // ...
    } catch (SQLException e) {
      for (Throwable t : e) {
        System.err.println("Erreur " + t);
      }
    } finally {
      // ...
    }
```

60.11.4. Le support du type XML de SQL

L'interface java.sql.SQLXML permet le mapping dans le langage Java pour le type SQL XML.

Le type XML de SQL est un type intégré qui stocke une valeur XML comme une valeur de colonne dans une table de base de données. Par défaut, les pilotes implémentent un objet SQLXML comme un pointeur logique vers les données XML plutôt que les données elles-mêmes. Un objet SQLXML est valide pour la durée de la transaction dans laquelle il a été créé.

L'interface SQLXML propose des méthodes pour accéder à la valeur XML avec différents types tels que String, Reader, Writer, Stream, Source, ...

On peut également accéder à la valeur XML par le biais d'une source ou d'un ensemble de résultats, qui sont utilisés avec les API de parser XML telles que DOM, SAX et StAX, ainsi qu'avec les transformations XSLT et les évaluations XPath.

Des méthodes getXMLSQL() et/ou setXMLSQL() dans les interfaces ResultSet, CallableStatement et PreparedStatement permettent de manipuler des valeurs XML.

La valeur XML d'une instance de type SQLXML peut être obtenue sous la forme d'un BinaryStream en utilisant la méthode getBinaryStream().

Exemple (code Java 6) :

```
SQLXML sqlxml = resultSet.getSQLXML(8);
InputStream binaryStream = sqlxml.getBinaryStream();
```

Il est possible d'analyser ce flux pour obtenir un Document DOM

Exemple (code Java 6) :

```
DocumentBuilder parser = DocumentBuilderFactory.newInstance().newDocumentBuilder();
Document document = parser.parse(binaryStream);
```

Ou pour obtenir un parser SAX

Exemple (code Java 6) :

```
SAXParser parser = SAXParserFactory.newInstance().newSAXParser();
parser.parse(binaryStream, monHandler);
```

Ou pour obtenir un parser StAX

Exemple (code Java 6) :

```
XMLInputFactory factory = XMLInputFactory.newInstance();
XMLStreamReader streamReader = factory.createXMLStreamReader(binaryStream);
```

Les bases de données peuvent utiliser une représentation optimisée pour le XML : dans ce cas, l'accès à la valeur en invoquant les méthodes getSource() et setResult() peut permettre d'améliorer les performances des traitements sans avoir à sérialiser la représentation en flux et à analyser le document XML.

La méthode getSource() permet obtenir un document DOM.

Exemple (code Java 6) :

```
   DOMSource domSource = sqlxml.getSource(DOMSource.class);
   Document document = (Document) domSource.getNode();
```

Ou de traiter le document XML avec des événements d'un parser SAX

Exemple (code Java 6) :

```
   SAXSource saxSource = sqlxml.getSource(SAXSource.class);
   XMLReader xmlReader = saxSource.getXMLReader();
   xmlReader.setContentHandler(monHandler);
   xmlReader.parse(saxSource.getInputSource());
```

Ou de traiter le document XML avec des événements StAX

Exemple (code Java 6) :

```
   StAXSource staxSource = sqlxml.getSource(StAXSource.class);
   XMLStreamReader streamReader = staxSource.getXMLStreamReader();
```

Ou d'appliquer une feuille de style XSLT sur le document XML pour créer un fichier.

Exemple (code Java 6) :

```
   File xsltFile = new File("a.xslt");
   File resultatFile = new File("resultat.xml");
   Transformer xslt = TransformerFactory.newInstance()
                                        .newTransformer(new StreamSource(xsltFile));
   Source source = sqlxml.getSource(null);
   Result result = new StreamResult(resultatFile);
   xslt.transform(source, result);
```

Il est aussi possible d'obtenir une valeur du document en appliquant une expression XPATH.

Exemple (code Java 6) :

```
   XPath xpath = XPathFactory.newInstance().newXPath();
   DOMSource domSource = sqlxml.getSource(DOMSource.class);
   Document document = (Document) domSource.getNode();
   String expression = "/employe/@nom";
   String valeur = xpath.evaluate(expression, document);
```

La méthode setSource() permet de mettre à jour une valeur XML à partir de différentes sources :

A partir d'un noeud DOM

Exemple (code Java 6) :

```
   DOMResult domResult = sqlxml.setResult(DOMResult.class);
   domResult.setNode(document);
```

A partir d'événements SAX

Exemple (code Java 6) :

```
SAXResult saxResult = sqlxml.setResult(SAXResult.class);
ContentHandler contentHandler = saxResult.getHandler();
contentHandler.startDocument();
// ajout des éléments et des attributs du document
contentHandler.endDocument();
```

A partir d'un flux StAX

Exemple (code Java 6) :

```
StAXResult staxResult = sqlxml.setResult(StAXResult.class);
XMLStreamWriter streamWriter = staxResult.getXMLStreamWriter();
```

A partir du résultat d'une transformation XSLT

Exemple (code Java 6) :

```
File sourceFile = new File("source.xml");
Transformer xslt = TransformerFactory.newInstance()
                                      .newTransformer(new StreamSource(xsltFile));
Source streamSource = new StreamSource(sourceFile);
Result result = sqlxml.setResult(null);
xslt.transform(streamSource, result);
```

Des valeurs XML incomplètes ou invalides peuvent provoquer la levée d'une exception SQLException lorsqu'une méthode de mise à jour est invoquée ou lorsque la méthode execute() est invoquée. Tous les flux doivent être fermés avant l'exécution de la fonction execute(), sinon une exception de type SQLException est levée.

La lecture et l'écriture de valeurs XML vers ou depuis un objet SQLXML ne peuvent se produire qu'une seule fois. Les états conceptuels readable et not readable déterminent si l'une des API de lecture renvoie une valeur ou lève une exception. Les états conceptuels de writable et not writable déterminent si l'une des API d'écriture va définir une valeur ou lever une exception.

L'état passe de readable à not readable lorsque la méthode free() ou l'une des méthodes de lecture est invoquée : getBinaryStream(), getCharacterStream(), getSource() et getString(). Les implémentations peuvent également changer l'état en non writable lorsque cela se produit.

L'état passe de writable à not writable lorsque la méthode free() ou l'une des méthodes d'écriture est invoquée : setBinaryStream(), setCharacterStream(), setResult() et setString(). Les implémentations peuvent également changer l'état en not readable lorsque cela se produit.

60.11.5. Le support des types National Character de SQL

Plusieurs améliorations ont été apportées à l'API JDBC pour permettre le support des types National Character de SQL :

- Les types de données JDBC NCHAR, NVARCHAR, LONGNVARCHAR et NBLOC
- Les méthodes setNString(), setNCharacterStream et setNClob() ont été ajoutées à l'interface java.sql.PreparedStatement
- Les méthodes getNString(), getNCharacterStream et getNClob() ont été ajoutées à l'interface java.sql.CallableStatement
- Les méthodes updateNClob(), updateNString et updateNCharacterStream() ont été ajoutées à l'interface java.sql.ResultSet

60.11.6. Des améliorations dans la prise en charge des objets de grande taille

Plusieurs améliorations ont été apportées dans la prise en charge des objets de grande taille (BLOB et CLOB) :

- Les méthodes createBlob(), createClob() et createNClob() ont été ajoutées à l'interface Connection pour respectivement obtenir une instance de type java.sql.Blob, java.sql.Clob et java.sql.NClob
- Des surchagres des méthodes setBlob(), setClob() et setNClob() ont été ajoutées à l'interface PreparedStatement pour fournir les valeurs en utilisant des instances de type InputStream ou Reader
- La méthode free() a été ajoutée aux interfaces Blob java.sql.Blob, java.sql.Clob et java.sql.NClob pour libérer les ressources

60.11.7. Les nouvelles fonctionnalités de l'API JDBC 4.1

JDBC 4.1 est inclus dans Java SE 7. JDBC 4.1 propose plusieurs améliorations dans l'API :

- L'utilisation des instances JDBC de type Connection, ResultSet et Statement dans un try-with-resources
- Les améliorations dans les méthodes valueOf() des classes Date et Timestamp
- Des mapping supplémentaires pour obtenir des objets Java selon des types JDBC dans l'interface CallableStatement
- Des changements dans l'API Connection : la possibilité d'interrompre une connexion et le support des schémas et des timeouts
- L'API RowSet 1.1 : la création d'instance de type RowSet avec des fabriques
- L'ajout d'une fonction de limitation des lignes retournées dans les requêtes SQL

60.11.7.1. L'utilisation des ressources JDBC dans un try-with-resources

Les interfaces java.sql.Connection, java.sql.Statement et java.sql.ResultSet héritent de l'interface java.lang.AutoClosable. Il est donc possible de définir des instances de ces types dans une instruction try-with-resources pour laisser le compilateur gérer proprement l'invocation de leur méthode close().

60.11.7.2. L'API RowSet 1.1 : la création d'instance de type RowSet avec des fabriques

Il est possible d'utiliser une instance de type javax.sql.rowset.RowSetFactory pour créer des instances de type RowSet.

Une telle instance peut être obtenue en utilisant la méthode newFactory() de la classe javax.sql.rowset.RowSetProvider.

Exemple (code Java 7) :

```
package fr.jmdoudoux.dej.jdbc;

import java.sql.SQLException;

import javax.sql.rowset.JdbcRowSet;
import javax.sql.rowset.RowSetFactory;
import javax.sql.rowset.RowSetProvider;

public class TestRowSetFactory {

  public static void main(String[] args) {
    RowSetFactory rowSetFactory = null;
    JdbcRowSet jdbcRowSet = null;

    try {
      rowSetFactory = RowSetProvider.newFactory();
      jdbcRowSet = rowSetFactory.createJdbcRowSet();

      jdbcRowSet.setUrl("jdbc:h2:~/appdb");
      jdbcRowSet.setname("");
      jdbcRowSet.setPassword("");
      jdbcRowSet.setCommand("SELECT * FROM employes");
      jdbcRowSet.execute();

      while (jdbcRowSet.next()) {
        System.out.println(jdbcRowSet.getInt(1) + " " + jdbcRowSet.getString(2)
                        + " " + jdbcRowSet.getString(3) + " " + jdbcRowSet.getString(4));
      }
    } catch (SQLException e) {
      e.printStackTrace();
    } finally {
      if (jdbcRowSet != null) {
        try {
          jdbcRowSet.close();
        } catch (SQLException e) {
          e.printStackTrace();
        }
      }
    }
  }
}
```

L'invocation de la méthode newFactory() retourne une instance de l'implémentation par défaut proposée par la classe com.sun.rowset.RowSetFactoryImpl.

Pour utiliser une autre implémentation, il faut utiliser la surcharge de la méthode qui attend en paramètre une chaîne de caractères qui précise le nom pleinement qualifié de l'implémentation à utiliser et le ClassLoader à utiliser.

L'interface RowSetFactory propose des méthodes permettant de créer les différents types d'implémentations de RowSet :

- createCachedRowSet()
- createFilteredRowSet()
- createJdbcRowSet()
- createJoinRowSet()
- createWebRowSet()

60.11.7.3. Mapping supplémentaires des objets Java aux types JDBC dans l'interface CallableStatement

Des méthodes ont été ajoutées pour mapper des valeurs JDBC vers des objets Java dans l'interface java.sql.CallableStatement :

- <T> T getObject(int parameterIndex, Class<T> type) throws SQLException
- <T> T getObject(String parameterName, Class<T> type) throws SQLException

Ces méthodes renvoient un objet représentant la valeur de OUT de l'index ou du nom du paramètre fourni convertie selon le type SQL dans le type Java demandé, si la conversion est prise en charge. Si la conversion n'est pas prise en charge ou si null est spécifié pour le type alors une exception de type SQLException est levée.

Une implémentation d'un driver doit obligatoirement supporter les conversions définies dans l'Appendix B des spécifications de JDBC. Des conversions supplémentaires peuvent aussi être proposées de manière spécifique à l'implémentation.

Si l'implémentation ne supporte pas cette méthode alors une exception de type SQLFeatureNotSupportedException est levée.

Si la conversion demandée n'est pas supportée alors la méthode lève une exception de type SQLException.

60.11.7.4. Les améliorations dans les méthodes valueOf() des classes Date et Timestamp

Les méthodes valueOf() des classes java.sql.Date et java.sql.Timestamp permettent d'omettre le zéro de tête pour le mois ou le jour.

Le format de la chaîne de caractère fournie en paramètre doit avoir la forme :

```
yyyy-[m]m-[d]d hh:mm:ss[.f...]
```

60.11.7.5. Des changements dans l'interface Connection

Plusieurs méthodes ont été ajoutées dans l'interface java.sql.Connection :

- void abort(Executor executor) : terminer une connexion ouverte. Elle ferme la connexion physique vers la base de données et libère les ressources associées. L'invocation sur une connexion déjà fermée n'a aucun effet
- int getNetworkTimeout() : retourner le nombre de millisecondes que le pilote attendra pour qu'une requête vers la base de données se termine. Si la limite est dépassée, une exception de type SQLException est levée
- String getSchema() : retourner le nom du schéma couramment utilisé
- void setNetworkTimeout(Executor executor,int milliseconds) : définir la durée maximale pendant laquelle une connexion ou des objets créés à partir de cette connexion attendent que la base de données réponde à une requête. Si une requête reste sans réponse, la méthode lèvre une exception de type SQLException, et la connexion ou les objets créés à partir de la connexion sont marqués comme fermés
- void setSchema(String schema) : définir le nom du schéma à utiliser. Si le pilote ne prend pas en charge les schémas, il ignorera silencieusement l'invocation. L'invocation de setSchema() n'a aucun effet sur les instances de type Statement précédemment créés ou préparés

60.11.7.6. L'ajout d'une fonction de limitation des lignes retournées

Une syntaxe particulière a été ajoutée pour limiter le nombre de lignes retournées par une requête. La syntaxe est de la forme :

```
{limit <clause de limitation>}
```

où la syntaxe de la clause de limitation est de la forme :

```
rows [offset row_offset]
```

La valeur donnée pour rows indique le nombre maximal de lignes à renvoyer par la requête.

Les crochets indiquent que la partie "offset row_offset" est facultative.

Le décalage row_offset indique le nombre de lignes à ignorer des lignes renvoyées par la requête avant de commencer à renvoyer des lignes. Une valeur de 0 pour row_offset signifie de n'ignorer aucune ligne. Les valeurs de rows et row_offset doivent être des nombres entiers supérieurs ou égaux à 0. Le nombre de lignes retournées lorsque la valeur de row vaut 0 peut être soit aucune soit toutes selon le driver JDBC utilisée.

Exemple (code Java 7) :

```
      String requetePage1 = "SELECT * FROM employes {limit 10}";
      String requetePage2 = "SELECT * FROM employes {limit 10 offset 10}";
```

60.11.8. Les nouvelles fonctionnalités de l'API JDBC 4.2

JDBC 4.2 est inclus dans Java SE 8. JDBC 4.2 propose plusieurs améliorations dans l'API :

- Le support du type JDBC REF_CURSOR dans les CallableStatement
- L'ajout de l'interface java.sql.DriverAction
- L'ajout de l'interface java.sql.SQLType
- L'ajout de l'énumération java.sql.JDBCType

60.11.8.1. L'ajout du support du type REF_CURSOR

Plusieurs bases de données, dont PL/SQL d'Oracle, supportent le type REF_CURSOR.

Pour retourner un REF_CURSOR à partir d'une procédure stockée, il faut utiliser la méthode registerOutParameter() de la classe CallableStatement et préciser Types.REF_CURSOR comme type de données à retourner. Pour récupérer l'instance de ResultSet représentant le REF_CURSOR, il faut invoquer la méthode getObject() de CallableStatement et préciser le type ResultSet comme type de données vers lequel convertir l'objet retourné. Le ResultSet obtenu n'est parcourable qu'en avançant et est en lecture seule.

Exemple (code Java 8) :

```
CallableStatement cstmt = conn.prepareCall("{call maProcStock(?)}");
cstmt.registerOutParameter(1, Types.REF_CURSOR);a
cstmt.executeQuery();
ResultSet rs = cstmt.getObject(1, ResultSet.class);
while(rs.next()){
  System.out.println(rs.getString(1));
}
```

Une exception de type SQLFeatureNotSupportedException est levée si le pilote JDBC ne prend pas en charge le type de données Types.REF_CURSOR et que la méthode registerOutParameter() est invoquée avec ce type en paramètre.

Pour déterminer si un driver JDBC supporte le type JDBC REF_CURSOR, il faut invoquer la méthode supportsRefCursors() de la classe DatabaseMetaData qui renvoie un booléen.

60.11.8.2. L'ajout de l'interface java.sql.DriverAction

Un pilote JDBC peut créer une implémentation de java.sql.DriverAction afin de recevoir des notifications lorsque DriverManager.deregisterDriver(java.sql.Driver) est invoquée.

L'interface java.sql.DriverAction ne définit qu'une seule méthode : void deregister()

Une implémentation de DriverAction n'est pas destinée à être utilisée directement par les applications. Un pilote JDBC peut choisir de créer son implémentation de DriverAction dans une classe privée pour éviter qu'elle ne soit appelée directement.

Le bloc d'initialisation statique du pilote JDBC doit invoquer DriverManager.registerDriver(java.sql.Driver, java.sql.DriverAction) afin d'indiquer au DriverManager quelle implémentation de DriverAction doit être invoquée lorsque le pilote JDBC est désenregistré.

La méthode DriverManager.deregisterDriver() requiert une permission SQLPermission("deregisterDriver"), si un SecurityManager est activé.

60.11.8.3. L'ajout de l'interface java.sql.SQLType

Une instance de type SQLType est utilisée pour identifier un type SQL générique, appelé type JDBC ou un type de données spécifique au fournisseur.

Elle définit plusieurs méthodes :

Méthode	Rôle
String getName()	Renvoyer le nom du SQLType qui représente un type de données SQL
String getVendor()	Renvoyer le nom du fournisseur qui prend en charge ce type de données
Integer getVendorTypeNumber()	Renvoyer le numéro de type spécifique au fournisseur pour le type de données

60.11.8.4. L'ajout de l'énumération java.sql.JDBCType

L'énumération java.sql.JDBCType contient les différents types SQL utilisables. Elle implémente l'interface java.sql.SQLType.

Il est préférable d'utiliser cette énumération à la place des constantes contenues dans java.sql.Types.

60.11.8.5. Le support de très nombreuses mises à jour exécutées

Historiquement, les méthodes de mises à jour de l'API JDBC envoient le nombre de modifications effectuées par l'exécution d'une requête sous la forme d'un entier de type int.

Plusieurs méthodes executeLargeXXX() ont été ajoutées à l'interface Statement :

- default long getLargeUpdateCount() throws SQLException
- default long getLargeMaxRows()throws SQLException
- default void setLargeMaxRows(long rows) throws SQLException

Ces méthodes retournent une valeur de type long au lieu des méthodes existantes qui renvoient une valeur de type int.

Il faut utiliser ses méthodes si le nombre de mises à jour effectuées dépasse la valeur Integer.MAX_VALUE.

Trois autres méthodes ont aussi été ajoutées dans l'interface Statement :

- default long getLargeUpdateCount() throws SQLException
- default long getLargeMaxRows()throws SQLException
- default void setLargeMaxRows(long rows) throws SQLException

Une nouvelle méthode long[] getLargeUpdateCounts() et un nouveau constructeur BatchUpdateException(String reason, String SQLState, int vendorCode, long[] updateCounts, Throwable cause) ont été ajouté à l'exception BatchUpdateException.

60.12. MySQL et Java

MySQL est une des bases de données open source les plus populaires.

60.12.1. Les opérations de base avec MySQL

Cette section est une présentation rapide de quelques fonctionnalités de base pour pouvoir utiliser MySQL. Pour un complément d'informations sur toutes les possibilités de MySQL, consultez la documentation de cet excellent outil.

Pour utiliser MySQL, il faut s'assurer que le serveur est lancé sinon il faut exécuter la commande
c:\mysql\bin\mysqld-max

Pour exécuter des commandes SQL, il faut utiliser l'outil c:\mysql\bin\mysql. Cet outil est un interpréteur de commandes en mode console.

Exemple : pour voir les databases existantes

```
mysql>show databases;
+----------+
| Database |
+----------+
| mysql    |
| test     |
+----------+
2 rows in set (0.00 sec)
```

Un des premières choses à faire, c'est de créer une base de données qui va recevoir les différentes tables.

Exemple : Pour créer une nouvelle base de données nommée "testjava"

```
mysql> create database testjava;
Query OK, 1 row affected (0.00 sec)

mysql>use testjava;
Database changed
```

Cette nouvelle base de données ne contient aucune table. Il faut créer la ou les tables utiles aux développements.

Exemple : Création d'une table nommée personne contenant trois champs : nom, prenom et date de naissance

```
mysql> show tables;
Empty set (0.06 sec)

mysql> create table personne (nom varchar(30), prenom varchar(30), datenais date
);
Query OK, 0 rows affected (0.00 sec)

mysql>show tables;
+--------------------+
| Tables_in_testjava |
+--------------------+
| personne           |
+--------------------+
1 row in set (0.00 sec)
```

Pour voir la définition d'une table, il faut utiliser la commande DESCRIBE :

Exemple : voir la définition de la table

```
mysql> describe personne;
+----------+-------------+------+-----+---------+-------+
| Field    | Type        | Null | Key | Default | Extra |
+----------+-------------+------+-----+---------+-------+
| nom      | varchar(30) | YES  |     | NULL    |       |
| prenom   | varchar(30) | YES  |     | NULL    |       |
| datenais | date        | YES  |     | NULL    |       |
+----------+-------------+------+-----+---------+-------+
3 rows in set (0.00 sec)
```

Cette table ne contient aucun enregistrement. Pour ajouter un enregistrement, il faut utiliser la commande SQL insert.

Exemple : insertion d'une ligne dans la table

```
mysql> select * from personne;
Empty set (0.00 sec)

mysql> insert into personne values ('Nom 1','Prenom 1','1970-08-11');
Query OK, 1 row affected (0.05 sec)

mysql> select * from personne;
+-------+----------+------------+
| nom   | prenom   | datenais   |
+-------+----------+------------+
| Nom 1 | Prenom 1 | 1970-08-11 |
+-------+----------+------------+
1 row in set (0.00 sec)
```

Il existe des outils graphiques libres ou commerciaux pour faciliter l'administration et l'utilisation de MySQL.

60.12.2. L'utilisation de MySQL avec JDBC

Le téléchargement du pilote JDBC Connector/J se fait àl'URL https://dev.mysql.com/downloads/connector/j/ .

Pour utiliser l'archive, il faut la décompresser, par exemple dans le répertoire d'installation de mysql.

Il faut s'assurer que les fichiers jar sont accessibles dans le classpath ou les préciser manuellement lors de la compilation et de l'exécution comme dans l'exemple ci-dessous.

Exemple :

```
import java.sql.*;

public class TestJDBC11 {

  private static void affiche(String message) {
    System.out.println(message);
  }

  private static void arret(String message) {
    System.err.println(message);
    System.exit(99);
  }

  public static void main(java.lang.String[] args) {
    Connection con = null;
    ResultSet resultats = null;
    String requete = "";

    // chargement du pilote
    try {
      Class.forName("org.gjt.mm.mysql.Driver").newInstance();
    } catch (Exception e) {
      arret("Impossible decharger le pilote jdbc pour mySQL");
    }

    //connexion a la base de données
    affiche("Connexion a la base de donnees");
    try {

      String DBurl = "jdbc:mysql://localhost/testjava";
      con = DriverManager.getConnection(DBurl);
    } catch (SQLException e) {
      arret("Connexion a la base de donnees impossible");
    }

    //creation et execution de la requête
    affiche("Creation et execution dela requête");
    requete = "SELECT * FROM personne";

    try {
      Statement stmt = con.createStatement();
      resultats = stmt.executeQuery(requete);
    } catch (SQLException e) {
      arret("Anomalie lors de l'execution de la requete");
    }

    //parcours des données retournees
    affiche("Parcours des donnees retournees");
    try {
      ResultSetMetaData rsmd = resultats.getMetaData();
      int nbCols = rsmd.getColumnCount();
      boolean encore = resultats.next();

      while (encore) {

        for (int i = 1; i <= nbCols; i++)
          System.out.print(resultats.getString(i) + "");
        System.out.println();
        encore = resultats.next();
      }

      resultats.close();
    } catch (SQLException e) {
      arret(e.getMessage());
    }
  }
}
```

Résultat :

```
C:\java>javac -classpath c:\java\lib\mysql-connector-j-8.2.0.jar TestJDBC11.java
C:\java>
C:\java>java -cp .;c:\java\lib\mysql-connector-j-8.2.0.jar TestJDBC11
Connexion a la base de donnees
Creation et execution de la requ_te
Parcours des donnees retournees
Nom 1 Prenom 1 1970-08-11
```

60.13. L'amélioration des performances avec JDBC

Les opérations d'accès à une base de données sont généralement nombreuses et source de nombreux ralentissements dans une application : il est donc nécessaire de procéder à des opérations de tuning sur ces traitements.

Ces opérations doivent être prises en compte dès le début d'un projet.

Comme pour toutes opérations de tuning, des outils de test de charge et de monitoring sont nécessaires pour pouvoir mesurer les performances des accès aux bases de données.

Le choix des outils utilisés peut grandement influencer les performances notamment :

- La version du JRE
- Le pilote JDBC (la version de JDBC supportée, optimisations proposées, cache, ...)

Voici quelques recommandations de base qui permettent d'améliorer les performances regroupées par catégories.

60.13.1. Le choix du pilote JDBC à utiliser

La qualité du pilote JDBC est importante notamment en termes de rapidité, type de pilote, version de JDBC supportée, ...

Le type du pilote influe grandement sur les performances :

- Le type 1 (pont JDBC/ODBC) : les pilotes de ce type sont à éviter car les différentes couches mises en oeuvre (JDBC, pilote JDBC, ODBC, pilote ODBC, base de données) dégradent les performances
- Le type 2 (utilise une API native) : les pilotes de ce type ont généralement des performances moyennes
- Le type 3 (JDBC, pilote JDBC, middleware, DB) : les pilotes de type 3 communiquent avec un middleware généralement sur le serveur. Ils sont le plus souvent plus performants que ceux de type 1 et 2
- Le type 4 (JDBC, pilote JDBC, DB) les pilotes de type 4 offre en général les meilleures performances car ils sont écrits en Java et communiquent directement avec la base de données

Il est donc préférable d'utiliser des pilotes de type 4 ou 3.

Il peut être intéressant de tester le pilote proposé par le fournisseur de la base de données mais aussi de tester des pilotes fournis par des tiers.

Il est préférable d'utiliser un pilote qui supporte la version la plus récente de JDBC.

60.13.2. La mise en oeuvre de bonnes practiques

Plusieurs bonnes practiques sont communément mises en oeuvre lors de l'utilisation de JDBC :

- Fermer les ressources inutilisées dès que possible (Connection, Statement, ResultSet)
- Ne retourner que les données utiles lors de l'utilisation de requêtes SQL
- Toujours assurer un traitement des warnings et des exceptions

Toutes les instances de type Statement seront fermées lorsque la connexion qui les a créées est fermée. Toutefois, c'est une bonne pratique de fermer les Statement dès que leur exploitation est terminée car cela permet à toutes les ressources externes utilisées par le Statement d'être libérées immédiatement.

La fermeture d'un objet Statement entraîne la fermeture et l'invalidation de toutes les instances de ResultSet créées par cette instance de Statement. Les ressources détenues par l'objet ResultSet peuvent ne pas être libérées jusqu'à ce que le ramasse-miettes fasse son office, c'est donc une bonne pratique de fermer explicitement une instance de type ResultSet qui n'est plus utile.

Une fois qu'un Statement a été fermé, toute tentative d'accès à l'une de ses méthodes, à l'exception de la méthode isClosed() ou close() lève une exception de type SQLException. Ces bonnes pratiques s'appliquent aussi aux objets de type PreparedStatement et CallableStatement.

60.13.3. L'utilisation des connexions et des Statements

Il est préférable de maintenir une connexion ouverte et la réutiliser plutôt que de créer une nouvelle connexion et la fermer à chaque opération sur la base de données. C'est ce que permettent les pools de connexions.

Si les accès sont en lecture seule, il est préférable d'utiliser la méthode setReadOnly() de l'objet Connection en lui passant le paramètre true pour permettre au pilote de faire des optimisations.

Il est possible de paramétrer la quantité de données reçues de la base de données en utilisant les méthodes setMaxRows(), setMaxFieldSize() et setFetchSize() de l'interface Statement.

La méthode nativeSQL() de la classe Connection permet d'obtenir la requête SQL native qui sera envoyée par le pilote à la base de données.

60.13.4. L'utilisation d'un pool de connexions

La création d'une connexion vers une base de données est coûteuse en temps et en ressources. Le rôle d'un pool de connexions est de maintenir un certain nombre de connexions ouvertes à disposition de l'application dans un pool et de les proposer à la demande.

Un pool peut être fourni par l'environnement d'exécution (par exemple un serveur d'application) soit être fourni par un tiers (il en existe plusieurs en open source) soit être développé de toute pièce.

L'utilisation d'un pool de connexions est sûrement l'action la plus efficace pour des applications qui utilisent les accès à la base de données de façon importante.

Il peut être important de configurer correctement le pool de connexions utilisé notamment la taille du pool pour limiter la création et la destruction des connexions.

Un pool de connexions peut fonctionner selon deux modes principaux :

- Taille fixe : l'obtention d'une connexion alors que toutes celles du pool sont en cours d'utilisation implique l'attente de la libération d'une des connexions
- Taille variable : le pool possède une taille minimale et maximale avec une possibilité d'extension en cas de surcharge de travail

60.13.5. La configuration et l'utilisation des ResultSets en fonction des besoins

Une bonne configuration et utilisation des objets de type ResultSet peuvent améliorer les performances.

Il faut utiliser le curseur adapté aux besoins :

- TYPE_FORWARD_ONLY : aucune mise à jour, à utiliser pour des lectures séquentielles
- TYPE_SCROLL-SENSITIVE : parcours avec mise à jour immédiate
- TYPE_SCROLL_INSENSITIVE : parcours avec mises à jour à la fermeture de la connexion. Il faut éviter ce type pour des requêtes qui ne retournent qu'une seule occurrence

Il faut éviter d'utiliser la méthode getObject() mais utiliser la méthode getXXX() adaptée au type d'une donnée pour extraire sa valeur.

60.13.6. L'utilisation des PreparedStatement

Il est intéressant d'utiliser les PreparedStatement notamment pour les requêtes qui sont exécutées plusieurs fois avec les mêmes paramètres ou des paramètres différents (les valeurs des données fournies à la requête peuvent être paramétrées).

Une même requête exécutée avec des paramètres différents nécessite certains traitements identiques par la base de données : une partie de ces traitements est réalisé une et une seule fois lors de la première utilisation d'un PreparedStatement par une connexion. Les appels suivants avec la même connexion sont plus rapides puisque ces traitements ne sont pas refaits.

A partir de JDBC 3.0, les objets de type PreparedStatement peuvent être stockés dans un cache partagé des connexions d'un même pool : ceci améliore les performances car cela évite d'avoir certaines opérations mises en oeuvre à chaque appel (vérification de la syntaxe, optimisation des chemins d'accès et des plans d'exécution, ...).

60.13.7. La maximisation des traitements effectués par la base de données :

Par exemple pour obtenir un nombre d'occurrences, il est préférable d'effectuer une requête SQL contenant un count(*) plutôt que de parcourir un ResultSet avec un compteur incrémenté à chaque itération.

Il est possible d'utiliser les procédures stockées pour les traitements lourds ou complexes sur la base de données plutôt que d'effectuer plusieurs appels à la base de données pour réaliser les mêmes traitements côté Java. Les performances sont accrues car les traitements sont réalisés par la base de données ce qui évite notamment des échanges réseaux.

Attention ceci n'est vrai que pour des traitements complexes : une simple requête SQL s'exécutera plus rapidement qu'en appelant une procédure stockée qui contient simplement la requête.

Il est préférable d'utiliser les marqueurs de paramètres dans les requêtes des objets de type Statement plutôt que de les passer en dur dans la requête.

60.13.8. L'exécution de plusieurs requêtes en mode batch

Il est possible d'exécuter de nombreuses requêtes en utilisant les BatchUpdates : ceci permet de regrouper plusieurs opérations sur la base de données en un seul appel.

Pour mettre en oeuvre le BatchUpdates, il faut :

- Inhiber l'autocommit en utilisant la méthode setAutoCommit(false) de l'objet Connection
- Ajouter les traitements SQL en utilisant la méthode Statement.addBatch()
- Exécuter les traitements en utilisant la méthode Statement.executeBatch()

60.13.9. Prêter une attention particulière aux transactions

Il faut minimiser les conflits engendrés par les transactions (deadlocks notamment)

Par défaut, une connexion est en mode autocommit ce qui implique la création et la validation d'une transaction à chaque opération.

L'autocommit qui est le mode par défaut pour une connexion implique une nouvelle transaction pour chaque opération réalisée.

Il est donc préférable d'inhiber l'autocommit en passant false à la méthode setAutoCommit() et de réaliser plusieurs opérations dans une même transaction avant de la valider par un commit. Il ne faut cependant pas laisser une transaction ouverte trop longtemps pour éviter des problèmes de concurrence d'accès : une transaction posant des verrous sur la base de données, il est important de minimiser le temps d'exécution d'une transaction.

Le choix du mode de transaction influe sur les performances. Il faut choisir en fonction des besoins car plus le niveau d'isolation est important moins les performances sont bonnes :

- TRANSACTION_NONE,
- TRANSACTION_READ_UNCOMMITED,
- TRANSACTION_READ_COMMITED,
- TRANSACTION_REPEATABLE_READ,
- TRANSACTION_SERIALIZABLE

La méthode setTransactionIsolation() permet de préciser le mode de transaction à utiliser.

L'utilisation de transactions locales est plus performants que celle de transactions distribuées si elles ne sont pas nécessaire.

60.13.10. L'utilisation des fonctionnalités de JDBC 3.0

JDBC 3.0 propose des fonctionnalités pour améliorer les performances notamment au niveau du cache des connexions et des objets de type PreparedStatement, les objets RowSet, ...

Le pool de connexions et le pool de Statement travaillent ensemble pour qu'une connexion puisse utiliser un objet Statement du pool qui a été créé par une autre connexion. Ainsi un objet de type Statement n'est plus lié à une connexion mais partagé entre les connexions d'un même pool ce qui améliore encore les performances.

Un objet de typeCacheRowSet permet d'obtenir des données, de libérer la connexion, de les modifier en local et de les resynchroniser dans la base de données avec une nouvelle connexion. Il n'est donc pas nécessaire d'avoir une connexion ouverte durant tous les traitements. Il faut cependant prêter une attention particulière aux éventuels conflits de mise à jour

Les savePoints sont assez gourmands en ressources : il est nécessaire de libérer ces ressources en utilisant la méthode releaseSavePoint() de la classe Connection.

60.13.11. Les optimisations sur la base de données

Les optimisations côté Java sont importantes mais il est aussi nécessaire de procéder à des optimisations côté base de données, généralement réalisées par un DBA dans des structures de taille moyenne ou importante.

Les quelques optimisations fournies ci-dessous sont assez généralistes : elles ne dispensent pas d'effectuer des optimisations spécifiques à la base de données utilisée.

- Il faut mettre en place les index utiles : l'ajout d'un index peut dramatiquement améliorer les performances mais trop d'index nuit car la base de données doit les maintenir à jour.
- Les bases de données fournissent des outils pour afficher le plan d'exécution d'une requête ou d'une procédure stockée pour faciliter leur optimisation (ajout d'index, modification des clauses de la requête, ...)

- Si le pilote JDBC le permet, il peut être intéressant d'ajuster la taille des paquets échangés avec la base de données
- Utiliser le type de données approprié aux données stockées en fonction des besoins (exemple : représenter une date avec un type DateTime (plus de sécurité dans l'utilisation de la donnée) ou varchar (traitement plus rapide))
- Il est préférable de stocker les chaînes de caractères en Unicode (encodage en UTF-8 par exemple) dans la base de données pour éviter les conversions. Ceci peut cependant avoit un impact sur la taille de la base de données

60.13.12. L'utilisation d'un cache

L'utilisation d'un cache pour stocker les données peut éviter des accès à la base de données. Ceci est particulièrement adapté pour des données lues de façon répétitives ou dont les valeurs évoluent très peu ou pas du tout (données en lecture seule, données de références, ...).

Il faut cependant faire attention à la durée de vie des objets dans le cache afin d'éviter des problèmes de rafraichissement de données.

Il ne faut pas mettre en cache les objets de type ResultSet : il faut les parcourir, stocker les données dans des objets du domaine et mettre ces objets dans le cache.

61. JDO (Java Data Object)

Chapitre 61

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Technologie legacy
Ce chapitre est conservé pour des raisons historiques

JDO (Java Data Object) est la spécification du JCP numéro 12 qui propose une technologie pour assurer la persistance d'objets Java dans un système de gestion de données. La spécification regroupe un ensemble d'interfaces et de règles qui doivent être implémentées par un fournisseur tiers.

La version 1.0 de cette spécification a été validée au premier trimestre 2002. Elle permet de réaliser le mapping entre des données stockées dans un format particulier (bases de données ...) et un objet, ce qui a toujours été difficile. JDO propose de faciliter cette tâche en fournissant un standard.

Ce chapitre contient plusieurs sections :

- ♦ La présentation de JDO
- ♦ Un exemple avec Lido
- ♦ L'API JDO
- ♦ La mise en oeuvre
- ♦ Le parcours de toutes les occurrences
- ♦ La mise en oeuvre de requêtes

Remarque : face à l'insuccès de JDO, il est dorénavant préférable d'utiliser JPA.

61.1. La présentation de JDO

Les principaux buts de JDO sont :

- la facilité d'utilisation (gestion automatique du mapping des données)
- la persistance universelle : persistance vers tout type de système de gestion de ressources (bases de données relationnelles, fichiers, ...)
- la transparence vis à vis du système de gestion de ressources utilisé : ce n'est plus le développeur mais JDO qui dialogue avec le système de gestion de ressources
- la standardisation des accès aux données
- la prise en compte des transactions

Le développement avec JDO se déroule en plusieurs étapes :

1. écriture des objets contenant les données (des beans qui encapsulent les données) : un tel objet est nommé instance JDO
2. écriture des objets qui utilisent les objets métiers pour répondre aux besoins fonctionnels. Ces objets utilisent l'API JDO.
3. écriture du fichier metadata qui précise le mapping entre les objets et le système de gestion des ressources. Cette partie est très dépendante du système de gestion de ressources utilisé

4. enrichissement des objets métiers
5. configuration du système de gestion des ressources

JDBC et JDO ont les différences suivantes :

JDBC	JDO
orienté SQL	orienté objets
le code doit être ajouté explicitement	code est ajouté automatiquement
	gestion d'un cache
	mapping réalisé automatiquement ou à l'aide d'un fichier de configuration au format XML
utilisation avec un SGBD uniquement	utilisation de tout type de format de stockage

JDO est une spécification qui définit un standard : pour pouvoir l'utiliser il faut utiliser une implémentation fournie par un fournisseur. Plusieurs implémentations existent et le choix de l'une d'elle doit tenir compte des performances, du prix, du support des cibles de stockage des données, etc ... L'intérêt des spécifications est qu'il est possible d'utiliser le même code avec des implémentations différentes tant que l'on utilise uniquement les fonctionnalités précisées dans les spécifications.

Chaque implémentation est capable d'utiliser un ou plusieurs systèmes de stockage de données particulier (base de données relationnel, base de données objets, fichiers, ...).

Attention : tous les objets ne peuvent pas être rendus persistants avec JDO.

61.2. Un exemple avec Lido

Les exemples de cette section ont été réalisés avec Lido Community Edition version 1.4.5. de la société Libelis.

Remarque : la société Libelis, renommée en Xcalia n'existe malheureusement plus. Le contenu de ce chapitre est conservé à titre indicatif.

Pour lancer l'installation, il suffit de double cliquer sur le fichier LiDO_Community_1[1].4.5.jar ou de saisir la commande :

Installation de Lido community edition de Libelis

```

java -jar LiDO_Community_1[1].4.5.jar
```

L'installation s'opère simplement en suivant les différentes étapes de l'assistant.

Le premier exemple permet simplement de rendre persistant un objet instancié dans une base de données MySQL.

Pour faciliter la mise en oeuvre des différentes étapes, un script batch pour Windows sera écrit tout au long de cette section et exécuté. Ce script débute par une initialisation de certaines variables d'environnement.

Début du script

```
@echo off
REM - script permettant la compilation, l'enrichissement, la creation du schema de
REM - base de données et l'execution du code de test de JDO avec Lido 1.4.5.
set LIDO_HOME=C:\java\LiDO
set JAVA_HOME=C:\java\j2sdk1.4.2_02

echo initialisation
echo.
```

```
SET OLD_PATH=%PATH%
SET PATH=%LIDO_HOME%\bin;%JAVA_HOME%\bin

SET OLD_CLASSPATH=%CLASSPATH%
SET CLASSPATH=.;.\mm.mysql-2.0.14-bin.jar
SET CLASSPATH=%CLASSPATH%;%JAVA_HOME%\lib\tools.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\lido-api.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\jdo_1_0_0.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\j2ee.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\bin
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\lido-dev.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\lido-rdb.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\lido-rt.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\lido.tasks
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\lido.tld
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\skinlf.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\connector_1_0_0.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\jta_1_0_1.jar
```

Le début du script initialise 4 variables :

- LIDO_HOME : cette variable contient le chemin du répertoire dans lequel Lido a été installé
- JAVA_HOME : cette variable contient le chemin du répertoire dans lequel J2SDK a été installé
- PATH : cette variable contient les différents répertoires contenant des exécutables
- CLASSPATH : cette variable contient les différents répertoires et fichiers jar nécessaires pour la compilation et l'exécution

Pour des raisons de facilité, le répertoire courant "." est ajouté dans le CLASSPATH. Le pilote JDBC pour MySQL est aussi ajouté à cette variable.

61.2.1. La création de la classe qui va encapsuler les données

Le code de cet objet reste très simple puisque c'est simplement un bean encapsulant une personne contenant des attributs nom, prenom et datenaiss.

Exemple :

```
package testjdo;

import java.util.*;

public class Personne {
      private String nom ="";
      private String prenom = "";
      private Date datenaiss = null;

      public Personne(String pNom, String pPrenom,Date pDatenaiss) {
            nom=pNom;
            prenom=pPrenom;
            datenaiss=pDatenaiss;
      }

      public String getNom() { return nom; }

      public String getPrenom() { return prenom;}

      public Date getDatenaisse() { return datenaiss; }

      public void setNom(String pNom) { nom = pNom; }

      public void setPrenom(String pPrenom) { nom = pPrenom; }

      public void setDatenaiss(Date pDatenaiss) { datenaiss = pDatenaiss; }

}
```

61.2.2. La création de l'objet qui va assurer les actions sur les données

Cet objet va utiliser des objets JDO pour réaliser les actions sur les données. Dans l'exemple ci-dessous, une seule action est codée : l'enregistrement dans la table des données du nouvel objet de type Personne instancié.

Exemple :

```
package testjdo;

import javax.jdo.*;
import java.util.*;

public class PersonnePersist {

      private PersistenceManagerFactory pmf = null;
      private PersistenceManager        pm = null;
      private Transaction               tx  = null;

      public PersonnePersist() {
            try {

            pmf = (PersistenceManagerFactory) (
            Class.forName("com.libelis.lido.PersistenceManagerFactory").newInstance());
            pmf.setConnectionDriverName("org.gjt.mm.mysql.Driver");
            pmf.setConnectionURL("jdbc:mysql://localhost/testjdo");
      } catch (Exception e ){
            e.printStackTrace();
            }
      }

      public void enregistrer() {
            Personne p = new Personne("mon nom","mon prenom",new Date());
            pm = pmf.getPersistenceManager();
            tx = pm.currentTransaction();
            tx.begin();
            pm.makePersistent(p);
            tx.commit();
            pm.close();
      }

      public static void main(String args[]) {
            PersonnePersist pp = new PersonnePersist();
            pp.enregistrer();
      }
}
```

61.2.3. La compilation

Les deux classes définies ci-dessus doivent être compilées normalement en utilisant l'outil javac.

Exemple :

```
...
echo Compilation en cours
javac -classpath %CLASSPATH% testjdo\*.java
echo Compilation effectuee
echo.
...
```

61.2.4. La définition d'un fichier metadata

Le fichier metadata est un fichier au format XML qui précise le mapping à réaliser.

Exemple : metadata.jdo

```
<? xml version="1.0" ?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
```

```
<jdo>
  <package name="testjdo">
    <class name="Personne" identity-type="datastore">
      <field name="nom" />
      <field name="prenom" />
      <field name="datenaiss" />
    </class>
  </package>
</jdo>
```

61.2.5. L'enrichissement des classes contenant des données

Pour permettre une bonne exécution, il faut enrichir l'objet Personne compilé avec du code pour assurer la persistance par JDO. Lido fournit un outil pour réaliser cette tâche. Cet outil dépend de l'implémentation qui en est faite par le fournisseur de la solution JDO.

La suite du script : enrichissement

```
...
echo Enrichissement
java -cp %CLASSPATH% com.libelis.lido.Enhance -metadata metadata.jdo -verbose
echo Enrichissement effectue
echo.
...
```

Le fichier Personne.class est enrichi (sa taille passe de 867 octets à 9693 octets)

61.2.6. La définition du schéma de la base de données

Lido fournit un outil qui permet de générer les tables de la base de données. Ces tables contiennent le mapping de la base de données ainsi que les données techniques nécessaires aux traitements.

Les paramètres nécessaires à l'outil de Libelis pour définir le schéma de la base de données doivent être rassemblés dans un fichier .properties.

propriété pour une base de données de type MySQL

```
# lido.properties file
# jdo standard properties
javax.jdo.option.connectionURL=jdbc:mysql://localhost/testjdo
javax.jdo.option.ConnectionDriverName=org.gjt.mm.mysql.Driver
javax.jdo.option.connectionUserName=root
javax.jdo.option.connectionPassword=
javax.jdo.option.msWait=5
javax.jdo.option.multithreaded=false
javax.jdo.option.optimistic=false
javax.jdo.option.retainValues=false
javax.jdo.option.restoreValues=true
javax.jdo.option.nontransactionalRead=true
javax.jdo.option.nontransactionalWrite=false
javax.jdo.option.ignoreCache=false


# set to PM, CACHE, or SQL to have some traces
# ex:
#lido.trace=SQL,DUMP,CACHE

#  set the Statement pool size
lido.sql.poolsize=10
lido.cache.entry-type=weak

# set the max batched statement
# 0: no batch
# default is 20
```

```
lido.sql.maxbatch=30
lido.objectpool=90

# set for PersistenceManagerFactory pool limits
lido.minPool=1
lido.maxPool=10

jdo.metadata=metadata.jdo
```

Il suffit alors d'utiliser l'application DefineSchema fournie par Lido en lui passant en paramètre le fichier .properties et le fichier .jdo

La suite du script : création du schéma de la base de données

```
 ...
echo DefineSchema en cours
java com.libelis.lido.DefineSchema -properties testjdo.properties -metadata metadata.jdo
echo DefineSchema termine
echo.
...
```

Il est facile de vérifier les traitements effectués par l'outil DefineSchema :

Exemple :

```
C:\java\testjdo>mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 25 to server version: 4.0.16-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> use testjdo
Database changed
mysql> show tables;
+-------------------+
| Tables_in_testjdo |
+-------------------+
| lidoidmax         |
| lidoidtable       |
| t_personne        |
+-------------------+
3 rows in set (0.00 sec)

mysql> describe lidoidmax;
+----------+-----------+------+-----+---------+-------+
| Field    | Type      | Null | Key | Default | Extra |
+----------+-----------+------+-----+---------+-------+
| LIDOLAST | bigint(20) | YES  |     | NULL    |       |
+----------+-----------+------+-----+---------+-------+
1 row in set (0.00 sec)

mysql> describe lidoidtable;
+----------+-------------+------+-----+---------+-------+
| Field    | Type        | Null | Key | Default | Extra |
+----------+-------------+------+-----+---------+-------+
| LIDOID   | bigint(20)  |      | PRI | 0       |       |
| LIDOTYPE | varchar(255) | YES  | MUL | NULL    |       |
+----------+-------------+------+-----+---------+-------+
2 rows in set (0.00 sec)

mysql> describe t_personne;
+-----------+------------+------+-----+---------+-------+
| Field     | Type       | Null | Key | Default | Extra |
+-----------+------------+------+-----+---------+-------+
| LIDOID    | bigint(20) |      | PRI | 0       |       |
| nom       | varchar(50) | YES  |     | NULL    |       |
| prenom    | varchar(50) | YES  |     | NULL    |       |
| datenaiss | datetime   | YES  |     | NULL    |       |
+-----------+------------+------+-----+---------+-------+
4 rows in set (0.00 sec)

mysql> select * from t_personne;
Empty set (0.39 sec)
```

61.2.7. L'exécution de l'exemple

Exemple :

```
@echo off
REM - script permettant la compilation, l'enrichissement, la creation du schema de
REM - base de données et l'execution du code de test de JDO avec Libelis 1.4.5.
set LIDO_HOME=C:\java\LiDO
set JAVA_HOME=C:\java\j2sdk1.4.2_02

echo initialisation
echo.
SET PATH=%LIDO_HOME%\bin;%JAVA_HOME%\bin

SET CLASSPATH=.;.\mm.mysql-2.0.14-bin.jar
SET CLASSPATH=%CLASSPATH%;%JAVA_HOME%\lib\tools.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\lido-api.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\jdo_1_0_0.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\j2ee.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\bin
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\lido-dev.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\lido-rdb.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\lido-rt.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\lido.tasks
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\lido.tld
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\skinlf.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\connector_1_0_0.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\jta_1_0_1.jar

echo Compilation en cours
javac -classpath %CLASSPATH% testjdo\*.java
echo Compilation effectuee
echo.

echo Enrichissement
java -cp %CLASSPATH% com.libelis.lido.Enhance -metadata metadata.jdo -verbose
echo Enrichissement effectue
echo.

echo DefineSchema en cours
java com.libelis.lido.DefineSchema -properties testjdo.properties -metadata metadata.jdo
echo DefineSchema termine
echo.

echo Execution du test
java -cp %CLASSPATH% testjdo.PersonnePersist
echo Execution terminee
echo.
```

A l'issu de l'exécution, un enregistrement est créé dans la table qui mappe l'objet Personne.

Exemple :

```
 mysql> select * from t_personne;
+--------+---------+------------+---------------------+
| LIDOID | nom     | prenom     | datenaiss           |
+--------+---------+------------+---------------------+
|      1 | mon nom | mon prenom | 2004-01-23 20:06:11 |
+--------+---------+------------+---------------------+
1 row in set (0.00 sec)
```

61.3. L'API JDO

L'API de JDO se compose de deux packages :

- javax.jdo : ce package est à utiliser par les développeurs pour utiliser JDO
- javax.jdo.spi : ce package est à utiliser par les tiers pour développer une implémentation de JDO

Le package javax.jdo contient essentiellement des interfaces ainsi que quelques classes notamment la classe JDOHelper et les diverses exceptions utilisées par JDO. Les interfaces définies sont : Extent, PersistenceManager, PersistenceManagerFactory, Query etTransaction.

Les exceptions définies par l'API JDO sont : JDOCanRetryException, JDODataStoreException, JDOException, JDOFatalDataStoreException, JDOFatalException, JDOFatalInternalException, JDOFatalUserException, JDOUnsupportedOptionException et JDOUserException.

61.3.1. L'interface PersistenceManager

Cette interface définit les méthodes pour l'objet principal de l'API JDO pour les développeurs.

Certaines méthodes permettent de gérer le cycle de vie d'une instance d'un objet de type PersistenceCapable.

Méthode	Rôle
void close()	Fermer
Transaction currentTransaction()	Renvoyer la transaction courante
void deletePersistent()	Permettre de détruire dans la source de données l'instance encapsulée
Extent getExtent(Class, boolean)	Renvoyer une collection d'instance encapsulant les données dans la source de données
void makePersistent()	Permettre de rendre persistantes les données encapsulées dans l'instance en créant une nouvelle occurrence dans le système de gestion de ressources
void evict()	Permettre de préciser que l'instance n'est plus utilisée
Query newQuery()	Renvoyer un objet de type Query qui permet d'effectuer des sélections dans la source de données
void refresh()	Permettre de redonner à une instance les valeurs contenues dans le système de gestion de ressources

Cette interface propose aussi deux méthodes possédant de nombreuses surcharges de la méthode newQuery() pour obtenir une instance d'un objet de type Query.

61.3.2. L'interface PersistenceManagerFactory

Un objet qui implémente cette interface à pour but de fournir une instance d'une classe qui implémente l'interface PersistenceManager. Un tel objet doit être configuré via des propriétés pour instancier un objet de type PersistenceManager. Ces propriétés doivent être fournies à la fabrique avant l'instanciation du premier objet de type PersistenceManager. Il n'est dès lors plus possible de changer la configuration de la fabrique.

Cette interface possède une méthode nommée getPersistenceManager() qui permet d'obtenir une instance de la classe PersistenceManager.

61.3.3. L'interface PersistenceCapable

Cette interface doit être implémentée lors de son enrichissement par la classe qui va contenir des données. A l'origine, la classe n'implémente pas cette interface ; pour assurer la persistance des données, l'implémentation de l'interface et la définition des méthodes de l'interface seront réalisées lors de la phase d'enrichissement.

Cet enrichissement peut se faire de deux façons :

- manuellement : ce qui peut être long et fastidieux
- automatiquement avec un outil fourni avec l'implémentation de JDO : généralement cet outil utilise un fichier au format XML pour obtenir les informations nécessaires à la génération des méthodes

Les méthodes définies dans cette interface sont à l'usage de JDO : une fois la classe enrichie, il ne faut surtout pas appeler directement ces méthodes : elles sont toutes préfixées par jdo.

Une classe qui implémente l'interface PersistenceCapable est nommée instance JDO.

61.3.4. L'interface Query

Cette interface définit des méthodes qui permettent d'obtenir des instances représentant des données issues de la source de données.

L'interface définit plusieurs surcharges de la méthode execute() pour exécuter la requête et renvoyer un ensemble d'instances.

La méthode compile() permet de vérifier la requête et préparer son exécution.

La méthode setFilter() permet de préciser un filtre pour la requête.

Une instance d'un objet implémentant l'interface Query est obtenue en utilisant une des nombreuses surcharges de la méthode newQuery() d'un objet de type PersistenceManager.

61.3.5. L'interface Transaction

Cette interface définit les méthodes pour la gestion des transactions avec JDO.

Elle possède trois méthodes principales qui sont classiques dans la gestion des transactions :

- begin : indique le début d'une transaction
- commit : valide la transaction
- roolback : invalide la transaction et annule toutes les opérations qu'elle contient

61.3.6. L'interface Extent

Une classe qui implémente cette interface permet d'encapsuler toute une collection contenant tous les objets d'un type PersistenceCapable particulier. La méthode iterator() renvoie un objet de type Iterator qui permet de parcourir l'ensemble des éléments de la collection.

L'interface Extent ne prévoit actuellement aucun moyen de filter les éléments de la collection et il est uniquement possible d'obtenir toutes les occurrences.

La méthode close(Iterator) permet de fermer l'objet de type Iterator passé en paramètre.

61.3.7. La classe JDOHelper

La classe JDOHelper permet de faciliter l'utilisation de JDO grâce à plusieurs méthodes statiques pouvant être regroupées dans plusieurs catégories :

- connaitre l'état d'une instance JDO.

Nom	Rôle
boolean isDeleted(Object)	renvoie un booléen qui précise si l'instance JDO fournie en paramètre vient d'être supprimée dans le système de gestion de ressources
boolean isDirty(Object)	renvoie un booléen qui précise si l'instance JDO a été modifiée dans la transaction courante
boolean isNew(Object)	renvoie un booléen qui précise si l'instance JDO fournie en paramètre vient d'être rendue persistante en créant une nouvelle instance dans le système de gestion de ressources
boolean isPersistent(Object)	
boolean isTransactional(Object)	

- obtenir des objets de l'implémentation JDO

Nom	Rôle
PersistenceManager getPersistenceManager(Object)	renvoie l'objet de type PersistenceManager utilisé pour rendre persistante l'instance JDO fournie en paramètre
getObjectId(Object)	
makeDirty(Object, String)	
PersistenceManagerFactory getPersistenceManagerFactory(Properties)	

61.4. La mise en oeuvre

La mise en oeuvre de JDO requiert plusieurs étapes :

- définition d'une classe qui va encapsuler les données (instance JDO)
- définition d'une classe qui va utiliser les données
- compilation des deux classes
- définition d'un fichier de description
- enrichissement de la classe qui va contenir les données

61.4.1. La définition d'une classe qui va encapsuler les données

Une telle classe se présente sous la forme d'un bean : elle représente une occurrence particulière dans le système de stockage des données.

Cette classe n'a pas besoin ni d'utiliser ni d'importer de classes de l'API JDO.

Pour la classe qui va contenir des données, JDO impose la présence d'un constructeur sans argument. Celui-ci est automatiquement ajouté à la compilation si aucun autre constructeur n'est défini, sinon il faut ajouter un constructeur sans argument manuellement.

61.4.2. La définition d'une classe qui va utiliser les données

Cette classe va réaliser des traitements en utilisant JDO pour accéder et/ou mettre à jour des données.

La suite de cette section sera développée dans une version future de ce document

61.4.3. La compilation des classes

Toutes les classes écrites doivent être compilées normalement comme toutes classes Java.

61.4.4. La définition d'un fichier de description

Pour indiquer à JDO quelles classes doivent être persistantes et préciser des informations concernant ces dernières, il faut utiliser un fichier particulier au format XML. Ce fichier désigné par "Metadata" dans les spécifications doit avoir pour extension .jdo.

Il est possible de définir un fichier de description pour chaque classes persistantes ou un fichier pour un package concernant toutes les classes persistantes du package. Dans le premier cas, le fichier doit se nommer nom_de_la_classe.jdo, dans le second nom_du_package.jdo.

Le fichier commence par un prologue :

<?xml version="1.0" encoding="UTF-8" ?>

Le fichier contient ensuite la DTD utilisée pour valider le fichier : soit une URL pointant sur la DTD du site de Sun soit une DTD sur le système de fichier.

<!DOCTYPE jdo PUBLIC "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 1.0//EN" "http://java.sun.com/dtd/jdo_1_0.dtd">

Le tag racine du document XML est <jdo>. Ce tag peut contenir un ou plusieurs tags <package> selon les besoins, chaque tag package concernant un seul package.

Chaque tag <package> contient autant de tags <class> que de classes de type instance JDO utilisées. L'attribut "name", obligatoire, permet de préciser le nom de la classe.

Les tags <jdo>, <package>, <class> et <field> peuvent aussi avoir un tag <extension> qui va contenir des paramètres particuliers dédiés à l'implémentation de JDO utilisée. Il faut un tag <extension> pour chaque implémentation utilisée.

61.4.5. L'enrichissement de la classe qui va contenir les données

Cette phase permet d'ajouter du code à chaque classe encapsulant une instance JDO. Ce code contient les méthodes définies par l'interface PersistenceCapable.

Le ou les outils fournis par le fournisseur sont particuliers pour chaque implémentation utilisée.

61.5. Le parcours de toutes les occurrences

Un objet qui implémente l'interface Extent permet d'accéder à toutes les instances d'une classe encapsulant des données.

Un objet de type Extent est obtenu en appelant la méthode getExtent() d'un objet PersistentManager. Cette méthode attend deux paramètres : un objet de type Class qui est la classe encapsulant les données et un booléen qui permet de préciser si les sous-classes doivent être prises en compte.

Un objet de type Extent ne permet qu'une seule opération sur l'ensemble des instances qu'il contient : obtenir un objet de type Iterator qui permet le parcours séquentiel de toutes les occurrences. La méthode iterator() permet de renvoyer cet objet de type Iterator : les méthodes hasNext() et next() assurent le parcours des occurrences.

L'appel de la méthode close() une fois que l'objet de type Iterator fourni en paramètre n'a plus d'utilité est obligatoire pour permettre de libérer les ressources allouées par l'objet pour son fonctionnement. La méthode closeAll() permet de fermer tout les objets de type Iterator instanciés par l'objet de type Extent.

Exemple : Afficher tous les données de la table personne

```
package testjdo;

import javax.jdo.*;
import java.util.*;

public class PersonneExtent {

  private PersistenceManagerFactory pmf = null;
  private PersistenceManager pm = null;

  public PersonneExtent() {
    try {

      pmf =
        (PersistenceManagerFactory) (Class
          .forName("com.libelis.lido.PersistenceManagerFactory")
          .newInstance());
      pmf.setConnectionDriverName("org.gjt.mm.mysql.Driver");
      pmf.setConnectionURL("jdbc:mysql://localhost/testjdo");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public void afficherTous() {
    pm = pmf.getPersistenceManager();

    Extent personneExtent = pm.getExtent(Personne.class, true);

    Iterator iter = personneExtent.iterator();
    while (iter.hasNext()) {
      Personne personne = (Personne) iter.next();
      System.out.println(personne.getNom() + " " + personne.getPrenom());
    }
    personneExtent.close(iter);
  }

  public static void main(String args[]) {
    PersonneExtent pe = new PersonneExtent();
    pe.afficherTous();
  }
}
```

Exemple : Contenu de la table au moment de l'exécution

```
C:\mysql\bin>mysql testjdo
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3 to server version: 4.0.16-nt
```

```
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql< select * from t_personne;
+--------+---------+------------+---------------------+
| LIDOID | nom     | prenom     | datenaiss           |
+--------+---------+------------+---------------------+
|      1 | mon nom | mon prenom | 2004-01-23 20:06:11 |
|   1025 | Nom1    | Jean       | 2004-02-10 00:20:39 |
|   2049 | Nom4    | Jean       | 2004-02-10 00:25:09 |
|   3073 | Nom3    | Louis      | 2004-02-10 21:36:32 |
+--------+---------+------------+---------------------+
4 rows in set (0.00 sec)

mysql<
```

Résultat :

```
Execution du test
mon nom mon prenom
Nom1 Jean
Nom4 Jean
Nom3 Louis
Execution terminee
```

61.6. La mise en oeuvre de requêtes

Avec JDO, les requêtes sont mises en oeuvre grâce à un objet de type Query. Les requêtes appliquent un filtre sur un ensemble d'objets encapsulant des données. Ces données sont encapsulées dans un objet de type Extent ou une collection.

Un filtre est une expression booléenne appliquée à chacune des occurrences : la requête renvoie toutes les occurrences pour lesquelles le résultat de l'évaluation de l'expression est vrai. Les expressions sont exprimées avec un langage particulier nommé JDO Query Langage (JDOQL)

Une instance d'un objet qui implémente l'interface Query est obtenue en utilisant la méthode newQuery() d'un objet de type PersistanceManager.

Exemple : afficher les occurrences dont le prénom est Jean

```java
package testjdo;

import javax.jdo.*;
import java.util.*;

public class PersonneQuery {

  private PersistenceManagerFactory pmf = null;
  private PersistenceManager pm = null;

  public PersonneQuery() {
    try {

      pmf =
        (PersistenceManagerFactory) (Class
          .forName("com.libelis.lido.PersistenceManagerFactory")
          .newInstance());
      pmf.setConnectionDriverName("org.gjt.mm.mysql.Driver");
      pmf.setConnectionURL("jdbc:mysql://localhost/testjdo");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public void filtrer() {
    pm = pmf.getPersistenceManager();

    Extent personneExtent = pm.getExtent(Personne.class, true);
    String filtre = "prenom == \"Jean\"";

    Query query = pm.newQuery(personneExtent, filtre);
    query.setOrdering("nom ascending, prenom ascending");
    Collection result = (Collection) query.execute();

    Iterator iter = result.iterator();
    while (iter.hasNext()) {
      Personne personne = (Personne) iter.next();
      System.out.println(personne.getNom() + " " + personne.getPrenom());
    }
    query.close(result);
  }

  public static void main(String args[]) {
    PersonneQuery pq = new PersonneQuery();
    pq.filtrer();
  }
}
```

Résultat :

```
Execution du test
Nom1 Jean
Nom4 Jean
Execution terminee
```

La suite de ce chapitre sera développée dans une version future de ce document

62. Hibernate

Chapitre 62

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Hibernate est une solution open source de type ORM (Object Relational Mapping) qui permet de faciliter le développement de la couche persistance d'une application. Hibernate permet donc de représenter une base de données en objets Java et vice versa.

Hibernate facilite la persistence et la recherche de données dans une base de données en réalisant lui-même la création des objets et les traitements de remplissage de ceux-ci en accédant à la base de données. La quantité de code ainsi épargnée est très importante d'autant que ce code est généralement fastidieux et redondant.

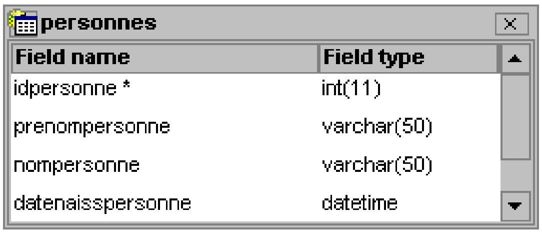
Hibernate est très populaire notamment à cause de ses bonnes performances et de son ouverture à de nombreuses bases de données.

Les bases de données supportées sont les principales du marché : DB2, Oracle, MySQL, PostgreSQL, Sybase, SQL Server, Sap DB, Interbase, ...

Le site officiel https://www.hibernate.org contient beaucoup d'informations sur l'outil et propose de le télécharger ainsi que sa documentation.

La version utilisée dans cette section est la 2.1.2 : il faut donc télécharger le fichier hibernate-2.1.2.zip et le décompresser dans un répertoire du système.

Ce chapitre va utiliser Hibernate avec une base de données de type MySQL possédant une table nommée "personnes".



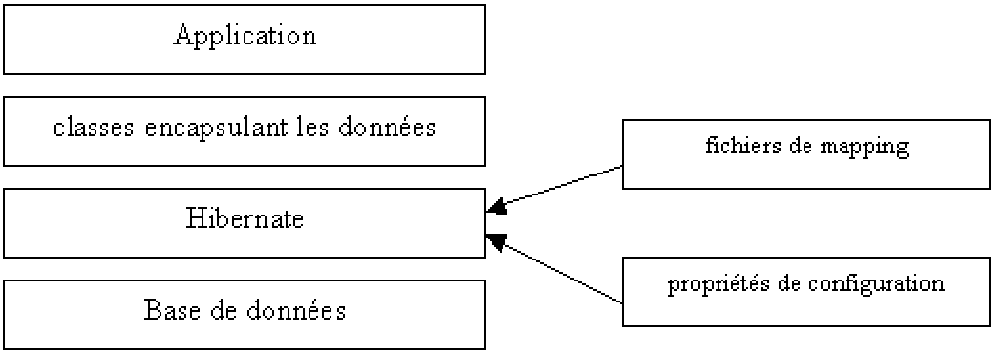
personnes

Field name	Field type
idpersonne *	int(11)
prenompersonne	varchar(50)
nompersonne	varchar(50)
datenaisspersonne	datetime

Hibernate a besoin de plusieurs éléments pour fonctionner :

- une classe de type javabean qui encapsule les données d'une occurrence d'une table
- un fichier de configuration qui assure la correspondance entre la classe et la table (mapping)
- des propriétés de configuration notamment des informations concernant la connexion à la base de données

Une fois ces éléments correctement définis, il est possible d'utiliser Hibernate dans le code des traitements à réaliser. L'architecture d'Hibernate est donc la suivante :



Ce chapitre survole uniquement les principales fonctionnalités d'Hibernate qui est un outil vraiment complet : pour de plus amples informations, il est nécessaire de consulter la documentation officielle fournie avec l'outil ou consultable sur le site web.

Ce chapitre contient plusieurs sections :

- La création d'une classe qui va encapsuler les données
- La création d'un fichier de correspondance
- Les propriétés de configuration
- L'utilisation d'Hibernate
- La persistance d'une nouvelle occurrence
- L'obtention d'une occurrence à partir de son identifiant
- L'obtention de données
- La mise à jour d'une occurrence
- La suppression d'une ou plusieurs occurrences
- Les relations
- Le mapping de l'héritage de classes
- Les caches d'Hibernate
- Les outils de génération de code

62.1. La création d'une classe qui va encapsuler les données

Cette classe doit respecter le standard des Javabeans, notamment, encapsuler les propriétés dans ses champs private avec des getters et setters et avoir un constructeur par défaut.

Les types utilisables pour les propriétés sont : les types primitifs, les classes String et Dates, les wrappers, et n'importe quelle classe qui encapsule une autre table ou une partie de la table.

Exemple :

```
import java.util.Date;

public class Personnes {

  private Integer idPersonne;
  private String nomPersonne;
  private String prenomPersonne;
  private Date datenaissPersonne;

  public Personnes(String nomPersonne, String prenomPersonne, Date datenaissPersonne) {
    this.nomPersonne = nomPersonne;
    this.prenomPersonne = prenomPersonne;
    this.datenaissPersonne = datenaissPersonne;
  }

  public Personnes() {
  }

  public Date getDatenaissPersonne() {
    return datenaissPersonne;
  }
  public Integer getIdPersonne() {
    return idPersonne;
  }

  public String getNomPersonne() {
    return nomPersonne;
  }

  public String getPrenomPersonne() {
    return prenomPersonne;
  }

  public void setDatenaissPersonne(Date date) {
    datenaissPersonne = date;
  }

  public void setIdPersonne(Integer integer) {
    idPersonne = integer;
  }

  public void setNomPersonne(String string) {
    nomPersonne = string;
  }

  public void setPrenomPersonne(String string) {
    prenomPersonne = string;
  }

}
```

62.2. La création d'un fichier de correspondance

Pour assurer le mapping, Hibernate a besoin d'un fichier de correspondance (mapping file) au format XML qui va contenir des informations sur la correspondance entre la classe définie et la table de la base de données.

Même si cela est possible, il n'est pas recommandé de définir un fichier de mapping pour plusieurs classes. Le plus simple est de définir un fichier de mapping par classe, nommé du nom de la classe suivi par ".hbm.xml". Ce fichier doit être situé dans le même répertoire que la classe correspondante ou dans la même archive pour les applications packagées.

Différents éléments sont précisés dans ce document XML :

- la classe qui va encapsuler les données
- l'identifiant dans la base de données et son mode de génération
- le mapping entre les propriétés de classe et les champs de la base de données
- les relations
- ...

Le fichier débute par un prologue et une définition de la DTD utilisée par le fichier XML.

Exemple :

```
<?xml version="1.0"?>
  <!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
```

Le tag racine du document XML est le tag <hibernate-mapping>. Ce tag peut contenir un ou plusieurs tag <class> : il est cependant préférable de n'utiliser qu'un seul tag <class> et de définir autant de fichiers de correspondance que de classes.

Exemple :

Exemple :

```
<?xml version="1.0"?><!DOCTYPE hibernate-mapping
```

```
  PUBLIC "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd"><hibernate-mapping>
  <class name="Personnes" table="personnes">
    <id name="idPersonne" type="int" column="idpersonne">
      <generator class="native"/>
    </id>
    <property name="nomPersonne" type="string" not-null="true" />
    <property name="prenomPersonne" type="string" not-null="true" />
    <property name="datenaissPersonne" type="date">
      <meta attribute="field-description">date de naissance</meta>
    </property>
  </class>
</hibernate-mapping>
```

Le tag <class> permet de préciser des informations sur la classe qui va encapsuler les données.

Ce tag possède plusieurs attributs dont les principaux sont:

Nom	Obligatoire	Rôle
name	oui	nom pleinement qualifié de la classe
table	oui	nom de la table dans la base de données
dynamic-update	non	booléen qui indique de ne mettre à jour que les champs dont la valeur a été modifiée (false par défaut)
dynamic-insert	non	booléen qui indique de ne générer un ordre insert que pour les champs dont la valeur est non nulle (false par défaut)
mutable	non	booléen qui indique si les occurrences peuvent être mises à jour (true par défaut)

Le tag enfant <id> du tag <class> permet de fournir des informations sur l'identifiant d'une occurrence dans la table.

Ce tag possède plusieurs attributs :

Nom	Obligatoire	Rôle
name	non	nom de la propriété dans la classe
type	non	le type Hibernate
column	non	le nom du champ dans la base de données (par défaut le nom de la propriété)
unsaved-value	non	permet de préciser la valeur de l'identifiant pour une instance non encore enregistrée dans la base de données. Les valeurs possibles sont : any, none, null ou une valeur fournie. Null est la valeur par défaut.

Le tag <generator>, fils obligatoire du tag <id>, permet de préciser quel est le mode de génération d'un nouvel identifiant.

Ce tag possède un attribut :

Attribut	Obligatoire	Rôle
class	oui	précise la classe qui va assurer la génération de la valeur d'un nouvel identifiant. Il existe plusieurs classes fournies en standard par Hibernate qui possèdent un nom utilisable comme valeur de cet attribut.

Les classes de génération fournies en standard par Hibernate possèdent chacun un nom :

Nom	Rôle

increment	incrémentation d'une valeur dans la JVM
identity	utilisation d'un identifiant auto-incrémenté pour les bases de données qui le supportent (DB2, MySQL, SQL Server, ...)
sequence	utilisation d'une séquence pour les bases de données qui le supportent (Oracle, DB2, PostgreSQL, ...)
hilo	utilisation d'un algorithme qui utilise une valeur réservée pour une table d'une base de données (par exemple une table qui stocke la valeur du prochain identifiant pour chaque table)
seqhilo	idem mais avec un mécanisme proche d'une séquence
uuid.hex	utilisation d'un algorithme générant un identifiant de type UUID sur 32 caractères prenant en compte entre autres l'adresse IP de la machine et l'heure du système
uuid.string	idem générant un identifiant de type UUID sur 16 caractères
native	utilise la meilleure solution proposée par la base de données
assigned	la valeur est fournie par l'application
foreign	la valeur est fournie par un autre objet avec lequel la classe est associée

Certains modes de génération nécessitent des paramètres : dans ce cas, il faut les définir en utilisant un tag fils <param> pour chaque paramètre.

Le tag <property>, fils du tag <class>, permet de fournir des informations sur une propriété et sa correspondance avec un champ dans la base de données.

Ce tag possède plusieurs attributs dont les principaux sont :

Nom	Obligatoire	Rôle
name	oui	précise le nom de la propriété
type	non	précise le type
column	non	précise le nom du champ dans la base de données (par défaut le nom de la propriété)
update	non	précise si le champ est mis à jour lors d'une opération SQL de type update (par défaut true)
insert	non	précise si le champ est mis à jour lors d'une opération SQL de type insert (par défaut true)

Le type doit être soit un type Hibernate (integer, string, date, timestamp, ...), soit les types primitif Java ou de certaines classes de base (int, java.lang.String, float, java.util.Date, ...), soit une classe qui encapsule des données à rendre persistantes.

Le fichier de correspondance peut aussi contenir une description des relations qui existent avec la table dans la base de données.

62.3. Les propriétés de configuration

Pour exécuter Hibernate, il faut lui fournir un certain nombre de propriétés concernant sa configuration pour qu'il puisse se connecter à la base de données.

Ces propriétés peuvent être fournies sous plusieurs formes :

- un fichier de configuration nommé hibernate.properties et stocké dans un répertoire inclus dans le classpath
- un fichier de configuration au format XML nommé hibernate.cfg.xml
- utiliser la méthode setProperties() de la classe Configuration
- définir des propriétés dans la JVM en utilisant l'option -Dpropriété=valeur

Les principales propriétés pour configurer la connexion JDBC sont :

Nom de la propriété	Rôle
hibernate.connection.driver_class	nom pleinement qualifié de la classe du pilote JDBC
hibernate.connection.url	URL JDBC désignant la base de données
hibernate.connection.username	nom de l'utilisateur pour la connexion
hibernate.connection.password	mot de passe de l'utilisateur
hibernate.connection.pool_size	nombre maximum de connexions dans le pool

Les principales propriétés à utiliser pour configurer une source de données (DataSource) sont :

Nom de la propriété	Rôle
hibernate.connection.datasource	nom du DataSource enregistré dans JNDI
hibernate.jndi.url	URL du fournisseur JNDI
hibernate.jndi.class	classe pleinement qualifiée de type InitialContextFactory permettant l'accès à JNDI
hibernate.connection.username	nom de l'utilisateur de la base de données
hibernate.connection.password	mot de passe de l'utilisateur

Les principales autres propriétés sont :

Nom de la propriété	Rôle
hibernate.dialect	nom de la classe pleinement qualifié qui assure le dialogue avec la base de données
hibernate.jdbc.use_scrollable_resultset	booléen qui permet le parcours dans les deux sens pour les connexions fournies à Hibernate utilisant des pilotes JDBC 2 supportant cette fonctionnalité
hibernate.show_sql	booléen qui précise si les requêtes SQL générées par Hibernate sont affichées dans la console (particulièrement utile lors du débogage)

Hibernate propose des classes qui héritent de la classe Dialect pour chaque base de données supportée. C'est le nom de la classe correspondant à la base de données utilisée qui doit être obligatoirement fourni à la propriété hibernate.dialect.

Pour définir les propriétés utiles, le plus simple est de définir un fichier de configuration qui en standard doit se nommer hibernate.properties. Ce fichier contient une paire clé=valeur pour chaque propriété définie.

Exemple : paramètres pour utiliser une base de données MySQL

```
hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.connection.driver_class=com.mysql.jdbc.Driver
hibernate.connection.url=jdbc:mysql://localhost/testDB
hibernate.connection.username=root
hibernate.connection.password=
```

Le pilote de la base de données utilisée, mysql-connector-java-3.0.11-stable-bin.jar dans l'exemple, doit être ajouté dans le classpath.

Il est aussi possible de définir les propriétés dans un fichier au format XML nommé en standard hibernate.cfg.xml

Les propriétés sont alors définies par un tag <property>. Le nom de la propriété est définie grâce à l'attribut « name » et sa valeur est fournie dans le corps du tag.

Exemple :

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 2.0//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost/testDB</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password"></property>
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="show_sql">true</property>
    <mapping resource="Personnes.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

62.4. L'utilisation d'Hibernate

Pour utiliser Hibernate dans le code, il est nécessaire de réaliser plusieurs opérations :

- création d'une instance de la classe
- création d'une instance de la classe SessionFactory
- création d'une instance de la classe Session qui va permettre d'utiliser les services d'Hibernate

Si les propriétés sont définies dans le fichier hibernate.properties, il faut tout d'abord créer une instance de la classe Configuration. Pour lui associer la ou les classes encapsulant les données, la classe propose deux méthodes :

- addFile() qui attend en paramètre le nom du fichier de mapping
- addClass() qui attend en paramètre un objet de type Class encapsulant la classe. Dans ce cas, la méthode va rechercher un fichier nommé nom_de_la_classe.hbm.xml dans le classpath (ce fichier doit se situer dans le même répertoire que le fichier .class de la classe correspondante)

Une instance de la classe Session est obtenue à partir d'une fabrique de type SessionFactory, elle-même obtenue à partir de l'instance du type Configuration en utilisant la méthode buildSessionFactory().

La méthode openSession() de la classe SessionFactory permet d'obtenir une instance de la classe Session.

Par défaut, c'est la méthode openSession() qui va ouvrir une connexion vers la base de données en utilisant les informations fournies par les propriétés de configuration.

Exemple :

```
import org.hibernate.*;
import org.hibernate.cfg.Configuration;
import java.util.Date;

public class TestHibernate1 {

  public static void main(String args[]) throws Exception {
    Configuration config = new Configuration();
    config.addClass(Personnes.class);
    SessionFactory sessionFactory = config.buildSessionFactory();
    Session session = sessionFactory.openSession();
...
```

Il est aussi possible de fournir en paramètre de la méthode openSession() une instance de la classe javax.sql.Connection qui encapsule la connexion à la base de données.

Pour une utilisation du fichier hibernate.cfg.xml, il faut créer une occurrence de la classe Configuration, appeler sa méthode configure() qui va lire le fichier XML et appeler la méthode buildSessionFactory() de l'objet renvoyé par la méthode configure().

Exemple :

```
import org.hibernate.*;
import org.hibernate.cfg.Configuration;
import java.util.*;

public class TestHibernate1 {
  public static void main(String args[]) throws Exception {
    SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
...
```

Il est important de clôturer l'objet Session, une fois que celui-ci est devenu inutile, en utilisant la méthode close().

62.5. La persistance d'une nouvelle occurrence

Pour créer une nouvelle occurrence dans la source de données, il suffit de créer une nouvelle instance de la classe encapsulant les données, de valoriser ses propriétés et d'appeler la méthode save() de la session en lui passant en paramètre l'objet encapsulant les données.

La méthode save() n'a aucune action directe sur la base de données. Pour enregistrer les données dans la base, il faut réaliser un commit sur la connexion ou la transaction ou faire appel à la méthode flush() de la classe Session.

Exemple :

```
import org.hibernate.*;
import org.hibernate.cfg.Configuration;
import java.util.Date;

public class TestHibernate1 {

  public static void main(String args[]) throws Exception {
    Configuration config = new Configuration();
    config.addClass(Personnes.class);
    SessionFactory sessionFactory = config.buildSessionFactory();
    Session session = sessionFactory.openSession();

    Transaction tx = null;
    try {
      tx = session.beginTransaction();
      Personnes personne = new Personnes("nom3", "prenom3", new Date());
      session.save(personne);
      session.flush() ;
      tx.commit();
    } catch (Exception e) {
      if (tx != null) {
        tx.rollback();
      }
      throw e;
    } finally {
      session.close();
    }

    sessionFactory.close();
  }
}
```

Résultat :

```
C:\java\test\testhibernate>ant TestHibernate1
Buildfile: build.xml
init:
[copy] Copying 1 file to C:\java\test\testhibernate\bin
```

```
compile:
TestHibernate1:
[java] 12:41:37,402 INFO Environment:462 - Hibernate 2.1.2
[java] 12:41:37,422 INFO Environment:496 - loaded properties from resource
hibernate.properties: {hibernate.connection.username=root, hibernate.connection
.password=, hibernate.cglib.use_reflection_optimizer=true, hibernate.dialect=net
.sf.hibernate.dialect.MySQLDialect, hibernate.connection.url=jdbc:mysql://localh
ost/testDB, hibernate.connection.driver_class=com.mysql.jdbc.Driver}
[java] 12:41:37,432 INFO Environment:519 - using CGLIB reflection optimizer
[java] 12:41:37,502 INFO Configuration:329 - Mapping resource: Personnes.hbm.xml
[java] 12:41:38,784 INFO Binder:229 - Mapping class: Personnes -> personnes
[java] 12:41:38,984 INFO Configuration:595 - processing one-to-many association mappings
[java] 12:41:38,994 INFO Configuration:604 - processing one-to-one association property
references
[java] 12:41:38,994 INFO Configuration:629 - processing foreign key constraints
[java] 12:41:39,074 INFO Dialect:82 - Using dialect: org.hibernate.dialect.MySQLDialect
[java] 12:41:39,084 INFO SettingsFactory:62 - Use outer join fetching: true
[java] 12:41:39,104 INFO DriverManagerConnectionProvider:41 - Using Hibernate
built-in connection pool (not for production use!)
[java] 12:41:39,114 INFO DriverManagerConnectionProvider:42 - Hibernate co
nnection pool size: 20
[java] 12:41:39,144 INFO DriverManagerConnectionProvider:71 - using driver
: com.mysql.jdbc.Driver at URL: jdbc:mysql://localhost/testDB
[java] 12:41:39,154 INFO DriverManagerConnectionProvider:72 - connection p
roperties: {user=root, password=}
[java] 12:41:39,185 INFO TransactionManagerLookupFactory:33 - No Transacti
onManagerLookup configured (in JTA environment, use of process level read-write
cache is not recommended)
[java] 12:41:39,625 INFO SettingsFactory:102 - Use scrollable result sets:true
[java] 12:41:39,635 INFO SettingsFactory:105 - Use JDBC3 getGeneratedKeys(): true
[java] 12:41:39,635 INFO SettingsFactory:108 - Optimize cache for minimal puts: false
[java] 12:41:39,635 INFO SettingsFactory:117 - Query language substitutions: {}
[java] 12:41:39,645 INFO SettingsFactory:128 - cache provider: org.ehcache.hibernate.
Provider
[java] 12:41:39,685 INFO Configuration:1080 - instantiating and configuring caches
[java] 12:41:39,946 INFO SessionFactoryImpl:119 - building session factory
[java] 12:41:41,237 INFO SessionFactoryObjectFactory:82 - no JNDI name configured
[java] 12:41:41,768 INFO SessionFactoryImpl:531 - closing
[java] 12:41:41,768 INFO DriverManagerConnectionProvider:137 - cleaning up
connection pool: jdbc:mysql://localhost/testDB
BUILD SUCCESSFUL
Total time: 7 seconds
C:\java\test\testhibernate>
```

62.6. L'obtention d'une occurrence à partir de son identifiant

La méthode load() de la classe Session permet d'obtenir une instance de la classe encapsulant les données de l'occurrence de la base dont l'identifiant est fourni en paramètre.

Il existe deux surcharges de la méthode :

- la première attend en premier paramètre le type de la classe des données et renvoie une nouvelle instance de cette classe
- la seconde attend en paramètre une instance de la classe des données et la met à jour avec les données retrouvées

Exemple :

```
import org.hibernate.*;
import org.hibernate.cfg.Configuration;

public class TestHibernate2 {

  public static void main(String args[]) throws Exception {
    Configuration config = new Configuration();
    config.addClass(Personnes.class);
    SessionFactory sessionFactory = config.buildSessionFactory();
    Session session = sessionFactory.openSession();

    try {
      Personnes personne = (Personnes) session.load(Personnes.class, new Integer(3));
      System.out.println("nom = " + personne.getNomPersonne());
    } finally {
      session.close();
    }

    sessionFactory.close();
  }
}
```

Résultat :

```
C:\java\test\testhibernate>ant TestHibernate2
Buildfile: build.xml

init:

compile:
    [javac] Compiling 1 source file to C:\java\test\testhibernate\bin

TestHibernate2:
     [java] nom = nom3

BUILD SUCCESSFUL
Total time: 9 seconds
```

62.7. L'obtention de données

Hibernate utilise plusieurs moyens pour obtenir des données de la base de données :

- Hibernate Query Language (HQL)
- API Criteria : Query By Criteria (QBC) et Query BY Example (QBE)
- Requêtes SQL natives

62.7.1. Le langage de requête HQL

Pour offrir un langage d'interrogation commun à toutes les bases de données, Hibernate propose son propre langage nommé HQL (Hibernate Query Language).

L'intérêt de HQL est d'être indépendant de la base de données sous-jacente : la requête SQL sera générée par Hibernate à partir du HQL en fonction de la base de données précisée via un dialect.

Hibernate Query Language (HQL) est un langage de requêtes orienté objets qui permet de représenter des requêtes SQL : les entités utilisées dans les requêtes HQL sont des objets et des propriétés. La syntaxe de HQL et ses fonctionnalités de base sont très similaire à SQL.

Il est aussi possible d'utiliser l'API Criteria qui va en interne exécuter une requête HQL. Le point d'entrée est d'obtenir une instance de type Criteria en invoquant la méthode createCriteria() de la session Hibernate courante : elle attend en paramètre la classe des objets attendus en résultat. L'API permet de préciser les différents critères qui seront utilisés pour générer la requête. L'API Criteria utilise HQL en sous-jacent.

Hibernate propose aussi un support pour exécuter des requêtes natives. Ceci permet d'utiliser des fonctionnalités de la base de données sous-jacente qui ne soient pas supportées par HQL. Cependant dans ce cas, le support multi-base de données offert par HQL sera probablement compromis.

Le langage HQL est proche de SQL avec une utilisation sous forme d'objets des noms de certaines entités : il n'y a aucune référence aux tables ou aux champs car ceux-ci sont référencés respectivement par leur classe et leurs propriétés. C'est Hibernate qui se charge de générer la requête SQL à partir de la requête HQL en tenant compte du contexte (type de base de données utilisée défini dans le fichier de configuration et la configuration du mapping).

62.7.1.1. La syntaxe de HQL

HQL possède une syntaxe similaire de celle de SQL : la différence majeure est que HQL utilise des objets et leurs propriétés alors que SQL utilise des tables et leurs colonnes

Exception faite des noms de classes et de variables, les requêtes HQL ne sont pas sensibles à la casse. Généralement les mots clé HQL sont en minuscule pour faciliter leur lecture.

Une requête HQL peut être composée :

- de clauses
- de fonctions d'agrégation
- de sous requêtes

Les clauses sont les mots clés HQL qui sont utilisés pour définir la requête :

Clause	Description	Syntaxe	Exemple
from	précise la classe d'objets dont les occurrences doivent être retrouvées. Il est possible de définir un alias pour un objet en utilisant le mot clé alias	from object [as objectalias]	from Personne as pers (retourne toutes les occurrences de type Personne)
select	précise les propriétés à renvoyer. Doit être utilisé avec une clause from		select pers.nom from Personne as pers (retourne le nom de toutes les personnes)
where	précise une condition qui permet de filtrer les occurrences retournées. Doit être utilisé avec une clause select et/ou from	where condition	from Personne as pers where pers.nom = "Dupond" (retourne toutes les personnes dont le nom est Dupond.
order by	précise un ordre de tri sur une ou plusieurs propriétés. L'ordre par défaut est ascendant	order by propriete [asc|desc] [, propriete] ...;	select pers.nom, pers.prenom from Personne as pers order by pers.nom asc, pers.prenom desc
group by	précise un critère de regroupement pour les résultats retournés. Doit être utilisé avec une clause select et/ou from	group by propriete [, propriete] ...	

Les fonctions d'agrégation HQL ont un rôle similaire à celles de SQL : elles permettent de calculer des valeurs agrégeant des valeurs de propriétés issues du résultat de la requête.

Fonction	Syntaxe
count	count([distinct|all|*] object | object.property)
sum	sum([distinct|all] object.property)
avg	avg([distinct|all] object.property)
max	max([distinct|all] object.property)
min	min([distinct|all] object.property)

Résultat :

```
select avg(emp.salaire) from Employe as emp
```

Les sous requêtes sont des requêtes imbriquées dans une autre requête

L'utilisation de sous requêtes dans HQL est conditionné par le support des sous requêtes par la base de données sous-jacente. Les sous requêtes sont entourées par des parenthèses : elles sont exécutées avant la requête principale puisque celle-ci a besoin des résultats pour son exécution.

Résultat :

```
from Employe as emp where emp.salaire >= (select avg(Employe.salaire)

from Employe)
```

62.7.1.2. La mise en oeuvre de HQL

La mise en oeuvre de HQL peut se faire de plusieurs manières.

Le plus courant est d'obtenir une instance de la classe Query en invoquant la méthode createQuery() de la session Hibernate courante : elle attend en paramètre la requête HQL qui devra être exécutée.

Exemple :

```
Session session;
Query query = session.createQuery("select pers.nom from Personne as pers");
List result = query.list();
```

La méthode list() de la classe Query permet d'obtenir une collection qui contient les résultats de l'exécution de la requête.

Il est également possible de définir des requêtes utilisant des paramètres nommés grâce à un objet implémentant l'interface Query. Dans ces requêtes, les paramètres sont précisés avec un caractère « : » suivi d'un nom unique.

L'interface Query propose de nombreuses méthodes setXXX() pour associer à chaque paramètre une valeur en fonction du type de la valeur (XXX représente le type). Chacune de ces méthodes possède deux surcharges permettant de préciser le paramètre (à partir de son nom ou de son index dans la requête) et sa valeur.

Pour parcourir la collection des occurrences trouvées, l'interface Query propose la méthode list() qui renvoie une collection de type List ou la méthode iterate() qui renvoie un itérateur sur la collection.

Exemple :

```
import org.hibernate.*;
import org.hibernate.cfg.Configuration;
import java.util.*;

public class TestHibernate8 {

  public static void main(String args[]) throws Exception {
    SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
    Session session = sessionFactory.openSession();

    try {
      Query query = session.createQuery("from Personnes p where p.nomPersonne = :nom");
      query.setString("nom", "nom2");
      Iterator personnes = query.iterate();

      while (personnes.hasNext()) {
        Personnes personne = (Personnes) personnes.next();
        System.out.println("nom = " + personne.getNomPersonne());
      }
    } finally {
      session.close();
    }

    sessionFactory.close();
  }
}
```

Résultat :

```
C:\java\test\testhibernate>ant TestHibernate8
```

```
Buildfile: build.xml

init:
     [copy] Copying 1 file to C:\java\test\testhibernate\bin

compile:

TestHibernate8:
     [java] nom = nom2

BUILD SUCCESSFUL
Total time: 7 seconds
```

La méthode find() de la classe Session permet d'effectuer une recherche d'occurrences grâce à la requête fournie en paramètre.

Exemple : rechercher toutes les occurrences d'une table

```
import org.hibernate.*;
import org.hibernate.cfg.Configuration;
import java.util.*;

public class TestHibernate3 {

  public static void main(String args[]) throws Exception {
    Configuration config = new Configuration();
    config.addClass(Personnes.class);
    SessionFactory sessionFactory = config.buildSessionFactory();
    Session session = sessionFactory.openSession();

    try {
      List personnes = session.find("from Personnes");
      for (int i = 0; i < personnes.size(); i++) {
        Personnes personne = (Personnes) personnes.get(i);
        System.out.println("nom = " + personne.getNomPersonne());
      }
    } finally {
      session.close();
    }
    sessionFactory.close();
  }
}
```

Résultat :

```
C:\java\test\testhibernate>ant TestHibernate3
Buildfile: build.xml

init:

compile:
    [javac] Compiling 1 source file to C:\java\test\testhibernate\bin

TestHibernate3:
     [java] nom = nom1
     [java] nom = nom2
     [java] nom = nom3

BUILD SUCCESSFUL
Total time: 14 seconds
```

La méthode find() possède deux surcharges pour permettre de fournir un seul ou plusieurs paramètres dans la requête.

La première surcharge permet de fournir un seul paramètre : elle attend en paramètre la requête, la valeur du paramètre et le type du paramètre.

Exemple :

```
import org.hibernate.*;
import org.hibernate.cfg.Configuration;
import java.util.*;

public class TestHibernate4 {

  public static void main(String args[]) throws Exception {
    Configuration config = new Configuration();
    config.addClass(Personnes.class);
    SessionFactory sessionFactory = config.buildSessionFactory();
    Session session = sessionFactory.openSession();

    try {
      List personnes = session.find("from Personnes p where p.nomPersonne=?",
        "nom1", Hibernate.STRING);
      for (int i = 0; i < personnes.size(); i++) {
        Personnes personne = (Personnes) personnes.get(i);
        System.out.println("nom = " + personne.getNomPersonne());
      }
    } finally {
      session.close();
    }

    sessionFactory.close();
  }
}
```

Résultat :

```
C:\java\test\testhibernate>ant TestHibernate4
Buildfile: build.xml

init:

compile:
    [javac] Compiling 1 source file to C:\java\test\testhibernate\bin

TestHibernate4:
     [java] nom = nom1

BUILD SUCCESSFUL
```

Dans la requête du précédent exemple, un alias nommé « p » est défini pour la classe Personnes. Le mode de fonctionnement d'un alias est similaire en HQL et en SQL.

La classe Session propose une méthode iterate() dont le mode de fonctionnement est similaire à la méthode find() mais elle renvoie un itérateur (objet de type Iterator) sur la collection des éléments retrouvés plutôt que la collection elle même.

Exemple :

```
import org.hibernate.*;
import org.hibernate.cfg.Configuration;
import java.util.*;

public class TestHibernate6 {

  public static void main(String args[]) throws Exception {
    Configuration config = new Configuration();
    config.addClass(Personnes.class);
    SessionFactory sessionFactory = config.buildSessionFactory();
    Session session = sessionFactory.openSession();

    try {
      Iterator personnes = session.iterate("from Personnes ");
      while (personnes.hasNext()) {
        Personnes personne = (Personnes) personnes.next();
        System.out.println("nom = " + personne.getNomPersonne());
      }
    } finally {
      session.close();
    }

    sessionFactory.close();
  }
}
```

Résultat :

```
C:\java\test\testhibernate>ant TestHibernate6
Buildfile: build.xml

init:

compile:
    [javac] Compiling 1 source file to C:\java\test\testhibernate\bin

TestHibernate6:
     [java] nom = nom1
     [java] nom = nom2
     [java] nom = nom3

BUILD SUCCESSFUL
```

Il est aussi possible d'utiliser la clause « order by » dans une requête HQL pour définir l'ordre de tri des occurrences.

Exemple :

List personnes = session.find("from Personnes p order by p.nomPersonne desc");

Il est possible d'utiliser des fonctions telles que count() pour compter le nombre d'occurrences.

Exemple :

```
import org.hibernate.*;
import org.hibernate.cfg.Configuration;
import java.util.*;

public class TestHibernate5 {

  public static void main(String args[]) throws Exception {
    Configuration config = new Configuration();
    config.addClass(Personnes.class);
    SessionFactory sessionFactory = config.buildSessionFactory();
    Session session = sessionFactory.openSession();

    try {
      int compteur = ( (Integer) session.iterate(
        "select count(*) from Personnes").next() ).intValue();
      System.out.println("compteur = " + compteur);
    } finally {
      session.close();
    }

    sessionFactory.close();
  }
}
```

Résultat :

```
C:\java\test\testhibernate>ant TestHibernate5
Buildfile: build.xml

init:

compile:
    [javac] Compiling 1 source file to C:\java\test\testhibernate\bin

TestHibernate5:
     [java] compteur = 3

BUILD SUCCESSFUL
```

Hibernate propose également d'externaliser une requête dans le fichier de mapping.

62.7.2. L'API Criteria

En plus du langage HQL pour réaliser des requêtes d'extraction de données, Hibernate propose une API qui permet de construire des requêtes pour interroger la base de données. L'API Criteria d'Hibernate propose donc une alternative à HQL sous la forme d'une API.

Le HQL possède une syntaxe dérivée de celle de SQL dans laquelle les notions relationnelles sont remplacées par des notions objets. Ceci oblige les développeurs à utiliser une syntaxe proche de celle du SQL.

L'API Criteria Query propose des objets pour définir les critères d'une requête ce qui permet aux développeurs de les définir d'une manière orientée objet plutôt que d'utiliser le HQL.

L'API Criteria propose donc d'avoir une approche orientée objet pour définir des requêtes et obtenir des données. L'utilisation de cette API permet d'avoir un meilleur contrôle grâce à la compilation.

Cette API permet de facilement combiner de nombreux critères optionnels pour créer une requête : elle est particulièrement adaptée pour créer dynamiquement des requêtes à la volée comme c'est le cas par exemple pour des requêtes effectuant des recherches multicritères à partir d'informations fournies par l'utilisateur.

Elle offre pour la plupart des fonctionnalités une approche bi directionnelle :

- appliquer un critère en désignant la propriété sur laquelle il s'applique
- appliquer un critère sur une propriété

Elle propose des classes et des interfaces qui encapsulent les fonctionnalités de SQL dont les principales sont :

- Criteria
- Criterion
- Restrictions
- Projection
- Order

L'interface org.hibernate.criterion.Criteria est le point d'entrée pour utiliser l'API Criteria. Elle permet de définir une requête à partir de critères pour retrouver des données.

Exemple :

```
select * from Personne
```

Voici le code équivalent avec l'API Criteria :

Exemple :

```
      List personnes = session.createCriteria(Personne.class).list();
```

Les critères de recherche permettant de restreindre les données retournées par la requête sont définis par l'interface org.hibernate.criterion.Criterion. Le type Criterion encapsule un élément de la clause "where" de la requête SQL qui sera générée.

Exemple :

```
      Criteria criteria = session.createCriteria(Personne.class);
      Criterion critere = Restrictions.eq("id", 2l);
      criteria.add(critere);
      List personnes = criteria.list();

      System.out.println("nb personnes = " + personnes.size());
      Iterator it = personnes.iterator();
      while (it.hasNext()) {
        Personne personne = (Personne) it.next();
        System.out.println("Personne : " + personne);
      }
```

La classe org.hibernate.criterion.Restrictions est une fabrique qui propose des méthodes statiques pour créer des instances de type Criterion

Depuis la version 3.x d'Hibernate, il est préférable d'utiliser la classe Restrictions à sa classe fille Expressions.

L'interface org.hibernate.criterion.Projection encapsule un champ en réponse de la requête (un champ dans la clause "select" de la requête SQL).

La classe org.hibernate.criterion.Projections est une fabrique pour les instances de type Projection.

Exemple :

```
SELECT nom FROM Personne
```

Exemple :

```
    List noms = session.createCriteria(Personne.class)
           .setProjection(Projections.property("nom"))
           .list();

    System.out.println("nb personnes = " + noms.size());

    Iterator it = noms.iterator();
    while (it.hasNext()) {
      String nom = (String) it.next();
      System.out.println("Personne : " + nom);
    }
```

La classe Order encapsule une clause SQL "order by".

62.7.2.1. L'utilité de HQL et de l'API Criteria

L'API Criteria permet de définir à la volée des recherches de données multicritères complexes. Un exemple typique d'utilisation où cette API est particulièrement utile est la recherche de données multicritères où elle va permettre de facilement construire dynamiquement les critères à appliquer en fonction de ceux précisés par l'utilisateur de l'application.

Ceci est d'autant plus vrai que le nombre de critères optionnels est important : la génération dynamique peut alors devenir particulièrement complexe surtout si elle implique des jointures conditionnées par les critères précisés.

L'approche traditionnelle consiste à construire dynamiquement la requête HQL par concaténation des chaînes de caractères qui correspondent à chaque critère renseigné avec généralement plusieurs problématiques à gérer :

- la gestion des opérateurs logiques notamment pour l'omettre dans le cas du premier critère
- la gestion des jointures à utiliser (ajout des classes et des critères de jointures)
- l'utilisation directe des données dans la requête générée (pouvant par exemple conduire à des failles de sécurité dans les applications web)

Exemple :

```
  public List rechercher(Session session,
                              String nom,
                              String prenom,
                              Date dateDeb,
                              Date dateFin,
                              Integer taille) {
    Map<String, Object> parametres = new HashMap<String, Object>();
    boolean premiereClause = true;
    StringBuffer requeteBuffer = new StringBuffer("from Personne p ");

    if (nom != null) {
      requeteBuffer.append(premiereClause ? "where " : " and ");
      requeteBuffer.append("p.nom = :nom");
      parametres.put("nom", nom);
      premiereClause = false;
    }
    if (prenom != null) {
      requeteBuffer.append(premiereClause ? " where " : " and ");
      requeteBuffer.append("p.prenom = :prenom");
      parametres.put("prenom", prenom);
      premiereClause = false;
    }
    if (dateDeb != null) {
      requeteBuffer.append(premiereClause ? " where " : " and ");
      requeteBuffer.append("p.dateNais >= :dateDeb");
      parametres.put("dateDeb", dateDeb);
      premiereClause = false;
    }
    if (dateFin != null) {
      requeteBuffer.append(premiereClause ? " where " : " and ");
      requeteBuffer.append("p.dateNais <= :dateFin");
      parametres.put("endDate", dateFin);
      premiereClause = false;
    }
    if (taille != null) {
      requeteBuffer.append(premiereClause ? " where " : " and ");
      requeteBuffer.append("p.taille = :taille");
      parametres.put("taille", taille);
      premiereClause = false;
    }

    String requeteHql = requeteBuffer.toString();

    Query query = session.createQuery(requeteHql);

    Iterator<String> iter = parametres.keySet().iterator();
    while (iter.hasNext()) {
      String name = iter.next();
      Object value = parametres.get(name);
      query.setParameter(name, value);
    }

    return query.list();
  }
```

Cette approche est lourde et source d'erreurs car le code contient des portions similaires répétées. Il est aussi très important de ne pas concaténer directement des valeurs saisies par l'utilisateur dans la requête.

L'API Criteria propose une solution plus propre, plus concise et plus sûre.

Exemple :

```
  public List rechercher(Session session,
                              String nom,
                              String prenom,
                              Date dateDeb,
                              Date dateFin,
                              Integer taille) {
    Criteria criteria = session.createCriteria(Personne.class);
    if (dateDeb != null) {
      criteria.add(Restrictions.ge("dateNais", dateDeb));
    }
    if (dateFin != null) {
      criteria.add(Restrictions.le("dateNais", dateFin));
    }
    if (nom != null) {
      criteria.add(Restrictions.eq("nom", nom));
    }
    if (prenom != null) {
      criteria.add(Restrictions.eq("prenom", prenom));
    }
    if (taille != null) {
      criteria.add(Restrictions.eq("taille", taille));
    }
    List resultat = criteria.list();

    return resultat;

  }
```

La quantité de code nécessaire en utilisant l'API Criteria est beaucoup moins importante que la quantité nécessaire à la construction dynamique de la requête HQL.

62.7.2.2. L'interface Criteria

L'interface org.hibernate.Criteria propose des fonctionnalités pour encapsuler une requête composée de critères.

Une instance de l'interface Criteria est obtenue en invoquant la méthode createCriteria() de la session hibernate. Elle attends en paramètre la classe d'une entité sur laquelle les critères vont s'appliquer.

L'interface Criteria propose différentes méthodes pour construire les critères d'interrogation sur une classe persistante.

Méthode	Rôle
Criteria add(Criterion criterion)	ajouter un critère Criteria
addOrder(Order order)	ajouter un ordre de tri
Criteria createAlias(String associationPath, String alias)	ajouter une jointure en lui assignant un alias
Criteria createAlias(String associationPath, String alias, int joinType)	créer un objet de type Criteria pour une entité donnée en précisant son type et en lui assignant un alias
Criteria createCriteria(String associationPath)	créer un objet de type Criteria pour une entité donnée
Criteria createCriteria(String associationPath, int joinType)	créer un objet de type Criteria pour une entité donnée en précisant son type de jointure
Criteria createCriteria(String associationPath, String alias)	créer un objet de type Criteria pour une entité donnée en lui assignant un alias
Criteria createCriteria(String associationPath, String alias, int joinType)	créer un objet de type Criteria pour une entité donnée en précisant son type, en lui assignant un alias et en précisant son type de jointure
String getAlias()	obtenir l'alias de l'entité encapsulée dans le Criteria
List list()	obtenir les résultats de la requête
ScrollableResults scroll()	obtenir les résultats comme une instance de type ScrollableResults
ScrollableResults scroll(ScrollMode scrollMode)	obtenir les résultats comme une instance de type ScrollableResults en précisant le mode de parcours
Criteria setCacheable(boolean cacheable)	activer ou non la mise en cache des résultats de la requête

Criteria setCacheMode(CacheMode cacheMode)	modifier le mode de mise en cache des résultats de la requête
Criteria setCacheRegion(String cacheRegion)	préciser le nom de la région du cache à utiliser pour stocker les résultats de la requête
Criteria setComment(String comment)	ajouter un commentaire à la requête SQL
Criteria setFetchMode(String associationPath, FetchMode mode)	préciser le mode de récupération des données dans le cas d'une association entre deux entités
Criteria setFetchSize(int fetchSize)	préciser le nombre d'occurrences retournées par la requête
Criteria setFirstResult(int firstResult)	préciser la première occurrence qui sera retournée
Criteria setFlushMode(FlushMode flushMode)	préciser le mode de flush de la requête
Criteria setLockMode(LockMode lockMode)	préciser le mode de verrou de l'entité
Criteria setLockMode(String alias, LockMode lockMode)	préciser le mode de verrou pour l'entité dont l'alias est fourni en paramètre
Criteria setMaxResults(int maxResults)	assigner un nombre maximum d'occurrences retournées dans le résultat
Criteria setProjection(Projection projection)	préciser le contenu du résultat de la requête
Criteria setResultTransformer(ResultTransformer resultTransformer)	préciser une stratégie de traitement des résultats de la requête
Criteria setTimeout(int timeout)	assigner un timeout à la requête
Object uniqueResult()	ne renvoyer qu'une seule instance en résultat de la requête ou null si la requête ne renvoie aucun résultat.

Exemple :

```
Criteria criteria = session.createCriteria(Personne.class);
criteria.setMaxResults(10);
List personnes = criteria.list();
```

62.7.2.3. L'interface Criterion

L'interface org.hibernate.criterion.Criterion définit les méthodes qui vont permettre de définir un critère à appliquer dans la requête.

Chaque instance d'un critère doit être ajoutée aux critères de la requête en utilisant la méthodes add() de l'instance de type Criteria.

L'API propose plusieurs fabriques qui permettent d'instancier les différents objets qui vont définir le contenu de la requête.

Exemple :

```
List personnes = session.createCriteria(Personne.class)
              .add(Restrictions.like("nom", "Dup%"))
              .add(Restrictions.gt("taille", new Integer(180)))
              .addOrder(Order.asc("age"))
              .list();
```

La classe org.hibernate.criterion.Restrictions est une fabrique qui propose des méthodes pour obtenir différentes instances de Criterion.

Exemple :

```
Calendar cal = Calendar.getInstance();
cal.set(1980, Calendar.JANUARY, 01);
```

```
      Date dateDeb = cal.getTime();
      cal.set(1980, Calendar.DECEMBER, 31);
      Date dateFin = cal.getTime();
      personnes = session.createCriteria(Personne.class)
                         .add(Restrictions.ge("dateNais", dateDeb))
                         .add(Restrictions.le("dateNais", dateFin))
                         .addOrder(Order.asc("dateNais"))
                         .setFirstResult(0)
                         .setMaxResults(10)
                         .list();
```

L'interface Criterion possède de nombreuses interfaces filles : AbstractEmptinessExpression, BetweenExpression, Conjunction, Disjunction, EmptyExpression, Example, ExistsSubqueryExpression, IdentifierEqExpression, IlikeExpression, InExpression, Junction, LikeExpression, LogicalExpression, NaturalIdentifier, NotEmptyExpression, NotExpression, NotNullExpression, NullExpression, PropertyExpression, PropertySubqueryExpression, SimpleExpression, SimpleSubqueryExpression, SizeExpression, SQLCriterion, SubqueryExpression.

62.7.2.4. Les restrictions et les expressions

La classe org.hibernate.criterion.Restrictions permet de créer des critères qui sont des conditions permettant de sélectionner les données à retrouver.

La classe Restrictions est une fabrique qui permet de créer des critères de recherche sous la forme d'instances de type Criterion. Les critères proposés encapsulent les opérateurs SQL standards.

Elle propose des méthodes statiques pour créer des instances des différentes implémentations de l'interface Criterion proposées par Hibernate. Ces critères sont utilisés pour définir les occurrences qui seront retournées par le résultat de la requête.

Exemple :

```
SELECT * FROM Personne WHERE IdPers=1;
```

Exemple :

```
      List personnes = session.createCriteria(Personne.class)
                              .add(Restrictions.eq("id", 1l))
                              .list();
```

Les conditions standards de SQL sont encapsulées dans des objets de type Criterion : pour obtenir une de leurs instances, il faut utiliser les méthodes statiques de la classe Restrictions dont les principales sont :

Méthode	Rôle
Criterion allEq(Map properties)	Méthode utilitaire qui permet de facilement vérifier que plusieurs propriétés ont une valeur particulière. Les clés du paramètre de type Map correspondent aux noms des propriétés concernées
LogicalExpression and(Criterion lhs, Criterion rhs)	Créer un critère de type "and" qui est vrai si les deux critères sont évalués à vrai
Criterion between(String propertyName, Object lo, Object hi)	Permet d'appliquer une contrainte SQL de type "between" : la valeur de la propriété dont le nom est fourni en paramètre doit être comprise entre les deux valeurs fournies
Conjunction conjunction()	Créer un objet de type Conjunction qui permet d'utiliser un critère de type and simplement en invoquant sa méthode add() pour chaque critère à prendre en compte. Le critère encapsulé dans l'objet de type Conjunction sera true si tous les critères qu'il contient sont true
Disjunction disjunction()	

	Créer un objet de type Disjunction qui permet d'utiliser un critère de type or simplement en invoquant sa méthode add() pour chaque critère à prendre en compte. Le critère encapsulé dans l'objet de type Disjunction sera true si au moins un des critères qu'il contient est true
SimpleExpression eq(String propertyName, Object value)	Permet d'appliquer une contrainte SQL de type égalité : la valeur de la propriété doit être égale à la valeur fournie en paramètre
PropertyExpression eqProperty(String propertyl, String property2)	La valeur des deux propriétés doit être égale
SimpleExpression ge(String propertyName, Object value)	Permet d'appliquer une contrainte SQL de type "supérieur ou égal" : la valeur de la propriété doit être supérieure ou égale à la valeur fournie en paramètre
PropertyExpression geProperty(String propertyl, String property2)	La valeur de la propriété doit être supérieure ou égale à la valeur de la seconde propriété fournie en paramètre
SimpleExpression gt(String propertyName, Object value)	Permet d'appliquer une contrainte SQL de type "supérieur à" : la valeur de la propriété doit être supérieure à la valeur fournie en paramètre
PropertyExpression gtProperty(String propertyl, String property2)	La valeur de la propriété doit être supérieure à la valeur de la seconde propriété fournie en paramètre
Criterion idEq(Object value)	permet d'appliquer une contrainte SQL de type égalité sur l'identifiant : la valeur de la propriété qui est l'identifiant doit être égale à celle fournie
Criterion ilike(String property, Object value)	Joue le même rôle que la méthode like() mais en étant insensible à la casse
Criterion ilike(String property, String value, MatchMode mode)	Joue le même rôle que la méthode like() mais en étant insensible à la casse et sans utiliser la syntaxe de l'opérateur like. MatchMode est une énumération qui peut prendre les valeurs START, END, ANYWHERE, ou EXACT.
Criterion in(String property, Collection values)	La valeur de la propriété dont le nom est fourni en paramètre doit être égale à l'une de celles fournies dans la collection
Criterion in(String propertyName, Collection values)	Permet d'appliquer une contrainte SQL de type "in" : la valeur de la propriété dont le nom est fourni en paramètre doit être égale à l'une de celles fournies dans le tableau
Criterion isEmpty(String property)	Le contenu de la collection de la propriété dont le nom est fourni en paramètre ne doit pas avoir d'éléments
Criterion isNotEmpty(String property)	Le contenu de la collection de la propriété dont le nom est fourni en paramètre doit avoir au moins un élément
Criterion isNotNull(String propertyName)	Permet d'appliquer une contrainte SQL de type "is not null" : la valeur de la propriété dont le nom est fourni en paramètre doit être non null
Criterion isNull(String propertyName)	Permet d'appliquer une contrainte SQL de type "is null" : la valeur de la propriété dont le nom est fourni en paramètre doit être null
SimpleExpression le(String property, Object value)	La valeur de la propriété dont le nom est fourni en paramètre doit être inférieure ou égale à la valeur fournie
PropertyExpression leProperty(String propertyl, String property2)	La valeur de la propriété fournie en paramètre doit être inférieure ou égale à la valeur fournie
SimpleExpression like(String property, Object value)	La valeur de la propriété dont le nom est fourni en paramètre doit respecter le motif de l'opérateur sql like fourni en paramètre
SimpleExpression like(String property, String value, MatchMode mode)	La valeur de la propriété dont le nom est fourni en paramètre doit respecter le motif de l'opérateur sql like déterminé à partir des paramètre valeur et mode. MatchMode est une énumération qui peut prendre les valeurs START, END, ANYWHERE, ou EXACT.

SimpleExpression lt(String property, Object value)	La valeur de la propriété doit être inférieure à la valeur fournie en paramètre
PropertyExpression ltProperty(String propertyl, String property2)	La valeur de la première propriété doit être inférieure à la seconde
SimpleExpression ne(String propertyName, Object value)	Permet d'appliquer une contrainte SQL de type "est différent de" : la valeur de la propriété dont le nom est fourni en paramètre doit être différente de la valeur fournie
PropertyExpression neProperty(String propertyName, String otherPropertyName)	La valeur des deux propriétés fournies en paramètres doit être différente
Criterion not(Criterion expression)	L'évaluation du critère fourni en paramètre doit être false
LogicalExpression or(Criterion lhs, Criterion rhs)	L'évaluation d'un des deux critères fourni en paramètre doit être true
Criterion sqlRestriction(String sql)	Appliquer une restriction en sql natif
Criterion sqlRestriction(String sql, Object[] values, Type[] types)	Appliquer une restriction en sql natif qui va utiliser les paramètres fournis
Criterion sqlRestriction(String sql, Object value, Type type)	Appliquer une restriction en sql natif qui va utiliser le paramètre fourni

Certaines de ces méthodes attendent un paramètre de type Criterion qui permet de faire des combinaisons de critères.

Les opérateurs de comparaison sont encapulés dans des méthodes de la classe Restrictions : eq(), lt(), le(), gt(), ge().

Exemple :

```
      List personnes = session.createCriteria(Personne.class)
                              .add(Restrictions.lt("dateNais", dateSaisie))
                              .list();
```

La classe Restrictions propose aussi des méthodes pour les opérateurs SQL : like, between, in, is null, is not null ...

Exemple :

```
      List personnes = session.createCriteria(Personne.class)
                              .add(Restrictions.between("dateNais", dateDeb, dateFin))
                              .add(Restrictions.like("nom", "Dup%"))
                              .list();
```

La classe Restrictions propose aussi des méthodes qui permettent de faire des comparaisons entre propriétés.

Exemple :

```
      List personnes = session.createCriteria(Personne.class)
                              .add(Restrictions.eqProperty("nom", "prenom"))
                              .list();
```

L'utilisation de la méthode sqlRestriction() peut être très pratique mais elle peut nuire à la portabilité entre bases de données.

Pour préciser l'alias de la table courante dans la portion de requête SQL fournie en paramètre de la méthode sqlRestriction(), il faut utiliser la syntaxe "{alias}".

Il est possible d'utiliser les méthodes or() et and() pour réaliser des combinaisons de critères.

Exemple :

```
      List personnes = session.createCriteria(Personne.class)
                              .add(Restrictions.or(Restrictions.eq("prenom", "Jean"),
                                   Restrictions.eq("prenom", "Paul")))
                              .list();
```

Remarque : il est préférable d'utiliser la classe Restrictions plutôt que sa classe fille Expressions qui est deprecated.

62.7.2.5. Les projections et les aggregations

La classe org.hibernate.criterion.Projection permet de préciser un champ qui sera retourné dans le résultat de la requête : ce champ peut être issu d'une table, du calcul d'une aggrégation, de la définition d'un alias, ...

Pour ajouter un champ, il faut passer le nom du champ en paramètre de la méthode statique property() de la classe Projection. L'instance retournée est passée en paramètre de la méthode setProjection().

La classe org.hibernate.criterion.Projections est une fabrique pour créer des instances de type Projection.

Pour préciser plusieurs champs, il faut utiliser la méthode propertyList() de la classe ProjectionList.

Exemple :

```
SELECT NOM, PRENOM FROM PERSONNE
```

Exemple :

```
      List resultats = session.createCriteria(Personne.class)
                .setProjection(Projections.projectionList()
                               .add(Projections.property("nom"))
                               .add(Projections.property("prenom")))
                .list();

      System.out.println("nb personnes = " + resultats.size());
      Iterator it = resultats.iterator();
      while (it.hasNext()) {
        Object[] donnees = (Object[]) it.next();
        System.out.println("Nom : " + donnees[0]
                         + " Prenom : " + donnees[1]);
      }
```

Pour appliquer une collection de type Projection aux critères de la requête, il faut utiliser la méthode setProjection() de la classe Criteria. Une collection d'objets de type Projection est encapsulée dans un objet de type ProjectionList. La méthode add() permet d'ajouter une Projection à la collection.

Exemple :

```
      List resultats = session.createCriteria(Personne.class)
                              .setProjection(Projections.rowCount())
                              .list();
      Long valeur = (Long) resultats.get(0);
      System.out.println("nb personnes = " + valeur);
```

Les données de certaines requêtes doivent parfois être groupées ou intervenir dans un calcul d'aggrégation : il faut pour cela utiliser les fonctionnalités encapsulées dans la classe Projections. Toutes les fonctions d'aggrégation de la classe Projections sont des méthodes statiques.

La classe Projections possède plusieurs méthodes statiques :

Méthode	Rôle

static Projections alias(Projection projection, String alias)	Assigner un alias
static AgregateProjection avg(String property)	Calculer la moyenne du champ dont le nom est fourni en paramètre
static CountProjection count(String property)	Calculer le nombre d'occurrences du champ dont le nom et fourni en paramètre
static CountProjection countDistinct(String property)	Calculer le nombre d'occurrences distinctes du champ dont le nom et fourni en paramètre
static Projection distinct(Projection projection)	Ne retourner que des valeurs uniques (supprimer les valeurs en doublon).
static PropertyProjection groupProperty(String property)	Grouper les résultats sur la propriété fournie
static IdentifierProjection id()	Renvoyer l'identifiant
static AggregateProjection max(String property)	Déterminer la plus grande valeur pour le champ dont le nom est fourni en paramètre
static AggregateProjection min(String property)	Déterminer la plus petite valeur pour le champ dont le nom est fourni en paramètre
static ProjectionList projectionList()	Retourner une collection de projections
static PropertyProjection property(String property)	Ajouter la propriété fournie en paramètre
static Projection rowCount()	Calculer le nombre d'occurrences retournées par la requête
static Projection sqlGroupProjection(String sql, String groupBy, String[] columnAliases, Type[] types)	Ajouter du code SQL spécifique à la base de données utilisée pour déterminer un groupage. Le paramètre groupBy peut contenir une clause GROUP BY
static Projection sqlProjection(String sql, String[] columnAliases, Type[] types)	Ajouter du code SQL spécifique à la base de données utilisée pour déterminer la liste de champs retournée par la requête
static AggregateProjection sum(String property)	Calculer la sommes des valeurs pour le champ dont le nom est fourni en paramètre

Lors de l'utilisation d'opérateurs d'aggrégation, il est fréquent de grouper les données par rapport à un champ particulier. La classe Projections prossède la méthode groupProperty() qui permet de définir une clause "group by" qui sera utilisée avec le nom du champ fourni en paramètre.

Exemple :

```
SELECT COUNT(ID) FROM PERSONNE GROUP BY TAILLE
```

Exemple :

```
      List resultats = session.createCriteria(Personne.class)
                    .setProjection(Projections.projectionList()
                                          .add(Projections.count("id"))
                                          .add(Projections.groupProperty("taille")))
                    .list();

      Iterator it = resultats.iterator();
      while (it.hasNext()) {
        Object[] donnees = (Object[]) it.next();
        System.out.println("Nombre : " + donnees[0] + " taille : " + donnees[1]);
      }
```

62.7.2.6. La classe Property

La classe org.hibernate.criterion.Property est une fabrique qui permet de créer des critères spécifiques appliqués à la propriété encapsulée. Ceci permet d'appliquer des critères directement sur une propriété.

La méthode statique forName() de la classe Property permet d'obtenir une instance qui encapsule une propriété.

La classe org.hibernate.criterion.Property propose plusieurs méthodes pour appliquer des critères sur la propriété qu'elle encapsule.

Les principales méthodes sont :

Méthode	Rôle
Order asc()	Trier les valeurs de la propriété dans un ordre ascendant
AggregateProjection avg()	Créer un champ qui calcule la moyenne des occurrences de la propriété
Criterion between(Object min, Object max)	Créer un critère qui requiert que la valeur de la propriété soit comprise entre les valeurs min et max fournies en paramètres
CountProjection count()	Créer un champ qui compte le nombre d'occurrences de la propriété
Order desc()	Trier les valeurs de la propriété dans un ordre descendant
SimpleExpression eq(Object value)	Créer un critère pour filtrer les résultats de façon à ne retourner que les occurrences dont la valeur de la propriété soit égale à celle fournie en paramètre
PropertyExpression eqProperty(Property other)	Créer un critère qui requiert que la valeur de la propriété soit égale à celle de la valeur de la propriété fournie en paramètre
PropertyExpression eqProperty(String property)	Créer un critère qui requiert que la valeur de la propriété soit égale à celle de la valeur de la propriété dont le nom est fourni en paramètre
SimpleExpression ge(Object value)	Créer un critère qui requiert que la valeur de la propriété soit supérieure ou égale à celle fournie en paramètre
PropertyExpression geProperty(Property other)	Créer un critère qui requiert que la valeur de la propriété soit supérieure ou égale à celle de la valeur de la propriété fournie en paramètre
PropertyExpression geProperty(String property)	Créer un critère qui requiert que la valeur de la propriété soit supérieure ou égale à celle de la valeur de la propriété dont le nom est fourni en paramètre
PropertyProjection group()	Demander un groupage sur les valeurs de la propriété
SimpleExpression gt(Object value)	Créer un critère qui requiert que la valeur de la propriété soit supérieure à celle fournie en paramètre
PropertyExpression gtProperty(Property other)	Créer un critère qui requiert que la valeur de la propriété soit supérieure à celle de la valeur de la propriété fournie en paramètre
PropertyExpression gtProperty(String property)	Créer un critère qui requiert que la valeur de la propriété soit supérieure à celle de la valeur de la propriété dont le nom est fourni en paramètre
Criterion in(Collection values)	Créer un critère qui requiert que la valeur de la propriété doit être égale à une de celles fournies
in(Object[] values)	Créer un critère qui requiert que la valeur de la propriété doit être égale à une de celles fournies
Criterion isEmpty()	Créer un critère qui requiert que la valeur de la propriété ne possède aucun élément
Criterion isNotEmpty()	Créer un critère qui requiert que la valeur de la propriété possède au moins un élément
Criterion isNotNull()	Créer un critère qui requiert que la valeur de la propriété ne soit pas null
Criterion isNull()	Créer un critère qui requiert que la valeur de la propriété soit null
SimpleExpression le(Object value)	Créer un critère qui requiert que la valeur de la propriété soit inférieure ou égale à celle fournie en paramètre

PropertyExpression leProperty(Property other)	Créer un critère qui requiert que la valeur de la propriété soit inférieure ou égale à celle de la valeur de la propriété fournie en paramètre
PropertyExpression leProperty(String property)	Créer un critère qui requiert que la valeur de la propriété soit inférieure ou égale à celle de la valeur de la paramètre dont le nom est fourni en paramètre
SimpleExpressionlike(Object value)	Créer un critère qui requiert que la valeur de la propriété respecte le motif fourni au sens de l'opérateur SQL like
like(String value, MatchMode mode)	Créer un critère qui requiert que la valeur de la propriété respecte un motif fourni au sens de l'opérateur SQL like Le motif est construit à partir de la sous chaîne fournie en paramètre et de son mode recherche qui peut prendre les valeurs START, END, ANYWHERE, and EXACT. Ceci permet d'éviter d'avoir à manipuler la syntaxe de l'opérateur SQL like.
SimpleExpression lt(Object value)	Créer un critère qui requiert que la valeur de la propriété soit supérieure ou égale à celle fournie en paramètre
PropertyExpression ltProperty(Property other)	Créer un critère qui requiert que la valeur de la propriété soit supérieure ou égale à celle de la valeur de la propriété fournie en paramètre
PropertyExpression ltProperty(String property)	Créer un critère qui requiert que la valeur de la propriété soit inférieure ou égale à celle de la valeur de la paramètre dont le nom est fourni en paramètre
AggregateProjection max()	Créer un champ qui va contenir la valeur la plus élevée de la propriété
AggregateProjection min()	Créer un champ qui va contenir la valeur la moins élevée de la propriété
PropertyProjection ne(Object value)	Créer un critère qui requiert que la valeur de la propriété soit différente de celle fournie en paramètre
PropertyProjection neProperty(Property other)	Créer un critère qui requiert que la valeur de la propriété soit différente de celle de la valeur de la propriété fournie en paramètre
PropertyExpression neProperty(String property)	Créer un critère qui requiert que la valeur de la propriété soit différente de celle de la valeur de la paramètre dont le nom est fourni en paramètre

62.7.2.7. Le tri des résultats

Il est possible de demander le tri des résultats de la requête en utilisant la méthode addOrderO de la classe Criteria et la classe Order.

La classe org.hibernate.criterion.Order permet d'encapsuler une clause de tri dans la requête en précisant le sens (ascendant ou descendant) sur un champ.

Elle possède plusieurs méthodes :

Méthode	Rôle
static Order asc(String)	Demander un tri ascendant sur le nom du champ fourni en paramètre de la méthode
static Order desc(String)	Demander un tri descendant sur le nom du champ fourni en paramètre de la méthode
Order ignoreCase()	Demander d'ignorer la casse lors du tri des données

Exemple :

```
    List resultats = session.createCriteria(Personne.class)
      .add(Restrictions.between("dateNais",dateDeb,dateFin))
      .addOrder(Order.desc("dateNais"))
      .addOrder(Order.asc("nom"));
```

62.7.2.8. La jointure de tables

Il est très fréquent d'avoir à effectuer une ou plusieurs jointures sur les tables d'une requête pour obtenir les données souhaitées.

En SQL, la jointure se fait en utilisant les tables dans la clause from et en indiquant les conditions de la jointure.

Exemple :

```
SELECT P.*, A.* FROM Personne P, Adresse A WHERE P.adresse_id=A.id
AND A.Nom = 'Dupond';
```

En HQL, il est possible de charger les données d'objets dépendants en utilisant la clause "left join fetch" :

Exemple :

```
from Personne personne
where Personne.nom = :nom left join fetch personne.adresse
```

La méthode setFetchMode() permet de faire une jointure avec l'API Criteria.

Exemple :

```
package fr.jmdoudoux.dej.hibernate;

import java.util.Iterator;
import java.util.List;
import org.hibernate.FetchMode;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.AnnotationConfiguration;

public class TestHibernateCriteria {

  public static void main(String args[]) {
    SessionFactory sessionFactory = new AnnotationConfiguration().configure()
                                                                 .buildSessionFactory();
    Transaction transaction = null;
    List personnes = null;
    Session session = sessionFactory.openSession();

    try {
      transaction = session.beginTransaction();
      personnes = session.createCriteria(Personne.class)
                         .setFetchMode("adresse", FetchMode.JOIN)
                         .list();
      System.out.println("nbpersonnes = " + personnes.size());

      Iterator it = personnes.iterator();
      while (it.hasNext()) {
        Personne personne = (Personne) it.next();

        System.out.println("Personne : " + personne);
        System.out.println("  Adresse : " + personne.getAdresse());
      }

      transaction.commit();
    } catch (Exception e) {
      transaction.rollback();
      e.printStackTrace();
    } finally {
      session.close();
    }
    sessionFactory.close();
  }
}
```

La méthode setFetchMode() attend deux paramètres :

- le nom de la classe de l'entité dont la table sera jointe avec celle de l'entité courante
- le mode de récupération des données : DEFAULT (valeur configurée dans le mapping), EAGER (deprecated : utiliser JOIN), JOIN (récupération des données par jointure), LAZY (deprecated : utiliser SELECT), SELECT (récupération des données par une requête dédiée)

Parfois, il est nécessaire de joindre les entités mais il est inutile de récupérer les données de l'entité jointe : dans ce cas la jointure n'est utile que pour définir un ou plusieurs critères de la requête

Exemple :

```
 from Personne p join p.adresse a where a.cp = '54000'
```

Il est possible d'utiliser la méthode createCriteria() pour créer une jointure :

Exemple :

```
package fr.jmdoudoux.dej.hibernate;

import java.util.Iterator;
import java.util.List;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.AnnotationConfiguration;
import org.hibernate.criterion.Restrictions;

public class TestHibernateCriteria {

  public static void main(String args[]) {
    SessionFactory sessionFactory = new AnnotationConfiguration().configure()
                                                                 .buildSessionFactory();

    Transaction transaction = null;
    List personnes = null;

    Session session = sessionFactory.openSession();
    try {
      transaction = session.beginTransaction();
      personnes = session.createCriteria(Personne.class)
                         .createCriteria("adresse", "a")
                         .add(Restrictions.eq("a.cp", "54700"))
                         .list();

      System.out.println("nb personnes = " + personnes.size());

      Iterator it = personnes.iterator();
      while (it.hasNext()) {
        Personne personne = (Personne) it.next();
        System.out.println("Personne : " + personne);
        System.out.println("  Adresse: " + personne.getAdresse());
      }

      transaction.commit();
    } catch (Exception e) {
      transaction.rollback();
      e.printStackTrace();
    } finally {
      session.close();
    }

    sessionFactory.close();
  }
}
```

Attention : dans ce cas, le premier paramètre de la méthode createCriteria() est le nom d'une propriété et non le nom de classe.

Il est aussi possible d'utiliser la méthode createAlias() qui évite d'avoir à créer une nouvelle instance de la classe Criteria.

Exemple :

```
      personnes = session.createCriteria(Personne.class)
                         .createAlias("adresse", "a")
                         .add(Restrictions.eq("a.cp", "54000"))
                         .list();
```

Les restrictions sont aussi utilisables avec une jointure dans laquelle les données de la table jointe sont récupérées.

Exemple :

```
SELECT G.*, P.* FROM Personnes P, Groupe G WHERE G.IdGroup=P.IdGroup AND G.IdGroup=l;
```

Exemple :

```
      List groupes = session.createCriteria(Groupe.class)
                            .setFetchMode("Personne",FetchMode.JOIN)
                            .add(Restrictions.eq("IdGroup","1"))
                            .list();
```

62.7.2.9. La création de critères à partir de données

La classe Example permet de créer des critères par l'exemple : pour cela, il faut instancier une entité et lui affecter les différentes valeurs qui seront utilisées comme critères.

La méthode static create() qui attend en paramètre une instance d'une entité va utiliser l'instrospection pour rechercher les propriétés qui possèdent une valeur et générer une instance de type Example contenant les critères correspondants aux propriétés renseignées.

Exemple :

```
      transaction = session.beginTransaction();
      Date dateDebut = new GregorianCalendar(1980, Calendar.JANUARY, 01).getTime();
      Date dateFin = new GregorianCalendar(1981, Calendar.JANUARY, 01).getTime();

      Personne personneExemple = new Personne();
      personneExemple.setNom("Dupond");
      personneExemple.setPrenom("Michel");
      Example example = Example.create(personneExemple).ignoreCase().excludeZeroes();
      List<Personne> resultats = session.createCriteria(Personne.class)
                                              .add(example)
                                              .add(Restrictions.between("dateNais",
                                                   dateDebut, dateFin)).list();
      System.out.println("nbpersonnes = " + personnes.size());

      Iterator it = personnes.iterator();
      while (it.hasNext()) {
        Personne personne = (Personne) it.next();
        System.out.println("Personne : " + personne);
        System.out.println("  Adresse : " + personne.getAdresse());
      }
      transaction.commit();
```

Il est possible de configurer le comportement de la classe Example en utilisant les différentes méthodes qu'elle propose à cette fin :

Méthode	Rôle
Example enableLike()	Utiliser l'opérateur like pour toutes les propriétés de type String
Example enableLike(MatchMode matchMode)	Utiliser l'opérateur like pour toutes les propriétés de type String avec la stratégie de correspondance fournie en paramètre

Example excludeNone()	Ne pas exclure les propriétés dont la valeur est null ou zéro
Example excludeProperty(String name)	Ignorer la propriété dont le nom est fourni en paramètre
Example excludeZeroes()	Ignorer les propriétés dont la valeur est zéro
Example ignoreCase()	Ignorer la casse des propriétés de type String
Example setEscapeCharacter(Character escapeCharacter)	Préciser le caractère d'échappement utilisé dans la clause like

Cette API est particulièrement utile si le nombre de propriétés est important. Comme la classe Example hérite de la classe Criterion, elle peut être utilisée pour créer les critères "simple" et utiliser d'autres classes pour les critères plus complexes ou spécifiques.

62.7.2.10. Le choix entre HQL et l'API Criteria

L'API Criteria est très puissante et elle est particulièrement bien adaptée pour certaines tâches (notamment la création de requêtes dynamiques à la volée impliquant de nombreux critères optionnels comme par exemple dans un formulaire de recherche multicritères).

Dans ces cas, sa mise en oeuvre peut permettre d'avoir un code plus propre, plus sûre et plus maintenable.

Cependant, son utilisation ne peut pas toujours être généralisée à tous les cas de figure car l'utilisation de HQL est parfois préférable, notamment si la requête n'est pas dynamique.

Il est par exemple préférable d'externaliser les requêtes HQL lorsque cela est possible ce qui présente plusieurs avantages :

- faciliter de maintenance
- faciliter pour recencer et optimiser les requêtes
- faciliter pour mettre les requêtes en cache

Il est donc nécessaire de bien choisir entre HQL et l'API Criteria en fonction des besoins et de leur adéquation avec ce que proposent les deux solutions qui se recouvrent mais sont aussi complémentaires.

62.8. La mise à jour d'une occurrence

Pour mettre à jour une occurrence dans la source de données, il suffit d'appeler la méthode update() de la session en lui passant en paramètre l'objet encapsulant les données.

Le mode de fonctionnement de cette méthode est similaire à celui de la méthode save().

La méthode saveOrUpdate() laisse Hibernate choisir entre l'utilisation de la méthode save() ou update() en fonction de la valeur de l'identifiant dans la classe encapsulant les données.

62.9. La suppression d'une ou plusieurs occurrences

La méthode delete() de la classe Session permet de supprimer une ou plusieurs occurrences en fonction de la version surchargée de la méthode utilisée.

Pour supprimer une occurrence encapsulée dans une classe, il suffit d'invoquer la méthode delete() en lui passant en paramètre l'instance de la classe.

Pour supprimer plusieurs occurrences, voire toutes, il faut passer en paramètre de la méthode delete(), une chaîne de caractères contenant la requête HQL pour préciser les éléments concernés par la suppression.

Exemple : suppression de toutes les occurrences de la table
`session.delete("from Personnes");`

62.10. Les relations

Un des fondements du modèle de données relationnelles repose sur les relations qui peuvent intervenir entre une table et une ou plusieurs autres tables ou la table elle même.

Les relations utilisables dans le monde relationnel et le monde objet sont cependant différentes.

Les relations du monde objets possèdent quelques caractéristiques :

- Elles sont réalisées via des références entre objets
- Elles peuvent mettre en oeuvre l'héritage et le polymorphisme

Les relations du monde relationnel possèdent quelques caractéristiques :

- Elles sont gérées par des clés étrangères et des jointures entre tables

Les caractéristiques de ces deux modèles sont assez différentes : le but d'un outil de type ORM comme Hibernate est de permettre de manipuler des entités objets et de masquer au développeur le monde relationnel en assurant un mapping entre les deux mondes. Cependant Hibernate n'assure pas en automatique la gestion inverse des relations qui reste à la charge du développeur.

Hibernate propose de transcrire les relations du modèle relationnel dans le modèle objet. Il supporte plusieurs types de relations :

- relation de type 1 - 1 (one-to-one)
- relation de type 1 - n (one-to-many)
- relation de type n - n (many-to-many)

Dans le fichier de mapping, il est nécessaire de définir les relations entre la table concernée et les tables avec lesquelles elle possède des relations.

Les relations peuvent aussi être définies avec des annotations.

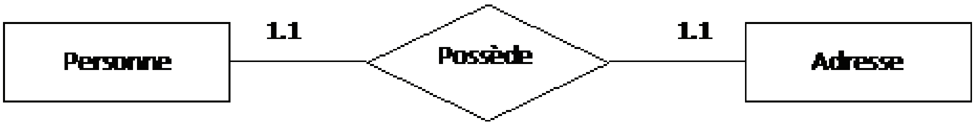
La version d'Hibernate utilisée dans cette section est la 3.5.1.

La base de données utilisée est une base MySQL version 4.1.9.

Chaque exemple possède plusieurs bibliothèques dans son classpath : commons-collections-3.1.jar, dom4j-1.6.1.jar, hibernate-jpa-2.0-api-1.0.0.Final.jar, hibernate3.jar, javassist-3.9.0.GA.jar, jta-1.1.jar, log4j-1.2.15.jar, mysql-connector-java-5.1.12-bin.jar, slf4j-api-1.5.8.jar, slf4j-log4j12-1.5.11.jar.

62.10.1. Les relations un / un

Dans ce type de relation, deux entités sont liées de façon à n'avoir qu'une seule et unique occurrence l'une pour l'autre.



Dans l'exemple ci-dessus, chaque personne ne peut avoir qu'une seule adresse et une adresse ne peut appartenir qu'à une seule personne.

Cette relation peut se traduire de plusieurs manières dans la base de données :

- Une seule table qui contient les données de la personne et son adresse
- Deux tables, une pour les personnes et une pour les adresses avec une clé primaire partagée
- Deux tables, une pour les personnes et une pour les adresses avec une clé étrangère

Il y a plusieurs façons de traiter ce cas avec une ou deux tables dans la base de données et Hibernate :

- Deux tables et une relation One-to-One d'Hibernate
- Une seule table avec un Component d'Hibernate

62.10.1.1. Le mapping avec un Component

Comme une personne ne peut avoir qu'une seule adresse, il est préférable pour des raisons de performance de stocker les données des deux entités dans une seule et même table. Ceci évite d'avoir à faire une jointure lors de l'accès aux données des deux entités.

La description de la table personne est la suivante :

Résultat :

```
mysql> desc personne;
+------------+--------------+------+-----+---------+----------------+
| Field      | Type         | Null | Key | Default | Extra          |
+------------+--------------+------+-----+---------+----------------+
| Id         | int(11)      |      | PRI | NULL    | auto_increment |
| Nom        | varchar(255) |      |     |         |                |
| Prenom     | varchar(255) |      |     |         |                |
| DateNais   | date         | YES  |     | NULL    |                |
| ligne1_adr | varchar(80)  |      |     |         |                |
| ligne2_adr | varchar(80)  | YES  |     | NULL    |                |
| cp_adr     | varchar(5)   | YES  |     | NULL    |                |
| ville_adr  | varchar(80)  | YES  |     | NULL    |                |
| ligne3_adr | varchar(80)  | YES  |     | NULL    |                |
+------------+--------------+------+-----+---------+----------------+
9 rows in set (0.00 sec)
```

Le script DDL correspondant est le suivant :

Résultat :

```
CREATE TABLE `personne` (
  `Id` int(11) NOT NULL auto_increment,
  `Nom` varchar(255) NOT NULL default '',
  `Prenom` varchar(255) NOT NULL default '',
  `DateNais` date default NULL,
  `ligne1_adr` varchar(80) NOT NULL default '',
  `ligne2_adr` varchar(80) default NULL,
  `cp_adr` varchar(5) default NULL,
  `ville_adr` varchar(80) default NULL,
  `ligne3_adr` varchar(80) default NULL,
  PRIMARY KEY  (`Id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 AUTO_INCREMENT=0 ;
```

62.10.1.1.1. La configuration dans le fichier de mapping

Les classes qui encapsulent l'entité personne et les données de l'adresse sont de simples POJO.

Exemple :

```
package fr.jmdoudoux.dej.hibernate;

public class Personne {
  private Long id;
  private String nom;
  private String prenom;
  private String dateNais;
  private Adresse adresse;

  public Personne(String nom, String prenom, String dateNais, Adresse adresse) {
    this.nom = nom;
    this.prenom = prenom;
    this.dateNais = dateNais;
    this.adresse = adresse;
  }

  public Personne() {
  }

  public String getNom() {
    return nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

  public String getPrenom() {
    return prenom;
  }

  public void setPrenom(String prenom) {
    this.prenom = prenom;
  }

  public String getDateNais() {
    return dateNais;
  }

  public void setDateNais(String dateNais) {
    this.dateNais = dateNais;
  }

  public Long getId() {
    return id;
  }

  // Attention le setter est requis par Hibernate
  public void setId(Long id) {
    this.id = id;
  }

  public Adresse getAdresse() {
    return adresse;
  }

  public void setAdresse(Adresse adresse) {
    this.adresse = adresse;
  }

  @Override
  public String toString() {
    return this.id + " : " + this.nom + " " + this.prenom;
  }
}
```

La classe Adresse ne possède pas de champ de type identifiant.

Exemple :

```
package fr.jmdoudoux.dej.hibernate;

public class Adresse {
  private String ligne1;
  private String ligne2;
  private String cp;
  private String ville;
  private String ligne3;

  public Adresse(String ligne1, String ligne2, String cp, String ville, String ligne3) {
    super();
    this.ligne1 = ligne1;
    this.ligne2 = ligne2;
    this.cp = cp;
    this.ville = ville;
    this.ligne3 = ligne3;
  }

  public Adresse() {
  }

  //
  // getter et setter sur les champs de la classe
  //

}
```

Les données de l'adresse sont encapsulées dans une classe Adresse : la définition des champs de cette classe est faite dans un élément de type component du fichier de mapping de l'entité Personne (Personne.hbm.xml).

Exemple :

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="fr.jmdoudoux.dej.hibernate.Personne" table="Personne">
    <id name="id" column="id">
      <generator class="increment" />
    </id>
    <property name="nom" column="Nom" />
    <property name="prenom" column="Prenom" />
    <property name="dateNais" column="DateNais" />
    <component name="adresse" class="fr.jmdoudoux.dej.hibernate.Adresse">
      <property name="ligne1" column="ligne1_adr" />
      <property name="ligne2" column="ligne2_adr" />
      <property name="cp" column="cp_adr" />
      <property name="ville" column="ville_adr" />
      <property name="ligne3" column="ligne3_adr" />
    </component>
  </class>
</hibernate-mapping>
```

Le fichier de configuration d'Hibernate définit les paramètres de connexion à la base de données et le fichier de mapping de l'entité Personne.

Exemple :

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
          "-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="connection.url">jdbc:mysql://localhost/mabdd</property>
    <property name="connection.username">root</property>
    <property name="connection.password"></property>
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="transaction.factory_class">
```

```
org.hibernate.transaction.JDBCTransactionFactory</property>
    <property name="current_session_context_class">thread</property>
    <property name="hibernate.show_sql">true</property>
    <mapping resource="com/jmdoudoux/test/hibernate/Personne.hbm.xml"></mapping>
  </session-factory>
</hibernate-configuration>
```

L'application de test est basique :

- créer une instance de type Adresse,
- créer une instance de type personne,
- et sauvegarder la personne dans la base de données

Exemple :

```
package fr.jmdoudoux.dej.hibernate;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

public class TestHibernate15 {

  public static void main(String args[]) {
    SessionFactory sessionFactory = new Configuration().configure()
                                                    .buildSessionFactory();

    Transaction transaction = null;
    int index = 6;

    Session session = sessionFactory.openSession();

    try {
      transaction = session.beginTransaction();

      Adresse adresse = new Adresse("ligne1_" + index, "ligne2_" + index, "cp_"
          + index, "ville" + index, "ligne3_" + index);
      Personne personne = new Personne("nom" + index,
                                       "prenom_" + index,
                                       null,
                                       adresse);

      session.save(personne);
      transaction.commit();

      System.out.println("La nouvelle personne a ete enregistree");

    } catch (Exception e) {
      transaction.rollback();
      e.printStackTrace();
    } finally {
      session.close();
    }

    sessionFactory.close();
  }
}
```

Lorsque la personne est enregistrée dans la base de données, son adresse l'est aussi dans la table Personne.

Résultat :

```
mysql> select * from personne;
+----+------+----------+----------+------------+------------+--------+----------
-+------------+
| Id | Nom  | Prenom   | DateNais | ligne1_adr | ligne2_adr |
cp_adr | ville_adr
 | ligne3_adr |
+----+------+----------+----------+-----------+-----------+--------+----------
-+------------+
|  1 | nom6 | prenom_6 | NULL     | ligne1_6   | ligne2_6
| cp_6   | ville6
 | ligne3_6   |
+----+------+----------+----------+-----------+-----------+--------+----------
-+------------+
1 row in set (0.00 sec)
```

62.10.1.1.2. La configuration avec les annotations

Le POJO qui encapsule une personne a quelques particularités relatives à la relation avec l'adresse :

Il possède un champ privé de type Adresse

Le champ adresse est annoté avec l'annotation @Embedded

Exemple :

```
package fr.jmdoudoux.dej.hibernate;

import javax.persistence.Column;
import javax.persistence.Embedded;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "personne")
public class Personne {

  @Id
  @GeneratedValue
  @Column(name = "Id")
  private Long id;

  @Column(name = "Nom")
  private String nom;

  @Column(name = "Prenom")
  private String prenom;

  @Column(name = "DateNais")
  private String dateNais;

  @Embedded
  private Adresse adresse;

  public Personne(String nom, String prenom, String dateNais, Adresse adresse) {
    this.nom = nom;
    this.prenom = prenom;
    this.dateNais = dateNais;
    this.adresse = adresse;
  }

  public Personne() {
  }

  public Long getId() {
    return id;
  }

  public Adresse getAdresse() {
    return adresse;
  }

  public void setAdresse(Adresse adresse) {
    this.adresse = adresse;
  }
  //
  // getter et setter sur les autres champs de la classe
  //

  @Override
  public String toString() {
    return this.id + " : " + this.nom + " " + this.prenom;
  }

}
```

L'annotation @Embedded permet de préciser que les données de la classe Adresse seront stockées dans la table Personne comme un component d'Hibernate.

Le POJO qui encapsule une adresse possède plusieurs particularités relatives à la relation avec la personne :

- La classe est annotée avec l'annotation @Embeddable (elle n'est pas annotée comme une entité avec les annotations @Entity et @Table)
- La classe ne possède pas de champ de type identifiant

Exemple :

```
package fr.jmdoudoux.dej.hibernate;

import javax.persistence.Column;
import javax.persistence.Embeddable;

@Embeddable
public class Adresse {

  @Column(name = "ligne1_adr", nullable = false)
  private String ligne1;

  @Column(name = "ligne2_adr")
  private String ligne2;

  @Column(name = "cp_adr")
  private String cp;

  @Column(name = "ville_adr")
  private String ville;

  @Column(name = "ligne3_adr")
  private String ligne3;

  public Adresse(String ligne1, String ligne2, String cp, String ville,
      String ligne3) {
    super();
    this.ligne1 = ligne1;
    this.ligne2 = ligne2;
    this.cp = cp;
    this.ville = ville;
    this.ligne3 = ligne3;
  }

  public Adresse() {
  }

  //
  // getter et setter sur les champs de la classe
  //

}
```

L'annotation @Embeddable permet de préciser que la classe sera utilisée comme un component. Un tel élément n'a pas d'identifiant puisque celui utilisé sera celui de l'entité englobante.

Le fichier de configuration d'Hibernate définit les paramètres de connexion à la base de données et les deux classes qui encapsulent l'entité Personne et le component Adresse.

Exemple :

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
          "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
          "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="connection.url">jdbc:mysql://localhost/mabdd</property>
    <property name="connection.username">root</property>
    <property name="connection.password"></property>
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="transaction.factory_class">
org.hibernate.transaction.JDBCTransactionFactory</property>
    <property name="current_session_context_class">thread</property>
    <property name="hibernate.show_sql">true</property>
    <mapping class="fr.jmdoudoux.dej.hibernate.Personne"></mapping>
    <mapping class="fr.jmdoudoux.dej.hibernate.Adresse"></mapping>
  </session-factory>
</hibernate-configuration>
```

L'application de test est basique :

- créer une instance de type Adresse,
- créer une instance de type Personne,
- et sauvegarder la personne dans la base de données

Exemple :

```
package fr.jmdoudoux.dej.hibernate;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.AnnotationConfiguration;

public class TestHibernate16 {

  public static void main(String args[]) {
    SessionFactory sessionFactory = new AnnotationConfiguration().configure()
                                                                 .buildSessionFactory();
    Transaction transaction = null;
    int index = 7;

    Session session = sessionFactory.openSession();

    try {
      transaction = session.beginTransaction();

      Adresse adresse = new Adresse("ligne1_" + index, "ligne2_" + index, "cp_"
          + index, "ville" + index, "ligne3_" + index);
      Personne personne = new Personne("nom" + index,
                                       "prenom_" + index,
                                       null,
                                       adresse);

      session.save(personne);
      transaction.commit();

      System.out.println("La nouvelle personne a ete enregistree");

    } catch (Exception e) {
      transaction.rollback();
      e.printStackTrace();
    } finally {
      session.close();
    }
    sessionFactory.close();
  }
}
```

Lorsque la personne est enregistrée dans la base de données, son adresse l'est aussi dans la table Personne.

Résultat :

```
mysql> select * from personne;
+----+------+----------+----------+-----------+-----------+--------+----------
-+------------+
| Id | Nom  | Prenom   | DateNais | ligne1_adr | ligne2_adr | cp_adr | ville_adr
 | ligne3_adr |
+----+------+----------+----------+-----------+-----------+--------+----------
-+------------+
|  2 | nom7 | prenom_7 | NULL     | ligne1_7   | ligne2_7   | cp_7   | ville7
 | ligne3_7   |
+----+------+----------+----------+-----------+-----------+--------+----------
-+------------+
1 row in set (0.00 sec)
```

62.10.1.2. Le mapping avec une relation One-to-One avec clé primaire partagée

La relation repose sur deux tables distinctes : une pour les personnes et une pour les adresses.

Chacune des deux tables possède un identifiant qui est sa clé primaire. La particularité est que la valeur des clés primaires est partagée entre les deux tables. L'identifiant de la table adresse n'est pas auto incrémenté et correspond à la valeur de l'identifiant de la table personne.

Hibernate ne sait pas gérer seul ce type de mapping : il sera nécessaire de l'aider en utilisant un mapping bidirectionnel qui permettra à Hibernate de connaitre la valeur de l'identifiant de la personne à utiliser comme valeur de l'identifiant pour l'adresse afin que les deux correspondent.

La description de la table personne est la suivante :

Résultat :

```
mysql> desc personne;
+----------+--------------+------+-----+---------+----------------+
| Field    | Type         | Null | Key | Default | Extra          |
+----------+--------------+------+-----+---------+----------------+
| Id       | bigint(20)   |      | PRI | NULL    | auto_increment |
| Nom      | varchar(255) |      |     |         |                |
| Prenom   | varchar(255) |      |     |         |                |
| DateNais | date         | YES  |     | NULL    |                |
+----------+--------------+------+-----+---------+----------------+
4 rows in set (0.00 sec)
```

Le script DDL correspondant est le suivant :

Résultat :

```
CREATE TABLE `personne` (
  `Id` bigint(20) NOT NULL auto_increment,
  `Nom` varchar(255) NOT NULL default '',
  `Prenom` varchar(255) NOT NULL default '',
  `DateNais` date default NULL,
  PRIMARY KEY  (`Id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 AUTO_INCREMENT=0 ;
```

La description de la table adresse est la suivante :

Résultat :

```
mysql> desc adresse;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| id         | bigint(20)  |      | PRI | 0       |       |
| ligne1_adr | varchar(80) |      |     |         |       |
| ligne2_adr | varchar(80) | YES  |     | NULL    |       |
| cp_adr     | varchar(5)  | YES  |     | NULL    |       |
| ville_adr  | varchar(80) | YES  |     | NULL    |       |
| ligne3_adr | varchar(80) | YES  |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
6 rows in set (0.11 sec)
```

Le script DDL correspondant est le suivant :

Résultat :

```
CREATE TABLE `adresse` (
  `id` bigint(20) NOT NULL default '0',
  `ligne1_adr` varchar(80) NOT NULL default '',
  `ligne2_adr` varchar(80) default NULL,
  `cp_adr` varchar(5) default NULL,
  `ville_adr` varchar(80) default NULL,
  `ligne3_adr` varchar(80) default NULL,
  PRIMARY KEY  (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

62.10.1.2.1. La configuration dans le fichier de mapping

Les classes qui encapsulent les entités personne et adresse sont de simples POJO.

Exemple :

```
package fr.jmdoudoux.dej.hibernate;

public class Adresse {

  private Long id;
  private String ligne1;
  private String ligne2;
  private String cp;
  private String ville;
  private String ligne3;

  private Personne personne;

  public Adresse(String ligne1, String ligne2, String cp, String ville,
      String ligne3) {
    super();
    this.ligne1 = ligne1;
    this.ligne2 = ligne2;
    this.cp = cp;
    this.ville = ville;
    this.ligne3 = ligne3;
  }

  public Adresse() {
  }

  public Long getId() {
    return id;
  }

  // setter requis par Hibernate
  public void setId(Long id) {
    this.id = id;
  }
  public Personne getPersonne() {
    return personne;
  }

  public void setPersonne(Personne personne) {
    this.personne = personne;
  }

  //
  // getter et setter sur les autres champs de la classe
  //

}
```

Exemple :

```
package fr.jmdoudoux.dej.hibernate;

public class Personne {

  private Long id;
  private String nom;
  private String prenom;
  private String dateNais;
  private Adresse adresse;

  public Personne(String nom, String prenom, String dateNais, Adresse adresse) {
    this.nom = nom;
    this.prenom = prenom;
    this.dateNais = dateNais;
    this.adresse = adresse;
  }

  public Personne() {
  }

  public Long getId() {
    return id;
  }

  // Attention le setter est requis par Hibernate
  public void setId(Long id) {
    this.id = id;
  }

  public Adresse getAdresse() {
    return adresse;
  }

  public void setAdresse(Adresse adresse) {
    this.adresse = adresse;
  }

  //
  // getter et setter sur les autres champs de la classe
  //

  @Override
  public String toString() {
    return this.id + " : " + this.nom + " " + this.prenom;
  }

}
```

Le fichier de mapping de l'entité Personne (Personne.hbm.xml) contient un élément fils <one-to-one> pour définir la relation entre Personne et Adresse.

Exemple :

```
<?xml version="1.0"?>
```

```
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="fr.jmdoudoux.dej.hibernate.Personne" table="Personne">
    <id name="id" column="id">
      <generator class="increment" />
    </id>
    <property name="nom" column="Nom" />
    <property name="prenom" column="Prenom" />
    <property name="dateNais" column="DateNais" />
    <one-to-one name="adresse" class="fr.jmdoudoux.dej.hibernate.Adresse"
      cascade="save-update" />
  </class>
</hibernate-mapping>
```

Le fichier de mapping de l'entité Adresse (Adresse.hbm.xml) possède plusieurs caractéristiques liées au type de la relation utilisée avec l'entité Personne :

- Le champ identifiant id est défini avec un générateur de type foreign avec un paramètre qui précise que la valeur sera celle de l'identifiant du champ personne
- La relation inverse avec Personne est définie avec un tag <one-to-one> avec l'attribut constrained ayant la valeur true

Exemple :

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="fr.jmdoudoux.dej.hibernate.Adresse" table="Adresse">

    <id name="id" column="Id">
      <generator class="foreign">
        <param name="property">personne</param>
      </generator>
    </id>
    <property name="ligne1" column="ligne1_adr" />
    <property name="ligne2" column="ligne2_adr" />
    <property name="cp" column="cp_adr" />
    <property name="ville" column="ville_adr" />
    <property name="ligne3" column="ligne3_adr" />
    <one-to-one name="personne" class="fr.jmdoudoux.dej.hibernate.Personne"
      constrained="true" />
  </class>
</hibernate-mapping>
```

Le fichier de configuration d'Hibernate définit les paramètres de connexion à la base de données et les deux fichiers de mapping des entités.

Exemple :

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
          "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
          "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
    <property name="connection.url">jdbc:mysql://localhost/mabdd</property>
    <property name="connection.username">root</property>
    <property name="connection.password"></property>
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="transaction.factory_class">
org.hibernate.transaction.JDBCTransactionFactory</property>
    <property name="current_session_context_class">thread</property>
    <property name="hibernate.show_sql">true</property>
    <mapping resource="com/jmdoudoux/test/hibernate/Personne.hbm.xml"></mapping>
    <mapping resource="com/jmdoudoux/test/hibernate/Adresse.hbm.xml"></mapping>

</session-factory>
</hibernate-configuration>
```

L'application de test est basique :

- créer une instance de type Adresse,
- créer une instance de type Personne,
- assurer le bon fonctionnement du lien bidirectionnel en fournissant une référence de l'objet Personne à l'instance de l'adresse. Ceci doit être fait manuellement car Hibernate ne prend pas en charge automatiquement les liens bidirectionnels
- et sauvegarder la personne dans la base de données

Exemple :

```
package fr.jmdoudoux.dej.hibernate;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

public class TestHibernate11 {

  public static void main(String args[]) {
    SessionFactory sessionFactory = new Configuration().configure()
                                                       .buildSessionFactory();
    Transaction transaction = null;
    int index = 3;

    Session session = sessionFactory.openSession();

    try {
      transaction = session.beginTransaction();

      Adresse adresse = new Adresse("ligne1_" + index, "ligne2_" + index, "cp_"
          + index, "ville" + index, "ligne3_" + index);
      Personne personne = new Personne("nom" + index,
                                        "prenom_" + index,
                                        null,
                                        adresse);
      adresse.setPersonne(personne);

      session.save(personne);
      transaction.commit();

      System.out.println("La nouvelle personne a ete enregistree");

    } catch (Exception e) {
      transaction.rollback();
      e.printStackTrace();
    } finally {
      session.close();
    }

    sessionFactory.close();
  }
}
```

Lorsque la personne est enregistrée dans la base de données, son adresse l'est aussi avec comme identifiant la valeur de l'identifiant de la personne.

Résultat :

```
mysql> select * from personne;
+----+------+----------+----------+
| Id | Nom  | Prenom   | DateNais |
+----+------+----------+----------+
|  1 | nom3 | prenom_3 | NULL     |
+----+------+----------+----------+
1 row in set (0.00 sec)

mysql> select * from adresse;
+----+------------+------------+--------+-----------+------------+
| id | ligne1_adr | ligne2_adr | cp_adr | ville_adr | ligne3_adr |
+----+------------+------------+--------+-----------+------------+
|  1 | ligne1_3   | ligne2_3   | cp_3   | ville3    | ligne3_3   |
+----+------------+------------+--------+-----------+------------+
1 row in set (0.00 sec)
```

62.10.1.2.2. La configuration avec les annotations

Le POJO qui encapsule une personne a quelques particularités relatives à la relation avec l'adresse :

- Il possède un champ privé de type Adresse
- Le champ adresse est annoté avec les annotations @OneToOne et @PrimaryKeyJoin

Exemple :

```
package fr.jmdoudoux.dej.hibernate;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.Table;

@Entity
@Table(name = "personne")
public class Personne {

  @Id
  @GeneratedValue
  @Column(name = "Id")
  private Long id;

  @Column(name = "Nom")
  private String nom;

  @Column(name = "Prenom")
  private String prenom;

  @Column(name = "DateNais")
  private String dateNais;

  @OneToOne(cascade = CascadeType.ALL)
  @PrimaryKeyJoinColumn
  private Adresse adresse;

  public Personne(String nom, String prenom, String dateNais, Adresse adresse) {
    this.nom = nom;
    this.prenom = prenom;
    this.dateNais = dateNais;
    this.adresse = adresse;
  }

  public Personne() {
  }

  public Long getId() {
    return id;
  }

  public Adresse getAdresse() {
    return adresse;
  }

  public void setAdresse(Adresse adresse) {
    this.adresse = adresse;
  }

  //
  // getter et setter sur les autres champs de la classe
  //

  @Override
  public String toString() {
    return this.id + " : " + this.nom + " " + this.prenom;
  }

}
```

Si le champ adresse n'est pas annoté avec l'annotation @PrimaryKeyJoin, alors une exception de type org.hibernate.id.IdentifierGenerationException avec le message « null id generated for:class fr.jmdoudoux.dej.hibernate.Adresse » est levée à l'exécution.

Le POJO qui encapsule une adresse possède plusieurs particularités relatives à la relation avec la personne :

- Le champ identifiant de l'entité est annoté avec @GeneratedValue et @GenericGenerator pour indiquer à Hibernate que la valeur du champ id doit être obtenue à partie de la valeur du champ id de la propriété personne
- Un champ de type Personne permet une relation bidirectionnelle annotée avec @OneToOne
- Un setter sur le champ personne permettra d'assurer la cohésion de la relation par le développeur

Exemple :

```
package fr.jmdoudoux.dej.hibernate;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import javax.persistence.Table;

import org.hibernate.annotations.Parameter;

@Entity
@Table(name = "adresse")
public class Adresse {

  @Id
  @GeneratedValue(generator = "adresseGenerator")
  @org.hibernate.annotations.GenericGenerator(name = "adresseGenerator",
    strategy = "foreign", parameters = @Parameter(name = "property", value = "personne"))
  @Column(name = "id")
  private Long id;

  @Column(name = "ligne1_adr", nullable = false)
  private String ligne1;

  @Column(name = "ligne2_adr")
  private String ligne2;

  @Column(name = "cp_adr")
  private String cp;

  @Column(name = "ville_adr")
  private String ville;

  @Column(name = "ligne3_adr")
  private String ligne3;

  @OneToOne(mappedBy = "adresse")
  private Personne personne;
  public Adresse(String ligne1, String ligne2, String cp, String ville,
      String ligne3) {
    super();
    this.ligne1 = ligne1;
    this.ligne2 = ligne2;
    this.cp = cp;
    this.ville = ville;
    this.ligne3 = ligne3;
  }

  public Adresse() {
  }

  public Long getId() {
    return id;
  }

  public Personne getPersonne() {
    return personne;
  }

  public void setPersonne(Personne personne) {
    this.personne = personne;
  }

  //
  // getter et setter sur les autres champs de la classe
  //

}
```

Le fichier de configuration d'Hibernate définit les paramètres de connexion à la base de données et les deux classes qui encapsulent des entités.

Exemple :

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
          "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
          "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
    <property name="connection.url">jdbc:mysql://localhost/mabdd</property>
    <property name="connection.username">root</property>
    <property name="connection.password"></property>
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="transaction.factory_class">
org.hibernate.transaction.JDBCTransactionFactory</property>
    <property name="current_session_context_class">thread</property>
    <property name="hibernate.show_sql">true</property>
    <mapping class="fr.jmdoudoux.dej.hibernate.Personne"></mapping>
    <mapping class="fr.jmdoudoux.dej.hibernate.Adresse"></mapping>

</session-factory>
</hibernate-configuration>
```

L'application de test est basique :

- créer une instance de type Adresse,
- créer une instance de type Personne,
- assurer le bon fonctionnement du lien bidirectionnel en fournissant une référence de l'objet Personne à l'instance de l'adresse. Ceci doit être fait manuellement car Hibernate ne prend pas en charge automatiquement les liens bidirectionnels
- et sauvegarder la personne dans la base de données

Exemple :

```
package fr.jmdoudoux.dej.hibernate;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.AnnotationConfiguration;

public class TestHibernate10 {

  public static void main(String args[]) {
    SessionFactory sessionFactory = new AnnotationConfiguration().configure()
                                                                 .buildSessionFactory();
    Transaction transaction = null;
    int index = 2;

    Session session = sessionFactory.openSession();

    try {
      transaction = session.beginTransaction();

      Adresse adresse = new Adresse("ligne1_" + index, "ligne2_" + index, "cp_"
          + index, "ville" + index, "ligne3_" + index);
      Personne personne = new Personne("nom" + index,
                                       "prenom_" + index,
                                       null,
                                       adresse);
      adresse.setPersonne(personne);

      session.save(personne);
      transaction.commit();

      System.out.println("La nouvelle personne a ete enregistree");

    } catch (Exception e) {
      transaction.rollback();
      e.printStackTrace();
    } finally {
      session.close();
    }

    sessionFactory.close();
  }
}
```

Lorsque la personne est enregistrée dans la base de données, son adresse l'est aussi avec comme identifiant la valeur de l'identifiant de la personne.

Résultat :

```
mysql> select * from adresse;
+----+------------+------------+--------+-----------+------------+
| id | ligne1_adr | ligne2_adr | cp_adr | ville_adr | ligne3_adr |
+----+------------+------------+--------+-----------+------------+
|  3 | ligne1_1   | ligne2_1   | cp_1   | ville1    | ligne3_1   |
+----+------------+------------+--------+-----------+------------+
1 row in set (0.00 sec)

mysql> select * from personne;
+----+------+----------+----------+
| Id | Nom  | Prenom   | DateNais |
+----+------+----------+----------+
|  3 | nom1 | prenom_1 | NULL     |
+----+------+----------+----------+
1 row in set (0.00 sec)
```

Si la référence de l'instance de type Personne n'est pas fournie à l'instance de type Adresse alors une exception de type org.hibernate.id.IdentifierGenerationException avec le message « attempted to assign id from null one-to-one property [fr.jmdoudoux.dej.hibernate.Adresse.personne] » est levée à l'exécution.

Si le générateur d'identifiant n'est pas correctement configuré pour l'entité Adresse, alors une exception de type org.hibernate.id.IdentifierGenerationException avec le message « ids for this class must be manually assigned before calling save(): fr.jmdoudoux.dej.hibernate.Adresse » lors de l'exécution.

62.10.1.3. Le mapping avec une relation One-to-One avec clé étrangère

La relation repose sur deux tables distinctes : une pour les personnes et une pour les adresses

Chacune des deux tables possède son propre identifiant et la relation entre les deux tables est assurée par une clé étrangère de la table personne vers la table adresse.

Hibernate sait gérer seul ce type de mapping s'il est unidirectionnel.

La description de la table personne est la suivante :

Résultat :

```
mysql> desc personne;
+------------+--------------+------+-----+---------+----------------+
| Field      | Type         | Null | Key | Default | Extra          |
+------------+--------------+------+-----+---------+----------------+
| Id         | int(11)      |      | PRI | NULL    | auto_increment |
| Nom        | varchar(255) |      |     |         |                |
| Prenom     | varchar(255) |      |     |         |                |
| DateNais   | date         | YES  |     | NULL    |                |
| adresse_id | int(11)      |      |     | 0       |                |
+------------+--------------+------+-----+---------+----------------+
5 rows in set (0.00 sec)
```

Le script DDL correspondant est le suivant :

Résultat :

```
CREATE TABLE `personne` (
  `Id` int(11) NOT NULL auto_increment,
  `Nom` varchar(255) NOT NULL default '',
  `Prenom` varchar(255) NOT NULL default '',
  `DateNais` date default NULL,
  `adresse_id` int(11) NOT NULL default '0',
  PRIMARY KEY  (`Id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 AUTO_INCREMENT=0 ;
```

La description de la table adresse est la suivante :

Résultat :

```
mysql> desc adresse;
+------------+-------------+------+-----+---------+----------------+
| Field      | Type        | Null | Key | Default | Extra          |
+------------+-------------+------+-----+---------+----------------+
| id         | bigint(20)  |      | PRI | NULL    | auto_increment |
| ligne1_adr | varchar(80) |      |     |         |                |
| ligne2_adr | varchar(80) | YES  |     | NULL    |                |
| cp_adr     | varchar(5)  | YES  |     | NULL    |                |
| ville_adr  | varchar(80) | YES  |     | NULL    |                |
| ligne3_adr | varchar(80) | YES  |     | NULL    |                |
+------------+-------------+------+-----+---------+----------------+
6 rows in set (0.00 sec)
```

Le script DDL correspondant est le suivant :

Résultat :

```
CREATE TABLE `adresse` (
  `id` bigint(20) NOT NULL auto_increment,
  `ligne1_adr` varchar(80) NOT NULL default '',
  `ligne2_adr` varchar(80) default NULL,
  `cp_adr` varchar(5) default NULL,
  `ville_adr` varchar(80) default NULL,
  `ligne3_adr` varchar(80) default NULL,
  PRIMARY KEY  (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 AUTO_INCREMENT=0 ;
```

62.10.1.3.1. La configuration dans le fichier de mapping

Les classes qui encapsulent les entités personne et adresse sont de simples POJO.

Exemple :

```
package fr.jmdoudoux.dej.hibernate;

public class Personne {

  private Long id;
  private String nom;
  private String prenom;
  private String dateNais;
  private Adresse adresse;

  public Personne(String nom, String prenom, String dateNais, Adresse adresse) {
    this.nom = nom;
    this.prenom = prenom;
    this.dateNais = dateNais;
    this.adresse = adresse;
  }

  public Personne() {
  }

  public Long getId() {
    return id;
  }

  // Attention le setter est requis par Hibernate
  public void setId(Long id) {
    this.id = id;
  }

  public Adresse getAdresse() {
    return adresse;
  }

  public void setAdresse(Adresse adresse) {
    this.adresse = adresse;
  }

  //
  // getter et setter sur les autres champs de la classe
  //

  @Override
  public String toString() {
    return this.id + " : " + this.nom + " " + this.prenom;
  }

}
```

Exemple :

```
package fr.jmdoudoux.dej.hibernate;
public class Adresse {
  private Long id;
  private String ligne1;
  private String ligne2;
  private String cp;
  private String ville;
  private String ligne3;

  public Adresse(String ligne1, String ligne2, String cp, String ville, String ligne3) {
    super();
    this.ligne1 = ligne1;
    this.ligne2 = ligne2;
    this.cp = cp;
    this.ville = ville;
    this.ligne3 = ligne3;
  }

  public Adresse() {
  }

  public Long getId() {
    return id;
  }

  // setter requis par Hibernate
  public void setId(Long id) {
    this.id = id;
  }

  //
  // getter et setter sur les autres champs de la classe
  //

}
```

Le fichier de mapping de l'entité Personne (Personne.hbm.xml) contient un élément fils <many-to-one> pour définir la relation entre Personne et Adresse : l'unicité de la relation est cependant garantie par la valeur true de l'attribut unique. Il faut aussi utiliser une propriété column pour préciser la colonne qui va contenir la clé étrangère vers la table adresse.

Exemple :

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="fr.jmdoudoux.dej.hibernate.Personne" table="Personne">
    <id name="id" column="Id">
       <generator class="increment" />
    </id>
    <property name="nom" column="Nom" />
    <property name="prenom" column="Prenom" />
    <property name="dateNais" column="DateNais" />
    <many-to-one name="adresse" class="fr.jmdoudoux.dej.hibernate.Adresse"
        column="adresse_id" cascade="all" unique="true" />
  </class>
</hibernate-mapping>
```

Le fichier de mapping de l'entité Adresse (Adresse.hbm.xml) ne contient aucune particularité.

Exemple :

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="fr.jmdoudoux.dej.hibernate.Adresse" table="Adresse">
    <id name="id">
      <generator class="increment" />
    </id>
    <property name="ligne1" column="ligne1_adr" />
    <property name="ligne2" column="ligne2_adr" />
    <property name="cp" column="cp_adr" />
    <property name="ville" column="ville_adr" />
    <property name="ligne3" column="ligne3_adr" />
  </class>
</hibernate-mapping>
```

Le fichier de configuration d'Hibernate définit les paramètres de connexion à la base de données et les deux fichiers de mapping des entités.

Exemple :

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
          "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
          "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
    <property name="connection.url">jdbc:mysql://localhost/mabdd</property>
    <property name="connection.username">root</property>
    <property name="connection.password"></property>
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="transaction.factory_class">
org.hibernate.transaction.JDBCTransactionFactory</property>
    <property name="current_session_context_class">thread</property>
    <property name="hibernate.show_sql">true</property>
    <mapping resource="com/jmdoudoux/test/hibernate/Personne.hbm.xml"></mapping>
    <mapping resource="com/jmdoudoux/test/hibernate/Adresse.hbm.xml"></mapping>

</session-factory>
</hibernate-configuration>
```

L'application de test est basique :

- créer une instance de type Adresse,
- créer une instance de type Personne,
- et sauvegarder la personne dans la base de données

Exemple :

```
package fr.jmdoudoux.dej.hibernate;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

public class TestHibernate12 {

  public static void main(String args[]) {
    SessionFactory sessionFactory = new Configuration().configure()
                                                       .buildSessionFactory();
    Transaction transaction = null;
    int index = 4;

    Session session = sessionFactory.openSession();

    try {
      transaction = session.beginTransaction();

      Adresse adresse = new Adresse("ligne1_" + index, "ligne2_" + index, "cp_"
          + index, "ville" + index, "ligne3_" + index);
      Personne personne = new Personne("nom_" + index,
                                       "prenom_" + index,
                                       null,
                                       adresse);
      session.save(personne);
      transaction.commit();

      System.out.println("La nouvelle personne a ete enregistree");

    } catch (Exception e) {
      transaction.rollback();
      e.printStackTrace();
    } finally {
      session.close();
    }

    sessionFactory.close();
  }
}
```

Lorsque la personne est enregistrée dans la base de données, son adresse l'est aussi. Les nouvelles occurrences des tables Personne et Adresse possèdent chacun leur propre identifiant et celui de l'adresse est reporté dans le champ adresse_id de la table Personne.

Résultat :

```
mysql> select * from personne;
+----+-------+----------+----------+------------+
| Id | Nom   | Prenom   | DateNais | adresse_id |
+----+-------+----------+----------+------------+
|  1 | nom_4 | prenom_4 | NULL     |          8 |
+----+-------+----------+----------+------------+
1 row in set (0.82 sec)

mysql> select * from adresse;
+----+------------+------------+--------+-----------+------------+
| id | ligne1_adr | ligne2_adr | cp_adr | ville_adr | ligne3_adr |
+----+------------+------------+--------+-----------+------------+
|  8 | ligne1_4   | ligne2_4   | cp_4   | ville4    | ligne3_4   |
+----+------------+------------+--------+-----------+------------+
1 row in set (0.00 sec)
```

62.10.1.3.2. La configuration avec les annotations

Le POJO qui encapsule une personne a quelques particularités relatives à la relation avec l'adresse :

- Il possède un champ privé de type Adresse
- Le champ adresse est annoté avec les annotations @OneToOne et @JoinColumn

Exemple :

```
package fr.jmdoudoux.dej.hibernate;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;
import javax.persistence.Table;

@Entity
@Table(name = "personne")
public class Personne {

  @Id
  @GeneratedValue
  @Column(name = "Id")
  private Long id;
  @Column(name = "Nom")
  private String nom;

  @Column(name = "Prenom")
  private String prenom;

  @Column(name = "DateNais")
  private String dateNais;

  @OneToOne(cascade = CascadeType.ALL)
  @JoinColumn(name = "adresse_id")
  private Adresse adresse;

  public Personne(String nom, String prenom, String dateNais, Adresse adresse) {
    this.nom = nom;
    this.prenom = prenom;
    this.dateNais = dateNais;
    this.adresse = adresse;
  }

  public Personne() {
  }

  public Long getId() {
    return id;
  }

  public Adresse getAdresse() {
    return adresse;
  }

  public void setAdresse(Adresse adresse) {
    this.adresse = adresse;
  }

  //
  // getter et setter sur les autres champs de la classe
  //

  @Override
  public String toString() {
    return this.id + " : " + this.nom + " " + this.prenom;
  }
}
```

Le POJO qui encapsule une adresse ne possède aucune particularité relative à la relation avec la personne. Il est cependant possible d'ajouter au besoin une relation inverse d'adresse vers personne en ajoutant un champ Personne annoté avec l'annotation @one-to-one possèdant un attribut mappedBy qui possède comme valeur le nom du champ de l'adresse dans l'entité Personne.

Dans ce cas, la gestion de l'alimentation du champ personne est à la charge du développeur en utilisant le setter sur le champ personne.

L'identifiant de l'entité est annoté avec @GeneratedValue

Exemple :

```
package fr.jmdoudoux.dej.hibernate;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "adresse")
public class Adresse {

  @Id
  @GeneratedValue
  @Column(name = "id")
  private Long id;

  @Column(name = "ligne1_adr", nullable = false)
  private String ligne1;

  @Column(name = "ligne2_adr")
  private String ligne2;

  @Column(name = "cp_adr")
  private String cp;

  @Column(name = "ville_adr")
  private String ville;

  @Column(name = "ligne3_adr")
  private String ligne3;

  public Adresse(String ligne1, String ligne2, String cp, String ville,
      String ligne3) {
    super();
    this.ligne1 = ligne1;
    this.ligne2 = ligne2;
    this.cp = cp;
    this.ville = ville;
    this.ligne3 = ligne3;
  }

  public Adresse() {
  }

  public Long getId() {
    return id;
  }

  //
  // getter et setter sur les autres champs de la classe
  //

}
```

Le fichier de configuration d'Hibernate définit les paramètres de connexion à la base de données et les deux classes qui encapsulent des entités.

Exemple :

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
          "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
          "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
    <property name="connection.url">jdbc:mysql://localhost/mabdd</property>
    <property name="connection.username">root</property>
    <property name="connection.password"></property>
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="transaction.factory_class">
org.hibernate.transaction.JDBCTransactionFactory</property>
    <property name="current_session_context_class">thread</property>
    <property name="hibernate.show_sql">true</property>
    <mapping class="fr.jmdoudoux.dej.hibernate.Personne"></mapping>
    <mapping class="fr.jmdoudoux.dej.hibernate.Adresse"></mapping>

</session-factory>
</hibernate-configuration>
```

L'application de test est basique :

- créer une instance de type Adresse,
- créer une instance de type Personne,
- et sauvegarder la personne dans la base de données

Exemple :

```
package fr.jmdoudoux.dej.hibernate;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.AnnotationConfiguration;

public class TestHibernate14 {

  public static void main(String args[]) {
    SessionFactory sessionFactory = new AnnotationConfiguration().configure()
                                                                 .buildSessionFactory();
    Transaction transaction = null;
    int index = 5;

    Session session = sessionFactory.openSession();

    try {
      transaction = session.beginTransaction();

      Adresse adresse = new Adresse("ligne1_" + index, "ligne2_" + index, "cp_"
          + index, "ville" + index, "ligne3_" + index);
      Personne personne = new Personne("nom" + index,
                                       "prenom_" + index,
                                       null,
                                       adresse);
      session.save(personne);
      transaction.commit();

      System.out.println("La nouvelle personne a ete enregistree");

    } catch (Exception e) {
      transaction.rollback();
      e.printStackTrace();
    } finally {
      session.close();
    }

    sessionFactory.close();
  }
}
```

Lorsque la personne est enregistrée dans la base de données, son adresse l'est aussi. Chaque nouvelle occurrence de la table Personne et de la table Adresse possède son propre identifiant et celui de l'adresse est reporté dans le champ adresse_id de la table Personne.

Résultat :

```
mysql> select * from adresse;
+----+------------+------------+--------+-----------+------------+
| id | ligne1_adr | ligne2_adr | cp_adr | ville_adr | ligne3_adr |
+----+------------+------------+--------+-----------+------------+
| 10 | ligne1_5   | ligne2_5   | cp_5   | ville5    | ligne3_5   |
+----+------------+------------+--------+-----------+------------+
1 row in set (0.00 sec)

mysql> select * from personne;
+----+-------+----------+----------+------------+
| Id | Nom   | Prenom   | DateNais | adresse_id |
+----+-------+----------+----------+------------+
|  8 | nom5  | prenom_5 | NULL     |         10 |
+----+-------+----------+----------+------------+
1 row in set (0.00 sec)
```

La suite de cette section sera développée dans une version future de ce document

62.11. Le mapping de l'héritage de classes

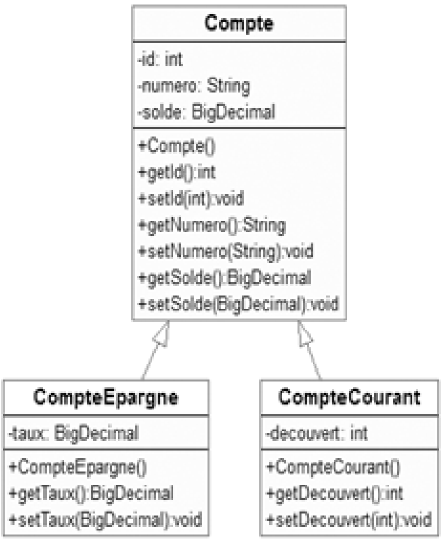
Hibernate propose un support des trois stratégies de base pour le mapping d'héritage de classes :

- une table par hiérarchie de classes (Table Per Hierarchy) : une seule table est utilisée. Elle possède une colonne supplémentaire qui sert de discriminant en précisant le type des données de la ligne
- une table par classe concrète (Table Per Concrete class) : à chaque classe concrète correspond une table. Les champs communs sont dupliqués dans chaque table fille
- une table par sous-classe (Table Per Subclass) : à chaque classe correspond une table. Les relations entre ces tables se font en utilisant des relations par clés étrangères. Il n'y a donc pas de colonnes dupliquées. Chaque classe est mappée à sa propre table.

Hibernate propose aussi une autre stratégie nommée une table par sous-classe avec discriminant.

Pour mapper une relation d'héritage dans le modèle relationnel, il faut donc choisir une stratégie qui sera adaptée en fonction des besoins.

Les exemples de cette section vont utiliser une hiérarchie de classes composée d'une classe mère Compte et deux classes filles CompteEpargne et CompteCourant.



La même classe est utilisée pour mettre en oeuvre des traitements avec les différentes stratégies de mapping.

Exemple :

```
package fr.jmdoudoux.dej.hibernate;

import java.math.BigDecimal;
import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import org.hibernate.service.ServiceRegistry;
import org.hibernate.service.ServiceRegistryBuilder;
import fr.jmdoudoux.dej.hibernate.entity.Compte;
import fr.jmdoudoux.dej.hibernate.entity.CompteCourant;
import fr.jmdoudoux.dej.hibernate.entity.CompteEpargne;
```

```
public class TestHibernate {

  private static SessionFactory sessionFactory = null;

  public static void main(String[] args) {
    try {
      Configuration hibernateConfig = new Configuration().configure();
      ServiceRegistry serviceRegistry = new ServiceRegistryBuilder()
          .applySettings(hibernateConfig.getProperties())
          .buildServiceRegistry();
      sessionFactory = hibernateConfig.buildSessionFactory(serviceRegistry);
      creerComptes();
      rechercherChaqueCompte();
      rechercherTousLesComptes();
      rechercheComptesPolymorphiques();
    } catch (Throwable ex) {
      ex.printStackTrace();
    } finally {
      sessionFactory.close();
    }
  }

  public static Transaction creerComptes() {
    Session session = sessionFactory.openSession();
    Transaction tx = null;
    try {
      Compte compte = new Compte();
      compte.setNumero("000012345000");
      compte.setSolde(BigDecimal.ZERO);
      CompteCourant compteCourant = new CompteCourant();
      compteCourant.setNumero("000012345010");
      compteCourant.setSolde(new BigDecimal("1200"));
      compteCourant.setDecouvert(2000);
      CompteEpargne compteEpargne = new CompteEpargne();
      compteEpargne.setNumero("000012345020");
      compteEpargne.setSolde(new BigDecimal(8000));
      compteEpargne.setTaux(new BigDecimal("2.10"));

      tx = session.beginTransaction();
      session.save(compte);
      session.save(compteCourant);
      session.save(compteEpargne);
      tx.commit();
    } catch (RuntimeException e) {
      try {
        tx.rollback();
      } catch (RuntimeException rbe) {
        rbe.printStackTrace();
      }
      throw e;
    } finally {
      if (session != null)
        session.close();
    }
    return tx;
  }

  public static void rechercherChaqueCompte() {
    Session session = sessionFactory.openSession();
    Transaction tx = null;
    try {
      tx = session.beginTransaction();
      System.out.println("Recherche d'un compte");
      Compte compte = (Compte) session.load(Compte.class, new Integer(1));
      System.out.println(compte);
      System.out.println();
      System.out.println("Recherche d'un compte courant");
      CompteCourant compteCourant = (CompteCourant) session.load(
          CompteCourant.class, new Integer(2));
      System.out.println(compteCourant);
      System.out.println();
      System.out.println("Recherche polymorphique d'un compte");
      compte = (Compte) session.load(Compte.class, new Integer(3));
      System.out.println(compte);
      System.out.println();
      tx.commit();
    } catch (RuntimeException e) {
      try {
        tx.rollback();
      } catch (RuntimeException rbe) {
        rbe.printStackTrace();
      }
      throw e;
    } finally {
      if (session != null)
        session.close();
    }
  }

  public static void rechercherTousLesComptes() {
    Session session = sessionFactory.openSession();
    Transaction tx = null;
    System.out.println("Recherche de tous les comptes");
    try {
      tx = session.beginTransaction();
      Query query = session.createQuery("from Compte");
      List<Compte> comptes = query.list();
      for (Compte compte : comptes) {
        System.out.println(compte);
      }
      System.out.println();
      tx.commit();
    } catch (RuntimeException e) {
      try {
        tx.rollback();
      } catch (RuntimeException rbe) {
        rbe.printStackTrace();
      }
      throw e;
    } finally {
      if (session != null)
        session.close();
    }
  }

  public static void rechercheComptesPolymorphiques() {
    Session session = sessionFactory.openSession();
    Transaction tx = null;
    System.out.println("Recherche polymorphique de comptes");
    try {
      tx = session.beginTransaction();
      Query query = session
          .createQuery("select c from Compte c where c.numero like :numero");
      query.setParameter("numero", "000012345%");
      List<Compte> comptes = query.list();
      for (Compte compte : comptes) {
        System.out.println(compte);
      }
      tx.commit();
    } catch (RuntimeException e) {
      try {
        tx.rollback();
      } catch (RuntimeException rbe) {
        rbe.printStackTrace();
      }
      throw e;
    } finally {
      if (session != null)
        session.close();
    }
    System.out.println();
  }
}
```

La version d'Hibernate utilisée est la 4.2.17.

62.11.1. XML

La déclaration du mapping dans un fichier XML utilise plusieurs tags en fonction de la stratégie utilisée : <class>, <union-subclass>, <subclass> et <joined-subclass>.

Il n'est pas possible de mixer l'utilisation du tag <subclass> et <joined-subclass> dans le même tag <class>.

Les exemples des sections suivantes utilisent trois classes qui sont les entités du modèle.

Exemple :

```
package fr.jmdoudoux.dej.hibernate.entity;

import java.math.BigDecimal;

public class Compte {
  protected int        id;
  protected String     numero;
  protected BigDecimal solde;

  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public String getNumero() {
    return numero;
  }

  public void setNumero(String numero) {
    this.numero = numero;
  }

  public BigDecimal getSolde() {
    return solde;
  }

  public void setSolde(BigDecimal solde) {
    this.solde = solde;
  }

  @Override
  public String toString() {
    return super.toString() + " [id=" + id + ", numero=" + numero + ", solde="
        + solde + "]";
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.hibernate.entity;

public class CompteCourant extends Compte {
  protected int decouvert;

  public int getDecouvert() {
    return decouvert;
  }

  public void setDecouvert(int decouvert) {
    this.decouvert = decouvert;
  }

  @Override
  public String toString() {
    return this.getClass().getName() + "@" + System.identityHashCode(this)
```

```
        + "CompteCourant [id=" + id + ", numero=" + numero + ", solde=" + solde
        + ", decouvert=" + decouvert + "]";
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.hibernate.entity;

import java.math.BigDecimal;

public class CompteEpargne extends Compte {
  protected BigDecimal taux;

  public BigDecimal getTaux() {
    return taux;
  }

  public void setTaux(BigDecimal taux) {
    this.taux = taux;
  }

  @Override
  public String toString() {
    return this.getClass().getName() + "@" + System.identityHashCode(this)
        + "CompteEpargne [ id=" + id + ", numero=" + numero + ", solde="
        + solde + ", taux=" + taux + "]";
  }
}
```

La configuration d'Hibernate est stockée dans le fichier hibernate.cfg.xml.

Exemple :

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost:3307/mabdd</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">root</property>
    <property name="hibernate.connection.pool_size">1</property>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</property>
    <property name="hibernate.current_session_context_class">thread</property>
    <!-- Cache de second niveau désactivé  -->
    <property name="hibernate.cache.provider_class">
      org.hibernate.cache.internal.NoCacheProvider</property>
    <!-- Afficher les requêtes SQL exécutées sur la sortie standard -->
    <property name="hibernate.show_sql">true</property>

    <property name="hbm2ddl.auto">create</property>

    <mapping resource="Compte.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

Remarque : La propriété hbm2dll.auto est initialisée avec la valeur create pour demander à Hibernate de recréer automatiquement les tables à chaque exécution.

62.11.1.1. XML, une table par hiérarchie de classes

Pour définir le mapping d'un héritage utilisant le modèle une table par hiérarchie de classe, il faut utiliser dans un même fichier hbm :

- un tag <class> pour le mapping classe mère
- un tag <subclass> pour le mapping de chaque classe fille

Le tag <class> possède l'attribut polymorphism qui peut prendre deux valeurs (implicit ou explicit) et permet de préciser le type de requête polymorphique.

L'attribut abstract permet de préciser si la classe mère de la hiérarchie est abstraite. Les valeurs possibles sont true et false

Le tag <discriminator> permet de préciser la colonne qui servira de discriminant.

Il possède plusieurs attributs :

column	Préciser le nom de la colonne dans la table, par défaut c'est le nom de la classe
formula	Une expression SQL qui sera exécutée pour obtenir la valeur. Optionnel
type	Préciser le type de la donnée. Optionnel : par défaut, String
not-null	true (par défaut) ou false
length	Préciser la taille de la colonne
force	true ou false. Optionnel : par défaut, false
insert	Préciser si la colonne doit être incluse dans les instructions insert : true ou false. Optionnel : par défaut, true

Le tag <subclass> possède plusieurs attributs :

entity-name	Optionnel
name	Nom pleinement qualifié de la classe de l'entité
proxy	Interface ou classe qui est utilisée pour la création des proxys. Optionnel
discriminator-value	Une valeur qui permet de distinguer chaque entité. Optionnel : par défaut c'est le nom de la classe
dynamic-update	true ou false (par défaut)
dynamic-insert	true ou false (par défaut)
select-before-update	true ou false (par défaut)
extends	Nom de la superclasse
lazy	Activation du lazy fetching : true ou false. Optionnel : par défaut true
abstract	true ou false
persister	Préciser un ClassPersister personnalisé. Optionnel
batch-size	Définir le nombre d'occurrences récupérées par lot. Optionnel
node	

Les tags <class> et <subclass> possèdent un attribut discriminator-value qui permet de préciser la valeur de la colonne discriminante qui sera utilisée pour la classe. Il n'est pas nécessaire de fournir une valeur à l'attribut discriminator-value pour les classes abstraites.

La colonne discriminator ne doit pas être définie dans la classe Java correspondante : c'est une colonne technique qui n'est utilisée que par Hibernate et la base de données.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
```

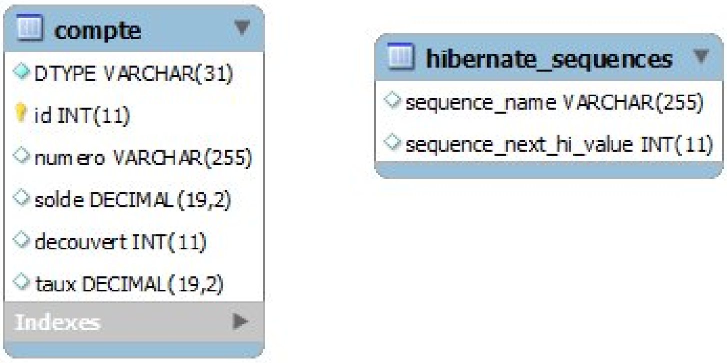
```
<hibernate-mapping>
  <class name="fr.jmdoudoux.dej.hibernate.entity.Compte"
    table="compte" discriminator-value="Compte">
    <id name="id" column="id" type="int" >
      <generator class="native" />
    </id>
    <discriminator column="DTYPE" type="string" />

    <property name="numero" column="numero" type="string" />
    <property name="solde" column="solde" type="big_decimal" />
    <subclass name="fr.jmdoudoux.dej.hibernate.entity.CompteCourant"
      discriminator-value="CompteCourant">
      <property name="decouvert" column="decouvert" type="int"/>
    </subclass>
    <subclass name="fr.jmdoudoux.dej.hibernate.entity.CompteEpargne"
      discriminator-value="CompteEpargne">
      <property name="taux" column="taux" type="big_decimal"/>
    </subclass>
  </class>
</hibernate-mapping>
```

Lorsque le type d'entité demandé à Hibernate est précisément identifié, il peut faire une requête qui ne récupère que les colonnes nécessaires à l'alimentation des propriétés de la classe.

Lorsque le type d'entité demandé à Hibernate est un supertype, celui-ci ne peut pas déterminer à l'avance les données nécessaires donc il doit obtenir toutes les données. A partir du discriminant, Hibernate peut créer la bonne instance et alimenter ses propriétés.

Résultat :

```
2015-02-23 23:18:53.650 INFO   [main]:org.hibernate.annotations.common.Version - HCANN000001:
Hibernate Commons Annotations {4.0.1.Final}
2015-02-23 23:18:53.663 INFO   [main]:org.hibernate.Version - HHH000412: Hibernate Core {4.1.
4.Final}
2015-02-23 23:18:53.665 INFO   [main]:org.hibernate.cfg.Environment - HHH000206: hibernate.
properties not found
2015-02-23 23:18:53.666 INFO   [main]:org.hibernate.cfg.Environment - HHH000021: Bytecode
provider name : javassist
2015-02-23 23:18:53.700 INFO   [main]:org.hibernate.cfg.Configuration - HHH000043:
Configuring from resource: /hibernate.cfg.xml
2015-02-23 23:18:53.700 INFO   [main]:org.hibernate.cfg.Configuration - HHH000040:
Configuration resource: /hibernate.cfg.xml
2015-02-23 23:18:53.753 INFO   [main]:org.hibernate.cfg.Configuration - HHH000221: Reading
mappings from resource: Compte.hbm.xml
2015-02-23 23:18:53.823 INFO   [main]:org.hibernate.cfg.Configuration - HHH000041: Configured
SessionFactory: null
2015-02-23 23:18:53.926 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000402: Using Hibernate built-in connection pool (not
for production use!)
2015-02-23 23:18:53.941 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000115: Hibernate connection pool size: 1
2015-02-23 23:18:53.941 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000006: Autocommit mode: false
2015-02-23 23:18:53.942 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000401: using driver [com.mysql.jdbc.Driver] at URL [
jdbc:mysql://localhost:3307/mabdd]
2015-02-23 23:18:53.942 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000046: Connection properties: {user=root, password=
****}
2015-02-23 23:18:54.340 INFO   [main]:org.hibernate.dialect.Dialect - HHH000400: Using
dialect: org.hibernate.dialect.MySQL5Dialect
2015-02-23 23:18:54.361 INFO   [main]:org.hibernate.engine.transaction.internal.
TransactionFactoryInitiator - HHH000399: Using default transaction strategy (direct JDBC
transactions)
2015-02-23 23:18:54.366 INFO   [main]:org.hibernate.hql.internal.ast.
ASTQueryTranslatorFactory - HHH000397: Using ASTQueryTranslatorFactory
2015-02-23 23:18:54.617 INFO     [main]:org.hibernate.tool.hbm2ddl.SchemaExport - HHH000227:
Running hbm2ddl schema export
Hibernate: drop table if exists compte
Hibernate: create table compte (id integer not null auto_increment, DTYPE varchar(255) not
null, numero varchar(255), solde decimal(19,2), decouvert integer, taux decimal(19,2), primary
key (id))
2015-02-23 23:18:54.707 INFO     [main]:org.hibernate.tool.hbm2ddl.SchemaExport - HHH000230:
Schema export complete
Hibernate: insert into compte (numero, solde, DTYPE) values (?, ?, 'Compte')
Hibernate: insert into compte (numero, solde, decouvert, DTYPE) values (?, ?, ?, '
CompteCourant')
Hibernate: insert into compte (numero, solde, taux, DTYPE) values (?, ?, ?, 'CompteEpargne')
Recherche d'un compte
Hibernate: select compte0_.id as id0_0_, compte0_.numero as numero0_0_, compte0_.solde as
solde0_0_, compte0_.decouvert as decouvert0_0_, compte0_.taux as taux0_0_, compte0_.DTYPE as
DTYPE0_0_ from compte compte0_ where compte0_.id=?
fr.jmdoudoux.dej.hibernate.entity.Compte@3c3daf [id=1, numero=000012345000, solde=0.00]

Recherche d'un compte courant
Hibernate: select comptecour0_.id as id0_0_, comptecour0_.numero as numero0_0_, comptecour0_.
solde as solde0_0_, comptecour0_.decouvert as decouvert0_0_ from compte comptecour0_ where
comptecour0_.id=? and comptecour0_.DTYPE='CompteCourant'
fr.jmdoudoux.dej.hibernate.entity.CompteCourant@26380823CompteCourant [id=2, numero=
000012345010, solde=1200.00, decouvert=2000]

Recherche polymorphique d'un compte
Hibernate: select compte0_.id as id0_0_, compte0_.numero as numero0_0_, compte0_.solde as
solde0_0_, compte0_.decouvert as decouvert0_0_, compte0_.taux as taux0_0_, compte0_.DTYPE as
DTYPE0_0_ from compte compte0_ where compte0_.id=?
fr.jmdoudoux.dej.hibernate.entity.CompteEpargne@641293CompteEpargne [ id=3, numero=
000012345020, solde=8000.00, taux=2.10]

Recherche de tous les comptes
Hibernate: select compte0_.id as id0_, compte0_.numero as numero0_, compte0_.solde as solde0_,
compte0_.decouvert as decouvert0_, compte0_.taux as taux0_, compte0_.DTYPE as DTYPE0_ from
compte compte0_
fr.jmdoudoux.dej.hibernate.entity.Compte@1c458a5 [id=1, numero=000012345000, solde=0.00]
fr.jmdoudoux.dej.hibernate.entity.CompteCourant@953079CompteCourant [id=2, numero=
000012345010, solde=1200.00, decouvert=2000]
fr.jmdoudoux.dej.hibernate.entity.CompteEpargne@12941261CompteEpargne [ id=3, numero=
000012345020, solde=8000.00, taux=2.10]

Recherche polymorphique de comptes
Hibernate: select compte0_.id as id0_, compte0_.numero as numero0_, compte0_.solde as solde0_,
compte0_.decouvert as decouvert0_, compte0_.taux as taux0_, compte0_.DTYPE as DTYPE0_ from
compte compte0_ where compte0_.numero like ?
fr.jmdoudoux.dej.hibernate.entity.Compte@4cb73c [id=1, numero=000012345000, solde=0.00]
fr.jmdoudoux.dej.hibernate.entity.CompteCourant@9557173CompteCourant [id=2, numero=
000012345010, solde=1200.00, decouvert=2000]
fr.jmdoudoux.dej.hibernate.entity.CompteEpargne@975425CompteEpargne [ id=3, numero=
000012345020, solde=8000.00, taux=2.10]
```



Résultat :

```
mysql> select * from compte;
+----+---------------+--------------+---------+-----------+------+
| id | DTYPE         | numero       | solde   | decouvert | taux |
+----+---------------+--------------+---------+-----------+------+
|  1 | Compte        | 000012345000 |    0.00 |      NULL | NULL |
|  2 | CompteCourant | 000012345010 | 1200.00 |      2000 | NULL |
|  3 | CompteEpargne | 000012345020 | 8000.00 |      NULL | 2.10 |
+----+---------------+--------------+---------+-----------+------+
3 rows in set (0.00 sec)
```

62.11.1.2. XML, une table par sous-classe

Le modèle relationnel et le modèle objet correspondant sont très proches : une table est utilisée pour chaque classe de la hiérarchie.

Les données d'une classe fille sont réparties dans la table de la classe mère pour les données héritées et dans la table de la classe fille pour ses données spécifiques. Cette stratégie ne requière pas de colonne de type discriminant.

La définition du mapping de la classe mère et de la classe fille se fait dans le même fichier .hbm. Pour définir le mapping d'un héritage utilisant le modèle une table par sous-classe, il faut utiliser dans un même fichier hbm :

- un tag <class> pour le mapping classe mère
- un tag <joined-subclass> pour le mapping de chaque classe fille

Le tag <joined-subclass> possède plusieurs attributs :

Nom	Rôle
entity-name	Par défaut, le nom de la classe
name	Nom pleinement qualifié de la classe de l'entité
proxy	Nom de la classe ou de l'interface utilisée pour créer des proxys lors des lectures lazy. Optionnel
table	Nom de la table. par défaut le nom de la classe
schema	Nom du schéma qui contiendra la table
catalog	Nom du catalogue qui contiendra la table
subselect	Requête SQL qui sera exécutée pour obtenir les données immuables de l'entité
dynamic-update	true ou false (par défaut)
dynamic-insert	true ou false (par défaut)
select-before-update	true ou false (par défaut)
extends	Nom de la superclasse pour une entité fille. Optionnel
lazy	Activation du lazy fetching : true ou false. Optionnel : par défaut true
abstract	true ou false : préciser si la classe est abstraite
persister	Nom de la classe de type ClassPersister à utiliser
check	Optionnel
batch-size	Préciser le nombre d'occurrences obtenues lors de la lecture par lots. Optionnel
node	

Lors de l'utilisation de cette stratégie, la superclasse a une table et chaque sous-classe a une table qui contient seulement ses propriétés non-héritées : les tables des classes filles ont une clé primaire est une clé étrangère vers la table de la superclasse. Hibernate va utiliser une relation de type 1-1 entre la clé primaire de la table mère et la clé étrangère de la table fille. Ceci implique une jointure entre les deux tables pour obtenir les données.

Le tag <id> du tag <class> permet de préciser la colonne qui est la clé primaire.

Le tag <key> fils du tag <joined-subclass> permet de préciser la clé étrangère qui sera utilisée pour réaliser la jointure avec la table mère sur sa clé primaire.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="fr.jmdoudoux.dej.hibernate.entity.Compte" table="compte" >
        <id name="id" column="id" type="int" >
            <generator class="native" />
        </id>
        <property name="numero" column="numero" type="string" />
        <property name="solde" column="solde" type="big_decimal" />
        <joined-subclass name="fr.jmdoudoux.dej.hibernate.entity.CompteCourant"
            table="compte_courant">
            <key column="id"/>
            <property name="decouvert" column="decouvert" type="int"/>
        </joined-subclass>
        <joined-subclass name="fr.jmdoudoux.dej.hibernate.entity.CompteEpargne"
            table="compte_epargne">
            <key column="id"/>
            <property name="taux" column="taux" type="big_decimal"/>
        </joined-subclass>
    </class>
</hibernate-mapping>
```

Pour obtenir une occurrence de l'entité, Hibernate va automatiquement effectuer une jointure entre la table mère et la table de la classe correspondante.

Si la requête demandée à Hibernate concerne le type d'une classe abstraite, Hibernate récupère de la base de données toutes les données des occurrences relatives à toutes les classes concrètes en effectuant une jointure sur toutes les tables. Hibernate est obligé de faire une jointure sur toutes les tables : il crée alors une instance de l'entité. Cependant, dans ce cas toutes les autres colonnes des autres tables sont lues.

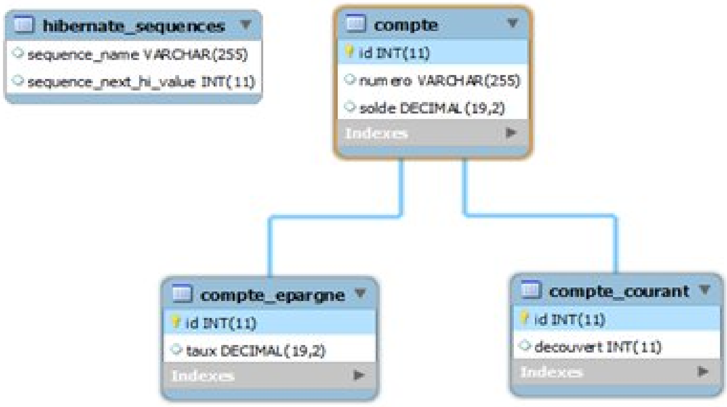
Si la requête demandée à Hibernate concerne le type d'une classe concrète, Hibernate effectue une jointure entre la table de la classe mère et la table de la classe fille correspondante.

Lors de l'enregistrement d'une nouvelle classe dans la base de données, Hibernate sa créer une nouvelle occurrence dans la table de la classe mère puis dans la table de la classe fille. La clé de la classe mère sera utilisée comme valeur dans la clé étrangère de la classe fille pour permettre à Hibernate de faire la jointure lors de la lecture des données.

Résultat :

```
2015-02-11 22:34:44.154 INFO   [main]:org.hibernate.annotations.common.Version - HCANN000001:
Hibernate Commons Annotations {4.0.1.Final}
2015-02-11 22:34:44.172 INFO   [main]:org.hibernate.Version - HHH000412: Hibernate Core {4.1.
4.Final}
2015-02-11 22:34:44.174 INFO   [main]:org.hibernate.cfg.Environment - HHH000206: hibernate.
properties not found
2015-02-11 22:34:44.175 INFO   [main]:org.hibernate.cfg.Environment - HHH000021: Bytecode
provider name : javassist
2015-02-11 22:34:44.211 INFO   [main]:org.hibernate.cfg.Configuration - HHH000043:
Configuring from resource: /hibernate.cfg.xml
2015-02-11 22:34:44.211 INFO   [main]:org.hibernate.cfg.Configuration - HHH000040:
Configuration resource: /hibernate.cfg.xml
2015-02-11 22:34:44.297 INFO   [main]:org.hibernate.cfg.Configuration - HHH000221: Reading
mappings from resource: Compte.hbm.xml
2015-02-11 22:34:44.366 INFO   [main]:org.hibernate.cfg.Configuration - HHH000041: Configured
SessionFactory: null
2015-02-11 22:34:44.475 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000402: Using Hibernate built-in connection pool (not
for production use!)
2015-02-11 22:34:44.482 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000115: Hibernate connection pool size: 1
2015-02-11 22:34:44.483 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000006: Autocommit mode: false
2015-02-11 22:34:44.483 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000401: using driver [com.mysql.jdbc.Driver] at URL [
jdbc:mysql://localhost:3307/mabdd]
```

```
2015-02-11 22:34:44.485 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000046: Connection properties: {user=root, password=
****}
2015-02-11 22:34:44.842 INFO   [main]:org.hibernate.dialect.Dialect - HHH000400: Using
dialect: org.hibernate.dialect.MySQL5Dialect
2015-02-11 22:34:44.870 INFO   [main]:org.hibernate.engine.transaction.internal.
TransactionFactoryInitiator - HHH000399: Using default transaction strategy (direct JDBC
transactions)
2015-02-11 22:34:44.874 INFO   [main]:org.hibernate.hql.internal.ast.
ASTQueryTranslatorFactory - HHH000397: Using ASTQueryTranslatorFactory
2015-02-11 22:34:45.100 INFO   [main]:org.hibernate.tool.hbm2ddl.SchemaExport - HHH000227:
Running hbm2ddl schema export
Hibernate: alter table compte_courant drop foreign key FK2435FDBF9CCD1BB4
2015-02-11 22:34:45.106 ERROR  [main]:org.hibernate.tool.hbm2ddl.SchemaExport - HHH000389:
Unsuccessful: alter table compte_courant drop foreign key FK2435FDBF9CCD1BB4
2015-02-11 22:34:45.106 ERROR  [main]:org.hibernate.tool.hbm2ddl.SchemaExport - Table 'mabdd.
compte_courant' doesn't exist
Hibernate: alter table compte_epargne drop foreign key FK8E9D8D439CCD1BB4
2015-02-11 22:34:45.107 ERROR  [main]:org.hibernate.tool.hbm2ddl.SchemaExport - HHH000389:
Unsuccessful: alter table compte_epargne drop foreign key FK8E9D8D439CCD1BB4
2015-02-11 22:34:45.107 ERROR  [main]:org.hibernate.tool.hbm2ddl.SchemaExport - Table 'mabdd.
compte_epargne' doesn't exist
Hibernate: drop table if exists compte
Hibernate: drop table if exists compte_courant
Hibernate: drop table if exists compte_epargne
Hibernate: create table compte (id integer not null auto_increment, numero varchar(255), solde
decimal(19,2), primary key (id))
Hibernate: create table compte_courant (id integer not null, decouvert integer, primary key (
id))
Hibernate: create table compte_epargne (id integer not null, taux decimal(19,2), primary key (
id))
Hibernate: alter table compte_courant add index FK2435FDBF9CCD1BB4 (id), add constraint
FK2435FDBF9CCD1BB4 foreign key (id) references compte (id)
Hibernate: alter table compte_epargne add index FK8E9D8D439CCD1BB4 (id), add constraint
FK8E9D8D439CCD1BB4 foreign key (id) references compte (id)
2015-02-11 22:34:45.189 INFO     [main]:org.hibernate.tool.hbm2ddl.SchemaExport - HHH000230:
Schema export complete
Hibernate: insert into compte (numero, solde) values (?, ?)
Hibernate: insert into compte (numero, solde) values (?, ?)
Hibernate: insert into compte_courant (decouvert, id) values (?, ?)
Hibernate: insert into compte (numero, solde) values (?, ?)
Hibernate: insert into compte_epargne (taux, id) values (?, ?)
Recherche d'un compte
Hibernate: select compte0_.id as id0_0_, compte0_.numero as numero0_0_, compte0_.solde as
solde0_0_, compte0_1_.decouvert as decouvert1_0_, compte0_2_.taux as taux2_0_, case when
compte0_1_.id is not null then 1 when compte0_2_.id is not null then 2 when compte0_.id is not
null then 0 end as clazz_0_ from compte compte0_ left outer join compte_courant compte0_1_ on
compte0_.id=compte0_1_.id left outer join compte_epargne compte0_2_ on compte0_.id=compte0_2_.
id where compte0_.id=?
fr.jmdoudoux.dej.hibernate.entity.Compte@e2ecb3 [id=1, numero=000012345000, solde=0.00]

Recherche d'un compte courant
Hibernate: select comptecour0_.id as id0_0_, comptecour0_1_.numero as numero0_0_,
comptecour0_1_.solde as solde0_0_, comptecour0_.decouvert as decouvert1_0_ from compte_courant
comptecour0_ inner join compte comptecour0_1_ on comptecour0_.id=comptecour0_1_.id where
comptecour0_.id=?
fr.jmdoudoux.dej.hibernate.entity.CompteCourant@1972965CompteCourant [id=2, numero=
000012345010, solde=1200.00, decouvert=2000]

Recherche polymorphique d'un compte
Hibernate: select compte0_.id as id0_0_, compte0_.numero as numero0_0_, compte0_.solde as
solde0_0_, compte0_1_.decouvert as decouvert1_0_, compte0_2_.taux as taux2_0_, case when
compte0_1_.id is not null then 1 when compte0_2_.id is not null then 2 when compte0_.id is not
null then 0 end as clazz_0_ from compte compte0_ left outer join compte_courant compte0_1_ on
compte0_.id=compte0_1_.id left outer join compte_epargne compte0_2_ on compte0_.id=compte0_2_.
id where compte0_.id=?
fr.jmdoudoux.dej.hibernate.entity.CompteEpargne@32800355CompteEpargne [ id=3, numero=
000012345020, solde=8000.00, taux=2.10]

Recherche de tous les comptes
Hibernate: select compte0_.id as id0_, compte0_.numero as numero0_, compte0_.solde as solde0_,
compte0_1_.decouvert as decouvert1_, compte0_2_.taux as taux2_, case when compte0_1_.id is not
null then 1 when compte0_2_.id is not null then 2 when compte0_.id is not null then 0 end as
clazz_ from compte compte0_ left outer join compte_courant compte0_1_ on compte0_.id=
compte0_1_.id left outer join compte_epargne compte0_2_ on compte0_.id=compte0_2_.id
fr.jmdoudoux.dej.hibernate.entity.CompteCourant@2473102CompteCourant [id=2, numero=
000012345010, solde=1200.00, decouvert=2000]
fr.jmdoudoux.dej.hibernate.entity.CompteEpargne@21374993CompteEpargne [ id=3, numero=
000012345020, solde=8000.00, taux=2.10]
fr.jmdoudoux.dej.hibernate.entity.Compte@f08a49 [id=1, numero=000012345000, solde=0.00]

Recherche polymorphique de comptes
Hibernate: select compte0_.id as id0_, compte0_.numero as numero0_, compte0_.solde as solde0_,
compte0_1_.decouvert as decouvert1_, compte0_2_.taux as taux2_, case when compte0_1_.id is not
null then 1 when compte0_2_.id is not null then 2 when compte0_.id is not null then 0 end as
clazz_ from compte compte0_ left outer join compte_courant compte0_1_ on compte0_.id=
compte0_1_.id left outer join compte_epargne compte0_2_ on compte0_.id=compte0_2_.id where
compte0_.numero like ?
fr.jmdoudoux.dej.hibernate.entity.CompteCourant@5917204CompteCourant [id=2, numero=
000012345010, solde=1200.00, decouvert=2000]
fr.jmdoudoux.dej.hibernate.entity.CompteEpargne@28671960CompteEpargne [ id=3, numero=
000012345020, solde=8000.00, taux=2.10]
fr.jmdoudoux.dej.hibernate.entity.Compte@ad3eb5 [id=1, numero=000012345000, solde=0.00]
```



Résultat :

```
mysql> select * from compte;
+----+--------------+---------+
| id | numero       | solde   |
+----+--------------+---------+
|  1 | 000012345000 |    0.00 |
|  2 | 000012345010 | 1200.00 |
|  3 | 000012345020 | 8000.00 |
+----+--------------+---------+
3 rows in set (0.00 sec)
mysql> select * from compte_courant;
+----+-----------+
| id | decouvert |
+----+-----------+
|  2 |      2000 |
+----+-----------+
1 row in set (0.00 sec)
mysql> select * from compte_epargne;
+----+------+
| id | taux |
+----+------+
|  3 | 2.10 |
+----+------+
1 row in set (0.00 sec)
```

Remarque : l'identifiant de l'occurrence de la table de la classe mère et celui de l'occurrence de la table de la classe fille doivent être identiques pour permettre de réaliser la jointure.

62.11.1.3. XML, une table par classe concrète

Dans cette stratégie, chaque classe concrète est mappée sur une table dans la base de données. Toutes les propriétés de la classe sont partagées par les classes filles : ceci implique une duplication de ces champs dans les tables des classes filles.

La classe mère est définie grâce à un tag <class>.

Chaque classe fille est définie grâce à un tag fils <union-subclass>. Il possède plusieurs attributs :

- name : nom pleinement qualifié de la classe
- table : nom de la table dans la base de données

Les colonnes de la table de la classe mère sont aussi présentes dans les tables des classes filles. Chaque propriété propre à une classe fille doit être mappée dans le tag <union-subclass> correspondant.

Le tag <union-subclass> possède plusieurs attributs :

Attribut	Rôle
entity-name	Par défaut, le nom de la classe
name	Nom pleinement qualifié de la classe
proxy	Interface ou classe qui est utilisée pour la création des proxys. Optionnel
table	Nom de la table. Par défaut : le nom de la classe
schema	Nom du schéma qui contiendra la table
catalog	Nom du catalogue qui contiendra la table
subselect	Requête SQL qui sera exécutée pour obtenir les données immuables de l'entité
dynamic-update	true ou false (par défaut)
dynamic-insert	true ou false (par défaut)
select-before-update	true ou false (par défaut)
extends	Nom de la super-classe pour une entité fille. Optionnel
lazy	true ou false
abstract	true ou false
persister	Préciser un ClassPersister personnalisé. Optionnel
check	Optionnel
batch-size	Définir le nombre d'occurrences récupérées par lot. Optionnel
node	

Le champ identifiant de la classe mère est partagé avec les classes filles : ce champ n'est d'ailleurs pas mappé dans les classes filles.

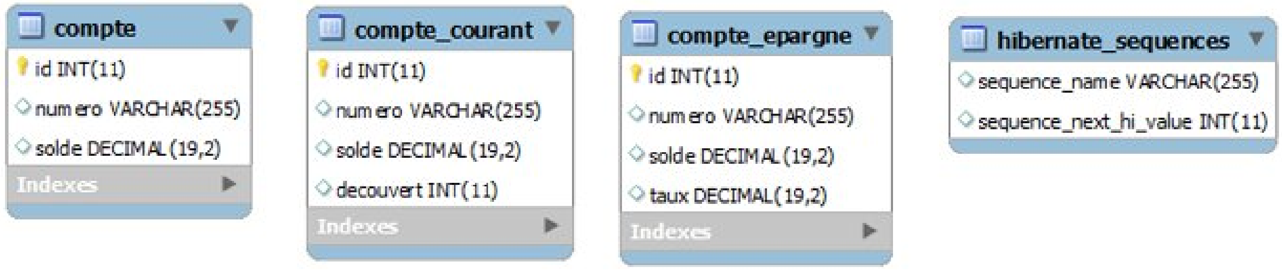
Remarque : il n'est pas possible d'utiliser la stratégie de gestion native des identifiants.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name="fr.jmdoudoux.dej.hibernate.entity.Compte"
    table="compte" >
    <id name="id" column="id" type="int" >
      <generator class="increment" />
    </id>
    <property name="numero" column="numero" type="string" />
    <property name="solde" column="solde" type="big_decimal" />
    <union-subclass name="fr.jmdoudoux.dej.hibernate.entity.CompteCourant"
      table="compte_courant">
      <property name="decouvert" column="decouvert" type="int"/>
    </union-subclass>
    <union-subclass name="fr.jmdoudoux.dej.hibernate.entity.CompteEpargne"
      table="compte_epargne">
      <property name="taux" column="taux" type="big_decimal"/>
    </union-subclass>
  </class>
</hibernate-mapping>
```

Résultat :

```
2015-02-11 22:37:45.178 INFO   [main]:org.hibernate.annotations.common.Version - HCANN000001:
Hibernate Commons Annotations {4.0.1.Final}
2015-02-11 22:37:45.194 INFO   [main]:org.hibernate.Version - HHH000412: Hibernate Core {4.1.
4.Final}
2015-02-11 22:37:45.194 INFO   [main]:org.hibernate.cfg.Environment - HHH000206: hibernate.
properties not found
2015-02-11 22:37:45.194 INFO   [main]:org.hibernate.cfg.Environment - HHH000021: Bytecode
provider name : javassist
2015-02-11 22:37:45.225 INFO   [main]:org.hibernate.cfg.Configuration - HHH000043:
Configuring from resource: /hibernate.cfg.xml
2015-02-11 22:37:45.225 INFO   [main]:org.hibernate.cfg.Configuration - HHH000040:
Configuration resource: /hibernate.cfg.xml
2015-02-11 22:37:45.272 INFO   [main]:org.hibernate.cfg.Configuration - HHH000221: Reading
mappings from resource: Compte.hbm.xml
2015-02-11 22:37:45.319 INFO   [main]:org.hibernate.cfg.Configuration - HHH000041: Configured
SessionFactory: null
2015-02-11 22:37:45.443 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000402: Using Hibernate built-in connection pool (not
for production use!)
2015-02-11 22:37:45.443 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000115: Hibernate connection pool size: 1
2015-02-11 22:37:45.443 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000006: Autocommit mode: false
2015-02-11 22:37:45.443 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000401: using driver [com.mysql.jdbc.Driver] at URL [
jdbc:mysql://localhost:3307/mabdd]
2015-02-11 22:37:45.459 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000046: Connection properties: {user=root, password=
****}
2015-02-11 22:37:45.787 INFO   [main]:org.hibernate.dialect.Dialect - HHH000400: Using
dialect: org.hibernate.dialect.MySQL5Dialect
2015-02-11 22:37:45.803 INFO   [main]:org.hibernate.engine.transaction.internal.
TransactionFactoryInitiator - HHH000399: Using default transaction strategy (direct JDBC
transactions)
2015-02-11 22:37:45.803 INFO   [main]:org.hibernate.hql.internal.ast.
ASTQueryTranslatorFactory - HHH000397: Using ASTQueryTranslatorFactory
2015-02-11 22:37:46.014 INFO   [main]:org.hibernate.tool.hbm2ddl.SchemaExport - HHH000227:
Running hbm2ddl schema export
Hibernate: drop table if exists compte
Hibernate: drop table if exists compte_courant
Hibernate: drop table if exists compte_epargne
Hibernate: create table compte (id integer not null, numero varchar(255), solde decimal(19,2),
primary key (id))
Hibernate: create table compte_courant (id integer not null, numero varchar(255), solde
decimal(19,2), decouvert integer, primary key (id))
Hibernate: create table compte_epargne (id integer not null, numero varchar(255), solde
decimal(19,2), taux decimal(19,2), primary key (id))
2015-02-11 22:37:46.046 INFO   [main]:org.hibernate.tool.hbm2ddl.SchemaExport - HHH000230:
Schema export complete
Hibernate: select max(ids_.id) from ( select id from compte_courant union select id from
compte union select id from compte_epargne ) ids_
Hibernate: insert into compte (numero, solde, id) values (?, ?, ?)
Hibernate: insert into compte_courant (numero, solde, decouvert, id) values (?, ?, ?, ?)
Hibernate: insert into compte_epargne (numero, solde, taux, id) values (?, ?, ?, ?)
Recherche d'un compte
Hibernate: select compte0_.id as id0_0_, compte0_.numero as numero0_0_, compte0_.solde as
solde0_0_, compte0_.decouvert as decouvert1_0_, compte0_.taux as taux2_0_, compte0_.clazz_ as
clazz_0_ from ( select id, numero, solde, null as decouvert, null as taux, 0 as clazz_ from
compte union select id, numero, solde, decouvert, null as taux, 1 as clazz_ from
compte_courant union select id, numero, solde, null as decouvert, taux, 2 as clazz_ from
compte_epargne ) compte0_ where compte0_.id=?
fr.jmdoudoux.dej.hibernate.entity.Compte@2c5b4e [id=1, numero=000012345000, solde=0.00]

Recherche d'un compte courant
Hibernate: select comptecour0_.id as id0_0_, comptecour0_.numero as numero0_0_, comptecour0_.
solde as solde0_0_, comptecour0_.decouvert as decouvert1_0_ from compte_courant comptecour0_
where comptecour0_.id=?
fr.jmdoudoux.dej.hibernate.entity.CompteCourant@21999623CompteCourant [id=2, numero=
000012345010, solde=1200.00, decouvert=2000]

Recherche polymorphique d'un compte
Hibernate: select compte0_.id as id0_0_, compte0_.numero as numero0_0_, compte0_.solde as
solde0_0_, compte0_.decouvert as decouvert1_0_, compte0_.taux as taux2_0_, compte0_.clazz_ as
clazz_0_ from ( select id, numero, solde, null as decouvert, null as taux, 0 as clazz_ from
compte union select id, numero, solde, decouvert, null as taux, 1 as clazz_ from
compte_courant union select id, numero, solde, null as decouvert, taux, 2 as clazz_ from
compte_epargne ) compte0_ where compte0_.id=?
fr.jmdoudoux.dej.hibernate.entity.CompteEpargne@7659203CompteEpargne [ id=3, numero=
000012345020, solde=8000.00, taux=2.10]

Recherche de tous les comptes
Hibernate: select compte0_.id as id0_, compte0_.numero as numero0_, compte0_.solde as solde0_,
compte0_.decouvert as decouvert1_, compte0_.taux as taux2_, compte0_.clazz_ as clazz_ from (
select id, numero, solde, null as decouvert, null as taux, 0 as clazz_ from compte union
select id, numero, solde, decouvert, null as taux, 1 as clazz_ from compte_courant union
select id, numero, solde, null as decouvert, taux, 2 as clazz_ from compte_epargne ) compte0_
fr.jmdoudoux.dej.hibernate.entity.Compte@12a81c9 [id=1, numero=000012345000, solde=0.00]
fr.jmdoudoux.dej.hibernate.entity.CompteCourant@24930042CompteCourant [id=2, numero=
000012345010, solde=1200.00, decouvert=2000]
fr.jmdoudoux.dej.hibernate.entity.CompteEpargne@5027644CompteEpargne [ id=3, numero=
000012345020, solde=8000.00, taux=2.10]

Recherche polymorphique de comptes
Hibernate: select compte0_.id as id0_, compte0_.numero as numero0_, compte0_.solde as solde0_,
compte0_.decouvert as decouvert1_, compte0_.taux as taux2_, compte0_.clazz_ as clazz_ from (
select id, numero, solde, null as decouvert, null as taux, 0 as clazz_ from compte union
select id, numero, solde, decouvert, null as taux, 1 as clazz_ from compte_courant union
select id, numero, solde, null as decouvert, taux, 2 as clazz_ from compte_epargne ) compte0_
where compte0_.numero like ?
fr.jmdoudoux.dej.hibernate.entity.Compte@1a93ff1 [id=1, numero=000012345000, solde=0.00]
fr.jmdoudoux.dej.hibernate.entity.CompteCourant@10814140CompteCourant [id=2, numero=
000012345010, solde=1200.00, decouvert=2000]
fr.jmdoudoux.dej.hibernate.entity.CompteEpargne@12935409CompteEpargne [ id=3, numero=
000012345020, solde=8000.00, taux=2.10]
```



compte
id INT(11)
numero VARCHAR(255)
solde DECIMAL(19,2)
Indexes

compte_courant
id INT(11)
numero VARCHAR(255)
solde DECIMAL(19,2)
decouvert INT(11)
Indexes

compte_epargne
id INT(11)
numero VARCHAR(255)
solde DECIMAL(19,2)
taux DECIMAL(19,2)
Indexes

hibernate_sequences
sequence_name VARCHAR(255)
sequence_next_hi_value INT(11)

Résultat :

```
mysql> select * from compte;
+----+--------------+---------+
| id | numero       | solde   |
+----+--------------+---------+
|  1 | 000012345000 |    0.00 |
|  2 | 000012345010 | 1200.00 |
|  3 | 000012345020 | 8000.00 |
|  4 | 000012345000 |    0.00 |
+----+--------------+---------+
4 rows in set (0.00 sec)
mysql> select * from compte_courant;
```

```
+----+--------------+---------+-----------+
| id | numero       | solde   | decouvert |
+----+--------------+---------+-----------+
|  5 | 000012345010 | 1200.00 |      2000 |
+----+--------------+---------+-----------+
1 row in set (0.00 sec)
mysql> select * from compte_epargne;
+----+--------------+---------+------+
| id | numero       | solde   | taux |
+----+--------------+---------+------+
|  6 | 000012345020 | 8000.00 | 2.10 |
+----+--------------+---------+------+
1 row in set (0.00 sec)
```

Il est important que deux tables de classes filles d'une même hiérarchie ne possèdent pas le même identifiant. Hibernate pourrait alors renvoyer plusieurs objets pour un même identifiant en effectuant une union sur les différentes tables. C'est la raison pour laquelle il n'est pas possible d'utiliser la stratégie native pour la génération des identifiants.

Exemple :

```
       <id name="id" column="id" type="int" >
           <generator class="native"/>
        </id>
```

Résultat :

```
org.hibernate.MappingException: Cannot use identity column key generation with <union-subclass>
mapping for: fr.jmdoudoux.dej.hibernate.entity.CompteEpargne
  at org.hibernate.persister.entity.UnionSubclassEntityPersister.<init>(
  UnionSubclassEntityPersister.java:93)
  at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
  at sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:
  57)
  at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImp
  l.java:45)
  at java.lang.reflect.Constructor.newInstance(Constructor.java:525)
  at org.hibernate.persister.internal.PersisterFactoryImpl.create(PersisterFactoryImpl.java:
  163)
  at org.hibernate.persister.internal.PersisterFactoryImpl.createEntityPersister(
  PersisterFactoryImpl.java:135)
  at org.hibernate.internal.SessionFactoryImpl.<init>(SessionFactoryImpl.java:386)
  at org.hibernate.cfg.Configuration.buildSessionFactory(Configuration.java:1744)
  at fr.jmdoudoux.dej.hibernate.TestHibernate.main(TestHibernate.java:27)
Exception in thread "main" java.lang.NullPointerException
  at fr.jmdoudoux.dej.hibernate.TestHibernate.main(TestHibernate.java:54)
```

62.11.1.4. XML, une table par sous-classe avec discriminant

La mise en oeuvre de cette stratégie requiert la définition d'un champ qui sera le discriminant dans la classe mère.

Chaque classe fille est mappée en utilisant un tag <subclass> avec deux propriétés :

- name : précise le nom pleinement qualifié de la classe
- discriminator_value : précise la valeur qui sera utilisée dans la colonne discriminator

Il faut lui ajouter un tag fils <join> pour définir le mapping de la table avec plusieurs propriétés :

- table permet de préciser le nom de la table.
- fetch permet de préciser la stratégie utilisée pour récupérer les données. La valeur select permet de demander à Hibernate de ne pas faire de jointure externe sur la table lors d'une lecture sur la classe mère

Le tag fils <key> permet de préciser le champ qui sera la clé de la table. Il possède plusieurs attributs :

Nom	Rôle

column	Le nom de la colonne qui est la clé étrangère. Optionnel
on-delete	noaction ou cascade. Optionnel, par défaut noaction
property-ref	Préciser un nom de propriété pour indiquer que la clé étrangère ne fait pas référence à la clé primaire de la table mère. Optionnel
not-null	Préciser si la clé étrangère peut être null : true ou false. Optionnel
update	Préciser si la clé étrangère doit être mise à jour : true ou false. Optionnel
unique	Préciser si la clé étrangère doit avoir une contrainte d'unicité : true ou false. Optionnel

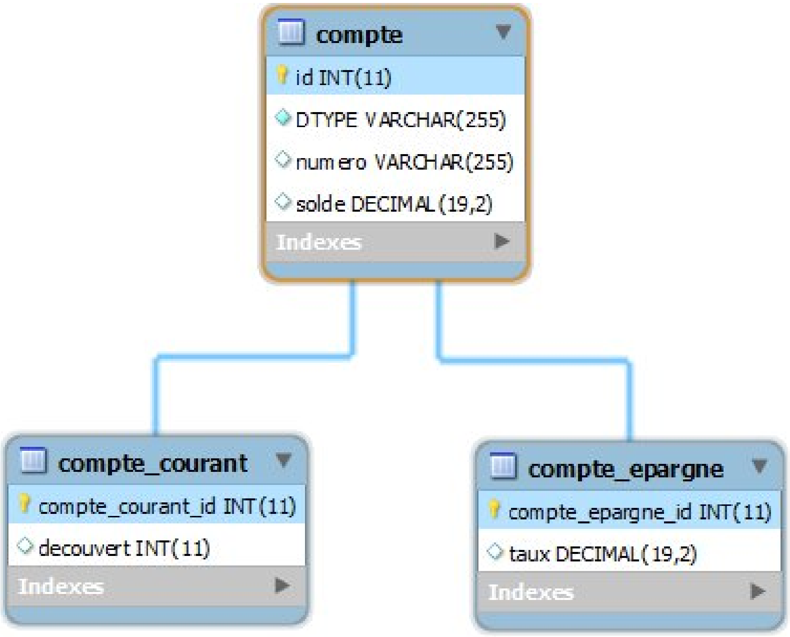
Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name="fr.jmdoudoux.dej.hibernate.entity.Compte"
    table="compte" discriminator-value="Compte">
    <id name="id" column="id" type="int" >
      <generator class="native" />
    </id>
    <discriminator column="DTYPE" type="string" />

    <property name="numero" column="numero" type="string" />
    <property name="solde" column="solde" type="big_decimal" />
    <subclass name="fr.jmdoudoux.dej.hibernate.entity.CompteCourant"
      discriminator-value="CompteCourant">
      <join table="compte_courant">
        <key column="compte_courant_id" />
        <property name="decouvert" column="decouvert" type="int"/>
      </join>
    </subclass>
    <subclass name="fr.jmdoudoux.dej.hibernate.entity.CompteEpargne"
      discriminator-value="CompteEpargne">
      <join table="compte_epargne">
        <key column="compte_epargne_id" />
        <property name="taux" column="taux" type="big_decimal"/>
      </join>
    </subclass>
  </class>
</hibernate-mapping>
```

Résultat :

```
2015-02-03 21:59:56.856 INFO   [main]:org.hibernate.annotations.common.Version - HCANN000001:
Hibernate Commons Annotations {4.0.1.Final}
2015-02-03 21:59:56.864 INFO   [main]:org.hibernate.Version - HHH000412: Hibernate Core {4.1.
4.Final}
2015-02-03 21:59:56.866 INFO   [main]:org.hibernate.cfg.Environment - HHH000206: hibernate.
properties not found
2015-02-03 21:59:56.868 INFO   [main]:org.hibernate.cfg.Environment - HHH000021: Bytecode
provider name : javassist
2015-02-03 21:59:56.895 INFO   [main]:org.hibernate.cfg.Configuration - HHH000043:
Configuring from resource: /hibernate.cfg.xml
2015-02-03 21:59:56.895 INFO   [main]:org.hibernate.cfg.Configuration - HHH000040:
Configuration resource: /hibernate.cfg.xml
2015-02-03 21:59:56.943 INFO   [main]:org.hibernate.cfg.Configuration - HHH000221: Reading
mappings from resource: Compte.hbm.xml
2015-02-03 21:59:57.000 INFO   [main]:org.hibernate.cfg.Configuration - HHH000041: Configured
SessionFactory: null
2015-02-03 21:59:57.104 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000402: Using Hibernate built-in connection pool (not
for production use!)
2015-02-03 21:59:57.115 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000115: Hibernate connection pool size: 1
2015-02-03 21:59:57.116 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000006: Autocommit mode: false
2015-02-03 21:59:57.116 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000401: using driver [com.mysql.jdbc.Driver] at URL
```

```
[jdbc:mysql://localhost:3307/mabdd]
2015-02-03 21:59:57.117 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000046: Connection properties: {user=root, password=
****}
2015-02-03 21:59:57.530 INFO   [main]:org.hibernate.dialect.Dialect - HHH000400: Using
dialect: org.hibernate.dialect.MySQL5Dialect
2015-02-03 21:59:57.552 INFO   [main]:org.hibernate.engine.transaction.internal.
TransactionFactoryInitiator - HHH000399: Using default transaction strategy (direct JDBC
transactions)
2015-02-03 21:59:57.557 INFO   [main]:org.hibernate.hql.internal.ast.
ASTQueryTranslatorFactory - HHH000397: Using ASTQueryTranslatorFactory
2015-02-03 21:59:57.897 INFO   [main]:org.hibernate.tool.hbm2ddl.SchemaExport - HHH000227:
Running hbm2ddl schema export
Hibernate: alter table compte_courant drop foreign key FK2435FDBFBBF1A400
2015-02-03 21:59:57.915 ERROR  [main]:org.hibernate.tool.hbm2ddl.SchemaExport - HHH000389:
Unsuccessful: alter table compte_courant drop foreign key FK2435FDBFBBF1A400
2015-02-03 21:59:57.915 ERROR  [main]:org.hibernate.tool.hbm2ddl.SchemaExport - Table
'mabdd.compte_courant' doesn't exist
Hibernate: alter table compte_epargne drop foreign key FK8E9D8D438FCF4580
2015-02-03 21:59:57.917 ERROR  [main]:org.hibernate.tool.hbm2ddl.SchemaExport - HHH000389:
Unsuccessful: alter table compte_epargne drop foreign key FK8E9D8D438FCF4580
2015-02-03 21:59:57.917 ERROR  [main]:org.hibernate.tool.hbm2ddl.SchemaExport - Table 'mabdd.
compte_epargne' doesn't exist
Hibernate: drop table if exists compte
Hibernate: drop table if exists compte_courant
Hibernate: drop table if exists compte_epargne
Hibernate: create table compte (id integer not null auto_increment, DTYPE varchar(255) not
null, numero varchar(255), solde decimal(19,2), primary key (id))
Hibernate: create table compte_courant (compte_courant_id integer not null, decouvert integer
, primary key (compte_courant_id))
Hibernate: create table compte_epargne (compte_epargne_id integer not null, taux decimal(19,2)
, primary key (compte_epargne_id))
Hibernate: alter table compte_courant add index FK2435FDBFBBF1A400 (compte_courant_id), add
constraint FK2435FDBFBBF1A400 foreign key (compte_courant_id) references compte (id)
Hibernate: alter table compte_epargne add index FK8E9D8D438FCF4580 (compte_epargne_id), add
constraint FK8E9D8D438FCF4580 foreign key (compte_epargne_id) references compte (id)
2015-02-03 21:59:58.088 INFO    [main]:org.hibernate.tool.hbm2ddl.SchemaExport - HHH000230:
Schema export complete
Hibernate: insert into compte (numero, solde, DTYPE) values (?, ?, 'Compte')
Hibernate: insert into compte (numero, solde, DTYPE) values (?, ?, 'CompteCourant')
Hibernate: insert into compte_courant (decouvert, compte_courant_id) values (?, ?)
Hibernate: insert into compte (numero, solde, DTYPE) values (?, ?, 'CompteEpargne')
Hibernate: insert into compte_epargne (taux, compte_epargne_id) values (?, ?)
```


compte
id INT(11)
DTYPE VARCHAR(255)
numero VARCHAR(255)
solde DECIMAL(19,2)
Indexes

compte_courant
compte_courant_id INT(11)
decouvert INT(11)
Indexes

compte_epargne
compte_epargne_id INT(11)
taux DECIMAL(19,2)
Indexes

Résultat :

```
CREATE TABLE `compte` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `DTYPE` varchar(255) NOT NULL,
  `numero` varchar(255) DEFAULT NULL,
  `solde` decimal(19,2) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8;

CREATE TABLE `compte_courant` (
  `compte_courant_id` int(11) NOT NULL,
  `decouvert` int(11) DEFAULT NULL,
  PRIMARY KEY (`compte_courant_id`),
  KEY `FK2435FDBFBBF1A400` (`compte_courant_id`),
  CONSTRAINT `FK2435FDBFBBF1A400` FOREIGN KEY (`compte_courant_id`) REFERENCES `compte` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `compte_epargne` (
  `compte_epargne_id` int(11) NOT NULL,
  `taux` decimal(19,2) DEFAULT NULL,
  PRIMARY KEY (`compte_epargne_id`),
  KEY `FK8E9D8D438FCF4580` (`compte_epargne_id`),
  CONSTRAINT `FK8E9D8D438FCF4580` FOREIGN KEY (`compte_epargne_id`) REFERENCES `compte` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Résultat :

```
mysql> select * from compte;
+----+---------------+--------------+---------+
| id | DTYPE         | numero       | solde   |
+----+---------------+--------------+---------+
|  1 | Compte        | 000012345000 |    0.00 |
|  2 | CompteCourant | 000012345010 | 1200.00 |
|  3 | CompteEpargne | 000012345020 | 8000.00 |
+----+---------------+--------------+---------+
3 rows in set (0.00 sec)
mysql> select * from compte_courant;
+-------------------+-----------+
| compte_courant_id | decouvert |
+-------------------+-----------+
|                 2 |      2000 |
+-------------------+-----------+
1 row in set (0.00 sec)
mysql> select * from compte_epargne;
+-------------------+------+
| compte_epargne_id | taux |
+-------------------+------+
|                 3 | 2.10 |
+-------------------+------+
1 row in set (0.00 sec)
```

Résultat :

```
Recherche d'un compte
Hibernate: select compte0_.id as id0_0_, compte0_.numero as numero0_0_, compte0_.solde as
solde0_0_, compte0_1_.decouvert as decouvert1_0_, compte0_2_.taux as taux2_0_, compte0_.DTYPE
as DTYPE0_0_ from compte compte0_ left outer join compte_courant compte0_1_ on compte0_.id=
compte0_1_.compte_courant_id left outer join compte_epargne compte0_2_ on compte0_.id=
compte0_2_.compte_epargne_id where compte0_.id=?
fr.jmdoudoux.dej.hibernate.entity.Compte@422ddc [id=1, numero=000012345000, solde=0.00]

Recherche d'un compte courant
Hibernate: select comptecour0_.id as id0_0_, comptecour0_.numero as numero0_0_, comptecour0_.
solde as solde0_0_, comptecour0_1_.decouvert as decouvert1_0_ from compte comptecour0_ inner
join compte_courant comptecour0_1_ on comptecour0_.id=comptecour0_1_.compte_courant_id where
comptecour0_.id=? and comptecour0_.DTYPE='CompteCourant'
fr.jmdoudoux.dej.hibernate.entity.CompteCourant@32800355CompteCourant [id=2, numero=
000012345010, solde=1200.00, decouvert=2000]

Recherche polymorphique d'un compte
Hibernate: select compte0_.id as id0_0_, compte0_.numero as numero0_0_, compte0_.solde as
solde0_0_, compte0_1_.decouvert as decouvert1_0_, compte0_2_.taux as taux2_0_, compte0_.DTYPE
as DTYPE0_0_ from compte compte0_ left outer join compte_courant compte0_1_ on compte0_.id=
compte0_1_.compte_courant_id left outer join compte_epargne compte0_2_ on compte0_.id=
compte0_2_.compte_epargne_id where compte0_.id=?
```

```
fr.jmdoudoux.dej.hibernate.entity.CompteEpargne@29827422CompteEpargne [ id=3, numero=
000012345020, solde=8000.00, taux=2.10]

Recherche polymorphique de comptes
Hibernate: select compte0_.id as id0_, compte0_.numero as numero0_, compte0_.solde as solde0_,
compte0_1_.decouvert as decouvert1_, compte0_2_.taux as taux2_, compte0_.DTYPE as DTYPE0_ from
compte compte0_ left outer join compte_courant compte0_1_ on compte0_.id=compte0_1_.
compte_courant_id left outer join compte_epargne compte0_2_ on compte0_.id=compte0_2_.
compte_epargne_id where compte0_.numero like ?
fr.jmdoudoux.dej.hibernate.entity.CompteCourant@27737342CompteCourant [id=2, numero=
000012345010, solde=1200.00, decouvert=2000]
fr.jmdoudoux.dej.hibernate.entity.CompteEpargne@31801703CompteEpargne [ id=3, numero=
000012345020, solde=8000.00, taux=2.10]
fr.jmdoudoux.dej.hibernate.entity.Compte@28e5a7 [id=1, numero=000012345000, solde=0.00]
```

Lors de la recherche des différentes entités, une seule requête SQL est exécutée à chaque fois requérant une jointure entre au moins deux tables voire toutes les tables. Ceci peut poser des problèmes de performance notamment lorsque le nombre de tables filles est plus important.

Pour palier partiellement cette problématique, il est possible de changer la stratégie de récupération des données des tables filles en demandant à Hibernate de ne pas faire de jointure mais de faire des requêtes SQL dédiées. Pour cela, il faut utiliser l'attribut fetch="select" du tag <join>.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name="fr.jmdoudoux.dej.hibernate.entity.Compte"
    table="compte" discriminator-value="Compte">
    <id name="id" column="id" type="int" >
      <generator class="native" />
    </id>
    <discriminator column="DTYPE" type="string" />

    <property name="numero" column="numero" type="string" />
    <property name="solde" column="solde" type="big_decimal" />
    <subclass name="fr.jmdoudoux.dej.hibernate.entity.CompteCourant"
      discriminator-value="CompteCourant">
      <join table="compte_courant" fetch="select">
         <key column="compte_courant_id" />
         <property name="decouvert" column="decouvert" type="int"/>
      </join>
    </subclass>
    <subclass name="fr.jmdoudoux.dej.hibernate.entity.CompteEpargne"
      discriminator-value="CompteEpargne">
      <join table="compte_epargne" fetch="select">
        <key column="compte_epargne_id" />
        <property name="taux" column="taux" type="big_decimal"/>
      </join>
    </subclass>
  </class>
</hibernate-mapping>
```

Résultat :

```
Recherche d'un compte
Hibernate: select compte0_.id as id0_0_, compte0_.numero as numero0_0_, compte0_.solde as
solde0_0_, compte0_.DTYPE as DTYPE0_0_ from compte compte0_ where compte0_.id=?
fr.jmdoudoux.dej.hibernate.entity.Compte@e15c54 [id=1, numero=000012345000, solde=0.00]

Recherche d'un compte courant
Hibernate: select comptecour0_.id as id0_0_, comptecour0_.numero as numero0_0_, comptecour0_.
solde as solde0_0_, comptecour0_1_.decouvert as decouvert1_0_ from compte comptecour0_ inner
join compte_courant comptecour0_1_ on comptecour0_.id=comptecour0_1_.compte_courant_id where
comptecour0_.id=? and comptecour0_.DTYPE='CompteCourant'
fr.jmdoudoux.dej.hibernate.entity.CompteCourant@24177821CompteCourant [id=2, numero=
000012345010, solde=1200.00, decouvert=2000]
```

```
Recherche polymorphique d'un compte
Hibernate: select compte0_.id as id0_0_, compte0_.numero as numero0_0_, compte0_.solde as
solde0_0_, compte0_.DTYPE as DTYPE0_0_ from compte compte0_ where compte0_.id=?
Hibernate: select compte_2_.taux as taux2_ from compte_epargne compte_2_ where compte_2_.
compte_epargne_id=?
fr.jmdoudoux.dej.hibernate.entity.CompteEpargne@11010466CompteEpargne [ id=3, numero=
000012345020, solde=8000.00, taux=2.10]

Recherche polymorphique de comptes
Hibernate: select compte0_.id as id0_, compte0_.numero as numero0_, compte0_.solde as solde0_,
compte0_.DTYPE as DTYPE0_ from compte compte0_ where compte0_.numero like ?
Hibernate: select compte_1_.decouvert as decouvert1_ from compte_courant compte_1_ where
compte_1_.compte_courant_id=?
Hibernate: select compte_2_.taux as taux2_ from compte_epargne compte_2_ where compte_2_.
compte_epargne_id=?
fr.jmdoudoux.dej.hibernate.entity.Compte@476e01 [id=1, numero=000012345000, solde=0.00]
fr.jmdoudoux.dej.hibernate.entity.CompteCourant@11305489CompteCourant [id=2, numero=
000012345010, solde=1200.00, decouvert=2000]
fr.jmdoudoux.dej.hibernate.entity.CompteEpargne@15939468CompteEpargne [ id=3, numero=
000012345020, solde=8000.00, taux=2.10]
```

Ce sont des requêtes SQL supplémentaires dont le nombre peut devenir important en fonction des entités retournées mais ces requêtes se font sur la clé primaire des tables filles.

62.11.2. Annotations

Comme Hibernate est une implémentation de JPA, il est possible de définir le mapping en utilisant des annotations.

Le fichier de configuration ci-dessous est utilisé dans les exemples de cette section.

Exemple :

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost:3307/mabdd</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">root</property>
    <property name="hibernate.connection.pool_size">1</property>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</property>
    <property name="hibernate.current_session_context_class">thread</property>
    <!-- Cache de second niveau désactivé  -->
    <property name="hibernate.cache.provider_class">
      org.hibernate.cache.internal.NoCacheProvider</property>
    <!-- Afficher les requêtes SQL exécutées sur la sortie standard -->
    <property name="hibernate.show_sql">true</property>
    <property name="hbm2ddl.auto">create</property>
    <mapping class="fr.jmdoudoux.dej.hibernate.entity.Compte"/>
    <mapping class="fr.jmdoudoux.dej.hibernate.entity.CompteCourant"/>
    <mapping class="fr.jmdoudoux.dej.hibernate.entity.CompteEpargne"/>
  </session-factory>
</hibernate-configuration>
```

Deux classes filles qui héritent de la classe Compte sont définies en tant qu'entités.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.hibernate.entity;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Table;

@Entity
@Table(name = "compte_courant")
public class CompteCourant extends Compte {

  @Column(name = "decouvert")
  private int decouvert;

  public int getDecouvert() {
    return decouvert;
  }

  public void setDecouvert(int decouvert) {
    this.decouvert = decouvert;
  }

  @Override
  public String toString() {
    return this.getClass().getName() + "@" + System.identityHashCode(this)
        + "CompteCourant [id=" + id + ", numero=" + numero + ", solde=" + solde
        + ", decouvert=" + decouvert + "]";
  }
}
```

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.hibernate.entity;

import java.math.BigDecimal;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Table;

@Entity
@Table(name = "compte_epargne")
public class CompteEpargne extends Compte {
  @Column(name = "taux")
  private BigDecimal taux;

  public BigDecimal getTaux() {
    return taux;
  }

  public void setTaux(BigDecimal taux) {
    this.taux = taux;
  }

  @Override
  public String toString() {
    return this.getClass().getName() + "@" + System.identityHashCode(this)
        + "CompteEpargne [ id=" + id + ", numero=" + numero + ", solde="
        + solde + ", taux=" + taux + "]";
  }
}
```

62.11.2.1. Annotations, une table par hiérarchie de classes (SINGLE_TABLE)

L'entité de la classe mère utilise la stratégie InheritanceType.SINGLE_TABLE grâce à l'annotation @Inheritance.

Exemple (code Java 5.0) :

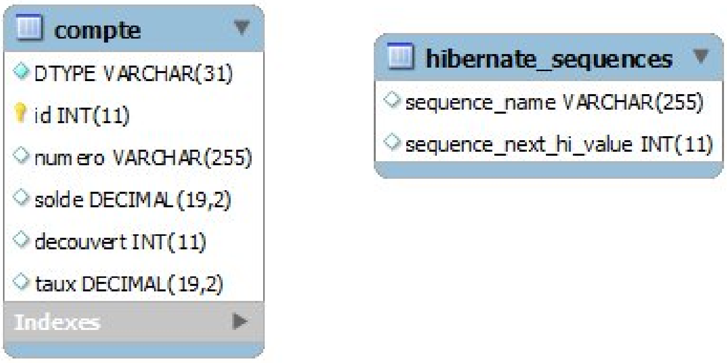
```
package fr.jmdoudoux.dej.hibernate.entity;

import java.math.BigDecimal;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.Table;

@Entity
@Table(name = "compte")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public class Compte {

  @Id
  @GeneratedValue(strategy = GenerationType.TABLE)
  @Column(name = "id")
  protected int        id;

  @Column(name = "numero")
  protected String     numero;

  @Column(name = "solde")
  protected BigDecimal solde;

  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public String getNumero() {
    return numero;
  }

  public void setNumero(String numero) {
    this.numero = numero;
  }

  public BigDecimal getSolde() {
    return solde;
  }

  public void setSolde(BigDecimal solde) {
    this.solde = solde;
  }

  @Override
  public String toString() {
    return super.toString() + " [id=" + id + ", numero=" + numero + ", solde="
        + solde + "]";
  }
}
```

Résultat :

```
2015-03-03 23:00:59.778 INFO   [main]:org.hibernate.annotations.common.Version - HCANN000001:
Hibernate Commons Annotations {4.0.1.Final}
2015-03-03 23:00:59.786 INFO   [main]:org.hibernate.Version - HHH000412: Hibernate Core {4.1.
4.Final}
2015-03-03 23:00:59.788 INFO   [main]:org.hibernate.cfg.Environment - HHH000206: hibernate.
properties not found
2015-03-03 23:00:59.790 INFO   [main]:org.hibernate.cfg.Environment - HHH000021: Bytecode
provider name : javassist
2015-03-03 23:00:59.818 INFO   [main]:org.hibernate.cfg.Configuration - HHH000043:
Configuring from resource: /hibernate.cfg.xml
2015-03-03 23:00:59.818 INFO   [main]:org.hibernate.cfg.Configuration - HHH000040:
Configuration resource: /hibernate.cfg.xml
2015-03-03 23:00:59.874 INFO   [main]:org.hibernate.cfg.Configuration - HHH000041: Configured
SessionFactory: null
2015-03-03 23:01:00.042 WARN   [main]:org.hibernate.cfg.AnnotationBinder - HHH000139: Illegal
use of @Table in a subclass of a SINGLE_TABLE hierarchy: fr.jmdoudoux.dej.hibernate.entity.
CompteCourant
2015-03-03 23:01:00.043 WARN   [main]:org.hibernate.cfg.AnnotationBinder - HHH000139: Illegal
use of @Table in a subclass of a SINGLE_TABLE hierarchy: fr.jmdoudoux.dej.hibernate.entity.
CompteEpargne
2015-03-03 23:01:00.056 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
```

```
DriverManagerConnectionProviderImpl - HHH000402: Using Hibernate built-in connection pool (not
for production use!)
2015-03-03 23:01:00.063 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000115: Hibernate connection pool size: 1
2015-03-03 23:01:00.063 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000006: Autocommit mode: false
2015-03-03 23:01:00.064 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000401: using driver [com.mysql.jdbc.Driver] at URL [
jdbc:mysql://localhost:3307/mabdd]
2015-03-03 23:01:00.064 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000046: Connection properties: {user=root, password=
****}
2015-03-03 23:01:00.367 INFO   [main]:org.hibernate.dialect.Dialect - HHH000400: Using
dialect: org.hibernate.dialect.MySQL5Dialect
2015-03-03 23:01:00.397 INFO   [main]:org.hibernate.engine.transaction.internal.
TransactionFactoryInitiator - HHH000399: Using default transaction strategy (direct JDBC
transactions)
2015-03-03 23:01:00.402 INFO   [main]:org.hibernate.hql.internal.ast.
ASTQueryTranslatorFactory - HHH000397: Using ASTQueryTranslatorFactory
2015-03-03 23:01:00.671 INFO   [main]:org.hibernate.tool.hbm2ddl.SchemaExport - HHH000227:
Running hbm2ddl schema export
Hibernate: drop table if exists compte
Hibernate: drop table if exists hibernate_sequences
Hibernate: create table compte (DTYPE varchar(31) not null, id integer not null, numero
varchar(255), solde decimal(19,2), decouvert integer, taux decimal(19,2), primary key (id))
Hibernate: create table hibernate_sequences ( sequence_name varchar(255),
sequence_next_hi_value integer )
2015-03-03 23:01:00.722 INFO     [main]:org.hibernate.tool.hbm2ddl.SchemaExport - HHH000230:
Schema export complete
Hibernate: select sequence_next_hi_value from hibernate_sequences where sequence_name = '
compte' for update
Hibernate: insert into hibernate_sequences(sequence_name, sequence_next_hi_value) values('
compte', ?)
Hibernate: update hibernate_sequences set sequence_next_hi_value = ? where
sequence_next_hi_value = ? and sequence_name = 'compte'
Hibernate: insert into compte (numero, solde, DTYPE, id) values (?, ?, 'Compte', ?)
Hibernate: insert into compte (numero, solde, decouvert, DTYPE, id) values (?, ?, ?, '
CompteCourant', ?)
Hibernate: insert into compte (numero, solde, taux, DTYPE, id) values (?, ?, ?, 'CompteEpargne'
, ?)
Recherche d'un compte
Hibernate: select compte0_.id as id0_0_, compte0_.numero as numero0_0_, compte0_.solde as
solde0_0_, compte0_.decouvert as decouvert0_0_, compte0_.taux as taux0_0_, compte0_.DTYPE as
DTYPE0_0_ from compte compte0_ where compte0_.id=?
fr.jmdoudoux.dej.hibernate.entity.Compte@af908f [id=1, numero=000012345000, solde=0.00]

Recherche d'un compte courant
Hibernate: select comptecour0_.id as id0_0_, comptecour0_.numero as numero0_0_, comptecour0_.
solde as solde0_0_, comptecour0_.decouvert as decouvert0_0_ from compte comptecour0_ where
comptecour0_.id=? and comptecour0_.DTYPE='CompteCourant'
fr.jmdoudoux.dej.hibernate.entity.CompteCourant@27723440CompteCourant [id=2, numero=
000012345010, solde=1200.00, decouvert=2000]

Recherche polymorphique d'un compte
Hibernate: select compte0_.id as id0_0_, compte0_.numero as numero0_0_, compte0_.solde as
solde0_0_, compte0_.decouvert as decouvert0_0_, compte0_.taux as taux0_0_, compte0_.DTYPE as
DTYPE0_0_ from compte compte0_ where compte0_.id=?
fr.jmdoudoux.dej.hibernate.entity.CompteEpargne@20700084CompteEpargne [ id=3, numero=
000012345020, solde=8000.00, taux=2.10]

Recherche de tous les comptes
Hibernate: select compte0_.id as id0_, compte0_.numero as numero0_, compte0_.solde as solde0_,
compte0_.decouvert as decouvert0_, compte0_.taux as taux0_, compte0_.DTYPE as DTYPE0_ from
compte compte0_
fr.jmdoudoux.dej.hibernate.entity.Compte@164fc53 [id=1, numero=000012345000, solde=0.00]
fr.jmdoudoux.dej.hibernate.entity.CompteCourant@13942651CompteCourant [id=2, numero=
000012345010, solde=1200.00, decouvert=2000]
fr.jmdoudoux.dej.hibernate.entity.CompteEpargne@23542324CompteEpargne [ id=3, numero=
000012345020, solde=8000.00, taux=2.10]

Recherche polymorphique de comptes
Hibernate: select compte0_.id as id0_, compte0_.numero as numero0_, compte0_.solde as solde0_,
compte0_.decouvert as decouvert0_, compte0_.taux as taux0_, compte0_.DTYPE as DTYPE0_ from
compte compte0_ where compte0_.numero like ?
```

```
fr.jmdoudoux.dej.hibernate.entity.Compte@6ca594 [id=1, numero=000012345000, solde=0.00]
fr.jmdoudoux.dej.hibernate.entity.CompteCourant@15384254CompteCourant [id=2, numero=
000012345010, solde=1200.00, decouvert=2000]
fr.jmdoudoux.dej.hibernate.entity.CompteEpargne@4681268CompteEpargne [ id=3, numero=
000012345020, solde=8000.00, taux=2.10]
```



Résultat :

```
mysql> select * from compte;
+---------------+----+--------------+---------+-----------+------+
| DTYPE         | id | numero       | solde   | decouvert | taux |
+---------------+----+--------------+---------+-----------+------+
| Compte        |  1 | 000012345000 |    0.00 |      NULL | NULL |
| CompteCourant |  2 | 000012345010 | 1200.00 |      2000 | NULL |
| CompteEpargne |  3 | 000012345020 | 8000.00 |      NULL | 2.10 |
+---------------+----+--------------+---------+-----------+------+
3 rows in set (0.00 sec)
```

62.11.2.2. Annotations, une table par classe concrète (TABLE_PER_CLASS)

Hibernate propose un support de la stratégie une table par classe concrète qui est définie comme étant optionnelle par JPA. Pour cela, il suffit d'utiliser sur la classe mère l'annotation @Inheritance avec la propriété strategy initialisée avec InheritanceType.TABLE_PER_CLASS.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.hibernate.entity;

import java.math.BigDecimal;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.Table;

@Entity
@Table(name = "compte")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Compte {
  @Id
  @GeneratedValue(strategy = GenerationType.TABLE)
  @Column(name = "id")
  protected int        id;

  @Column(name = "numero")
  protected String     numero;

  @Column(name = "solde")
  protected BigDecimal solde;
  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public String getNumero() {
    return numero;
  }

  public void setNumero(String numero) {
    this.numero = numero;
  }

  public BigDecimal getSolde() {
    return solde;
  }

  public void setSolde(BigDecimal solde) {
    this.solde = solde;
  }

  @Override
  public String toString() {
    return super.toString() + " [id=" + id + ", numero=" + numero + ", solde="
        + solde + "]";
  }
}
```

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.hibernate.entity;

import javax.persistence.AttributeOverride;
import javax.persistence.AttributeOverrides;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Table;

@Entity
@Table(name = "compte_courant")
@AttributeOverrides({
    @AttributeOverride(name = "id", column = @Column(name = "id")),
    @AttributeOverride(name = "numero", column = @Column(name = "numero")),
    @AttributeOverride(name = "solde", column = @Column(name = "solde")) })
public class CompteCourant extends Compte {
  @Column(name = "decouvert")
  private int decouvert;

  public int getDecouvert() {
    return decouvert;
  }

  public void setDecouvert(int decouvert) {
    this.decouvert = decouvert;
  }

  @Override
  public String toString() {
    return this.getClass().getName() + "@" + System.identityHashCode(this)
        + "CompteCourant [id=" + id + ", numero=" + numero + ", solde=" + solde
        + ", decouvert=" + decouvert + "]";
  }
}
```

Exemple (code Java 5.0) :

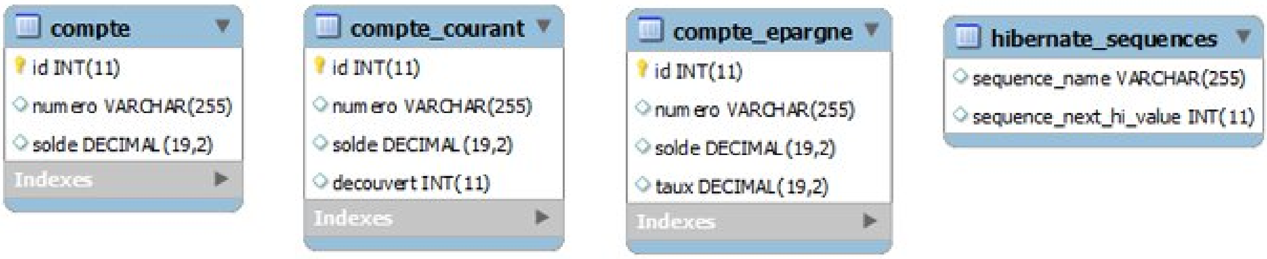
```
package fr.jmdoudoux.dej.hibernate.entity;

import java.math.BigDecimal;
import javax.persistence.AttributeOverride;
import javax.persistence.AttributeOverrides;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Table;

@Entity
@Table(name = "compte_epargne")
@AttributeOverrides({
    @AttributeOverride(name = "id", column = @Column(name = "id")),
    @AttributeOverride(name = "numero", column = @Column(name = "numero")),
    @AttributeOverride(name = "solde", column = @Column(name = "solde")) })
public class CompteEpargne extends Compte {
  @Column(name = "taux")
  private BigDecimal taux;

  public BigDecimal getTaux() {
    return taux;
  }

  public void setTaux(BigDecimal taux) {
    this.taux = taux;
  }

  @Override
  public String toString() {
    return this.getClass().getName() + "@" + System.identityHashCode(this)
        + "CompteEpargne [ id=" + id + ", numero=" + numero + ", solde="
        + solde + ", taux=" + taux + "]";
  }
}
```

Dans les exemples ci-dessus, les annotations @AttributesOverrides et @AttributesOverride sont employées à titre indicatif car elles utilisent les valeurs par défaut et pourraient donc être retirées.

Résultat :

```
2015-03-03 23:18:12.766 INFO   [main]:org.hibernate.annotations.common.Version - HCANN000001:
Hibernate Commons Annotations {4.0.1.Final}
2015-03-03 23:18:12.776 INFO   [main]:org.hibernate.Version - HHH000412: Hibernate Core {4.1.
4.Final}
2015-03-03 23:18:12.779 INFO   [main]:org.hibernate.cfg.Environment - HHH000206: hibernate.
properties not found
2015-03-03 23:18:12.780 INFO   [main]:org.hibernate.cfg.Environment - HHH000021: Bytecode
provider name : javassist
2015-03-03 23:18:12.814 INFO   [main]:org.hibernate.cfg.Configuration - HHH000043:
Configuring from resource: /hibernate.cfg.xml
2015-03-03 23:18:12.814 INFO   [main]:org.hibernate.cfg.Configuration - HHH000040:
Configuration resource: /hibernate.cfg.xml
2015-03-03 23:18:12.891 INFO   [main]:org.hibernate.cfg.Configuration - HHH000041: Configured
SessionFactory: null
2015-03-03 23:18:13.080 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000402: Using Hibernate built-in connection pool (not
for production use!)
2015-03-03 23:18:13.087 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000115: Hibernate connection pool size: 1
2015-03-03 23:18:13.088 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000006: Autocommit mode: false
2015-03-03 23:18:13.088 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000401: using driver [com.mysql.jdbc.Driver] at URL [
jdbc:mysql://localhost:3307/mabdd]
2015-03-03 23:18:13.089 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000046: Connection properties: {user=root, password=
****}
2015-03-03 23:18:13.392 INFO   [main]:org.hibernate.dialect.Dialect - HHH000400: Using
dialect: org.hibernate.dialect.MySQL5Dialect
2015-03-03 23:18:13.424 INFO   [main]:org.hibernate.engine.transaction.internal.
TransactionFactoryInitiator - HHH000399: Using default transaction strategy (direct JDBC
transactions)
2015-03-03 23:18:13.428 INFO   [main]:org.hibernate.hql.internal.ast.
ASTQueryTranslatorFactory - HHH000397: Using ASTQueryTranslatorFactory
2015-03-03 23:18:13.736 INFO   [main]:org.hibernate.tool.hbm2ddl.SchemaExport - HHH000227:
Running hbm2ddl schema export
```

```
Hibernate: drop table if exists compte
Hibernate: drop table if exists compte_courant
Hibernate: drop table if exists compte_epargne
Hibernate: drop table if exists hibernate_sequences
Hibernate: create table compte (id integer not null, numero varchar(255), solde decimal(19,2),
primary key (id))
Hibernate: create table compte_courant (id integer not null, numero varchar(255), solde
decimal(19,2), decouvert integer, primary key (id))
Hibernate: create table compte_epargne (id integer not null, numero varchar(255), solde
decimal(19,2), taux decimal(19,2), primary key (id))
Hibernate: create table hibernate_sequences ( sequence_name varchar(255),
sequence_next_hi_value integer )
2015-03-03 23:18:13.843 INFO     [main]:org.hibernate.tool.hbm2ddl.SchemaExport - HHH000230:
Schema export complete
Hibernate: select sequence_next_hi_value from hibernate_sequences where sequence_name = '
compte' for update
Hibernate: insert into hibernate_sequences(sequence_name, sequence_next_hi_value) values('
compte', ?)
Hibernate: update hibernate_sequences set sequence_next_hi_value = ? where
sequence_next_hi_value = ? and sequence_name = 'compte'
Hibernate: insert into compte (numero, solde, id) values (?, ?, ?)
Hibernate: insert into compte_courant (numero, solde, decouvert, id) values (?, ?, ?, ?)
Hibernate: insert into compte_epargne (numero, solde, taux, id) values (?, ?, ?, ?)
Recherche d'un compte
Hibernate: select compte0_.id as id0_0_, compte0_.numero as numero0_0_, compte0_.solde as
solde0_0_, compte0_.decouvert as decouvert1_0_, compte0_.taux as taux2_0_, compte0_.clazz_ as
clazz_0_ from ( select id, numero, solde, null as decouvert, null as taux, 0 as clazz_ from
compte union select id, numero, solde, decouvert, null as taux, 1 as clazz_ from
compte_courant union select id, numero, solde, null as decouvert, taux, 2 as clazz_ from
compte_epargne ) compte0_ where compte0_.id=?
fr.jmdoudoux.dej.hibernate.entity.Compte@198f6dd [id=1, numero=000012345000, solde=0.00]

Recherche d'un compte courant
Hibernate: select comptecour0_.id as id0_0_, comptecour0_.numero as numero0_0_, comptecour0_.
solde as solde0_0_, comptecour0_.decouvert as decouvert1_0_ from compte_courant comptecour0_
where comptecour0_.id=?
fr.jmdoudoux.dej.hibernate.entity.CompteCourant@3316864CompteCourant [id=2, numero=
000012345010, solde=1200.00, decouvert=2000]

Recherche polymorphique d'un compte
Hibernate: select compte0_.id as id0_0_, compte0_.numero as numero0_0_, compte0_.solde as
solde0_0_, compte0_.decouvert as decouvert1_0_, compte0_.taux as taux2_0_, compte0_.clazz_ as
clazz_0_ from ( select id, numero, solde, null as decouvert, null as taux, 0 as clazz_ from
compte union select id, numero, solde, decouvert, null as taux, 1 as clazz_ from
compte_courant union select id, numero, solde, null as decouvert, taux, 2 as clazz_ from
compte_epargne ) compte0_ where compte0_.id=?
fr.jmdoudoux.dej.hibernate.entity.CompteEpargne@20376252CompteEpargne [ id=3, numero=
000012345020, solde=8000.00, taux=2.10]

Recherche de tous les comptes
Hibernate: select compte0_.id as id0_, compte0_.numero as numero0_, compte0_.solde as solde0_,
compte0_.decouvert as decouvert1_, compte0_.taux as taux2_, compte0_.clazz_ as clazz_ from (
select id, numero, solde, null as decouvert, null as taux, 0 as clazz_ from compte union
select id, numero, solde, decouvert, null as taux, 1 as clazz_ from compte_courant union
select id, numero, solde, null as decouvert, taux, 2 as clazz_ from compte_epargne ) compte0_
fr.jmdoudoux.dej.hibernate.entity.Compte@165d7a2 [id=1, numero=000012345000, solde=0.00]
fr.jmdoudoux.dej.hibernate.entity.CompteCourant@30230592CompteCourant [id=2, numero=
000012345010, solde=1200.00, decouvert=2000]
fr.jmdoudoux.dej.hibernate.entity.CompteEpargne@4358252CompteEpargne [ id=3, numero=
000012345020, solde=8000.00, taux=2.10]

Recherche polymorphique de comptes
Hibernate: select compte0_.id as id0_, compte0_.numero as numero0_, compte0_.solde as solde0_,
compte0_.decouvert as decouvert1_, compte0_.taux as taux2_, compte0_.clazz_ as clazz_ from (
select id, numero, solde, null as decouvert, null as taux, 0 as clazz_ from compte union
select id, numero, solde, decouvert, null as taux, 1 as clazz_ from compte_courant union
select id, numero, solde, null as decouvert, taux, 2 as clazz_ from compte_epargne ) compte0_
where compte0_.numero like ?
fr.jmdoudoux.dej.hibernate.entity.Compte@129d5bf [id=1, numero=000012345000, solde=0.00]
fr.jmdoudoux.dej.hibernate.entity.CompteCourant@26225329CompteCourant [id=2, numero=
000012345010, solde=1200.00, decouvert=2000]
fr.jmdoudoux.dej.hibernate.entity.CompteEpargne@11121275CompteEpargne [ id=3, numero=
000012345020, solde=8000.00, taux=2.10]
```



compte
- id INT(11)
- numero VARCHAR(255)
- solde DECIMAL(19,2)

compte_courant
- id INT(11)
- numero VARCHAR(255)
- solde DECIMAL(19,2)
- decouvert INT(11)

compte_epargne
- id INT(11)
- numero VARCHAR(255)
- solde DECIMAL(19,2)
- taux DECIMAL(19,2)

hibernate_sequences
- sequence_name VARCHAR(255)
- sequence_next_hi_value INT(11)

Résultat :

```
Enter password: ****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 73
Server version: 5.6.15 MySQL Community Server (GPL)

Copyright (c) 2000, 2013, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates.
Other names may be trademarks of their respective owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use mabdd
Database changed
mysql> select * from compte;
+----+--------------+-------+
| id | numero       | solde |
+----+--------------+-------+
|  1 | 000012345000 |  0.00 |
+----+--------------+-------+
1 row in set (0.02 sec)
mysql> select * from compte_courant;
+----+--------------+---------+-----------+
| id | numero       | solde   | decouvert |
+----+--------------+---------+-----------+
|  2 | 000012345010 | 1200.00 |      2000 |
+----+--------------+---------+-----------+
1 row in set (0.00 sec)
mysql> select * from compte_epargne;
+----+--------------+---------+------+
| id | numero       | solde   | taux |
+----+--------------+---------+------+
|  3 | 000012345020 | 8000.00 | 2.10 |
+----+--------------+---------+------+
1 row in set (0.00 sec)
mysql>
```

62.11.2.3. Annotations, une table par sous-classe (JOINED)

L'entité de la classe mère utilise la stratégie InheritanceType.JOINED grâce à l'annotation @Inheritance.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.hibernate.entity;

import java.math.BigDecimal;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.Table;

@Entity
@Table(name = "compte")
@Inheritance(strategy =
InheritanceType.JOINED)
public class Compte {
  @Id
  @GeneratedValue
  @Column(name = "id")
  protected int        id;

  @Column(name = "numero")
  protected String     numero;

  @Column(name = "solde")
  protected BigDecimal solde;

  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public String getNumero() {
    return numero;
  }

  public void setNumero(String numero) {
    this.numero = numero;
  }

  public BigDecimal getSolde() {
    return solde;
  }

  public void setSolde(BigDecimal solde) {
    this.solde = solde;
  }

  @Override
  public String toString() {
    return super.toString() + " [id="+ id + ", numero=" + numero + ", solde="
        + solde + "]";
  }
}
```

L'annotation @PrimaryKeyJoinColumn est utilisée sur les classes filles pour préciser la colonne qui est la clé primaire et qui servira de clé étrangère lors de la jointure avec la table de la classe mère.

Exemple (code Java 5.0) :

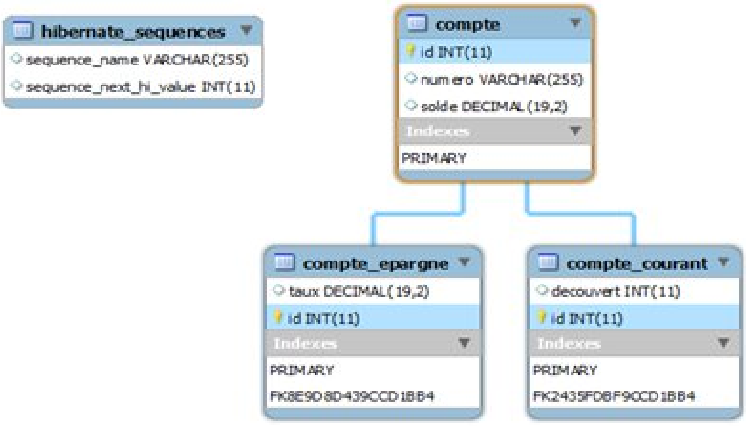
```
package fr.jmdoudoux.dej.hibernate.entity;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.Table;

@Entity
@Table(name = "compte_courant")
@PrimaryKeyJoinColumn(name = "id")
public class CompteCourant extends Compte {
  @Column(name = "decouvert")
  private int decouvert;

  public int getDecouvert() {
    return decouvert;
  }

  public void setDecouvert(int decouvert){
    this.decouvert = decouvert;
  }

  @Override
  public String toString() {
    return this.getClass().getName() + "@" + System.identityHashCode(this)
        + "CompteCourant [id=" + id + ", numero=" + numero + ", solde=" + solde
        + ", decouvert=" + decouvert + "]";
  }
}
```

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.hibernate.entity;

import java.math.BigDecimal;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.Table;

@Entity
@Table(name = "compte_epargne")
@PrimaryKeyJoinColumn(name = "id")
public class CompteEpargne extends Compte {
  @Column(name = "taux")
  private BigDecimal taux;

  public BigDecimal getTaux() {
    return taux;
  }

  public void setTaux(BigDecimal taux) {
    this.taux = taux;
  }

  @Override
  public String toString() {
    return this.getClass().getName() + "@" + System.identityHashCode(this)
        + "CompteEpargne [ id=" + id + ", numero=" + numero + ", solde="
        + solde + ", taux=" + taux + "]";
  }
}
```

Résultat :

```
2015-03-05 21:04:07.364 INFO   [main]:org.hibernate.annotations.common.Version - HCANN000001:
Hibernate Commons Annotations {4.0.1.Final}
2015-03-05 21:04:07.373 INFO   [main]:org.hibernate.Version - HHH000412: Hibernate Core {4.1.
4.Final}
2015-03-05 21:04:07.375 INFO   [main]:org.hibernate.cfg.Environment - HHH000206: hibernate.
properties not found
2015-03-05 21:04:07.377 INFO   [main]:org.hibernate.cfg.Environment - HHH000021: Bytecode
provider name : javassist
2015-03-05 21:04:07.405 INFO   [main]:org.hibernate.cfg.Configuration - HHH000043:
Configuring from resource: /hibernate.cfg.xml
2015-03-05 21:04:07.405 INFO   [main]:org.hibernate.cfg.Configuration - HHH000040:
Configuration resource: /hibernate.cfg.xml
2015-03-05 21:04:07.463 INFO   [main]:org.hibernate.cfg.Configuration - HHH000041: Configured
SessionFactory: null
2015-03-05 21:04:07.641 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000402: Using Hibernate built-in connection pool (not
for production use!)
2015-03-05 21:04:07.648 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000115: Hibernate connection pool size: 1
2015-03-05 21:04:07.648 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000006: Autocommit mode: false
2015-03-05 21:04:07.649 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000401: using driver [com.mysql.jdbc.Driver] at URL [
jdbc:mysql://localhost:3307/mabdd]
2015-03-05 21:04:07.649 INFO   [main]:org.hibernate.service.jdbc.connections.internal.
DriverManagerConnectionProviderImpl - HHH000046: Connection properties: {user=root, password=
****}
2015-03-05 21:04:07.955 INFO   [main]:org.hibernate.dialect.Dialect - HHH000400: Using
dialect: org.hibernate.dialect.MySQL5Dialect
2015-03-05 21:04:07.983 INFO   [main]:org.hibernate.engine.transaction.internal.
TransactionFactoryInitiator - HHH000399: Using default transaction strategy (direct JDBC
transactions)
2015-03-05 21:04:07.988 INFO   [main]:org.hibernate.hql.internal.ast.
ASTQueryTranslatorFactory - HHH000397: Using ASTQueryTranslatorFactory
2015-03-05 21:04:08.211 INFO   [main]:org.hibernate.tool.hbm2ddl.SchemaExport - HHH000227:
Running hbm2ddl schema export
Hibernate: alter table compte_courant drop foreign key FK2435FDBF9CCD1BB4
Hibernate: alter table compte_epargne drop foreign key FK8E9D8D439CCD1BB4
Hibernate: drop table if exists compte
Hibernate: drop table if exists compte_courant
Hibernate: drop table if exists compte_epargne
Hibernate: create table compte (id integer not null auto_increment, numero varchar(255), solde
decimal(19,2), primary key (id))
Hibernate: create table compte_courant (decouvert integer, id integer not null, primary key (
id))
Hibernate: create table compte_epargne (taux decimal(19,2), id integer not null, primary key (
id))
Hibernate: alter table compte_courant add index FK2435FDBF9CCD1BB4 (id), add constraint
FK2435FDBF9CCD1BB4 foreign key (id) references compte (id)
Hibernate: alter table compte_epargne add index FK8E9D8D439CCD1BB4 (id), add constraint
FK8E9D8D439CCD1BB4 foreign key (id) references compte (id)
2015-03-05 21:04:08.315 INFO     [main]:org.hibernate.tool.hbm2ddl.SchemaExport - HHH000230:
Schema export complete
Hibernate: insert into compte (numero, solde) values (?, ?)
Hibernate: insert into compte (numero, solde) values (?, ?)
Hibernate: insert into compte_courant (decouvert, id) values (?, ?)
Hibernate: insert into compte (numero, solde) values (?, ?)
Hibernate: insert into compte_epargne (taux, id) values (?, ?)
Recherche d'un compte
Hibernate: select compte0_.id as id0_0_, compte0_.numero as numero0_0_, compte0_.solde as
solde0_0_, compte0_1_.decouvert as decouvert1_0_, compte0_2_.taux as taux2_0_, case when
compte0_1_.id is not null then 1 when compte0_2_.id is not null then 2 when compte0_.id is not
null then 0 end as clazz_0_ from compte compte0_ left outer join compte_courant compte0_1_ on
compte0_.id=compte0_1_.id left outer join compte_epargne compte0_2_ on compte0_.id=compte0_2_.
id where compte0_.id=?
fr.jmdoudoux.dej.hibernate.entity.Compte@24ad9b [id=1, numero=000012345000, solde=0.00]

Recherche d'un compte courant
Hibernate: select comptecour0_.id as id0_0_, comptecour0_1_.numero as numero0_0_,
comptecour0_1_.solde as solde0_0_, comptecour0_.decouvert as decouvert1_0_ from compte_courant
comptecour0_ inner join compte comptecour0_1_ on comptecour0_.id=comptecour0_1_.id where
comptecour0_.id=?
fr.jmdoudoux.dej.hibernate.entity.CompteCourant@29984851CompteCourant [id=2, numero=
000012345010, solde=1200.00, decouvert=2000]

Recherche polymorphique d'un compte
Hibernate: select compte0_.id as id0_0_, compte0_.numero as numero0_0_, compte0_.solde as
solde0_0_, compte0_1_.decouvert as decouvert1_0_, compte0_2_.taux as taux2_0_, case when
compte0_1_.id is not null then 1 when compte0_2_.id is not null then 2 when compte0_.id is not
null then 0 end as clazz_0_ from compte compte0_ left outer join compte_courant compte0_1_ on
compte0_.id=compte0_1_.id left outer join compte_epargne compte0_2_ on compte0_.id=compte0_2_.
id where compte0_.id=?
fr.jmdoudoux.dej.hibernate.entity.CompteEpargne@6285813CompteEpargne [ id=3, numero=
000012345020, solde=8000.00, taux=2.10]

Recherche de tous les comptes
Hibernate: select compte0_.id as id0_, compte0_.numero as numero0_, compte0_.solde as solde0_,
compte0_1_.decouvert as decouvert1_, compte0_2_.taux as taux2_, case when compte0_1_.id is not
null then 1 when compte0_2_.id is not null then 2 when compte0_.id is not null then 0 end as
clazz_ from compte compte0_ left outer join compte_courant compte0_1_ on compte0_.id=
compte0_1_.id left outer join compte_epargne compte0_2_ on compte0_.id=compte0_2_.id
fr.jmdoudoux.dej.hibernate.entity.CompteCourant@15462959CompteCourant [id=2, numero=
000012345010, solde=1200.00, decouvert=2000]
fr.jmdoudoux.dej.hibernate.entity.CompteEpargne@12544708CompteEpargne [ id=3, numero=
000012345020, solde=8000.00, taux=2.10]
fr.jmdoudoux.dej.hibernate.entity.Compte@fa8f2c [id=1, numero=000012345000, solde=0.00]

Recherche polymorphique de comptes
Hibernate: select compte0_.id as id0_, compte0_.numero as numero0_, compte0_.solde as solde0_,
compte0_1_.decouvert as decouvert1_, compte0_2_.taux as taux2_, case when compte0_1_.id is not
null then 1 when compte0_2_.id is not null then 2 when compte0_.id is not null then 0 end as
clazz_ from compte compte0_ left outer join compte_courant compte0_1_ on compte0_.id=
compte0_1_.id left outer join compte_epargne compte0_2_ on compte0_.id=compte0_2_.id where
compte0_.numero like ?
fr.jmdoudoux.dej.hibernate.entity.CompteCourant@9037617CompteCourant [id=2, numero=
000012345010, solde=1200.00, decouvert=2000]
fr.jmdoudoux.dej.hibernate.entity.CompteEpargne@27879211CompteEpargne [ id=3, numero=
000012345020, solde=8000.00, taux=2.10]
fr.jmdoudoux.dej.hibernate.entity.Compte@1a4654d [id=1, numero=000012345000, solde=0.00]
```



Résultat :

```
Enter password: ****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 82
Server version: 5.6.15 MySQL Community Server (GPL)

Copyright (c) 2000, 2013, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates.
Other names may be trademarks of their respective owners.
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
mysql> use mabdd
Database changed
mysql> select * from compte;
+----+--------------+---------+
| id | numero       | solde   |
+----+--------------+---------+
|  1 | 000012345000 |    0.00 |
|  2 | 000012345010 | 1200.00 |
|  3 | 000012345020 | 8000.00 |
+----+--------------+---------+
3 rows in set (0.00 sec)
mysql> select * from compte_courant;
+-----------+----+
| decouvert | id |
+-----------+----+
|      2000 |  2 |
+-----------+----+
1 row in set (0.00 sec)
mysql> select * from compte_epargne;
+------+----+
| taux | id |
+------+----+
| 2.10 |  3 |
+------+----+
1 row in set (0.00 sec)
mysql>
```

62.12. Les caches d'Hibernate

Un volume important d'échanges entre l'application et la base de données est fréquemment à l'origine des problèmes de performance d'une application.

Le but d'un cache est de réduire les échanges entre l'application et la base de données en stockant en mémoire les entités déjà lues de la base de données. Une lecture dans la base de données n'est alors nécessaire que lorsque l'entité n'est pas déjà présente dans le cache (si l'entité n'a jamais été lue ou si l'entité a été retirée du cache).

Le cache permet de stocker des données qui ont déjà été lues de la base de données, ce qui permet de réduire les échanges entre la base de données et l'application lorsque celle-ci a de nouveau besoin des données. L'accès à la mémoire est beaucoup plus rapide que l'accès à la base de données surtout si celui-ci nécessite un échange utilisant le réseau.

Hibernate propose une fonctionnalité de mise en cache des données qui peut permettre, si elle est correctement configurée et utilisée, d'améliorer les performances de l'application. Il est nécessaire de bien comprendre le mode de fonctionnement du cache pour l'utiliser à bon escient et le configurer en conséquence : dans le cas contraire, les performances peuvent se dégrader de façon sérieuse voire dramatique.

L'utilisation du cache peut donc être une solution à certaines problématiques de performance fréquemment rencontrées lorsqu'Hibernate n'est pas bien compris ou n'est pas utilisé de manière optimale. Même si ces deux critères sont parfaitement maîtrisés, le cache peut être utile pour améliorer les performances dans certaines circonstances notamment lors de la lecture de données statiques dans la base de données.

Hibernate propose d'utiliser un cache pour plusieurs types de fonctionnalités de manière à améliorer les performances en lecture et/ou écriture des entités dans la base de données :

- cache de premier niveau : son utilisation est implicite car il est toujours actif et est utilisé par défaut. Implémenté dans la session, son champ d'action est limité à la transaction courante
- cache de second niveau : son utilisation est optionnelle ; il doit être activé et configuré pour pouvoir être utilisé. Implémenté dans l'objet de type SessionFactory, son champ d'action est l'application : il est donc utilisable par toutes les transactions
- le cache des requêtes : son utilisation est optionnelle ; il doit être activé et configuré pour pouvoir être utilisé. Sa mise en oeuvre utilise le cache de second niveau.

Une bonne connaissance de la façon dont ces caches fonctionnent et de la manière dont ils interagissent avec les autres API est essentielle pour obtenir les meilleurs résultats. Activer le cache et configurer les entités est facile mais cela ne garantit pas de pouvoir tirer le meilleur parti des fonctionnalités proposées par les caches d'Hibernate.

Un objet de type Session est une unité de travail, qui correspond à une transaction côté base de données : elle stocke l'état sur les entités suite à des lectures et/ou modifications et/ou suppressions. Ces modifications et suppressions sont transformées en requêtes SQL pour être reportées dans la base de données. Le stockage de cet état représente le premier niveau de cache d'Hibernate.

Dans le cache de premier niveau, implémenté dans la session, les entités sont directement stockées dans le cache. Si une entité est de nouveau récupérée de la session alors qu'elle a déjà été chargée c'est le même objet encapsulant l'entité qui est retourné. Ceci ne pose aucun problème puisque la portée de la session est la transaction courante. Ce mode de fonctionnement est utilisé jusqu'à ce que la transaction soit terminée ou que la méthode flush de la session soit invoquée.

Le cache de premier niveau est essentiellement utilisé pour limiter le nombre de requêtes SQL requises par la transaction : par exemple, si une entité est modifiée plusieurs fois durant la transaction, l'état final de l'entité est stocké dans la session qui ne générera qu'une seule requête SQL de type update, par défaut à la fin de la transaction.

Le cache de second niveau stocke les objets lus de la base de données au niveau de l'objet SessionFactory : les entités sont donc partagées entre les transactions et utilisables au niveau de l'application. Dans ce cas, lorsqu'une transaction a besoin de lire une entité, si celle-ci est déjà présente dans le cache alors l'exécution d'une ou plusieurs requêtes SQL est évitée.

L'utilisation du cache de second niveau est optionnelle. La durée de vie du cache de second niveau est liée à celle de l'application. La durée de vie des entités contenues dans le cache est configurable selon le cache utilisé. Le cache de second niveau requiert l'utilisation d'une implémentation d'un cache par un tiers.

Par défaut, si le cache de second niveau est activé, la recherche d'une entité se fait prioritairement dans le cache de premier niveau, puis dans le cache de second niveau et enfin dans la base de données par l'exécution d'une requête SQL.

Hibernate propose une fonctionnalité qui permet de mettre en cache le résultat de requêtes SQL. Cette fonctionnalité possède des contraintes mais peut être intéressante dans certains cas de figure.

62.12.1. Des recommandations pour l'utilisation des caches

La bonne compréhension du mode de fonctionnement et de la configuration des caches d'Hibernate est importante pour permettre d'en tirer le meilleur parti. C'est notamment le cas du cache de second niveau qui peut, en étant correctement configuré et utilisé, améliorer les performances d'Hibernate.

Attention : l'utilisation d'un cache ne peut pas être l'unique solution aux problèmes de performance avec Hibernate. Comme pour d'autres besoins, le cache peut être une solution pour certaines problématiques de performances mais il induit aussi d'autres soucis (fraicheur des données, durée avant l'invalidation des données, réplication des données dans un cluster, ...). De plus, mal utilisé ou configuré, l'utilisation du cache peut aussi dégrader les performances ou poser des problèmes d'accès concurrents.

La gestion de la fraicheur des données est une problématique fréquente lors de l'utilisation d'un cache mais elle devient cruciale dans le cas du cache des entités d'Hibernate.

Généralement par méconnaissance, Hibernate est souvent perçu comme peu performant car générateur de nombreuses requêtes SQL. Il est important de connaître son mode de fonctionnement afin de s'y adapter pour obtenir les meilleurs résultats. Sa première approche est plutôt facile mais elle masque aux développeurs un moteur complexe dont la connaissance du fonctionnement est requise pour ne pas être déçu et même pour ne pas avoir de gros problèmes d'utilisation généralement relatifs à la performance.

L'activation du cache de second niveau est facile mais il est très important de bien comprendre comment il fonctionne dans les situations où il peut s'appliquer pour éviter d'avoir un surcoût lié à l'utilisation du cache sans en obtenir les bénéfices.

Il est aussi important de noter que les caches ne sont jamais informés des modifications qui sont faites sur les données de la base par des applications tierces. Par exemple, Hibernate n'est pas capable de savoir si une donnée mise en cache est modifiée par une autre application ou par la base de données elle-même (exécution de procédures stockées ou de triggers).

Si ce cas de figure se présente, il est nécessaire de configurer la région pour qu'elle s'invalide périodiquement dans un délais plutôt court ceci afin de régulièrement retirer les données qu'elle contient. Cette périodicité est à définir selon les besoins.

62.12.2. Les différents caches d'Hibernate

Plusieurs actions peuvent faire réaliser une requête de type SQL par Hibernate :

- obtenir une entité par son identifiant, par exemple en utilisant la méthode find() ou get()
- obtenir une entité lors du parcours d'une relation puisque par défaut seuls leurs identifiants sont encapsulés dans un objet de type proxy
- obtenir une entité en utilisant une requête HQL ou l'API Criteria

L'utilisation d'un cache a pour but d'améliorer les performances : il se place entre l'application et la base de données pour stocker des données et ainsi éviter de les rechercher systématiquement de la base de données. L'objectif principal des caches est de limiter la répétition des requêtes de type select en stockant les entités lues pour les obtenir du cache lors des invocations suivantes.

Hibernate propose deux types de caches, chacun ayant un but précis :

- le cache de niveau 1 ou cache de premier niveau : il est toujours actif car il est implémenté dans la classe Session. La visibilité de ce cache est la transaction. Sa durée de vie est donc celle de la session. Le but est de réduire le nombre de requêtes SQL d'une même transaction.
- le cache de niveau 2 ou cache de second niveau : il n'est pas actif par défaut. Il est implémenté dans la SessionFactory. La visibilité de ce cache est l'application. Une fois activé, sa durée de vie est celle de l'instance de type SessionFactory. Le rôle principal du cache de second niveau est de partager des données entre sessions.

L'utilisation du cache de premier niveau est obligatoire puisqu'il est implémenté dans l'objet de type Session : toutes les requêtes permettant l'obtention ou la mise à jour des données passent par la session. Une fois que la session est fermée, le contenu du cache de premier niveau est effacé.

Pour réduire les échanges entre l'application et la base de données, le cache de second niveau conserve les données lues de la base de données pour les partager entre les sessions. Ces données stockées au niveau de la SessionFactory sont donc accessibles par toutes les sessions, évitant ainsi l'exécution de requêtes SQL si les résultats de celles-ci sont déjà dans le cache.

Le cache de second niveau est optionnel. Le cache de premier niveau est toujours consulté avant de consulter de cache de second niveau.

Le cache de second niveau peut être utilisé pour stocker trois types d'éléments correspondant à trois cas d'utilisations du cache:

- les entités (class cache)
- les associations de type many (collection cache)
- le résultat des requêtes (query cache)

L'implémentation du cache de second niveau requiert l'utilisation d'une solution de cache d'un tiers. Cette solution doit proposer une classe qui implémente l'interface org.hibernate.cache.CacheProvider.

La configuration du cache de second niveau requiert plusieurs étapes :

- définir les stratégies transactionnelles qui devront être utilisées par le cache de second niveau pour chaque entité et association
- choisir une implémentation du cache qui supporte la ou les stratégies requises en plus des critères de choix plus généraux (fiabilité, performance, fonctionnalités, documentation, ...)
- configurer Hibernate pour utiliser le cache
- configurer de manière spécifique le cache

Le cache des requêtes permet de conserver le résultat de l'exécution des requêtes. Ce cache utilise la requête avec ses paramètres comme clé à laquelle il associe uniquement les identifiants et les types des entités qui sont obtenus lors de l'exécution de la requête. Il repose sur l'utilisation du cache de second niveau pour obtenir les données des entités concernées.

L'utilisation du cache des requêtes est optionnelle : il n'est pas activé par défaut. Il requiert la définition de deux régions particulières dans le cache de second niveau :

- la première stocke les résultats (identifiants et types) des requêtes
- la seconde stocke le timestamp de dernière mise à jour de chaque table

Ce cache est essentiellement utile pour les requêtes fréquemment exécutées avec les mêmes valeurs de paramètres.

62.12.2.1. La base des exemples de cette section

Les exemples de cette section utilisent la version 4.1 d'Hibernate.

Cette section va utiliser une petite application qui accède à une base de données composées de deux tables : pays et devise. Le besoin fonctionnel précise qu'l existe une relation 1-N entre devise et pays partant du principe qu'un pays possède une devise et qu'une devise peut être utilisée par plusieurs pays.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate
Mapping DTD//EN" "http://hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="fr.jmdoudoux.dej.hibernatecache.entity.Pays"
       table="pays" >
        <id name="id" column="id" type="int" >
            <generator class="identity" />
        </id>
        <property name="codeIso" column="code_iso"
           not-null="true" type="string" />
        <property name="nom" not-null="true" type="string" />
        <many-to-one name="devise" column="FK_DEVISE"
           class="fr.jmdoudoux.dej.hibernatecache.entity.Devise" />
    </class>
</hibernate-mapping>
```

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC  "-//Hibernate/Hibernate Mapping
DTD//EN" "http://hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="fr.jmdoudoux.dej.hibernatecache.entity.Devise"
      table="DEVISE" >
        <id name="id" column="ID" >
            <generator class="identity" />
        </id>
        <property name="code" column="CODE" />
        <property name="libelle" column="LIBELLE" />
        <set name="pays" >
           <key column="FK_DEVISE" />
           <one-to-many class="fr.jmdoudoux.dej.hibernatecache.entity.Pays"/>
        </set>
    </class>
</hibernate-mapping>
```

Dans le fichier de configuration d'Hibernate, le cache de second niveau est désactivé et l'affichage des requêtes SQL exécutées est demandé.

Résultat :

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/mabdd</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password"></property>
    <property name="hibernate.connection.pool_size">1</property>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.current_session_context_class">thread</property>
    <!-- Cache de second niveau désactivé  -->
    <property name="hibernate.cache.provider_class">
      org.hibernate.cache.internal.NoCacheProvider</property>
    <!-- Afficher les requêtes SQL exécutées sur la sortie standard -->
    <property name="hibernate.show_sql">true</property>
    <mapping resource="Pays.hbm.xml" />
    <mapping resource="Devise.hbm.xml" />
  </session-factory>
</hibernate-configuration>
```

62.12.2.2. Le cache de premier niveau

Le cache de premier niveau est implémenté dans la classe Session. Hibernate l'utilise par défaut pour maintenir l'état des entités lues et modifiées dans la transaction courante.

Une entité est stockée dans le cache de la session, c'est à-dire le cache de premier niveau, à chaque fois :

- que les méthodes load() ou get() sont invoquées
- que les méthodes save(), update() ou saveOrUpdate() sont invoquées

Le cache de premier niveau est activé par défaut et il n'est pas possible de le désactiver.

Le cache de premier niveau permet à Hibernate de conserver les données lues et celles qui sont modifiées : ceci permet de limiter le nombre de requêtes exécutées pour les lectures et les mises à jour sur la base de données.

Grâce au cache de premier niveau, sur une même session, plusieurs invocations de la méthode get() qui retourneraient la même entité ne nécessiteront qu'une seule requête SQL lors de la première invocation. Pour les invocations suivantes, les données seront obtenues directement du cache sans relecture dans la base de données.

Exemple :

```
  public static void lirePaysDeuxFois() throws Exception {
    Session session = sessionFactory.openSession();
    Transaction tx = null;
    try {
      tx = session.beginTransaction();
      Pays pays = (Pays) session.load(Pays.class, new Integer(4));
      System.out.println("pays : id=" + pays.getId() + " codeIso="
          + pays.getCodeIso() + " nom=" + pays.getNom());
      pays = (Pays) session.load(Pays.class, new Integer(4));
      System.out.println("pays : id=" + pays.getId() + " codeIso="
          + pays.getCodeIso() + " nom=" + pays.getNom());
      tx.commit();
    } catch (Exception e) {
      if (tx != null) {
        tx.rollback();
      }
      throw e;
    } finally {
      session.close();
    }
  }
```

Résultat :

```
Hibernate: select pays0_.id as
id0_0_, pays0_.code_iso as code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE
as FK4_0_0_ from pays pays0_ where pays0_.id=?
pays : id=4 codeIso=LU
nom=Luxembourg
pays : id=4 codeIso=LU
nom=Luxembourg
```

Bien que la méthode load() soit invoquée deux fois, l'entité n'est lue qu'une seule fois dans la base de données et est stockée dans le cache de premier niveau. Lors de la seconde invocation de la méthode load(), l'entité est obtenue du cache de premier niveau évitant ainsi une seconde lecture dans la base de données.

Hibernate utilise toujours de manière transparente le cache de premier niveau.

L'interface Session propose plusieurs méthodes pour gérer le contenu du cache de premier niveau.

Lorsque la méthode flush() de la session est invoquée, l'état des entités contenues dans la session est synchronisé dans la base de données. Ces entités restent dans la session jusqu'à sa fermeture.

La méthode contains() renvoie un booléen qui précise si une instance d'une entité est associée à la session.

La méthode clear() permet de supprimer toutes les entités contenues dans la session.

La méthode evict() permet de retirer une entité de la session et les entités qui lui sont associées si l'attribut cascade de l'association à pour valeur all ou all-delete-orphan. Cette méthode est particulièrement pratique lorsque des requêtes retournent de nombreuses entités et que ces dernières peuvent être retirées du cache car devenues inutiles.

62.12.2.3. Le cache de second niveau

Dès que l'on utilise le cache de second niveau, il ne faut pas oublier que le cache de premier niveau est toujours actif. Hibernate tente de trouver une entité dans le cache de premier niveau, si elle n'est pas trouvée, Hibernate tente de la trouver dans le cache de second niveau. Si elle n'est toujours pas trouvée alors elle est lue de la base de données.

L'utilisation du cache de premier niveau est transparente car complètement prise en charge par Hibernate. L'utilisation du cache de second niveau est plus complexe : les données contenues dans le cache sont partagées entre les transactions et peuvent même être partagées entre plusieurs JVM.

Le cache de second niveau est partagé entre toutes les sessions alors que la portée du cache de premier niveau est au niveau d'une seule session. Hibernate propose une API pour permettre de choisir l'implémentation du cache qui sera utilisée pour le cache de second niveau.

Le cache de second niveau est implémenté dans un objet de type SessionFactory : généralement une application ne possède qu'une seule instance de ce type d'objet. La durée de vie du cache de second niveau est donc liée à la durée de vie de l'instance de type SessionFactory. L'accès à cet objet est commun à toutes les sessions. Le contenu du cache de second niveau peut donc être partagé entre plusieurs sessions.

La portée du cache de second niveau est la JVM ou un cluster si l'implémentation du cache supporte l'utilisation dans ce cadre.

Le cache de second niveau propose un mécanisme puissant qui peut permettre, dans des cas précis, d'améliorer les performances d'une application. Cependant, l'utilisation du cache de second niveau introduit une complexité supplémentaire : par exemple, l'utilisation de caches augmente toujours le risque d'obtenir des données inconsistantes.

Les données en lecture seule dans le cache sont faciles à utiliser et à gérer. Dans le cache, la gestion et l'utilisation des données pouvant être mises à jour est plus délicate.

L'usage du cache de second niveau le rend surtout intéressant pour des données en lecture seule. Des données contenues dans le cache pouvant être mises à jour risque de ne pas être fraîches. Il est alors très important de bien définir la politique d'éviction des entrées dans le cache, surtout si la fraîcheur des données est importante.

Les objets lus de la base de données peuvent être stockés dans le cache de second niveau et ainsi être partagés par les sessions pour éviter leur relecture par les autres sessions. Bien sûr ce partage est facile si les données sont uniquement lues et jamais modifiées. Ce partage est plus complexe lorsque les données peuvent être mises à jour par une session.

Le cache de second niveau ne peut jamais être informé d'une modification réalisée sans utiliser Hibernate. C'est le cas notamment si ces modifications sont réalisées par d'autres applications, des procédures stockées, des triggers, ... Dans ce cas, il faut utiliser des mécanismes qui permettent de supprimer une entité du cache ou le contenu d'une région ou prévoir des suppressions périodiques du contenu du cache.

Hibernate impose une utilisation sélective du cache de second niveau car il ne peut pas être appliqué sur toutes les opérations. Le cache de second niveau ne s'utilise que sur trois types d'éléments :

- les données des entités
- les associations de type many
- le résultat des requêtes

Par défaut, le cache de second niveau n'est pas activé : pour l'utiliser, il est nécessaire de l'activer et de le configurer. Une fois le cache de second niveau activé, seuls les éléments configurés (entités, associations, requêtes) pourront y être stockés.

Le cache de premier niveau stocke directement les instances des entités. Le cache de second niveau, lui, stocke les entités sous une forme «sérialisée». Les entités ne sont pas stockées sous la forme d'instance : Hibernate stocke les attributs de l'entité sous une forme nommée état déshydraté (dehydrated state). Lorsqu'une entité est obtenue du cache, une nouvelle instance est créée à partir des informations stockées dans le cache. La clé dans le cache est l'identifiant de l'entité et la valeur est la forme déshydratée de l'entité.

Il est très important de comprendre le fonctionnement du cache de second niveau pour en tirer le meilleur parti.

62.12.3. La configuration du cache de second niveau

Par défaut, le cache de second niveau n'est pas activé. Son activation se fait en modifiant la configuration d'Hibernate.

La propriété hibernate.cache.use_second_level permet d'activer le cache de second niveau en lui affectant la valeur true.

Hibernate propose une solution de type plug-in pour lui permettre d'utiliser une implémentation d'un cache tiers comme cache de second niveau.

Hibernate offre une implémentation reposant sur un objet de type HashTable mais son utilisation n'est pas recommandée car elle ne propose pas les fonctionnalités minimum d'un cache (gestion de sa taille, de la durée de vie des éléments contenus, ...)

La plupart des principaux caches open source peuvent s'utiliser avec Hibernate moyennant un peu de configuration. Une partie de cette configuration est faite dans le fichier de configuration d'Hibernate en utilisant plusieurs propriétés.

La propriété hibernate.cache.provider_class permet de préciser le nom pleinement qualifié de la classe du fournisseur du cache jusqu'à la version 3.2 d'Hibernate.

Exemple pour ne préciser aucun fournisseur

Exemple :

```
    <property name="hibernate.cache.provider_class">
org.hibernate.cache.internal.NoCacheProvider</property>
```

A partir de la version 3.3, c'est la propriété hibernate.cache.region.factory_class qui doit être utilisée.

Exemple pour ne préciser aucun fournisseur

Exemple :

```
    <property name="hibernate.cache.region.factory_class">
org.hibernate.cache.internal.NoCachingRegionFactory</property>
```

La propriété hibernate.cache.use_structured_entries à la valeur true permet de demander à Hibernate de stocker les données dans le cache de manière lisible.

La propriété hibernate.cache.use_minimal_puts permet de demander à Hibernate de limiter les écritures dans le cache, ce qui entrainera plus de lectures dans la base de données. La valeur true est intéressante pour les caches en cluster.

La propriété hibernate.cache.use.region_prefix permet de préciser un préfixe qui sera utilisé pour le nom de chaque région du cache de second niveau.

La propriété hibernate.cache.defaut_cache_concurrency_strategy permet de préciser la stratégie d'usage transactionnel utilisée par l'annotation @Cache. Il est possible de remplacer cette stratégie définie par défaut en utilisant l'attribut strategy de l'annotation @Cache.

62.12.3.1. Les différents caches supportés par Hibernate

Hibernate fournit en standard une implémentation du cache utilisant des objets de type HashTable : c'est une solution minimaliste qu'il n'est pas recommandé d'utiliser en production.

Hibernate propose le support en standard de plusieurs solutions open source comme implémentation du cache de second niveau :

- Terracotta Ehcache (Easy Hibernate Cache)
- OSCache (Open Symphony Cache)

- Swarm Cache
- JBoss Tree Cache : cache utilisable en cluster qui nécessite un gestionnaire de transactions

Chacun de ces caches possède des caractéristiques et des fonctionnalités :

- Ehcache : léger, rapide, facile à configurer et à utiliser, supporte les stratégies read-only, nonstrict-read-only et read-write, cache mémoire avec débordement sur disque, support du cluster
- OSCache : supporte les stratégies read-only et read-write, cache mémoire avec débordement sur disque, support du cluster en utilisant JGroups ou JMS
- SwarmCache : supporte les stratégies read-only et nonstrict-read-write, support du cluster en utilisant JGroups, approprié pour des applications qui réalisent plus de lectures que d'écritures
- JBoss TreeCache : riche en fonctionnalités et répliqué, supporte la stratégie transactional, utilisation de la classe org.hibernate.cache.TreeCacheProvider (pour la version 1) et org.hibernate.cache.jbc.JbossCacheRegionFactory (pour la version 2)

Avant la version 3.2, Hibernate utilisait par défaut Ehcache. A partir de la version 3.2 d'Hibernate, il n'y a plus d'implémentation par défaut : celle à utiliser doit être explicitement précisée dans la configuration d'Hibernate.

Hibernate impose de n'utiliser qu'une seule implémentation pour le cache de second niveau : il faut donc la choisir judicieusement en fonction des besoins.

Cache	Type	Support en Cluster	Cache de requêtes supporté
Hashtable (ne pas utiliser en production)	Mémoire		Oui
Ehcache	Mémoire, disque	Oui	Oui
OSCache	Mémoire, disque		Oui
SwarmCache	En cluster (multicast ip)	Oui (invalidation de cluster)	
JBoss TreeCache	En cluster (multicast ip), transactionnel	Oui (replication)	Oui

62.12.3.2. La configuration du cache de second niveau

Une implémentation du cache doit fournir une classe qui implémente l'interface org.hibernate.cache.CacheProvider (jusqu'à la version 3.2) ou l'interface org.hibernate.RegionFactory (à partir de la version 3.3).

Il est nécessaire de préciser la classe du fournisseur qui doit être utilisée en donnant son nom pleinement qualifié comme valeur d'une propriété dans le fichier de configuration d'Hibernate.

La configuration du cache de second niveau est différente selon la version d'Hibernate utilisée :

- Jusqu'à la version 3.2, il faut utiliser la propriété hibernate.cache.provider_class
- A partir de la version 3.3, il faut utiliser la propriété hibernate.cache.region.factory_class

L'exemple ci-dessous permet d'utiliser le cache Ehcache avec Hibernate 3.2.

Exemple :

```
<property name="hibernate.cache.provider_class">
org.hibernate.cache.EhCacheProvider</property>
```

La configuration du cache dépend de l'implémentation utilisée.

Par défaut, la région utilisée est celle définie par défaut dans le cache. Il est aussi possible de définir et configurer des régions dédiées.

Hibernate utilise des conventions de nommage particulières pour les noms des régions du cache :

- pour les entités : c'est le nom pleinement qualifié de la classe de l'entité
- pour les associations : c'est le nom pleinement qualifié de la classe de l'entité, suivi d'un caractère point, suivi du nom du champ de la collection
- pour les requêtes : Hibernate utilise par défaut deux régions (StandardQueryCache et UpdateTimestampsCache).

Le nom de ces deux régions est différent selon la version d'Hibernate utilisée :

- jusqu'à Hibernate version 3.1 : net.sf.hibernate.cache.StandardQueryCache et net.sf.hibernate.cache.UpdateTimestampsCache
- à partir de la version 3.2 : org.hibernate.cache.StandardQueryCache et org.hibernate.cache.UpdateTimestampsCache

62.12.3.3. La configuration du cache Ehcache

La version d'Ehcache utilisée dans cette section est la version 2.4.3.

Dans le fichier de configuration d'Hibernate, il faut activer l'utilisation du cache de second niveau avec Ehcache en donnant à la valeur de la propriété hibernate.cache.region.factory_class le nom pleinement qualifié d'une classe qui hérite de AbstractEhcacheRegionFactory.

Deux implémentations sont fournies par Hibernate dans la bibliothèque hibernate-ehcache-xxx.jar :

- org.hibernate.cache.ehcache.EhCacheRegionFactory
- org.hibernate.cache.ehcache.SingletonEhCacheRegionFactory : cette implémentation est à utiliser lorsqu'une seule configuration est requise. Il ne faut pas l'utiliser si plusieurs instances d'Hibernate sont requises.

Exemple :

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!-- ... -->
    <!-- Cache de second niveau activé avec ehcache  -->
    <!-- property name="hibernate.cache.provider_class">
      org.hibernate.cache.internal.NoCacheProvider</property-->
    <property name="hibernate.cache.region.factory_class">
      org.hibernate.cache.ehcache.SingletonEhCacheRegionFactory</property>
    <!-- ... -->
  </session-factory>
</hibernate-configuration>
```

EhCache fournit aussi sa propre implémentation qui est différente selon la version d'Hibernate utilisée.

Jusqu'à la version 3.2 incluse, il faut utiliser la classe net.sf.ehcache.hibernate.EhCacheProvider ou la classe net.sf.ehcache.hibernate.SingletonEhCacheProvider comme valeur de la propriété hibernate.cache.provider_class.

A partir de la version 3.3, il faut utiliser la classe net.sf.ehcache.hibernate.EhCacheRegionFactory ou la classe net.sf.ehcache.hibernate.SingletonEhCacheRegionFactory comme valeur de la propriété hibernate.cache.region.factory_class.

La configuration d'EhCache se fait dans un fichier XML nommé par défaut ehcache.xml qui doit être dans le classpath. Cette configuration porte sur EhCache lui-même, la région par défaut et éventuellement chaque région utilisée.

Le tag racine de ce document XML est <ehcache>.

Le tag <diskStore> permet de configurer le stockage sur disque des données du cache.

L'attribut path permet de préciser le chemin du répertoire dans le lequel EhCache va stocker les données du cache sur disque.

EhCache créé selon la configuration des fichiers pour chaque cache concerné avec l'extension .index et .data.

Le tag <defaultCache> permet de configurer le cache par défaut.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache>
      <diskStore path="C:\temp\cache" />
      <defaultCache maxElementsInMemory="100" eternal="false"
            timeToIdleSeconds="120" timeToLiveSeconds="120" overflowToDisk="true" />
</ehcache>
```

La propriété net.sf.ehcache.configurationResourceName du fichier de configuration permet de préciser le nom du chemin du fichier de configuration d'EhCache.

Il est nécessaire de configurer la région par défaut et éventuellement des régions dédiées.

Par défaut, Hibernate va rechercher et utiliser une région dont le nom est le nom pleinement qualifié de l'entité qui doit être mise en cache. Si elle n'est pas trouvée, alors la région par défaut sera utilisée.

Résultat :

```
WARN
[main]:org.hibernate.cache.ehcache.AbstractEhcacheRegionFactory -
HHH020003: Could not find a specific ehcache configuration for cache named
[fr.jmdoudoux.dej.hibernatecache.entity.Pays]; using defaults.
```

La configuration d'une région se fait en utilisant le tag <cache>.

La configuration peut se faire en utilisant plusieurs attributs :

Attribut	Rôle
name	Le nom du cache. Il doit être unique et sera utilisé par Hibernate comme la désignation d'une région
maxElementsInMemory	Préciser le nombre d'éléments maximum que peut contenir la région en mémoire. Si le maximum est atteint et qu'un nouvel élément doit être inséré, alors il y a une éviction d'un élément selon la configuration. La valeur par défaut est 0, indiquant un nombre infini
timeToIdleSeconds	Durée de vie en secondes des objets inaccédés (TTI). La valeur par défaut est 0, indiquant un temps infini
timeToLiveSeconds	Durée de vie en secondes des objets dans le cache quelque soit leur utilisation ou non (TTL) . La valeur par défaut est 0, indiquant un temps infini
eternal	true ou false. La valeur true permet de préciser que les données contenues dans la région ne peuvent pas être retirées. Ce paramètre est prioritaire sur les TTI et TTL si sa valeur est true
overflowToDisk	true ou false : écriture d'éléments sur le système de fichiers si la région contient trop d'éléments
diskPersistent	true ou false. La valeur permet de demander de conserver les données écrites sur le système de fichiers lorsque la JVM est redémarrée. La valeur par défaut est false
diskExpiryThreadIntervalSeconds	Préciser l'intervalle en secondes entre chaque exécution de la règle d'éviction des données du cache sur disque. La valeur par défaut est 120.
memoryStoreEvictionPolicy	

	Règle à appliquer pour l'éviction de données lorsque la taille maximale du cache est atteinte. Les règles possibles sont : • LRU (Least Recently Used) • FIFO (First In First Out) • LFU (Least Frequently Used)
maxElementsOnDisk	Préciser le nombre maximum d'éléments du cache sur disque. La valeur par défaut est 0, indiquant un nombre illimité
diskSpoolBufferSizeMB	Préciser la taille d'un tampon qui est utilisé pour stocker temporairement les données à écrire de manière asynchrone sur disque. Chaque cache possède son propre tampon. La taille par défaut est 30Mo.

Il est important de bien tenir compte dans la configuration de l'expiration des données du cache.

Le paramètre TTI permet notamment de retirer du cache des éléments qui sont peu fréquemment utilisés et ainsi faire de la place aux nouveux éléments.

Le paramètre TTL permet de rafraîchir périodiquement des données en forçant leur éviction de manière répétée.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache>
      <diskStore path="C:\temp\cache" />
      <defaultCache maxElementsInMemory ="100" eternal="false"
            timeToIdleSeconds="120" timeToLiveSeconds="120" overflowToDisk="true" />
      <cache name="fr.jmdoudoux.dej.hibernatecache.entity.Pays"
            maxElementsInMemory="250" eternal="true" overflowToDisk="false" />
      <cache name="fr.jmdoudoux.dej.hibernatecache.entity.Section"
            maxElementsInMemory="100" eternal="false" timeToIdleSeconds="300"
            timeToLiveSeconds="600" overflowToDisk="true" />
</ehcache>
```

Un message est affiché dans le journal pour chaque entité configurée dans le cache EhCache.

Résultat :

```
WARN
[main]:org.hibernate.cache.ehcache.internal.strategy.EhcacheAccessStrategyFactoryImpl
- HHH020007: read-only cache configured for mutable entity
[fr.jmdoudoux.dej.hibernatecache.entity.Pays]
```

Pour le cache des relations, par défaut l'implémentation d'EhCache utilise une région qui se nomme du nom pleinement qualifié de l'entité suivi du caractère point et du nom de l'attribut de la relation.

Résultat :

```
WARN
[main]:org.hibernate.cache.ehcache.AbstractEhcacheRegionFactory -
HHH020003: Could not find a specific ehcache configuration for cache named
[fr.jmdoudoux.dej.hibernatecache.entity.Devise.pays]; using defaults.
```

Dans l'exemple ci-dessus, la région du cache concernant la relation 1-N entre devise et pays n'est pas configurée. Il suffit alors de définir une nouvelle région dans la configuration d'EhCache pour éviter l'utilisation de la configuration par défaut.

Pour le cache des requêtes, par défaut Hibernate utilise deux régions qui se nomment :

- org.hibernate.cache.spi.UpdateTimestampsCache

- org.hibernate.cache.internal.StandardQueryCache

Résultat :

```
INFO     [main]:org.hibernate.cache.spi.UpdateTimestampsCache
- HHH000250: Starting update timestamps cache at region:
org.hibernate.cache.spi.UpdateTimestampsCache
WARN
[main]:org.hibernate.cache.ehcache.AbstractEhcacheRegionFactory -
HHH020003: Could not find a specific ehcache configuration for cache named
[org.hibernate.cache.spi.UpdateTimestampsCache]; using defaults.
INFO
[main]:org.hibernate.cache.internal.StandardQueryCache - HHH000248:
Starting query cache at region: org.hibernate.cache.internal.StandardQueryCache
WARN
[main]:org.hibernate.cache.ehcache.AbstractEhcacheRegionFactory -
HHH020003: Could not find a specific ehcache configuration for cache named
[org.hibernate.cache.internal.StandardQueryCache]; using defaults.
```

Dans l'exemple ci-dessus, les deux régions du cache concernant les requêtes ne sont pas configurées. Il suffit alors de définir deux nouvelles régions dans la configuration d'EhCache pour éviter l'utilisation de la configuration par défaut.

Le nom de la région utilisée par Hibernate pour le cache des résultats des requêtes est org.hibernate.cache.StandardQueryCache.

Exemple :

```
<cache name="org.hibernate.cache.StandardQueryCache"
maxEntriesLocalHeap="50" eternal="false" timeToLiveSeconds="300"
overflowToDisk="true"/>
```

Le cache des requêtes utilise aussi une autre région dont le nom est org.hibernate.cache.UpdateTimestampsCache. Cette région contient la date/heure de dernière mise à jour de chaque table. Pour le bon fonctionnement du cache des requêtes, il est préférable que cette région ne soit jamais invalidée.

Exemple :

```
<cache name="org.hibernate.cache.UpdateTimestampsCache"
maxEntriesLocalHeap="500" eternal="true" overflowToDisk="true"/>
```

Il est possible de définir et d'utiliser une région dédiée pour une ou plusieurs requêtes. Le nom attribué dans la configuration de la région dans le fichier ehcache.xml doit être utilisé comme paramètre de la méthode setCacheRegion() pour les requêtes (Query ou Criteria) qui doivent être mises en cache. La valeur de l'attribut name n'est pas imposée mais par convention elle commence par "query." et ce nom doit être unique pour toutes les régions.

Exemple :

```
<cache name="query.paysPourUneMonnaie" maxEntriesLocalHeap="20"
eternal="false" timeToLiveSeconds="3600" overflowToDisk="true"/>
```

Enfin, il faut ajouter les bibliothèques requises pour EhCache dans le classpath. Celles-ci peuvent être trouvées dans le sous-répertoire lib/optional/ehcache.

62.12.4. Les stratégies d'usage transactionnel du cache

Une stratégie d'usage transactionnel permet de déterminer comment les éléments vont être obtenus et stockés dans le cache de second niveau et comment les accès concurrents vont être gérés.

Hibernate propose le support de quatre stratégies transactionnelles pour le cache de second niveau :

- read-only : utilisable pour des entités qui ne sont jamais mises à jour.
- read-write : utilisable pour des entités qui sont mises à jour occasionnellement en utilisant la sémantique du niveau d'isolation read committed. Cette stratégie impose un léger surcoût lors de son utilisation.
- nonstrict-read-write : utilisable pour des entités qui sont mises à jour occasionnellement. L'entité dans le cache n'est jamais verrouillée. Si un accès concurrent à une entité est fait, cette stratégie ne garantit pas que ce qui est retourné du cache soit l'image des données correspondantes dans la base de données. Elle ne propose donc aucune gestion des accès concurrents.
- transactional : utilisable uniquement dans un environnement possédant un gestionnaire de transactions distribuées respectant l'API JTA qui permet d'utiliser la sémantique du niveau d'isolation repeatable read.

Une stratégie d'usage transactionnel doit être précisée pour chaque entité et collection configurées pour être mises dans le cache de second niveau.

62.12.4.1. La stratégie read-only

La stratégie read-only est utilisable sur des données qui ne seront utilisées qu'en lecture seule et ne seront donc jamais mises à jour. Elle convient parfaitement à des entités de type données de référence.

La mise en oeuvre de cette stratégie est la plus simple et la plus performante, de plus elle est utilisable en cluster.

Il est préférable d'utiliser la stratégie read_only sur des entités dont l'attribut mutable possède la valeur false.

Remarque : l'utilisation de la stratégie read-only interdit les mises à jour sur une entité mais elle n'interdit pas l'ajout de nouvelles entités.

62.12.4.2. La stratégie read-write

La stratégie read-write est utilisable sur des données qui pourront être lues et/ou modifiées sans requérir un niveau d'isolation transactionnelle de type Serializable.

Les mises à jour des entités mises en cache sont aussi faites dans le cache. Ces mises à jour sont faites de manière concurrente grâce à la pose d'un verrou. Cette stratégie est similaire au niveau d'isolation read committed.

Avec la stratégie read-write, dès qu'une entité est mise à jour, un verrou est posé pour ses données dans le cache empêchant leur accès par d'autres sessions qui sont alors obligées de relire les données dans la base de données. Lorsque la transaction est validée, les données du cache sont rafraichies et le verrou est retiré.

Cette stratégie permet d'éviter d'avoir des dirty read et permet aux sessions d'obtenir des données de manière read committed aussi bien de la base de données que du cache.

Cette stratégie induit un léger surcoût lié au verrouillage des données dans le cache.

Il est important que la transaction soit terminée avant l'invocation de la méthode close() ou de la méthode disconnect() de la session.

Cette stratégie ne doit pas être utilisée si le niveau d'isolation des transactions requis est Serializable.

Pour pouvoir être utilisé dans un cluster, l'implémentation du cache doit permettre de poser les verrous dans les différents noeuds du cluster. C'est par exemple le cas du cache Coherence d'Oracle.

62.12.4.3. La stratégie nonstrict-read-write

La stratégie nonstrict_read_write est utilisable pour des données qui ne changent pas fréquemment voire même jamais. Elle ne vérifie pas que deux transactions mettent à jour des données qui sont dans le cache : aucun verrou n'est posé lors de ces modifications. Les accès concurrents sont donc possibles mais cette stratégie ne garantit pas que les données retournées seront les plus fraiches et donc le reflet de ce qui est dans la base de données.

Si l'application peut accepter que les données ne soient pas toujours consistantes et que les données ne soient fréquemment modifiées, l'utilisation de la stratégie nonstrict-read-write à la place de la stratégie read-write peut améliorer les performances.

La stratégie nonstrict-read-write retire les données du cache d'une entité mise à jour lorsque la méthode flush() de la session est invoquée.

La stratégie nonstrict-read-write ne pose jamais de verrou. Lorsque qu'un objet doit être modifié, les anciennes valeurs restent dans le cache jusqu'à ce que la transaction soit validée par un commit. Si une autre session veut accéder à l'objet, elle obtiendra les données du cache (dirty read). Dès que la transaction est validée, les données de l'entité sont supprimées du cache : lorsqu'une session voudra obtenir les données de l'entité, elle sera donc forcer de relire les données de la base de données et de les insérer dans le cache.

La stratégie nonstrict-read-write est donc utilisable si l'application peut supporter des dirty reads qui peuvent arriver lorsque les données sont répercutées dans la base de données sans être encore retirées du cache.

Avec la stratégie nonstrict-read-write, il est préférable de configurer l'invalidation périodique de la région afin de permettre d'améliorer la fraîcheur des données.

62.12.4.4. La stratégie transactional

La stratégie transactional doit être utilisée dans un environnement utilisant un gestionnaire de transactions distribuées de type JTA.

Cette stratégie n'est utilisable qu'avec un cache transactionnel comme JBoss TreeCache.

Avec les stratégies nonstrict-read-write et read-write, le cache est mis à jour de manière asynchrone une fois que la transaction est validée. Avec la stratégie transactional, le cache est mis à jour en même temps que la transaction est validée.

Il faut préciser à Hibernate le nom d'une classe qui sera instanciée pour lui permettre d'obtenir le gestionnaire de transactions du conteneur dans lequel l'application s'exécute. La valeur de cette propriété est donc dépendante de l'environnement d'exécution.

Comme l'accès au TransactionManager de JTA n'est pas standardisé, cette classe permet d'obtenir l'instance de l'environnement d'exécution correspondant.

Jusqu'à la version 3.6 incluse, la classe à utiliser doit implémenter l'interface TransactionManagerLookup. Il faut fournir son nom pleinement qualifié à la propriété hibernate.transaction.manager_lookup_class.

Conteneur	Propriété hibernate.transaction.manager_lookup_class
Oracle OAS	org.hibernate.transaction.OC4JtransactionManagerLookup
JBoss AS	org.hibernate.transaction.JbossTransactionManagerLookup
JBoss Transactions	org.hibernate.transaction.JbossTSStandaloneTransactionManagerLookup
GlassFish	org.hibernate.transaction.SunONETransactionManagerLookup
JOTM	org.hibernate.transaction.JOTMTransactionManagerLookup
IBM Websphere AS 4 à 5.1	org.hibernate.transaction.WebSphereTransactionManagerLookup
IBM Websphere AS 6 et supérieur	org.transaction.WebSphereExtendedJTATransactionLookup
Atomikos	com.atomikos.icatch.jta.hibernate3.TransactionManagerLookup
Resin	org.hibernate.transaction.ResinTransactionManagerLookup
Orion	org.hibernate.transaction.OrionTransactionManagerLookup
Oracle Weblogic	org.hibernate.transaction.WeblogicTransactionManagerLookup
Infinispan	org.infinispan.transaction.lookup.JBossStandaloneJTAManagerLookup

Apache Tomee	org.apache.openejb.hibernate.TransactionManagerLookup
JonAs	org.hibernate.transaction.JOnASTransactionManagerLookup
Jrun4	org.hibernate.transaction.JRun4TransactionManagerLookup

A partir de la version 4.0 d'Hibernate, il faut utiliser le service de type JtaPlatform.

Il faut utiliser la propriété hibernate.transaction.jta.platform et lui passer en paramètre le nom pleinement qualifié d'une classe qui implémente l'interface org.hibernate.service.jta.platform.spi.JtaPlatform.

Conteneur	Propriété hibernate.transaction.jta.platform
Bitronix	org.hibernate.service.jta.platform.internal.BitronixJtaPlatform
Borland Enterprise Server	org.hibernate.service.jta.platform.internal.BorlandEnterpriseServerJtaPlatform
JBoss AS	org.hibernate.service.jta.platform.internal.JBossAppServerJtaPlatform
JBoss TM	org.hibernate.service.jta.platform.internal.JBossStandAloneJtaPlatform
JonAs	org.hibernate.service.jta.platform.internal.JOnASJtaPlatform
JOTM	org.hibernate.service.jta.platform.internal.JOTMJtaPlatform
JRun 4 AS	org.hibernate.service.jta.platform.internal.JRun4JtaPlatform
Aucun	org.hibernate.service.jta.platform.internal.NoJtaPlatform
Oracle OC4J	org.hibernate.service.jta.platform.internal.OC4JJtaPlatform
Orion AS	org.hibernate.service.jta.platform.internal.OrionJtaPlatform
Resin AS	org.hibernate.service.jta.platform.internal.ResinJtaPlatform
Oracle Glassfish	org.hibernate.service.jta.platform.internal.SunOneJtaPlatform
Pont vers TransactionManagerLookup	org.hibernate.service.jta.platform.internal.TransactionManagerLookupBridge
Oracle Weblogic	org.hibernate.service.jta.platform.internal.WeblogicJtaPlatform
IBM Websphere 6 et supérieur	org.hibernate.service.jta.platform.internal.WebSphereExtendedJtaPlatform
IBM Websphere 4 à 5.1	org.hibernate.service.jta.platform.internal.WebSphereJtaPlatform

Exemple :

```
<property name="hibernate.transaction.jta.platform"
value="org.hibernate.service.jta.platform.internal.SunOneJtaPlatform" />
```

L'utilisation de l'API TransactionManagerLookup affiche une entrée de type warning dans le journal.

Résultat :

```
HHH000427:
Using deprecated org.hibernate.transaction.TransactionManagerLookup strategy
[hibernate.transaction.manager_lookup_class], use newer
org.hibernate.service.jta.platform.spi.JtaPlatform strategy instead
[hibernate.transaction.jta.platform]
```

62.12.4.5. Le support des stratégies par les différents caches

Les caches supportés en standard par Hibernate proposent le support d'une ou plusieurs stratégies.

Cache	read-only (lecture seule)	nonstrict-read-write (lecture-écriture non	read-write (lecture-écriture)	transactional (transactionnel)

		stricte)		
Hashtable (ne pas utiliser en production)	oui	oui	oui	
EHCache	oui	oui	oui	oui (depuis la version 2.1 d'EhCache)
OSCache	oui	oui	oui	
SwarmCache	oui	oui		
JBoss TreeCache 1.x	oui	oui		
JBoss TreeCache 2	oui			oui

62.12.5. Le cache des entités

Le cache des entités est utilisé lors de la lecture d'entités de la base de données à partir de leur identifiant (par exemple en utilisant les méthodes Session.get() ou Session.load()) et lors du parcours des relations de type OneToOne ou ManyToOne.

Par défaut, Hibernate utilise pour une entité une région dont le nom est le nom pleinement qualifié de la classe de l'entité. Si aucune région possédant ce nom n'est définie alors c'est la région par défaut qui est utilisée.

Hibernate ne stocke pas directement les instances des entités lues de la base de données dans le cache mais une copie sérialisée : ceci permet d'éviter des problèmes d'accès concurrents si plusieurs transactions font référence au même objet.

Les identifiants des entités sont utilisés comme index dans le cache.

Exemple :

```
package fr.jmdoudoux.dej.hibernatecache;

import java.util.Set;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import fr.jmdoudoux.dej.hibernatecache.entity.Devise;
import fr.jmdoudoux.dej.hibernatecache.entity.Pays;

public class TestHibernateCache {
  private static SessionFactory sessionFactory = null;
  public static void main(String[] args) {
    try {
      sessionFactory = new Configuration().configure().buildSessionFactory();
      for (int i = 0; i < 3; i++) {
        lirePays();
      }
    } catch (Throwable ex){
      System.err.println("Erreur durant les traitements" + ex);
    } finally {
      sessionFactory.close();
    }
  }

  public static void lirePays() throws Exception {
    Session session = sessionFactory.openSession();
    Transaction tx = null;
    try {
      tx = session.beginTransaction();
      Pays pays = (Pays) session.load(Pays.class, new Integer(4));
      System.out.println("pays : id=" + pays.getId() + "codeIso="
          + pays.getCodeIso() + " nom=" + pays.getNom());
      tx.commit();
    } catch (Exception e) {
      if (tx != null) {
        tx.rollback();
      }
      throw e;
    } finally {
      session.close();
    }
  }
}
```

Résultat :

```
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
pays : id=4 codeIso=LU nom=Luxembourg
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
pays : id=4 codeIso=LU nom=Luxembourg
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
pays : id=4 codeIso=LU nom=Luxembourg
```

Par défaut, aucune entité n'est mise dans le cache de second niveau qui a été activé et configuré.

Chaque entité concernée doit être configurée pour être stockée dans le cache de second niveau. Cette configuration doit permettre de préciser la stratégie de gestion des accès concurrents aux données des entités.

Il existe plusieurs manières de déclarer qu'une entité doit être mise dans le cache de second niveau :

- utiliser le tag <cache> dans le fichier de mapping hbm.xml
- utiliser l'annotation @Cache sur la classe de l'entité
- utiliser le tag <class-cache> dans le fichier de configuration d'Hibernate

Dans le fichier de mapping, la définition de la mise en cache pour une entité se fait en utilisant le tag <cache>.

Exemple :

```
<cache usage="transactional|read-write|nonstrict-read-write|read-only"
region="nom_de_la_region" include="all|non-lazy" />
```

L'attribut obligatoire usage permet de préciser la stratégie de concurrence d'accès transactionnel qui sera appliquée avec l'entité dans le cache.

L'attribut optionnel region permet de préciser le nom de la région à utiliser dans le cache. Par défaut, c'est le nom pleinement qualifié de la classe de l'entité.

L'attribut optionnel include peut prendre deux valeurs : all et non-lazy. La valeur non-lazy permet de ne pas mettre en cache des entités chargées de manière lazy. La valeur par défaut est all.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate
Mapping DTD//EN"
 "http://hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="fr.jmdoudoux.dej.hibernatecache.entity.Pays" table="pays" >
        <cache usage="read-only"/>
        <id name="id" column="id" type="int" >
            <generator class="identity" />
        </id>
        <property name="codeIso" column="code_iso" not-null="true" type="string" />
        <property name="nom" not-null="true" type="string" />
        <many-to-one name="devise" column="FK_DEVISE"
           class="fr.jmdoudoux.dej.hibernatecache.entity.Devise" />
    </class>
</hibernate-mapping>
```

L'utilisation du cache de second niveau doit être activée et configurée comme indiqué dans la section concernée.

Résultat :

```
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
pays : id=4 codeIso=LU nom=Luxembourg
pays : id=4 codeIso=LU nom=Luxembourg
pays : id=4 codeIso=LU nom=Luxembourg
```

La requête n'est plus effectuée qu'une seule fois sur la base de données : pour les autres accès, les données de l'entité sont extraites du cache.

Dans la classe de l'entité, il est possible d'utiliser le tag @org.hibernate.annotations.Cache.

Exemple :

```
package fr.jmdoudoux.dej.hibernatecache.entity;

import java.io.Serializable;
import javax.persistence.Basic;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;
import org.hibernate.annotations.Cache;
import org.hibernate.annotations.CacheConcurrencyStrategy;

@Entity
@Table(name = "PAYS")
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class Pays implements Serializable {
  private static final long serialVersionUID = 1L;

  @Id
  private int               id;

  @Column(name = "code_iso")
  private String            codeIso;

  @Basic
  private String            nom;

  @ManyToOne(cascade = { CascadeType.PERSIST, CascadeType.MERGE })
  @JoinColumn(name = "FK_DEVISE")
  private Devise            devise;

  public Pays() {
  }

  // getters et setters

  @Override
  public String toString() {
    return "Pays [id=" + id + ", codeIso=" + codeIso + ", nom=" + nom
        + ", devise=" + devise + ", getClass()=" + getClass() + ", hashCode()="
        + hashCode() + ", toString()=" + super.toString() + "]";
  }
}
```

L'attribut obligatoire usage permet de préciser la stratégie de gestion des accès concurrents grâce à l'énumération org.hibernate.annotations.CacheConcurrencyStrategy qui possède les valeurs NONE, READ_ONLY, READ_WRITE, NONSTRICT_READ_ONLY et TRANSACTIONAL.

L'attribut optionnel region permet de préciser le nom de la région du cache à utiliser. Par défaut, c'est le nom pleinement qualifié de la classe de l'entité.

L'attribut optionnel include peut prendre deux valeurs : all pour inclure toutes les propriétés ou non-lazy pour inclure uniquement les propriétés chargées de manière différée.

Enfin, avec le fichier de configuration hibernate.cfg.xml, il est possible d'utiliser le tag <class-cache> pour les entités.

Le tag <class-cache> possède deux attributs :

- class : permet de préciser le nom pleinement qualifié de la classe de l'entité
- usage : permet de préciser la stratégie de gestion des accès concurrents

Exemple :

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/mabdd</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password"></property>
    <property name="hibernate.connection.pool_size">1</property>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.current_session_context_class">thread</property>
    <!-- Cache de second niveau désactivé  -->
    <!-- property name="hibernate.cache.provider_class">
      org.hibernate.cache.internal.NoCacheProvider</property-->
    <property name="hibernate.cache.region.factory_class">
      org.hibernate.cache.ehcache.SingletonEhCacheRegionFactory</property>
    <property name="hibernate.generate_statistics">true</property>
    <property name="hibernate.cache.use_structured_entries">true</property>
    <!-- Afficher les requêtes SQL exécutées sur la sortie standard -->
    <property name="hibernate.show_sql">true</property>
    <mapping resource="Pays.hbm.xml" />
    <mapping resource="Devise.hbm.xml" />
    <class-cache class="fr.jmdoudoux.dej.hibernatecache.entity.Pays" usage="read-only"/>
    </session-factory>
</hibernate-configuration>
```

Les méthodes load() et get() vérifient toujours le cache avant d'exécuter une requête sur la base de données.

Le cache de second niveau ne stocke pas une instance d'une entité mais une forme dite déshydratée (dehydrated) d'une entité : ce sont les valeurs de chaque propriété qui sont stockées dans le cache. Si cette propriété est du type d'une entité (dans une association de type xxx-to-one), alors c'est l'identifiant de cette entité qui est stocké dans le cache.

Seules les données de l'entité elle-même sont mises en cache : les entités de ses associations ne sont pas mises dans le cache par défaut sauf si les entités correspondantes sont explicitement configurées pour y être placées.

A chaque fois qu'une instance de l'entité doit être obtenue du cache, une nouvelle instance est créée à partir des valeurs des propriétés stockées dans le cache.

Il est donc très important de tenir compte du fait que si de nombreuses entités sont obtenues du cache de second niveau alors il y a un surcoût lié à chaque fois à la création d'une nouvelle instance.

Le cache de second niveau ne contient pas de graphe d'objets : les relations ne contiennent que les identifiants des entités. Ceci permet à Hibernate de ne pas répliquer de données.

Une exception de type java.lang.UnsupportedOperationException avec le message «Can't write to a readonly object» est levée si une modification est faite sur une entité dont la stratégie de cache est read-only.

Si une entité stockée en cache doit être mise à jour, il faut configurer sa mise en cache en utilisant une stratégie le permettant. Par exemple, la stratégie read-write permet de mettre à jour les données de l'entité mise en cache.

Exemple :

```
  public static void lireEtModifierPays() throws Exception {
    Session session1 = sessionFactory.openSession();
    Session session2 = null;
    Transaction tx = null;
    try {
      tx = session1.beginTransaction();
      Pays pays = (Pays) session1.load(Pays.class, new Integer(99));
      System.out.println("session1 pays : id=" + pays.getId() + " codeIso="
          + pays.getCodeIso() + " nom=" + pays.getNom());
      pays.setNom(pays.getNom() + " modifie");
      tx.commit();
      session2 = sessionFactory.openSession();
      tx = session2.beginTransaction();
      pays = (Pays) session2.load(Pays.class, new Integer(99));
      System.out.println("session2 pays : id=" + pays.getId() +" codeIso="
          + pays.getCodeIso() + " nom=" + pays.getNom());
      tx.commit();
    } catch (Exception e) {
      if (tx != null) {
        tx.rollback();
      }
      throw e;
    } finally {
      if (session2 != null)
        session2.close();
      if (session1 != null)
        session1.close();
    }
  }
```

Résultat :

```
Hibernate: select pays0_.id as
id0_0_, pays0_.code_iso as code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE
as FK4_0_0_ from pays pays0_ where pays0_.id=?
session1 pays : id=99 codeIso=XXX nom=test
Hibernate: update pays set
code_iso=?, nom=?, FK_DEVISE=? where id=?
session2 pays : id=99 codeIso=XXX nom=test modifie
2012-08-28 22:40:01.982 INFO
[main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000239:
Second level cache puts: 2
2012-08-28 22:40:01.998 INFO
[main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000237:
Second level cache hits: 1
2012-08-28 22:40:01.998
INFO
[main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000238:
Second level cache misses: 1
```

Dans l'exemple ci-dessus, l'entité est mise deux fois dans le cache de second niveau : une première fois après la lecture et une seconde fois après la mise à jour de l'entité. Lorsque l'entité est relue dans la seconde session, l'entité est obtenue du cache de second niveau des entités.

Cet exemple fonctionne comme attendu car les deux sessions ne sont pas imbriquées.

Exemple :

```
  public static void lireEtModifierPays() throws Exception {
    Session session1 = sessionFactory.openSession();
    Session session2 = sessionFactory.openSession();
    Transaction tx = null;
    try {
      tx = session1.beginTransaction();
      Pays pays = (Pays) session1.load(Pays.class, new Integer(99));
      System.out.println("session1 pays : id=" + pays.getId() + " codeIso="
          + pays.getCodeIso() + " nom=" + pays.getNom());
      pays.setNom(pays.getNom() + " modifie");
      tx.commit();
      tx = session2.beginTransaction();
      pays = (Pays) session2.load(Pays.class, new Integer(99));
      System.out.println("session2 pays : id=" + pays.getId() + " codeIso="
          + pays.getCodeIso() + " nom=" + pays.getNom());
      tx.commit();
    } catch (Exception e) {
      if (tx != null) {
        tx.rollback();
      }
      throw e;
    } finally {
      if (session2 != null)
        session2.close();
      if (session1 != null)
        session1.close();
    }
  }
```

Résultat :

```
Hibernate: select pays0_.id as
id0_0_, pays0_.code_iso as code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE
as FK4_0_0_ from pays pays0_ where pays0_.id=?
2012-08-19 21:55:51.721 INFO
session1 pays : id=99 codeIso=XXX nom=test modifie
Hibernate: update pays set
code_iso=?, nom=?, FK_DEVISE=? where id=?
Hibernate: select pays0_.id as
id0_0_, pays0_.code_iso as code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE
as FK4_0_0_ from pays pays0_ where pays0_.id=?
session2 pays : id=99 codeIso=XXX nom=test modifie modifie
```

Dans l'exemple ci-dessus, l'entité est mise deux fois dans le cache de second niveau : une première fois après la lecture et une seconde fois après la mise à jour de l'entité. Lorsque l'entité est relue dans la seconde session, l'entité n'est pas obtenue du cache de second niveau des entités mais elle est relue de la base de données.

Hibernate vérifie que la date de création de la session est supérieure à la date de mise en cache de l'entité : si ce n'est pas le cas, la donnée est relue de la base de données.

Dans l'exemple ci-dessus, l'entité est mise deux fois dans le cache de second niveau mais elle n'est jamais obtenue du cache, ce qui est contraire à ce qui pourrait être voulu.

La stratégie nonstrict-read-write permet aussi de mettre à jour un entité mise en cache mais les données du cache ne sont pas modifiées : les données de l'entité son simplement invalidées dans le cache, ce qui reforce une lecture pour le prochain accès à l'entité.

Résultat :

```
Hibernate: select pays0_.id as
id0_0_, pays0_.code_iso as code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE
as FK4_0_0_ from pays pays0_ where pays0_.id=?
session1 pays : id=99 codeIso=XXXtest nom=test modifie
Hibernate: update pays set
code_iso=?, nom=?, FK_DEVISE=? where id=?
Hibernate: select pays0_.id as
id0_0_, pays0_.code_iso as code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE
as FK4_0_0_ from pays pays0_ where pays0_.id=?
session2 pays : id=99 codeIso=XXXtest nom=test modifie
2012-08-28 22:43:53.960 INFO     [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl
- HHH000239: Second level cache puts: 2
2012-08-28 22:43:53.960 INFO
```

```
[main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000237:
Second level cache hits: 0
2012-08-28 22:43:53.960
INFO     [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl
- HHH000238: Second level cache misses: 2
```

62.12.6. Le cache des associations many

Le cache des associations est utilisé lors du parcours des relations de type xxxToMany.

Par défaut, les associations ne sont pas mises dans le cache de second niveau par Hibernate. Hibernate laisse la possibilité de choisir les associations qui doivent être mises en cache ou être systématiquement relues de la base de données. Celles qui doivent donc être mises en cache doivent être configurées comme telles.

L'exemple de cette section va afficher la liste des pays de la base de données qui utilisent l'euro, trois fois de suite. Cette opération va exploiter la relation 1-N entre devise et pays.

Exemple :

```
package fr.jmdoudoux.dej.hibernatecache;

import java.util.Set;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import fr.jmdoudoux.dej.hibernatecache.entity.Devise;
import fr.jmdoudoux.dej.hibernatecache.entity.Pays;

public class TestHibernateCache {
  private static SessionFactory sessionFactory = null;
  public static void main(String[] args) {
    try {
      sessionFactory = new Configuration().configure().buildSessionFactory();
      for (int i = 0; i < 3; i++) {
        // lirePays();
        listerPaysEuro();
      }
    } catch (Throwable ex) {
      System.err.println("Erreur durant les traitements" + ex);
    } finally {
      sessionFactory.close();
    }
  }

  public static void listerPaysEuro() throws Exception {
    Session session = sessionFactory.openSession();
    Transaction tx = null;
    try {
      tx = session.beginTransaction();
      Devise deviseEur = (Devise) session.load(Devise.class, new Integer(1));
      System.out.println("Devise : id=" + deviseEur.getId() + " code="
          + deviseEur.getCode() + " libelle=" + deviseEur.getLibelle());
      Set<Pays> paysEur = deviseEur.getPays();
      for (Pays pays : paysEur) {
        System.out.println("  pays : id=" + pays.getId() + " codeIso="
            + pays.getCodeIso() + " nom=" + pays.getNom());
      }
      tx.commit();
    } catch (Exception e) {
      if (tx != null) {
        tx.rollback();
      }
      throw e;
    } finally {
      session.close();
    }
  }
}
```

Résultat :

```
Hibernate: select devise0_.ID as ID1_0_, devise0_.CODE as
CODE1_0_, devise0_.LIBELLE as LIBELLE1_0_ from DEVISE devise0_ where
devise0_.ID=?
Devise : id=1 code=EUR libelle=Euro
Hibernate: select pays0_.FK_DEVISE as FK4_1_1_, pays0_.id as id1_,
pays0_.id as id0_0_, pays0_.code_iso as code2_0_0_, pays0_.nom as nom0_0_,
pays0_.FK_DEVISE as FK4_0_0_ from pays pays0_ where pays0_.FK_DEVISE=?
  pays : id=4 codeIso=LU nom=Luxembourg
  pays : id=3 codeIso=I nom=Italie
  pays : id=2 codeIso=D nom=Allemagne
  pays : id=1 codeIso=FR nom=France
Hibernate: select devise0_.ID as ID1_0_, devise0_.CODE as
CODE1_0_, devise0_.LIBELLE as LIBELLE1_0_ from DEVISE devise0_ where
devise0_.ID=?
Devise : id=1 code=EUR libelle=Euro
Hibernate: select pays0_.FK_DEVISE as FK4_1_1_, pays0_.id as id1_,
pays0_.id as id0_0_, pays0_.code_iso as code2_0_0_, pays0_.nom as nom0_0_,
pays0_.FK_DEVISE as FK4_0_0_ from pays pays0_ where pays0_.FK_DEVISE=?
  pays : id=2 codeIso=D nom=Allemagne
  pays : id=3 codeIso=I nom=Italie
  pays : id=4 codeIso=LU nom=Luxembourg
  pays : id=1 codeIso=FR nom=France
Hibernate: select devise0_.ID as ID1_0_, devise0_.CODE as
CODE1_0_, devise0_.LIBELLE as LIBELLE1_0_ from DEVISE devise0_ where
devise0_.ID=?
Devise : id=1 code=EUR libelle=Euro
Hibernate: select pays0_.FK_DEVISE as FK4_1_1_, pays0_.id as id1_,
pays0_.id as id0_0_, pays0_.code_iso as code2_0_0_, pays0_.nom as nom0_0_,
pays0_.FK_DEVISE as FK4_0_0_ from pays pays0_ where pays0_.FK_DEVISE=?
  pays : id=2 codeIso=D nom=Allemagne
  pays : id=1 codeIso=FR nom=France
  pays : id=4 codeIso=LU nom=Luxembourg
  pays : id=3 codeIso=I nom=Italie
```

A chaque itération, la devise est lue dans la base de données et une requête est effectuée sur la base de données pour obtenir les pays possédant cette devise.

Pour limiter le nombre de requêtes effectuées sur la base de données, il est possible d'utiliser le cache des associations.

Lors du parcours des éléments de l'association, Hibernate stocke dans le cache les identifiants des entités mais pas les entités elles-mêmes. Si une entité associée est configurée pour être mise en cache, alors les données de l'entité sont retrouvées dans le cache, sinon elle sera relue de la base de données.

Le cache des associations ne stockent que des identifiants : pour l'entité et pour ses entités associées. De ce fait, si le cache d'associations est activé alors il est important d'activité aussi le cache sur les entités correspondantes pour éviter de gérer des requêtes inutiles sur la base de données.

Par défaut, aucune association n'est mise dans le cache de second niveau qui a été activé et configuré. Chaque association concernée doit être configurée pour être stockée dans le cache de second niveau. Cette configuration doit permettre de préciser la stratégie de gestion des accès concurrents aux données des associations.

Il existe plusieurs manières de déclarer qu'une association doit être mise dans le cache de second niveau :

- utiliser le tag <cache> dans le fichier de mapping hbm.xml
- utiliser l'annotation @Cache sur la collection dans la classe de l'entité
- utiliser le tag <collection-cache> dans le fichier de configuration d'Hibernate

Une seule de ces solutions doit être utilisée.

Dans le fichier de mapping, la définition de la mise en cache de l'association se fait en utilisant le tag <cache>

Exemple :

```
<class name="Devise" table="DEVISE">
   <cache usage="read-only"/>
   <id name="id" column="ID">
     <generator class="identity"/>
   </id>
   <property name="code" column="CODE"/>
   <property name="libelle" column="LIBELLE"/>
   <set name="pays">
     <cache usage="read-only"/>
     <key column="FK_DEVISE"/>
     <one-to-many class="Pays"/>
   </set>
 </class>
```

Attention : il est nécessaire que le cache pour les entités des deux relations soit actif puisque le cache des associations ne stocke que les identifiants des entités. Ceci évitera à Hibernate de refaire une lecture pour chacune des entités car il pourra directement obtenir les entités du cache.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate
Mapping DTD//EN" "http://hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="fr.jmdoudoux.dej.hibernatecache.entity.Pays"
      table="pays" >
        <cache usage="read-only"/>
        <id name="id" column="id" type="int" >
            <generator class="identity" />
        </id>
        <property name="codeIso" column="code_iso"
            not-null="true" type="string" />
        <property name="nom" not-null="true" type="string" />
        <many-to-one name="devise" column="FK_DEVISE"
           class="fr.jmdoudoux.dej.hibernatecache.entity.Devise" />
    </class>
</hibernate-mapping>
```

Dans le fichier de configuration hibernate.cfg.xml, il est possible d'utiliser le tag <collection-cache> pour les associations.

Le tag <collection-cache> possède deux attributs :

- collection : permet de préciser le nom de la collection
- usage : permet de préciser la stratégie de gestion des accès concurrents

Exemple :

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/mabdd</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password"></property>
    <property name="hibernate.connection.pool_size">1</property>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.current_session_context_class">thread</property>
    <!-- Cache de second niveau désactivé  -->
    <!-- property name="hibernate.cache.provider_class">
      org.hibernate.cache.internal.NoCacheProvider</property-->
    <property name="hibernate.cache.region.factory_class">
      org.hibernate.cache.ehcache.SingletonEhCacheRegionFactory</property>
    <property name="hibernate.generate_statistics">true</property>
    <property name="hibernate.cache.use_structured_entries">true</property>
    <!-- Afficher les requêtes SQL exécutées sur la sortie standard -->
    <property name="hibernate.show_sql">true</property>
    <mapping resource="Pays.hbm.xml" />
    <mapping resource="Devise.hbm.xml" />
    <class-cache class="fr.jmdoudoux.dej.hibernatecache.entity.Pays" usage="read-only"/>
    <class-cache class="fr.jmdoudoux.dej.hibernatecache.entity.Devise" usage="read-only" />
    <collection-cache collection="fr.jmdoudoux.dej.hibernatecache.entity.Devise.pays"
      usage="read-only"/>
    </session-factory>
</hibernate-configuration>
```

Pour mettre en cache les données d'une relation OneToMany ou ManyToMany, il est possible d'utiliser l'annotation @Cache.

Exemple :

```
package fr.jmdoudoux.dej.hibernatecache.entity;

import java.io.Serializable;
import java.util.Set;
import javax.persistence.Basic;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;
import org.hibernate.annotations.Cache;
import org.hibernate.annotations.CacheConcurrencyStrategy;

@Entity
@Table(name = "DEVISE")
@Cache(usage = CacheConcurrencyStrategy.READ_ONLY)
public class Devise implements Serializable {
  private static final long serialVersionUID = 1L;

  @Id
  private int               id;

  @Basic
  private String            code;

  @Basic
  private String            libelle;

  @OneToMany(mappedBy = "devise")
  @Cache(usage = CacheConcurrencyStrategy.READ_ONLY)
  private Set<Pays>         pays;

  public Devise() {
  }

  // getters et setters
}
```

L'utilisation du cache de second niveau doit être activée et configurée comme indiqué dans la section concernée.

Résultat :

```
Hibernate: select devise0_.ID as ID1_0_, devise0_.CODE as CODE1_0_,
devise0_.LIBELLE as LIBELLE1_0_ from DEVISE devise0_ where devise0_.ID=?
Devise : id=1 code=EUR libelle=Euro
Hibernate: select pays0_.FK_DEVISE as FK4_1_1_, pays0_.id as id1_,
pays0_.id as id0_0_, pays0_.code_iso as code2_0_0_, pays0_.nom as nom0_0_,
pays0_.FK_DEVISE as FK4_0_0_ from pays pays0_ where pays0_.FK_DEVISE=?
  pays : id=3 codeIso=I nom=Italie
  pays : id=1 codeIso=FR nom=France
  pays : id=2 codeIso=D nom=Allemagne
  pays : id=4 codeIso=LU nom=Luxembourg
Devise : id=1 code=EUR libelle=Euro
  pays : id=3 codeIso=I nom=Italie
  pays : id=2 codeIso=D nom=Allemagne
  pays : id=4 codeIso=LU nom=Luxembourg
  pays : id=1 codeIso=FR nom=France
Devise : id=1 code=EUR libelle=Euro
  pays : id=4 codeIso=LU nom=Luxembourg
  pays : id=1 codeIso=FR nom=France
  pays : id=2 codeIso=D nom=Allemagne
  pays : id=3 codeIso=I nom=Italie
```

Les deux requêtes ne sont plus effectuées qu'une seule fois sur la base de données : pour les autres accès, les données de l'entité sont extraites du cache des associations et du cache des entités.

Si le cache n'est pas activé sur l'entité possédant la relation many-to-one, alors l'entité est relue systématiquement dans la base de données malgré l'activation du cache sur la relation.

Résultat :

```
Hibernate: select devise0_.ID as ID1_0_, devise0_.CODE as
CODE1_0_, devise0_.LIBELLE as LIBELLE1_0_ from DEVISE devise0_ where
devise0_.ID=?
Devise : id=1 code=EUR libelle=Euro
Hibernate: select pays0_.FK_DEVISE as FK4_1_1_, pays0_.id as id1_,
pays0_.id as id0_0_, pays0_.code_iso as code2_0_0_, pays0_.nom as nom0_0_,
pays0_.FK_DEVISE as FK4_0_0_ from pays pays0_ where pays0_.FK_DEVISE=?
  pays : id=4 codeIso=LU nom=Luxembourg
  pays : id=2 codeIso=D nom=Allemagne
  pays : id=3 codeIso=I nom=Italie
  pays : id=1 codeIso=FR nom=France
Devise : id=1 code=EUR libelle=Euro
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
  pays : id=3 codeIso=I nom=Italie
  pays : id=4 codeIso=LU nom=Luxembourg
  pays : id=2 codeIso=D nom=Allemagne
  pays : id=1 codeIso=FR nom=France
Devise : id=1 code=EUR libelle=Euro
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
  pays : id=4 codeIso=LU nom=Luxembourg
  pays : id=1 codeIso=FR nom=France
  pays : id=3 codeIso=I nom=Italie
  pays : id=2 codeIso=D nom=Allemagne
```

Dans ce cas, l'utilisation du cache perd toute son efficacité. La requête pour obtenir les identifiants de la relation n'est bien effectuée qu'une seule fois et les données des invocations suivantes sont obtenues du cache. Cependant, comme l'entité pays n'est pas en cache, Hibernate qui ne possèdent que l'identifiant est obligé de refaire une lecture dans la base de données pour chaque entité.

C'est le même comportement si le cache sur l'entité pays est activé mais le cache sur l'entité devise ne l'est pas.

Résultat :

```
Hibernate: select devise0_.ID as ID1_0_, devise0_.CODE as
CODE1_0_, devise0_.LIBELLE as LIBELLE1_0_ from DEVISE devise0_ where
devise0_.ID=?
Devise : id=1 code=EUR libelle=Euro
Hibernate: select pays0_.FK_DEVISE as FK4_1_1_, pays0_.id as id1_,
pays0_.id as id0_0_, pays0_.code_iso as code2_0_0_, pays0_.nom as nom0_0_,
pays0_.FK_DEVISE as FK4_0_0_ from pays pays0_ where pays0_.FK_DEVISE=?
  pays : id=4 codeIso=LU nom=Luxembourg
  pays : id=1 codeIso=FR nom=France
  pays : id=2 codeIso=D nom=Allemagne
  pays : id=3 codeIso=I nom=Italie
Hibernate: select devise0_.ID as ID1_0_, devise0_.CODE as
CODE1_0_, devise0_.LIBELLE as LIBELLE1_0_ from DEVISE devise0_ where
devise0_.ID=?
Devise : id=1 code=EUR libelle=Euro
  pays : id=3 codeIso=I nom=Italie
  pays : id=1 codeIso=FR nom=France
  pays : id=4 codeIso=LU nom=Luxembourg
  pays : id=2 codeIso=D nom=Allemagne
Hibernate: select devise0_.ID as ID1_0_, devise0_.CODE as
CODE1_0_, devise0_.LIBELLE as LIBELLE1_0_ from DEVISE devise0_ where
devise0_.ID=?
Devise : id=1 code=EUR libelle=Euro
  pays : id=3 codeIso=I nom=Italie
  pays : id=2 codeIso=D nom=Allemagne
  pays : id=4 codeIso=LU nom=Luxembourg
  pays : id=1 codeIso=FR nom=France
```

A chaque itération la devise est relue. La requête de la relation n'est effectuée qu'une seule fois puisque les identifiants de son résultat sont mis en cache ainsi que les entités pays correspondantes.

La pire en termes de nombre de requêtes effectuées sur la base survient si le cache est activé sur la relation mais ne l'est pas pour les deux entités.

Résultat :

```
Hibernate: select devise0_.ID as ID1_0_, devise0_.CODE as
CODE1_0_, devise0_.LIBELLE as LIBELLE1_0_ from DEVISE devise0_ where
devise0_.ID=?
Devise : id=1 code=EUR libelle=Euro
Hibernate: select pays0_.FK_DEVISE as FK4_1_1_, pays0_.id as id1_,
pays0_.id as id0_0_, pays0_.code_iso as code2_0_0_, pays0_.nom as nom0_0_,
pays0_.FK_DEVISE as FK4_0_0_ from pays pays0_ where pays0_.FK_DEVISE=?
  pays : id=2 codeIso=D nom=Allemagne
  pays : id=3 codeIso=I nom=Italie
  pays : id=1 codeIso=FR nom=France
  pays : id=4 codeIso=LU nom=Luxembourg
Hibernate: select devise0_.ID as ID1_0_, devise0_.CODE as
CODE1_0_, devise0_.LIBELLE as LIBELLE1_0_ from DEVISE devise0_ where
devise0_.ID=?
Devise : id=1 code=EUR libelle=Euro
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays pays0_
where pays0_.id=?
  pays : id=4 codeIso=LU nom=Luxembourg
  pays : id=1 codeIso=FR nom=France
  pays : id=3 codeIso=I nom=Italie
  pays : id=2 codeIso=D nom=Allemagne
Hibernate: select devise0_.ID as ID1_0_, devise0_.CODE as
CODE1_0_, devise0_.LIBELLE as LIBELLE1_0_ from DEVISE devise0_ where
```

```
devise0_.ID=?
Devise : id=1 code=EUR libelle=Euro
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
  pays : id=3 codeIso=I nom=Italie
  pays : id=1 codeIso=FR nom=France
  pays : id=4 codeIso=LU nom=Luxembourg
  pays : id=2 codeIso=D nom=Allemagne
```

Dans ce cas, la requête de la relation n'est effectuée qu'une seule fois et les identifiants sont mis dans le cache. Pour les itérations suivantes, Hibernate est obligé de relire les entités devises et pays puisque les identifiants correspondants ne sont pas trouvés dans le cache des entités.

62.12.7. Le cache des requêtes

La mise en cache des résultats d'une requête n'apporte pas toujours un gain significatif. Ce sont essentiellement les requêtes qui sont fréquemment exécutées avec les mêmes paramètres qu'il est intéressant de mettre en cache.

Par défaut, l'activation du cache de second niveau ne met aucune requête dans le cache. Comme la plupart des requêtes ne tireront pas de bénéfice à être mises en cache, aucune ne l'est par défaut. Il faut explicitement demander la mise en cache d'une requête.

Hibernate utilise une combinaison de la requête SQL exécutée et des valeurs des paramètres de cette requête pour composer la valeur de la clé dans le cache. La valeur associée à cette clé est la liste des identifiants des entités retournées par la requête. Ainsi, si la requête est de nouveau invoquée avec les mêmes paramètres, alors les identifiants des entités en résultat seront directement retrouvés dans le cache.

Pour renvoyer le résultat du cache des requêtes, Hibernate va rechercher les entités dans le cache des entités à partir des identifiants stockés dans le cache des requêtes. Si l'entité est trouvée, alors une nouvelle instance est créée à partir des informations du cache (cache de premier niveau ou cache des entités) sinon l'entité est relue dans la base de données.

L'exemple de cette section va afficher la liste de tous les pays de la base de données, trois fois de suite. Cette opération va utiliser l'API Criteria qui génère une requête HQL, elle-même transformée en requête SQL exécutée sur la base de données.

Exemple :

```
package fr.jmdoudoux.dej.hibernatecache;

import java.util.List;
import java.util.Set;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import fr.jmdoudoux.dej.hibernatecache.entity.Devise;
import fr.jmdoudoux.dej.hibernatecache.entity.Pays;

public class TestHibernateCache {
  private static SessionFactory sessionFactory = null;

  public static void main(String[] args) {
    try {
      sessionFactory = new Configuration().configure().buildSessionFactory();
      for (int i = 0; i < 3; i++) {
        listerTousPays();
      }
    } catch (Throwable ex) {
      System.err.println("Erreur durant les traitements" + ex);
    } finally {
      sessionFactory.close();
    }
  }

  public static void listerTousPays() throws Exception {
    Session session = sessionFactory.openSession();
    Transaction tx = null;
    try {
      tx = session.beginTransaction();
      List<Pays> listPays = session.createCriteria(Pays.class).list();
      for (Pays pays : listPays) {
        System.out.println("pays : id=" + pays.getId() + " codeIso="
            + pays.getCodeIso() + " nom=" + pays.getNom());
      }
      tx.commit();
    } catch (Exception e) {
      if (tx != null) {
        tx.rollback();
      }
      throw e;
    } finally {
      session.close();
    }
  }
}
```

Résultat :

```
Hibernate: select this_.id as id0_0_, this_.code_iso as
code2_0_0_, this_.nom as nom0_0_, this_.FK_DEVISE as FK4_0_0_ from pays this_
pays : id=1 codeIso=FR nom=France
pays : id=2 codeIso=D nom=Allemagne
pays : id=3 codeIso=I nom=Italie
pays : id=4 codeIso=LU nom=Luxembourg
pays : id=5 codeIso=GB nom=Grande Bretagne
pays : id=6 codeIso=US nom=Etats Unis
pays : id=7 codeIso=JA nom=Japon
Hibernate: select this_.id as id0_0_, this_.code_iso as
code2_0_0_, this_.nom as nom0_0_, this_.FK_DEVISE as FK4_0_0_ from pays this_
pays : id=1 codeIso=FR nom=France
pays : id=2 codeIso=D nom=Allemagne
pays : id=3 codeIso=I nom=Italie
pays : id=4 codeIso=LU nom=Luxembourg
pays : id=5 codeIso=GB nom=Grande Bretagne
pays : id=6 codeIso=US nom=Etats Unis
pays : id=7 codeIso=JA nom=Japon
Hibernate: select this_.id as id0_0_, this_.code_iso as code2_0_0_,
this_.nom as nom0_0_, this_.FK_DEVISE as FK4_0_0_ from pays this_
pays : id=1 codeIso=FR nom=France
pays : id=2 codeIso=D nom=Allemagne
pays : id=3 codeIso=I nom=Italie
pays : id=4 codeIso=LU nom=Luxembourg
pays : id=5 codeIso=GB nom=Grande Bretagne
pays : id=6 codeIso=US nom=Etats Unis
pays : id=7 codeIso=JA nom=Japon\n
```

A chaque itération, la requête est effectuée sur la base de données pour obtenir les pays. Pour limiter le nombre de requêtes effectuées, il est possible d'utiliser le cache des requêtes.

Pour utiliser le cache des requêtes, il faut :

- activer et configurer le cache de second niveau (l'implémentation du cache utilisé doit proposer un support du cache des requêtes d'Hibernate)

- activer l'utilisation du cache des requêtes en passant la valeur true à la propriété hibernate.cache.use_query_cache dans le fichier de configuration d'Hibernate
- activer le cache sur les entités qui seront renvoyées par les requêtes mises en cache

Exemple :

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!-- ...  -->
    <!-- Cache de second niveau activé avec ehcache  -->
    <property name="hibernate.cache.region.factory_class">
      org.hibernate.cache.ehcache.SingletonEhCacheRegionFactory</property>
    <!-- Activation du cache des requêtes  -->
    <property name="hibernate.cache.use_query_cache">true</property>
    <!-- ...  -->
  </session-factory>
</hibernate-configuration>
```

Pour chaque requête que l'on souhaite mettre en cache, il faut invoquer la méthode setCacheable() en lui passant la valeur true en paramètre. La mise en cache des requêtes peut être utilisée sur des objets de type Criteria et Query.

Exemple :

```
  public static void listerTousPays() throws Exception {
    Session session = sessionFactory.openSession();
    Transaction tx = null;
    try {
      tx = session.beginTransaction();
      List<Pays> listPays = session.createCriteria(Pays.class)
        .setCacheable(true).list();
      for (Pays pays : listPays) {
        System.out.println("pays : id=" + pays.getId() + " codeIso="
            + pays.getCodeIso() + " nom=" + pays.getNom());
      }
      tx.commit();
    } catch (Exception e) {
      if (tx != null) {
        tx.rollback();
      }
      throw e;
    } finally {
      session.close();
    }
  }
```

Résultat :

```
Hibernate: select pays0_.id as id0_, pays0_.code_iso as code2_0_,
pays0_.nom as nom0_, pays0_.FK_DEVISE as FK4_0_ from pays pays0_
pays : id=1 codeIso=FR nom=France
pays : id=2 codeIso=D nom=Allemagne
pays : id=3 codeIso=I nom=Italie
pays : id=4 codeIso=LU nom=Luxembourg
pays : id=5 codeIso=GB nom=Grande Bretagne
pays : id=6 codeIso=US nom=Etats Unis
pays : id=7 codeIso=JA nom=Japon
pays : id=1 codeIso=FR nom=France
pays : id=2 codeIso=D nom=Allemagne
pays : id=3 codeIso=I nom=Italie
pays : id=4 codeIso=LU nom=Luxembourg
pays : id=5 codeIso=GB nom=Grande Bretagne
pays : id=6 codeIso=US nom=Etats Unis
pays : id=7 codeIso=JA nom=Japon
pays : id=1 codeIso=FR nom=France
pays : id=2 codeIso=D nom=Allemagne
pays : id=3 codeIso=I nom=Italie
pays : id=4 codeIso=LU nom=Luxembourg
pays : id=5 codeIso=GB nom=Grande Bretagne
pays : id=6 codeIso=US nom=Etats Unis
pays : id=7 codeIso=JA nom=Japon\n
```

La requête n'est exécutée qu'une seule fois car pour les itérations suivantes les données sont extraites du cache des requêtes et du cache des entités.

Le mode de fonctionnement est identique pour des requêtes HQL.

Exemple :

```
  public static void listerTousPays() throws Exception {
    Session session = sessionFactory.openSession();
    Transaction tx = null;
    try {
      tx = session.beginTransaction();
      Query query = session.createQuery("from Pays").setCacheable(true);
      List<Pays> listPays = query.list();
      for (Pays pays : listPays) {
        System.out.println("pays : id=" + pays.getId() + " codeIso="
            + pays.getCodeIso() + " nom=" + pays.getNom());
      }
      tx.commit();
    } catch (Exception e) {
      if (tx != null) {
        tx.rollback();
      }
      throw e;
    } finally {
      session.close();
    }
  }
```

Le résultat à l'exécution est le même mais le temps est réduit.

Le cache des requêtes ne stocke pas les entités qui sont retournées par la requêtes mais uniquement la valeur de leur identifiant et leur type. Pour tirer pleinement parti du cache des requêtes, il est donc important de le coupler avec l'utilisation du cache de second niveau des entités.

Il est important que le cache soit activé sur les entités en résultat des requêtes qui sont mises dans le cache des requêtes. La clé du cache est la requête et la valeur ne contient que les identifiants des entités. Celles-ci doivent être lues soit dans le cache des entités soit être relues de la base de données.

Dans l'exemple ci-dessous, l'entité Pays n'est pas mise en cache.

Résultat :

```
Hibernate:
select pays0_.id as id0_, pays0_.code_iso as code2_0_, pays0_.nom as nom0_,
pays0_.FK_DEVISE as FK4_0_ from pays pays0_
pays : id=1 codeIso=FR nom=France
pays : id=2 codeIso=D nom=Allemagne
pays : id=3 codeIso=I nom=Italie
pays : id=4 codeIso=LU nom=Luxembourg
pays : id=5 codeIso=GB nom=Grande Bretagne
pays : id=6 codeIso=US nom=Etats Unis
pays : id=7 codeIso=JA nom=Japon
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
```

```
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
pays : id=1 codeIso=FR nom=France
pays : id=2 codeIso=D nom=Allemagne
pays : id=3 codeIso=I nom=Italie
pays : id=4 codeIso=LU nom=Luxembourg
pays : id=5 codeIso=GB nom=Grande Bretagne
pays : id=6 codeIso=US nom=Etats Unis
pays : id=7 codeIso=JA nom=Japon
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
Hibernate: select pays0_.id as id0_0_, pays0_.code_iso as
code2_0_0_, pays0_.nom as nom0_0_, pays0_.FK_DEVISE as FK4_0_0_ from pays
pays0_ where pays0_.id=?
pays : id=1 codeIso=FR nom=France
pays : id=2 codeIso=D nom=Allemagne
pays : id=3 codeIso=I nom=Italie
pays : id=4 codeIso=LU nom=Luxembourg
pays : id=5 codeIso=GB nom=Grande Bretagne
pays : id=6 codeIso=US nom=Etats Unis
pays : id=7 codeIso=JA nom=Japon\n
```

La requête pour obtenir tous les pays est bien exécutée une seule fois et mise en cache. Malheureusement comme l'entité pays n'est pas mise en cache, Hibernate n'a pas d'autre solution que de relire chaque entité dans la base de données, ce qui a un effet catastrophique sur les performances car au lieu d'une seule requête sur la base de données, ce sont de nombreuses requêtes qui sont exécutées.

Pour éviter à Hibernate de faire des relectures pour chacune des entités, il est préférable d'activer la mise en cache des entités concernées. Il est donc important d'utiliser en conjonction le cache des requêtes et le cache des entités : le type des entités retournées par les résultats d'une requête mise en cache doit être configuré pour être pris en compte par le cache des entités.

Le cache des requêtes d'Hibernate utilise deux régions particulières par défaut :

- org.hibernate.cache.StandardQueryCache : stocke les identifiants des entités retournées par une requête pour des paramètres donnés
- org.hibernate.cache.UpdateTimestampsCache : stocke pour chaque table la date/heure de dernière mise à jour d'un enregistrement. Chaque modification dans une table met à jour la date de dernière modification dans l'entrée correspondante de cette région du cache.

Il est très important que la région UpdateTimestampsCache soit configurée pour ne jamais être invalidée périodiquement.

Il est possible de configurer une région dédiée du cache pour stocker le résultat de certaines requêtes. Cette configuration particulière peut notamment permettre de définir la stratégie d'éviction du contenu du cache. Par défaut, les requêtes mises en cache le sont dans la région par défaut pour les requêtes. Pour préciser une autre région, il faut préciser son nom en invoquant la méthode setCacheRegion().

Exemple :

```
      Query query = session.createQuery("from Pays").setCacheable(true)
        .setCacheRegion("query.pays");
      List<Pays> listPays = query.list();
```

Si plusieurs régions peuvent être utilisées pour mettre en cache les requêtes, il ne peut y avoir qu'une seule région de type timestamps.

La région UpdateTimestampsCache permet à Hibernate de stocker la date/heure de dernière mise à jour pour toutes les tables utilisées.

Hibernate utilise les données de la région UpdateTimestampCache pour invalider les données contenues dans la région StandardQueryCache : Hibernate invalide le contenu du cache pour une requête si, pour une des tables figurant dans le résultat, la date/heure de mise en cache de la requête est antérieure à celle de la dernière modification stockée dans la région UpdateTimestampsCache.

En d'autres termes, lorsqu'une requête qui a été mise en cache est réexécutée, Hibernate vérifie la date/heure de dernière modification (insert, update ou delete) dans la ou les tables utilisées comme résultat. Si cette date/heure est plus récente que la date/heure de mise en cache des résultats de la requête alors celle-ci est invalidée dans le cache et la requête est réexécutée sur la base de données.

La date/heure stockée dans la région UpdateTimestampsCache concerne une mise à jour sur n'importe quelle occurrence de la table.

La méthode setCacheMode() des interfaces Query et Criteria permet de modifier le mode d'utilisation du cache pour la requête selon l'instance de type CacheMode fournie en paramètre.

Instance	Rôle
CacheMode.NORMAL	Des éléments peuvent être lus et écrits dans le cache
CacheMode.GET	Les éléments peuvent être uniquement lus du cache
CacheMode.PUT	Les éléments ne sont pas lus du cache mais ils sont écrits dans le cache lorsqu'ils sont lus dans la base de données
CacheMode.IGNORE	Les éléments ne sont ni lus ni écrits dans le cache
CacheMode.REFRESH	Forcer le rafraichissement des éléments du cache

En passant la valeur CacheMode.REFRESH à la méthode setCacheMode(), Hibernate ne va pas rechercher le résultat de la requête dans le cache mais exécuter la requête et insérer ou remplacer le résultat dans le cache. Ceci permet de forcer le rafraîchissement des données du cache : c'est particulièrement utile si un autre processus met à jour la base de données pour des données référencées dans le cache.

Il est nécessaire d'être prudent lors de l'utilisation du cache des requêtes.

Le cache des requêtes doit être utilisé avec soin car son activation peut introduire une certaine latence et limiter les possibilités de montée en charge.

Il est par exemple contre-productif :

- d'activer le cache des requêtes si celui-ci n'est pas utilisé.
- de mettre en cache les résultats d'une requête dont la ou les tables des entités concernées sont modifiées fréquemment impliquant de fait une invalidation des résultats dans le cache

Une modification dans une table implique la mise à jour du cache timestamps pour la table et invalide automatiquement les résultats d'une requête du cache qui contiennent au moins une entité de cette table même si la modification ne concerne pas cette entité. Si une table est souvent modifiée, le taux de hit dans le cache est très bas puisque les données du cache sont fréquemment invalidées.

Le cache des requêtes peut être assez consommateur en ressources surtout si de nombreuses requêtes avec leurs paramètres sont mises en cache. Les requêtes SQL sont généralement composées de plusieurs centaines de caractères fréquemment utilisées avec des paramètres différents.

La mise à jour du cache des timestamps introduit une contention liée à la pose d'un verrou. Ce verrou est mis en oeuvre à chaque accès au cache ce qui peut créer un goulot d'étranglement lorsque la charge augmente ou lorsque la même table est mise à jour par plusieurs threads.

L'utilisation du cache des requêtes induit donc un certain surcoût dans les traitements transactionnels. Ceci est particulièrement vrai si les entités mises en cache sont mises à jour car dans ce cas Hibernate doit invalider certaines données du cache.

62.12.8. La gestion du cache de second niveau

Jusqu'à la version 3.3 d'Hibernate incluse, la classe SessionFactory proposait plusieurs méthodes pour retirer des données du cache de second niveau.

Méthode	Rôle
void evict(Class persistentClass)	Retirer du cache toutes les données d'un type d'entité
void evict(Class persistentClass, Serializable id)	Retirer du cache les données d'une entité dont l'identifiant est fourni en paramètre
void evictCollection(String roleName)	Supprimer toutes les données d'une association dont le nom est fourni en paramètre
void evictCollection(String roleName, Serializable id)	Supprimer les données d'une association dont le nom est fourni en paramètres etconcernant l'entité dont l'identifiant est fourni
void evictEntity(String entityName)	Supprimer toutes les données du type d'entité dont le nom est fourni en paramètre
void evictEntity(String entityName, Serializable id)	Supprimer les données d'une entité dont le type et l'identifiant sont fournis en paramètres
void evictQueries()	Supprimer tout le contenu de la région par défaut du cache des requêtes
void evictQueries(String cacheRegion)	Supprimer tout le contenu d'une région particulière du cache des requêtes dont le nom est passé en paramètre

La méthode getAllClassMetadata() renvoie une map dont la clé est le nom de l'entité et la valeur est un objet de type ClassMetadata qui encapsule les métadonnées de l'entité.

La méthode getAllCollectionMetadata() renvoie une map dont la clé est le nom de l'association et la valeur est un objet de type CollectionMetadata qui encapsule les métadonnées de l'association.

L'exemple ci-dessous purge entièrement le contenu du cache de second niveau (associations, entités et requêtes).

Exemple :

```
Map<String, CollectionMetadata> collectionMetadatas =
  sessionFactory.getAllCollectionMetadata();
for (String nom : collectionMetadatas.keySet()) {
  sessionFactory.evictCollection(nom);
}

Map<String, ClassMetadata> classMetadatas = sessionFactory.getAllClassMetadata();
```

```
for (String nom : classMetadatas.keySet()) {
  sessionFactory.evictEntity(nom);
}

sessionFactory.evictQueries();
```

A partir de la version 3.5 d'Hibernate, toutes les méthodes de l'interface SessionFactory relatives à la gestion du contenu du cache sont deprecated (evit(), evictCollection(), evictEntity(), evictQueries()) : il faut utiliser un objet de type Cache.

Un objet de type Cache est obtenu en invoquant la méthode getCache() de la SessionFactory. L'interface Cache décrit les fonctionnalités permettant de déterminer la présence et de supprimer des données dans les régions du cache de second niveau.

Méthode	Rôle
boolean containsCollection(String role, Serializable ownerIdentifier)	Renvoyer un booléen qui permet de préciser si une association est présente dans le cache
boolean containsEntity(Class entityClass, Serializable identifier)	Renvoyer un booléen qui permet de préciser si une entité est présente dans le cache
boolean containsEntity(String entityName, Serializable identifier)	Renvoyer un booléen qui permet de préciser si une entité est présente dans le cache
boolean containsQuery(String regionName)	Renvoyer un booléen qui permet de préciser si une région utilisée pour le cache des requêtes contient des éléments
void evictCollection(String role, Serializable ownerIdentifier)	Supprimer des éléments du cache de second niveau relatifs à une association
void evictCollectionRegion(String role)	Supprimer les données des entités du cache de second niveau relatives à une association
void evictCollectionRegions()	Supprimer tous les éléments des régions du cache relatives aux associations
void evictDefaultQueryRegion()	Supprimer les éléments de la région par défaut du cache des requêtes
void evictEntity(Class entityClass, Serializable identifier)	Supprimer les données d'une entité du cache de second niveau
void evictEntity(String entityName, Serializable identifier)	Supprimer les données d'une entité du cache de second niveau
void evictEntityRegion(Class entityClass)	Supprimer toutes les données d'une entité
void evictEntityRegion(String entityName)	Supprimer toutes les données d'une entité
void evictEntityRegions()	Supprimer toutes les données des caches des entités
void evictQueryRegion(String regionName)	Supprimer toutes les données de la région du cache des requêtes
void evictQueryRegions()	Supprimer toutes les données de toutes les régions utilisées pour le cache des requêtes

Attention : aucune de ces méthodes ne prend en compte un aspect transactionnel : leur exécution est immédiate sans gestion des accès concurrents réalisés par les transactions en cours.

Exemple :

```
Cache cache = sessionFactory.getCache();

System.out.println(cache.containsEntity(Pays.class, 2));

Cache.evictEntity(Pays.class,2);
```

62.12.9. Le monitoring de l'utilisation du cache

Une fois un cache de second niveau mis en place, il est nécessaire de régulièrement surveiller son activité pour vérifier sa bonne utilisation et éventuellement modifier sa configuration pour obtenir les meilleures performances.

L'utilisation du cache de second niveau implique que l'accès aux données de ce cache peut réussir (hit) ou échouer (miss).

Il est possible de demander à Hibernate d'être verbeux sur l'utilisation du cache dans les logs en utilisant le niveau DEBUG pour le logger org.hibernate.cache.

Exemple :

```
<logger name="org.hibernate.cache">
  <level value="DEBUG" />
</logger>
```

L'inconvénient de cette activation est qu'elle est très verbeuse.

Hibernate propose un mécanisme de calcul de statistiques sur son activité incluant entre autres des informations sur l'utilisation du cache. Ce mécanisme doit être activé en donnant la valeur true à la propriété hibernate.generate_statistics.

L'inconvénient des statistiques est qu'elles consomment de la ressource mais aussi qu'elles induisent une légère dégradation des performances liée à la gestion des accès concurrents.

Hibernate propose des statistiques sur l'utilisation de l'objet de type SessionFactory. Une API permet de les consulter et elles sont facilement exposables sous la forme d'un MBean JMX.

62.12.9.1. L'activation et l'obtention de données statistiques

Les statistiques permettent d'obtenir des informations utiles sur l'utilisation du cache et des sessions :

- Obtenir des informations globales : le nombre d'entités obtenues du cache (hit), le nombre d'entités non obtenues du cache (miss), ...
- Obtenir des informations précises sur l'utilisation dans le cache d'une entité ou d'une association particulière (EntityStatistics et CollectionStatistics)
- Obtenir des informations sur l'exécution des requêtes mises en cache (QueryStatistics)
- Obtenir des informations sur l'utilisation d'une région particulière du cache (SecondLevelCacheStatistics)

Hibernate propose des statistiques d'utilisation fournies par la classe SessionFactory. Ces informations sont disponibles de deux manières :

- La publication par JMX en activant le MBean StatisticsService
- L'utilisation de la méthode getStatistics() de la classe SessionFactory

Par défaut, les statistiques sont désactivées. Il y a deux manières de les activer :

- en utilisant la propriété hibernate.generate_statistics du fichier de configuration d'Hibernate et en lui passant la valeur true

Exemple :

```
<prop key="hibernate.generate_statistics">true</prop>\n
```

- en invoquant la méthode SessionFactory.getStatistics().setStatisticsEnabled() et en lui passant la valeur true en paramètre

Exemple :

```
Statistics statistics = sessionFactory.getStatistics();
statistics.setStatisticsEnabled(true);
```

Plusieurs méthodes concernent les statistiques sur l'utilisation des sessions :

Méthode	Rôle
long getCloseStatementCount()	Obtenir le nombre d'objets de type PrepareStatement qui ont été fermés
long getCollectionFetchCount()	
long getCollectionLoadCount()	Obtenir le nombre d'associations lues de la base de données
long getCollectionRecreateCount()	
long getCollectionRemoveCount()	Obtenir le nombre d'associations supprimées
String[] getCollectionRoleNames()	Obtenir le nom de toutes les associations
long getCollectionUpdateCount()	Obtenir le nombre d'associations mises à jour
long getConnectCount()	Obtenir le nombre de connexions demandées par les sessions
long getEntityDeleteCount()	Obtenir le nombre d'entités supprimées dans la base de données
long getEntityFetchCount()	
long getEntityInsertCount()	Obtenir le nombre d'entités insérées dans la base de données
long getEntityLoadCount()	Obtenir le nombre d'entités lues de la base de données
String[] getEntityNames()	Obtenir le nom de toutes les entités
long getEntityUpdateCount()	Obtenir le nombre d'entités mises à jour
long getFlushCount()	Obtenir le nombre de flushes implicites ou explicites fait par les sessions
long getOptimisticFailureCount()	Obtenir le nombre d'exceptions de type StaleObjectStateExceptions qui sont levées
long getPrepareStatementCount()	Obtenir le nombre de PrepareStatement
String[] getQueries()	Obtenir les requêtes SQL exécutées
long getQueryExecutionCount()	Obtenir le nombre de requêtes exécutées
long getQueryExecutionMaxTime()	Obtenir le temps de la requête dont l'exécution est la plus longue
String getQueryExecutionMaxTimeQueryString()	Obtenir la requête dont le temps d'exécution est le plus long
long getSessionCloseCount()	Obtenir le nombre de sessions fermées
long getSessionOpenCount()	Obtenir le nombre de sessions ouvertes
long getSuccessfulTransactionCount()	Obtenir le nombre de transactions qui ont réussies
long getTransactionCount()	Obtenir le nombre de transactions utilisées

Plusieurs méthodes permettent d'obtenir le nom des régions utilisées par le cache de second niveau : getEntityNames(), getCollectionRoleNames(), getQueries() et getSecondLevelCacheRegionNames().

Plusieurs méthodes concernent l'utilisation du cache de second niveau et des régions qu'il utilise.

Méthode	Rôle

CollectionStatistics getCollectionStatistics(String role)	Obtenir des statistiques pour une association
EntityStatistics getEntityStatistics(String entityName)	Obtenir des statistiques pour une entité
long getQueryCacheHitCount()	Obtenir le nombre de requêtes obtenues du cache
long getQueryCacheMissCount()	Obtenir le nombre de requêtes non obtenues du cache
long getQueryCachePutCount()	Obtenir le nombre de requêtes mises en cache
QueryStatistics getQueryStatistics(String queryString)	Obtenir des statistiques pour une requête
long getSecondLevelCacheHitCount()	Obtenir le nombre d'entités et d'associations obtenues du cache
long getSecondLevelCacheMissCount()	Obtenir le nombre d'entités et d'associations non obtenues du cache et donc relues de la base de données
long getSecondLevelCachePutCount()	Obtenir le nombre d'entités et associations mises dans le cache
String[] getSecondLevelCacheRegionNames()	Obtenir le nom de toutes les régions du cache
SecondLevelCacheStatistics getSecondLevelCacheStatistics(String regionName)	Obtenir les statistiques d'utilisation d'une région

Plusieurs méthodes concernent la gestion des statistiques

Méthode	Rôle
void clear()	Réinitialiser toutes les valeurs de statistiques
long getStartTime()	Date/heure de démarrage (en ms) de calcul des statistiques
boolean isStatisticsEnabled()	Renvoyer un booléen qui précise si les statistiques sont calculées ou non
void logSummary()	Inscrire dans le journal un résumé des statistiques
void setStatisticsEnabled(boolean b)	Activer ou non le calcul des statistiques

La méthode clear() de la classe Statistics permet de réinitialiser les valeurs des statistiques calculées par Hibernate.

La méthode logSummary() de la classe Statistics permet d'envoyer dans le journal un résumé des statistiques calculées par Hibernate avec un niveau Info.

Résultat :

```
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000161:
Logging statistics....
INFO     [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000251:
Start time: 1342448010929
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000242:
Sessions opened: 3
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000241:
Sessions closed: 3
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000266:
Transactions: 3
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000258:
Successful transactions: 3
INFO     [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000187:
Optimistic lock failures: 0
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000105:
Flushes: 3
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000048:
Connections obtained: 3
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000253:
Statements prepared: 1
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000252:
Statements closed: 0
```

```
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000239:
Second level cache puts: 7
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000237:
Second level cache hits: 14
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000238:
Second level cache misses: 0
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000079:
Entities loaded: 7
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000080:
Entities updated: 0
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000078:
Entities inserted: 0
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000076:
Entities deleted: 0
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000077:
Entities fetched (minimize this): 0
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000033:
Collections loaded: 0
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000036:
Collections updated: 0
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000035:
Collections removed: 0
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000034:
Collections recreated: 0
INFO     [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000032:
Collections fetched (minimize this): 0
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000438:
NaturalId cache puts: 0
INFO     [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000439:
NaturalId cache hits: 0
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000440:
NaturalId cache misses: 0
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000441:
Max NaturalId query time: 0ms
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000442:
NaturalId queries executed to database: 0
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000210:
Queries executed to database: 1
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000215:
Query cache puts: 1
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000433:
update timestamps cache puts: 0
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000434:
update timestamps cache hits: 0
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000435:
update timestamps cache misses: 2
INFO     [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000213:
Query cache hits: 2
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000214:
Query cache misses: 1
INFO    [main]:org.hibernate.stat.internal.ConcurrentStatisticsImpl - HHH000173:
Max query time: 455ms
```

Plusieurs classes permettent d'obtenir des informations sur une région particulière du cache.

Attention : les informations temporelles fournies par les statistiques possèdent une précision qui dépend de la JVM et est généralement est de 3 millisecondes ou plus.

La classe EntityStatistics encapsule des informations statistiques sur une entité du cache. Elle possède plusieurs méthodes pour obtenir les valeurs :

Méthodes	Rôles
long getLoadCount()	Nombre d'entités lues
long getFetchCount()	
long getInsertCount()	Nombre d'entités ajoutées
long getDeleteCount()	Nombre d'entités supprimées

long getUpdateCount()	Nombre d'entités mises à jour
long getOptimisticFailureCount()	Nombre de verrous optimistes qui ont échoués

Exemple :

```
      Statistics stats = sessionFactory.getStatistics();
      EntityStatistics entityStats = stats
         .getEntityStatistics("fr.jmdoudoux.dej.hibernatecache.entity.Pays");
      System.out.println(entityStats.getFetchCount());
      System.out.println(entityStats.getLoadCount());
      System.out.println(entityStats.getInsertCount());
      System.out.println(entityStats.getUpdateCount());
      System.out.println(entityStats.getDeleteCount());
      System.out.println(entityStats.getOptimisticFailureCount());
```

La classe CollectionStatistics encapsule des informations statistiques sur une collection du cache. Elle possède plusieurs méthodes pour obtenir les valeurs :

Méthodes	Rôles
long getLoadCount()	
long getFetchCount()	
long getRecreateCount()	
long getRemoveCount()	
long getUpdateCount()	

Exemple :

```
      Statistics stats = sessionFactory.getStatistics();
      CollectionStatistics collectionStats = stats
         .getCollectionStatistics("fr.jmdoudoux.dej.hibernatecache.entity.Devise.pays");
      System.out.println(collectionStats.getFetchCount());
      System.out.println(collectionStats.getLoadCount());
      System.out.println(collectionStats.getRecreateCount());
      System.out.println(collectionStats.getRemoveCount());
      System.out.println(collectionStats.getUpdateCount());
```

La classe QueryStatistics encapsule des informations statistiques sur une requête du cache. Elle possède plusieurs méthodes pour obtenir les valeurs :

Méthodes	Rôles
long getCacheHitCount()	Nombre d'objets retrouvés dans le cache lors des exécutions de cette requête
long getCacheMissCount()	Nombre d'objets non retrouvés dans le cache lors des exécutions de cette requête
long getCachePutCount()	Nombre d'objets mis dans le cache suite aux exécutions de cette requête
long getExecutionAvgTime()	Temps moyen d'exécution de la requête
long getExecutionCount()	Nombre d'invocations de la requête
long getExecutionMaxTime()	Temps maximum d'exécution de la requête
long getExecutionMinTime()	Temps minimum d'exécution de la requête
long getExecutionRowCount()	Nombre d'entités retournées par toutes les invocations de la requête

Exemple :

```
      Statistics stats = sessionFactory.getStatistics();
      QueryStatistics queryStats = stats.getQueryStatistics("from Pays");
      System.out.println(queryStats.getCacheHitCount());
      System.out.println(queryStats.getCacheMissCount());
      System.out.println(queryStats.getCachePutCount());
      System.out.println(queryStats.getExecutionAvgTime());
      System.out.println(queryStats.getExecutionCount());
      System.out.println(queryStats.getExecutionMaxTime());
      System.out.println(queryStats.getExecutionMinTime());
      System.out.println(queryStats.getExecutionRowCount());
```

La classe SecondLevelCacheStatistics encapsule des informations statistiques sur l'utilisation d'une région du cache. Elle possède plusieurs méthodes pour obtenir les valeurs :

Méthodes	Rôles
long getHitCount()	Nombre d'éléments obtenus de la région
long getMissCount()	Nombre d'éléments non obtenus de la région
long getPutCount()	Nombre d'éléments insérés dans la région
long getElementCountInMemory()	Nombre d'éléments dans la région en mémoire
long getElementCountOnDisk()	Nombre d'éléments dans la région stockés sur disque
long getSizeInMemory()	Nombre d'octets consommés en mémoire par la région
Map getEntries()	Obtenir les éléments contenus dans la région du cache

Il est préférable de mettre la valeur true à la propriété hibernate.cache.use_structured_entries si la méthode getEntries() est utilisée.

Exemple :

```
      Statistics stats = sessionFactory.getStatistics();
      SecondLevelCacheStatistics cacheStats = stats
        .getSecondLevelCacheStatistics("fr.jmdoudoux.dej.hibernatecache.entity.Pays");
      System.out.println(cacheStats.getElementCountInMemory());
      System.out.println(cacheStats.getElementCountOnDisk());
      System.out.println(cacheStats.getEntries());
      System.out.println(cacheStats.getHitCount());
      System.out.println(cacheStats.getMissCount());
      System.out.println(cacheStats.getPutCount());
      System.out.println(cacheStats.getSizeInMemory());
```

Il est possible d'exposer les statistiques via JMX en utilisant un Mbean de type StatisticsServiceMBean. Il suffit d'enregistrer dans le serveur de Mbeans une instance de type StatisticsService. Sa méthode setSessionFactory() permet de fournir en paramètre la SessionFactory dont les statistiques doivent être exposées.

Exemple :

```
      MBeanServer mBeanServer = ManagementFactory.getPlatformMBeanServer();
      ObjectName objName = new ObjectName("Hibernate:application=Statistics");
      StatisticsService statsMBean = new StatisticsService();
      statsMBean.setSessionFactory(sessionFactory);
      statsMBean.setStatisticsEnabled(true);
      mBeanServer.registerMBean(statsMBean, objName);
```

62.13. Les outils de génération de code

Hibernate fournit séparément un certain nombre d'outils. Ces outils sont livrés dans un fichier nommé hibernate-extensions-2.1.zip.

Il faut télécharger et décompresser le contenu de cette archive par exemple dans le répertoire où Hibernate a été décompressé.

L'archive contient deux répertoires :

- hibern8ide
- tools

Le répertoire tools propose trois outils :

- class2hbm :
- ddl2hbm :
- hbm2java :

Pour utiliser ces outils, il y a deux solutions possibles :

- utiliser les fichiers de commandes .bat fournis dans le répertoire /tools/bin
- utiliser ant pour lancer ces outils

Pour utiliser les fichiers de commandes .bat, il est nécessaire au préalable de configurer les paramètres dans le fichier setenv.bat. Il faut notamment correctement renseigner les valeurs associées aux variables JDBC_DRIVER qui précise le pilote de la base de données et HIBERNATE_HOME qui précise le répertoire où est installé Hibernate.

Pour utiliser les outils dans un script ant, il faut créer ou modifier un fichier build en ajoutant une tâche pour l'outil à utiliser.

Il faut copier le fichier hibernate2.jar et les fichiers contenus dans le répertoire /lib d'Hibernate dans le répertoire lib du projet. Il faut aussi copier dans ce répertoire les fichiers contenus dans le répertoire /tools/lib et le fichier /tools/hibernate-tools.jar.

Pour éviter les messages d'avertissement sur la configuration manquante de log4j, le plus simple est de copier le fichier /src/log4j.properties d'Hibernate dans le répertoire bin du projet.

Hibernate permet aussi de créer la structure du schéma d'une base de base de données en utilisant la configuration du mapping des entités grâce à un outil nommé hbm2ddl.

Il faut utiliser la propriété hibernate.hbm2ddl.auto qui peut prendre plusieurs valeurs :

- validate: valider la structure du schéma sans faire de modification dans la base de données
- update: mettre à jour le schéma
- create: créer le schéma en supprimant celui existant
- create-drop: créer le schéma et le supprimer lorsque la sessionFactory est fermée

Exemple :

```
<hibernate-configuration>

  <session-factory>
...
    <property name="hbm2ddl.auto">create</property>
...
  </session-factory>
</hibernate-configuration>
```

Lorsque les valeurs create et create-drop sont utilisées, il est possible de demander à Hibernate d'initialiser des données dans le schéma créé. Par défaut, Hibernate recherche un fichier import.sql dans le classpath. Si celui-ci est trouvé, Hibernate exécute les ordres SQL qu'il contient.

Depuis la version 3.6 d'Hibernate, la propriété hibernate.hbm2ddl.import_files permet de préciser un ou plusieurs fichiers qui seront utilisés pour créer les données. Si plusieurs fichiers sont indiqués, il faut les séparer avec un caractère virgule. Ils seront exécutés dans leur ordre de définition.

Exemple :

```
<hibernate-configuration>
 <session-factory>
...
  <property name="hbm2ddl.auto">create</property>
  <property name="hbm2ddl.import_files">/import-donnees-1.sql, /import-donnees-2.sql</property>
...
 </session-factory>
</hibernate-configuration>
```

Résultat :

```
INFO : org.hibernate.tool.hbm2ddl.SchemaExport -
Running hbm2ddl schema export
INFO : org.hibernate.tool.hbm2ddl.SchemaExport -
exporting generated schema to database
INFO : org.hibernate.tool.hbm2ddl.SchemaExport -
Executing import script: /import-donnees-1.sql
INFO : org.hibernate.tool.hbm2ddl.SchemaExport -
Executing import script: /import-donnees-2.sql
INFO : org.hibernate.tool.hbm2ddl.SchemaExport -
schema export complete
```

Cette fonctionnalité est particulièrement utile pour des tests automatisés mais n'est pas recommandée en production quand la gestion des évolutions du schéma doit être effectuée de manière plus attentive.

63. JPA (Java Persistence API)

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

L'utilisation pour la persistance d'un mapping O/R permet de proposer un niveau d'abstraction plus élevé que la simple utilisation de JDBC : ce mapping permet d'assurer la transformation d'objets vers la base de données et vice versa que cela soit pour des lectures ou des mises à jour (création, modification ou suppression).

Développée dans le cadre de la version 3.0 des EJB, cette API ne se limite pas aux EJB puisqu'elle peut aussi être mise en oeuvre dans des applications Java SE.

L'utilisation de l'API ne requiert aucune ligne de code mettant en oeuvre l'API JDBC.

L'API propose un langage d'interrogation similaire à SQL mais utilisant des objets plutôt que des entités relationnelles de la base de données.

L'API Java Persistence repose sur des entités qui sont de simples POJOs annotés et sur un gestionnaire de ces entités (EntityManager) qui propose des fonctionnalités pour les manipuler (ajout, modification suppression, recherche). Ce gestionnaire est responsable de la gestion de l'état des entités et de leur persistance dans la base de données.

Ce chapitre contient plusieurs sections :

- L'installation de l'implémentation de référence
- Les entités
- Le fichier de configuration du mapping
- L'utilisation du bean entité
- Le fichier persistence.xml
- La gestion des transactions hors Java EE
- La gestion des relations entre tables dans le mapping
- Le mapping de l'héritage de classes
- Les callbacks d'événements

63.1. L'installation de l'implémentation de référence

L'implémentation de référence est incluse dans le projet GlassFish. Elle peut être téléchargée à l'url : https://glassfish.dev.java.net/downloads/persistence/JavaPersistence.html

Cette implémentation de référence repose sur l'outil TopLink d'Oracle dans sa version essential.

Il suffit d'exécuter la commande java -jar avec en paramètre le fichier jar téléchargé.

Exemple :

```
C:\>java -jar glassfish-persistence-installer-v2-b52.jar
glassfish-persistence
glassfish-persistence\README
glassfish-persistence\3RD-PARTY-LICENSE.txt
glassfish-persistence\LICENSE.txt
```

```
glassfish-persistence\toplink-essentials-agent.jar
glassfish-persistence\toplink-essentials.jar
installation complete
```

Lisez la licence et si vous l'acceptez, cliquez sur le bouton « Accept ».

Un répertoire glassfish-persistence est créé contenant les bibliothèques de l'implémentation de référence de JPA.

63.2. Les entités

Les entités dans les spécifications de l'API Java Persistence permettent d'encapsuler les données d'une occurrence d'une ou plusieurs tables. Ce sont de simples POJO (Plain Old Java Object). Un POJO est une classe Java qui n'implémente aucune interface particulière ni n'hérite d'aucune classe mère spécifique.

Un objet Java de type POJO mappé vers une table de la base de données grâce à des méta data via l'API Java Persistence est nommé bean entité (Entity bean).

Un bean entité doit obligatoirement avoir un constructeur sans argument et la classe du bean doit obligatoirement être marquée avec l'annotation @javax.persistence.Entity. Cette annotation possède un attribut optionnel nommé name qui permet de préciser le nom de l'entité dans les requêtes. Par défaut, ce nom est celui de la classe de l'entité.

En tant que POJO, le bean entity n'a pas à implémenter d'interface particulière mais il doit respecter les règles de tous Java beans :

- Être déclaré avec l'annotation @javax.persistence.Entity
- Posséder au moins une propriété déclarée comme clé primaire avec l'annotation @Id

Le bean entity est composé de propriétés qui seront mappées sur les champs de la table de la base de données sous-jacente. Chaque propriété encapsule les données d'un champ d'une table. Ces propriétés sont utilisables au travers de simples accesseurs (getter/setter).

Une propriété particulière est la clé primaire qui sert d'identifiant unique dans la base de données mais aussi dans le POJO. Elle peut être de type primitif ou de type objet. La déclaration de cette clé primaire est obligatoire.

63.2.1. Le mapping entre le bean entité et la table

La description du mapping entre le bean entité et la table peut être faite de deux façons :

- Utiliser des annotations
- Utiliser un fichier XML de mapping

L'API propose plusieurs annotations pour supporter un mapping O/R assez complet.

Annotation	Rôle
@javax.persistence.Table	Préciser le nom de la table concernée par le mapping
@javax.persistence.Column	Associer un champ de la table à la propriété (à utiliser sur un getter)
@javax.persistence.Id	Associer un champ de la table à la propriété en tant que clé primaire (à utiliser sur un getter)
@javax.persistence.GeneratedValue	Demander la génération automatique de la clé primaire au besoin
@javax.persistence.Basic	Représenter la forme de mapping la plus simple. Cette annotation est utilisée par défaut
@javax.persistence.Transient	Demander de ne pas tenir compte du champ lors du mapping

L'annotation @javax.persistence.Table permet de lier l'entité à une table de la base de données. Par défaut, l'entité est liée à la table de la base de données correspondant au nom de la classe de l'entité. Si ce nom est différent alors l'utilisation de l'annotation @Table est obligatoire. C'est notamment le cas si des conventions de nommage des entités de la base de données sont mises en place.

L'annotation @Table possède plusieurs attributs :

Attributs	Rôle
name	Nom de la table
catalog	Catalogue de la table
schema	Schéma de la table
uniqueConstraints	Contraintes d'unicité sur une ou plusieurs colonnes

L'annotation @javax.persistence.Column permet d'associer un membre de l'entité à une colonne de la table. Par défaut, les champs de l'entité sont liés aux champs de la table dont les noms correspondent. Si ces noms sont différents alors l'utilisation de l'annotation @Column est obligatoire. C'est notamment le cas si des conventions de nommage des entités de la base de données sont mises en place.

L'annotation @Column possède plusieurs attributs :

Attributs	Rôle
name	Nom de la colonne
table	Nom de la table dans le cas d'un mapping multi-table
unique	Indique si la colonne est unique
nullable	Indique si la colonne est nullable
insertable	Indique si la colonne doit être prise en compte dans les requêtes de type insert
updatable	Indique si la colonne doit être prise en compte dans les requêtes de type update
columnDefinition	Précise le DDL de définition de la colonne
length	Indique la taille d'une colonne de type chaîne de caractères
precision	Indique la taille d'une colonne de type numérique
scale	Indique la précision d'une colonne de type numérique

Hormis les attributs name et table, tous les autres attributs ne sont utilisés que par un éventuel outil du fournisseur de l'implémentation de l'API pour générer automatiquement la table dans la base de données.

Il faut obligatoirement définir une des propriétés de la classe avec l'annotation @Id pour la déclarer comme étant la clé primaire de la table.

Cette annotation peut marquer soit le champ de la classe concernée soit le getter de la propriété. L'utilisation de l'un ou l'autre précise au gestionnaire s'il doit se baser sur les champs ou les getter pour déterminer les associations entre l'entité et les champs de la table. La clé primaire peut être constituée d'une seule propriété ou composées de plusieurs propriétés qui peuvent être de type primitif ou chaîne de caractères.

La clé primaire composée d'un seul champ peut être une propriété d'un type primitif, ou une chaîne de caractères (String).

La clé primaire peut être générée automatiquement en utilisant l'annotation @javax.persistence.GeneratedValue. Cette annotation possède plusieurs attributs :

Attributs	Rôle
strategy	Précise le type de générateur à utiliser : TABLE, SEQUENCE, IDENTITY ou AUTO. La valeur par défaut est AUTO
generator	Nom du générateur à utiliser

Exemple :

```
package fr.jmdoudoux.dej.jpa;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Personne implements Serializable {
  @Id
  @GeneratedValue
  private int id;

  private String prenom;

  private String nom;

  private static final long serialVersionUID = 1L;

  public Personne() {
    super();
  }

  public int getId() {
    return this.id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public String getPrenom() {
    return this.prenom;
  }

  public void setPrenom(String prenom) {
    this.prenom = prenom;
  }

  public String getNom() {
    return this.nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

}
```

Le type AUTO est le plus généralement utilisé : il laisse l'implémentation générer la valeur de la clé primaire.

Le type IDENTITY utilise un type de colonne spécial de la base de données.

Le type TABLE utilise une table dédiée qui stocke les clés des tables générées. L'utilisation de cette stratégie nécessite l'utilisation de l'annotation @javax.persistence.TableGenerator

L'annotation @TableGenerator possède plusieurs attributs :

Attributs	Rôle

name	Nom identifiant le TableGenerator : il devra être utilisé comme valeur dans l'attribut generator de l'annotation @Id
table	Nom de la table utilisée
catalog	Nom du catalogue utilisé
schema	Nom du schéma utilisé
pkColumnName	Nom de la colonne qui précise la clé primaire à générer
valueColumnName	Nom de la colonne qui contient la valeur de la clé primaire générée
pkColumnValue	
allocationSize	Valeur utilisée lors de l'incrémentation de la valeur de la clé primaire
uniqueConstraints	

Le type SEQUENCE utilise un mécanisme nommé séquence proposé par certaines bases de données notamment celles d'Oracle. L'utilisation de cette stratégie nécessite l'utilisation de l'annotation @javax.persistence.SequenceGenerator

L'annotation @SequenceTableGenerator possède plusieurs attributs :

Attributs	Rôle
name	Nom identifiant le SequenceTableGenerator : il devra être utilisé comme valeur dans l'attribut generator de l'annotation @Id
sequenceName	Nom de la séquence dans la base de données
initialValue	Valeur initiale de la séquence
allocationSize	Valeur utilisée lors de l'incrémentation de la valeur de la séquence

L'annotation @SequenceGenerator s'utilise sur la classe de l'entité

Exemple :

```
@Entity
@Table(name="PERSONNE")
@SequenceGenerator(name="PERSONNE_SEQUENCE",
sequenceName="PERSONNE_SEQ")
public class Personne implements Serializable {
  @Id
  @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="PERSONNE_SEQUENCE")
  private int id;
```

Le modèle de base de données relationnelle permet la définition d'une clé primaire composée de plusieurs colonnes. L'API Java Persistence propose deux façons de gérer ce cas de figure :

- L'annotation @javax.persistence.IdClass
- L'annotation @javax.persistence.EmbeddedId

L'annotation @IdClass s'utilise avec une classe qui va encapsuler les propriétés qui composent la clé primaire. Cette classe doit obligatoirement :

- Être sérialisable
- Posséder un constructeur sans argument
- Fournir une implémentation dédiée des méthodes equals() et hashCode()

Exemple : la clé primaire est composée des champs nom et prenom (exemple théorique qui présume que deux personnes ne peuvent avoir le même nom et prénom)

```
package fr.jmdoudoux.dej.jpa;

public class PersonnePK implements java.io.Serializable {

  private static final long serialVersionUID = 1L;

  private String nom;

  private String prenom;

  public PersonnePK() {
  }

  public PersonnePK(String nom, String prenom) {
    this.nom = nom;
    this.prenom = prenom;
  }

  public String getNom() {
    return this.nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

  public String getPrenom() {
    return prenom;
  }

  public void setPrenom(String prenom) {
    this.nom = prenom;
  }

  public boolean equals(Object obj) {
    boolean resultat = false;

    if (obj == this) {
      resultat = true;
    } else {
      if (!(obj instanceof PersonnePK)) {
        resultat = false;
      } else {
        PersonnePK autre = (PersonnePK) obj;
        if (!nom.equals(autre.nom)) {
          resultat = false;
        } else {
          if (prenom != autre.prenom) {
            resultat = false;
          } else {
            resultat = true;
          }
        }
      }
    }
    return resultat;
  }

  public int hashCode() {
    return (nom + prenom).hashCode();
  }
}
```

Il est nécessaire de définir la classe de la clé primaire dans le fichier de configuration persistence.xml

Exemple :

```
<?xml version="1.0"  encoding="UTF-8"?>
<persistence  xmlns="http://java.sun.com/xml/ns/persistence"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              version="1.0"  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
              http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
```

```
    <persistence-unit  name="MaBaseDeTestPU">
         <provider>oracle.toplink.essentials.PersistenceProvider</provider>
         <class>fr.jmdoudoux.dej.jpa.Personne</class>
         <class>fr.jmdoudoux.dej.jpa.PersonnePK</class>
    </persistence-unit>
</persistence>
```

L'annotation @IdClass possède un seul attribut :

Attributs	Rôle
Class	Classe qui encapsule la clé primaire composée

Il faut utiliser l'annotation@IdClass sur la classe de l'entité.

Il est nécessaire de marquer chacune des propriétés de l'entité qui compose la clé primaire avec l'annotation @Id. Ces propriétés doivent avoir le même nom dans l'entité et dans la classe qui encapsule la clé primaire.

Exemple :

```
package fr.jmdoudoux.dej.jpa;

import java.io.Serializable;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.IdClass;

@Entity
@IdClass(PersonnePK.class)
public class Personne implements Serializable {
  private String prenom;
  private String nom;
  private int taille;

  private static final long serialVersionUID = 1L;

  public Personne() {
    super();
  }

  @Id
  public String getPrenom() {
    return this.prenom;
  }

  public void setPrenom(String prenom) {
    this.prenom = prenom;
  }

  @Id
  public String getNom() {
    return this.nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

  public int getTaille() {
    return this.taille;
  }

  public void setTaille(int taille) {
    this.taille = taille;
  }
}
```

Remarque : il n'est pas possible de demander la génération automatique d'une clé primaire composée. Les valeurs de chacune des propriétés de la clé doivent être fournies explicitement.

La classe de la clé primaire est utilisée notamment lors des recherches.

Exemple :

```
PersonnePK clePersonne = new PersonnePK("nom1", "prenom1");
Personne personne = entityManager.find(Personne.class, clePersonne);
```

L'annotation @EmbeddedId s'utilise avec l'annotation @javax.persistence.Embeddable

Exemple :

```
package fr.jmdoudoux.dej.jpa;

import javax.persistence.Embeddable;

@Embeddable
public class PersonnePK implements java.io.Serializable {

  private static final long serialVersionUID = 1L;

  private String nom;

  private String prenom;

  public PersonnePK() {
  }

  public PersonnePK(String nom, String prenom) {
    this.nom = nom;
    this.prenom = prenom;
  }

  public String getNom() {
    return this.nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

  public String getPrenom() {
    return prenom;
  }

  public void setPrenom(String prenom) {
    this.nom = prenom;
  }

  public boolean equals(Object obj) {
    boolean resultat = false;

    if (obj == this) {
      resultat = true;
    } else {
      if (!(obj instanceof PersonnePK)) {
        resultat = false;
      } else {
        PersonnePK autre = (PersonnePK) obj;
        if (!nom.equals(autre.nom)) {
          resultat = false;
        } else {
          if (prenom != autre.prenom) {
            resultat = false;
          } else {
            resultat = true;
          }
        }
      }
    }
    return resultat;
  }

  public int hashCode() {
    return (nom + prenom).hashCode();
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.jpa;

import java.io.Serializable;

import javax.persistence.EmbeddedId;
import javax.persistence.Entity;

@Entity
public class Personne implements Serializable {
  @EmbeddedId
  private PersonnePK clePrimaire;
  private int taille;

  private static final long serialVersionUID = 1L;

  public Personne() {
    super();
  }

  public PersonnePK getClePrimaire() {
    return this.clePrimaire;
  }

  public void setNom(PersonnePK clePrimaire) {
    this.clePrimaire = clePrimaire;
  }

  public int getTaille() {
    return this.taille;
  }

  public void setTaille(int taille) {
    this.taille = taille;
  }
}
```

La classe qui encapsule la clé primaire est utilisée notamment dans les recherches

Exemple :

```
PersonnePK clePersonne = new PersonnePK("nom1", "prenom1");
Personne personne = entityManager.find(Personne.class, clePersonne);
```

L'annotation @AttributeOverrides est une collection d'attribut @AttributeOverride. Ces annotations permettent de ne pas avoir à utiliser l'annotation @Column dans la classe de la clé ou de modifier les attributs de cette annotation dans l'entité qui la met en oeuvre.

L'annotation @AttributeOverride possède plusieurs attributs :

Attributs	Rôle
name	Précise le nom de la propriété de la classe imbriquée
column	Précise la colonne de la table à associer à la propriété

Exemple :

```
package fr.jmdoudoux.dej.jpa;

import java.io.Serializable;

import javax.persistence.AttributeOverrides;
import javax.persistence.AttributeOverride;
import javax.persistence.EmbeddedId;
import javax.persistence.Entity;
import javax.persistence.Column;

@Entity
public class Personne4 implements Serializable {
  @EmbeddedId
  @AttributeOverrides({
    @AttributeOverride(name="nom", column=@Column(name="NOM") ),
    @AttributeOverride(name="prenom", column=@Column(name="PRENOM") )
    })
  private PersonnePK clePrimaire;
  private int taille;
...
```

Par défaut, toutes les propriétés sont mappées sur la colonne correspondante dans la table. L'annotation @javax.persistence.Transient permet d'indiquer au gestionnaire de persistance d'ignorer cette propriété.

L'annotation @javax.persistence.Basic représente la forme de mapping la plus simple. C'est aussi celle par défaut ce qui rend son utilisation optionnelle. Ce mapping concerne les types primitifs, les wrappers de type primitifs, les tableaux de ces types et les types java.math.BigInteger, java.math.BigDecimal, java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time et java.sql.Timestamp.

L'annotation @Basic possède plusieurs attributs :

Attributs	Rôle
fetch	Permet de préciser comment la propriété est chargée selon deux modes : • LAZY : la valeur est chargée uniquement lors de son utilisation • EAGER : la valeur est toujours chargée (valeur par défaut) Cette fonctionnalité permet de limiter la quantité de données obtenue par une requête
optionnal	Indique que la colonne est nullable

Généralement, cette annotation peut être omise sauf dans le cas où le chargement de la propriété doit être de type LAZY.

Exemple :

```
package fr.jmdoudoux.dej.jpa;

import java.io.Serializable;

import javax.persistence.Basic;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Personne implements Serializable {
  @Id
  @GeneratedValue
  private int id;

  @Basic(fetch=FetchType.LAZY, optional=false)
  private String prenom;
  private String nom;
...
```

L'annotation @javax.persistence.Temporal permet de fournir des informations complémentaires sur la façon dont les propriétés encapsulant des données temporelles (Date et Calendar) sont associées aux colonnes dans la table (date, time ou timestamp). La valeur par défaut est timestamp.

Exemple :

```
package fr.jmdoudoux.dej.jpa;

import java.io.Serializable;
import java.util.Date;

import javax.persistence.Basic;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
public class Personne implements Serializable {
  @Id
  @GeneratedValue
  private int id;

  @Basic(fetch = FetchType.LAZY, optional = false)
  private String prenom;

  private String nom;

  @Temporal(TemporalType.TIME)
  private Date heureNaissance;

  private static final long serialVersionUID = 1L;

  public Personne() {
    super();
  }

  public int getId() {
    return this.id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public Date getHeureNaissance() {
    return heureNaissance;
  }

  public void setTimeCreated(Date heureNaissance) {
    this.heureNaissance = heureNaissance;
  }
...
```

63.2.2. Le mapping de propriétés complexes

L'API Java persistence permet de mapper des colonnes qui concernent des données de type plus complexe que les types de base tels que les champs blob, clob ou des objets.

L'annotation @javax.persistence.Lob permet de mapper une propriété sur une colonne de type Blob ou Clob selon le type de la propriété :

- Blob pour les tableaux de byte ou Byte ou les objets sérializables

- Clob pour les chaînes de caractères et les tableaux de caractères char ou Char

Fréquemment ce type de propriété est chargé de façon LAZY.

Exemple :

```
package fr.jmdoudoux.dej.jpa;

import java.io.Serializable;

import javax.persistence.Basic;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Lob;

import com.sun.imageio.plugins.jpeg.JPEG;

@Entity
public class Personne implements Serializable {
  @Id
  @GeneratedValue
  private int id;

  @Basic(fetch = FetchType.LAZY, optional = false)
  private String prenom;

  private String nom;

  @Lob
  @Basic(fetch = FetchType.LAZY)
  private JPEG photo;

  private static final long serialVersionUID = 1L;

  public Personne() {
    super();
  }

  public int getId() {
    return this.id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public JPEG getPhoto() {
    return photo;
  }

  public void setPhoto(JPEG photo) {
    this.photo = photo;
  }

  ...
```

L'annotation @javax.persistence.Enumerated permet d'associer une propriété de type énumération à une colonne de la table sous la forme d'un numérique ou d'une chaîne de caractères.

Cette forme est précisée en paramètre de l'annotation grâce à l'énumération EnumType qui peut avoir comme valeur EnumType.ORDINAL (valeur par défaut) ou EnumType.STRING.

Exemple : énumération des genres d'une personne

```
package fr.jmdoudoux.dej.jpa;

public enum Genre {
   HOMME,
   FEMME,
   INCONNU
}
```

Exemple :

```
package fr.jmdoudoux.dej.jpa;

import java.io.Serializable;

import javax.persistence.Basic;
import javax.persistence.Entity;
import javax.persistence.EnumType;
import javax.persistence.Enumerated;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Personne implements Serializable {
  @Id
  @GeneratedValue
  private int id;

  @Basic(fetch = FetchType.LAZY, optional = false)
  private String prenom;

  private String nom;

  @Enumerated(EnumType.STRING)
  private Genre genre;

  private static final long serialVersionUID = 1L;

  public Personne() {
    super();
  }

  public int getId() {
    return this.id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public Genre getGenre() {
    return genre;
  }

  public void setGenre(Genre genre) {
    this.genre = genre;
  }

  ...
```

63.2.3. Le mapping d'une entité sur plusieurs tables

Le modèle objet et le modèle relationnel associé ne correspondent pas toujours car les critères de conception ne sont pas forcement les mêmes. Ainsi, il est courant d'avoir une entité qui mappe des colonnes de plusieurs tables.

Exemple : création de la table adresse utilisée dans cette section

```
ij> create table ADRESSE
(
ID_ADRESSE integer primary key not null,
RUE varchar(250) not null,
CODEPOSTAL varchar(7) not null,
VILLE varchar(250) not null
);
```

```
0 lignes insérées/mises à jour/supprimées
ij> INSERT INTO ADRESSE VALUES (1,'rue1','11111','ville1'), (2,'rue2','22222','v
ille2'), (3,'rue3','33333','ville3');
3 lignes insérées/mises à jour/supprimées
ij> select * from adresse;
ID_ADRESSE |RUE
                                                            |CODEPO&|VILLE


--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
-------------------------------------
1          |rue1
                                                            |11111  |ville1


2          |rue2
                                                            |22222  |ville2


3          |rue3
                                                            |33333  |ville3



3 lignes sélectionnées
ij>
```

L'annotation @javax.persistence.SecondaryTable permet de préciser qu'une autre table sera utilisée dans le mapping.

Pour utiliser cette fonctionnalité, la seconde table doit posséder une jointure entre sa clé primaire et une ou plusieurs colonnes de la première table.

L'annotation @SecondaryTable possède plusieurs attributs :

Attribut	Rôle
Name	Nom de la table
Catalogue	Nom du catalogue
Schema	Nom du schéma
pkJoinsColumns	Collection des clés primaires de la jointure sous la forme d'annotations de type @PrimaryKeyJoinColumn
uniqueConstraints	

L'annotation @PrimaryKeyJoinColumn permet de préciser une colonne qui compose la clé primaire de la seconde table et entre dans la jointure avec la première table. Elle possède plusieurs attributs :

Attribut	Rôle
name	Nom de la colonne
referencedColumnName	Nom de la colonne dans la première table (obligatoire si les noms de colonnes sont différents entre les deux tables)
columnDefinition	

Il est nécessaire pour chaque propriété de l'entité qui est mappée sur la seconde table de renseigner le nom de la table dans l'attribut table de l'annotation @Column

Exemple : la classe PersonneAdresse

```
package fr.jmdoudoux.dej.jpa;

import java.io.Serializable;

import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.SecondaryTable;
import javax.persistence.Table;

@Entity
@Table(name="PERSONNE")
@SecondaryTable(name="ADRESSE",
pkJoinColumns={
@PrimaryKeyJoinColumn(name="ID_ADRESSE")})
public class PersonneAdresse implements Serializable {
  @Id
  @GeneratedValue
  private int id;

  @Basic(fetch = FetchType.LAZY, optional = false)
  private String prenom;

  private String nom;

  @Column(name="RUE", table="ADRESSE")
  private String rue;

  @Column(name="CODEPOSTAL", table="ADRESSE")
  private String codePostal;

  @Column(name="VILLE", table="ADRESSE")
  private String ville;

  private static final long serialVersionUID = 1L;

  public PersonneAdresse() {
    super();
  }

  public int getId() {
    return this.id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public String getPrenom() {
    return this.prenom;
  }

  public void setPrenom(String prenom) {
    this.prenom = prenom;
  }

  public String getNom() {
    return this.nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

  public String getCodePostal() {
    return codePostal;
  }

  public void setCodePostal(String codePostal) {
    this.codePostal = codePostal;
  }

  public String getRue() {
    return rue;
  }

  public void setRue(String rue) {
    this.rue = rue;
  }

  public String getVille() {
    return ville;
  }

  public void setVille(String ville) {
    this.ville = ville;
  }

}
```

Résultat :

```
nom prenom=nom1 prenom1
adresse=rue1, 11111 ville1
```

Si le mapping d'une entité met en oeuvre plus de deux tables, il faut utiliser l'annotation @javax.persistence.SecondaryTables qui est une collection d'annotations @SecondaryTable

Exemple :

```
package fr.jmdoudoux.dej.jpa;

import java.io.Serializable;

import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.SecondaryTable;
import javax.persistence.SecondaryTables;
import javax.persistence.Table;

@Entity
@Table(name="PERSONNE")
@SecondaryTables({
  @SecondaryTable(name="ADRESSE",
     pkJoinColumns={@PrimaryKeyJoinColumn(name="ID_ADRESSE")}),
  @SecondaryTable(name="INFO_PERS",
     pkJoinColumns={@PrimaryKeyJoinColumn (name="ID_INFO_PERS")})
})
public class PersonneAdresse implements Serializable {
...
```

63.2.4. L'utilisation d'objets embarqués dans les entités

L'API Java Persistence permet d'utiliser dans les entités des objets Java qui ne sont pas des entités mais qui sont agrégés dans l'entité et dont les propriétés seront mappées sur les colonnes correspondantes dans la table.

La mise en oeuvre de cette fonctionnalité est similaire à celle utilisée avec l'annotation @EmbeddedId pour les clés primaires composées.

La classe embarquée est un simple POJO qui doit être marquée avec l'annotation @javax.persistence.Embeddable

Exemple :

```
package fr.jmdoudoux.dej.jpa;

import java.io.Serializable;

import javax.persistence.Column;
import javax.persistence.Embeddable;

@Embeddable
public class Adresse implements Serializable{

  private static final long serialVersionUID = 1L;

  @Column(name="RUE", table="ADRESSE")
  private String rue;

  @Column(name="CODEPOSTAL", table="ADRESSE")
  private String codePostal;

  @Column(name="VILLE", table="ADRESSE")
  private String ville;

  public String getCodePostal() {
    return codePostal;
  }

  public void setCodePostal(String codePostal) {
    this.codePostal = codePostal;
  }

  public String getRue() {
    return rue;
  }

  public void setRue(String rue) {
    this.rue = rue;
  }

  public String getVille() {
    return ville;
  }

  public void setVille(String ville) {
    this.ville = ville;
  }

}
```

Les propriétés de cette classe peuvent être marquées avec l'annotation @Column au besoin.

Dans l'entité, il faut utiliser l'annotation @javax.persistence.Embedded sur la propriété du type de la classe embarquée.

Exemple :

```
package fr.jmdoudoux.dej.jpa;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class TestJPA3 {

  public static void main(String[] argv) {
    EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("MaBaseDeTestPU");
    EntityManager em = emf.createEntityManager();
    PersonneAdresse2 personneAdresse = em.find(PersonneAdresse2.class, 1);
    System.out.println("nom prenom="
        + personneAdresse.getNom()
        + " "
        + personneAdresse.getPrenom());
    System.out.println("adresse="
        + personneAdresse.getAdresse().getRue()
        + ", "
        + personneAdresse.getAdresse().getCodePostal()
        + " "
        + personneAdresse.getAdresse().getVille());
    em.close();
    emf.close();
  }
}
```

L'annotation @AttributesOverride peut être utilisée pour adapter au contexte le mapping des propriétés de l'objet embarqué.

Exemple :

```
nom prenom=nom1 prenom1
adresse=rue1, 11111 ville1
```

Si l'annotation @Embedded n'est pas utilisée alors le gestionnaire de persistance va mapper la propriété sur le champ correspondant sous sa forme sérialisée. Avec l'annotation @Embedded chaque propriété de l'objet embarqué est mappée sur la colonne correspondante de la table.

63.3. Le fichier de configuration du mapping

Il est aussi possible de définir le mapping dans un fichier de mapping nommé par défaut orm.xml stocké dans le répertoire META-INF.

Ce fichier orm.xml est un fichier au format xml. L'élément racine est le tag <entity-mappings>.

Pour chaque entité, il faut utiliser un tag fils <entity>. Ce tag possède deux attributs :

- Class qui permet de préciser le nom pleinement qualifié de la classe de l'entité
- Access qui permet de préciser le type d'accès aux données (PROPERTY pour un accès via les getter/setter ou FIELD pour un accès via les champs).

La déclaration de la clé primaire se fait dans un tag <id> fils d'un tag <attributes>. Ce tag <id> possède un attribut nommé name qui permet de préciser le nom du champ qui est la clé primaire.

La suite de ce chapitre sera développée dans une version future de ce document

Le fichier de mapping peut aussi avoir un nom arbitraire mais dans ce cas, il devra être précisé avec le tag <mapping-file> dans le fichier de configuration persistence.xml

63.4. L'utilisation du bean entité

Comme c'est un POJO, il est possible d'ajouter des méthodes à la classe mais il est cependant conseillé de maintenir le rôle du bean entity au transfert de données : il faut éviter de lui ajouter des méthodes métiers mais il est possible de définir des méthodes de validation des données qu'il encapsule.

La mise en oeuvre de POJO permet de les utiliser directement lors d'échanges entre le client et le serveur car ils peuvent être sérialisés comme tout objet de base Java.

Les POJO ne servent qu'à définir le mapping et encapsuler des données. L'instanciation d'une entité n'a aucune conséquence sur la table de la base de données mappée avec l'objet.

Toutes les actions de persistance sur ces objets sont réalisées grâce à un objet dédié de l'API : l'EntityManager.

63.4.1. L'utilisation du bean entité

Un contexte de persistance (persistence context) est un ensemble d'entités géré par un EntityManager.

Les entités peuvent ainsi être de deux types :

- Gérée (Managed) :
- Non gérée (Unmanaged) :

Lorsqu'un contexte de persistance est fermé, toutes les entités du contexte deviennent non gérées.

Il existe deux types de contexte de persistance :

- Transaction-scoped : le contexte est ouvert pendant toute la durée d'une transaction. La fermeture de la transaction entraîne la fermeture du contexte. Ce type de contexte n'est utilisable que dans le cadre de l'utilisation dans un conteneur qui va assurer la prise en charge de la transaction
- Extended persitence : le contexte reste ouvert après la fermeture de la transaction

63.4.2. L'EntityManager

Les interactions entre la base de données et les beans entités sont assurées par un objet de type javax.persistence.EntityManager : il permet de lire et rechercher des données mais aussi de les mettre à jour (ajout, modification, suppression). L'EntityManager est donc au coeur de toutes les actions de persistance.

Les beans entités étant de simple POJO, leur instanciation se fait comme pour tout autre objet Java. Les données de cette instance ne sont rendues persistantes que par une action explicite demandée à l'EntityManager sur le bean entité.

L'EntityManager assure aussi les interactions avec un éventuel gestionnaire de transactions.

Un EntityManager gère un ensemble défini de beans entités nommé persistence unit. La définition d'un persistence unit est assurée dans un fichier de description nommé persistence.xml.

63.4.2.1. L'obtention d'une instance de la classe EntityManager

Lors d'une utilisation dans un conteneur Java EE, il est possible d'obtenir un objet de type EntityManager en utilisant l'injection de dépendance pour l'objet lui-même ou d'obtenir une fabrique de type EntityManagerFactory qui sera capable de créer l'objet.

Dans un environnement Java SE, comme par exemple dans Tomcat ou dans une application de type client lourd, l'instanciation d'un objet de type EntityManager doit être codée.

Sous Java SE, pour obtenir une instance de type EntityManager, il faut utiliser une fabrique de type EntityManagerFactory. Cette fabrique propose la méthode createEntityManager() pour obtenir une instance.

Pour obtenir une instance de la fabrique, il faut utiliser la méthode statique createEntityManagerFactory() de la classe javax.persistence.Persistence qui attend en paramètre le nom de l'unité de persistence à utiliser. Elle va rechercher le fichier persistence.xml dans le classpath et recherche dans ce fichier l'unité de persistance dont le nom est fourni.

Pour libérer les ressources, il faut utiliser la méthode close() de la fabrique une fois que cette dernière n'a plus d'utilité.

Sous Java EE, il est préférable d'utiliser l'injection de dépendance pour obtenir une fabrique ou un contexte de persistance.

L'annotation @javax.persistence.PersistenceUnit sur un champ de type EntityManagerFactory permet d'injecter une fabrique. Cette annotation possède un attribut unitName qui précise le nom de l'unité de persistance.

Exemple :

```
@PersistenceUnit(unitName="MaBaseDeTestPU")
private EntityManagerFactory factory;
```

Il est alors possible d'utiliser la fabrique pour obtenir un objet de type EntityManager qui encapsule un contexte de persistence de type extended. Pour associer ce contexte à la transaction courante, il faut utiliser la méthode joinTransaction().

La méthode close() est automatiquement appelée par le conteneur : il ne faut pas utiliser cette méthode dans un conteneur sinon une exception de type IllegalStateException est levée.

L'annotation @javax.persistence.PersistenceContext sur un champ de type EntityManager permet d'injecter un contexte de persistance. Cette annotation possède un attribut unitName qui précise le nom de l'unité de persistance.

Exemple :

```
@PersistenceContext(unitName="MaBaseDeTestPU")
private EntityManager entityManager;
```

63.4.2.2. L'utilisation de la classe EntityManager

La méthode contains() de l'EntityManager permet de savoir si une instance fournie en paramètre est gérée par le contexte. Dans ce cas, elle renvoie true, sinon elle renvoie false.

La méthode clear() de l'EntityManager permet de détacher toutes les entités gérées par le contexte. Dans ce cas, toutes les modifications apportées aux entités sont perdues : il est préférable d'appeler la méthode flush() avant la méthode clear() afin de rendre persistante toutes les modifications.

L'appel des méthodes de mise à jour persist(), merge() et remove() ne réalise pas d'actions immédiates dans la base de données sous-jacente. L'exécution de ces actions est à la discrétion de l'EntityManager selon le FlushModeType (AUTO ou COMMIT).

Dans le mode AUTO, les mises à jour sont reportées dans la base de données avant chaque requête. Dans le mode COMMIT, les mises à jour sont reportées dans la base de données lors du commit de la transaction.

Le mode COMMIT est plus performant car il limite les échanges avec la base de données.

Il est possible de forcer l'enregistrement des mises à jour dans la base de données en utilisant la méthode flush() de l'EntityManager.

63.4.2.3. L'utilisation de la classe EntityManager pour la création d'une occurrence

Pour insérer une nouvelle entité dans la base de données, il faut :

- Instancier une occurrence de la classe de l'entité
- Initialiser les propriétés de l'entité
- Définir les relations de l'entité avec d'autres entités si besoin
- Utiliser la méthode persist() de l'EntityManager en passant en paramètre l'entité

Exemple :

```
package fr.jmdoudoux.dej.jpa;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;

public class TestJPA4 {
  public static void main(String[]argv) {
    EntityManagerFactory emf = Persistence.createEntityManagerFactory("MaBaseDeTestPU");
    EntityManager em = emf.createEntityManager();
    EntityTransaction transac = em.getTransaction();
    transac.begin();
    Personne nouvellePersonne = new Personne();
    nouvellePersonne.setId(4);
    nouvellePersonne.setNom("nom4");
    nouvellePersonne.setPrenom("prenom4");
    em.persist(nouvellePersonne);
    transac.commit();

    em.close();
    emf.close();
  }
}
```

Remarque : l'exemple ci-dessous utilise JPA dans un environnement qui ne propose aucune fonctionnalité pour assurer les transactions (Java SE) : il est donc nécessaire de créer et gérer manuellement une transaction afin d'assurer la persistance des données.

63.4.2.4. L'utilisation de la classe EntityManager pour rechercher des occurrences

Pour effectuer des recherches de données, l'EntityManager propose deux mécanismes :

- La recherche à partir de la clé primaire
- La recherche à partir d'une requête utilisant une syntaxe dédiée

Pour la recherche par clé primaire, la classe EntityManager possède les méthodes find() et getReference() qui attendent toutes les deux en paramètres un objet de type Class représentant la classe de l'entité et un objet qui contient la valeur de la clé primaire.

La méthode find() renvoie null si l'occurrence n'est pas trouvée dans la base de données.

Exemple :

```
package fr.jmdoudoux.dej.jpa;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class TestJPA5 {
  public static void main(String[] argv) {
    EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("MaBaseDeTestPU");
    EntityManager em = emf.createEntityManager();

    Personne personne = em.find(Personne.class, 4);
    if (personne != null) {
      System.out.println("Personne.nom=" + personne.getNom());
    }
    em.close();
    emf.close();
  }
}
```

La méthode getReference() lève une exception de type javax.persistence.EntityNotFoundException si l'occurrence n'est pas trouvée dans la base de données.

Exemple :

```
package fr.jmdoudoux.dej.jpa;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityNotFoundException;
import javax.persistence.Persistence;

public class TestJPA6 {
  public static void main(String[] argv) {
    EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("MaBaseDeTestPU");
    EntityManager em = emf.createEntityManager();

    try {
    Personne personne = em.getReference(Personne.class, 5);
      System.out.println("Personne.nom=" + personne.getNom());
    } catch (EntityNotFoundException e) {
      System.out.println("personne non trouvée");
    }
    em.close();
    emf.close();
  }
}
```

63.4.2.5. L'utilisation de la classe EntityManager pour rechercher des données par requête

La recherche par requête repose sur des méthodes dédiées de la classe EntityManager (createQuery(), createNamedQuery() et createNativeQuery()) et sur un langage de requête spécifique nommé EJB QL.

L'objet Query encapsule et permet d'obtenir les résultats de son exécution. La méthode getSingleResult() permet d'obtenir un objet unique retourné par la requête.

Exemple :

```
package fr.jmdoudoux.dej.jpa;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class TestJPA7 {
  public static void main(String[] argv) {
    EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("MaBaseDeTestPU");
    EntityManager em = emf.createEntityManager();

    Query query = em.createQuery("select p from Personne p where p.nom='nom2'");
    Personne personne = (Personne) query.getSingleResult();
    if (personne == null) {
      System.out.println("Personne non trouvée");
    } else {
      System.out.println("Personne.nom=" + personne.getNom());
    }

    em.close();
    emf.close();
  }
}
```

La méthode getResultList() renvoie une collection qui contient les éventuelles occurrences retournées par la requête.

Exemple :

```
package fr.jmdoudoux.dej.jpa;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class TestJPA8 {
  public static void main(String[] argv) {
    EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("MaBaseDeTestPU");
    EntityManager em = emf.createEntityManager();

    Query query = em.createQuery("select p.nom from Personne p where p.id > 2");
    List noms = query.getResultList();
    for (Object nom : noms) {
      System.out.println("nom = "+nom);
    }

    em.close();
    emf.close();
  }
}
```

L'objet Query gère aussi des paramètres nommés dans la requête. Le nom de chaque paramètre est préfixé par « : » dans la requête. La méthode setParameter() permet de fournir une valeur à chaque paramètre.

Exemple :

```
package fr.jmdoudoux.dej.jpa;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class TestJPA9 {
  public static void main(String[] argv) {
    EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("MaBaseDeTestPU");
    EntityManager em = emf.createEntityManager();

    Query query = em.createQuery("select p.nom from Personne p where p.id > :id");
    query.setParameter("id", 1);
    List noms = query.getResultList();
    for (Object nom : noms) {
      System.out.println("nom = "+nom);
    }

    em.close();
    emf.close();
  }
}
```

63.4.2.6. L'utilisation de la classe EntityManager pour modifier une occurrence

Pour modifier une entité existante dans la base de données, il faut :

- Obtenir une instance de l'entité à modifier (par recherche sur la clé primaire ou l'exécution d'une requête)
- Modifier les propriétés de l'entité
- Selon le mode de synchronisation des données de l'EntityManager, il peut être nécessaire d'appeler la méthode flush() explicitement

Exemple :

```
package fr.jmdoudoux.dej.jpa;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class TestJPA10 {
  public static void main(String[] argv) {
    EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("MaBaseDeTestPU");
    EntityManager em = emf.createEntityManager();

    EntityTransaction transac = em.getTransaction();
    transac.begin();

    Query query = em.createQuery("select p from Personne p where p.nom='nom2'");
    Personne personne = (Personne) query.getSingleResult();
    if (personne == null) {
      System.out.println("Personne non trouvée");
    } else {
      System.out.println("Personne.prenom=" + personne.getPrenom());

      personne.setPrenom("prenom2 modifie");
      em.flush();

      personne = (Personne) query.getSingleResult();
      System.out.println("Personne.prenom=" + personne.getPrenom());
    }

    transac.commit();

    em.close();
    emf.close();
  }
}
```

63.4.2.7. L'utilisation de la classe EntityManager pour fusionner des données

L'EntityManager propose la méthode merge() pour fusionner les données d'une entité non gérée avec la base de données. Ceci est particulièrement utile notamment lorsque l'entité est sérialisée pour être envoyée au client : dans ce cas, l'entité n'est plus gérée par le contexte. Lorsque le client renvoie l'entité modifiée, il faut synchroniser les données qu'elle contient avec celles de la base de données. C'est le rôle de la méthode merge().

Exemple :

```
package fr.jmdoudoux.dej.jpa;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class TestJPA11 {
  public static void main(String[] argv) {
    EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("MaBaseDeTestPU");
    EntityManager em = emf.createEntityManager();

    EntityTransaction transac = em.getTransaction();
    transac.begin();

    Query query = em.createQuery("select p from Personne p where p.nom='nom2'");
    Personne personne = (Personne) query.getSingleResult();
    if (personne == null) {
      System.out.println("Personne non trouvée");
    } else {
      System.out.println("Personne.prenom=" + personne.getPrenom());

      Personne pers = new Personne();
      pers.setId(personne.getId());
      pers.setNom(personne.getNom());
      pers.setPrenom("prenom2 REmodifie");

      em.merge(pers);

      personne = (Personne) query.getSingleResult();
      System.out.println("Personne.prenom=" + personne.getPrenom());
    }

    transac.commit();

    em.close();
    emf.close();
  }
}
```

La méthode merge() renvoie une instance gérée de l'entité.

63.4.2.8. L'utilisation de la classe EntityManager pour supprimer une occurrence

Pour supprimer une entité existante dans la base de données, il faut :

- Obtenir une instance de l'entité à supprimer (par recherche sur la clé primaire ou l'exécution d'une requête)
- Appeler la méthode remove() de l'EntityManager en lui passant en paramètre l'instance de l'entité

Exemple :

```
package fr.jmdoudoux.dej.jpa;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;

public class TestJPA12 {
  public static void main(String[] argv) {
    EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("MaBaseDeTestPU");
    EntityManager em = emf.createEntityManager();

    EntityTransaction transac = em.getTransaction();
    transac.begin();

    Personne personne = em.find(Personne.class, 4);
    if (personne == null) {
      System.out.println("Personne non trouvée");
    } else {
      em.remove(personne);
    }

    transac.commit();

    em.close();
    emf.close();
  }
}
```

La seule façon d'annuler une suppression est de recréer l'entité en utilisant la méthode persist().

63.4.2.9. L'utilisation de la classe EntityManager pour rafraîchir les données d'une occurrence

La méthode refresh() de l'EntityManager permet de rafraîchir les données de l'entité avec celles contenues dans la base de données.

Exemple :

```
package fr.jmdoudoux.dej.jpa;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class TestJPA13 {
  public static void main(String[] argv) {
    EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("MaBaseDeTestPU");
    EntityManager em = emf.createEntityManager();

    Personne personne = em.find(Personne.class, 4);
    if (personne == null) {
      System.out.println("Personne non trouvée");
    } else {
      em.refresh(personne);
    }

    em.close();
    emf.close();
  }
}
```

La méthode refresh() peut lever une exception de type EntityNotFoundException si l'occurrence correspondante dans la base de données n'existe plus.

63.5. Le fichier persistence.xml

Ce fichier persistence.xml contient la configuration de base pour le mapping notamment en fournissant les informations sur la connexion à la base de données à utiliser.

Le fichier persistence.xml doit être stocké dans le répertoire META-INF

La racine du document XML du fichier persistence.xml est le tag <persistence>.

Il contient un ou plusieurs tags <persistence-unit> qui va contenir les paramètres d'un persistence unit. Ce tag possède deux attributs : name (obligatoire) qui précise le nom de l'unité et qui servira à y faire référence et transaction-type (optionnel) qui précise le type de transaction utilisée (ceci dépend de l'environnement d'exécution : Java SE ou Java EE).

Le tag <persistence-unit> peut avoir les tags fils suivants :

Tag	Rôle
<description>	Fournir une description purement informative de l'unité de persistance(optionnel)
<provider>	Définir le nom pleinement qualifié d'une classe de type javax.persistence. PersistenceProvider (optionnel). Généralement fournie par le fournisseur de l'implémentation de l'API : une utilisation de ce tag n'est requise que pour des besoins spécifiques
<jta-data-source>	Définir le nom JNDI de la DataSource utilisée dans un environnement avec support de JTA (optionnel)
<non-jta-data-source>	Définir le nom JNDI de la DataSource utilisée dans un environnement sans support de JTA (optionnel)

<mapping-file>	Préciser un fichier de mapping supplémentaire (optionnel)
<jar-file>	Préciser un fichier jar qui contient des entités à inclure dans l'unité de persistance : le chemin précisé est relatif au fichier persistence.xml (optionnel)
<class>	Préciser une classe d'une entité qui sera incluse dans l'unité de persistence (optionnel)
<properties>	Définir des paramètres spécifiques au fournisseur. Comme Java SE ne propose pas de serveur JNDI, c'est fréquemment grâce à ce tag que les informations concernant la source de données sont définies (optionnel)
<exclude-unlisted-classes>	Inhiber la recherche automatique des classes des entités (optionnel)

L'ensemble des classes des entités qui compose l'unité de persistance peut être spécifié explicitement dans le fichier persistence.xml ou déterminé dynamiquement à l'exécution par recherche de toutes les classes possédant une annotation @javax.persistence.Entity.

Par défaut, la liste de classes explicite est complétée par la liste des classes issue de la recherche dynamique. Pour empêcher la recherche dynamique, il faut utiliser le tag <exclude-unlisted-classes>. Sous Java SE, il est recommandé de préciser explicitement la liste de classes.

Chaque unité de persistance ne peut être liée qu'à une seule source de données.

Exemple :

```
<?xml version="1.0"  encoding="UTF-8"?>
<persistence  xmlns="http://java.sun.com/xml/ns/persistence"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              version="1.0"  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
              http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
  <persistence-unit  name="MaBaseDeTestPU">
    <provider>oracle.toplink.essentials.PersistenceProvider</provider>
    <class>fr.jmdoudoux.dej.jpa.Adresse</class>
    <class>fr.jmdoudoux.dej.jpa.Personne</class>
    <class>fr.jmdoudoux.dej.jpa.PersonneAdresse</class>
    <class>fr.jmdoudoux.dej.jpa.PersonneAdresse2</class>
    <class>fr.jmdoudoux.dej.jpa.PersonnePK</class>
    <properties>
      <property name="toplink.jdbc.driver"
              value="org.apache.derby.jdbc.EmbeddedDriver"/>
      <property name="toplink.jdbc.url"
              value="jdbc:derby:C:/Program  Files/Java/jdk1.6.0/db/MaBaseDeTest"/>
      <property name="toplink.jdbc.user" value="APP"/>
      <property name="toplink.jdbc.password" value=""/>
      <property name="toplink.logging.level" value="INFO"/>
    </properties>
  </persistence-unit>
</persistence>
```

63.6. La gestion des transactions hors Java EE

Le conteneur Java EE propose un support des transactions grâce à l'API JTA : c'est la façon standard de gérer les transactions par le conteneur.

Hors d'un tel conteneur, par exemple dans une application Java SE, les transactions ne sont pas supportées.

Dans un tel contexte, l'API Java Persistence propose une gestion des transactions grâce à l'interface EntityTransaction.

Cette interface propose plusieurs méthodes :

Méthode	Rôle
void begin()	Débuter la transaction
void commit()	Valider la transaction

void roolback()	Annuler la transaction
boolean isActive()	Déterminer si la transaction est active

Pour obtenir une instance de la transaction, il faut utiliser la méthode getTransaction() de l'EntityManager.

La méthode begin() lève une exception de type IllegalStateException si une transaction est déjà active.

Les méthodes commit() et rollback() lèvent une exception de type IllegalStateException si aucune transaction n'est active.

Exemple :

```
package fr.jmdoudoux.dej.jpa;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;

public class TestJPA12 {
  public static void main(String[] argv) {
    EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("MaBaseDeTestPU");
    EntityManager em = emf.createEntityManager();

    EntityTransaction transac = em.getTransaction();
    transac.begin();

    Personne personne = em.find(Personne.class, 4);
    if (personne == null) {
      System.out.println("Personne non trouvée");
    } else {
      em.remove(personne);
    }

    transac.commit();

    em.close();
    emf.close();
  }
}
```

63.7. La gestion des relations entre tables dans le mapping

Dans le modèle des bases de données relationnelles, les tables peuvent être liées entre elles grâce à des relations.

Ces relations sont transposées dans les liaisons que peuvent avoir les différentes entités correspondantes.

Les relations peuvent avoir différentes cardinalités :

- 1-1 (one-to-one)
- 1-n (one-to-many)
- n-1 (many-to-one)
- n-n (many-to-many)

Chacune de ces relations peut être unidirectionnelle ou bidirectionnelle sauf one-to-many et many-to-one qui sont par définition bidirectionnelles.

La suite de ce chapitre sera développée dans une version future de ce document

63.8. Le mapping de l'héritage de classes

JPA 1.0 propose trois stratégies pour mapper une hiérarchie de classes :

- single table (SINGLE_TABLE) : une seule table est utilisée pour toutes les classes de la hiérarchie
- joined subclass (JOINED) : une table est utilisée pour chaque classe de la hiérarchie
- table per class (TABLE_PER_CLASS) : une table est utilisée pour chaque classe concrète

La hiérarchie de classes peut être composée de différentes typologies de classes abstraites ou concrètes :

- des classes non annotées : les champs ne sont pas persistants
- des classes annotées avec @Entity : les champs sont persistants et il est possible d'effectuer des requêtes
- des classes annotées avec @MappedSuperclass : les champs sont persistants et il n'est pas possible d'effectuer des requêtes

La classe d'une entité, annotée avec @Entity, peut être abstraite ou concrète.

63.8.1. Les annotations

JPA propose plusieurs annotations spécifiques à la définition du mapping d'une hiérarchie de classes.

63.8.1.1. L'annotation @MappedSuperclass

L'inconvénient d'une entité dont la classe est abstraite est qu'elle ne peut pas être instanciée et ne peut donc pas être utilisée comme entité par un EntityManager.

JPA propose l'annotation @MappedSuperclass qu'il est possible d'utiliser sur une classe. Cette classe peut alors être utilisée comme classe mère pour des entités. Les champs d'une classe annotée avec @MappedSuperclass sont persistantes dans la base de donnée mais il n'est pas possible de faire des requêtes sur cette classe. Une classe annotée avec @MappedSuperclass ne sera pas mappée sur une table dédiée.

Cette annotation ne possède pas d'attributs.

Exemple :

```
package fr.jmdoudoux.dej.jpa.entity;

import java.io.Serializable;
import java.util.Date;
import javax.persistence.Column;
import javax.persistence.MappedSuperclass;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@MappedSuperclass
public abstract class Audit implements Serializable {

  private static final long serialVersionUID = 1L;

  @Column(name = "DAT_INS")
  @Temporal(TemporalType.TIMESTAMP)
  private Date dateInsert;
  @Column(name = "USER_INSERT")
  private String userInsert;

  @Column(name = "DAT_UPD")
  @Temporal(TemporalType.TIMESTAMP)
  private Date dateUpdate;

  @Column(name = "USER_UPDATE")
  private String userUpdate;

  public Date getDateInsert() {
    return dateInsert;
  }

  public void setDateInsert(Date dateInsert) {
    this.dateInsert = dateInsert;
  }

  public Date getDateUpdate() {
    return dateUpdate;
  }

  public void setDateUpdate(Date dateUpdate) {
    this.dateUpdate = dateUpdate;
  }

  public String getUserInsert() {
    return userInsert;
  }

  public void setUserInsert(String userInsert) {
    this.userInsert = userInsert;
  }

  public String getUserUpdate() {
    return userUpdate;
  }

  public void setUserUpdate(String userUpdate) {
    this.userUpdate = userUpdate;
  }
}
```

Une entité peut hériter d'une classe annotée avec @MappedSuperclass.

Exemple :

```
package fr.jmdoudoux.dej.jpa.entity;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "compte")
public class Compte extends Audit {
  private static final long serialVersionUID = 1L;

  @Id
  @GeneratedValue
  private Long Id;

  public Long getId() {
    return Id;
  }

  public void setId(Long Id) {
    this.Id = Id;
  }
  // ...
}
```

Certaines informations de mapping dans une classe fille peuvent être redéfinies en utilisant l'annotation @AttributeOverride.

Exemple :

```
package fr.jmdoudoux.dej.jpa.entity;

import javax.persistence.AttributeOverride;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "compte")
@AttributeOverride(name="dateInsert",column=@Column(name="D_CREA"))
public class Compte extends Audit {
  private static final long serialVersionUID = 1L;

  @Id
  @GeneratedValue
  private Long Id;

  public Long getId() {
    return Id;
  }

  public void setId(Long Id) {
    this.Id = Id;
  }

  // ...
}
```

63.8.1.2. L'annotation @Inheritance

L'annotation @Inheritance permet de préciser la stratégie de mapping de la hiérarchie de classes à utiliser. Elle s'utilise sur la classe mère qui est la racine de la hiérarchie de classes.

Elle possède un seul attribut nommé strategy du type de l'énumération javax.persistence.InheritanceType.

Il permet de préciser la stratégie à utiliser grâce à une énumération de javax.persistence.InheritanceType. Elle définit trois valeurs :

- SINGLE_TABLE : une table pour une hiérarchie de classes
- TABLE_PER_CLASS : une table par classe concrète
- JOINED : une table par classe fille

L'attribut strategy est optionnel : par défaut, si celui-ci n'est pas utilisé alors c'est la stratégie SINGLE_TABLE qui est utilisée.

63.8.1.3. L'annotation @DiscriminatorColumn

L'annotation @DiscriminatorColumn permet de définir la colonne qui servira de discriminant dans la table : elle permet de préciser la colonne qui va contenir la valeur utilisée comme discriminant pour savoir à quelle classe correspondent les données. Cette annotation s'utilise pour les stratégies SINGLE_TABLE et JOINED sur la classe mère qui est la racine de la hiérarchie de classes.

L'annotation @DiscriminatorColumn permet de préciser quel champ est le discriminant lors de l'utilisation des stratégies SINGLE_TABLE et JOINED. Elle s'utilise sur la classe mère.

Elle possède plusieurs attributs optionnels :

Attribut	Type	Rôle
columnDefinition	String	Expression SQL du DDL qui permet de définir la colonne. Son utilisation réduit généralement la portabilité de la base de données utilisée (Optionnel). Par défaut, chaîne vide
discriminatorType	javax.persistence.DiscriminatorType	Type de la colonne qui sert de discriminant. La valeur est du type de l'énumération DiscriminatorType (STRING, CHAR, INTEGER) (Optionnel). Par défaut, DiscriminatorType.STRING
length	int	Taille de la colonne si le type est STRING, sinon elle est ignorée (Optionnel). Par défaut, 31
name	String	Nom de la colonne qui sert de discriminant (Optionnel). Par défaut, DTYPE

L'énumération de type DiscriminatorType définit plusieurs valeurs :

- CHAR : un seul caractère
- INTEGER : une valeur numérique
- STRING : une chaîne de caractères

Il est recommandé de préciser explicitement l'attribut name.

Si un discriminant est requis par la stratégie et que l'annotation @DiscriminatorColumn n'est pas utilisée alors une colonne nommée «DTYPE» ayant pour type DiscriminatorType.STRING sera utilisée.

63.8.1.4. L'annotation @DiscriminatorValue

L'annotation @DiscriminatorValue permet de préciser pour chaque classe la valeur qui sera utilisée dans la colonne servant de discriminant. Elle s'utilise sur une classe concrète de la hiérarchie dont la stratégie de mapping a besoin d'un discriminant.

La valeur fournie sous la forme d'une chaîne de caractères doit pouvoir être convertie dans le type de la colonne.

Si l'annotation @DiscriminatorValue n'est pas utilisée dans une stratégie qui le requiert, alors l'implémentation peut utiliser une valeur appropriée selon le type de la colonne comme discriminant pour chaque classe. Par exemple, si le type est STRING alors c'est le nom de la classe qui sera utilisée par défaut.

63.8.2. Des exemples de mises en oeuvre des stratégies

Les exemples de cette section vont mettre en oeuvre les trois stratégies en utilisant MySQL 5.6 comme base de données et Hibernate 4.1 comme implémentation de JPA.

Chaque stratégie utilise la même classe de tests.

Exemple (code Java 5.0) :

```
import javax.persistence.EntityManager;
import javax.persistence.Persistence;

import fr.jmdoudoux.dej.jpa.heritage.entity.Compte;
import fr.jmdoudoux.dej.jpa.heritage.entity.CompteCourant;
import fr.jmdoudoux.dej.jpa.heritage.entity.CompteEpargne;

public class TestJPAHeritage {

  public static void main(String[] args) {

    EntityManager em = Persistence
        .createEntityManagerFactory("TestJPAHeritage").createEntityManager();

    em.getTransaction().begin();

    Compte compte = em.find(Compte.class, 1);
    System.out.println("Compte=" + compte);

    CompteCourant compteCourant = em.find(CompteCourant.class, 2);
    System.out.println("CompteCourant=" + compteCourant);

    CompteEpargne compteEpargne = em.find(CompteEpargne.class, 3);
    System.out.println("CompteEpargne=" + compteEpargne);

    em.getTransaction().commit();
  }
}
```

Le fichier META-INF/persistence.xml contient la définition de la persistence unit.

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.0"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
    <persistence-unit name="TestJPAHeritage"
        transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <class>fr.jmdoudoux.dej.jpa.heritage.entity.Compte</class>
        <class>fr.jmdoudoux.dej.jpa.heritage.entity.CompteCourant</class>
        <class>fr.jmdoudoux.dej.jpa.heritage.entity.CompteEpargne</class>
        <properties>
            <property name="javax.persistence.jdbc.url"
                value="jdbc:mysql://localhost:3307/mabdd"></property>
            <property name="javax.persistence.jdbc.driver"
                value="com.mysql.jdbc.Driver"></property>
            <property name="javax.persistence.jdbc.user" value="root"></property>
            <property name="javax.persistence.jdbc.password" value="root"></property>
        </properties>
    </persistence-unit>
</persistence>
```

63.8.3. La stratégie une table par hiérarchie de classes (SINGLE_TABLE)

La stratégie utilise une seule table pour stocker les données de toutes les instances de la classe mère et de toutes les classes filles : cette table contient des colonnes pour stocker les propriétés mappées de toutes ces classes.

Résultat :

```
CREATE DATABASE  IF NOT EXISTS `mabdd` DEFAULT CHARACTER SET utf8;
USE `mabdd`;
DROP TABLE IF EXISTS `compte`;
CREATE TABLE `compte` (
  `discriminant` varchar(31) NOT NULL,
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `numero` varchar(255) DEFAULT NULL,
  `solde` decimal(19,2) DEFAULT NULL,
  `decouvert` int(11) DEFAULT NULL,
  `taux` decimal(19,2) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8;
```

```

LOCK TABLES `compte` WRITE;
INSERT INTO `compte` VALUES ('Compte',1,'000012345000',0.00,NULL,NULL),
('CompteCourant',2,'000012345010',1200.00,2000,NULL),
('CompteEpargne',3,'000012345020',8000.00,NULL,2.10);
UNLOCK TABLES;
```

Pour mettre en oeuvre cette stratégie, il faut :

- utiliser l'annotation @Inheritance(strategy=InheritanceType.SINGLE_TABLE) sur la classe mère
- utiliser l'annotation @DiscriminatorColumn sur la classe mère pour préciser la colonne qui servira de discriminant
- utiliser l'annotation @DiscriminatorValue sur chaque classe de la hiérarchie pour préciser la valeur dans la colonne discriminant

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.jpa.heritage.entity;

import java.io.Serializable;
import java.math.BigDecimal;
import javax.persistence.Column;
import javax.persistence.DiscriminatorColumn;
import javax.persistence.DiscriminatorType;
import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.Table;

@Entity
@Table(name = "compte")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "discriminant", discriminatorType = DiscriminatorType.STRING)
@DiscriminatorValue(value = "compte")
public class Compte implements Serializable {

  private static final long serialVersionUID = 1L;

  @Id
  @GeneratedValue
  @Column(name = "id")
  protected int             id;

  @Column(name = "numero")
  protected String          numero;

  @Column(name = "solde")
  protected BigDecimal      solde;

  public Compte() {
  }

  // getters et setters

  @Override
  public String toString() {
    return super.toString() + " [id=" + id + ", numero=" + numero + ", solde="
        + solde + "]";
  }
}
```

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.jpa.heritage.entity;

import javax.persistence.Column;
import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@Entity
@DiscriminatorValue(value = "compte_courant")
public class CompteCourant extends Compte {

  private static final long serialVersionUID = 1L;

  @Column(name = "decouvert")
  private int               decouvert;

  public int getDecouvert() {
    return decouvert;
  }

  public void setDecouvert(int decouvert) {
    this.decouvert = decouvert;
  }

  @Override
  public String toString() {
    return this.getClass().getName() + "@" + System.identityHashCode(this)
        + "CompteCourant [id=" + id + ", numero=" + numero + ", solde=" + solde
        + ", decouvert=" + decouvert + "]";
  }
}
```

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.jpa.heritage.entity;

import java.math.BigDecimal;
import javax.persistence.Column;
import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@Entity
@DiscriminatorValue(value = "compte")
public class CompteEpargne extends Compte {

  private static final long serialVersionUID = 1L;

  @Column(name = "taux")
  private BigDecimal        taux;

  public BigDecimal getTaux() {
    return taux;
  }

  public void setTaux(BigDecimal taux) {
    this.taux = taux;
  }

  @Override
  public String toString() {
    return this.getClass().getName() + "@" + System.identityHashCode(this)
        + "CompteEpargne [ id=" + id + ", numero=" + numero + ", solde="
        + solde + ", taux=" + taux + "]";
  }
}
```

Il est possible de mapper la colonne servant de discriminant à une propriété de l'entité : ceci permet d'avoir un accès à sa valeur. Dans ce cas, les attributs insertable et updatable doivent être définis à false dans l'annotation @Column de la propriété.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.hibernate.entity;

import java.math.BigDecimal;
import javax.persistence.Column;
import javax.persistence.DiscriminatorColumn;
import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.Table;

@Entity
@Table(name = "compte")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "discriminant")
@DiscriminatorValue(value = "compte")
public class Compte {
  @Id
  @GeneratedValue
  @Column(name = "id")
  protected int        id;

  @Column(name = "numero")
  protected String     numero;

  @Column(name = "solde")
  protected BigDecimal solde;

  @Column(name = "discriminant", insertable = false, updatable = false)
  protected String     discriminant;

  // getters et setters

  @Override
  public String toString() {
    return super.toString() + " [id=" + id + ", numero=" + numero + ", solde="
        + solde + "]";
  }
}
```

63.8.4. La stratégie : une classe par classe concrète (TABLE_PER_CLASS)

La stratégie Table per concrete class utilise une table pour chaque entité. Le support de cette stratégie par l'implémentation de JPA est optionnel.

Pour l'exemple de cette section, il y a trois tables qui contiennent les données des différentes classes : chaque entité est mappée sur sa propre table. Les tables sont logiquement liées : les champs hérités existent dans chacune des tables.

Résultat :

```
CREATE DATABASE  IF NOT EXISTS `mabdd` DEFAULT CHARACTER SET utf8 ;
USE `mabdd`;
DROP TABLE IF EXISTS `compte`;
CREATE TABLE `compte` (
  `id` int(11) NOT NULL,
  `numero` varchar(255) DEFAULT NULL,
  `solde` decimal(19,2) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
LOCK TABLES `compte` WRITE;
INSERT INTO `compte` VALUES (1,'000012345000',0.00);
UNLOCK TABLES;

DROP TABLE
IF EXISTS `compte_courant`;
CREATE TABLE `compte_courant` (
  `id` int(11) NOT NULL,
  `numero` varchar(255) DEFAULT NULL,
  `solde` decimal(19,2) DEFAULT NULL,
  `decouvert` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

```
LOCK TABLES `compte_courant` WRITE;
INSERT INTO `compte_courant` VALUES (2,'000012345010',1200.00,2000);
UNLOCK TABLES;

DROP TABLE IF EXISTS `compte_epargne`;
CREATE TABLE `compte_epargne` (
  `id` int(11) NOT NULL,
  `numero` varchar(255) DEFAULT NULL,
  `solde` decimal(19,2) DEFAULT NULL,
  `taux` decimal(19,2) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
LOCK TABLES `compte_epargne` WRITE;
INSERT INTO `compte_epargne` VALUES (3,'000012345020',8000.00,2.10);
UNLOCK TABLES;
```

Pour mettre en oeuvre cette stratégie, il faut :

- utiliser l'annotation @Inheritance(strategy=InheritanceType.TABLE_PER_CLASS) sur la classe mère
- il n'est pas possible d'utiliser l'annotation @DiscriminatorColumn

Cette stratégie de mapping limite le choix dans la stratégie de génération des identifiants à utiliser : les identifiants doivent être uniques malgré leur partage sur plusieurs tables. Il n'est donc pas possible d'utiliser les stratégies AUTO et IDENTITY.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.jpa.heritage.entity;

import java.io.Serializable;
import java.math.BigDecimal;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.Table;

@Entity
@Table(name = "compte")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Compte implements Serializable {

  private static final long serialVersionUID = 1L;

  @Id
  @GeneratedValue(strategy = GenerationType.TABLE)
  @Column(name = "id")
  protected int             id;

  @Column(name = "numero")
  protected String          numero;

  @Column(name = "solde")
  protected BigDecimal      solde;

  public Compte() {
  }

  // getters et setters

  @Override
  public String toString() {
    return super.toString() + " [id=" + id + ", numero=" + numero + ", solde="
        + solde + "]";
  }
}
```

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.jpa.heritage.entity;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Table;

@Entity
@Table(name = "compte_courant")
public class CompteCourant extends Compte {

  private static final long serialVersionUID = 1L;

  @Column(name = "decouvert")
  private int               decouvert;

  public int getDecouvert() {
    return decouvert;
  }

  public void setDecouvert(int decouvert) {
    this.decouvert = decouvert;
  }

  @Override
  public String toString() {
    return this.getClass().getName() + "@" + System.identityHashCode(this)
        + "CompteCourant [id=" + id + ", numero=" + numero + ", solde=" + solde
        + ", decouvert=" + decouvert + "]";
  }
}
```

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.jpa.heritage.entity;

import java.math.BigDecimal;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Table;

@Entity
@Table(name = "compte_epargne")
public class CompteEpargne extends Compte {

  private static final long serialVersionUID = 1L;

  @Column(name = "taux")
  private BigDecimal        taux;

  public BigDecimal getTaux() {
    return taux;
  }

  public void setTaux(BigDecimal taux) {
    this.taux = taux;
  }

  @Override
  public String toString() {
    return this.getClass().getName() + "@" + System.identityHashCode(this)
        + "CompteEpargne [ id=" + id + ", numero=" + numero + ", solde="
        + solde + ", taux=" + taux + "]";
  }
}
```

Il est possible d'utiliser l'annotation @AttributeOverride pour redéfinir le mapping des propriétés héritées. L'annotation @AttributeOverrides permet d'utiliser plusieurs annotations @AttributeOverride.

63.8.5. La stratégie une table par sous-classe (JOINED)

La stratégie une table par sous-classe utilise une table par classe fille sur laquelle est effectuée une jointure sur la table de la classe mère pour obtenir toutes les données.

Les tables sont liées entre elles par clés étrangères entre les clés primaires des tables des classes fille et la clé primaire de la table de la classe mère. Les clés primaires des tables des classes filles doivent donc obligatoirement être uniques.

Exemple :

```
CREATE DATABASE  IF NOT EXISTS `mabdd` DEFAULT CHARACTER SET utf8;
USE `mabdd`;
DROP TABLE IF EXISTS `compte`;
CREATE TABLE `compte` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `numero` varchar(255) DEFAULT NULL,
  `solde` decimal(19,2) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8;
LOCK TABLES `compte` WRITE;
INSERT INTO `compte` VALUES (1,'000012345000',0.00),(2,'000012345010',1200.00),
(3,'000012345020',8000.00);
UNLOCK TABLES;

DROP TABLE IF EXISTS `compte_courant`;
CREATE TABLE `compte_courant` (
  `decouvert` int(11) DEFAULT NULL,
  `id` int(11) NOT NULL,
  PRIMARY KEY (`id`),
  KEY `FK_qlp4ap2qk7y2vegbmyrtj2ij2` (`id`),
  CONSTRAINT `FK_qlp4ap2qk7y2vegbmyrtj2ij2`
  FOREIGN KEY (`id`) REFERENCES `compte` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
LOCK TABLES `compte_courant` WRITE;
INSERT INTO `compte_courant` VALUES (2000,2);
UNLOCK TABLES;

DROP TABLE IF EXISTS `compte_epargne`;
CREATE TABLE `compte_epargne` (
  `taux` decimal(19,2) DEFAULT NULL,
  `id` int(11) NOT NULL,
  PRIMARY KEY (`id`),
  KEY `FK_cvpsyxwdfnq90nj88vj93ppvj` (`id`),
  CONSTRAINT `FK_cvpsyxwdfnq90nj88vj93ppvj`
  FOREIGN KEY (`id`) REFERENCES `compte` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
LOCK TABLES `compte_epargne` WRITE;
INSERT INTO `compte_epargne` VALUES (2.10,3);
UNLOCK TABLES;
```

Pour mettre en oeuvre cette stratégie, il faut :

- utiliser l'annotation @Inheritance(strategy=InheritanceType.JOINED) sur la classe mère
- utiliser l'annotation @PrimaryKeyJoinColumn sur chaque classe fille pour préciser la colonne qui est la clé primaire et qui servira de clé étrangère lors de la jointure avec la table de la classe mère

Si l'annotation@PrimaryKeyJoinColumn n'est pas utilisée sur une classe fille alors la colonne de la clé étrangère est considérée comme équivalente à la clé primaire de la table de la classe mère.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.jpa.heritage.entity;

import java.io.Serializable;
import java.math.BigDecimal;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.Table;

@Entity
@Table(name = "compte")
@Inheritance(strategy = InheritanceType.JOINED)
public class Compte implements Serializable {

  private static final long serialVersionUID = 1L;
  @Id
  @GeneratedValue(strategy = GenerationType.TABLE)
  @Column(name = "id")
  protected int             id;

  @Column(name = "numero")
  protected String          numero;

  @Column(name = "solde")
  protected BigDecimal      solde;

  public Compte() {
  }

  // getters et setters

  @Override
  public String toString() {
    return super.toString() + " [id=" + id + ", numero=" + numero + ", solde="
        + solde + "]";
  }
}
```

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.jpa.heritage.entity;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.Table;

@Entity
@Table(name = "compte_courant")
@PrimaryKeyJoinColumn(name = "id")
public class CompteCourant extends Compte {

  private static final long serialVersionUID = 1L;

  @Column(name = "decouvert")
  private int               decouvert;

  public int getDecouvert() {
    return decouvert;
  }

  public void setDecouvert(int decouvert) {
    this.decouvert = decouvert;
  }

  @Override
  public String toString() {
    return this.getClass().getName() + "@" + System.identityHashCode(this)
        + "CompteCourant [id=" + id + ", numero=" + numero + ", solde=" + solde
        + ", decouvert=" + decouvert + "]";
  }
}
```

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.jpa.heritage.entity;
```

```
import java.math.BigDecimal;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.Table;

@Entity
@Table(name = "compte_epargne")
@PrimaryKeyJoinColumn(name = "id")
public class CompteEpargne extends Compte {

  private static final long serialVersionUID = 1L;

  @Column(name = "taux")
  private BigDecimal        taux;

  public BigDecimal getTaux() {
    return taux;
  }

  public void setTaux(BigDecimal taux) {
    this.taux = taux;
  }

  @Override
  public String toString() {
    return this.getClass().getName() + "@" + System.identityHashCode(this)
        + "CompteEpargne [ id=" + id + ", numero=" + numero + ", solde="
        + solde + ", taux=" + taux + "]";
  }
}
```

Il est possible d'utiliser une colonne de type discriminant en utilisant l'annotation @DiscriminatorColumn sur la classe mère pour préciser le nom de cette colonne.

La classe mère peut être abstraite. Dans ce cas, elle doit tout de même être annotée avec @Entity et @Inheritance.

63.9. Les callbacks d'événements

L'API Java Persistence permet de définir des callbacks qui seront appelés sur certains événements. Ces callbacks doivent être annotés avec une des annotations définies par JPA dans le package javax.persistence.

Annotation	Invocation de la méthode de callback annotée
@PrePersist	avant l'ajout d'une nouvelle entity à persiter (ajout à l'EntityManager)
@PostPersist	après avoir persisté une nouvelle entité dans la base de données (commit ou flush)
@PostLoad	après avoir obtenu une entité de la base de données
@PreUpdate	lorsqu'une entité est marquée comme étant modifiée par l'EntityManager
@PostUpdate	après qu'une entité est été mise à jour dans la base de données (commit ou flush)
@PreRemove	lorsqu'une entité est marquée pour suppression
@PostRemove	après d'une entité est été supprimée dans la base de données (commit ou flush)

La classe d'une entité peut inclure des méthodes de callback pour chacun de ses événements du cycle de vie, mais une seule méthode pour le même événement. Une même méthode peut être annotée plusieurs événements en utilisant plusieurs annotations.

Les méthodes de callback dans une classe d'entité peuvent avoir un nom et un modificateur de visibilité libres. Elles ne doivent pas avoir de paramètre et ne rien retourner.

Exemple :

```
@Entity
public class MonEntite {
    @PrePersist
    void onPrePersist() {}

    @PostPersist
    void onPostPersist() {}

    @PostLoad
    void onPostLoad() {}

    @PreUpdate
    void onPreUpdate() {}

    @PostUpdate
    void onPostUpdate() {}

    @PreRemove
    void onPreRemove() {}

    @PostRemove
    void onPostRemove() {}
}
```

Par défaut, une méthode de callback dans une classe mère d'une entité est invoquée pour les instances d'entité de sous-classes, sauf la méthode de callback est redéfinie.

Pour éviter les conflits avec l'action sur la base de données qui a déclenché l'événement du cycle de vie de l'entité (qui peut être en cours), les méthodes de callback ne doivent pas appeler les méthodes de l'EntityManager ou exécuter une Query et ne doivent pas accéder à d'autres entités.

Si une méthode de callback lève une exception, la transaction est marquée pour être annulée avec un rollback.

Il est possible de définir les méthodes de callback dans une classe dédiée qui doit posséder un constructeur par défaut. Dans ce cas, la signature des méthodes doit avoir en paramètre une instance de type Object ou d'un type d'une entité qui sera l'entité à l'origine de l'événement et ne rien retourner

Exemple :

```
public class EntiteListener {
    @PrePersist
    void onPrePersist(Object o) {}

    @PostPersist
    void onPostPersist(Object o) {}

    @PostLoad
    void onPostLoad(Object o) {}

    @PreUpdate
    void onPreUpdate(Object o) {}

    @PostUpdate
    void onPostUpdate(Object o) {}

    @PreRemove
    void onPreRemove(Object o) {}

    @PostRemove
    void onPostRemove(Object o) {}
}
```

L'annotation @EntityListeners permet d'associer la classe listener avec une l'entité.

Exemple :

```
@Entity
@EntityListeners(EntiteListener.class)
public class MonEntite {
  // ...
}
```

Plusieurs classes listener peuvent être associées à l'entité en les passant dans un tableau en attribut de l'annotation @EntityListeners.

La suite de ce chapitre sera développée dans une version future de ce document

Partie 9 : La machine virtuelle Java (JVM)

Cette partie concerne la machine virtuelle Java ou JVM (Java Virtual Machine). La JVM est un des éléments les plus important de la plate-forme Java : une bonne compréhension de son fonctionnement et des concepts qu'elle met en oeuvre est très importante pour obtenir les meilleures performances avec certaines applications.

Cette partie regroupe plusieurs chapitres :

- La JVM (Java Virtual Machine) : ce chapitre détaille les différents éléments et concepts qui sont mis en oeuvre dans la JVM.
- La gestion de la mémoire dans la JVM HotSpot : ce chapitre détaille la gestion de la mémoire dans la JVM HotSpot et notamment les concepts et le paramétrage du ramasse-miettes.
- La JVM HotSpot dans un conteneur Docker : ce chapitre détaille les points d'attention lors de l'utilisation d'une JVM HotSpot dans un conteneur Docker.
- La décompilation et l'obfuscation : ce chapitre présente la décompilation qui permet de transformer du bytecode en code source et l'obfuscation qui est l'opération permettant de limiter cette transformation.
- Programmation orientée aspects (AOP) : ce chapitre présente le concept de l'AOP (Apsect Oriented Programming
- Terracotta : Ce chapitre détaille les possibilités de l'outil open source Terracotta qui permet de mettre en cluster des JVM

64. La JVM (Java Virtual Machine)

Chapitre 64

Niveau : 🟩🟩🟨🟨🟥⬜ Confirmé

La machine virtuelle Java ou JVM (Java Virtual Machine) est un environnement d'exécution pour applications Java.

C'est un des éléments les plus importants de la plate-forme Java. Elle assure l'indépendance du matériel et du système d'exploitation lors de l'exécution des applications Java. Une application Java ne s'exécute pas directement dans le système d'exploitation mais dans une machine virtuelle qui s'exécute dans le système d'exploitation et propose une couche d'abstraction entre l'application Java et ce système.

La machine virtuelle permet notamment :

- l'interprétation du bytecode
- l'interaction avec le système d'exploitation
- la gestion de sa mémoire grâce au ramasse-miettes

Son mode de fonctionnement est relativement similaire à celui d'un ordinateur : elle exécute des instructions qui manipulent différentes zones de mémoire dédiées de la JVM.

Une application Java ne fait pas d'appel direct au système d'exploitation (sauf en cas d'utilisation de JNI) : elle n'utilise que les API qui sont pour une large part écrites en Java sauf quelques-unes qui sont natives. Ceci permet à Java de rendre les applications indépendantes de l'environnement d'exécution.

La machine virtuelle ne connaît pas le langage Java : elle ne connaît que le bytecode qui est issu de la compilation de codes sources écrits en Java.

Les spécifications de la machine virtuelle Java définissent :

- Les concepts du langage Java
- Le format des fichiers .class
- Les fonctionnalités de la JVM
- Le chargement des fichiers .class
- Le bytecode
- La gestion des threads et des accès concurrents
- ...

Les fonctionnalités de la JVM décrites dans les spécifications sont abstraites : elles décrivent les fonctionnalités requises mais ne fournissent aucune implémentation ou algorithme d'implémentation. L'implémentation est à la charge du fournisseur de la JVM. Il existe de nombreuses implémentations de JVM dont les plus connues sont celles de Sun Microsystems/Oracle (HotSpot), IBM (J9), BEA/Oracle (JRockit), Azul (Zing), ...

Le respect strict de ces spécifications par une implémentation de la JVM garantit la portabilité et la bonne exécution du bytecode.

Ces spécifications sont consultables à l'url : https://docs.oracle.com/javase/specs/

Ce chapitre contient plusieurs sections :

- ♦ La mémoire de la JVM
- ♦ Le cycle de vie d'une classe dans la JVM
- ♦ Les ClassLoaders
- ♦ Le bytecode
- ♦ Le compilateur JIT
- ♦ Les paramètres de la JVM HotSpot
- ♦ Les interactions de la machine virtuelle avec des outils externes
- ♦ Service Provider Interface (SPI)
- ♦ Les JVM 32 et 64 bits

64.1. La mémoire de la JVM

Pour faciliter la gestion de la mémoire, Java propose de simplifier la vie des développeurs :

- Il n'est pas possible d'allouer de la mémoire explicitement : c'est la création d'un nouvel objet avec l'opérateur new qui alloue la mémoire requise
- La JVM dispose d'un ramasse-miettes qui se charge de libérer la mémoire des objets inutilisés

La machine virtuelle Java utilise un processus de récupération automatique de la mémoire des objets inutilisés nommé ramasse-miettes (Garbage Collector en anglais). Les objets inutilisés sont les objets qui ne sont référencés par aucun autre objet.

Ceci évite aux développeurs d'avoir à se soucier de cette récupération dans le code mais présente au moins deux inconvénients :

- il n'est pas possible de connaître le moment où la mémoire d'un objet sera libérée
- le ramasse-miettes ne dispense pas le développeur de connaître son mode de fonctionnement et de prendre quelques précautions pour éviter les fuites de mémoires

Le ramasse-miettes est une fonctionnalité de la machine virtuelle qui peut mettre en oeuvre plusieurs algorithmes pour rechercher les objets inutilisés et récupérer automatiquement la mémoire de ces objets. Chaque JVM implémente son propre ramasse-miettes en utilisant un ou plusieurs algorithmes.

Une JVM 32bits utilise un adressage sur 32 bits ce qui lui permet de gérer jusqu'à 4 Go de mémoire.

64.1.1. Le Java Memory Model

Les règles de gestion de la mémoire dans une JVM sont définies dans le JMM (Java Memory Model). Initialement ces règles sont définies dans la Java Specification Langage : elles ont été revues dans la JSR 133

64.1.2. Les différentes zones de la mémoire

Le stockage des données dans la JVM est opéré dans différentes zones réparties en deux grandes catégories :

- Les zones de mémoire dont la durée de vie est égale à celle de la JVM : elles sont créées au lancement de la JVM et sont détruites à son arrêt
- Les zones de mémoire liées à un thread dont la durée de vie est égale à celle du thread concerné

Plusieurs zones de mémoire sont utilisées par la JVM :

- les registres (register) : ces zones de mémoires sont utilisées par la JVM exclusivement lors de l'exécution des instructions du bytecode.
- une ou plusieurs piles (stack)
- un tas (heap)
- une zone de méthodes (method area)

64.1.2.1. La Pile (Stack)

Chaque thread possède sa propre pile qui contient les variables qui ne sont accessibles que par le thread telles que les variables locales, les paramètres, les valeurs de retour de chaque méthode invoquée par le thread.

Seules des données de type primitif et des références à des objets peuvent être stockées dans la pile. La pile ne peut pas contenir d'objets.

La taille d'une pile peut être précisée à la machine virtuelle.

Si la taille d'une pile est trop petite pour les besoins des traitements d'un thread alors une exception de type StackOverflowError est levée.

Si la mémoire de la JVM ne permet pas l'allocation de la pile d'un nouveau thread alors une exception de type OutOfMemoryError est levée.

64.1.2.2. Le tas (Heap)

Cette zone de mémoire est partagée par tous les threads de la JVM : elle stocke toutes les instances des objets créés.

Tous les objets créés sont obligatoirement stockés dans le tas (heap) et sont donc partagés par tous les threads. Comme les tableaux sont des objets en Java, les tableaux sont stockés dans le tas même si ce sont des tableaux de types primitifs.

La libération de cet espace mémoire est effectuée grâce à un mécanisme automatique implémenté dans la JVM : le ramasse-miettes (garbage collector). Le ou les algorithmes utilisés pour l'implémentation du ramasse-miettes sont à la discrétion du fournisseur de la JVM.

La taille du tas peut être fixe ou variable durant l'exécution de la JVM : dans ce dernier cas, une taille initiale est fournie et cette taille peut grossir jusqu'à un maximum défini.

Si la taille du heap ne permet pas le stockage d'un objet en cours de création, alors une exception de type OutOfMemoryError est levée.

64.1.2.3. La zone de mémoire "Method area"

Cette zone de la mémoire, partagée par tous les threads, stocke la définition des classes et interfaces, le code des constructeurs et des méthodes, les constantes, les variables de classe (variables static) ...

Comme pour la pile, seules des données de type primitif ou des références à des objets peuvent être stockées dans cette zone de mémoire. La différence est que cette zone de mémoire est accessible à tous les threads. Il est donc important dans un contexte multithread de sécuriser l'accès à une variable static même si elle est de type primitif.

64.1.2.4. La zone de mémoire "Code Cache"

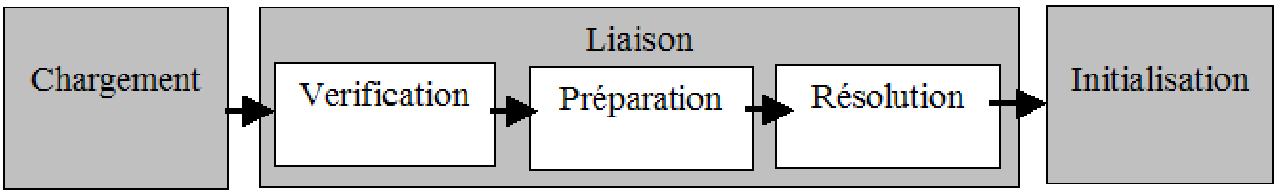
Cette zone de la mémoire stocke le résultat compilé du code des méthodes. La taille par défaut est généralement de 32Mo.

64.2. Le cycle de vie d'une classe dans la JVM

Une classe ou une interface suit un cycle de vie particulier dans la machine virtuelle de son chargement à son retrait.

1. chargement (loading)

2. liaison (linking)
3. initialisation (initialization)
4. instanciation (instantiation)
5. récupération de la mémoire (garbage collection)
6. finalisation (finalization)
7. déchargement (unloading)



Chaque étape est dédiée à une tâche spécifique :

- chargement (load) : permet de lire le bytecode dans la machine virtuelle

- liaison (link) : permet de rendre utilisable le bytecode. Cette étape est composée de trois processus (vérification, préparation, résolution)

    - La vérification (verify) permet de s'assurer que le bytecode est compatible avec la machine virtuelle
    - La préparation (prepare) effectue l'allocation mémoire nécessaire à la classe
    - La résolution (resolve) transforme les références symboliques du constant pool en références mémoire.

- Initialisation (initialize) : initialisation des valeurs des variables.

64.2.1. Le chargement des classes

La machine virtuelle charge, lie et initialise les classes et interfaces requises à l'exécution.

Le démarrage d'une application commence par le chargement de sa classe principale (celle fournie en paramètre de la JVM)

Toutes les classes utilisées pour l'instanciation de cette classe et celles utilisées dans sa méthode main() sont chargées à leur première utilisation.

Un classloader est un objet qui charge dynamiquement et initialise des classes et interfaces Java requises par la JVM lors de l'exécution d'une application. Un classloader hérite de la classe java.lang.ClassLoader.

Un classloader effectue généralement plusieurs opérations pour charger une classe :

- Vérifie si la classe est déjà chargée et initialisée
- Tentative de chargement du bytecode
- Si le chargement réussit, initialisation du bytecode dans la JVM

Le chargement des classes s'effectue en respectant un modèle de délégation de la responsabilité du chargement. Chaque classloader doit déléguer le chargement de la classe à son classloader père : si ce dernier ne peut mener à bien l'opération alors c'est le classloader lui-même qui tente le chargement.

La méthode loadClass() de la classe ClassLoader exécute par défaut les traitements suivants :

- si la classe est déjà chargée alors elle la renvoie
- sinon délégation du chargement au classloader père
- si la délégation du chargement échoue alors la méthode findClass() est invoquée pour tenter de charger la classe

C'est pour cette raison qu'il n'est pas recommandé lors de la création d'un classloader de redéfinir la méthode loadClass() mais de redéfinir la méthode findClass().

64.2.1.1. La recherche des fichiers .class

La JVM recherche et charge les classes requises dans un ordre bien précis grâce à la délégation des classloaders :

- Les classes de bootstrap (bootstrap classes) qui sont les classes fournies avec la plate-forme Java SE dans le fichier rt.jar
- Les classes d'extension (extension classes) qui sont packagées sous forme de fichiers .jar et stockées dans le répertoire lib/ext du JRE
- Les classes d'utilisateurs (user classes) qui sont écrites par les développeurs ou des tiers

Les classes de bootstrap et d'extension n'ont pas besoin d'être précisées explicitement : elles sont trouvées automatiquement. Les autres classes doivent être précisées en utilisant le classpath.

Les classes utilitaires contenues dans le fichier tools.jar doivent être ajoutées explicitement dans le classpath pour pouvoir être utilisées.

Les classes de bootstrap sont les classes fournies avec la plate-forme Java. Elles sont principalement dans le fichier rt.jar mais aussi dans quelques fichiers .jar stockés dans le répertoire lib du JRE. L'ensemble des classes de bootstrap est précisé dans la propriété sun.boot.class.path de la JVM.

Même si cela n'est pas recommandé, il est possible de modifier la propriété sun.boot.class.path en utilisant l'option non standard -Xbootclasspath pour définir sa valeur ou ajouter des éléments en début ou en fin de liste.

Le support des classes d'extension a été ajouté dans Java 1.2. Ces bibliothèques permettent d'enrichir les API de base de Java : il faut donc utiliser ce mécanisme de façon judicieuse.

Les classes d'extension sont des extensions de la plate-forme Java qui sont stockées dans le répertoire lib/ext du JRE. Seules les bibliothèques (.jar ou .zip) sont prises en compte. Il n'est pas possible de préciser ou modifier ce chemin. L'ordre de chargement d'une classe contenue dans plusieurs bibliothèques de ce répertoire n'est pas prévisible.

A partir de Java 1.6, il est possible d'utiliser la variable d'environnement java.ext.dirs pour préciser un ou plusieurs répertoires qui permettront le stockage des extensions. Ceci permet d'utiliser ces répertoires par plusieurs JDK sans être obligé de dupliquer les fichiers .jar dans chaque sous-répertoire lib/ext de chaque JRE.

Les classes d'utilisateurs sont écrites en reposant sur les classes de bootstrap et d'extension. Pour les trouver, la JVM utilise le classpath qui contient un ensemble de répertoires, et de bibliothèques contenant des classes sous la forme de fichiers .jar et/ou .zip.

Il faut mettre dans le classpath l'entité (répertoire ou bibliothèque) qui contient la classe pleinement qualifiée à utiliser.

Exemple :

- Si une classe fr.jmdoudouxfr.dej.MaClasse est stockée dans le répertoire monapp/classes alors le répertoire monapp/classes doit être ajouté au classpath pour utiliser la classe
- Si une classe fr.jmdoudouxfr.dej.MaClasse est stockée dans la bibliothèque monapp.jar alors le fichier monapp.jar doit être ajouté au classpath.

Le séparateur des différents éléments du classpath dépend de la plate-forme d'exécution (; sous Windows et : sous Unix).

Le classpath peut être obtenu grâce à la variable d'environnement java.class.path de la JVM.

Le classpath peut être précisé de plusieurs façons :

- Par défaut, il ne contient que le répertoire courant « . »
- La variable d'environnement système CLASSPATH
- L'option -cp ou -classpath des outils en ligne de commande
- L'option -jar qui précise une bibliothèque : dans ce cas c'est cette dernière que doit préciser le classpath

64.2.1.2. Le chargement du bytecode

La JVM demande au classloader de rechercher et charger le bytecode d'une classe uniquement à sa première utilisation.

Le processus de chargement est composé de trois étapes :

- ouverture d'un flux pour la lecture du bytecode
- analyse du bytecode et création de données dans la zone de méthode
- création d'une instance de la classe java.lang.class pour la classe

La source du flux n'est pas imposée et peut être un fichier .class local, un fichier .class sur le réseau, une archive (jar ou zip), une génération à la volée, ...

L'instance de la classe Class créée permet une interaction entre une application et la représentation interne de la classe : elle permet par exemple d'obtenir des informations sur la classe.

64.2.2. La liaison de la classe

La liaison de la classe comporte trois étapes :

- La vérification
- La préparation
- La résolution

La vérification est la première étape du processus de liaison : elle permet de s'assurer que la classe chargée est conforme aux spécifications et qu'elle ne risque pas de dégrader la machine virtuelle. La vérification consiste donc en une analyse de la structure et des informations de la classe. Par exemple, pour un fichier .class : vérifier qu'il commence par le nombre magique CAFEBABE, la longueur du fichier, la structure des données, ...

Les spécifications de la JVM détaillent une liste d'exceptions et d'erreurs qui doivent être levées lors de cette étape.

La vérification effectue de nombreux contrôles sur le bytecode tels que :

- vérifie les instructions (utilisation d'instructions valides, véracité des sauts, ...)
- vérifie les déclarations d'entités du constant pool (numéro de classes, de méthodes, de champs, ...)
- recherche les classes mères et vérifie que toutes les classes héritent de la classe Object (sauf la classe Object elle-même).
- vérifie que les classes final n'ont pas de classes fille
- vérifie que les méthodes final ne sont pas réécrites
- vérifie que les méthodes des interfaces implémentées soient définies
- vérifie que deux méthodes n'ont pas la même signature
- ...

Certains de ces contrôles nécessitent des informations sur les classes parentes ou sur d'autres classes utilisées qui seront alors chargées mais pas initialisées.

Tous ces contrôles peuvent paraître redondants avec ceux effectués par le compilateur lors de la génération du bytecode mais en fait, il est tout à fait possible que le bytecode ait été altéré, généré à la volée ou que le compilateur présente un ou plusieurs bugs.

Un mécanisme, déjà utilisé depuis longtemps par Java ME, permet d'ajouter des informations de prévérification lors de la compilation. Ainsi l'étape de validation du bytecode est plus rapide à s'exécuter. Depuis la version 6 de Java, le compilateur Java inclut une étape de prévérification qui ajoute des informations dans le fichier.class (StackMap et StackMapTable).

Durant l'étape de préparation, la machine virtuelle alloue la mémoire requise par chaque champs et initialise leurs valeurs avec la valeur par défaut de leur type respectif.

int	0

long	0l
short	0
char	'\u0000'
byte	(byte) 0
float	0.0f
double	0.0d
object	null
boolean (int)	false (0)

Cette étape n'exécute aucun code Java : les valeurs de chaque champ ne sont déterminées que lors de la phase d'initialisation.

Remarque : la machine virtuelle ne définit pas le type booléen. Elle utilise le type int pour sa représentation interne et initialise donc sa valeur à 0 qui correspond à false.

L'étape de résolution permet de rechercher les classes, les interfaces et les membres possédant une référence symbolique dans le constant pool. La résolution permet de remplacer ces références symboliques par des références concrètes.

64.2.3. L'initialisation de la classe

Ce processus a pour rôle d'initialiser les variables de classe avec leurs valeurs initiales telles que définies dans le code source. La valeur initiale peut être définie de deux façons :

- lors de la déclaration du champ static
- dans un bloc d'initialisation static

Exemple :

```
public class MaClasse {
  static List maListe1 = new ArrayList() ;
  static List maListe2 = null;

  static {
    maListe2 = new ArrayList();
  }
}
```

Ces traitements d'initialisation sont regroupés par le compilateur dans une méthode nommée <clinit (class initialization method). Ils ne concernent que l'exécution de code Java : l'initialisation à l'aide de constantes n'est pas reprise dans cette méthode.

Cette méthode ne peut être invoquée que par la machine virtuelle. Les traitements d'initialisation contenus dans la méthode se font dans l'ordre du code source.

L'initialisation d'une classe implique au préalable l'initialisation de sa classe mère si cela n'a pas déjà été fait et ainsi de suite jusqu'à la classe Object : ainsi toutes les classes mères sont initialisées avant la classe elle-même.

Une classe n'a pas obligatoirement de méthode <clinit() : si la classe ne contient aucune variable de classe ou que toutes ses variables sont déclarées finales avec une valeur constante, elle ne possédera pas de méthode <clinit()

Les spécifications de la JVM imposent que l'initialisation d'une classe intervienne à sa première utilisation active :

- création d'une nouvelle instance en utilisant l'opérateur new
- création d'un tableau du type de la classe

- utilisation d'un membre de la classe qui ne soit pas hérité ni ne soit une constante
- utilisation d'une de ses sous-classes (l'initialisation d'une classe impose l'initialisation de toutes ses super-classes).

64.2.4. Le chargement des classes et la police de sécurité

L'utilisation d'un classloader implique la mise en oeuvre de la police de sécurité qui lui est associée.

Le simple fait d'utiliser une classe provoque son chargement à sa première utilisation mais il est possible de demander explicitement le chargement d'une classe en invoquant la méthode loadClass() du classloader d'un objet.

Sans police de sécurité, toutes les classes sont considérées comme sûres par défaut.

Même avec une police de sécurité, les classes du bootstrap sont toujours considérées comme sûres.

La police de sécurité repose sur la configuration de la police de sécurité globale et celle de l'application. Par défaut dans la police de sécurité globale, les classes d'extension sont toujours sûres et les autres classes possèdent quelques restrictions.

64.3. Les ClassLoaders

Le contrôle sur le chargement d'une classe permet notamment de mettre en oeuvre certaines techniques avancées telles que la modification du bytecode, son instrumentation, son cryptage, ...

Les classloaders étant responsables du chargement d'une classe et comme un ClassLoader est une classe, il existe un classloader particulier, le classloader de bootstrap, qui est implémenté en code natif et qui charge les classes de base de Java dont la classe ClassLoader.

Un autre classloader est dédié au chargement des classes d'extensions (celles des bibliothèques stockées dans le sous-répertoire lib/ext du JRE ou à partir de Java 6 celles définies par la propriété java.ext.dirs qui par défaut pointe sur le sous-répertoire lib/ext du JRE).

Le troisième classloader créé automatiquement est celui qui permet de charger les autres classes en particulier celles définies dans le classpath : il se nomme classloader d'application. Il permet le chargement des classes définies dans la propriété java.class.path qui par défaut correspond à la variable d'environnement système CLASSPATH.

Les classloaders ont une organisation hiérarchique permettant la mise en oeuvre d'un mécanisme de délégation du chargement d'une classe : un classloader demande toujours à son classloader père d'essayer de charger la classe.

Le mécanisme de délégation permet de s'assurer qu'une classe sera chargée par le classloader qui lui est dédié :

- Les classes de bootstrap sont toujours chargées par le classloader de bootstrap
- Si la classe n'est pas chargée par le classloader de bootstrap, alors le classloader d'extension tente de charger la classe
- Si la classe n'est pas chargée par le classloader d'extension alors le classloader d'application tente de charger la classe
- Si la classe n'est pas chargée par le classloader d'application et qu'aucun classloader dédié n'est définit alors une exception de type ClassNotFoundException est levée

Une classe est associée au classloader qui l'a chargée. Une fois une classe chargée, celle-ci est identifiée par son nom et son classloader. Ainsi, deux classes de même nom chargées par deux classloaders différents sont considérées comme différentes par la JVM.

Si une classe C1 utilise une classe C2 qui n'est pas encore chargée, alors le classloader par défaut pour charger C2 sera celui de C1. Ainsi une classe de bootstrap ne peut pas utiliser une classe du classpath sauf si c'est le classloader system ou un classloader dédié qui est utilisé.

Un thread est associé à un classloader : pour obtenir une référence sur ce classloader, il faut utiliser la méthode getContextClassLoader(). C'est en général le classloader de la classe qui a démarré le thread. Il est parfois nécessaire d'utiliser le classloader du thread notamment avec les servlets qui sont généralement chargées par un classloader dédié du conteneur web. Ce classloader ne permet généralement que de charger des classes contenues dans l'application web, ce qui lui interdit le chargement des classes du classpath.

64.3.1. Le mode de fonctionnement d'un ClassLoader

Dans une JVM, il existe deux classloaders par défaut :

- Le classloader de bootstrap utilisé uniquement par la JVM
- Le classloader système utilisé pour charger les autres classes

Il est aussi possible de définir son propre classloader.

Le classloader possède une méthode loadClass() qui permet de charger une classe à partir de son nom de fichier binaire. Le nom de binaire de la classe correspond au nom pleinement qualifié de la classe incluant le signe $ et l'incrémentation pour les classes anonymes.

Exemple :

java.lang.String
fr.jmdoudoux.dej.monapp.MonApp
fr.jmdoudoux.dej.monapp.MonApp$1

La méthode loadClass() lève une exception de type ClassNotFoundException si la classe n'est pas trouvée.

Il y a plusieurs façons pour forcer le chargement d'une classe par un classloader :

- Utiliser explicitement la méthode loadClass() du classloader qui peut lever une exception de type ClassNotFoundException

Exemple :

```
package fr.jmdoudoux.dej.classloader;

public class TestClassLoaderla {

  public static void main(String[] args) {
    try {
      System.out.println(
          TestClassLoader1a.class.getClassLoader().loadClass("java.lang.Number"));
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
    }

  }

}
```

- Utiliser la méthode statique Class.forName() qui peut lever une exception de type ClassNotFoundException

Exemple :

```
package fr.jmdoudoux.dej.classloader;

public class TestClassLoader1b {

  public static void main(String[] args) {
    try {
      System.out.println(Class.forName("java.lang.Number"));
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
    }
  }

}
```

- Utiliser la notation .class qui peut lever une exception de type ClassNotFoundError

Exemple :

```
package fr.jmdoudoux.dej.classloader;

public class TestClassLoader1c {

  public static void main(String[] args) {
    System.out.println(Number.class);
  }

}
```

Chaque objet chargé par un classloader conserve une référence sur ce dernier : la méthode getClassLoader() de la classe Class permet d'obtenir cette référence.

Seule la JVM peut utiliser le classloader de bootstrap : ainsi l'appel de la méthode getClassLoader() d'une classe chargée par le classloader de bootstrap renvoie null.

La classe ClassLoader propose la méthode statique getSystemClassLoader() pour obtenir le classloader système. Sauf création d'un classloader dédié, c'est ce classloader qui charge les classes utilisateurs.

Exemple :

```
package fr.jmdoudoux.dej.classloader;

public class TestClassLoader4 {

  public static void main(String[] args) {
    System.out.println(String.class.getClassLoader());
    System.out.println(ClassLoader.getSystemClassLoader());
    System.out.println(TestClassLoader4.class.getClassLoader());
  }

}
```

Résultat :

```
null
sun.misc.Launcher$AppClassLoader@11b86e7
sun.misc.Launcher$AppClassLoader@11b86e7
```

La classe URLClassLoader est un classloader qui charge des classes à partir d'une ou plusieurs URL fournies en paramètre. Ces urls peuvent correspondre à des répertoires ou à des fichiers jar.

Exemple : le fichier c:\java\test.jar contient la classe fr.jmdoudoux.dej.MaClasse

```
package fr.jmdoudoux.dej.classloader;

import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLClassLoader;

public class TestClassLoader5 {

  public static void main(String[] args) {
    try {
      URLClassLoader loader = new URLClassLoader(new URL[] {
        new URL("file:///C:/java/test.jar") });
      Class<?> maClasseClass = loader.loadClass("fr.jmdoudoux.dej.MaClasse");
      System.out.println(maClasseClass);
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
    }
  }
}
```

Les classloaders assurent qu'une même classe n'est chargée qu'une seule fois par une même hiérarchie de classloaders.

Exemple :

```
package fr.jmdoudoux.dej.classloader;

public class TestClassLoader3 {

  public static void main(String[] args) {

    try {
      boolean resultat = (String.class == Class.forName("java.lang.String"));
      System.out.println("comparaison = "+resultat);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
comparaison = true
```

Le classloader permet aussi de charger des ressources grâce à plusieurs méthodes :

Méthode	Rôle
URL getResource(String name)	Renvoie l'url d'une ressource trouvée par le classloader
InputStream getResourceAsStream(String name)	Renvoie un flux pour lire la ressource
URL getSystemResource(String name)	Méthode statique utilisant le classloader système qui renvoie l'url d'une ressource trouvée
InputStream getSystemResourceAsStream(String name)	Méthode statique utilisant le classloader système qui renvoie un flux pour lire la ressource

Le chargement des ressources en utilisant le classloader est obligatoire par exemple pour charger une ressource incluse dans un fichier .jar.

L'option -verbose:class de la JVM permet de demander l'affichage d'informations sur le chargement des classes.

Résultat :

```
...
[Loaded java.lang.StrictMath from shared objects file]
[Loaded sun.security.provider.NativePRNG from shared objects file]
[Loaded sun.misc.CharacterDecoder from shared objects file]
[Loaded sun.misc.BASE64Decoder from shared objects file]
[Loaded sun.security.util.SignatureFileVerifier from shared objects file]
[Loaded fr.jmdoudoux.dej.MaClasse from file:/C:/java/test.jar]
...
```

Cette option peut permettre de déterminer à partir de quelle source une classe est chargée.

64.3.2. La délégation du chargement d'une classe

Il existe une hiérarchie dans les classloaders ce qui permet à un classloader de déléguer le chargement d'une classe à son classloader père. La méthode getParent() de la classe ClassLoader permet de connaître le classloader père. Le classloader de bootstrap ne possède pas de père.

Cette délégation permet notamment de s'assurer que les classes de bootstrap sont chargées par la classeloader de bootstrap. La délégation est effectuée dans la méthode loadClass() qui devrait toujours demander au classloader père de charger la classe.

Exemple :

```
package fr.jmdoudoux.dej.classloader;

import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLClassLoader;

public class TestClassLoader6 {

  public static void main(String[] args) {

    URLClassLoader loader;
    try {
      loader = new URLClassLoader(new URL[] {
        new URL("file:///C:/Program Files/Java/jre1.6.0_03/lib/rt.jar") });
      Class<?> stringClass = loader.loadClass("java.lang.String");
      System.out.println(stringClass.getClassLoader());
      System.out.println(String.class.getClassLoader());
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
null
null
```

Bien que le chargement de la classe String soit demandé par une instance de la classe URLClassLoader, la classe est chargée par le classloader de bootstrap. Comme les deux demandes de chargement sont réalisées par le même classloader, les deux classes sont identiques.

Une classe est liée à son classloader : une même classe chargée par deux classloaders sera chargée deux fois (il y aura deux instances de la classe Class correspondante)

Exemple :

```
package fr.jmdoudoux.dej.classloader;

import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLClassLoader;

public class TestClassLoader7 {

  public static void main(String[] args) {
    try {
      URLClassLoader loader1 = new URLClassLoader(new URL[] {
        new URL("file:///C:/java/test.jar") });
      Class<?> maClasseClass1 = loader1.loadClass("fr.jmdoudoux.dej.MaClasse");
      System.out.println(maClasseClass1);
      URLClassLoader loader2 = new URLClassLoader(
        new URL[] { new URL("file:///C:/java/test.jar") });
      Class<?> maClasseClass2 = loader2.loadClass("fr.jmdoudoux.dej.MaClasse");
      System.out.println(maClasseClass2);
      System.out.println(maClasseClass1 == maClasseClass2);
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
class fr.jmdoudoux.dej.MaClasse
class fr.jmdoudoux.dej.MaClasse
false
```

64.3.2.1. L'écriture d'un classloader personnalisé

L'utilisation d'un classloader dédié peut avoir plusieurs utilités par exemple :

- Personnaliser le chargement du bytecode (par exemple : en l'encryptant après la compilation et en le décryptant au chargement)
- Permettre le rechargement des classes
- Permettre une séparation des classes de plusieurs applications dans une même JVM (exemple avec le conteneur web)
- Enrichir le bytecode pour ajouter des fonctionnalités
- Générer du bytecode à la volée
- ...

Les conteneurs web et les serveurs d'applications sont de bons exemples d'applications qui utilisent des classloaders personnalisés. Généralement, chaque application déployée possède son propre classloader ce qui permet une meilleure isolation des applications exécutées dans la JVM.

Ceci est vrai parce qu'une classe chargée dans la JVM est identifiée par son nom et son classloader. Ceci permet par exemple à un singleton utilisé par plusieurs applications d'être unique par application et non unique dans la JVM puisque chaque application possède son propre classloader.

Ceci permet aussi un rechargement des classes d'une application déployée sans être obligé de relancer la JVM.

L'écriture d'un classloader personnalisé peut permettre de modifier le bytecode : une fois le bytecode chargé le classloader peut le modifier avant de demander son initialisation par la JVM.

Un classloader doit hériter de la classe java.lang.ClassLoader.

La classe ClassLoader possède plusieurs méthodes :

Méthode	Rôle
loadClass()	charger une classe en demandant au préalable au classloader père de réaliser l'opération
findClass()	charger une classe
defineClass()	Ajouter le bytecode de la classe dans la machine virtuelle

Lors de la création de son propre classloader, il faut redéfinir la méthode findClass() plutôt que la méthode loadClass() pour respecter le mécanisme de délégation de chargement de classes des classloaders.

La méthode findClass() ne doit donc être invoquée que si la classe n'a pas pu être chargée par un des classloaders pères.

Les données binaires issues de la lecture du fichier et éventuellement enrichies sont passées en paramètres de la méthode defineClass().

Pour utiliser un classloader personnalisé, il faut explicitement demander son utilisation :

- soit en passant en paramètre de la classe méthode forName() de la classe Class une instance du classloader à utiliser
- soit en utilisant la méthode loadClass() du classloader

Comme par défaut le mécanisme de délégation du chargement d'une classe demande au classloader parent de tenter de charger la classe, il faut être sûr que la classe à charger ne se trouve pas dans un classpath particulier (bootstrap classpath, extension classpath, system classpath). Sinon la classe ne sera pas chargée par le classloader personnalisé mais par un de ses classloaders pères.

Cette fonctionnalité est utilisée par les conteneurs web qui encouragent l'utilisation du sous-répertoire WEB-INF/classes plutôt que de mettre les bibliothèques dans le classpath.

64.4. Le bytecode

Le bytecode est un langage intermédiaire entre le code source et le code machine qui permet de rendre l'exécution d'applications Java multiplate-forme puisque le bytecode est un langage intermédiaire indépendant de tout système d'exploitation.

La JVM fournit un environnement d'exécution pour le bytecode en le convertissant en code machine du système d'exploitation utilisé.

Le bytecode peut être modifié avant son exécution par un classloader dédié. Cette modification ou génération de bytecode est par exemple utilisée par :

- Hibernate qui se sert de la génération de bytecode à l'excution pour produire des proxies pour les classes de persistance
- Certaines implémentations d'AOP qui tissent leurs aspects en enrichissant le bytecode.

La génération directe de bytecode est plus efficace que la génération de code source puisqu'elle évite l'étape de compilation mais elle est aussi de fait plus compliquée.

Le bytecode est défini dans les spécifications de la machine virtuelle Java.

Le bytecode est composé de mnémoniques qui réalisent des opérations sur éventuellement un ou plusieurs opérandes. A chaque mnémonique correspond un opcode.

Le compilateur transforme le code source Java en fichiers .class contenant entre autres le bytecode.

Lors de la compilation du code source en bytecode, le compilateur effectue de nombreuses vérifications notamment sur la syntaxe du code source pour garantir que le bytecode produit est valide et qu'il ne risque pas de nuire à la JVM qui va l'exécuter.

Lors du chargement d'un fichier .class, le classloader effectue des vérifications sur le contenu du fichier afin de s'assurer qu'il ne soit pas en mesure de mettre à mal l'intégrité de la machine virtuelle :

- les 4 premiers octets doivent contenir le chiffre magique (la valeur hexadécimale est "CAFEBABE") qui identifie le fichier comme étant un fichier .class
- chaque classe déclarée final n'est pas sous-classée
- tous les attributs et méthodes contiennent une référence dans le pool de constantes (constants pool)
- ...

D'autres langages peuvent être utilisés avec un compilateur dédié pour créer du bytecode par exemple :

- Groovy : https://groovy-lang.org/
- Scala : http://www.scala-lang.org/
- JRuby : https://www.jruby.org/

- Jython : https://www.jython.org/
- Closure : https://clojure.org/
- Ceylon : https://ceylon-lang.org/
- Kotlin : https://kotlinlang.org/
- JBasic : https://sourceforge.net/projects/jbasic
- Nice : http://nice.sourceforge.net/

64.4.1. La version du bytecode

Lorsque le compilateur crée un fichier .class, il incorpore la version du bytecode qui est une représentation numérique de la valeur fournir avec les options -target ou --release. Si elles ne sont pas précisées, c'est la version du JDK qui est utilisée par défaut.

Les versions de bytecode sont calculées avec une simple formule : 44 + la version de Java précisée avec -target ou --release.

Version du JDK	Version du bytecode	Version du JDK	Version du bytecode	Version du JDK	Version du bytecode
Java 1.0	45.0	Java 10	54.0	Java 20	64.0
Java 1.1	45.3	Java 11	55.0	Java 21	65.0
Java 1.2	46.0	Java 12	56.0	Java 22	66.0
Java 1.3	47.0	Java 13	57.0		
Java 1.4	48.0	Java 14	58.0		
Java 5	49.0	Java 15	59.0		
Java 6	50.0	Java 16	60.0		
Java 7	51.0	Java 17	61.0		
Java 8	52.0	Java 18	62.0		
Java 9	53.0	Java 19	63.0		

Lorsque la JVM charge une classe, elle vérifie si elle supporte la version du bytecode et si ce n'est pas le cas, elle lève une exception de type Error.

Jusqu'à Java 7, une exception de type UnsupportedClassVersionError est levée par la JVM si la version du bytecode est incompatible avec la version de la JVM utilisée

Résultat :

```
C:\java>java -version
openjdk version "1.7.0_75"
OpenJDK Runtime Environment (build 1.7.0_75-b13)
OpenJDK Client VM (build 24.75-b04, mixed mode)

C:\java>java Hello
Exception in thread "main" java.lang.UnsupportedClassVersionError: Hello : Unsupported
major.minor version 63.0
```

A partir de Java 8, le message de l'exception de type UnsupportedClassVersionError est plus précis.

Résultat :

```
C:\java>java -version
openjdk version "1.8.0_252"
OpenJDK Runtime Environment (AdoptOpenJDK)(build 1.8.0_252-b09)
OpenJDK 64-Bit Server VM (AdoptOpenJDK)(build 25.252-b09, mixed mode)

C:\java>java Hello
Error: A JNI error has occurred, please check your installation and try again
Exception in thread "main" java.lang.UnsupportedClassVersionError: Hello has been
 compiled by a more recent version of the Java Runtime (class file version 63.0), this version
 of the Java Runtime only recognizes class file versions up to 52.0
```

A partir de Java 9, une LinkageError est levée, chaînée avec l'exception de type UnsupportedClassVersionError. La stacktracce n'est plus affichée car elle n'apportait rien d'utile.

Résultat :

```
C:\java>java -version
java version "9.0.1"
Java(TM) SE Runtime Environment (build 9.0.1+11)
Java HotSpot(TM) 64-Bit Server VM (build 9.0.1+11, mixed mode)

C:\java>java Hello
Erreur : LinkageError lors du chargement de la classe principale Hello
        java.lang.UnsupportedClassVersionError: Hello has been compiled by a more recent
 version of the Java Runtime (class file version 63.0), this version of the Java Runtime
 only recognizes class file versions up to 53.0
```

64.4.2. L'outil Jclasslib bytecode viewer

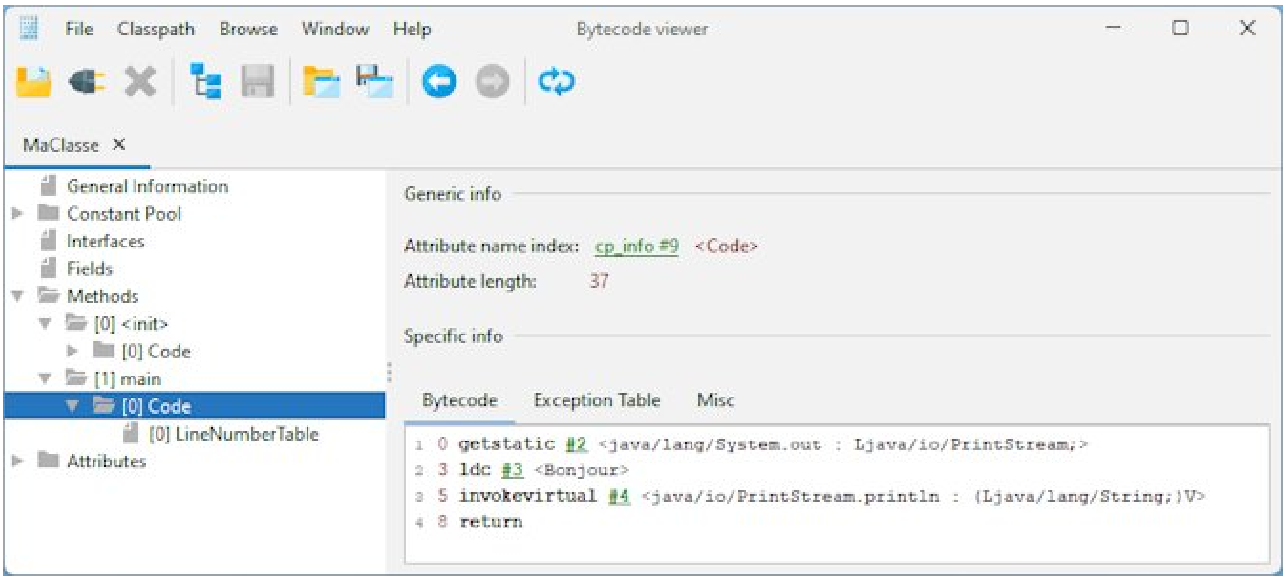
Jclasslib est un outil graphique gratuit qui permet de visualiser le bytecode contenu dans un fichier .class.

Il peut être téléchargé à l'url https://github.com/ingokegel/jclasslib

L'installation se fait sous Windows avec l'aide d'un assistant en exécutant le fichier jclasslib_win64_6_0_4.exe. Il suffit d'exécuter l'outil et d'utiliser l'option « File/Open Class File » en sélectionnant le fichier .class.

Exemple :

```
public class MaClasse {

  public static void main(String[] args) {
    System.out.println("Bonjour");
  }
}
```



Attribute name index: cp_info #9 <Code>

Attribute length: 37

```
0 getstatic #2 <java/lang/System.out : Ljava/io/PrintStream;>
3 ldc #3 <Bonjour>
5 invokevirtual #4 <java/io/PrintStream.println : (Ljava/lang/String;)V>
8 return
```

La partie de gauche affiche une vue hiérarchique de la structure du fichier. La partie de droite affiche le contenu de l'élément sélectionné dans la partie de gauche

A partir de la version 6.0, de nombreux aspects des fichiers .class peuvent être modifiés dans l'interface utilisateur.

64.4.3. Le jeu d'instructions de la JVM

La JVM possède un ensemble d'instructions qui sont utilisées pour définir des traitements. Le code source représentant la logique des traitements est compilé pour générer un fichier binaire .class.

Les instructions de la JVM sont des opérations basiques qui combinées permettent de réaliser les traitements.

Une instruction est composée d'un code opération (opcode) suivi d'aucun, un ou plusieurs opérandes qui représentent les paramètres de l'instruction.

Chaque code opération correspond à une valeur stockée sur un octet.

Exemple :

```
package fr.jmdoudoux.dej;

public class ClasseDeTest {

  public static void main(String[] args) {
    for (int i =1; i <=10; i++) {}
  }
}
```

Résultat : utilisation de l'outil de desassemblage javap

```
C:\Documents and Settings\jmd\workspace\Tests\bin\com\jmdoudoux\test>javap -c Cl
asseDeTest
Compiled from "ClasseDeTest.java"
public class fr.jmdoudoux.dej.ClasseDeTest extends java.lang.Object{
public fr.jmdoudoux.dej.ClasseDeTest();
  Code:
   0:   aload_0
   1:   invokespecial   #8; //Method java/lang/Object."<init>":()V
   4:   return

public static void main(java.lang.String[]);
  Code:
   0:   iconst_1
   1:   istore_1
   2:   goto    8
   5:   iinc    1, 1
   8:   iload_1
   9:   bipush  10
   11:  if_icmple       5
   14:  return

}
```

La plupart des instructions sont très basiques. Par exemple, les instructions de l'exemple précédent sont :

iconst_1 : définit une constante entière ayant pour valeur 1
istore_1 : copie la valeur en haut de la pile dans la variable dont l'index est précisé

64.4.4. Le format des fichiers .class

Le format des fichiers .class est décrit dans les spécifications de la JVM.

Un fichier .class est un fichier binaire qui contient :

- un ensemble de structures de données sur la classe elle-même et ses membres (méthodes et champs)
- ...

La suite de cette section sera développée dans une version future de ce document

64.5. Le compilateur JIT

Le bytecode est indépendant de toute plate-forme : une fois le code source compilé en bytecode, celui-ci peut être exécuté tel quel sur toute plate-forme disposant d'une JVM sous réserve qu'aucun appel à du code natif ne soit fait avec l'API JNI.

La JVM se charge alors d'interpréter le bytecode lors de son exécution pour le transformer en instructions compréhensibles par le processeur de la plate-forme. Ce processus qui assure l'indépendance du bytecode vis-à-vis de la plate-forme à aussi l'inconvénient d'être lent car il nécessite une interprétation du bytecode et que cette interprétation doit avoir lieu à chaque appel d'une méthode même si cette méthode doit être invoquée plusieurs fois.

L'idée d'un compilateur JIT est de compiler en code natif le bytecode d'une méthode, de stocker le résultat de cette compilation et d'exécuter ce code compilé chaque fois que la méthode est invoquée.

Le but d'un compilateur JIT (Just In Time) est donc d'améliorer les performances de l'exécution du bytecode.

Ce compilateur est intégré à la JVM pour que son action n'intervienne qu'à l'exécution et préserve la portabilité du bytecode. Le compilateur JIT modifie le rôle de la machine virtuelle qui interprète le bytecode en compilant ce dernier à la volée en code natif. Ceci améliore généralement les performances puisqu'une fois le bytecode compilé en natif il peut être exécuté directement par le système.

Les méthodes ne sont compilées par le compilateur JIT qu'au moment de leur exécution. Une fois celle-ci compilée, c'est la version compilée qui sera exécutée au lieu de la version interprétée. L'intérêt du compilateur JIT est donc d'autant plus grand que la méthode est invoquée plus souvent.

Un compilateur JIT est inclus dans la JVM hotspot depuis la version 1.2 de Java.

La performance ajoutée par l'utilisation d'un compilateur JIT est induite par plusieurs faits :

- le code natif s'exécute plus rapidement que le code interprété
- la réutilisation du code déjà compilé nativement est plus rapide que la réinterprétation de chaque ligne de code à chaque invocation de la méthode

Le temps nécessaire au compilateur JIT pour compiler le code peut être pénalisant d'autant que le temps nécessaire au compilateur peut augmenter avec la quantité d'optimisation réalisée par le compilateur.

La machine virtuelle HotSpot peut fonctionner selon deux modes. Dans le mode client, c'est la réduction du temps de compilation qui est privilégiée au détriment des optimisations. Dans le mode serveur, c'est l'optimisation qui est privilégiée ce qui allonge le temps de compilation.

64.6. Les paramètres de la JVM HotSpot

La JVM Hotpsot possède des options standard et des options non standard qui peuvent être dépendantes de la plate-forme d'exécution. Les options standards sont décrites dans la section dédiée à la commande java.

Les paramètres non standard sont préfixés par -X : il n'y a aucune garantie sur leur support dans les différentes versions de la JVM. L'option -X permet d'obtenir un résumé des options non standard supportées par la JVM.

Résultat :

```
C:\>java -version
java version "1.6.0_11"
Java(TM) SE Runtime Environment (build 1.6.0_11-b03)
Java HotSpot(TM) Client VM (build 11.0-b16, mixed mode, sharing)

C:\>java -X
    -Xmixed           mixed mode execution (default)
    -Xint             interpreted mode execution only
    -Xbootclasspath:<directories and zip/jar files separated by ;>
                      set search path for bootstrap classes and resources
    -Xbootclasspath/a:<directories and zip/jar files separated by ;>
                      append to end of bootstrap class path
    -Xbootclasspath/p:<directories and zip/jar files separated by ;>
                      prepend in front of bootstrap class path
    -Xnoclassgc       disable class garbage collection
    -Xincgc           enable incremental garbage collection
    -Xloggc:<file>    log GC status to a file with time stamps
    -Xbatch           disable background compilation
    -Xms<size>        set initial Java heap size
    -Xmx<size>        set maximum Java heap size
    -Xss<size>        set java thread stack size
    -Xprof            output cpu profiling data
    -Xfuture          enable strictest checks, anticipating future default
    -Xrs              reduce use of OS signals by Java/VM (see documentation)
    -Xcheck:jni       perform additional checks for JNI functions
    -Xshare:off       do not attempt to use shared class data
    -Xshare:auto      use shared class data if possible (default)
    -Xshare:on        require using shared class data, otherwise fail.

The -X options are non-standard and subject to change without notice.
```

Les principales options non standard sont :

Option	Rôle
-Xint	Désactiver le compilateur JIT : dans ce cas tout le bytecode est exécuté en mode interprété uniquement.
-Xbatch	Désactiver la compilation en tâche de fond
-Xbootclasspath:*bootclasspath*	Définir les répertoires, les jar ou les archives zip qui composent les classes de bootstrap. Chaque élément est séparé par un point virgule.
-Xbootclasspath/a:*path*	Ajouter des répertoires, des jar ou des archives zip aux classes de bootstrap
-Xcheck:jni	Effectuer des contrôles poussés sur les paramètres utilisés lors d'appels à des méthodes natives avec JNI. Si un problème est détecté lors de ces contrôles, alors la machine virtuelle est arrêtée avec une erreur fatale. L'activation de cette option dégrade les performances mais renforce la stabilité de la JVM lors des appels à des méthodes natives.
-Xnoclassgc	Désactiver la récupération de la mémoire par le ramasse-miettes des classes chargées mais inutilisées. Ceci peut légèrement améliorer les performances mais provoquer un manque de mémoire.
-Xincgc	Active les collectes incrémentales pour le ramasse-miettes. Ceci permet de réduire les longs temps de pauses nécessaires au ramasse-miettes en réalisant une partie de son activité de façon concomitante avec l'exécution de l'application.
-Xloggc:*file*	Active les traces d'exécution du ramasse-miettes dans un fichier de log fourni en paramètre. Il est recommandé d'utiliser un fichier sur le système de fichiers local pour éviter des problèmes de latences réseaux. Cette option est prioritaire sur l'option -verbose:gc si les deux sont fournies à la JVM

-Xms*n*	Permet de préciser la taille initiale du tas. La valeur par défaut dépend du système d'exploitation.
-Xmx*n*	Permet de préciser la taille maximale du tas. La valeur par défaut dépend du système d'exploitation.
-Xprof	Activer l'affichage sur la console de traces de profiling. A défaut de mieux, cette option peut être utilisée dans un environnement de développement mais ne doit pas être utilisée en production car elle dégrade les performances
-Xss*n*	Permet de définir la taille de la pile des threads

Tous les paramètres qui sont préfixés par -XX sont spécifiques à la JVM HotSpot et parfois dépendants de la plate-forme d'exécution.

La syntaxe de ces options dépend de leur type :

options booléennes	la syntaxe pour activer l'option est -XX:+<option> et -XX:-<option> pour la désactiver
options numériques	-XX:<option>=<valeur>. Si la valeur représente une quantité de données, il est possible de préfixer la valeur avec une lettre qui représente l'unité utilisée ('k' ou 'K' pour kilo bytes, 'm' ou 'M' pour mega bytes, et 'g' ou 'G' pour giga bytes)
options littérales	-XX:<option>=<valeur>

La JVM possède selon sa version de nombreuses options -XX dont voici les principales :

Option	Type	Rôle
-XX:DisableExplicitGC	booléen	Empêcher l'invocation explicite de la méthode System.gc()
-XX:ScavengeBeforeFullGC	booléen	Effectuer une récupération de la mémoire de la young generation avant d'effectuer un full garbage collector
-XX:UseConcMarkSweepGC	booléen	Utiliser l'algorithme concurrent mark and sweep pour la récupération de la mémoire de la tenured generation
-XX:UseGCOverheadLimit	booléen	Activer ou non la levée d'une exception de type OutOfMemoryError si la VM passe 98% de son temps dans l'activité du ramasse-miettes pour ne récupérer qu'une faible quantité de mémoire. Le but étant d'éviter des traitements longs qui sont quasi inutiles (Depuis Java 6)
-XX:UseParallelGC	booléen	Demander l'utilisation de l'algorithme parallel collector par le ramasse-miettes (Depuis Java 1.4.1)
-XX:UseParallelOldGC	booléen	Demander l'utilisation de l'algorithme parallel compacting collector par le ramasse-miettes (Depuis Java 5 update 6)
-XX:UseSerialGC	booléen	Utiliser l'algorithme serial pour la récupération de la mémoire de la tenured generation (depuis Java 5.0)
-XX:UseThreadPriorities	booléen	Demander l'utilisation des priorités des threads natifs
-XX:MaxHeapFreeRatio	numérique	Préciser le pourcentage maximum de mémoire libre du tas après une récupération de mémoire afin de provoquer une réduction au besoin de la taille du tas
-XX:MaxNewSize	numérique	Préciser la taille maximale de la young generation (depuis Java 1.4)
-XX:MaxPermSize	numérique	Préciser la taille maximale de la permanent generation. La taille par défaut dépend de la plate-forme

-XX:MinHeapFreeRatio	numérique	Préciser le pourcentage minimum de mémoire libre du tas après une récupération de mémoire afin de provoquer une extension de la taille du tas
-XX:NewRatio	numérique	Préciser le ratio de la taille des deux générations (old et tenured). Les valeurs par défaut dépendent de la plate-forme d'exécution
-XX:NewSize	numérique	Préciser la taille de la young generation
-XX:ReservedCodeCacheSize	numérique	Préciser la taille de la zone de mémoire code cache
-XX:SurvivorRatio	numérique	Préciser le ratio de la taille des espaces eden et des deux survivors de la young generation
-XX:ThreadStackSize	numérique	Préciser la taille en kilo octets de la pile d'un thread. La valeur 0 indique d'utiliser la valeur par défaut
-XX:UseFastAccessorMethods	booléen	Demander l'utilisation de la version optimisée des getters
-XX:StringCache	booléen	Activer la mise en cache des chaînes de caractères.
-XX:CITime	booléen	Afficher des informations sur le temps d'exécution du compilateur JIT (depuis Java 1.4)
-XX:ErrorFile	littéral	Préciser le fichier qui va contenir la log en cas d'erreur fatale. (depuis Java 6)
-XX:HeapDumpPath	littéral	Préciser le chemin ou le nom du fichier qui va contenir le dump du tas (depuis Java 1.4.2, Java 5 update 7)
-XX:HeapDumpOnOutOfMemoryError	booléen	Demander la génération d'un dump au format binaire HPROF dans un fichier du répertoire courant dans le cas où une exception de type OutOfMemoryError est levée. Le nom de ce fichier est de la forme java_pidxxxx.hprof où xxxx est le pid de la JVM. (depuis Java 1.4.2 update 12 et Java 5.0 update 7)
-XX:OnError	littéral	Demander l'exécution d'un script ou d'une ou plusieurs commandes séparées par un point virgule lorsqu'une erreur fatale survient. (depuis Java 1.4.2 update 9). La séquence %p peut être utilisée pour indiquer le process ID (pid) Exemple : java -XX:OnError="cat hs_err_pid%p.log|mail jmd@test.fr" MomApp
-XX:OnOutOfMemoryError	littéral	Demander l'exécution d'un script ou d'une ou plusieurs commandes séparées par un point virgule lorsqu'une exception de type OutOfMemomryError est levée. (depuis Java 1.4.2 update 12)
-XX:PrintClassHistogram	booléen	Afficher un histogramme des instances de classes du tas lors de l'appui sur Ctrl-Break (Depuis Java 1.4.2)
-XX:PrintConcurrentLocks	booléen	Afficher une liste des verrous d'accès concurrents (locks) de chaque thread lors de l'appui sur Ctrl-Break (Depuis Java 6)
-XX:PrintCommandLineFlags	booléen	Afficher les options fournies à la JVM par la ligne de commande (depuis Java 5)
-XX:PrintCompilation	booléen	Activer l'affichage de messages d'information lors de la compilation du bytecode d'une méthode.
-XX:PrintGC	booléen	Activer l'affichage de messages d'informations lors de l'exécution du ramasse-miettes
-XX:PrintGCDetails	booléen	Activer l'affichage de messages d'informations détaillées lors de

		l'exécution du ramasse-miettes (depuis Java 1.4)
-XX:PrintGCTimeStamps	booléen	Afficher un timestamp à chaque exécution du ramasse-miettes (depuis Java 1.4)
-XX:PrintTenuringDistribution	booléen	Afficher une liste de la taille des objets ayant survécus aux dernières exécutions du ramasse-miettes dans la young generation (Depuis Java 6 pour le parallel collector)
-XX:TraceClassLoading	booléen	Activer l'affichage de messages lors du chargement des classes
-XX:TraceClassUnloading	booléen	Activer l'affichage de messages lors du déchargement des classes
-XX:ShowMessageBoxOnError	booléen	Afficher une demande à l'utilisateur s'il souhaite lancer le débogueur natif (exemple Visual Studio sous Windows) si la JVM rencontre une erreur fatale.

Depuis Java 6, il est possible de modifier dynamiquement certaines de ces options en utilisant le MBean HotSpotDiagnostic exposé par la JVM.

Le contenu de ce chapitre concerne la version 1.6 de la JVM HotSpot.

Résultat :

```
C:\Documents and Settings\T30>java -version
java version "1.6.0_02"
Java(TM) SE Runtime Environment (build 1.6.0_02-b06)
Java HotSpot(TM) Client VM (build 1.6.0_02-b06, mixed mode, sharing)
```

L'option -X permet d'obtenir de l'aide sur les paramètres de la JVM dont la plupart concerne la gestion de la mémoire.

Résultat :

```
C:\>java -X
    -Xmixed           mixed mode execution (default)
    -Xint             interpreted mode execution only
    -Xbootclasspath:<directories and zip/jar files separated by ;>
                      set search path for bootstrap classes and resources
    -Xbootclasspath/a:<directories and zip/jar files separated by ;>
                      append to end of bootstrap class path
    -Xbootclasspath/p:<directories and zip/jar files separated by ;>
                      prepend in front of bootstrap class path
    -Xnoclassgc       disable class garbage collection
    -Xincgc           enable incremental garbage collection
    -Xloggc:<file>    log GC status to a file with time stamps
    -Xbatch           disable background compilation
    -Xms<size>        set initial Java heap size
    -Xmx<size>        set maximum Java heap size
    -Xss<size>        set java thread stack size
    -Xprof            output cpu profiling data
    -Xfuture          enable strictest checks, anticipating future default
    -Xrs              reduce use of OS signals by Java/VM (see documentation)
    -Xcheck:jni       perform additional checks for JNI functions
    -Xshare:off       do not attempt to use shared class data
    -Xshare:auto      use shared class data if possible (default)
    -Xshare:on        require using shared class data, otherwise fail.
The -X options are non-standard and subject to change without notice.
```

L'option -Xms permet de préciser la taille initiale du tas (heap) de la JVM

Résultat :

```
-Xms256m
```

Généralement la valeur par défaut de ce paramètre est insuffisante surtout pour des applications serveur.

L'option -Xmx permet de préciser la taille maximale du tas (heap) de la JVM.

Résultat :

```
-Xmx512m
```

La quantité de mémoire peut être précisée avec plusieurs unités :

- 'k' or 'K' pour kilobytes,
- 'm' or 'M' pour megabytes,
- 'g' or 'G' pour gigabytes

La JVM étend automatiquement la taille du tas de la taille précisée par Xms jusqu'à Xmx lorsque le pourcentage de l'espace libre devient inférieur à la valeur précisée par le paramètre -XX:MinHeapFreeRatio. Le paramètre -XX:MaxHeapFreeRatio est équivalent mais réduit la taille du tas si le pourcentage d'espace libre est supérieur à celui fourni.

L'option -verbose:gc permet d'afficher des informations sur chaque récupération de mémoire dont chacune sera sur une ligne distincte.

Résultat :

```
[GC 896K->248K(5056K), 0.0057627 secs]
[GC 1144K->343K(5056K), 0.0034792 secs]
[GC 1239K->504K(5056K), 0.0035857 secs]
```

Les valeurs numériques de part et d'autre du signe -> correspondent à la valeur de mémoire occupée avant et après la récupération de mémoire.

Le nombre de secondes indique le temps utilisé par la récupération de mémoire.

Le paramètre -XX:+PrintGCTimesStamps permet d'ajouter en début de ligne un timestamp pour chaque exécution.

Résultat :

```
0.296: [GC 896K->248K(5056K), 0.0057633 secs]
0.439: [GC 1144K->343K(5056K), 0.0033870 secs]
0.548: [GC 1239K->504K(5056K), 0.0035510 secs]
```

L'option -Xnoclassgc permet de désactiver le déchargement d'une classe lorsque plus aucune instance de cette classe n'est présente dans la mémoire de la JVM. Ceci évite d'avoir à recharger la classe.

64.7. Les interactions de la machine virtuelle avec des outils externes

La machine virtuelle propose des interfaces pour permettre sa connexion avec des outils externes de profiling ou de débogage.

64.7.1. L'API Java Virtual Machine Debug Interface (JVMDI)

JVMDI est une des couches de l'architecture de débogage de la plate-forme Java.

JVMDI est une API native de bas niveau utilisée par les débogueurs et d'autres outils de programmation pour interagir de manière bidirectionnelle avec la JVM. Elle permet d'inspecter l'état et de contrôler l'exécution des applications dans une

JVM.

Les clients JVMDI s'exécutent dans la même machine virtuelle que l'application en cours de débogage et accèdent à JVMDI en utilisant une interface native.

Un client JVMDI peut être notifié d'événements qui surviennent durant l'exécution. JVMDI peut interroger et contrôler l'application en utilisant différentes fonctions, soit en réponse à des événements, soit indépendamment de ceux-ci.

À partir de Java 5.0, JVMDI est dépréciée et est retirée à partir de Java 6.

64.7.2. L'API Java Virtual Machine Profiler Interface (JVMPI)

L'API Java Virtual Machine Profiler Interface (JVMPI) standardise les interactions entre la JVM et un profiler.

C'est une interface bidirectionnelle qui définit

- comment la JVM notifie des événements d'exécution (appel de méthodes, création d'objets, démarrage de threads, ...)
- comment un profiler s'abonne aux événements et obtient des informations

Un agent du profiler est exécuté directement dans la JVM sous une forme native.

Pour exécuter l'agent, la JVM doit être lancée avec le paramètre -XrunProfilerLibrary où ProfilerLibrary est le nom de la bibliothèque native de l'agent.

JVMPI est deprecated à partir de la version 5.0 de Java et est désactivée à partir de la version 6.0 de Java.

64.7.3. L'API Java Virtual Machine Tools Interface (JVMTI)

L'API JVMPI est remplacée par l'API Java Virtual Machine Tools Interface (JVMTI). Cette API est spécifiée dans la JSR 163 (Java Platform Profiling Architecture)

Cette API est composée d'une partie native (en C/C++) et d'une partie en pure Java. Cette API permet le développement d'outils qui vont interroger l'état de la JVM (outil de profiling, monitoring, débogage, ...). Ces outils sont développés sous la forme d'agents.

L'API Java est contenue dans le package java.lang.instrument.

Les spécifications de la version 1.2 sont consultables à l'url
https://docs.oracle.com/javase/6/docs/platform/jvmti/jvmti.html

Le JDK propose plusieurs exemples de mise en oeuvre de JVMTI dans le sous-répertoire demo/jvmti du répertoire d'installation du JDK.

64.7.4. L'architecture Java Platform Debugger Architecture (JPDA)

Java Platform Debugger Architecture (JPDA) est une architecture pour les outils de type débogueur.

Cette architecture repose sur deux API :

- Java Virtual Machine Tools Interface (JVMTI)
- Java Debug Interface (JDI) : cette API Java doit être implémentée par l'outil de débogage.

Le protocole Java Debug Wire Protocol (JDWP) formalise les échanges entre le débogueur et les traitements en cours de débogage.

Les spécifications de JPDA sont consultables à l'url
https://docs.oracle.com/javase/6/docs/technotes/guides/jpda/index.html

Un exemple de mise en oeuvre de JPDA est proposé dans le sous-répertoire /demo/jpda du répertoire d'installation du JDK.

64.7.5. Des outils de profiling

Il existe plusieurs profilers open source notamment :

- VisualVM : https://visualvm.github.io/ propose des profilers basiques
- Netbeans profiler : http://profiler.netbeans.org/
- Eclipse Test & Performance Tools Platform : https://projects.eclipse.org/projects/tptp.platform n'est malheureusement plus maintenu
- JBoss Profiler : http://www.jboss.org/jbossprofiler/
- Jprof : http://perfinsp.sourceforge.net/jprof.html

Une liste complète des profilers open source est disponible à l'url :

https://java-source.net/open-source/profilers

Il existe aussi plusieurs solutions commerciales notamment JProfiler ou OptimizeIt.

64.8. Service Provider Interface (SPI)

Java 6 propose un mécanisme pour charger dynamiquement des services définis dans un fichier de configuration dédié. Un service est un ensemble d'interfaces et de classes qui fournissent des fonctionnalités particulières.

Le Service Provider Interface (SPI) est un mécanisme qui permet de charger dynamiquement des objets respectant une interface définie. Le JDK propose plusieurs services de type SPI dans ses API, par exemple :

- javax.annotation.processing.Processor
- java.sql.Driver
- javax.persistence.spi.PersistenceProvider
- ...

Par convention dans le JDK, certains services sont dans un sous-package spi (exemple : java.text.spi, java.nio.channels.spi, java.nio.charset.spi, ...).

Cette fonctionnalité permet la recherche et le chargement de classes dynamiques au chargement d'un jar. Le service provider est donc un mécanisme simple et pratique pour permettre l'utilisation d'implémentations différentes d'un service.

Un service provider interface (SPI) est une interface ou une classe abstraite qui définit les fonctionnalités du service. Un service provider est une implémentation d'un SPI.

Le mécanisme de SPI permet de mettre en place une certaine extensibilité et modularité dans une application. Ceci peut par exemple permettre la mise en oeuvre d'un mécanisme d'utilisation de plug-in basique.

64.8.1. La mise en oeuvre du SPI

Il faut définir une interface qui décrit les fonctionnalités proposées par le service.

Exemple :

```
package fr.jmdoudoux.dej.spi;

```

```
public interface MonService {
  public void executer();
}
```

Il faut définir une ou plusieurs implémentations de l'interface du service qui doivent obligatoirement avoir un constructeur sans paramètres.

Exemple :

```
package fr.jmdoudoux.dej.spi;

public class MonServiceSimple implements MonService {

  public void executer() {
    System.out.println("Mon service simple");
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.spi;

public class MonServiceComplexe implements MonService {

  public void executer() {
    System.out.println("Mon service complexe");
  }
}
```

Il faut créer un sous-répertoire META-INF/services dans le jar.

Il faut créer un fichier dans le sous-répertoire services qui se nomme du nom pleinement qualifié de l'interface. Ce fichier est le fichier de configuration qui doit contenir le nom pleinement qualifié de chaque implémentation utilisable, chacune étant sur une ligne dédiée. Il est possible de mettre des commentaires en utilisant le caractère #. Le fichier de configuration doit être encodé en UTF-8.

Exemple : le fichier META-INF/services/fr.jmdoudoux.dej.spi.MonService

```
# implementation simple du service
fr.jmdoudoux.dej.spi.MonServiceSimple
# implementation complexe du service
fr.jmdoudoux.dej.spi.MonServiceComplexe
```

Au runtime, une classe dédiée va regarder le contenu des fichiers de configuration contenus dans le sous-répertoire META-INF/services du jar.

Les services providers compilés doivent être ajoutés dans le classpath : une solution pratique est de les packager dans une archive jar dédiée. Cela permet de remplacer la ou les implémentations utilisées simplement en remplaçant le jar par celui qui contient la ou les nouvelles versions.

64.8.2. La classe ServiceLoader

La classe java.util.ServiceLoader permet de rechercher, charger et utiliser un service provider défini dans un fichier de configuration.

La recherche se fait dans le classpath de l'application.

La classe java.util.ServiceLoader possède plusieurs méthodes dont :

- load() : fabrique qui permet de créer une instance de ServiceLoader

- loadlnstalled() : cette méthode charge des providers uniquement dans le répertoire d'extension du JRE (jre/lib/ext)
- iterator() : obtenir un itérator sur les providers trouvés
- reload() : permet de vider le cache des providers et de recharger les providers

La méthode load() est une fabrique d'une instance de ServiceLoad capable de charger les instances d'un service dont l'interface est fournie en paramètre.

La méthode loadlnstalled() effectue la recherche de providers uniquement dans le répertoire d'extension du JRE qui par défaut est son répertoire lib/ext.

Le chargement du provider peut se faire avec un classloader dédié fourni en paramètre de la méthode load() ou loadlnstalled().

Exemple :

```
package fr.jmdoudoux.dej.spi;

import java.util.Iterator;
import java.util.ServiceLoader;

public class MonApp {

  public static void main(final String[] args) {
    final MonService service;
    final ServiceLoader<MonService> loader = ServiceLoader.load(MonService.class);
    final Iterator<MonService> iterator = loader.iterator();
    if (iterator.hasNext()) {
      service = iterator.next();
      service.executer();
    }
  }
}
```

L'exemple ci-dessus va utiliser le premier service trouvé lors de la recherche.

Si le service MonServicesSimple est packagé dans un jar avec le fichier
META-INF/services/fr.jmdoudoux.dej.spi.MonService contenant

Exemple :

```
# implementation simple du service
fr.jmdoudoux.dej.spi.MonServiceSimple
```

L'exécution de la classe MonApp avec le jar dans le classpath affiche :

Résultat :

```
Mon service simple
```

Si le service MonServiceComplexe est packagé dans un jar avec le fichier
META-INF/services/fr.jmdoudoux.dej.spi.MonService contenant

Exemple :

```
# implementation complexe du service
fr.jmdoudoux.dej.spi.MonServiceComplexe
```

L'exécution de la classe MonApp avec le jar dans le classpath affiche :

Résultat :

```
Mon service complexe
```

Il est possible de parcourir l'iterator pour sélectionner un ou plusieurs services à utiliser.

Exemple :

```
package fr.jmdoudoux.dej.spi;

import java.util.Iterator;
import java.util.ServiceLoader;

public class MonApp {

  public static void main(final String[] args) {
    MonService service;
    final ServiceLoader<MonService> loader = ServiceLoader.load(MonService.class);
    final Iterator<MonService> iterator = loader.iterator();
    while (iterator.hasNext()) {
      service = iterator.next();
      service.executer();
    }
  }
}
```

Résultat :

```
Mon service simple
Mon service complexe
```

Par défaut, la classe ServiceLoader utilise un cache pour ne pas rechercher les implémentations à chaque fois. La classe ServiceLoader est final : elle ne peut donc pas être sous classée pour par exemple modifier l'emplacement de recherche des providers.

L'API est détaillée dans le chapitre «L'API Service Loader».

64.9. Les JVM 32 et 64 bits

Il est fréquent d'entendre que 64 bits c'est mieux que 32 bits : c'est vrai pour certaines fonctionnalités mais pas toujours vrai pour une JVM.

Les JVM 32 et 64 bits diffèrent essentiellement par leur façon d'accéder à la mémoire et dans la quantité de mémoire qu'elles peuvent utiliser. Les JVM 32 bits possèdent une limite maximale de la taille du heap qui dépend du système d'exploitation et de l'implémentation mais elle varie généralement entre 1,5 et 2,5 gigaoctets.

Le grand intérêt d'utiliser une JVM 64 bits est de pouvoir utiliser un heap d'une taille bien supérieure à celle d'une JVM 32 bits. Cependant, il est important de prendre en compte que la même quantité de données occupera plus de mémoire dans une JVM 64 bits.

Les JVM Hotspot d'Oracle et J9 d'IBM sont proposées en version 32 et 64 bits.

Dans un navigateur, il faut installer la version 32 ou 64 bits de Java qui correspond à la version 32 ou 64 bits du navigateur. Même sur un Windows 64 bits, c'est un navigateur 32 bits qui est exécuté par défaut.

64.9.1. L'avantage des architectures 64 bits

Les JVM 64 bits permettent d'utiliser une plus grande quantité de mémoire pour le heap. Généralement, cette mémoire est uniquement limitée par la mémoire physique disponible sur le système qui, en 64 bits, peut être de grande taille.

L'utilisation d'un système d'exploitation 64 bits peut cependant intrinsèquement offrir aussi des avantages non négligeables sur des fonctionnalités natives notamment pour des serveurs : c'est par exemple le cas pour les IO et pour les calculs en virgule flottante.

Sur une architecture 64 bits, les calculs sur des types primitifs long ou double sont plus rapides puisque toute leur valeur (stockée dans 64 bits) peut être chargée directement en une seule opération de lecture alors qu'il en faut deux sur une architecture 32bits.

Dans tous les autres cas, les performances pourront aussi être obtenues indirectement grâce au fait que le processeur (registre ou instructions spécifiques) et le système d'exploitation sont en 64 bits ou que la taille du heap permet une mise en cache plus importante des données ce qui limitera par exemple les échanges I/O ou réseau.

Généralement, un système d'exploitation 64 bits n'impose pas l'utilisation de processus 64 bits.

64.9.2. JLS et JVM 64 bits

La taille des variables de type primitives reste la même sur une JVM 32 et 64 bits : c'est un des fondements même du bytecode pour assurer la portabilité de Java. Les spécifications de Java imposent la taille en octets des types primitifs : la taille d'une variable de type long ou double est déjà de 64 bits et une variable de type int ou float est toujours sur 32 bits.

L'utilisation d'une JVM 32 ou 64 bits n'a aucune incidence sur le code source Java. Il n'est d'ailleurs pas possible de demander la compilation en 64 bits du code source Java. Le bytecode est par définition indépendant de sa plate-forme d'exécution pour assurer sa portabilité. Chaque opérateur du bytecode repose sur un octet sur une JVM 32 et 64 bits. Sans utiliser de code natif, grâce à JNI, le bytecode s'exécute de façon identique sur une JVM 32 et 64 bits.

Intrinsèquement une JVM est définie pour être 32 bits. Ainsi l'utilisation d'une JVM 64 bits ne permettra pas de dépasser certaines limites comme par exemple le nombre maximum d'éléments dans un tableau dont l'index est stocké dans une variable 32 bits.

L'utilisation d'une JVM 64 bits n'améliore généralement pas les performances pures de la JVM sauf si elle est capable d'utiliser certaines caractéristiques du processeur comme des registres spécifiques.

64.9.3. L'introduction de la fonctionnalité compressed OOPS (Ordinary Object Pointers)

Les JVM 64 bits d'Oracle et IBM (depuis la version 1.6) proposent une fonctionnalité qui permet de comprimer les références (compressed oops) : elle permet de réduire la taille des références. La compression des pointeurs de mémoire internes à la JVM permet de réduire la taille du heap nécessaire pour stocker une même quantité de données dans une JVM 32 et une 64 bits.

Plusieurs JVM possèdent des fonctionnalités similaires dont l'implémentation et les limitations notamment en termes de taille maximale de heap varient

- Oracle/Sun Hotspot : -XX:+UseCompressedOops
- Oracle/BEA JRockit : -XXcompressedRefs
- IBM J9 : -Xcompressedrefs

L'utilisation de cette fonctionnalité limite la taille maximale du heap. Par exemple, sur une J9 d'IBM avec les compressed references, qui sont activées par défaut, la taille maximale du heap est à 28 Go.

L'utilisation des compressed oops ajoute cependant un léger overhead pour permettre de transformer les adresses réduites vers les adresses natives et vice versa. Cette légère dégradation des performances est due aux calculs réalisés pour compresser sur 4 octets et décompresser sur 8 octets les pointeurs car physiquement sur la machine les pointeurs sont sur 64 bits.

L'option -XX:+UseCompressedOops de la JVM Hotspot 64 bits permet de demander la compression de la taille des pointeurs sur 32 bits ce qui réduit l'empreinte mémoire occupée par une même quantité de données dans le heap : au lieu d'utiliser la taille native d'un pointeur, elle compresse cette taille pour être équivalente à celle requise sur une JVM 32 bits.

Cette option n'est utilisable que si la taille maximale du heap est inférieure à 32 Go qui correspond à la taille maximale adressable avec des pointeurs 32 bits. Elle permet l'utilisation de heaps de plus grande taille qu'une JVM 32 bits tout en utilisant une taille de pointeurs sur 4 octets comme sur une JVM 32 bits.

L'utilisation de cette option n'est possible qu'à partir de la version Java 6 update 14 de la JVM Hotspot d'Oracle. L'option est activée par défaut à partir de la version Java 6 update 23 de la JVM Hotspot selon la taille maximale demandée au lancement de la JVM. Avec Java 7 64 bits, l'option est activée par défaut quand l'option -Xmx n'est pas précisée et quand la taille précisée par l'option -Xmx est inférieure à 32 Go.

64.9.4. Les limitations et les contraintes avec une JVM 32 et 64 bits

Il n'est pas obligatoire d'utiliser une JVM 64bits sur un système d'exploitation 64 bits : il est même courant d'utiliser une JVM 32 bits sur un OS 64 bits. Il n'est pas possible de forcer l'utilisation en 32 bits d'une JVM 64 bits.

Généralement, les performances d'une même application exécutée sur une JVM 64 bits sont légèrement moins bonne que lors de son exécution sur une JVM 32 bits.

De plus la même quantité de données occupe plus de place dans le heap d'une JVM 64 bits par rapport à une JVM 32 bits.

64.9.4.1. Les limites de la taille du heap d'une JVM 32 bits

Un processeur 32 bits ne représente les adresses que sur quatre octets : ainsi le nombre maximum d'adresses utilisables est de 2 puissance 32 ce qui correspond à espace d'adressage de 4294967296 octets soit 4 gigaoctets.

En théorie, il est donc possible d'adresser un peu moins de 4Go de mémoire sur un système 32 bits mais dans la pratique c'est toujours inférieur à 3 Go et même dans certains cas inférieur à 1,5 Go. Une partie de la mémoire est utilisée par le système d'exploitation et par d'autres applications. La fragmentation de la mémoire peut aussi réduire la taille maximale allouable. Ceci dépend du système d'exploitation et de l'implémentation de la JVM.

La limite la plus importante pour une JVM 32 bits est probablement sur Windows. Ceci est dû à la façon dont Windows 32 bits gère son espace mémoire. Le système d'exploitation Windows sépare ces 4 Go en deux parties : une utilisée par le noyau et la pagination de la mémoire et l'autre est utilisée pour les processus. Windows 32 bits fixe une limite de 2Go de mémoire pour chaque processus.

Le heap ne représente pas l'intégralité de la mémoire requise par une JVM : il y a aussi le processus chargé en mémoire, la permgen et différentes dll associées aux processus et chargées par le système. Ceci limite généralement la taille maximale du heap d'une JVM sous Windows 32 bits à moins de 1,6 Go. De plus l'implémentation Hotspot sous Windows requiert que l'espace mémoire utilisé par le tas soit d'un seul tenant, ce qui peut ajouter une contrainte sur la taille maximale du heap utilisable et encore diminuer sa valeur.

Pour parer à cette limitation, Windows propose l'API AWE qui permet à un processus d'utiliser plus de 3 Go de mémoire. Cette fonctionnalité présente une contrainte : le process doit être compilé avec cette API pour pouvoir l'exploiter. Cependant le choix a été fait de ne pas utiliser cette API dans la JVM Hotspot.

Sous un Windows 32 bits avec une JVM JRockit, il est possible d'allouer jusqu'à 2,8 Go de heap grâce notamment à la gestion de la mémoire qui autorise le morcellement.

64.9.4.2. L'utilisation d'une JVM 64 bits

Une implémentation 64 bits d'une JVM est prévue pour s'exécuter dans un système d'exploitation 64 bits sur une machine avec un processeur 64 bits.

L'utilisation d'une JVM 64 bits implique généralement :

- Soit une légère amélioration des performances grâce à l'exploitation des optimisations du hardware par l'implémentation de la JVM (ajout de nouveaux registres par exemple sur certains processeurs AMD)
- Soit une légère dégradation des performances liée au traitement des pointeurs dont la taille est plus importante

L'intérêt principal d'utiliser une JVM 64 bits concerne la possibilité d'utiliser un heap de grande taille.

Résultat :

```
C:\Users\jm>java
-Xms4096m -Xmx24096m -version\n
java version "1.7.0_15"\n
Java(TM) SE Runtime Environment (build 1.7.0_15-b03)\n
Java HotSpot(TM) 64-Bit Server VM (build 23.7-b01, mixed mode)
```

L'adressage d'une plus grande quantité de mémoire implique une légère dégradation des performances et une augmentation de l'espace mémoire requis pour stocker une même quantité de données. Ceci est lié au fait qu'une référence requiert 8 octets dans une JVM 64 bits contre seulement 4 octets pour une JVM 32bits. Il faut donc plus de mémoire pour stocker la même quantité d'objets sur une JVM 64 bits.

Dans une JVM 64 bits, la taille d'un pointeur sur un objet est doublée par rapport au même pointeur dans une JVM 32 bits et la taille du header d'un objet est aussi plus importante. Dans une JVM 32 bits, la référence d'un objet nécessite 4 octets et le header d'un objet nécessite 8 octets. Dans une JVM 64 bits, le header d'un objet est de 12 octets et la référence d'un objet nécessite 8 octets.

Plus le nombre d'objets est important dans la JVM, plus leur taille dans une JVM 64 bits sera importante par rapport à la taille requise dans une JVM 32 bits.

Une taille de heap importante peut avoir des effets bénéfiques : par exemple, la quantité de données en cache peut être augmentée. Mais elle présente aussi des inconvénients : par exemple, plus la taille du heap est importante plus long sera le temps nécessaire au ramasse-miettes pour effectuer ses traitements.

L'exécution du ramasse-miettes sur un heap plus grand augmente le temps de pause requis pour effectuer ses traitements. Plus la taille du heap augmente, moins le ramasse-miettes effectue de full garbage collections mais la durée de leurs exécutions est plus longue.

Attention : l'utilisation d'un heap de grande taille dans une JVM 64 bits nécessite de prendre en compte que le temps de pause requis pour les full garbage collections soit plus long. Il est donc nécessaire d'optimiser la configuration de la JVM et du ramasse-miettes en particulier pour limiter la durée de ces temps de pause notamment en utilisant un algorithme de type concurrent ou parallèle. Ceux-ci pourront réaliser certaines de leurs actions dans des threads dédiés en concurrence ou en parallèle avec l'exécution de l'application ce qui permettra de réduire ces temps de pause.

Avec une JVM Hotspot, il est recommandé d'utiliser des ramasse-miettes de type Parallel ou Concurrent pour des heaps dont la taille dépasse les 2 Go ceci afin de limiter l'overhead induit par les traitements de récupération de la mémoire inutilisée notamment :

- en effectuant une partie de ces traitements de manière concurrente à l'exécution de l'application
- en utilisant plusieurs CPU pour ces traitements

Un second effet indirect de l'augmentation de la taille du heap est qu'il est possible d'avoir aussi plus de threads puisque chacun d'eux à besoin d'un petit espace pour stocker sa pile.

Dans une JVM 32 bits, le nombre de threads qu'il est possible de lancer est généralement de l'ordre du millier. Dans une JVM 64 bits, il est possible de lancer une centaine de milliers de threads avec la mémoire nécessaire.

La taille de la pile de chaque thread est cependant plus importante sur une JVM 64 bits que sur une 32 bits. Cette taille par défaut varie selon le système d'exploitation et l'implémentation de la JVM mais elle est en générale de 1024Ko pour une JVM 64 bits et de 320Ko ou 512Ko pour une JVM 32 bits.

64.9.4.3. Les contraintes liées à l'utilisation de bibliothèques natives

Une bibliothèque native 32 bits ne peut être chargée que dans une JVM 32 bits, idem pour une bibliothèque 64 bits qui ne peut être chargée que dans une JVM 64 bits. En revanche, il n'est pas possible de charger une bibliothèque 32 bits dans une JVM 64 bits.

L'utilisation de code natif dans une JVM doit donc correspondre : du code natif 32 bits doit être recompilé en 64 bits pour pouvoir être utilisé dans une JVM 64 bits.

64.9.5. Le choix entre une JVM 32 ou 64 bits

Le choix d'utiliser une JVM 64 bits est généralement dicté par le besoin d'une taille de heap importante (supérieure à la taille maximale allouable à une JVM 32 bits) et/ou par le besoin d'utiliser des bibliothèques natives compilées en 64 bits.

L'utilisation d'une JVM 64 bits peut être intéressante si l'application réalise beaucoup de calculs avec des variables de type long ou double ou si elle a besoin de faire beaucoup d'I/O.

Si le heap requis est compris entre 2Go et 4Go, il peut être intéressant d'utiliser une JVM 32 bits sur un système d'exploitation 64 bits.

Dans les autres cas, une JVM 32 bits doit pouvoir répondre aux besoins.

Par exemple, le tableau ci-dessous aide à choisir la JVM Hotspot à utiliser selon le système d'exploitation et la taille du heap souhaitée :

Système d'exploitation	Taille du heap	JVM
Windows	< 1,4 Go	32 bits
Windows 64 bits	> 1,4 Go et < 32 Go	64 bits avec -XX :+UseCompressedOops
Windows 64 bits	> 32 Go	64 bits
Linux	< 2 Go	32 bits
Linux 64 bits	> 2 Go et < 32 Go	64 bits avec -XX :+UseCompressedOops
Linux	> 32 Go	64 bits

Une JVM 64 bits est particulièrement adaptée pour des applications manipulant de très grandes quantités de données :

- batches
- caches
- base de données en mémoire
- moteur de recherche

64.9.6. Déterminer si la JVM est 32 ou 64 bits

Pour connaitre le mode de fonctionnement d'une JVM, il suffit de la lancer avec l'option -version. La version 32 bits ne fournie aucune information explicite sur l'architecture alors que la version 64 bits la précise clairement.

Résultat :

```
C:\Users\jm>java -version\n
java version "1.7.0_15"\n
Java(TM) SE Runtime Environment (build 1.7.0_15-b03)\n
Java HotSpot(TM) 64-Bit Server VM (build 23.7-b01, mixed mode)
C:\Users\jm>cd C:\Program Files (x86)\Java\jdk1.7.0_15\bin
C:\Program Files (x86)\Java\jdk1.7.0_15\bin>java -version
java version "1.7.0_15"
```

```
Java(TM) SE Runtime Environment (build 1.7.0_15-b03)
Java HotSpot(TM) Client VM (build 23.7-b01, mixed mode, sharing)
```

La JVM Hotspot propose les options -d32 et -d64 mais elles sont informatives et ne fonctionnent pas correctement sur Windows XP.

Elles ne permettent pas de forcer le mode de fonctionnement de la JVM.

Résultat :

```
C:\Users\jm>java -version\n
java version "1.7.0_15"\n
Java(TM) SE Runtime Environment (build 1.7.0_15-b03)\n
Java HotSpot(TM) 64-Bit Server VM (build 23.7-b01, mixed mode)\n
C:\Users\jm>java -d32 -version\n
Error: This Java instance does not support a 32-bit JVM.\n
Please install the desired version.
```

Normalement, le code Java ne devrait jamais dépendre de cette propriété pour permettre la mise en oeuvre de « write once, run everywhere ».

Elle peut cependant être utile pour conditionner le chargement dynamique d'une bibliothèque native.

Il est possible de consulter la valeur de la propriété sun.arch.data.model d'une JVM Hotspot pour déterminer si la JVM est 32 ou 64 bits.

Exemple :

```
package fr.jmdoudoux.dej;

public class TestJVM3264 {
  public static void main(final String[] args) {
    System.out.println(System.getProperty("sun.arch.data.model"));
  }
}
```

La propriété vaut «32» si la JVM est une 32 bits et «64» si la JVM est une 64 bits ou «unknown» si la valeur n'est pas définie.

Il est aussi possible de vérifier la valeur de la propriété «os.arch» de la JVM qui renvoie une chaîne de caractères précisant l'architecture de la plate-forme.

La valeur est «x86» sur une machine avec un Windows 32 bits.

65. La gestion de la mémoire dans la JVM HotSpot

Chapitre 65

Niveau : 🟩🟩🟨🟨🟥⬜ Confirmé

Ce chapitre détaille la gestion de la mémoire dans la JVM HotSpot.

Celle-ci repose en grande partie sur le ramasse-miettes ou garbage collector (les deux désignations sont utilisées dans ce chapitre) dont le mode de fonctionnement et la mise oeuvre sont largement détaillés dans ce chapitre.

Ce chapitre présente comment obtenir des informations sur la mémoire, sur les différentes exceptions liées à la mémoire et sur les fuites de mémoire.

Ce chapitre contient plusieurs sections :

- Le ramasse-miettes (Garbage Collector ou GC)
- Le fonctionnement du ramasse-miettes de la JVM Hotspot
- Le paramétrage du ramasse-miettes de la JVM HotSpot
- Le monitoring de l'activité du ramasse-miettes
- Les différents types de référence
- L'obtention d'informations sur la mémoire de la JVM
- Les fuites de mémoire (Memory leak)
- Les exceptions liées à un manque de mémoire

65.1. Le ramasse-miettes (Garbage Collector ou GC)

Le ramasse-miettes est une fonctionnalité de la JVM qui a pour rôle de gérer la mémoire notamment en libérant celle des objets qui ne sont plus utilisés.

La règle principale pour déterminer qu'un objet n'est plus utilisé est de vérifier qu'il n'existe plus aucun autre objet qui lui fait référence. Ainsi un objet est considéré comme libérable par le ramasse-miettes lorsqu'il n'existe plus aucune référence dans la JVM pointant vers cet objet.

Lorsque le ramasse-miettes va libérer la mémoire d'un objet, il a l'obligation d'exécuter un éventuel finalizer définit dans la classe de l'objet. Attention, l'exécution complète de ce finalizer n'est pas garantie : si une exception survient durant son exécution, les traitements sont interrompus et la mémoire de l'objet est libérée sans que le finalizer soit entièrement exécuté.

La mise en oeuvre d'un ramasse-miettes possède plusieurs avantages :

- elle améliore la productivité du développeur qui est déchargé de la libération explicite de la mémoire
- elle participe activement à la bonne intégrité de la machine virtuelle : une instruction ne peut jamais utiliser un objet qui n'existe plus en mémoire

Mais elle possède aussi plusieurs inconvénients :

- le ramasse-miettes consomme des ressources en terme de CPU et de mémoire
- il peut être à l'origine de la dégradation plus ou moins importante des performances de la machine virtuelle
- le mode de fonctionnement du ramasse miettes n'interdit pas les fuites de mémoires si le développeur ne prend pas certaines précautions. Généralement issues d'erreurs de programmation subtiles, ces fuites sont assez difficiles à corriger.

65.1.1. Le rôle du ramasse-miettes

Le garbage collector a plusieurs rôles :

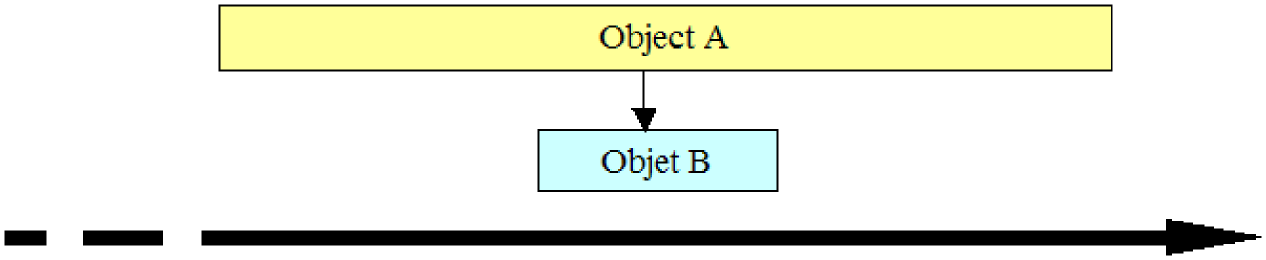
- s'assurer que tout objet dont il existe encore une référence n'est pas supprimé
- récupérer la mémoire des objets inutilisés (dont il n'existe plus aucune référence)
- éventuellement défragmenter (compacter) la mémoire de la JVM selon l'algorithme utilisé
- intervenir dans l'allocation de la mémoire pour les nouveaux objets à cause du point précédent

Le ramasse-miettes s'exécute dans un ou plusieurs threads de la JVM.

Les objets en cours d'utilisation (dont il existe encore une référence) sont considérés comme "vivants". Les objets inutilisés (ceux dont plus aucun autre objet ne possède une référence) sont considérés comme pouvant être libérés. Les traitements pour identifier ces objets et libérer la mémoire qu'ils occupent se nomment garbage collection. Ces traitements sont effectués par le garbage collector ou ramasse-miettes en français.

Le rôle primaire d'un ramasse-miettes est de trouver les objets de la mémoire qui ne sont plus utilisés par l'application et de libérer l'espace qu'ils occupent. Le principe général d'exécution du ramasse-miettes est de parcourir l'espace mémoire, marquer les objets dont il existe au moins une référence de la part d'un autre objet. Tous les objets qui ne sont pas marqués sont éligibles pour récupérer leur mémoire. Leur espace mémoire sera libéré par le ramasse-miettes ce qui augmentera l'espace mémoire libre de la JVM.

Il est important de comprendre comment le ramasse-miettes détermine si un objet est encore utilisé ou pas : un objet est considéré comme inutilisé s'il n'existe plus aucune référence sur cet objet dans la mémoire.



Dans l'exemple ci-dessus, un objet A est créé. Au cours de sa vie, un objet B est instancié et l'objet A possède une référence sur l'objet B. Tant que cette référence existe, l'objet B ne sera pas supprimé par le ramasse-miettes même si l'objet B n'est plus considéré comme utile d'un point de vue fonctionnel. Ce cas de figure est fréquent notamment avec les objets des interfaces graphiques, les listeners ou avec les collections.

L'algorithme le plus basique pour un ramasse-miettes, parcourt tous les objets, marque ceux dont il existe au moins une référence. A la fin de l'opération, tous les objets non marqués peuvent être supprimés de la mémoire. Le gros inconvénient de cet algorithme est que son temps d'exécution est proportionnel au nombre d'objets contenus dans la mémoire. De plus, les traitements de l'application sont arrêtés durant l'exécution du ramasse-miettes.

Plusieurs autres algorithmes ont été développés pour améliorer les performances et diminuer les temps de pauses liés à l'exécution du ramasse-miettes.

65.1.2. Les différents concepts des algorithmes du ramasse-miettes

Plusieurs considérations doivent être prises en compte dans le choix de l'algorithme à utiliser lors d'une collection par le ramasse-miettes :

serial ou parallel : avec une collection de type serial, une seule tâche peut être exécutée à un instant donné même si plusieurs processeurs sont disponibles sur la machine.

Avec une collection de type parallel, les traitements du garbage collector sont exécutés en concomitance par plusieurs processeurs. Le temps global de traitement est ainsi plus court mais l'opération est plus complexe et augmente généralement la fragmentation de la mémoire

stop the world ou concurrent : avec une collection de type stop the world, l'exécution de l'application est totalement suspendue durant les traitements d'une collection. Stop the world utilise un algorithme assez simple puisque durant ses traitements, les objets ne sont pas modifiés. Son inconvénient majeur est la mise en pause de l'application durant l'exécution de la collection.

Avec une collection de type concurrent, une ou plusieurs collections peuvent être exécutées simultanément avec l'application. Cependant, une collection de type concurrent ne peut pas réaliser tous ses traitements de façon concurrente et doit parfois en réaliser certains sous la forme stop the world.

De plus, l'algorithme d'une collection de type concurrent est beaucoup plus complexe puisque les objets peuvent être modifiés par l'application durant la collection : il nécessite généralement plus de ressources CPU et de mémoire pour s'exécuter.

compacting, non compacting ou copying : une fois la libération de la mémoire effectuée par le garbage collector, il peut être intéressant pour ce dernier d'effectuer un compactage de la mémoire en regroupant les objets alloués d'une part et la mémoire libre de l'autre.

Ce compactage nécessite un certain temps de traitement mais il accélère ensuite l'allocation de mémoire car il n'est plus utile de déterminer quel espace libre utiliser : s'il y a eu compactage, cette espace correspond obligatoirement à la première zone de mémoire libre rendant l'allocation très rapide.

Si la mémoire n'est pas compactée, le temps nécessaire à la collection est réduit mais il est nécessaire de parcourir la mémoire pour rechercher le premier espace de mémoire qui permettra d'allouer la mémoire requise, ce qui augmente les temps d'allocation de mémoire aux nouveaux objets et la fragmentation de cette dernière.

Il existe aussi une troisième forme qui consiste à copier les objets survivants à différentes collections dans des zones de mémoires différentes (copying). Ainsi la zone de création des objets se vide au fur et à mesure, ce qui rend l'allocation rapide. Le copying nécessite plus de mémoire.

65.1.3. L'utilisation de générations

Suite à diverses observations, plusieurs constats ont été faits sur la durée de vie des objets d'une application en général :

- un nombre particulièrement important d'objets ont une durée de vie courte (exemple : un iterator utilisé dans les traitements d'une méthode)
- la plupart des objets ayant une durée de vie longue ne possèdent pas de référence vers des objets ayant une durée de vie courte

Il est alors apparu l'idée d'introduire la notion de générations dans le traitement des collections (generational collection). L'idée est de repartir les différents objets dans différentes zones de la mémoire nommées générations selon leur durée de vie. Généralement deux générations principales sont utilisées :

- une pour les jeunes objets (Young Generation)
- et une pour les objets avec une durée de vie plus longue (Old Generation ou Tenured Generation).

L'utilisation de générations possède plusieurs intérêts :

- la portion de mémoire à collecter est réduite
- il est possible d'appliquer des algorithmes différents pour chaque génération
- et d'utiliser un algorithme optimisé selon les caractéristiques d'utilisation d'une génération

Il est facile de conclure que le nombre de collections à réaliser sur la Young Generation sera beaucoup plus important que sur la Old Generation. De plus, les collections sur la Young Generation devraient être rapides puisque, vraisemblablement, la taille sera relativement réduite et le nombre d'objets sans référence important. Les collections dans la Young Generation sont appelées collections mineures (minor collections) puisque très rapide.

Si un objet survit à plusieurs collections, il peut être promu (Tenured) dans la Old Generation. Généralement, la taille de la Old Generation est plus importante que celle de la Young Generation. Les collections sur la Old Generation sont généralement plus longues puisque la taille de la génération est plus importante mais elles sont aussi moins fréquentes.

En effet, une collection dans la Old Generation n'intervient en général qu'une fois que l'espace mémoire libre dans cette génération devient faible. Une collection dans la Old Generation étant généralement longue elle est désignée par le terme collection majeure (major collection).

Le but de l'utilisation des générations est de limiter le nombre de collections majeures effectuées.

65.1.4. Les limitations du ramasse-miettes

Le ramasse-miettes ne résout pas tous les problèmes de mémoires :

- il ne peut empêcher un manque de mémoire si trop d'objets sont à créer dans un espace mémoire trop petit
- il n'empêche pas les fuites de mémoire potentielles

De plus le ramasse-miettes est un processus complexe qui consomme des ressources et nécessite un temps d'exécution non négligeable pouvant être à l'origine de problèmes de performance.

Une bonne connaissance du mode de fonctionnement du ramasse-miettes est obligatoire pour apporter une solution lorsque celui-ci est à l'origine de goulets d'étranglements lors de l'exécution de l'application.

Le mécanisme d'allocation de mémoire est aussi lié au garbage collector car il nécessite de trouver un espace mémoire suffisant pour les besoins de l'allocation. Ceci implique pour le garbage collector de compacter la mémoire lors de la récupération de celle-ci pour limiter les effets inévitables de fragmentation.

Le garbage collector est un mécanise complexe mais fiable. Bien que complexe, son fonctionnement doit essayer de limiter l'impact sur les performances de l'application notamment en essayant de limiter son temps de traitement et la fréquence de son exécution. Pour atteindre ces objetifs, des travaux sont constamment en cours de développement afin de trouver de nouveaux algorithmes. Il peut aussi être nécessaire d'effectuer un tuning du ramasse-miettes en utilisant les nombreuses options proposées par la JVM.

La performance du garbage collector est intimement liée à la taille de la mémoire qu'il a à gérer. Ainsi, si la taille de la mémoire est petite, le temps de traitement du garbage collector sera court mais il interviendra plus fréquemment. Si la taille de la mémoire est grande, la fréquence d'exécution sera moindre mais le temps de traitement sera long. Le réglage de la taille de la mémoire influe sur les performances du garbage collector et est un des facteurs importants en fonction des besoins de chaque application.

Le ramasse-miettes fait son travail dans la JVM mais il se limite aux instances des objets créés par la machine virtuelle. Cependant, dans une application, il peut y avoir des allocations de mémoire en dehors des instances d'objets Java.

Ceci concerne des ressources natives du système qui sont allouées par un processus hors du contexte Java. C'est notamment le cas lors de l'utilisation de JNI. Dans ce cas, il faut explicitement demander la libération des ressources en invoquant une méthode dédiée car le ramasse-miettes n'a aucun contrôle sur l'espace mémoire de ces entités. Par exemple, certaines classes qui encapsulent des composants de AWT proposent une méthode dispose() qui se charge de libérer les ressources natives du système.

Le traitement du ramasse-miettes dans la Permanent Generation suit des règles particulières :

- les classes chargées par le classloader primordial ne sont jamais traitées par le ramasse-miettes
- les classes chargées par un classloader personnalisé sont considérées comme inutilisées uniquement s'il n'existe plus aucune instance de ces classes et qu'il n'existe plus de référence sur le classloader personnalisé

65.1.5. Les facteurs de performance du ramasse-miettes

Pour optimiser les performances du ramasse-miettes, il est nécessaire d'avoir des indicateurs sous la forme de métriques :

- throughput : pourcentage de temps dédié par la JVM à l'exécution de l'application
- GC overhead : pourcentage de temps dédié par la JVM à l'exécution du GC
- temps de pause : durée durant laquelle l'exécution de l'application est interrompue à cause de l'exécution des collections du ramasse-miettes
- fréquence des collections : nombre de collections exécutées
- footprint : empreinte mémoire du tas
- promptness : durée entre le moment où l'objet n'est plus utilisé et le moment où son espace mémoire est libéré

L'importance de ces indicateurs dans le tuning du ramasse-miettes dépend du type d'application utilisée, par exemple :

- une application avec une IHM doit limiter les temps de pause
- une application temps réels doit limiter le temps d'exécution et la fréquence d'exécution des collections
- l'empreinte mémoire est primordiale sur un système possédant de faibles ressources
- ...

Le choix de l'algorithme utilisé pour les collections mineures et majeures est important pour les performances globales du ramasse-miettes. Il est préférable d'utiliser un algorithme rapide pour la Young Generation et un algorithme privilégiant l'espace pour la old generation.

En général, il faut aussi privilégier la vitesse d'allocation de mémoire pour les nouveaux objets qui sont des opérations à la demande plutôt que la libération de la mémoire qui n'a pas besoin d'intervenir dès que l'objet n'est plus utilisé sauf si la JVM manque de mémoire.

Pour réaliser des applications pointues et permettre leur bonne montée en charge, il est important de comprendre les mécanismes utilisés par la JVM pour mettre en oeuvre le ramasse-miettes car celui-ci peut être à l'origine de fortes dégradations des performances.

L'algorithme le plus simple d'un ramasse-miettes parcourt tous les objets pour déterminer ceux dont il n'existe plus aucune référence. Ceux-ci peuvent alors être libérés. Ce temps de traitement du ramasse-miettes est alors proportionnel au nombre d'objets présents dans la mémoire de la JVM. Ce nombre peut facilement être très important et ainsi dégrader les performances car durant cette opération l'exécution de tous les threads doit être interrompue.

65.2. Le fonctionnement du ramasse-miettes de la JVM Hotspot

Le fonctionnement du ramasse-miettes de la JVM Hotspot évolue au fur et à mesure de ses versions et repose sur plusieurs concepts :

- l'utilisation de générations
- plusieurs algorithmes de GC sont proposés dans le but de toujours améliorer les performances selon les besoins

L'idée est toujours de réduire la fréquence des invocations et les temps de traitements du ramasse-miettes.

De nombreux paramètres permettent de configurer le comportement du ramasse-miettes de la JVM.

Depuis Java 1.2, le ramasse-miettes implémente plusieurs algorithmes et utilise la notion de génération. Les ingénieurs ont constaté que d'une façon générale, il y a deux grandes typologies d'objets créés dans une application :

- les objets à durée de vie courte : exemple des objets créés dans les traitements d'une méthode
- les objets à durée de vie longue

La notion de génération est issue de l'observation du mode de fonctionnement de différentes typologies d'applications relativement à la durée de leurs objets. Ainsi, il a été constaté que de nombreux objets avaient une durée de vie relativement courte.

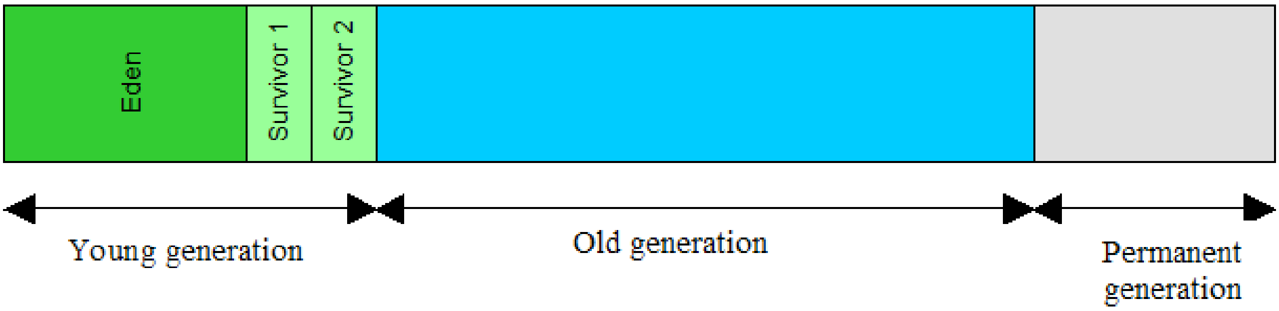
La notion de génération divise la mémoire de la JVM en différentes portions qui vont contenir des objets en fonction de leur âge. Une grande majorité des objets sont créés dans la generation des objets jeunes (Young Generation) et meurent dans cette génération.

Ainsi le tas est découpé en générations dans lesquelles les objets sont passés au fur et à mesure de l'allongement de leur durée de vie :

- Young Generation : tous les objets sont créés dans cette generation. L'opération de libération de la mémoire dans cette génération est désignée par le terme collection mineure. Comme le nombre objets qui n'ont plus de référence est important, le temps nécessaire à leur libération est très rapide. Après avoir survécu à plusieurs collections mineures, l'objet est promu dans la Old Generation
- Old Generation (Tenured Generation) : tous les objets de cette génération ont été promus de la Young Generation. L'opération de libération de la mémoire dans cette génération est désignée par le terme collection majeure (full collection). Le temps nécessaire à une collection majeure est beaucoup plus long : c'est pour cette raison que l'exécution est généralement limitée à un besoin absolu comme le manque d'espace dans la Tenured Generation

La JVM dispose aussi d'une troisième génération nommée Permanent Generation qui contient des données nécessaires au fonctionnement de la JVM comme par exemple la description de chaque classes et le code de chaque méthodes.

Sauf pour l'algorithme throughput collector, le découpage de la mémoire de la JVM est généralement sous la forme ci-dessous



La Young Generation est composée de trois parties :

- Eden : les objets créés le sont dans cette partie.
- deux parties nommées Survivor : ces deux parties sont alternativement remplies à chaque collection mineure

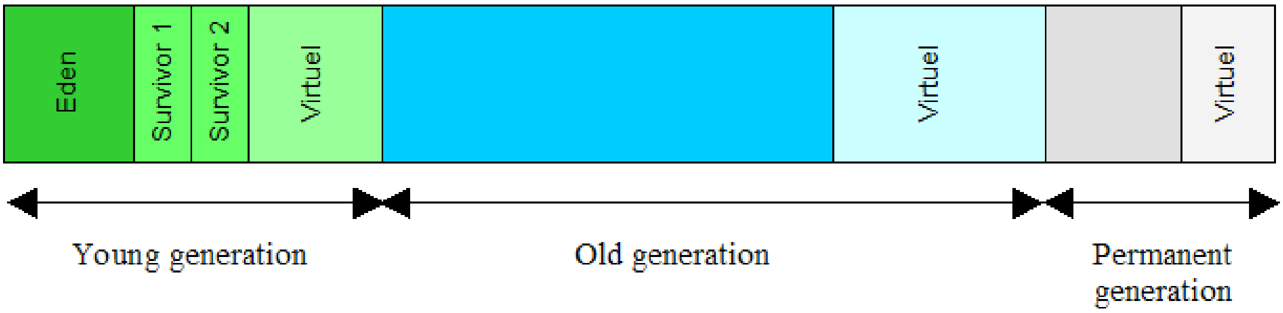
La taille de la Young Generation doit être suffisante pour permettre à un maximum d'objets d'être libérables entre deux collections mineures.

65.2.1. Les trois générations utilisées

La mémoire d'une JVM Hotspot est composée de trois générations :

- Young Generation : la plupart des objets sont instanciés dans cette génération
- Old (Tenured) Generation : contient les objets qui ont survécu à plusieurs collections de la Young Generation et qui ont été promus dans cette génération. Cette génération peut aussi contenir des objets de grandes tailles directement créés dans cette génération
- Permanent Generation : contient des objets nécessaires au fonctionnement de la JVM tels que la description des classes et méthodes et les classes et méthodes elles-mêmes

L'organisation des générations est généralement la suivante (sauf pour l'algorithme parallel collector)



Au démarrage de la JVM, tout l'espace mémoire maximum n'est pas physiquement alloué et l'espace de mémoire utilisable en cas de besoin est dit virtuel.

L'espace mémoire utilisé pour stocker les instances d'objets est nommé tas (heap). Il est composé de la Young Generation et Tenured Generation

La Young Generation est composée de plusieurs espaces :

- un espace principal nommé Eden : les objets sont créés dans cet espace sauf ceux de grandes tailles
- deux espaces plus petits nommés Survivor 1 (to) et 2 (from) : un espace Survivor contient les objets qui ont survécus à au moins une collection mineure et qui peuvent donc potentiellement être promus dans la Old Generation. L'un des deux espaces Survivor est toujours vide pendant que l'autre est utilisé par une collection. Ils changent alternativement leur rôle.

Lorsque la Young Generation est remplie, une collection mineure est exécutée par le ramasse-miettes. Une collection mineure peut être optimisée puisqu'elle part du prérequis que l'espace occupé par la plupart des objets de la Young Generation va être récupéré lors d'une collection mineure. Comme la durée d'une collection dépend du nombre d'objets utiles, une collection mineure doit s'exécuter rapidement.

Un objet toujours vivant après plusieurs collections mineures est promu dans la Tenured Generation. Si la Tenured Generation est remplie, une collection majeure est exécutée par le ramasse-miettes. Lors d'une collection majeure, le ramasse-miettes opère sur l'intégralité du tas (Young et Tenured Generation). Généralement une collection majeure est beaucoup plus longue qu'une collection mineure puisqu'elle implique beaucoup plus d'objets à traiter.

L'allocation d'objets est aussi soumise à des problématiques multithreads puisque plusieurs threads peuvent demander l'allocation d'objets. Pour tenir compte de ces contraintes et ne pas dégrader les performances, la JVM HotSpot réserve à chaque thread une zone de mémoire de l'espace Eden nommée Thread Local Allocation Buffer (TLAB) dans laquelle les objets du thread sont créés. Une synchronisation est cependant nécessaire si le TLAB est plein et qu'il faut en allouer un supplémentaire au thread. Des fonctionnalités sont mises en place pour limiter l'espace inutilisé des TLAB.

Lorsque la Old Generation ou la Permanent Generation se remplit, une collection majeure est effectuée, impliquant une collection sur toutes les générations.

De même, si l'espace libre de la Old Generation est insuffisant pour stocker les objets promus de la Young Generation, alors l'algorithme de collection de la Old Generation est appliqué sur l'intégralité du tas.

La mémoire de la JVM contient une section nommée génération permanente (Perm Gen). La JVM stocke dans cet espace les classes et leurs méthodes.

Du point de vue d'une collection les instances et les classes sont considérées comme des objets puisque ces deux types d'entités ont une représentation similaire dans la JVM. Le stockage dans la Permanent Generation des classes se justifie par le fait que les classes sont généralement des objets ayant une durée de vie relativement longue. Ainsi pour limiter le travail du ramasse-miettes et pour séparer les entités applicatives de celles de la JVM, les classes sont stockées dans cette generation dédiée.

Les principaux objets qui sont stockés dans la Permanent Generation sont :

- les méthodes des classes ainsi que leur bytecode
- les données lues du fichier .class (exemple : constant pool)
- des objets internes utilisés par la JVM
- ...

Les entités stockées dans cet espace ne sont pas vraiment permanentes sauf si l'option -noclassgc est utilisée lors du lancement de la JVM.

La taille de l'espace Permanent est indépendante de la taille du tas.

Des applications qui chargent de nombreuses classes ont besoin d'un espace plus important pour l'espace Permanent que celui proposé par défaut. La taille de l'espace Permanent peut être précisée en utilisant l'option -XX:PermSize au lancement de la JVM. La valeur fournie à ce paramètre sera utilisée pour définir la taille de l'espace Permanent au lancement de la JVM. Il est possible de définir la taille maximale de l'espace Permanent en utilisant l'option -xx:MaxPermSize au lancement de la JVM.

Si la taille de l'espace Permanent n'est pas assez importante pour les besoins de la JVM, une exception de type OutOfMemoryError est levée avec dans son message une référence au PermGen.

Actuellement, les collections sur la Permanent Generation sont toujours de type serial. Lorsqu'une collection est effectuée sur la Permanent Generation, elle est toujours réalisée avant celle de la Tenured Generation. Un objet de la Permanent Generation ne change jamais de generation suite à une collection.

65.2.2. Les algorithmes d'implémentation du GC

La version 1.5 de Java SE propose quatre algorithmes d'implémentation pour le ramasse-miettes :

- serial collector : c'est l'algorithme utilisé par défaut qui répond à la majorité des besoins standards pour de petites applications. Ses traitements utilisent un seul thread en mode stop the world
- parallel collector ou throughput collector : cet algorithme est le même que le Serial Collector mais il utilise une version qui parallélise les traitements de la Young Generation. Il est recommandé pour une application multithreadée exécutée sur une machine avec beaucoup de mémoire et plusieurs CPU comme des serveurs d'applications.
- parallel compacting collector : cet algorithme parallélise les traitements de la Tenured Generation (depuis Java 5u6, amélioré en Java 6)
- concurrent mark sweep collector (CMS) ou concurrent low pause collector : une grande partie des traitements sur la Tenured Generation se fait de façon concurrente avec l'application, ceci réduit les temps de pause de l'application.

Remarque : L'algorithme incremental low pause collector ou train collector n'est plus supporté depuis la version 1.4.2 de Java.

Tous ces algorithmes reposent sur l'utilisation de générations.

65.2.2.1. Le Serial Collector

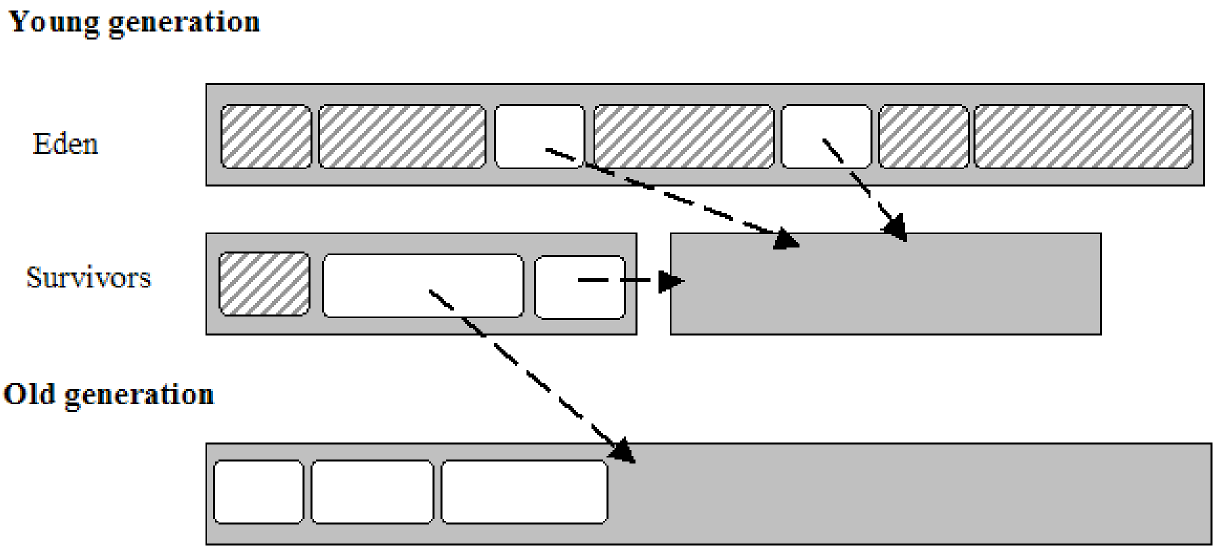
Avec un algorithme de type Serial Collector, les collections de la Young et Tenured Generation sont faites de manière séquentielle, par un seul processeur, à la façon stop the world : ainsi l'exécution de l'application est suspendue durant l'exécution des collections.

65.2.2.1.1. L'utilisation du serial collector dans la Young Generation

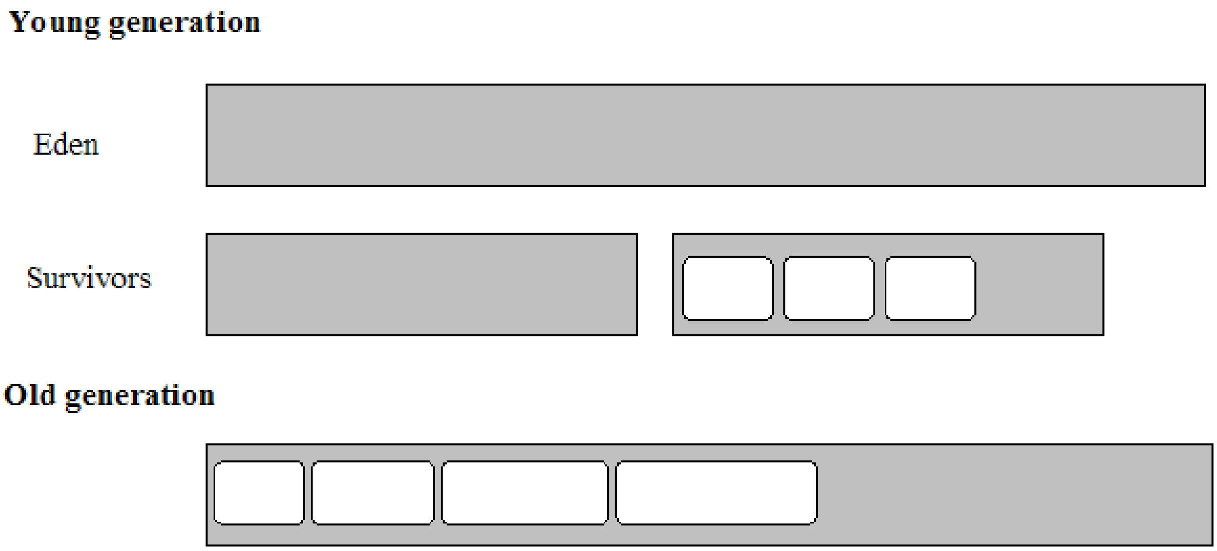
Les objets utilisés sont déplacés de l'espace Eden vers l'espace Survivor qui est vide. Les objets qui sont trop gros pour être déplacés dans l'espace Survivor sont directement déplacés dans la Tenured Generation.

Les objets les plus jeunes de l'espace Survivor rempli (Survivor From) sont déplacés dans l'autre espace Survivor en cours de remplissage (Survivor To). Les objets les plus anciens sont déplacés dans la Tenured Generation. A la fin de la collection l'espace Survivor qui était rempli doit être vide. Si l'espace Survivor en cours de remplissage ne peut plus recevoir d'objets alors tous les objets sont directement promus dans la Tenured Generation sans tenir compte de leur âge.

Après le déplacement des objets en cours d'utilisation, tous les objets qui restent dans l'espace Eden et l'espace Survivor qui était rempli sont des objets inutilisés dont l'espace peut être récupéré.



A la fin de la collection, l'espace Eden et l'espace Survivor initialement rempli sont vides. Ainsi dans la Young Generation, seul l'espace Survivor qui a été rempli contient encore des objets. Les deux espaces Survivor échangent leur rôle pour la prochaine collection.

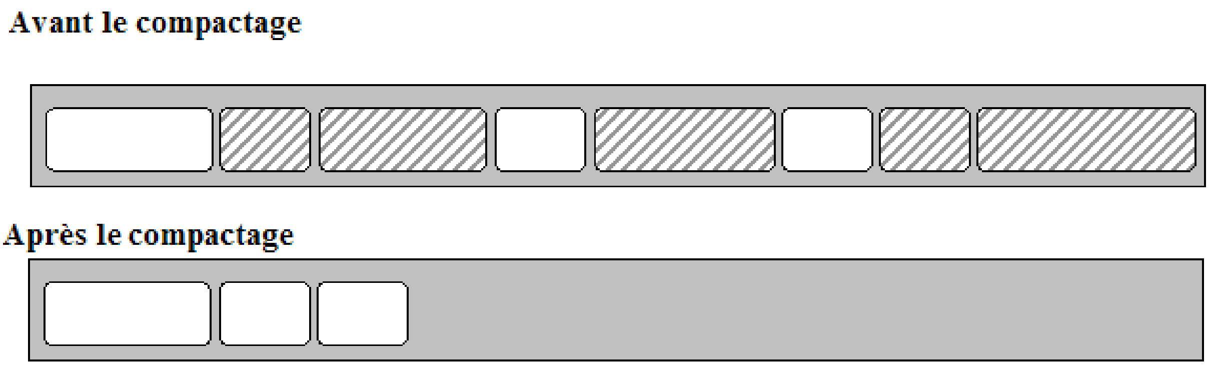


65.2.2.1.2. L'utilisation du Serial Collector dans la Tenured Generation

Le Serial Collector utilise un algorithme de type Mark/Sweep/Compact pour traiter la Tenured Generation et la Permanent Generation :

- Durant l'étape Mark, le ramasse-miettes identifie les objets qui sont encore utilisés
- Durant l'étape Sweep, le ramasse-miettes identifie les objets qui ne sont plus utilisés, en fait ceux qui n'ont pas été marqués lors de l'étape précédente, et récupère la mémoire qu'ils occupent.
- Durant l'étape Compact, le ramasse-miettes compacte la mémoire en regroupant tous les objets utilisés au début de la mémoire de la Tenured Generation. Le compactage de la mémoire permet de rendre la création d'objets dans la Tenured Generation plus rapide.

Un traitement similaire est effectué dans la Permanent Generation.



65.2.2.1.3. Le choix de l'utilisation du Serial Collector

Le Serial Collector est recommandé pour les applications clientes : ces applications sont généralement peu gourmandes en mémoire et le Serial Collector peut facilement effectuer un full GC en moins d'une demi seconde.

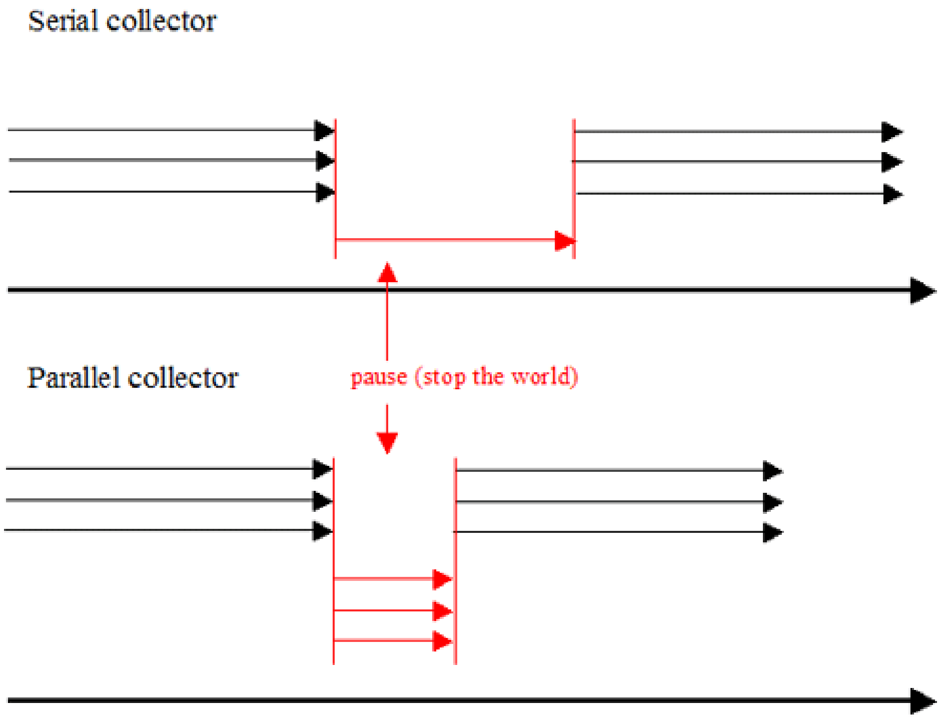
Depuis la version 5 de Java, le Serial Collector est choisi par défaut sur des machines de type client. Pour demander l'utilisation du Serial Collector sur des machines de type server, il faut utiliser l'option -XX:+UseSerialGC

65.2.2.2. Le Parallel Collector ou Throughput Collector

Pour profiter des avantages offerts par des machines disposant de plusieurs coeurs ou de plusieurs CPU, le Parallel Collector, aussi appelé Throughput Collector, a été développé. Il permet de réaliser les traitements de la collection en utilisant plusieurs CPU au lieu d'un seul.

65.2.2.2.1. L'utilisation du Parallel Collector dans la Young Generation

Le Parallel Collector utilise une version multithreads de l'algorithme utilisé par le Serial Collector : c'est toujours un algorithme de type stop the world avec déplacement des objets selon leur ancienneté mais ces traitements sont réalisés en parallèle par plusieurs threads. Le temps de ces traitements est ainsi réduit.



Java 5 propose quelques options supplémentaires pour configurer le comportement souhaité du parallel collector notamment en ce qui concerne le temps maximum de pause de l'application et le throughput.

Le temps maximum de pause de l'application souhaité peut être précisé avec l'option -XX:MaxGCPauseMillis=n où n représente une valeur en millisecondes. Le Parallel Collector va tenter de respecter ce souhait en procédant à des ajustements de paramètres mais il n'y a aucune garantie sur sa mise en oeuvre. Par défaut, aucun temps maximum de pause n'est défini.

Le throughput souhaité peut être exprimé avec l'option -XX:GCTimeRatio=n où n entre dans le calcul du ratio entre le temps du ramasse-miettes et le temps de l'application selon la formule 1 / (1 + n).

Exemple

-XX:GCTimeRatio=19 correspond à 5% (1 / (1 + 19)) du temps pour le ramasse-miettes

La valeur par défaut est 99, ce qui représente 1% du temps pour le ramasse-miettes.

Si ce souhait n'est pas atteint, l'algorithme va agrandir la taille des générations pour allonger le délai entre deux collections.

Les priorités dans les souhaits pris en compte par l'algorithme sont dans l'ordre :

- le temps maximum de pause
- le throughput
- l'empreinte mémoire

Pour être pris en compte, les souhaits précédents doivent être atteints avant que l'algorithme ne tente de réaliser le souhait suivant.

Le Parallel Collector utilise les générations et un algorithme similaire à celui du Serial Collector pour le traitement de la Young Generation mais il parallélise ses traitements sur plusieurs threads. Par défaut, le Parallel Collector utilise autant de threads que de processeurs.

Sur une machine avec un seul processeur, le Parallel Collector est moins performant que le Serial Collector notamment à cause du coût de synchronisation des traitements.

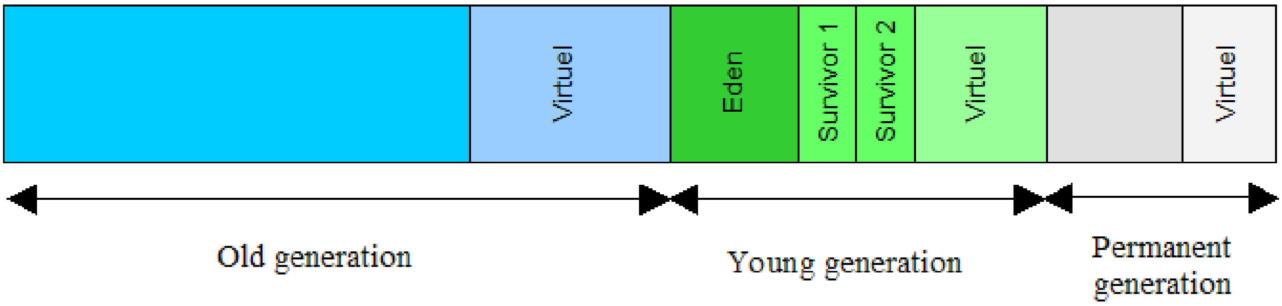
Sur une machine avec deux processeurs, le Parallel Collector et le Serial Collector ont des performances similaires.

Le gain en performance sur le temps des collections mineures croît sur des machines avec plus de deux processeurs.

Le nombre de threads utilisés peut être précisé explicitement en utilisant -XX:ParallelGCThreads=n ou n correspond au nombre de threads.

Comme plusieurs threads sont utilisés pour réaliser une collection mineure, il est possible que la Tenured Generation se fragmente. Chaque thread effectue la promotion d'un objet de la Young Generation vers la Tenured Generation dans une portion de cette dernière qui lui est dédiée

Les générations sont organisées de façon particulière dans le tas de la JVM.



A la fin de chaque collection, l'algorithme met à jour ses statistiques et test si les souhaits sont atteints. Si ce n'est pas le cas, l'algorithme va modifier les paramètres du ramasse-miettes et la taille du tas et des générations pour tenter d'atteindre les souhaits exprimés.

Par défaut, la taille d'une generation est augmentée de 20% et réduite de 5%. Ces valeurs peuvent être modifiées en utilisant plusieurs options :

- -XX:YoungGenerationSizeIncrement permet de préciser le pourcentage d'augmentation de la Young Generation
- -XX:TenuredGenerationSizeIncrement permet de préciser le pourcentage d'augmentation de la Tenured Generation
- -XX:AdaptiveSizeDecrementScaleFactor permet de préciser un pourcentage de réduction de la taille des générations sous la forme taille d'incrémentation / n

Les demandes explicites d'exécution du ramasse-miettes ne rentrent pas dans le calcul des statistiques.

Si le temps maximum de pause souhaité n'est pas atteint, la taille d'une seule generation est réduite à la fois (celle dont le temps de pause a été le plus long).

Si le throughput souhaité n'est pas atteint, la taille des deux générations est augmentée.

Si la taille minimale et maximale du tas ne sont pas précisées au lancement de la JVM alors elles sont déterminées en fonction de la taille de la mémoire physique de la machine. Par défaut, la taille minimale est égale à la taille de la mémoire / 64. Par défaut, la taille maximale est égale à la plus petite valeur entre taille de la mémoire / 4 et 1 Go.

Le parallel collector lève une exception de type OutOfMemoryError si plus de 98% du temps est passé à exécuter le ramasse-miettes et que moins de 2% du tas est libéré. Cette sécurité peut être désactivée en utilisant l'option -XX:-UseGCOverheadLimit

65.2.2.2.2. L'utilisation du Parallel Collector dans la Tenured Generation

L'algorithme utilisé par le Parallel Collector dans la Tenured Generation est identique à celui utilisé par le Serial Collector (Mark-Sweep-Compact) réalisé par un seul CPU. Ainsi le temps nécessaire à d'éventuels full GC peut être long.

65.2.2.2.3. Le choix de l'utilisation du Parallel Collector

Le Parallel Collector est intéressant pour des applications exécutées sur une machine avec plusieurs CPU n'ayant pas de contrainte forte sur les temps de pause liés au GC (exemple : des applications de type batch).

L'utilisation du Parallel Collector est intéressante pour des machines disposant de plusieurs processeurs. Le Serial Collector n'utilise qu'un seul thread pour effectuer ses traitements sur la Young Generation alors que le Parallel Collector en utilise plusieurs pour faire les mêmes traitements. Ceci est particulièrement intéressant pour des applications qui utilisent beaucoup de threads et instancient beaucoup d'objets car le temps de traitement du ramasse-miettes pour la Young Generation est réduit.

65.2.2.3. Le Parallel Compacting Collector

Le Parallel Compacting Collector est utilisable depuis la version 5 update 6 de Java. Par rapport au Parallel Collector, le Parallel Compacting Collector utilise un algorithme différent et multithread pour le traitement de la Tenured Generation.

65.2.2.3.1. L'utilisation du Parallel Compacting Collector dans la Young Generation

L'algorithme utilisé par le Parallel Compacting Collector dans la Young Generation est identique à celui utilisé par le Parallel Collector.

65.2.2.3.2. L'utilisation du Parallel Compacting Collector dans la Tenured Generation

L'algorithme utilisé par le Parallel Compacting Collector dans la Tenured Generation est de type stop the world avec compactage de la mémoire en utilisant plusieurs CPU pour paralléliser ses traitements.

Les traitements du Parallel Compacting Collector comporte trois étapes :

- marking : chaque génération est découpée de façon logique en région de tailles fixes : plusieurs threads travaillent en parallèle pour marquer les objets utilisés de chaque région. Pour chaque objet marqué, des informations sont stockées dans la région
- summary : les régions sont vérifiées pour déterminer si elles doivent être compactées selon leur densité de remplissage
- compaction : plusieurs threads déplacent les objets pour remplir les régions qui doivent être compactées afin de compacter les objets du tas

65.2.2.3.3. Le choix de l'utilisation du Parallel Compacting Collector

L'utilisation de cet algorithme est intéressante lorsque la machine dispose de plusieurs CPU. Par rapport au Parallel Collector, il permet de réduire les temps de pause de l'application.

Pour utiliser le Parallel Compacting Collector, il faut utiliser l'option -XX:+UseParallelOldGC de la JVM.

Le nombre de threads utilisés pour les traitements en parallèle peut être limité en utilisant l'option -XX:ParallelGCThreads=n. Ceci peut être utile notamment sur de gros serveurs afin que la JVM ne monopolise pas trop de CPU.

65.2.2.4. Le Concurrent Mark-Sweep (CMS) Collector

De nombreuses applications ont besoin de la meilleure réactivité possible. Généralement, la collection de la Young Generation est assez rapide. Par contre la collection de la Tenured Generation peut être assez longue d'autant que cette durée est en relation avec la taille du tas. L'algorithme CMS collector aussi nommé low latency collector permet de réduire la durée d'une collection de la Tenured Generation en effectuant une partie de ses traitements de façon concurrente avec ceux de l'application.

Le CMS collector a un mode de fonctionnement similaire au Serial Collector mais certains traitements réalisés dans la Tenured Generation sont faits dans un ou plusieurs threads dédiés, donc de façon concurrente à l'exécution de l'application. Le but est de réduire les temps de pause de l'application qui sont requis pour les collections de la Tenured Generation.

65.2.2.4.1. L'utilisation du CMS collector dans la Young Generation

L'algorithme utilisé par le CMS collector dans la Young Generation est identique à celui utilisé par le Parallel Collector (plusieurs threads sont utilisés pour réaliser les traitements).

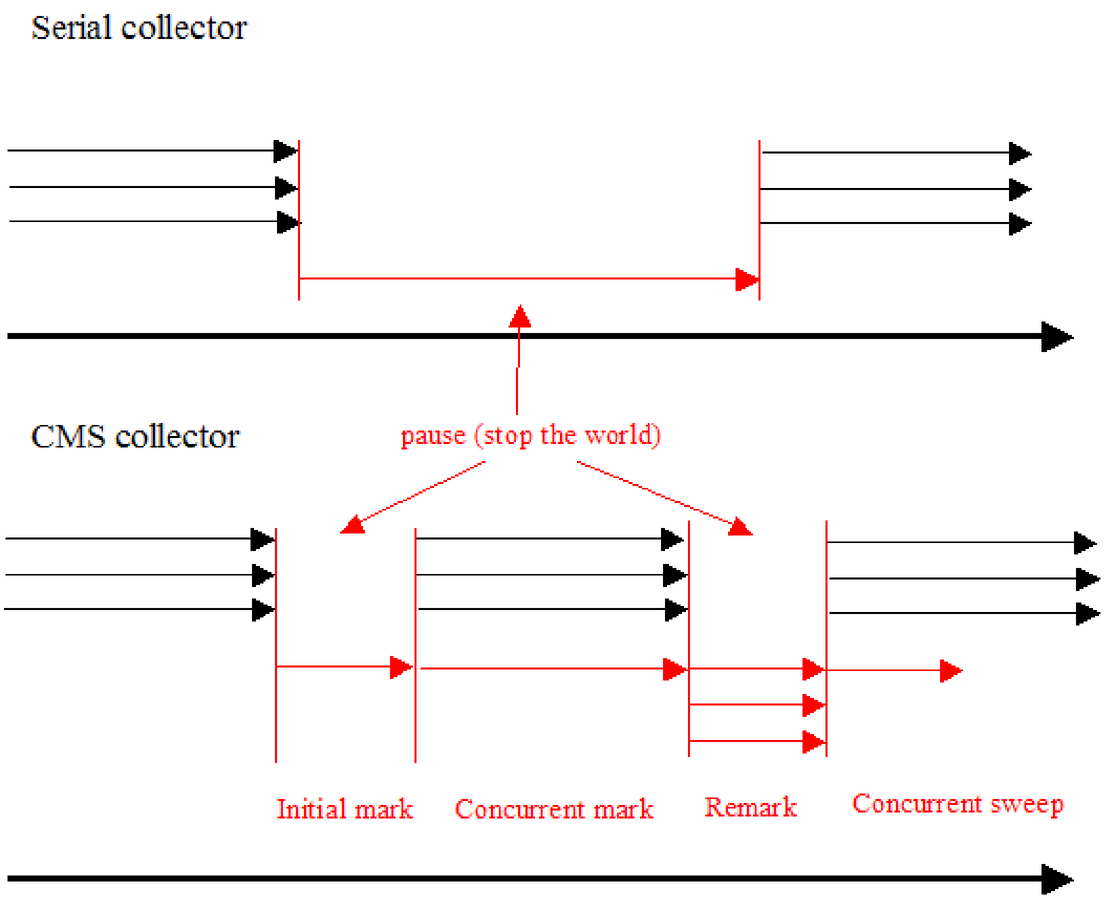
65.2.2.4.2. L'utilisation du CMS collector dans la Tenured Generation

Une collection réalisée par le CMS collector comporte plusieurs étapes :

- initial mark : cette étape qui met en pause l'application permet de déterminer un ensemble d'objets initiaux utilisés par l'application
- concurrent marking : cette étape permet de marquer, de façon concurrente à l'exécution de l'application, les objets qui sont utilisés par l'ensemble d'objets initiaux. Comme l'application est en cours d'exécution, les objets vivent et il n'y a donc aucune garantie sur le fait que tous les objets utiles aient été marqués à la fin de cette étape

- remark : cette étape qui met en pause l'application permet de reparcourir les objets modifiés durant l'étape précédente pour s'assurer que tous les objets utiles soient marqués. Les traitements de cette étape sont réalisés par plusieurs threads
- concurrent sweep : toute la mémoire des objets inutiles est récupérée. Les traitements de cette étape sont réalisés par un seul thread
- recalcul de la taille du tas, des statistiques et des données pour la prochaine collection

Le schéma ci-dessous illustre le fonctionnement du Serial Collector et du CMS collector.



L'étape initial mark est une pause relativement courte. Le temps des traitements concurrents peut être relativement long. Le temps de traitements de la seconde étape provoquant une pause, nommée remark, dépend de l'activité de modification des objets par l'application durant l'étape concurrent mark.

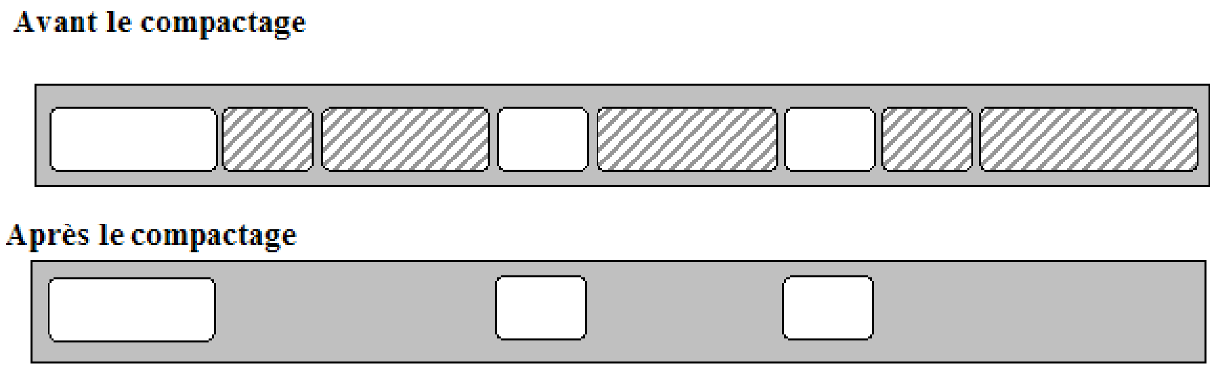
L'exécution concurrente de certains traitements du ramasse-miettes avec l'application consomme des ressources CPU qui ne sont pas affectées à cette dernière. Les traitements du ramasse-miettes effectués en concurrence sont réalisés avec un seul thread. Sur une machine mono processeur, cela va réduire les performances de l'application tendant à ne pas apporter de gain. Sur une machine bi processeur, un processeur est dédié à l'application, l'autre au thread du ramasse-miettes. Plus le nombre de processeurs est important dans la machine, meilleur est le gain en utilisant le CMS Collector.

Il est possible de demander l'utilisation du mode incremental dans lequel les traitements réalisés en concurrence avec l'application sont faits de façon incrémentale sur une petite période entre chaque collection sur la Young Generation. Ceci permet de donner plus de temps de traitement à l'application. Ce mode peut être utile notamment sur des machines ayant un ou deux processeurs.

Les traitements du CMS Collector dans la Tenured Generation sont censés être réalisés avant que cette génération ne soient pleine. Il peut arriver que la génération soit remplie avant la fin des traitements : dans ce cas, l'application est mise en pause pour permettre l'exécution complète des traitements du ramasse-miettes.

L'exécution de certains traitements en parallèle implique une surcharge de travail pour l'algorithme notamment durant l'étape Remark.

Le CMS collector est le seul algorithme qui ne compacte pas la mémoire après la libération des objets inutilisés. Ceci permet d'économiser du temps de traitements lors des collections mais complexifie l'allocation de mémoire pour de nouveaux objets. Dans le cas d'un compactage, il est facile de connaître le prochain espace mémoire à utiliser puisqu'il correspond à la première adresse mémoire libre de la génération. Sans compactage, il faut gérer une liste des espaces de mémoire disponible (adresse et quantité de mémoire continue). A chaque instanciation, il faut rechercher dans la liste un espace mémoire adéquat et mettre à jour la liste ce qui rend l'instanciation d'un objet dans la Tenured Generation plus lente. Ce mécanisme nécessite donc aussi plus d'espace mémoire dans la JVM.



Ceci ralentit aussi les traitements de collection sur la Young Generation puisque la plupart des allocations de mémoire dans la Tenured Generation sont réalisées par les collections lors de la promotion des objets de la Young Generation.

De part son mode de fonctionnement, le CMS collector requiert plus de mémoire que les autres collectors pour son propre usage mais aussi parce que l'application peut créer de nouveaux objets pendant l'exécution d'une partie des traitements du ramasse-miettes.

Bien que l'algorithme garantisse que tous les objets utilisés soient marqués, il est possible que certains objets marqués soient devenus inutilisés du fait de l'exécution en concurrence de l'application. Dans ce cas, l'espace mémoire de ces objets ne sera pas récupéré durant la collection en cours mais le sera à la prochaine collection. L'ensemble de ces objets est nommé floating garbage.

Pour limiter la fragmentation de la mémoire liée au fait que la génération n'est pas compactée, le CMS collector peut fusionner des espaces de mémoires contigus devenus disponibles.

Contrairement aux autres algorithmes de collections, le CMS collector n'attend pas que la génération soit pleine pour commencer une collection. Il tente d'anticiper son exécution afin d'éviter que cela ne survienne sinon son temps de traitement serait supérieur à celui d'un serial ou parallel collector. Avec un algorithme de type CMS collector, une collection doit être démarrée de telle sorte que les traitements de la collection soient terminés avant que la Old Generation ne soit entièrement remplie.

Le démarrage de la collection peut être lancé selon deux facteurs :

- des statistiques sur le temps d'exécution des précédentes collections et de remplissage de la Tenured Generation
- à partir d'un certain pourcentage de remplissage de la Tenured Generation. Ce pourcentage qui par défaut est de 68 peut être modifié en utilisant l'option -XX:CMSInitiatingOccupancyFraction=n

Le CMS collector calcule des statistiques basées sur le fonctionnement de l'application pour tenter d'estimer le temps nécessaire avant que la Old Generation ne soit remplie et le temps nécessaire aux cycles pour effectuer la collection.

Sur la base des statistiques, les cycles de traitements de la collection sont démarrés avec pour objectif que ceux-ci soient terminés avant que la Old Generation ne soit pleine.

Il est possible que les estimations provoquent un démarrage trop tardif de la collection ce qui fait rentrer l'algorithme dans un mode nommé concurrent mode failure qui est très coûteux car les temps de pauses sont alors beaucoup plus longs, ce qui a un effet inverse à celui escompté par l'algorithme sur l'exécution de l'application.

Malheureusement, le calcul de statistiques sur des traitements ayant eu lieu n'est pas toujours le reflet de ce qui se passe ou va se passer. Si de trop nombreux full garbages sont exécutés les uns à la suite des autres, il est possible de modifier plusieurs paramètres pour tenter d'y remédier :

1. augmenter la valeur du paramètre -XX:CMSIncrementalSafetyFactor
2. augmenter la valeur du paramètre -XX:CMSIncrementalDutyCycleMin

3. désactiver le calcul automatique de la durée des cycles avec l'option -XX:-CMSIncrementalPacing et donner une durée fixe à la durée des cycles de traitements avec - XX:CMSIncrementalDutyCycle

Remarque : ces différentes modifications doivent être faites les unes à la suite des autres, dans l'ordre indiqué jusqu'à ce que le problème disparaisse.

Le traitement de la Young Generation peut avoir lieu en concurrence avec le traitement de la Old Generation. Comme le traitement de la Young Generation est similaire à celui utilisé par le Parallel Collector en utilisant un algorithme de type stop the world, les threads de traitements de la Tenured Generation sont interrompus

Les pauses liées à une collection sur la Young Generation et la Old Generation sont indépendantes mais peuvent survenir l'une à la suite de l'autre ce qui peut donc allonger le temps de pause de l'application. Pour éviter ce phénomène, l'algorithme tente d'exécuter l'étape remark entre deux pauses liées à la collection de la Young Generation.

La ramasse-miettes CMS est deprecated en Java 9.

65.2.2.4.3. L'utilisation du mode incrémental

Le mode incremental permet d'utiliser le CMS collector sur des machines avec un seul processeur.

Par défaut, l'algorithme du CMS collector utilise un ou plusieurs threads pour exécuter ses traitements concurrents en dédiant ces threads à ses activités puisque l'algorithme est prévu pour fonctionner sur des machines avec plusieurs CPU.

L'utilisation de cet algorithme peut cependant être intéressante sur des machines avec uniquement un ou deux processeurs. Dans ce cas, l'algorithme propose le mode incremental "i-cms" qui découpe les traitements concurrents de l'algorithme en plusieurs morceaux (duty cycle) exécutés entre les pauses des collections mineures. En dehors de ces cycles, le ou les threads sont suspendus pour permettre au processeur d'exécuter d'autres threads. Le temps d'exécution des cycles est calculé par défaut par l'algorithme en fonction du comportement de l'application dans la JVM (automatic pacing).

Ainsi, le mode incrémental permet de réduire l'impact des traitements concurrents sur l'application en rendant périodiquement la main au processeur pour exécuter l'application. Ces traitements sont découpés en petites unités qui sont exécutées entre deux collections sur la Young Generation.

L'option -XX:+CMSIncrementalMode permet de demander l'utilisation du mode incrémental : dans ce mode, les traitements réalisés de façon concurrente partagent leur temps d'exécution selon des cycles interrompus par un retour à l'exécution de l'application par le processeur. Le temps d'exécution alloué à un cycle est défini par un pourcentage de temps du processeur accordé pour la collection. Ce pourcentage peut être précisé comme attribut fourni à la JVM ou calculé par l'algorithme en fonction du comportement de l'application.

L'option -XX:+CMSIncrementalPacing de la JVM permet de demander de calculer le temps du cycle en fonction de statistiques issues du comportement de l'application. Par défaut, cette option n'est pas activée en Java 5 et est activée en Java 6.

L'option -XX:CMSIncrementalDutyCycle permet de préciser le pourcentage du temps accordé pour les traitements de l'algorithme entre deux collections mineures. Si l'option -XX:+CMSIncrementalPacing est activée alors cela représente la valeur initiale. La valeur par défaut est 50 en Java 5, ce qui est généralement trop, et 10 en Java 6.

L'option -XX:CMSIncrementalDutyCycleMin permet de préciser le pourcentage du temps minimum accordé pour les traitements de l'algorithme lorsque l'option -XX:+CMSIncrementalPacing est activée. La valeur par défaut est 10 en Java 5 et 0 en Java 6.

Les options recommandées pour i-cms avec Java 5 sont :
-XX:+UseConcMarkSweepGC
-XX:+CMSIncrementalMode
-XX:+CMSIncrementalPacing
-XX:CMSIncrementalDutyCycleMin=0
-XX:CMSIncrementalDutyCycle=10
-XX:+PrintGCDetails
-XX:+PrintGCTimeStamps

Les options recommandées pour i-cms avec Java 6 sont :
-XX:+UseConcMarkSweepGC
-XX:+CMSIncrementalMode
-XX:+PrintGCDetails
-XX:+PrintGCTimeStamps

En fait, ce sont les mêmes options mais les valeurs par défaut, non précisées en Java 6, sont celles recommandées avec Java 5.

Le mode incrémental de CMS est deprecated en Java 8 et retiré en Java 9.

65.2.2.4.4. Le choix de l'utilisation du CMS collector

Cet algorithme réduit les temps de pause de l'application liés à son activité en exécutant une partie de celle-ci en concurrence avec l'exécution de l'application. Notamment la recherche initiale des objets utilisés est réalisée dans plusieurs threads. Cet algorithme a toujours besoin de temps de pauses mais leurs durées sont réduites grâce à l'exécution de certains traitements en concurrence avec l'application.

Le CMS collector est prévu pour être utilisé dans une JVM exécutant une application souhaitant avoir de faibles temps de pause et qui permette de partager ses ressources processeurs durant son exécution. Généralement, il s'agit d'pplications avec une grosse Tenured Generation exécutée sur une machine avec plusieurs processeurs. Mais cet algorithme peut aussi être utilisé pour des applications avec une Tenured Generation de petite taille, exécutée sur une machine mono processeur avec le mode incrémental activé.

Le CMS collector est recommandé pour des applications qui ont besoin de temps de pauses liés au ramasse-miettes les plus courts possibles et qui peuvent accepter d'avoir une partie des traitements du ramasse-miettes exécutés en concurrence avec elle. Dans les faits, ce sont des applications avec une Tenured Generation de tailles importantes, exécutée sur une machine avec plusieurs processeurs.

Les applications qui possèdent de nombreux objets ayant une durée de vie assez longue et qui s'exécutent sur une machine multiprocesseur peuvent tirer avantage de ce collector : c'est notamment le cas pour les serveurs ou conteneurs web.

Pour utiliser le i-CMS collector, il faut utiliser l'option -XX:+UseConcMarkSweepGC et pour demander l'utilisation du mode incremental, il faut utiliser l'option -XX:+CMSIncrementalMode.

65.2.3. L'auto sélection des paramètres du GC par la JVM Hotspot

Les types d'applications qui peuvent être développées en Java et exécutées dans une JVM sont nombreux allant d'une petite applet à une grosse application web ou d'entreprise.

Pour répondre aux besoins variés de ces différents types d'applications, la JVM Hotspot propose plusieurs algorithmes utilisables pour le ramasse-miettes. Depuis Java 5, la JVM définit des paramètres de configuration par défaut du ramasse-miettes en fonction du type de machine et du système d'exploitation. Cependant ces valeurs prédéfinies ne sont pas toujours optimales pour une application donnée et il est parfois nécessaire de définir une autre configuration explicitement.

Java 5 propose une fonctionnalité nommée ergonomics dont le but est de configurer certains éléments de la JVM pour permettre d'obtenir de bonnes performances sans configuration. Cette fonctionnalité repose sur un ensemble de règles qui définissent des valeurs par défaut pour :

- la taille du tas
- le ramasse-miettes
- le compilateur JIT

Ceci permet d'avoir automatiquement un léger tuning plutôt que d'avoir des valeurs par défaut identiques dans tous les contextes.

La définition de ces valeurs par défaut convient généralement pour des cas standard mais elle n'exclut pas d'avoir à configurer soi-même ces paramètres pour qu'ils correspondent mieux aux besoins de l'application.

A partir de Java 5, la JVM détermine, par défaut, plusieurs options de configuration pour le ramasse-miettes en fonction de la machine et du système d'exploitation. Ces valeurs par défaut conviennent généralement pour la majorité des applications.

Parmi les valeurs déterminées, il y a le mode de fonctionnement de la JVM HotSpot :

- server : pour une machine avec au moins 2Go de mémoire et plusieurs processeurs sauf pour les machines 32 bits exécutant un système Windows
- client : dans les autres cas

Les valeurs par défaut pour le mode client sont :

- utilisation du serial collector
- taille du tas minimal : 4 Mo
- taille du tas maximal : 64 Mo

Les valeurs par défaut pour le mode server sont :

- utilisation du parallel collector

Quel que soit le mode d'utilisation de la VM, si le parallel collector est utilisé les tailles du tas sont :

- taille du tas minimal : 1/64 de la mémoire physique limité à 1Go
- taille du tas maximal : 1/4 de la mémoire physique limité à 1Go

65.2.4. La sélection explicite d'un algorithme pour le GC

Depuis Java 5, avant de se lancer dans une adaptation personnalisée du GC, il faut étudier si la configuration par défaut déterminée par la JVM répond aux besoins. Si ce n'est pas le cas, il est possible de la modifier explicitement.

La meilleure méthodologie pour améliorer les performances est de mesurer, analyser, modifier et d'itérer sur ces trois étapes jusqu'à l'obtention d'un résultat satisfaisant.

Si la configuration par défaut définie par la JVM ne répond pas au besoin, une première amélioration peut être de modifier la taille du tas et des générations qu'il contient. Si cela ne convient toujours pas, il est possible d'essayer un autre algorithme pour le ramasse-miettes.

Le choix d'un algorithme pour le ramasse-miettes doit prendre en compte plusieurs facteurs, en particulier :

- la taille du tas
- le nombre et la durée de vie des objets
- le nombre de processeurs de la machine

Voici quelques exemples de recommandations :

Conditions	Algorithme recommandé
Si la taille du tas est inférieure à 100Mb	serial collector -XX:+UseSerialGC
Si la machine est mono processeur	serial collector -XX:+UseSerialGC
Sans contrainte sur les pauses de l'application	parallel collector -XX:+UseParallelGC ou parallel compacting collector -XX:+UseParallelOldGC
Limiter le plus possible les temps de pause de l'application	CMS collector (avec le mode incremental activé si la machine dispose d'un ou deux processeurs) -XX:+UseConcMarkSweepGC

Dans tous les cas, ces recommandations sont à tester pour valider si l'algorithme proposé répond aux besoins de l'application.

Certaines combinaisons d'algorithmes ne sont pas autorisées : dans ce cas la JVM ne démarre pas et affiche un message d'erreur explicite.

Résultat :

```
C:\java\tests>java -XX:+UseConcMarkSweepGC -XX:+UseSerialGC -jar test.jar
Conflicting collector combinations in option list; please refer to the release n
otes for the combinations allowed
Could not create the Java virtual machine.

C:\java\tests>java -XX:+UseConcMarkSweepGC -XX:+UseParallelGC -jar test.jar
Conflicting collector combinations in option list; please refer to the release n
otes for the combinations allowed
Could not create the Java virtual machine.
```

65.2.5. Demander l'exécution d'une collection majeure.

Généralement l'exécution du ramasse-miettes est conditionnée par un manque d'espace libre. Cela permet dans un premier temps de libérer de l'espace. Si cela ne suffit pas alors l'espace mémoire est agrandit jusqu'à atteindre le maximum défini.

La méthode gc() de la classe System permet de demander l'exécution du ramasse-miettes.

Cependant, le moment d'exécution du ramasse-miettes n'est pas facilement prédictible. Même l'appel à la méthode gc() de la classe System n'implique pas obligatoirement l'exécution du ramasse-miettes mais sollicite simplement une demande d'exécution.

Avec certains algorithmes du ramasse-miettes, forcer l'invocation du ramasse-miettes peut être préjudiciable sur les performances notamment pour de grosses applications. L'option -XX:+DisableExplicitGC de la JVM demande à la JVM d'ignorer les demandes explicites d'exécution du ramasse-miettes.

65.2.6. La méthode finalize()

Les Java Langage Specifications imposent au ramasse-miettes d'appeler la méthode de finalisation de l'objet héritée de la classe Object avant de libérer la mémoire. Ainsi, le ramasse-miettes a l'obligation par ces spécifications d'invoquer la méthode finalize() lorsque l'espace mémoire d'une instance d'un objet va être récupéré.

Il est alors facile d'imaginer de mettre des traitements de libération de ressources par exemple dans cette méthode puisque la JVM garantit l'appel de cette méthode lorsque l'objet n'est plus utilisé.

Malheureusement, seule l'invocation est garantie : l'exécution de cette méthode dans son intégralité ne l'est pas notamment si une exception est levée durant ses traitements.

De plus, le moment où l'espace mémoire va être récupéré et donc le moment où la méthode finalize() sera invoquée n'est pas prédictible.

Pour ces deux raisons, il ne faut surtout pas utiliser la méthode finalize() pour libérer des ressources.

Généralement une bonne pratique est de ne pas faire usage dans la mesure du possible de la méthode finalize() et de ne surtout pas utiliser le garbage collector pour faire autre chose que la libération de la mémoire.

En Java 9, la méthode finalize() dépréciée.

65.3. Le paramétrage du ramasse-miettes de la JVM HotSpot

La JVM Hotpsot propose de nombreux paramètres relatifs à l'activité du ramasse-miettes.

65.3.1. Les options pour configurer le ramasse-miettes

La JVM propose de nombreuses options pour configurer le ramasse-miettes : elles permettent notamment de modifier la taille du tas et des générations, choisir un algorithme, le configurer et obtenir des informations sur son exécution.

Plusieurs paramètres permettent de configurer la taille des différents espaces mémoire de la JVM.

L'allocation de la mémoire pour la JVM se fait au moyen de plusieurs options préfixées par -X

Option	Rôle
-Xms	Taille initiale du tas (heap)
-Xmx	Taille maximale du tas (heap)
-Xmn	Taille de la Young Generation du tas
-Xss	Taille de la pile (stack) de chaque thread. Si celle-ci est trop petite, une exception de type StackOverFlowError est levée Exemple : `-Xss1024k`

Remarque : sur certaines machines utilisant Linux, il est parfois nécessaire de modifier la taille de la pile au niveau des paramètres du système d'exploitation en utilisant la commande ulimit -s.

Plusieurs paramètres non standard sont proposés par la JVM Hotspot pour gérer la taille du tas et des générations.

Option	Description
-XX:MinHeapFreeRatio=*n*	Définir le ratio minimum d'espace libre du tas avant que sa taille ne soit agrandie. Exemple : si le ratio est de 20 et que le pourcent d'espace libre n'est plus que de 20% alors la taille du tas est agrandie pour qu'il y ait 20% d'espace libre.
-XX:MaxHeapFreeRatio=*n*	Définir le ratio maximum d'espace libre du tas avant que sa taille ne soit réduite. Exemple : si le ratio est de 70 et que le pourcent d'espace libre dépasse les 70% alors la taille du tas est réduite pour qu'il y est 70% d'espace libre.
-XX:NewSize=*n*	Définir la taille initiale de la Young Generation.
-XX:NewRatio=*n*	Définir le ratio entre la Young Generation et la Old Generation. Exemple : Si n vaut 3 alors le ratio est de 1:3. Ainsi la taille de la Young Generation est le quart de celle du tas.
-XX:SurvivorRatio=*n*	Définir le ratio entre les espaces Survivor et Eden dans la Young Generation Exemple : Si n vaut 3 alors le ratio est de 1:3. Ainsi la taille de la Young Generation est le quart de celle du tas. Exemple :

	si n vaut 6 alors le ratio est 1:8 de la Young Generation (ce n'est pas 1:7 car il y a deux espaces survivor).
-XX:MaxPermSize=*n*	Définir la taille maximale de la Permanent Generation

Il est tout à fait normal que la taille de la JVM observée sur le système soit supérieure à la taille fournie par l'option -Xmx. La valeur de cette option ne concerne que le tas et non tout l'espace mémoire de la JVM qui inclus aussi entre autres les piles (stack), la permanente generation, ...

Les valeurs à affecter aux options -Xms et -Xmx dépendent de l'application exécutée dans la JVM. La taille minimale doit supporter l'espace requis par l'application. Sur un poste utilisateur il est généralement préférable de mettre des valeurs minimales et maximales différentes pour limiter la consommation de mémoire sur la machine tout en lui permettant de grossir aux besoins. Sur un serveur, il est généralement préférable de mettre la même valeur car les ressources mémoire sont généralement moins limitées et cela évite les allocations de mémoire successives.

Les options pour choisir l'algorithme utilisées par le ramasse-miettes sont :

Option	Algorithme utilisé par le ramasse-miettes
-XX:+UseSerialGC	Serial collector
-XX:+UseParallelGC	Parallel collector
-XX:+UseParallelOldGC	Parallel compacting collector
-XX:+UseConcMarkSweepGC	Concurrent mark-sweep collector (CMS collector)

Les options pour afficher des informations sur l'exécution du ramasse-miettes sont :

Option	Description
-XX:+PrintGC	Afficher des informations de base à chaque exécution du ramasse-miettes
-XX:+PrintGCDetails	Afficher des informations détaillées à chaque exécution du ramasse-miettes
-XX:+PrintGCTimeStamps	Afficher la date-heure de début d'exécution du ramasse-miettes

Le paramètre -verbose:gc permet aussi d'afficher dans la console des informations sur chaque collecte effectuée par le ramasse-miettes.

Les options pour les algorithmes parallel et parallel compacting collector sont :

Option	Description
-XX:ParallelGCThreads=*n*	Nombre de threads utilisés par le ramasse-miettes (par défaut, c'est le nombre de CPU de la machine)
-XX:MaxGCPauseMillis=*n*	Demander au ramasse-miettes d'essayer de limiter son temps d'exécution à celui fourni en millisecondes en paramètre.
-XX:GCTimeRatio=*n*	Demander au ramasse-miettes d'essayer de respecter un ratio 1/(1+n) du temps total utilisé pour les activités du ramasse-miettes. La valeur par défaut est 99

Les principales options pour l'algorithme CMS collector sont :

Option	Description
-XX:+CMSIncrementalMode	Activer le mode d'exécution des traitements concurrents de façon incrémentale
-XX:+CMSIncrementalPacing	Activer le calcul automatique de statistiques pour déterminer le temps de traitement du ramasse-miettes dans le mode incrémental.

-XX:ParallelGCThreads=*n*	Nombre de threads utilisés par le ramasse-miettes pour le traitement de la Young Generation et des traitements concurrents de la old generation.

65.3.2. La configuration de la JVM pour améliorer les performances du GC

Le ramasse-miettes peut induire de véritables problèmes de performance pour certaines applications en fonction des besoins de celles-ci et du paramétrage de la JVM.

En cas de problème de performance avec le ramasse-miettes, si la taille du tas doit être modifiée, le plus souvent il faudra aussi adapter la taille de chacune des générations.

En cas de fuite de mémoire ou d'inadéquation de la taille du tas avec les besoins de l'application, il est possible que les performances de la JVM se dégradent très fortement car elle peut occuper une large partie de ses traitements à l'exécution du ramasse-miettes de façon répétée.

Une bonne adéquation entre les besoins de l'application et la configuration de la JVM peut permettre de réduire le temps nécessaire à l'exécution du ramasse-miettes durant l'exécution de l'application.

Plusieurs indicateurs peuvent être utilisés pour mesurer les performances du ramasse-miettes :

- throughput : pourcentage du temps consacré à l'exécution de l'application par la JVM
- pause : temps d'arrêt de l'application lié à l'exécution du ramasse-miettes
- footprint : espace mémoire consommé par la JVM
- promptness : temps entre le moment où un objet n'est plus utilisé et le moment où son espace mémoire est libéré

Il n'y a pas de règle absolue pour optimiser les performances du ramasse-miettes : cette optimisation est dépendante de l'application exécutée dans la JVM. Par exemple, dans une application standalone, il est très important d'avoir l'indicateur pause le plus court possible alors qu'il n'est généralement pas crucial pour une application de type web.

Les besoins relatifs aux performances du ramasse-miettes dépendent de la typologie des applications exécutées, par exemple :

- dans une application standalone, les temps de pause doivent être réduits au maximum
- dans une application web, c'est le throughput qui est important dans la mesure où les pauses peuvent être masquées par les temps de latence du réseau

Cette optimisation doit tenir compte des priorités données à chaque indicateur.

Par exemple, plus la taille de la Young Generation est importante, plus le throughput devrait s'améliorer mais l'empreinte mémoire et les temps de pause augmentent.

A l'inverse, plus la taille de la Young Generation est petite plus les temps de pause sont réduit mais au détriment du throughput.

Il n'y a donc pas qu'une façon d'optimiser la taille des générations mais il faut tenir compte des besoins et des exigences de l'application.

Plusieurs paramètres peuvent avoir une influence sur la taille des générations. Au lancement de la JVM, l'espace entier défini par le paramètre -Xmx du tas est réservé. Si la valeur du paramètre -Xms est plus petite que celle de -Xmx alors seule la quantité indiquée par -Xms est immédiatement disponible pour le tas. Le reste de la mémoire est dite virtuelle : elle sera utilisée au besoin.

Les différentes générations (young, tenured, permanent) peuvent ainsi grossir jusqu'à atteindre leur taille maximale respective. Certaines caractéristiques sont fournies sous la forme de ratio. Par exemple, le paramètre NewRatio définit la proportion de la taille de la young et Tenured Generation dans le tas.

La quantité de mémoire du tas gérée est un facteur important pour les performances du ramasse-miettes.

Par défaut, les traitements du ramasse-miettes peuvent faire grossir ou réduire la taille du tas lors de chaque collection afin de respecter la quantité de mémoire libre souhaitée précisée sous la forme de pourcentage par les paramètres -XX:MinHeapFreeRatio=<minimum> et -XX:MaxHeapFreeRatio=<maximum>

Attention : l'augmentation de la taille de mémoire du tas provoque généralement des effets de bord sur les temps de traitements liés à l'exécution du garbage collector notamment parce que ce dernier est invoqué moins fréquemment mais que ses temps de traitement sont plus longs.

Donner la même valeur au paramètre -Xms et -Xmx permet d'éviter à la JVM de devoir effectuer des calculs sur la taille des différentes régions mais cela empêche aussi la JVM de procéder à des ajustements si les valeurs fournies ne sont pas judicieuses.

Le ratio entre la Young Generation et la Tenured Generation dans le tas est un facteur important dans les performances du ramasse-miettes. Plus le ratio de la Young Generation est important, moins il y a de collections mineures qui sont effectuées. Par contre, cela implique une taille de la Tenured Generation plus importante et donc un accroissement du nombre potentiel de collections majeures. La valeur du ratio entre les deux générations dépend donc du nombre d'objets créés et de la durée de vie de ces objets dans l'application.

La taille de la Young Generation est déterminée par le paramètre NewRatio qui indique le ratio de la young et de la Tenured Generation dans le tas.

La taille de la Young Generation peut être précisée avec le paramètre -XX:NewRatio=n où n représente le ratio entre la Young Generation et la old generation.

Exemple : si n vaut 3, la Young Generation aura une taille de 1/4 de la taille totale du tas.

Le paramètre -XX:NewSize permet de préciser la taille initiale de la Young Generation.

Le paramètre -XX:MaxNewSize permet de préciser la taille maximale de la Young Generation.

La valeur par défaut de l'attribut NewRatio est dépendante de la plate-forme et du mode d'exécution de la VM Hotspot (client ou server).

La définition de la taille des générations du tas peut se faire de plusieurs façons :

- en utilisant l'attribut NewRatio : pour avoir une définition sous la forme de ratio
- en utilisant les attributs NewSize et MaxNewSize pour avec une définition précise

Remarque : l'attribut -Xmn est un raccourci pour NewSize

Plus la taille de la Young Generation est importante plus les chances qu'un objet meurt dans la Young Generation et ne soit donc pas promu dans la old generation est élevé. Cependant, il n'est pas recommandé d'avoir une taille très importante pour la Young Generation car cela fera exécuter le ramasse-miettes moins souvent mais son temps d'exécution sera plus long et comme ces traitements sont de type stop the world, les temps de pause de l'application seront plus longs. L'idéal est d'adapter la taille de la génération en fonction de l'application exécutée dans la JVM.

Le paramètre -XX:SurvivorRatio=n permet de préciser le ratio entre l'espace Eden et les deux espaces survivor dans la Young Generation.

Exemple : si n vaut 6 alors le ratio est 1:6. Dans ce cas, chaque espace survivor occupera 1/8 de la taille de la Young Generation (ce n'est pas 1/7 car il y a deux espaces de type survivor)

Si l'espace requit pour copier un objet dans l'espace survivor n'est pas assez grand, alors l'objet est promu directement dans la old generation.

A chaque collection mineure, le ramasse-miettes détermine le nombre de collections qu'un objet doit avoir subi avant d'être promu dans la old generation. Ce nombre est déterminé de façon à ce qu'à la fin de la collection l'espace survivor soit à moitié rempli

L'option -XX:+PrintTenuringDistribution permet de voir la répartition par âges des objets de la Young Generation. Ceci permet de voir la répartition de la durée des objets de la Young Generation.

Il faut tout d'abord décider de la quantité de mémoire qui sera affectée au tas. Ensuite, il est possible de mesurer les performances et d'ajuster la taille de la Young Generation en fonction du comportement de l'application.

Pour la plupart des applications, la Permanent Generation n'influe pas de façon importante sur les performances du ramasse-miettes. Pour des applications qui chargent et/ou génèrent beaucoup de classes, il faut augmenter la taille de cette génération pour éviter des manques de mémoire.

La taille du tas ne permet pas à elle seule de déterminer la quantité de mémoire utilisée par le processus système de la JVM puisque le tas ne représente qu'une partie de la mémoire de la JVM.

Il ne doit donc pas être étonnant que la quantité de mémoire affichée par le système (TaskManager sous Windows ou top sous Unix like par exemple) soit supérieure à la taille maximale du tas précisée avec l'option -Xmx.

65.4. Le monitoring de l'activité du ramasse-miettes

La suite de cette section sera développée dans une version future de ce document

65.5. Les différents types de référence

Java 1.2 propose plusieurs types de références qui contiennent une référence particulière sur un objet.

Ces références sont définies dans des classes du package java.lang.ref :

- Soft Reference
- Weak Reference
- Hard Reference
- Phantom Reference

Ces différentes références peuvent être utilisées par le ramasse-miettes pour récupérer de la mémoire au cas où celle-ci commence à manquer.

La suite de cette section sera développée dans une version future de ce document

65.6. L'obtention d'informations sur la mémoire de la JVM

La classe Runtime propose deux méthodes pour obtenir des informations basiques sur la mémoire occupée par le tas :

- totalMemory() : renvoie la quantité totale de mémoire du tas
- freeMemory() : renvoie la quantité de mémoire libre du tas

Depuis la version 5 de la JVM, il est aussi possible d'obtenir des informations sur la mémoire en utilisant les MBeans JMX exposés par la JVM.

65.7. Les fuites de mémoire (Memory leak)

La libération de la mémoire des objets inutilisés est implicite en Java grâce au ramasse-miettes alors qu'elle est explicite dans d'autres langages (en Pascal avec l'instruction dispose, en C avec l'instruction free, ...). Ceci facilite le travail du développeur puisqu'il n'a pas à libérer explicitement la mémoire des objets.

Il est facile de penser que la libération de mémoire étant assurée par le garbage collector de la JVM, le développeur n'a plus aucune responsabilité à ce sujet et que les fuites de mémoires sont impossibles. Ce raisonnement est le résultat de la méconnaissance du mode de fonctionnement du garbage collector.

Le ramasse-miettes doit s'assurer pour libérer la mémoire d'un objet que celui-ci n'est plus utilisé : pour le déterminer, il recherche s'il existe parmi les objets de la JVM, une référence vers l'objet. Même s'il n'est plus utilisé mais qu'il existe encore une référence sur l'objet, la mémoire de celui-ci n'est pas libérée. Ceci rend la tâche de détection d'une fuite de mémoire particulièrement difficile car il est facile de savoir si un objet est encore utile mais il est difficile de savoir s'il existe encore une référence vers l'objet.

Une mauvaise utilisation de l'API collection par exemple peut notamment favoriser les fuites de mémoires.

Une fuite de mémoire se traduit généralement par une augmentation de la taille du heap pouvant aller jusqu'à un arrêt de la JVM avec une exception de type OutOfMemoryError.

Le premier réflexe lorsqu'une exception de type OutOfMemoryError est levée concernant le tas est d'augmenter la taille de la mémoire de JVM. Cependant, si cette erreur est liée à une fuite de mémoire cela ne fait que reporter sa levée.

L'indicateur le plus visible lors d'une possible fuite de mémoire est la levée d'une exception de type OutOfMemory. La levée de cette exception n'implique pas obligatoirement une fuite mémoire mais peut simplement traduire un manque de mémoire pour permettre l'exécution de l'application.

Cependant, c'est l'issue fatale suite à une fuite de mémoire qui peut être plus ou moins longue. Ceci est particulièrement vrai pour des applications exécutées sur des serveurs car elles ne sont généralement pas redémarrées fréquemment.

Même si c'est un travail long et difficile, il faut toujours traiter une fuite de mémoire avec une grande attention. La solution n'est pas d'augmenter la taille du tas car cela ne fera que reporter l'échéance fatale. La solution n'est pas non plus de redémarrer périodiquement la JVM car généralement les applications concernées doivent avoir un taux de disponibilité élevé.

La suite de cette section sera développée dans une version future de ce document

65.7.1. Diagnostiquer une fuite de mémoire

Le diagnostique d'une fuite de mémoire dans une application Java est une tâche difficile et longue qui nécessite généralement des outils qui vont permettre de voir quels sont les objets stockés dans la mémoire de la JVM.

Le JDK fournit de plus en plus d'outils pour effectuer ces recherches et donner des indications sur l'origine du problème mais ils sont en ligne de commande et sont donc peu productifs notamment jmap, jhat et jconsole. Leur intérêt est cependant d'être fourni en standard. A partir de Java 6 update 7 l'outil graphique Visual VM propose de regrouper ces fonctionnalités de façon conviviale.

Le paramètre -XX:+HeapDumpOnOutOfMemoryError de la JVM HotSpot permet de demander la création d'un dump du tas au cas où l'exception de type OutOfMemoryError est levée. La commande jmap permet alors de fournir un histogramme en utilisant l'option -histo suivi du fichier contenant le dump.

Le paramètre -XX:+PrintClassHistogram de la JVM HotSpot permet de demander l'affichage dans la console de l'histogramme des classes du tas lorsque la combinaison Ctrl + Arret défil est utilisée.

Il existe aussi plusieurs outils de profiling open source (Eclipse TPTP, Eclipse Mat, Netbeans Profiler, ...) ou commerciaux.

Bien que leur utilisation facilite le travail, la détection d'une fuite de mémoire est souvent délicate car :

- le tas d'une JVM contient généralement de très nombreux objets : par exemple une petite application peut facilement avoir plusieurs milliers d'objets en mémoire

- l'analyse des objets contenus dans le tas nécessite une bonne connaissance de l'application
- généralement la fuite est légère
- l'analyse de la liste des objets et du nombre de leurs instances est le principal indicateur pour déterminer l'origine de la fuite

La détection d'une fuite de mémoire n'est souvent qu'une hypothèse en cas d'arrêt de la JVM par manque de mémoire. Dans ce cas, la suspicion de fuite de mémoire est conditionnée par l'importance de la fuite, la taille de la mémoire de la JVM et la durée de vie de la JVM.

Exemple : une petite fuite de mémoire dans une application de type client lourd fréquemment relancée ne sera peut être jamais détectée par contre une petite fuite de mémoire dans une application de type web dont la durée de vie est longue a des chances de provoquer tôt ou tard une erreur de type OutOfMemoryError.

Le grand avantage des fuites de mémoire en Java est qu'elles n'ont pas d'impact sur le système d'exploitation. La JVM et donc l'application s'arrête et la mémoire qui lui était allouée est restituée au système.

Le simple fait de regarder, avec les outils du système d'exploitation, la quantité de mémoire consommée par la JVM ne peut en aucun cas fournir d'indication sur une possible fuite de mémoire dans l'application.

La quantité de mémoire indiquée par le système ne contient qu'une partie relative au tas de la JVM. De plus la taille du tas peut varier entre le minimum et le maximum défini. Au démarrage de la VM, l'espace de mémoire du tas alloué correspond à la valeur minimale. Au fur et à mesure des besoins, la taille du tas peut grossir jusqu'à la valeur maximale et cela sans attendre la fin des traitements du ramasse-miettes. Dans ce cas, la quantité de mémoire indiquée par le système grossit mais n'implique pas obligatoirement une fuite de mémoire.

Un outil de type profiler est nécessaire pour permettre d'inspecter le contenu du tas et connaître le nombre d'objets, le nombre d'instances de chaque classe, ...

Même avec ce type d'outil, la recherche d'une fuite de mémoire est un processus généralement long et itératif.

Certaines entités sont propices à la génération de fuites de mémoire :

- utilisation de collections qui ont une durée de vie assez longue (par exemple les collections déclarées static).
- dans une application graphique, abonnement à un listener et oubli de se désabonner
- ...

Les conséquences d'une fuite de mémoire dans une application Java sont généralement moins dramatiques que dans des applications natives dans la mesure où en Java seule la JVM et donc l'application risque de s'arrêter. Une fuite de mémoire peut engendrer un arrêt de la JVM dans laquelle l'application s'exécute mais le système d'exploitation reste opérationnel. Dans une application native; une fuite de mémoire peut aller jusqu'à nécessiter le redémarrage du système d'exploitation si ce dernier n'a plus assez de mémoire.

Les fuites de mémoire dans une application peuvent avoir plusieurs origines dont les plus communes sont :

- la présence de références sur des objets non désirés ou inconnus
- l'oubli de la libération de certaines ressources externes
- bug dans une bibliothèque tierce

Le ramasse-miettes invoque la méthode finalize() si celle-ci est implémentée pour un objet avant que son espace mémoire ne soit récupéré. Il n'y a cependant aucune garantie sur la bonne exécution de cette méthode : il ne faut surtout pas s'en servir pour libérer des ressources sous prétexte qu'elle est invoquée automatiquement par le ramasse-miettes.

65.8. Les exceptions liées à un manque de mémoire

Deux exceptions différentes peuvent être levées par la JVM selon l'origine du manque de mémoire StackOverFlowError et OutOfMemoryError. Ces deux exceptions ne sont pas checkées mais provoquent un arrêt de la JVM si elles ne sont pas traitées dans un bloc catch durant la remontée de la pile d'appels du thread.

65.8.1. L'exception de type StackOverFlowError

Si la taille de la pile est trop petite alors une exception de type java.lang.StackOverFlowError est levée.

Il y a deux grandes origines au fait que la taille de la pile ne soit pas assez importante :

- généralement c'est un appel récursif à une méthode sans condition d'arrêt (une méthode qui s'appelle elle-même)
- la quantité de données à stocker dans la pile est supérieure à la taille de la pile : dans ce cas il faut agrandir la taille des piles en utilisant l'option -Xss. Attention cependant car cette option s'applique à toutes les piles (une par thread)

65.8.2. L'exception de type OutOfMemoryError

Une exception de type OutOfMemoryError est assez courante lors de l'exécution d'applications Java : elle indique qu'il n'y a pas l'espace disponible pour créer de nouveaux objets même après l'exécution du ramasse-miettes et qu'il n'y a pas la possibilité d'agrandir la taille du tas.

L'exception OutOfMemoryError peut concerner plusieurs parties de la mémoire de la JVM : cette partie est explicitement indiquée dans le message de l'exception :

- java.lang.OutOfMemoryError: Java heap space
- java.lang.OutOfMemoryError: PermGen space
- java.lang.OutOfMemoryError: Requested array size exceeds VM limit
- java.lang.OutOfMemoryError: Request <size> bytes for <reason>. Out of swap space?.
- java.lang.OutOfMemoryError: <reason> <stack trace> (Native method)

Il est donc important de bien prendre en compte le message de l'exception OutOfMemoryError pour pouvoir y apporter une solution.

Une exception de type OutOfMemoryError n'est pas obligatoirement un problème de fuite de mémoire mais simplement une mauvaise adaptation de la configuration de la JVM aux besoins de l'application.

S'il est nécessaire de réduire l'empreinte mémoire de l'application dans la JVM, il faut tenter de réduire le nombre d'objets ou de limiter la durée de vie de certains objets, par exemple :

- Essayer de réduire la durée de vie de certains objets, par exemple réduire le timeout des sessions http d'un conteneur web
- Si l'application utilise un cache alors il faut vérifier sa taille et la limiter. Il est aussi possible d'utiliser des objets ayant des soft références dans les caches ce qui permettra au ramasse-miettes de supprimer ces objets en cas de manque d'espace dans le tas
- S'assurer de la correcte libération de ressources externes
- ...

L'utilisation d'un outil de profiling peut être nécessaire voire obligatoire pour analyser le contenu de la mémoire de la JVM et déterminer l'origine de la consommation mémoire. L'utilisation de ce type d'outils induit forcément un overhead et réduit donc sensiblement les performances. Leur utilisation doit donc être limitée dans un environnement de production.

65.8.2.1. L'exception OutOfMemoryError : Java heap space

Une exception de type OutOfMemory est levée avec le message "Java heap space" lorsque l'espace mémoire libre du tas (heap) ne permet plus la création de nouveaux objets malgré l'exécution du ramasse-miettes.

Dans ce cas, l'exception peut avoir plusieurs origines :

- l'espace mémoire alloué au tas de la JVM est insuffisant pour créer les objets requis par l'application. C'est généralement le cas pour des applications gourmandes en ressources (grosses applications web ou graphiques, ...)

- une fuite de mémoire empêche le ramasse-miettes de libérer des objets qui sont pourtant inutilisés mais dont il existe encore des références. Ainsi ces objets ne sont jamais libérés et occupent de plus en plus d'espace dans le tas jusqu'à occuper tout l'espace disponible.
- de nombreux objets possèdent un finalizer. Lors de la prise en compte de ces objets par le ramasse-miettes, ils sont mis dans une file pour être ultérieurement traités par un thread dédié qui va exécuter le finalizer avant le libérer la mémoire
- ...

65.8.2.2. L'exception OutOfMemoryError : PermGen space

Une exception de type OutOfMemory est levée avec le message "PermGen space" lorsque l'espace mémoire alloué à la Permanent Generation n'est pas assez important pour contenir toutes les métadonnées utilisées par la JVM.

C'est généralement pour des applications côté serveur car elles s'exécutent dans un même conteneur et utilisent généralement de nombreuses classes différentes liées à l'utilisation de plusieurs frameworks.

La représentation interne des chaînes de caractères de type constante est aussi stockée dans un pool de la Permanent Generation. Lorsque la méthode intern() de la classe String est invoquée, elle recherche dans le pool si la chaîne existe déjà. Si c'est le cas, elle renvoie celle du pool sinon elle l'ajoute. L'espace mémoire requis pour le Permanent Generation est donc plus imporant lorsqu'une application utilise beaucoup de chaînes de caractères sous la forme de constantes.

La seule solution est alors d'agrandir l'espace mémoire alloué à la Permanent Generation, par exemple en utilisant l'option -XX:MaxPermSize pour une JVM Hotspot.

65.8.2.3. L'exception OutOfMemoryError : Requested array size exceeds VM limit

Une exception de type OutOfMemory est levée avec le message "Requested array size exceeds VM limit" lorsqu'une tentative de création d'un tableau requiert plus de mémoire que l'espace libre du tas.

Si la taille du tableau à créer est normale alors la seule solution est d'augmenter la taille du tas de la JVM.

66. La JVM HotSpot dans un conteneur Docker

Chapitre 66

Niveau : 🟩🟩🟨🟨🟥⬜ Confirmé

L'utilisation des conteneurs s'est généralisée notamment avec l'accroissement de l'utilisation du cloud pour le déploiement et l'exécution d'applications.

Il est donc courant de devoir conteneuriser une application Java exécutée dans une JVM.

Sous Linux, plusieurs gestionnaires de conteneurs sont disponibles :

- Docker,
- rkt,
- runC,
- Podman,
- etc

Docker est historiquement le plus utilisé.

Généralement, surtout lors de l'utilisation d'orchestrateur comme Kubernetes, il est courant de limiter les ressources du conteneur en termes de CPU et de mémoire. Ceci facilite les capacités d'orchestration automatisé mais aussi limite la possibilité d'un processus exécuté dans un conteneur à consommer toutes les ressources sur le noeud dans lequel il s'exécute.

La limitation des ressources allouées à l'environnement d'exécution d'une JVM peut avoir des conséquences notamment sur ses performances car elle est historiquement créée et utilisée sur des serveurs avec des ressources plus ou moins importantes.

La machine virtuelle Java, jusqu'à la version du JDK 10, n'est pas pleinement consciente des mécanismes d'isolation utilisés par les conteneurs, ce qui peut conduire à un comportement inattendu entre différents environnements.

Ce chapitre contient plusieurs sections :

- ♦ La limitation des ressources d'un conteneur Docker
- ♦ L'utilisation et la configuration des ressources utilisables par une JVM
- ♦ Le support de Docker par la JVM
- ♦ Le support de cgroups v2

66.1. La limitation des ressources d'un conteneur Docker

La popularité des conteneurs est en partie due aux facilités de mise en ouvre des fonctionnalités sous-jacentes. Ces fonctionnalités sont proposées par le noyau Linux : ce sont notamment cgroups et namespaces.

Les conteneurs mettent en ouvre des mécanismes d'isolation où les ressources (CPU, mémoire, système de fichiers, réseau, etc.) d'un conteneur sont isolées les unes des autres. Cette isolation est possible grâce à une fonctionnalité du noyau Linux nommée namespaces.

Les namespaces sont essentiellement utilisés pour offrir une des vues isolées du système. Ainsi, chaque conteneur peut voir sa propre vue sur différents namespaces :

- pid (processus)
- net (interfaces réseaux, routing.)
- ipc (System V IPC)
- mnt (points de montage, filesystems)
- uts (hostname)
- user (UIDs)

Ainsi, chaque conteneur possède un processus avec l'ID 1.

Les control groups (cgroups en abrégé) sont utilisés pour mesurer et limiter l'accès aux ressources.

Il existe deux versions majeures de cgroups :

- v1 : version initiale
- v2 : nouvelle version de l'API. Requiert un noyau Linux 5.8 minimum

Docker propose ainsi de limiter les ressources utilisables par le conteneur : il est possible de limiter les ressources mémoire et CPU utilisables par un conteneur Docker. Ces fonctionnalités sont intéressantes lors de l'exécution de nombreux conteneurs dans un ensemble de serveurs orchestré par Kubernetes. Kubernetes tente de rationnaliser le déploiement des conteneurs sur les serveurs notamment en tenant compte des ressources utilisables par les conteneurs.

66.1.1. La limitation en mémoire

Certaines commandes comme top ou free ainsi que la JVM pour les versions antérieures à Java 8u131 et Java 10 ne prennent pas en compte les restrictions d'utilisation de ressources pouvant être définies avec cgroups.

Exemple :

```
$ docker run -it ubuntu free -h
              total        used        free      shared  buff/cache   available
Mem:           2.0G         95M        1.5G        229M        345M        1.6G
Swap:          1.4G          0B        1.4G

$ docker run -it -m128M --memory-swap=128M ubuntu free -h
              total        used        free      shared  buff/cache   available
Mem:           2.0G         96M        1.5G        229M        346M        1.6G
Swap:          1.4G          0B        1.4G
```

Les options --memory et --memory-swap de Docker permettent de définir les limitations de ressources en mémoire utilisables par les processus exécutés dans le conteneur.

La documentation de Docker précise que ces deux options peuvent être utilisées pour limiter la quantité de mémoire utilisable par un conteneur :

- --memory ou -m : taille maximale de la mémoire sous la forme : <valeur>[<unité>]. La valeur est un entier positif. L'unité peut être b, k, m, ou g. La valeur minimum est 4M
- --memory-swap : taille maximale de la mémoire totale (mémoire + swap) sous la forme : <valeur>[<unité>]. La valeur est un entier positif. L'unité peut être b, k, m, ou g

La mémoire swap est utilisable si celle-ci est supporté par le système hôte.

Il y a plusieurs possibilités concernant la quantité de mémoire utilisable par un conteneur :

- aucune limite, ce qui est le comportement par défaut lorsqu'aucune option de limitation n'est utilisée : le conteneur peut utiliser la RAM disponible sur l'hôte et le swap
- utilisation uniquement de l'option --memory : dans ce cas, le conteneur dispose de la mémoire précisée mais aussi d'une quantité équivalente en swap

- utilisation de l'option --memory et de l'option --memory-swap avec la valeur -1 : dans ce cas, le conteneur dispose de la mémoire précisée et d'une zone de swap dont la limite n'est pas précisée
- utilisation des options -memory avec la valeur M et --memory-swap avec la valeur S : dans ce cas, le conteneur dispose de la mémoire M et d'une zone de swap dont la taille est S-M

Si --memory-swap est précisée alors sa valeur doit toujours être supérieure à celle de l'option --memory car malgré son nom elle précise la quantité totale de mémoire incluant le swap.

Exemple :

```
$ docker run --rm --memory=256m --memory-swap=128m -it ubuntu /bin/bash
C:\Program Files\Docker Toolbox\docker.exe: Error response from daemon: Minimum memoryswap limit
should be larger than memory limit, see usage.
See 'C:\Program Files\Docker Toolbox\docker.exe run --help'.
```

Si la limite de consommation mémoire est atteinte, alors le noyau Linux tue le processus.

Par exemple, en lançant un conteneur sur l'image jboss/wildfly en limitant la mémoire du conteneur à 64Mo. Le serveur Wildfly démarre mais au bon d'un certain temps, le conteneur est arrêté avec le message « *** JBossAS process (80) received KILL signal *** » dans la sortie standard.

Exemple :

```
$ docker run -it --name wildfly -m=64M jboss/wildfly
=========================================================================

  JBoss Bootstrap Environment

  JBOSS_HOME: /opt/jboss/wildfly

  JAVA: /usr/lib/jvm/java/bin/java

  JAVA_OPTS:  -server -Xms64m -Xmx512m -XX:MetaspaceSize=96M -XX:MaxMetaspaceSize=256m
  -Djava.net.preferIPv4Stack=true -Djboss.modules.system.pkgs=org.jboss.byteman
  -Djava.awt.headless=true --add-exports=java.base/sun.nio.ch=ALL-UNNAMED
  --add-exports=jdk.unsupported/sun.misc=ALL-UNNAMED
  --add-exports=jdk.unsupported/sun.reflect=ALL-UNNAMED

=========================================================================

13:28:54,179 INFO  [org.jboss.modules] (main) JBoss Modules version 1.9.1.Final
13:29:08,588 INFO  [org.jboss.msc] (main) JBoss MSC version 1.4.8.Final
13:29:08,887 INFO  [org.jboss.threads] (main) JBoss Threads version 2.3.3.Final
13:29:14,499 INFO  [org.jboss.as] (MSC service thread 1-2) WFLYSRV0049: WildFly
 Full 17.0.1.Final (WildFly Core 9.0.2.Final) starting
*** JBossAS process (80) received KILL signal ***
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
              PORTS               NAMES

$
```

Il est possible de regarder les informations relatives au conteneur et notamment l'objet Json State :

Exemple :

```
$ docker inspect wildfly
[
    {
        "Id": "179552e4dc026789f6546fc5b04b5375808841ce8990f1a9cd031fbbdad10f44",
        "Created": "2019-08-14T13:28:35.900271741Z",
        "Path": "/opt/jboss/wildfly/bin/standalone.sh",
        "Args": [
            "-b",
            "0.0.0.0"
        ],
        "State": {
            "Status": "exited",
            "Running": false,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": true,
            "Dead": false,
            "Pid": 0,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2019-08-27T09:15:14.831865506Z",
            "FinishedAt": "2019-08-27T09:47:02.238997935Z"
        },
```

La propriété OOMKilled est à true, ce qui signifie que le démon a tué le conteneur par manque de ressource mémoire.

66.1.2. La limitation en CPU

Par défaut, un processus exécuté dans un conteneurisé peut utiliser toute la CPU disponible sur le système l'hôte. Cependant, il est possible de limiter la consommation de ressource CPU par un conteneur Docker de plusieurs manières.

Cela peut être utile notamment :

- si un grand nombre de conteurs requérant beaucoup de threads est exécuté sur un même hôte : cela peut saturer la CPU et induire beaucoup de contention.
- pour éviter qu'un processus, pour une raison non souhaitée comme une boucle infinie, consomme toute la CPU

L'utilisation de ces fonctionnalités pour limiter la consommation CPU a une incidence sur la valeur retournée par la méthode availableProcessors() de la classe Runtime selon la version de Java utilisée.

Dans un conteneur Linux, il est possible de limiter les ressources CPU de plusieurs manières :

- cpu_shares : une valeur relative qui permet de donner au conteneur une part de la CPU. La valeur par défaut est 1024. Par exemple, si un conteneur a 512 et un autre a 1024, le second conteneur aura le double du temps CPU par rapport au premier. S'il n'y a pas de concurrence d'utilisation, par exemple si le premier conteneur n'utilise pas de temps CPU, alors le deuxième conteneur peut utiliser tous les cycles CPU disponibles.
  cpu_shares permet de rester équitable lorsque le nombre de conteneurs augmente et d'utiliser toute la CPU même si certains conteneurs n'en nécessitent pas.
  L'utilisation de cpu_share avec une JVM doit être évalué en fonction de l'utilisation des threads de l'application.
- cpu_quota : un temps CPU qui peut être utilisé sur une période (cpu_period). La période par défaut est de 100 microsecondes, donc régler cette valeur à 50 microsecondes, c'est un peu comme donner au conteneur un demi-coeur et la régler à 200 microsecondes, c'est un peu comme donner 2 coeurs au conteneur.
  Cependant, si l'hôte a plus de coeurs que cela et que votre application est multithread, les threads peuvent fonctionner sur plusieurs cours et utiliser le quota très rapidement. Les threads risquent de se retrouver fréquemment en attente de CPU.
  La JVM étant multi-threads et le nombre de threads étant généralement supérieur au nombre de coeurs, l'utilisation de cpu_quota n'est pas toujours une bonne solution.
- cpu_sets : cette fonctionnalité permet de demander l'exécution du conteneur sur le ou les cours précisés.

La prise en charge de ces fonctionnalités dépend de la version de la JVM, notamment depuis l'ajout de l'option ContainerSupport qui améliore le support de la prise en compte des restrictions de ressources CPU définies sur un conteneur.

La JVM de Java 10 divise les cpu_shares par 1024 pour déterminer le nombre de coeurs utilisables.

La JVM de Java 11 regarde d'abord cpu_quota, si celui-ci est défini, il est divisé par cpu_period pour obtenir un nombre effectif de coeurs. Si ce n'est pas le cas, les cpu_shares sont utilisés comme le fait le JDK 10.

La limitation de la CPU pour un conteneur qui exécute une JVM doit être soigneusement évalué selon l'application.

La prise en compte de ses restrictions par la JVM est importante pour éviter qu'une JVM n'utilise trop de threads pour son ramasse-miette ou des pools de threads notamment celui du framefork Fork/Join. C'est d'autant plus important pour ne pas saturer la CPU d'un hôte sur lequel s'exécute de nombreux conteneurs.

66.1.2.1. CPU Shares

CPU shares définit une pondération de priorité sur tous les cycles du CPU à travers tous les cours. Le nombre de cycles dépend de l'ensemble des processus qui s'exécutent sur le noeud.

Cela rationne l'utilisation de la CPU par le processus du conteneur selon la proportion précisée par rapport aux autres conteneurs mais uniquement lorsque le système est sous forte charge. Si la CPU est libre alors le processus peut dépasser la limite fixée.

Par défaut, tous les conteneurs ont la même proportion de cycles CPU. Cette proportion peut être modifiée en changeant la pondération de la part de la CPU du conteneur par rapport à celle de tous les autres conteneurs en cours d'exécution sur le système hôte.

La contrainte de type CPU shares est précisée en utilisation l'option --cpu-shares ou -c.

La valeur par défaut est 1024. L'argument précisé est interprété par rapport à tous les autres conteneurs du système, ce qui rend difficile l'interprétation concrète de la valeur.

Par exemple, avec trois conteneurs, l'un a cpu-shares à 1024 et les deux autres ont un cpu-shares à 512. Lorsque les processus des trois conteneurs tentent d'utiliser 100% de la CPU, le premier conteneur reçoit 50% du temps CPU total et les deux autres 25%. Si on ajoute un quatrième conteneur avec un cpu-shares à 1024, le premier et le quatrième conteneur obtiennent 33% du temps CPU. Les deux autres conteneurs restants reçoivent 16,5% de la CPU.

La proportion ne s'applique que lorsque des processus gourmands en ressources CPU sont en cours d'exécution. Lorsque les tâches d'un conteneur sont inactives, les autres conteneurs peuvent utiliser le temps CPU restant. La quantité réelle de temps processeur varie donc en fonction des autres conteneurs en cours d'exécution sur le système.

Sur un système multicoeurs, les parts de temps CPU sont réparties sur tous les cours de la CPU. Même si un conteneur est limité à moins de 100% du temps processeur, il peut utiliser 100% de chaque cour de processeur.

Exemple :

```
$ docker run -it --rm --cpu-shares=1024 openjdk:10.0.2-jdk
Aug 25, 2019 7:54:09 AM java.util.prefs.FileSystemPreferences$1 run
INFO: Created user preferences directory.
|  Welcome to JShell -- Version 10.0.2
|  For an introduction type: /help intro

jshell> System.out.println(Runtime.getRuntime().availableProcessors())
2

jshell> /ex
|  Goodbye

$ docker run -it --rm --cpu-shares=512 openjdk:10.0.2-jdk
Aug 25, 2019 7:54:57 AM java.util.prefs.FileSystemPreferences$1 run
INFO: Created user preferences directory.
|  Welcome to JShell -- Version 10.0.2
|  For an introduction type: /help intro

jshell> System.out.println(Runtime.getRuntime().availableProcessors())
1

$ docker run -it --rm --cpu-shares=4096 openjdk:10.0.2-jdk
Aug 25, 2019 7:55:41 AM java.util.prefs.FileSystemPreferences$1 run
INFO: Created user preferences directory.
|  Welcome to JShell -- Version 10.0.2
|  For an introduction type: /help intro

jshell> System.out.println(Runtime.getRuntime().availableProcessors())
2
```

66.1.2.2. CPU Period/Quota

Cette contrainte utilise le Linux Completely Fair Scheduler pour restreindre l'utilisation de la CPU par le conteneur. Le conteneur aura un temps CPU limité même lorsque la machine est peu chargée et que la charge de travail peut être répartie sur tous les processeurs de l'hôte.

La contrainte de type CPU period/quota est précisée en utilisation l'option --cpus ou les options --cpu-period et --cpu-quota (cpus = cpu-quota / cpu-period).

La période par défaut de CPU CFS est 100ms. Cette valeur peut être modifiée grâce à l'option --cpu-period. Cette option s'utilise en conjonction de l'option --cpu-quota

De manière plus simple, il est possible d'utiliser l'option --cpus avec une valeur flottante. Par exemple, la valeur 0.5 correspond à 50% de CPU.

La valeur par défaut est 0 qui implique aucune restriction.

Exemple :

```
$ docker run -it --rm --cpus=1 openjdk:10.0.2-jdk
Aug 25, 2019 8:54:02 AM java.util.prefs.FileSystemPreferences$1 run
INFO: Created user preferences directory.
|  Welcome to JShell -- Version 10.0.2
|  For an introduction type: /help intro

jshell> System.out.println(Runtime.getRuntime().availableProcessors())
1

jshell> /ex
|  Goodbye

$ docker run -it --rm --cpus=0.5 openjdk:10.0.2-jdk
Aug 25, 2019 8:54:36 AM java.util.prefs.FileSystemPreferences$1 run
INFO: Created user preferences directory.
|  Welcome to JShell -- Version 10.0.2
|  For an introduction type: /help intro

jshell> System.out.println(Runtime.getRuntime().availableProcessors())
1

jshell> /ex
|  Goodbye

$ docker run -it --rm --cpus=2 openjdk:10.0.2-jdk
Aug 25, 2019 8:55:07 AM java.util.prefs.FileSystemPreferences$1 run
INFO: Created user preferences directory.
|  Welcome to JShell -- Version 10.0.2
|  For an introduction type: /help intro

jshell> System.out.println(Runtime.getRuntime().availableProcessors())
2

jshell> /ex
|  Goodbye
```

66.1.2.3. CPU Sets

La contrainte CPU set permet de limiter l'exécution des processus du conteneur uniquement sur les cours précisés. Cela peut être intéressant pour éviter que les processus ne changent de CPU ou profiter sur des systèmes NUMA où les CPU ont un accès rapide à différentes régions de la mémoire.

Contrairement aux deux contraintes précédentes, le processus exécuté dans le conteneur est rattaché à des cours spécifiques dédiés. Il est possible que le processus doive partager ces cours, mais ne pourra pas utiliser d'autres cours.

La contrainte de type CPU sets est précisée en utilisation l'option --cpuset-cpus.

Si un seul CPU est utilisé, le premier CPU dans l'exemple ci-dessous, alors le nombre de coeurs détecté par la JVM est bien 1 seul.

Exemple :

```
$ docker run -it --rm --cpuset-cpus="0" openjdk:10.0.2-jdk
Aug 29, 2019 8:53:50 PM java.util.prefs.FileSystemPreferences$1 run
INFO: Created user preferences directory.
|  Welcome to JShell -- Version 10.0.2
|  For an introduction type: /help intro

jshell> System.out.println(Runtime.getRuntime().availableProcessors())
1

jshell> /ex
|  Goodbye
```

Si plusieurs CPU sont précisés, les deux premiers CPU dans l'exemple ci-dessous, alors le nombre de coeurs détecté par la JVM est bien 2.

Exemple :

```
$ docker run -it --rm --cpuset-cpus="0-1" openjdk:10.0.2-jdk
Aug 29, 2019 8:56:55 PM java.util.prefs.FileSystemPreferences$1 run
INFO: Created user preferences directory.
|  Welcome to JShell -- Version 10.0.2
|  For an introduction type: /help intro

jshell> System.out.println(Runtime.getRuntime().availableProcessors())
2

jshell> /ex
|  Goodbye
```

Il est aussi possible de préciser chaque CPU séparé par une virgule.

Exemple :

```
$ docker run -it --rm --cpuset-cpus="0,2,4" openjdk:10.0.2-jdk
```

66.1.2.4. Le choix du type de limitation

Sur une machine avec un nombre suffisant de coeurs, il peut être intéressant d'utiliser le CPU set cela devrait réduire les changements de contexte et de profiter mieux des caches CPU.

Il peut être intéressant d'utiliser CPU period/quota pour des applications qui requièrent de nombreux threads qui sont souvent en attente.

L'utilisation de CPU shares peut être intéressant si les applications peuvent partager des cycles de processeurs disponibles.

Dans tous les cas, pour valider son choix, il faut profiler et mesurer.

66.2. L'utilisation et la configuration des ressources utilisables par une JVM

Une JVM propose de très nombreuses options pour sa configuration. Pour permettre une utilisation simple, la JVM propose un mécanisme appelé ergonomics qui définit certaines valeurs par défaut en tenant compte de l'environnement d'exécution.

Ces valeurs par défaut peuvent être remplacées par des valeurs fournies explicitement.

66.2.1. Les ergonomics de la JVM

Lors de l'exécution d'une application sans configuration particulière de la JVM, cette dernière va déterminer des valeurs de paramètres par défaut. Certaines valeurs sont en dur et d'autres sont déterminées à partir d'informations extraites de l'environnement d'exécution. Ce mécanisme de la JVM est désigné par le terme ergonomics.

Les ergonomics sont des règles utilisées par la JVM pour définir certaines valeurs de configurations par défaut. Ils utilisent le nombre de CPU et la quantité de RAM pour définir certaines valeurs par défaut de plusieurs propriétés de la JVM si elles ne sont pas explicitement définies notamment :

- La taille max de heap
- Le nombre de threads à utiliser par le ramasse-miettes (ConcGCThreads, ParallelGCThreads, .)
- Le nombre de threads à utiliser par le compilateur JIT
- Le nombre de threads à utiliser par le pool Fork/Join

Les valeurs définies via les ergonomics de la JVM dépendent de la version et du fournisseur. Pour éviter des surprises, il est préférable de définir explicitement ces valeurs.

Pour demander à la JVM d'afficher les valeurs de toutes ses propriétés, il faut utiliser l'option -XX:+PrintFlagsFinal

Exemple :

```
java -XX:+PrintFlagsFinal -version | grep -Ei "gcthreads|maxheap|maxram"
     uint ConcGCThreads                = 1           {product} {ergonomic}
    uintx MaxHeapFreeRatio             = 70          {manageable} {default}
   size_t MaxHeapSize                  = 2122317824  {product} {ergonomic}
 uint64_t MaxRAM                      = 137438953472 {pd product} {default}
    uintx MaxRAMFraction              = 4            {product} {default}
     uint ParallelGCThreads           = 4            {product} {default}
     bool UseDynamicNumberOfGCThreads = false        {product} {default}
java version "9.0.1"
Java(TM) SE Runtime Environment (build 9.0.1+11)
Java HotSpot(TM) 64-Bit Server VM (build 9.0.1+11, mixed mode)
```

Attention : les valeurs peuvent et parfois changent selon la version, le fournisseur et le système d'exploitation utilisé.

66.2.2. L'utilisation du nombre de CPU par la JVM

Si les options -XX:ParallelGCThreads (nombre de threads pour les ramasse-miettes parallèles) ou -XX:CICompilerCount (nombre de threads pour le compilateur JIT) ne sont pas précisées, la JVM récupère le nombre de CPU du système hôte pour déterminer leur valeur.

Selon certaines versions de Java, la JVM va déterminer le nombre de CPU éventuellement en tenant compte des restrictions définies sur le conteneur si elles sont utilisées :

- soit avec l'utilisation de l'option expérimentale -XX:+UseCGroupMemoryLimitForHeap
- soit avec l'option -XX:+UseContainerSupport activée par défaut sur Linux

A partir de Java 11, si des limitations sur CPU sont appliquées sur le conteneur, la formule utilisée par la JVM pour déterminer le nombre de coeur utilisable par la JVM dans le conteneur est : min(cpuset-cpus, cpu-shares/1024, cpus) arrondi au nombre entier supérieur.

En se basant sur le nombre de coeurs disponible, la JVM détermine certaines valeurs par défaut si celles-ci ne sont pas explicitement définies, notamment :

- le nombre de theads utilisé par le ramasse-miettes
- le nombre de threads utilisé par le JIT
- la taille du pool de threads par défaut du framework Fork/join

De nombreux outils ou frameworks utilisent aussi la méthode availableProcessors() de la classe Runtime pour déterminer la taille de pool de threads (Netty, ElasticSearch, ...).

66.2.3. La gestion de la mémoire

L'empreinte mémoire d'une JVM est composée de différents éléments notamment :

- le heap, la zone de mémoire la plus connue dont la taille peut être précisée avec les options -Xmx
- la permgen / le metaspace
- la pile des threads
- la mémoire native
- le cache du code natif compilé par le JIT

Il est nécessaire de tenir compte de tout cela lors de la limitation de la mémoire d'un conteneur exécutant une application Java et ne pas uniquement prendre en compte du heap.

La quantité maximale de heap utilisable par JVM est soit définie explicitement avec l'option -Xmx soit déterminé par un mécanisme de la JVM appelé ergonomics.

Sans configuration explicite, les ergonomics d'une JVM définissent entre-autre la taille max du heap en fonction de la RAM totale de la machine sur laquelle elle s'exécute en appliquant la formule :

MaxHeapSize = TailleRAM / MaxRAMFraction

Avec par défaut, TailleRAM est la taille de la RAM de la machine hôte et MaxRAMFraction est par défaut à 4.

Exemple : sur une machine avec 8Go de RAM, la taille max du heap est fixée à 2Go soit ¼ de la taille de la RAM.

Exemple :

```
java -XX:+PrintFlagsFinal -version | grep -Ei "maxheapsize|maxram"
   size_t MaxHeapSize            = 2122317824       {product} {ergonomic}
 uint64_t MaxRAM                 = 137438953472     {pd product} {default}
    uintx MaxRAMFraction         = 4                {product} {default}
java version "9.0.1"
Java(TM) SE Runtime Environment (build 9.0.1+11)
Java HotSpot(TM) 64-Bit Server VM (build 9.0.1+11, mixed mode)
```

L'option la plus couramment utilisée pour gérer la mémoire de la JVM est de définir la taille maximale du heap en utilisant l'option -Xmx.

Exemple :

```
java -XX:+PrintFlagsFinal -Xmx1g -version | grep -Ei "maxheapsize|maxram"
   size_t MaxHeapSize          = 1073741824     {product} {command line}
 uint64_t MaxRAM               = 137438953472   {pd product} {default}
    uintx MaxRAMFraction       = 4              {product} {default}
java version "9.0.1"
Java(TM) SE Runtime Environment (build 9.0.1+11)
Java HotSpot(TM) 64-Bit Server VM (build 9.0.1+11, mixed mode)
```

Une autre manière de contrôler la taille maximale du heap est de définir la valeur de la propriété MaxRAM.

Exemple :

```
java -XX:+PrintFlagsFinal -XX:MaxRAM=1g -version | grep -Ei "maxheapsize|maxram"
   size_t MaxHeapSize          = 268435456      {product} {ergonomic}
 uint64_t MaxRAM               = 1073741824     {pd product} {command line}
    uintx MaxRAMFraction       = 4              {product} {default}
java version "9.0.1"
Java(TM) SE Runtime Environment (build 9.0.1+11)
```

```
Java HotSpot(TM) 64-Bit Server VM (build 9.0.1+11, mixed mode)
```

Dans ce cas, les ergonomics définissent la taille maximale du heap.

Lorsque la machine hôte possède au moins deux processeurs 64bits et 2Go de Ram, alors la JVM s'exécute en mode server. Dans le mode server, les ergonomics définissent la taille maximale du tas à un quart de la mémoire RAM totale de la machine. La valeur est cependant limitée à 32 go (la taille maximale adressable avec les compressed oops). Pour préciser une valeur supérieure, il faut utiliser l'option -Xmx.

Le heap est quasiment toujours la plus grande partie de mémoire utilisée par une JVM. Mais il est aussi important de prendre en compte que la mémoire utilisée par une JVM n'est pas que le heap.

La JVM a aussi besoin pour son fonctionnement interne d'une zone de mémoire : Permgen avant Java 8, remplacée par le MetaSpace à partir de Java 8. Elle contient notamment les classes chargées.

Il faut aussi tenir compte des piles de threads. Chaque thread possède une pile dont la taille par défaut est fixée par le paramètre -Xss. La valeur par défaut du paramètre -Xss dépend du système hôte et de la JVM utilisée. Une pile de thread contient les variables locales et les valeurs intermédiaires entre différents appels de méthodes. Même si une application ne créé pas elle-même de threads, la JVM en créé un certain nombre : un pour exécuter la méthode main(), éventuellement plusieurs pour le ramasse-miettes et le compilateur JIT, ...

Le code natif issu de la compilation du bytecode par le JIT est stocké dans une zone dédiée nommé Code Cache dont la taille est configurable en utilisant les options -XX:InitialCodeCacheSize et -XX:ReservedCodeCacheSize.

Une application peut allouer de la mémoire native en dehors du heap en utilisant les ByteBuffers de l'API NIO ou par malloc en utilisant JNI. La JVM peut aussi allouer de la mémoire native en plus du heap.

Il est possible d'activer le traçage de l'utilisation de mémoire native dans la JVM grâce à NMT (Native Memory Tracking) en utilisant l'option -XX:NativeMemoryTracking. Elle peut prendre trois valeurs :

- off : désactivé, valeur par défaut
- summary : permet d'obtenir un résumé de l'occupation des différentes zones utilisées
- detail : permet d'obtenir le détail

Exemple :

```
$ java -XX:NativeMemoryTracking=summary -Xms256m -Xmx256m -jar monapp.jar
```

Une fois activée, il est possible d'utiliser la commande jcmd avec en paramètres le pid de la JVM concernée et la valeur VM.native_memory pour obtenir des informations instantanées de la part de NMT.

Exemple :

```
$ jcmd 6741 VM.native_memory
```

Il est aussi possible d'obtenir une évolution entre deux instants dans le temps en utilisant les options baseline puis summary.diff

Exemple :

```
$ jcmd 6741 VM.native_memory baseline
...
$ jcmd 6741 VM.native_memory summary.diff
```

Il est possible de limiter la quantité totale de mémoire utilisable par la JVM (pour toutes les zones de mémoire dont elle a besoin) en utilisant l'option -XX:MaxRAM.

66.3. Le support de Docker par la JVM

Avant Java 10, exécuter des applications Java dans des conteneurs Linux était un peu délicat et nécessitait une configuration particulière pour éviter des surprises. Ces problèmes n'affectent pas seulement les versions de Java antérieures à 10, mais aussi certains outils qui collectent des informations du système comme top ou free. C'est parce que ces outils et la JVM ont été implémentés avant l'utilisation de cgroups et qu'ils ne sont donc pas optimisés pour une exécution dans un conteneur.

Avant Java 10, la JVM ne proposait pas de support pour détecter qu'elle fonctionnait à l'intérieur d'un conteneur et donc que certaines ressources qui lui sont allouées sont limitées en mémoire et/ou CPU. C'est la raison pour laquelle, il ne faut pas laisser les ergonomics de la JVM déterminer certaines valeurs par défaut notamment celles qui utilisent la quantité de mémoire et le nombre de CPU.

Historiquement, pour pallier une partie de ces problèmes, il était possible d'utiliser l'image de Fabric8.

A partir de Java 9 et Java 8u131, il est possible d'utiliser l'option expérimentale -XX:+UseCGroupMemoryLimitForHeap pour demander à la JVM de tenir compte des limites de mémoire définies par cgroups.

Java 10 a apporté plusieurs améliorations dans le support de l'exécution d'une JVM dans un conteneur.

Dans tous les cas, il est nécessaire de faire des tests.

66.3.1. Java SE 8 < u131

Lors de l'utilisation d'une version antérieure de Java à 8u131 dans un conteneur, laisser la JVM déterminer ces valeurs par défaut via ses ergonomics peut entraîner un comportement différent de celui lors de son exécution sur le système hôte. Spécifier la quantité de mémoire disponible pour un conteneur n'affecte pas ce que la JVM croit être disponible.

A moins de préciser explicitement la taille maximale du heap, la JVM détermine celle-ci en fonction de la RAM de l'hôte sur lequel elle s'exécute. Dans une JVM en mode server, la taille par défaut est un quart de la taille totale de la RAM de l'hôte.

Par défaut, si elle n'est pas précisée avec l'option -Xmx, la JVM de Java 8 utilise un quart de la mémoire totale du système hôte comme taille maximale du heap. La valeur de la taille mémoire est extraite du système hôte même si la JVM est exécutée dans un conteneur.

Exemple sur une machine hôte avec 8Go de Ram

Exemple :

```
$ java -XX:+PrintFlagsFinal -version | grep MaxHeapSize
    uintx MaxHeapSize                              := 2122317824                          {product}
```

La taille max du heap déterminée par la JVM est ¼ de 8go soit 2Go.

Exemple dans une VM avec 1Go de Ram

Exemple :

```
$ docker container run -it openjdk:8u121-jre-alpine java -XX:+PrintFlagsFinal -version |
 grep MaxHeapSize
    uintx MaxHeapSize                              := 260046848                           {product}
```

La taille max du heap déterminée par la JVM est ¼ de 1Go soit 256Mo.

Si la mémoire utilisable par le conteneur est limitée par configuration, alors la valeur extraite du système ne correspond pas à celle allouée au conteneur.

Exemple :

```
$ docker container run -it -m=128M openjdk:8u121-jre-alpine java -XX:+PrintFlagsFinal -version
| grep MaxHeapS ize
    uintx MaxHeapSize         := 524288000         {product}

$ docker container run -it -m=256M openjdk:8u121-jre-alpine java -XX:+PrintFlagsFinal -version
| grep MaxHeapSize
    uintx MaxHeapSize         := 524288000         {product}
```

La valeur fournie à l'option -m de Docker est utilisée comme valeur maximale de la mémoire utilisable ainsi que la valeur de l'espace de swap. Ainsi dans le conteneur, seul le double de la valeur précisée pourra être utilisée par la JVM. Au-delà de cette valeur, la JVM tente d'agrandir le heap jusqu'à dépasser la limite indiquée au conteneur. Le démon Docker va alors tuer le conteneur.

Par exemple, si le conteneur est exécuté sur une machine hôte possédant 32 Go de mémoire. La taille de la mémoire utilisable par le conteneur est limitée à 1Go. Si l'option -Xmx n'est pas utilisée pour la JVM, celle-ci va utiliser les valeurs par défaut déterminées par les ergonomics.

La JVM récupère la mémoire totale de la machine hôte. La JVM ne sachant pas détecter qu'elle s'exécute dans un conteneur, elle pense qu'elle s'exécute sur la machine hôte et donc que la mémoire disponible est de 32 Go. Par défaut, la JVM utilisera, comme taille maximale de son heap, le quart de 32Go soit 8Go comme valeur du paramètre -Xmx.

Au fur et à mesure de son utilisation, la taille du heap grossit jusqu'à dépasser 1Go. Une fois la taille maximale dépassée, le démon Docker tue le conteneur

Lorsque l'exécution d'un conteneur est interrompue de manière inopportune, il est possible d'utiliser la commande inspect de Docker sur le conteur pour obtenir les raisons de cet arrêt.

Une solution possible est de fixer la taille maximale du heap de la JVM en utilisant le paramètre -Xmx, mais cela implique qu'il faut configurer la mémoire deux fois, une fois pour le conteneur et une fois pour la JVM. De plus en cas de changement, il faut synchroniser l'autre valeur en correspondance.

Il est préférable de spécifier la taille du heap lors de l'utilisation de la JVM pour ne pas dépendre des valeurs déterminées par les ergonomics de la JVM.

De la même manière, la limitation de la CPU utilisable sur le conteneur peut avoir différents effets de bord. La JVM utilise le nombre de coeurs du système dans ses ergonomics. Le nombre de coeurs est obtenu du système hôte même si la JVM est exécutée dans un conteneur.

Si la JVM n'est pas capable d'obtenir les limitations en CPU dans cgroups, il faut préciser le nombre de threads utilisable par le ramasse-miette avec l'option -XX:ParallelGCThreads si le conteneur peut utiliser au moins 2 CPU. Si le conteneur ne peut utiliser qu'un seul CPU, il faut demander explicitement le ramasse-miettes Serial grâce à l'option -XX:+UseSerialGC pour éviter aux ergonomics de choisir un ramasse-miettes parallèle car l'hôte possède au moins 2 CPU.

66.3.2. Les évolutions dans Java SE 9 et Java SE 8u131

Java 8 update 131 (publié en avril 2017) inclut quelques améliorations pour faciliter la gestion de la mémoire et de la CPU pour une JVM exécutée dans un conteneur :

- Docker CPU limits - JDK-8140793
- Experimental support for Docker memory limits - JDK-8170888
- Docker memory limits - JDK-8146115

Java 9 propose plusieurs fonctionnalités expérimentales pour améliorer le support de Docker par la JVM. Ces fonctionnalités sont reportées de manière rétroactive dans Java 8u131 :

- -XX:+UseCGroupMemoryLimitForHeap : les ergonomics de la JVM utilise la valeur de cgroup.limit_in_bytes si elle est définie

- -XX:MaxRAMFraction

Comme elles sont expérimentales, il faut utiliser l'option -XX:+UnlockExperimentalVMOptions pour pouvoir les utiliser.

Java SE 9 et 8u131 propose une option expérimentale de la JVM pour récupérer la quantité de mémoire allouée au conteneur notamment lorsque celle-ci est limitée via cgroups. L'option -XX:+UseCGroupMemoryLimitForHeap demande à la JVM d'obtenir la taille maximale à partir de cgroups plutôt que du système hôte.

Si l'option -Xmx n'est pas utilisée, alors la JVM va rechercher la quantité de mémoire définie dans cgroups si des restrictions sont définies dans le conteneur.

Sans utiliser cette fonctionnalité, la JVM récupère toujours la taille de la RAM de la machine hôte (2Go pour la VM dans l'exemple ci-dessous) et ne tient pas compte des limitations configurées pour le conteneur.

Exemple :

```
$ docker run openjdk:8u131-jre-alpine java -XX:+PrintFlagsFinal -version | grep -E "(
InitialHeapSize|MaxHeapSize)"
    uintx InitialHeapSize             := 33554432        {product}
    uintx MaxHeapSize                 := 524288000       {product}
openjdk version "1.8.0_131"
OpenJDK Runtime Environment (IcedTea 3.4.0) (Alpine 8.131.11-r2)
OpenJDK 64-Bit Server VM (build 25.131-b11, mixed mode)

$ docker run -m 1g openjdk:8u131-jre-alpine java -XX:+PrintFlagsFinal -version | grep -E "(
InitialHeapSize|MaxHeapSize)"
    uintx InitialHeapSize             := 33554432        {product}
    uintx MaxHeapSize                 := 524288000       {product}
```

Pour activer cette fonctionnalité, il faut utiliser deux options de la JVM :

-XX:+UnlockExperimentalVMOptions et -XX:+UseCGroupMemoryLimitForHeap

Exemple :

```
$ docker run -m 1g openjdk:8u131-jre-alpine java -XX:+UnlockExperimentalVMOptions
-XX:+UseCGroupMemoryLimitForHeap -XX:+PrintFlagsFinal -version | grep -E "(
InitialHeapSize|MaxHeapSize)"
    uintx InitialHeapSize             := 16777216        {product}
    uintx MaxHeapSize                 := 268435456       {product}
```

Au lieu d'utiliser la valeur de l'option -XX:MaxRAM, l'option -XX:+UseCGroupMemoryLimitForHeap lit et utilise la valeur du fichier /sys/fs/cgroup/memory/memory.limit_in_bytes :

Exemple :

```
$ docker run --rm -m 512m openjdk:9.0.4-jre cat /sys/fs/cgroup/memory/memory.limit_in_bytes
536870912

$ docker run --rm -m 512m openjdk:9.0.4-jre sh -c "java -XX:+PrintFlagsFinal -version |
grep -Ei 'maxheapsize|maxram'"
openjdk version "9.0.4"
OpenJDK Runtime Environment (build 9.0.4+12-Debian-4)
OpenJDK 64-Bit Server VM (build 9.0.4+12-Debian-4, mixed mode)
   size_t MaxHeapSize           = 524288000      {product} {ergonomic}
 uint64_t MaxRAM                = 137438953472   {pd product} {default}
    uintx MaxRAMFraction        = 4              {product} {default}

$ docker run --rm -m 512m openjdk:9.0.4-jre sh -c "java -XX:+PrintFlagsFinal
 -XX:+UnlockExperimentalVMOptions -XX:+UseCGroupMemoryLimitForHeap -version |
 grep -Ei 'maxheapsize|maxram'"
openjdk version "9.0.4"
OpenJDK Runtime Environment (build 9.0.4+12-Debian-4)
OpenJDK 64-Bit Server VM (build 9.0.4+12-Debian-4, mixed mode)
   size_t MaxHeapSize        = 134217728         {product} {ergonomic}
 uint64_t MaxRAM             = 137438953472      {pd product} {default}
```

```
    uintx MaxRAMFraction     = 4                 {product} {default}
```

Dans l'exemple ci-dessus, sans activer l'option, la taille maximale du heap est définie à 512Mo ce qui correspond à ¼ des 2Go de RAM de la machine hôte. Avec l'option activée, la taille maximale du heap est définie à 128Mo ce qui correspond à ¼ de la limitation à 512Mo de RAM pour le conteneur.

Il est aussi possible d'utiliser l'option -XX:MaxRAMFraction pour indiquer à la JVM quelle fraction de la mémoire maximale peut être utilisable par la JVM. Cependant la valeur est exprimée par une fraction de la taille totale.

L'option MaxRAMFraction permet d'exprimer la quantité de RAM utilisable par la JVM sous la forme d'une fraction. La valeur à fournir doit donc être un nombre entier strictement supérieur à zéro. La fraction (1/MaxRAMFraction) ainsi définie correspond à une proportion de MaxRAM utilisable pour le heap :

- 1 : la totalité de la RAM
- 2 : la moitié
- 3 : le tiers
- 4 : le quart
- Et ainsi de suite avec un minimum environ à 8

Cette option de la JVM pour contrôler la taille maximale du heap utilise une fraction plutôt qu'un pourcentage, ce qui rend difficile la définition de valeurs qui permettraient d'utiliser efficacement la RAM disponible.

Exemples avec un conteneur dont la mémoire est limitée à 1Go :

- -XX:MaxRAMFraction=1 : la taille maximale du heap est 1Go. Attention dans ce cas, l'intégralité de la mémoire sera utilisée par la JVM, ce qui peut poser des problèmes pour exécuter d'autres processus dans le conteneur et même pour exécuter la JVM si elle a besoin de mémoire native
- -XX:MaxRAMFraction=2 : la taille maximale du heap est 500Mo. Dans ce cas, la moitié de la mémoire du conteneur est utilisée
- -XX:MaxRAMFraction=4 : la taille maximale du heap est 250Mo.
- Et ainsi de suite

Par exemple, pour une JVM configurée avec MaxRAM à 1Go et MaxRAMFraction à 2, les ergonomics définissent la taille maximale du heap à 524Mo.

Exemple :

```
java -XX:+PrintFlagsFinal -XX:MaxRAM=1g -XX:MaxRAMFraction=2 -version |
grep -Ei "maxheapsize|maxram"
   size_t MaxHeapSize          = 536870912    {product} {ergonomic}
 uint64_t MaxRAM               = 1073741824   {pd product} {command line}
    uintx MaxRAMFraction       = 2            {product} {command line}
java version "9.0.1"
Java(TM) SE Runtime Environment (build 9.0.1+11)
Java HotSpot(TM) 64-Bit Server VM (build 9.0.1+11, mixed mode)
```

Il n'est pas recommandé d'utiliser la valeur 1 pour l'option -XX:MaxRAMFraction ce qui demande à la JVM d'utiliser l'intégralité de la RAM pour le heap. Or la JVM a besoin de mémoire en dehors du heap pour fonctionner. Le conteneur peut aussi exécuter des processus comme un shell qui ont aussi besoin de mémoire.

Il est préférable d'avoir un OutOfMemoryError de la JVM, notamment pour permettre de faire un heap dump et de l'analyser que de laisser le démon Docker tuer le conteneur avec un OOMKilled.

Le fait de ne pas utiliser -XX:MaxRAMFraction=1 implique de n'avoir au mieux que la moitié au maximum de mémoire heap utilisable par la JVM.

Avec Java 9, la JVM d'OpenJDK est capable de déterminer l'utilisation de cpusets pour la limitation CPU du conteneur et d'exploiter cette valeur. Attention avec Java 9, la JVM n'est pas en mesure de détecter une limitation avec cpu_shares.

Si les valeurs déterminées ne sont pas celles souhaitées, il est toujours possible de définir explicitement certains paramètres de la JVM notamment en utilisant les options -Xmx, - XX:ParallelGCThreads, -XX:ConcGCThreads, . en adéquation avec les limites fixées au conteneur.

66.3.3. Les évolutions dans Java SE 10 et Java SE 8u191

Java 10 propose un meilleur support de Docker par la JVM : plusieurs améliorations permettent à la JVM de tenir compte des restrictions de ressources appliquées sur le conteneur dans lequel elle s'exécute. C'est la première version de Java avec une prise en compte réelle automatique de l'exécution d'une JVM dans un conteneur Docker.

La prise en compte de la limitation des ressources CPU et de mémoire à un conteneur dans lequel une JVM est exécutée est grandement facilitée : la JVM détecte correctement la configuration du conteneur pour l'utiliser dans ses ergonomics.

Sur un système hôte Linux, la JVM est par défaut capable de détecter qu'elle s'exécute dans un conteneur Docker. Si c'est le cas, elle est en mesure d'obtenir des informations sur les ressources utilisables (CPU et mémoire) par le conteneur notamment si des restrictions lui sont appliquées plutôt que du système hôte.

Ceci est possible grâce à la nouvelle option -XX:+UseContainerSupport, activée par défaut pour une JVM exécutée sous Linux.

Exemple :

```
$ docker run -m 1g openjdk:10.0.2-jre java -XX:+PrintFlagsFinal -version |
grep -E "(UseContainerSupport)"
     bool UseContainerSupport            = true         {product} {default}
openjdk version "10.0.2" 2018-07-17
OpenJDK Runtime Environment (build 10.0.2+13-Debian-2)
OpenJDK 64-Bit Server VM (build 10.0.2+13-Debian-2, mixed mode)
```

L'option UseCGroupMemoryLimitForHeap est dépréciée car l'option UseContainerSupport la remplace.

L'utilisation de l'option UseCGroupMemoryLimitForHeap affiche un avertissement : «warning: Option UseCGroupMemoryLimitForHeap was deprecated in version 10.0 and will likely be removed in a future release".

Exemple :

```
$ docker run -m 1g openjdk:10.0.2-jre java -XX:+UnlockExperimentalVMOptions
-XX:+UseCGroupMemoryLimitForHeap -XX:+PrintFlagsFinal -version |
grep -E "(InitialHeapSize|MaxHeapSize)"
OpenJDK 64-Bit Server VM warning: Option UseCGroupMemoryLimitForHeap was deprecated
in version 10.0 and will likely be removed in a future release.
   size_t InitialHeapSize           = 16777216       {product} {ergonomic}
   size_t MaxHeapSize               = 268435456      {product} {ergonomic}
openjdk version "10.0.2" 2018-07-17
OpenJDK Runtime Environment (build 10.0.2+13-Debian-2)
OpenJDK 64-Bit Server VM (build 10.0.2+13-Debian-2, mixed mode)

$ docker run -m 1g openjdk:10.0.2-jre java -XX:+PrintFlagsFinal -version |
grep -E "(InitialHeapSize|MaxHeapSize)"
   size_t InitialHeapSize           = 16777216       {product} {ergonomic}
   size_t MaxHeapSize               = 268435456      {product} {ergonomic}
```

Ce meilleur support par Java 10 facilite la configuration de la JVM via la ligne de commande ou le Dockerfile qui a moins besoin d'options dédiées pour une meilleure configuration par défaut.

La propriété UserContainerSupport est activée par défaut sous Linux. Il est possible de la désactiver en utilisant l'option -XX:-UseContainerSupport.

Trois Nouvelles options permettent d'améliorer le contrôle de la quantité de mémoire utilisable sous des valeurs exprimées en pourcentage :

- -XX:InitialRAMPercentage,
- -XX:MinRAMPercentage

- -XX:MaxRAMPercentage

Les options xxxRAMPercentage attendent une valeur comprise entre 0 et 100 qui permettent un contrôle très précis sur la quantité de mémoire utilisable pour le heap de la JVM

Par défaut avec un conteneur limité à 1Go, la taille minimale du heap est à 16Mo et la taille maximale à 256Mo.

Exemple :

```
$ docker run -m 1g openjdk:10.0.2-jre java -XX:+PrintFlagsFinal -version |
grep -E "(InitialHeapSize|MaxHeapSize)"
   size_t InitialHeapSize        = 16777216        {product} {ergonomic}
   size_t MaxHeapSize            = 268435456       {product} {ergonomic}
openjdk version "10.0.2" 2018-07-17
OpenJDK Runtime Environment (build 10.0.2+13-Debian-2)
OpenJDK 64-Bit Server VM (build 10.0.2+13-Debian-2, mixed mode)
```

Avec les options -XX:InitialRAMPercentage=10 et -XX:MaxRAMPercentage=80, la taille minimale du heap est à 104Mo et la taille maximale à 820Mo.

Exemple :

```
$ docker run -m 1g openjdk:10.0.2-jre java -XX:InitialRAMPercentage=10
-XX:MaxRAMPercentage=80 -XX:+PrintFlagsFinal -version | grep -E "(InitialHeapSize|MaxHeapSize)"
   size_t InitialHeapSize        = 109051904     {product} {ergonomic}
   size_t MaxHeapSize            = 859832320     {product} {ergonomic}
openjdk version "10.0.2" 2018-07-17
OpenJDK Runtime Environment (build 10.0.2+13-Debian-2)
OpenJDK 64-Bit Server VM (build 10.0.2+13-Debian-2, mixed mode)
```

Comme la JVM de Java 10 est capable de détecter son exécution dans un conteneur Docker, la prise en compte de la limitation de ressources ne requière plus de configuration spécifique.

Exemple :

```
$ docker run -it --entrypoint bash openjdk:10.0.2-jdk
root@1e39f4cf2236:/# jshell
Jul 20, 2019 8:01:59 AM java.util.prefs.FileSystemPreferences$1 run
INFO: Created user preferences directory.
|  Welcome to JShell -- Version 10.0.2
|  For an introduction type: /help intro

jshell> Runtime runtime = Runtime.getRuntime();
runtime ==> java.lang.Runtime@27efef64

jshell> runtime.availableProcessors();
$2 ==> 2
```

La limitation de la CPU utilisable via le CPU set est correctement prise en compte dans la JVM.

Exemple :

```
$ docker run -it --cpuset-cpus 0 --entrypoint bash openjdk:10.0.2-jdk
root@32d18e9680dd:/# jshell
Jul 20, 2019 8:07:58 AM java.util.prefs.FileSystemPreferences$1 run
INFO: Created user preferences directory.
|  Welcome to JShell -- Version 10.0.2
|  For an introduction type: /help intro

jshell> Runtime.getRuntime().availableProcessors()
$1 ==> 1
```

La limitation de la CPU utilisable via le CPU share est aussi correctement pris en compte dans la JVM.

Exemple :

```
$ docker run -it -c=512 --entrypoint bash openjdk:10.0.2-jdk
root@40ef1e702e0e:/# jshell
Jul 20, 2019 8:12:44 AM java.util.prefs.FileSystemPreferences$1 run
INFO: Created user preferences directory.
|  Welcome to JShell -- Version 10.0.2
|  For an introduction type: /help intro

jshell> Runtime.getRuntime().availableProcessors()
$1 ==> 1
```

Les fonctionnalités introduites dans Java 10 pour le support de l'exécution de la JVM dans des conteneurs Docker sont reportées dans l'update 191 de Java 8 :

- JDK-8146115 : sous Linux, l'option -XX:+UseContenainerSupport, activée par défaut, permet à la JVM de détecter son exécution dans un conteneur Docker et de récupérer correctement les limitations de ressources mémoire et CPU définies via la cgroups.
  Le nombre total de processeurs disponibles par la JVM est calculé à partir des cpu sets, cpu shares et cpu quota si utilisés : un algorithme utilise une combinaison de number_of_cpus() et cpu_sets() pour déterminer le nombre de processeurs disponibles et valoriser la valeur d'active_processor_count de la JVM. Le number_of_cpus() est calculé via la formule : number_of_cpus() = cpu_quota() / cpu_period(). Si cpu_shares a été configuré pour le conteneur, le number_of_cpus() est calculé avec la formule cpu_shares()/1024. 1024 est l'unité standard par défaut pour calculer l'utilisation relative des processeurs dans les processus exécutés dans des conteneurs
  Grâce à l'option -XX:ActiveProcessorCount, il est possible de forcer ce nombre de CPU utilisable par les ergonomics de la JVM.
  L'option -Xlog:os+container=trace permet de journaliser dans la sortie d'erreur des informations sur le conteneur.
- JDK-8186248 : les options -XX:InitialRAMPercentage, -XX:MaxRAMPercentage et -XX:MinRAMPercentage permettent de définir le pourcentage de RAM utilisé par le heap.
  Les options -XX:InitialRAMFraction, -XX:MaxRAMFraction et -XX:MinRAMFraction sont dépréciées
- JDK-8179498 : le mécanisme d'attachement à partir d'un hôte vers une JVM exécutée dans un conteneur utilisé par les outils de diagnostic utilise la valeur stockée dans /proc/pid/root et le namespace

66.3.4. Les évolutions dans Java SE 11

Une nouvelle option est ajoutée dans la JVM : -XX:+PreferContainerQuotaForCPUCount. Elle permet de demander la prise en compte de cpu_quota au lieu de cpu_shares pour déterminer le nombre de coeurs.

66.3.5. L'influence des ergonomics

Les ergonomics ont une influence sur les valeurs par défaut dans la JVM. Sur une machine hôte récente avec plusieurs CPU et Go de RAM, ces valeurs sont généralement largement suffisantes.

Lorsque l'on exécute plusieurs ou de nombreux conteneurs sur un même hôte, par exemple via un mécanisme d'orchestration, il est courant de limiter les ressources des conteneurs. Attention, ces limitations peuvent avoir des répercutions notamment sur les valeurs déterminées par les ergonomics.

Les exemples de cette section sont exécutés dans une VM disposant de 2 CPU et 2Go de RAM.

Le ramasse-miettes utilisé par défaut dépend du nombre de coeurs et de la RAM disponible.

Si le nombre de coeurs est supérieure ou égal à 2 alors le ramasse-miettes G1 est utilisé par la JVM Hotpost d'un JDK 11.

Exemple :

```
$ docker run --cpus="2" adoptopenjdk:11.0.3_7-jre-hotspot java -XX:+PrintFlagsFinal -version |
grep -E "(UseSerialGC|UseG1GC|MaxHeapSize)"
   size_t MaxHeapSize              = 524288000         {product} {ergonomic}
```

```
     bool UseG1GC                 = true              {product} {ergonomic}
     bool UseSerialGC             = false             {product} {default}
openjdk version "11.0.3" 2019-04-16
OpenJDK Runtime Environment AdoptOpenJDK (build 11.0.3+7)
OpenJDK 64-Bit Server VM AdoptOpenJDK (build 11.0.3+7, mixed mode)
```

Si le nombre de coeurs est inférieure à 2 alors le ramasse-miettes SerialGC est utilisé par la JVM Hotpost d'un JDK 11 puisque le nombre de coeurs est insuffisant pour envisager des traitements en parallèle des activités du ramasse-miettes.

Exemple :

```
$ docker run --cpus="1" adoptopenjdk:11.0.3_7-jre-hotspot java -XX:+PrintFlagsFinal -version |
grep -E "(UseSerialGC|UseG1GC|MaxHeapSize)"
   size_t MaxHeapSize              = 524288000        {product} {ergonomic}
     bool UseG1GC                  = false            {product} {default}
     bool UseSerialGC              = true             {product} {ergonomic}
openjdk version "11.0.3" 2019-04-16
OpenJDK Runtime Environment AdoptOpenJDK (build 11.0.3+7)
OpenJDK 64-Bit Server VM AdoptOpenJDK (build 11.0.3+7, mixed mode)
```

Selon la taille de la RAM (+/- 2Go) alors G1 ou SerialGC est utilisé par la JVM Hotpost d'un JDK 11.

Exemple :

```
$ docker run -m 1.8g adoptopenjdk:11.0.3_7-jre-hotspot java -XX:+PrintFlagsFinal -version |
grep -E "(UseSerialGC|UseG1G
C|MaxHeapSize)"
   size_t MaxHeapSize              = 484442112        {product} {ergonomic}
     bool UseG1GC                  = true             {product} {default}
     bool UseSerialGC              = false            {product} {ergonomic}
openjdk version "11.0.3" 2019-04-16
OpenJDK Runtime Environment AdoptOpenJDK (build 11.0.3+7)
OpenJDK 64-Bit Server VM AdoptOpenJDK (build 11.0.3+7, mixed mode)

$ docker run -m 1.7g adoptopenjdk:11.0.3_7-jre-hotspot java -XX:+PrintFlagsFinal -version |
grep -E "(UseSerialGC|UseG1G
C|MaxHeapSize)"
   size_t MaxHeapSize              = 457179136        {product} {ergonomic}
     bool UseG1GC                  = false            {product} {default}
     bool UseSerialGC              = true             {product} {ergonomic}
openjdk version "11.0.3" 2019-04-16
OpenJDK Runtime Environment AdoptOpenJDK (build 11.0.3+7)
OpenJDK 64-Bit Server VM AdoptOpenJDK (build 11.0.3+7, mixed mode)
```

La taille max par défaut du heap dépend de la taille totale de la RAM. A partir de 2Go de RAM, la taille max du heap est fixé à ¼ de la taille de la RAM. En dessous, la JVM utilise un ratio différent qui dépend de la taille de la RAM.

Exemple :

```
$ docker run -m 2g adoptopenjdk:11.0.3_7-jre-hotspot java -XX:+PrintFlagsFinal -version |
grep -E "MaxHeapSize"
   size_t MaxHeapSize            = 536870912        {product} {ergonomic}

$ docker run -m 1g adoptopenjdk:11.0.3_7-jre-hotspot java -XX:+PrintFlagsFinal -version |
grep -E "MaxHeapSize"
   size_t MaxHeapSize            = 268435456        {product} {ergonomic}

$ docker run -m 512m adoptopenjdk:11.0.3_7-jre-hotspot java -XX:+PrintFlagsFinal -version |
grep -E "MaxHeapSize"
   size_t MaxHeapSize            = 134217728        {product} {ergonomic}

$ docker run -m 256m adoptopenjdk:11.0.3_7-jre-hotspot java -XX:+PrintFlagsFinal -version |
grep -E "MaxHeapSize"
   size_t MaxHeapSize            = 132120576        {product} {ergonomic}

$ docker run -m 128m adoptopenjdk:11.0.3_7-jre-hotspot java -XX:+PrintFlagsFinal -version |
grep -E "MaxHeapSize"
```

```
   size_t MaxHeapSize            = 67108864         {product} {ergonomic}

$ docker run -m 64m adoptopenjdk:11.0.3_7-jre-hotspot java -XX:+PrintFlagsFinal -version |
grep -E "MaxHeapSize"
   size_t MaxHeapSize            = 33554432         {product} {ergonomic}
```

66.4. Le support de cgroups v2

A partir de sa version 20.10 de l'engine, Docker utilise cgroups (control groups) v2 à la place de cgroups v1.

Java 15 propose un support de cgroups v2 (JDK-8230305), les versions précédentes de Java utilisaient cgroups v1. Un backport a été effectué dans les versions 11.0.16 et 8u372 de Java.

Si la version de Java utilisée ne supporte que cgroups v1 et que le moteur d'exécution des conteneurs utilise cgroups v2, alors la détermination de la mémoire de l'environnement d'exécution par la JVM sera erronée ce qui pourra induire de OOM Kill par l'orchestrateur.

Il est possible de forcer (temporairement) le moteur Docker à utiliser cgroups v1 en utilisant l'option "deprecatedCgroupv1": true dans le fichier de configuration.

67. La décompilation et l'obfuscation

Chapitre 67

Niveau : 🟩🟩🟨🟨🟥⬜ Confirmé

Le compilateur transforme un fichier source en fichier de classe contenant du bytecode. Ce bytecode est ensuite lu et interprété par la JVM.

La décompilation consiste à générer du code source à partir du bytecode pour effectuer un reverse engineering. Un des outils pionniers dans cette activité est Mocha qui a fait couler beaucoup d'encre.

L'obfuscation consiste à rendre le résultat d'une décompilation difficilement lisible voire impossible.

Ce chapitre contient plusieurs sections :

- Décompiler du bytecode
- Obfusquer le bytecode

67.1. Décompiler du bytecode

La décompilation consiste à produire un fichier source Java à partir d'un fichier de classe contenant du bytecode. C'est l'opération inverse de la compilation. Ce processus est possible car le bytecode est standardisé et parfaitement documenté.

Attention : ce processus est généralement prohibé pour du code dont on n'est pas l'auteur ou qui n'est pas open source. Avant de réaliser une décompilation, il est important de se renseigner sur la licence du code qui va subir cette opération afin de ne pas enfreindre la licence d'utilisation.

La décompilation est possible parce que la compilation du code source ne produit pas du code machine binaire mais produit du bytecode qui est un langage indépendant de toute plate-forme. Lors de son exécution, le bytecode peut être interprété ou compilé en code machine. Le format du bytecode est assez proche du code source, ce qui permet de réaliser une décompilation relativement facilement notamment pour ce qui concerne la logique des traitements.

Il existe plusieurs outils pour décompiler du bytecode :

Outils	Url
JReversePro	http://jrevpro.sourceforge.net/
Jad (the fast Java Decompiler)	http://www.kpdus.com/jad.html
IdeaJad (utilise Jad)	https://www.tagtraum.com/ideajad.html
JODE	http://jode.sourceforge.net/

67.1.1. JAD : the fast Java Decompiler

Jad est un décompilateur gratuit pour un usage non commercial ou personnel qui est particulièrement efficace et véloce car il est écrit en C++.

Il faut télécharger le fichier jadnt158.zip à l'url http://www.kpdus.com/jad.html et décompresser l'archive dans un répertoire du système. Le plus simple est d'ajouter ce répertoire à la variable Path du système.

La classe ci-dessous est utilisée comme exemple :

Exemple :

```
package fr.jmdoudoux.dej;

public class MaClasse {

  /**
   * @param args
   */
  public static void main(
      String[] args) {
    System.out.println("Bonjour");
  }

}
```

Exécuter jad en lui passant en paramètre le nom du fichier .class à décompiler.

Exemple :

```
C:\Documents and Settings\jmd\workspace\Tests\bin\com\jmdoudoux\test>jad MaClass
e.class
Parsing MaClasse.class... Generating MaClasse.jad
```

L'exécution produit un fichier MaClasse.jad.

Exemple :

```
// Decompiled by Jad v1.5.8g. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.kpdus.com/jad.html
// Decompiler options: packimports(3)
// Source File Name:   MaClasse.java

package fr.jmdoudoux.dej;

import java.io.PrintStream;

public class MaClasse
{

    public MaClasse()
    {
    }

    public static void main(String args[])
    {
        System.out.println("Bonjour");
    }
}
```

67.1.2. La mise en oeuvre et les limites de la décompilation

Cette section va utiliser la classe ci-dessous :

Exemple :

```
package fr.jmdoudoux.dej.decompile;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;

/**
 * Classe de test
 *
 */
public class MaClasse {

  private String nom;
  protected String prenom;
  public Date dateNaissance;
  public List commandes = new ArrayList();

  public static void main(
      String[] args) {
    MaClasse maClasse = new MaClasse("nom1","prenom1",new Date());
    maClasse.ajouterCommande("commande 1");
    maClasse.ajouterCommande("commande 2");
    System.out.println(maClasse);
  }

  /**
   * Constructeur
   * @param nom
   * @param prenom
   * @param dateNaissance
   */
  public MaClasse(String nom, String prenom, Date dateNaissance) {
    super();
    this.nom = nom;
    this.prenom = prenom;
    this.dateNaissance = dateNaissance;
  }

  /**
   * Ajouter une commande
   * @param libelle libelle de la commande
   */
  public void ajouterCommande(String libelle) {
    commandes.add(libelle);
  }

  @Override
  public String toString() {
    StringBuilder sb = new StringBuilder("");
    sb.append("Nom : ");
    sb.append(nom);
    sb.append("\n");
    sb.append("Prenom : ");
    sb.append(prenom);
    sb.append("\n");
    sb.append("Date de naissance : ");
    sb.append(dateNaissance);
    sb.append("\n");
    sb.append("Commandes :\n");
    for(Object commande : commandes) {
      sb.append("  ");
      sb.append(commande);
      sb.append("\n");
    }

    return sb.toString();
  }
}
```

Exemple : décompilation avec jad

Exemple :

```
C:\java\tests>javac com/jmdoudoux/test/decompile/MaClasse.java
Note: com\jmdoudoux\test\decompile\MaClasse.java uses unchecked or unsafe operat
ions.
Note: Recompile with -Xlint:unchecked for details.

C:\java\tests>cd com/jmdoudoux/test/decompile

C:\java\tests\com\jmdoudoux\test\decompile>dir
 Volume in drive C has no label.
 Volume Serial Number is 1F23-7A9

 Directory of C:\java\tests\com\jmdoudoux\test\decompile

01/04/2008  10:15    <DIR>          .
01/04/2008  10:15    <DIR>          ..
01/04/2008  10:15             1 755 MaClasse.class
01/04/2008  09:58             1 545 MaClasse.java
               2 File(s)          3 300 bytes
               2 Dir(s)  57 852 260 352 bytes free

C:\java\tests\com\jmdoudoux\test\decompile>jad MaClasse.class
Parsing MaClasse.class... Generating MaClasse.jad
```

Exemple : le fichier MaClasse.jad généré

```
// Decompiled by Jad v1.5.8g. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.kpdus.com/jad.html
// Decompiler options: packimports(3)
// Source File Name:   MaClasse.java

package fr.jmdoudoux.dej.decompile;

import java.io.PrintStream;
import java.util.*;

public class MaClasse
{

    public static void main(String args[])
    {
        MaClasse maclasse = new MaClasse("nom1", "prenom1", new Date());
        maclasse.ajouterCommande("commande 1");
        maclasse.ajouterCommande("commande 2");
        System.out.println(maclasse);
    }

    public MaClasse(String s, String s1, Date date)
    {
        commandes = new ArrayList();
        nom = s;
        prenom = s1;
        dateNaissance = date;
    }

    public void ajouterCommande(String s)
    {
        commandes.add(s);
    }

    public String toString()
    {
        StringBuilder stringbuilder = new StringBuilder("");
        stringbuilder.append("Nom : ");
        stringbuilder.append(nom);
        stringbuilder.append("\n");
        stringbuilder.append("Prenom : ");
        stringbuilder.append(prenom);
        stringbuilder.append("\n");
        stringbuilder.append("Date de naissance : ");
        stringbuilder.append(dateNaissance);
        stringbuilder.append("\n");
        stringbuilder.append("Commandes :\n");
        for(Iterator iterator = commandes.iterator();
          iterator.hasNext(); stringbuilder.append("\n"))
        {
            Object obj = iterator.next();
            stringbuilder.append("  ");
            stringbuilder.append(obj);
        }

        return stringbuilder.toString();
    }

    private String nom;
    protected String prenom;
    public Date dateNaissance;
    public List commandes;
}
```

Le code décompilé est similaire au code source original exceptés :

- Le nom des variables
- L'ordre de déclaration des membres
- Les commentaires sont absents
- Le formatage est différent
- L'initialisation des attributs est déplacée dans le constructeur
- Certaines fonctionnalités de Java 5 ne sont pas décompilées à l'identique de l'original

Les fonctionnalités de Java 5 sont rarement décompilées à l'identique de l'original car ces fonctionnalités sont des raccourcis syntaxiques qui sont traités par le compilateur pour générer du code compatible avec les versions précédentes (l'annotation @Override est absente, la boucle for est étendue). La décompilation restitue le code tel qu'il a été généré par le compilateur à partir du bytecode : c'est notamment le cas dans l'exemple de la boucle for.

Si les informations de débogage sont incluses dans le bytecode lors de la compilation, alors le résultat de la décompilation est plus proche du code source original notamment la décompilation pourra restituer le nom des variables locales et des paramètres des méthodes. Pour demander l'ajout des informations de débogage, il faut utiliser l'option -g du compilateur.

Exemple :

```
C:\java\tests>javac -g com/jmdoudoux/test/decompile/MaClasse.java
Note: com\jmdoudoux\test\decompile\MaClasse.java uses unchecked or unsafe operat
ions.
Note: Recompile with -Xlint:unchecked for details.

C:\java\tests>cd com/jmdoudoux/test/decompile

C:\java\tests\com\jmdoudoux\test\decompile>jad MaClasse.class
Parsing MaClasse.class...Overwrite MaClasse.jad [y/n/a/s] ? y
 Generating MaClasse.jad
```

Exemple : le fichier MaClass.jad généré

```
// Decompiled by Jad v1.5.8g. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.kpdus.com/jad.html
// Decompiler options: packimports(3)
// Source File Name:   MaClasse.java

package fr.jmdoudoux.dej.decompile;

import java.io.PrintStream;
import java.util.*;

public class MaClasse
{

    public static void main(String args[])
    {
        MaClasse maClasse = new MaClasse("nom1", "prenom1", new Date());
        maClasse.ajouterCommande("commande 1");
        maClasse.ajouterCommande("commande 2");
        System.out.println(maClasse);
    }

    public MaClasse(String nom, String prenom, Date dateNaissance)
    {
        commandes = new ArrayList();
        this.nom = nom;
        this.prenom = prenom;
        this.dateNaissance = dateNaissance;
    }

    public void ajouterCommande(String libelle)
    {
        commandes.add(libelle);
    }

    public String toString()
    {
        StringBuilder sb = new StringBuilder("");
        sb.append("Nom : ");
        sb.append(nom);
        sb.append("\n");
        sb.append("Prenom : ");
        sb.append(prenom);
        sb.append("\n");
        sb.append("Date de naissance : ");
        sb.append(dateNaissance);
        sb.append("\n");
        sb.append("Commandes :\n");
        for(Iterator i$ = commandes.iterator(); i$.hasNext(); sb.append("\n"))
        {
            Object commande = i$.next();
            sb.append("  ");
            sb.append(commande);
        }

        return sb.toString();
    }

    private String nom;
    protected String prenom;
    public Date dateNaissance;
    public List commandes;
}
```

67.2. Obfusquer le bytecode

Pour diverses raisons, il n'est pas toujours souhaitable de proposer le code source ou de permettre son obtention grâce à une décompilation, notamment pour protéger des droits sur la propriété intellectuelle.

Il existe plusieurs outils pour obfusquer le bytecode produit par le compilateur. Plusieurs outils open source ou gratuits sont utilisables pour réaliser ce type d'opérations.

Outils	Url
ProGuard	https://www.guardsquare.com/proguard
RetroGuard (Plusieurs licences dont une open source)	http://www.retrologic.com/
yGuard	https://www.yworks.com/products/yguard
JavaGuard (Plus d'évolution depuis 2002)	https://sourceforge.net/projects/javaguard/
jarg (Plus d'évolution depuis 2003)	http://jarg.sourceforge.net/
JODE (Plus d'évolution depuis 2004)	http://jode.sourceforge.net/

Il existe aussi plusieurs outils commerciaux dont un des plus puissants est Klassmaster de Zelix (http://www.zelix.com/klassmaster/)

67.2.1. Le mode de fonctionnement de l'obfuscation

L'obfuscation rend parfois la décompilation impossible ou le code source produit non compilable mais plus généralement elle rend le code source issu de la décompilation très peut lisible et donc difficilement compréhensible.

L'obfuscation consiste donc à transformer le bytecode pour le rendre le moins compréhensible par un humain suite à un processus de décompilation.

Il n'existe pas de standard concernant l'obfuscation et chaque outil propose ses propres mécanismes pour obtenir un niveau de protection plus ou moins élevé. Ces mécanismes peuvent inclure entre autres :

- La suppression des informations de débogage (nom des variables, numéro de lignes, ...) : ces informations ne sont pas nécessaires à l'exécution de la classe mais sont utilisées par les débogueurs et les outils de décompilation. Si elles sont présentes le décompilateur les utilise dans le code source généré sinon il génère des noms automatiquement, le plus souvent composés d'une lettre et d'un chiffre.
- Renommage des packages, des classes, des méthodes : l'utilisation de noms explicites est importante pour le développement et la maintenance du code mais ils sont inutiles pour la JVM. Ainsi si l'obfuscateur renomme ces entités avec des noms générés, cela rend la compréhension plus difficile suite à un processus de décompilation. De nombreuses classes, méthodes et variables avec le même nom rendent le code particulièrement difficile à comprendre. L'obfuscation peut aussi exploiter le polymorphisme : plusieurs méthodes ayant des noms différents avec des paramètres et une valeur de retour différents peuvent être renommées avec le même nom.
- Encodage des chaînes de caractères : comme les chaînes de caractères sont stockées telles quelles dans le bytecode, elles sont facilement identifiables. L'obfuscation peut les encoder pour les rendre illisibles.
- Modification du flux de contrôles des traitements : l'obfuscation peut altérer le flux de contrôles des traitements notamment en faisant usage de l'instruction goto. La lecture des traitements décompilés est ainsi plus difficile à suivre car le code source devient du code « spaghetti »
- Insertion de bytecode « bogué » qui n'est jamais exécuté mais qui empêche la décompilation. Ce bytecode exploite généralement quelques flous dans les spécifications de la JVM.

Cette transformation doit cependant garantir que le bytecode modifié est toujours valide et surtout que les fonctionnalités soient toujours les mêmes.

L'outil d'obfuscation charge le fichier .class, analyse la structure et le bytecode, applique les transformations et sauvegarde le résultat dans un nouveau fichier .class qui est différent de l'original mais qui doit proposer exactement les mêmes fonctionnalités.

Il est possible que l'obfuscation rende le résultat d'une décompilation non compilable grâce à l'exploitation des spécifications de Java. Une des techniques consiste à renommer des entités pour les rendre ambigües à la compilation. Au chargement d'un fichier .class le bytecode est vérifié mais certaines vérifications ne sont faites que par le compilateur et ne sont pas reproduites au chargement de la classe. Ainsi le bytecode obfusqué est exécuté dans la JVM mais le résultat d'une décompilation ne se recompile pas.

La plupart des outils d'obfuscation réalisent durant leur traitement une opération de shrinking qui consiste à supprimer les portions de code inutilisées : ceci permet de réduire la taille du bytecode. Certains outils d'obfuscation proposent aussi d'optimiser le bytecode.

L'opération d'obfuscation rend moins facile l'exploitation des piles d'appels des exceptions. La plupart des outils d'obfuscation fournissent une solution pour restituer la pile d'appels telle qu'elle serait affichée avec le code non obfusqué.

67.2.2. Un exemple de mise en oeuvre avec ProGuard

ProGuard est un outil open source sous licence GPL écrit en Java qui permet d'effectuer plusieurs opérations sur une application packagée :

- Shrinker qui supprime les classes, les méthodes et les champs inutilisés
- Optimisation du bytecode en supprimant les instructions inutilisées
- Obfuscation en supprimant les informations de débogage et en renommant les classes, méthodes et les champs lorsque cela est possible
- Prévérification du bytecode pour Java 6 et Java ME

Pour lancer Proguard en ligne de commande, il faut exécuter la commande

java -jar proguard.jar [options ...]

Le fichier proguard.jar se trouve dans le sous-répertoire lib de ProGuard.

Pour faciliter la gestion des options, il est possible de les regrouper dans un fichier de configuration. Ce fichier de configuration peut facilement être créé avec l'interface graphique fournie par ProGuard (proguardgui).

Exemple partiel du fichier config.pro :

```
-injars 'C:\java\test.jar'
-outjars 'C:\java\test.jar'

-libraryjars 'C:\Program Files\Java\jre1.6.0_05\lib\rt.jar'

# Keep - Applications. Keep all application classes, along with their 'main'
# methods.
-keepclasseswithmembers public class * {
    public static void main(java.lang.String[]);
}

# Keep - Library. Keep all public and protected classes, fields, and methods.
-keep public class * {
    public protected <fields>;
    public protected <methods>;
}
...
```

Pour lancer l'application avec un fichier de configuration, il suffit de le préciser en paramètre précédé d'un caractère @.

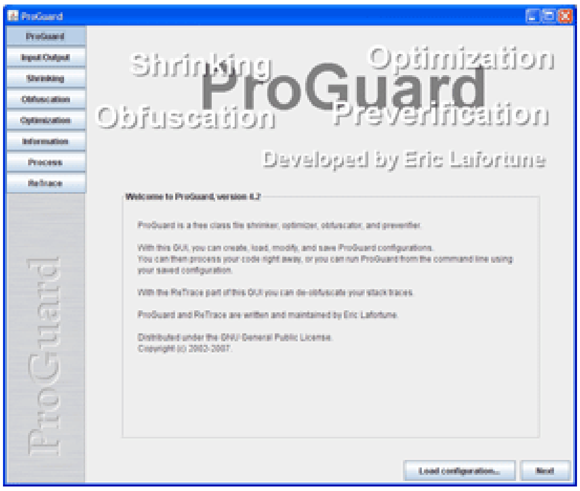
Exemple :

```
java -jar proguard.jar @config.pro
```

Proguard peut être utilisé avec une interface graphique. Pour lancer cette interface graphique, il faut saisir dans le répertoire lib de ProGuard la commande :

Exemple :

```
C:\java\proguard4.2\lib>java -jar proguardgui.jar
```



Welcome to ProGuard, version 4.2

ProGuard is a free class file shrinker, optimizer, obfuscator, and preverifier.

With this GUI, you can create, load, modify, and save ProGuard configurations.
You can then process your code right away, or you can run ProGuard from the command line using your saved configuration.

With the ReTrace part of this GUI you can de-obfuscate your stack traces.

ProGuard and ReTrace are written and maintained by Eric Lafortune.

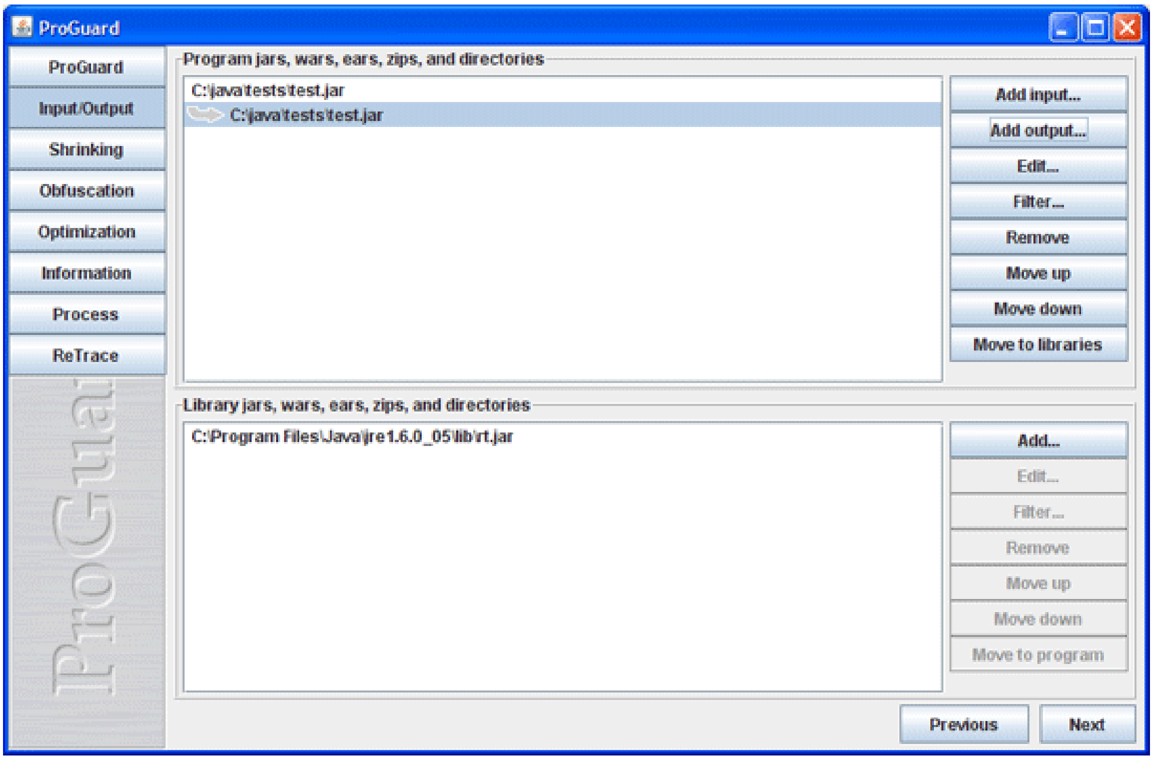
Distributed under the GNU General Public License.
Copyright (c) 2002-2007.

Pour utiliser ProGuard, il faut packager les fichiers .class dans une archive (de type jar, war, ...)

Exemple :

```
C:\java\tests>jar -cvfm test.jar manifest.mf com
manifest ajoutï¿½
ajout : com/ (entrï¿½e = 0) (sortie = 0) (0% stockï¿½)
ajout : com/jmdoudoux/ (entrï¿½e = 0) (sortie = 0) (0% stockï¿½)
ajout : com/jmdoudoux/test/ (entrï¿½e = 0) (sortie = 0) (0% stockï¿½)
ajout : com/jmdoudoux/test/decompile/ (entrï¿½e = 0) (sortie = 0) (0% stockï¿½)
ajout : com/jmdoudoux/test/decompile/MaClasse.class (entrï¿½e = 1755) (sortie = 97
1) (44% compressï¿½s)
ajout : com/jmdoudoux/test/decompile/MaClasse.java (entrï¿½e = 1545) (sortie = 535
) (65% compressï¿½s)
```

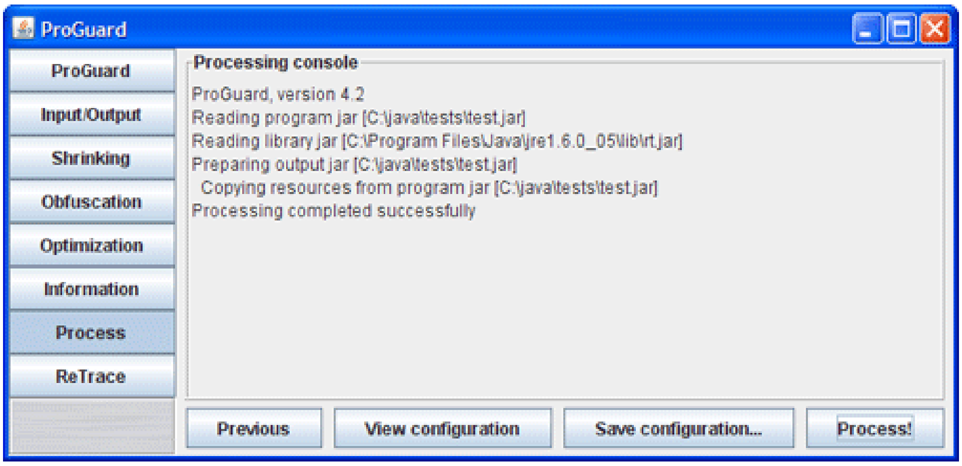
Cliquez sur le bouton « Input/ouput », puis sur le bouton « Add input... » et sélectionnez le fichier test.jar précédemment créé. Cliquez sur le bouton « Add Output » et sélectionnez le fichier test.jar.



C:\java\tests\test.jar
C:\java\tests\test.jar

C:\Program Files\Java\jre1.6.0_05\lib\rt.jar

Cliquez sur le bouton « Process » puis sur le bouton « Process ! »



```
ProGuard, version 4.2
Reading program jar [C:\java\tests\test.jar]
Reading library jar [C:\Program Files\Java\jre1.6.0_05\lib\rt.jar]
Preparing output jar [C:\java\tests\test.jar]
  Copying resources from program jar [C:\java\tests\test.jar]
Processing completed successfully
```

Résultat de l'exécution :

```
C:\java\tests>dir
 Volume in drive C has no label.
 Volume Serial Number is 1F23-7A9

 Directory of C:\java\tests

01/04/2008  15:31    <DIR>          .
01/04/2008  15:31    <DIR>          ..
31/03/2008  10:05    <DIR>          com
01/04/2008  15:29                51 manifest.mf
01/04/2008  15:31             2 680 test.jar
               3 File(s)          2 806 bytes
               3 Dir(s)  57 784 393 728 bytes free

C:\java\tests>dir
 Volume in drive C has no label.
 Volume Serial Number is 1F23-7A9

 Directory of C:\java\tests

01/04/2008  15:31    <DIR>          .
01/04/2008  15:31    <DIR>          ..
31/03/2008  10:05    <DIR>          com
01/04/2008  15:29                51 manifest.mf
01/04/2008  15:47             1 954 test.jar
               3 File(s)          2 080 bytes
               3 Dir(s)  57 783 062 528 bytes free
```

Pour vérifier le travail effectué par ProGuard, il faut décompiler le fichier MaClass.class obfusqué.

Exemple :

```
C:\java\tests>mkdir temp

C:\java\tests>copy test.jar temp
        1 file(s) copied.

C:\java\tests>cd temp

C:\java\tests\temp>dir
 Volume in drive C has no label.
 Volume Serial Number is 1F23-7A9

 Directory of C:\java\tests\temp

01/04/2008  15:49    <DIR>          .
01/04/2008  15:49    <DIR>          ..
01/04/2008  15:47             1 954 test.jar
               1 File(s)          1 954 bytes
               2 Dir(s)  57 783 058 432 bytes free

C:\java\tests\temp>jar -xvf test.jar
```

```
dï¿½compressï¿½e: META-INF/MANIFEST.MF
dï¿½compressï¿½e: com/jmdoudoux/test/decompile/MaClasse.class
dï¿½compressï¿½e: com/jmdoudoux/test/decompile/MaClasse.java

C:\java\tests\temp>cd com/jmdoudoux/test/decompile

C:\java\tests\temp\com\jmdoudoux\test\decompile>jad MaClasse.class
Parsing MaClasse.class... Generating MaClasse.jad
```

Exemple :

```
// Decompiled by Jad v1.5.8g. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.kpdus.com/jad.html
// Decompiler options: packimports(3)

package fr.jmdoudoux.dej.decompile;

import java.io.PrintStream;
import java.util.*;

public class MaClasse
{

    public static void main(String args[])
    {
        (args = new MaClasse("nom1", "prenom1", new Date())).a("commande 1");
        args.a("commande 2");
        System.out.println(args);
    }

    private MaClasse(String s, String s1, Date date)
    {
        d = new ArrayList();
        a = s;
        b = s1;
        c = date;
    }

    private void a(String s)
    {
        d.add(s);
    }

    public final String toString()
    {
        StringBuilder stringbuilder;
        (stringbuilder = new StringBuilder("")).append("Nom : ");
        stringbuilder.append(a);
        stringbuilder.append("\n");
        stringbuilder.append("Prenom : ");
        stringbuilder.append(b);
        stringbuilder.append("\n");
        stringbuilder.append("Date de naissance : ");
        stringbuilder.append(c);
        stringbuilder.append("\n");
        stringbuilder.append("Commandes :\n");
        for(this = d.iterator(); hasNext(); stringbuilder.append("\n"))
        {
            Object obj = next();
            stringbuilder.append("  ");
            stringbuilder.append(obj);
        }

        return stringbuilder.toString();
    }

    private String a;
    private String b;
    private Date c;
    private List d;
}
```

67.2.3. Les problèmes possibles lors de l'obfuscation

L'obfuscation rend le traitement des bugs d'exploitation beaucoup plus difficile. Par exemple, un moyen efficace de comprendre et isoler un problème est d'utiliser la pile d'appels (stacktrace) d'une exception qui contient les appels des différentes méthodes. Si le nom des méthodes a été modifié, la pile d'appels devient beaucoup plus difficile à exploiter vis-à-vis du code source. La pile d'appels peut aussi être plus efficace si elle exploite les informations de débogage. Hors généralement, ces informations sont supprimées lors de l'obfuscation.

L'obfuscation doit garantir que le bytecode obfusqué propose les mêmes fonctionnalités que le bytecode initial. Cependant les transformations réalisées par les outils d'obfuscation peuvent parfois avoir des effets de bords importants notamment avec certaines technologies de Java :

- L'introspection repose sur l'accès dynamique à certaines entités grâce à leurs noms. La modification de ces noms entraine inévitablement des problèmes lors de l'utilisation de l'introspection à l'exécution.
- Le chargement dynamique de classe par les méthodes Class.forName() ou ClassLoader.loadClass() utilise le nom de la classe pour s'exécuter. Si la classe est renommée, cela lèvera une exception de type ClassNotFoundException à l'exécution. Ceci est d'autant plus vrai si le nom de la classe n'est pas fourni en dur mais contenu dans une variable dont la valeur est déterminée dynamiquement (par exemple à la lecture d'un fichier de configuration)
- La sérialisation d'un objet inclus des informations sur la classe. Si la classe ou son numéro de version SerialIUD sont modifiés cela empêche la désérialisation. Ainsi il n'est pas possible de sérialiser un objet et de le désérialiser avec sa version obfusquée.
- Certaines API nécessitent le respect de conventions de nommage strictes de certaines méthodes (exemple avec les EJB avant leur version 3.0 : les méthodes ejbCreate() et ejbRemove())

67.2.4. L'utilisation d'un ClassLoader dédié

Pour rendre la décompilation plus difficile, il est possible d'encoder les fichiers .class avec un algorithme de cryptage et d'utiliser un ClassLoader dédié qui va décrypter ces fichiers avant de les charger en mémoire.

Ainsi, les fichiers .class ne peuvent plus être décompilés puisque le bytecode est illisible. Cependant cette technique est loin d'être infaillible car il suffit de décompiler le ClassLoader pour obtenir l'algorithme de décryptage et de l'utiliser pour décrypter les fichiers .class qui pourront ainsi être décompilés.

68. Programmation orientée aspects (AOP)

Chapitre 68

Niveau : 🟩🟩🟨⬜⬜⬜ Intermédiaire

AOP est l'acronyme d'Aspect Oriented Programming: la traduction française est programmation orientée aspects. AOSD (Aspect Oriented Software Development) est aussi utilisé.

L'AOP a été créée par Gregor Kickzales pour le laboratoire Xerox en 1996. L'AOP propose une façon de mettre en oeuvre certaines fonctionnalités de façon élégante et pratique.

Le développement orienté aspects permet la mise en oeuvre de la séparation des préoccupations (separation of concerns : SOC). Il s'adresse essentiellement à des fonctionnalités transverses.

L'AOP est un type de programmation qui propose de séparer le code technique du code métier d'une application pour des préoccupations transversales qu'elles soient techniques ou architecturales. L'AOP enrichit un modèle de programmation (POO ou procédurale) mais ne le remplace pas.

L'AOP ne remplace donc pas la POO mais la complète en proposant des solutions mises en oeuvre de façon élégante à certaines de ses limitations ou fonctionnalités manquantes. AOP permet de facilement implémenter des fonctionnalités transverses de façon modulaire. Traditionnellement, ces fonctionnalités sont partiellement mises en oeuvre sans AOP en utilisant des design patterns, des frameworks ou des outils (générateurs, précompilateurs).

L'idée principale de l'AOP est de considérer qu'un système est mieux développé lorsque l'implémentation de ses fonctionnalités est séparée notamment celles qui concernent des fonctionnalités techniques et transverses.

Les composants ou entités développés dans une application ont généralement besoin de plusieurs fonctionnalités transverses purement techniques : log, habilitation, gestion de transactions, ... Ces fonctionnalités ajoutent du code dans les traitements ce qui les alourdit.

La programmation orientée aspect propose d'externaliser ces fonctionnalités sous la forme d'aspects.

L'AOP est un concept qui est indépendant de tout langage : il est possible de fournir des implémentations d'AOP dans différents langages.

Ce chapitre contient plusieurs sections :

- Le besoin d'un autre modèle de programmation
- Les concepts de l'AOP
- La mise en oeuvre de l'AOP
- Les avantages et les inconvénients
- Des exemples d'utilisation
- Des implémentations pour la plate-forme Java

68.1. Le besoin d'un autre modèle de programmation

Généralement le code d'une application peut être regroupé dans deux catégories :

- les traitements fonctionnels qui permettent à l'application de répondre aux besoins pour lesquels elle a été développée
- les traitements techniques

Le développement d'applications évolue avec différents modèles de programmation pour faciliter le développement d'applications de plus en plus complexes :

Modèle	Préoccupation	Elément
Programmation linéaire		
Programmation structurée	Flot de contrôle	Instructions
Programmation procédurale	Découper le code en portions	Fonction, procédure
Programmation orientée objets	Données sous la forme d'objets	Classe
Programmation orientée aspects	Fonctionnalités transverses	Aspect

Le but de ces différents modèles est d'améliorer la structuration du code d'une application afin de faciliter son écriture et sa maintenance.

Le développement d'une application en couches permet à chacune de ces dernières d'être dédiée à une activité particulière. Cependant, il existe de nombreux traitements transversaux, généralement purement techniques qui sont utilisés dans les différentes couches.

Aucune méthode de programmation ne propose de solution pour une mise en oeuvre facile et pratique de ces traitements.

La programmation orientée objet ne propose que peut de solutions efficaces pour ce type de traitements : généralement cela passe par des appels à des objets dédiés dans les traitements.

La programmation orientée aspect propose d'externaliser ces traitements dans des entités nommées aspect.

Actuellement, les modules d'une application utilisent du code pour des traitements transversaux : l'AOP permet d'inverser cette dépendance. Aucun code de ces traitements n'est utilisé dans le module. Le code de ces traitements est regroupé dans des greffons qui sont insérés à la compilation ou à l'exécution.

68.2. Les concepts de l'AOP

Des points d'insertions (jointpoints) permettent d'utiliser ces aspects en précisant l'endroit où ils pourront être insérés. Le nombre de jointpoints n'est pas illimité : par exemple, il n'est pas possible d'insérer un greffon dans le code d'une méthode.

Les points de coupes sont des points d'insertions où les greffons seront insérés sans avoir à ajouter de code dans les traitements.

Un tisseur d'aspects permet d'insérer les aspects dans le code de l'application. Un aspect est un traitement particulier qui doit insérer à un ou plusieurs endroits définis par les points de coupe.

L'AOP utilise plusieurs concepts qui lui sont propres :

- point d'exécution (Joinpoint) : aussi appelé point de jonction, c'est un endroit particulier où il est possible d'invoquer un greffon dans le flot des traitements des composants (exemple : appel d'une méthode ou d'un constructeur, exécution d'une méthode, accès à un attribut, ...). Potentiellement n'importe quelle instruction peut être un point d'exécution mais l'AOP propose un sous-ensemble bien défini qui constitue les points d'exécution utilisables
- point de coupe (jointcut) : aussi appelé point d'action ou point de greffe ou point de recouvrement, c'est l'endroit où le greffon sera précisément invoqué lors du tissage. Un point de coupe est un sous-ensemble défini de points de jonctions

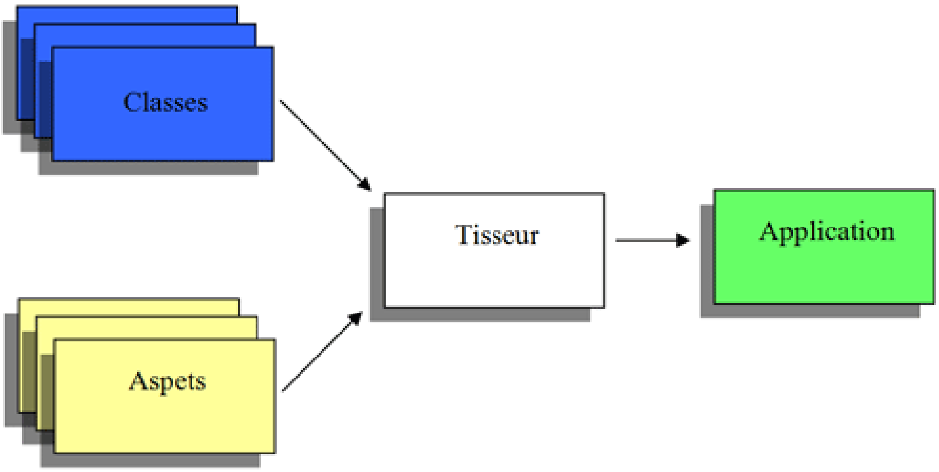
- greffon (advice) : aussi appelé perfectionnement, il contient des traitements techniques qui seront insérés à des jointcuts et exécutés
- aspect (aspect) : encapsule une fonctionnalité transverse et est composé d'un ou plusieurs points de coupe et greffons
- tissage (weaving) : action d'insertion des aspects
- tisseur (weaver) : outil qui réalise des tissages des aspects
- inter-type declarations : permet de déclarer de nouveau membres dans une classe

68.3. La mise en oeuvre de l'AOP

La mise en oeuvre de l'AOP consiste à :

- Définir les fonctionnalités
- Implémenter ces fonctionnalités sous la forme d'aspects
- Tisser les aspects pour les intégrer dans le code existant

La mise en oeuvre des aspects dans le code d'une application est réalisée lors d'une opération de tissage (weaving) par un tisseur (weaver)



C'est un tisseur d'aspects qui est responsable de la mise en oeuvre de l'AOP. L'AOP peut donc être mis en oeuvre avec n'importe quel langage qui possède un tisseur d'aspects.

Le tissage d'aspects permet d'insérer du code à des points d'exécution de l'application. Le tissage peut se faire de deux façons :

- statique : le tissage se fait avant l'exécution, généralement à la compilation, ce qui fait que le code exécuté mixe du code de l'application et des aspects. Ceci implique une recompilation en cas de modification ou d'ajout d'aspects mais ce type de tissage est généralement plus performant. (exemple : AspectJ)
- dynamique : l'avantage est qu'il est facilement possible de modifier les aspects (exemple : JAC, JBoss-AOP)

Le tissage peut être réalisé de trois manières :

- A la compilation : lors de la compilation si le compilateur supporte l'AOP ou post compilation avec un compilateur dédié. Cette compilation peut se faire au niveau du code source ou au niveau du bytecode. Dans ce dernier cas, il est possible de tisser n'importe quelle application sans avoir son code source. Il est aussi plus facile d'analyser le bytecode que son code source correspondant
- Au chargement : lors du chargement de la classe avec un classloader dédié
- A l'exécution : en utilisant des mécanismes reposant sur des proxys ou des interceptions

Un tisseur qui agit au niveau du bytecode peut utiliser des bibliothèques dédiées à sa manipulation telles que Byte Code Engineering Library (BCEL), ASM ou Javassist.

68.4. Les avantages et les inconvénients

La programmation orientée aspects possèdent des avantages mais aussi quelques inconvénients dont il faut tenir compte avant de la mettre en oeuvre.

Les avantages sont :

- Facilité de maintenance du code invoquant les traitements car il est épuré des fonctionnalités et des services techniques : le code est allégé et moins emmêlé puisque les fonctionnalités transverses sont regroupées dans les aspects
- Permet de contourner certaines faiblesses de la programmation orientée objet
- Particulièrement adapté pour les fonctionnalités techniques
- Permet une meilleure modularité du code et des applications ce qui augmente la réutilisation du code et la modularité des systèmes

Les inconvénients sont :

- La lecture du code contenant les traitements ne permet pas de connaitre les aspects qui seront exécutés (sans utiliser un outil).
- Nécessite un temps de prise en main
- Pas de normalisation : il existe plusieurs approches et implémentations, chaque implémentation proposant sa propre solution

68.5. Des exemples d'utilisation

L'AOP permet une implémentation modulaire de nombreuses fonctionnalités transverses.

Les utilisations possibles d'AOP sont nombreuses, notamment :

Le logging

Il peut être intéressant de définir des aspects qui vont réaliser des tâches de logging.

Le monitoring

Des aspects peuvent être définis pour obtenir des informations sur l'exécution de certaines méthodes pour par exemple :

- suivre un flot d'exécution
- compter le nombre d'invocations d'une méthode
- calculer des temps d'exécution d'une méthode
- ...

Ces informations peuvent ensuite être agrégées et diffusées pour consultation.

La gestion des exceptions

De nombreuses exceptions sont gérées de la même façon et de façon répétitive. Ces traitements peuvent donc être pour partie pris en charge par AOP, notamment la partie logging de ces traitements.

Le débogage

L'AOP peut permettre l'ajout de traces qu'il sera facile de désactiver.

Le profiling

La persistance

L'AOP peut être utilisée pour exécuter des traitements avant ou après des mises à jour.

La sécurité

L'AOP peut être utilisée pour gérer les habilitations d'accès à une méthode.

Les tests

La gestion des pré-conditions et post-conditions

L'AOP permet facilement la mise en place de pré et post conditions sur une ou plusieurs méthodes.

68.6. Des implémentations pour la plate-forme Java

Malheureusement, l'implémentation de l'AOP n'est pas standardisée : chaque implémentation fournie sa propre syntaxe pour mettre en oeuvre tout ou partie des fonctionnalités de l'AOP.

Il existe donc plusieurs implémentations de l'AOP pour Java qui implémentent tout ou partie des concepts de l'AOP de façon différente.

Les implémentations d'AOP suivent deux approches :

- Une approche généraliste : par exemple AspectJ
- Une approche spécifique : par exemple pour les besoins d'un framework tel que Spring avant sa version 2.0

Plusieurs implémentations d'AOP sont disponibles pour la plate-forme Java notamment :

Implémentation	URL
AspectJ	https://www.eclipse.org/aspectj
Spring AOP	https://spring.io/
JBoss-AOP	https://jbossaop.jboss.org/ Ce projet n'est plus maintenu
AspectWerkz	
Java Aspect Components	
HyperJ	

69. Terracotta

Chapitre 69

Niveau : 🟩🟩🟨🟨🟥🟥 Avancé

Le clustering est un terme qui désigne une solution technique dont le but est d'améliorer la disponibilité (fail-over) et/ou la montée en charge (load-balancing) d'une application. Concrètement cela se traduit par un groupement de serveurs qui sont vus comme un seul serveur logique. Cette redondance peut être utilisée pour :

- le fail-over : l'exécution des traitements est réalisée sur les serveurs actifs évitant ainsi de ne pas avoir de réponse si l'unique serveur est indisponible
- le load-balancing : la charge est répartie sur chacun des serveurs actifs

Toutes les solutions de clustering sont propriétaires puisqu'aucune des plates-formes Java (même Java EE) ne propose de spécifications relatives au clustering.

Pour permettre une meilleure montée en charge et un meilleur taux de disponibilité, les applications Java sont de plus en plus réparties sur différentes JVM.

Terracotta est une solution open source puissante de clustering au niveau de la JVM qui permet de facilement mettre en clusters une application dans plusieurs JVM. Ceci permet à une application d'être exécutée dans plusieurs JVM, Terracotta prenant en charge de façon transparente les interactions entre ces JVM pour faire en sorte que l'application s'exécute comme dans une seule JVM.

Terracotta fournit un environnement d'exécution qui facilite grandement la mise en clusters d'une application Java en proposant cette mise en clusters au niveau de la JVM plutôt qu'au niveau de l'application.

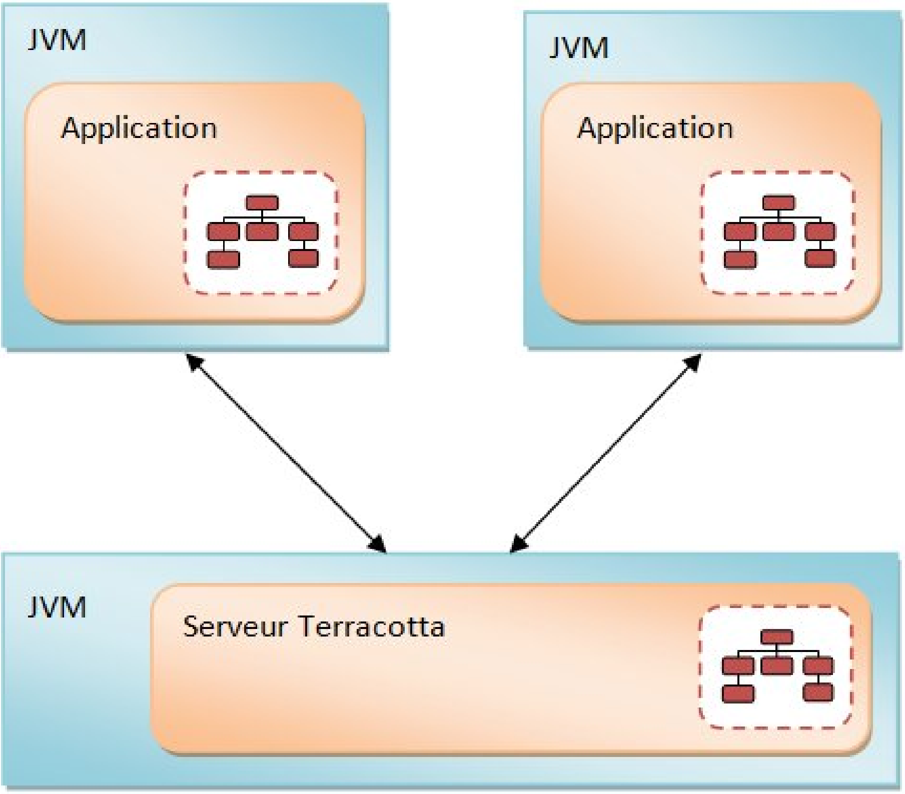
Le coeur de Terracotta est composé de deux parties :

- un serveur qui stocke les objets et coordonne les éléments du cluster
- une librairie qui permet la manipulation du bytecode pour ajouter des traitements gérant les interactions avec la JVM et le serveur

Terracotta agit directement sur les JVM pour maintenir l'état des objets gérés dans ces différentes JVM notamment en permettant entre autres :

- le partage d'objets
- le maintient des références
- la gestion des accès concurrents
- la synchronisation des threads (méthode wait(), notify(), notifyAll() de la classe Object)
- la gestion du ramasse-miettes
- ...

L'état des objets gérés est conservé dans le serveur (ou un ensemble de serveurs) qui est une application Java donc exécutée dans sa propre JVM.



Terracotta permet à une application exécutée dans plusieurs JVM d'accéder à des objets partagés dans une mémoire virtuelle.

Terracotta permet le partage de graphes d'objets entre plusieurs JVM sans avoir recours à une API dédiée : pour cela, Terracotta manipule dynamiquement le bytecode de l'application ce qui évite d'avoir à mettre en oeuvre une API particulière dans le code de l'application. Il n'est donc pas utile de mettre en oeuvre des API dédiées à l'échange de données comme des sockets, RMI, JMS, des services web, ...

L'accès aux données se fait de façon transparente : Terracotta s'occupe de stocker les données sur le serveur, de les copier localement lors de leur utilisation et de maintenir leur état sur le serveur et sur les clients.

Côté client, un classloader particulier instrumente les objets dans la JVM pour assurer le dialogue avec le serveur. Ainsi, si l'application est prévue pour fonctionner en multithread dans une JVM, elle pourra bénéficier de sa mise en cluster grâce à Terracotta sans altération de son code source.

Les échanges entre les JVM et le serveur sont particulièrement optimisés : ils ne reposent pas sur le transfert complet des objets sérialisés mais utilisent un mécanisme propre à Terracotta qui réduit les échanges au minimum.

Le fichier de configuration tc-config.xml permet de configurer la mise en oeuvre de Terracotta.

Terracotta est un projet open source pour lequel une version commerciale permet d'avoir du support et des fonctionnalités avancées.

Le site web officiel est à l'url : https://www.terracotta.org/

La version de Terracotta utilisée dans ce chapitre est la 3.2.

Ce chapitre contient plusieurs sections :

- La présentation de Terrocatta
- Les concepts utilisés
- La mise en oeuvre des fonctionnalités
- Les cas d'utilisation
- Quelques exemples de mise en oeuvre
- La console développeur
- Le fichier de configuration

- La fiabilisation du cluster
- Quelques recommandations

69.1. La présentation de Terrocatta

Terracotta permet à une application d'être déployée sur plusieurs JVM et de s'exécuter comme si elle l'était sur une seule. Terracotta permet d'appliquer le modèle de gestion de la mémoire d'une JVM dans plusieurs JVM qui peuvent être sur différentes machines.

En pratique, pour le développeur, Terracotta permet de voir un cluster de JVM comme une unique JVM : il est par exemple possible d'utiliser un singleton dans le cluster sans code supplémentaire.

Terracotta permet le partage d'objets et la coordination de threads entre différentes JVM.

La gestion des accès concurrents est assurée simplement en utilisant les mécanismes fournis par Java (moniteur avec le mot clé synchronized, utilisation de la bibliothèque java.util.concurrent, ...).

Terracotta prend aussi en charge des fonctionnalités Java de base au travers du cluster comme la gestion des accès concurrents (synchronized), la gestion de la mémoire (ramasse-miettes), la gestion des threads (wait(), notify(), ...). Terracotta prend en charge ces mécanismes de façon distribuée sous réserve qu'ils soient mis en oeuvre dans l'application par le développeur.

Les mécanismes de sérialisation, très coûteux, ne sont pas utilisés et les modifications ne sont pas broadcastées à tous les noeuds du cluster mais simplement à ceux qui en ont besoin : ceci permet de maximiser les performances des échanges réseaux réalisés par Terracotta.

Les objets partagés dans le cluster sont nommés Distributed Shared Objects (DSO). Ces objets sont vus comme des objets standards dans le heap de chaque client mais c'est Terracotta qui se charge de la gestion de ces objets de façon transparente grâce à l'instrumentation des objets gérés et de ceux qui les utilisent.

Terracotta met en oeuvre un ramasse-miettes distribué (DGC : Distributed Garbage Collector) qui s'assure avant de récupérer la mémoire d'un objet que celui-ci n'a plus de référence sur le serveur mais aussi sur chacun des clients.

69.1.1. Le mode de fonctionnement

Les solutions de mise en oeuvre de clusters d'applications Java repose généralement sur la sérialisation des objets modifiés pour les répliquer dans les différentes JVM. Ce type de solution est coûteux du fait même de l'utilisation de la sérialisation :

- coût de la sérialisation
- échange de l'intégralité de l'objet même si un seul champ a été modifié

Terracotta propose une solution alternative qui réduit ces coûts en n'échangeant que les modifications faites sur un objet. Il utilise son propre mécanisme natif pour l'échange des données : il n'est donc pas utile que les objets gérés soient sérialisables. L'état d'un objet est stocké sur un serveur et les clients ne reçoivent les modifications que lorsque l'objet est utilisé en local. Terracotta se charge alors de rafraichir l'objet à partir du serveur de façon transparente. Les échanges réseaux sont ainsi réduits.

Terracotta met en oeuvre le principe de Network Attached Memory (NAM). Ceci permet d'avoir une sorte de super JVM qui coordonne les JVM clientes du cluster. Comme certains objets et les threads sont partagés et synchronisés, les clients du cluster sont vus comme s'il ne s'agissait que d'une seule JVM.

La définition des objets gérés par Terracotta se fait dans un fichier de configuration au format XML : cette définition peut se faire de façon très fine et évite d'avoir à partager tous les objets de chaque JVM cliente.

Le bytecode de certaines classes de l'application est enrichi au chargement des classes : ceci permet une instrumentation des classes grâce à un classloader dédié.

La bibliothèque Terracotta analyse chaque classe chargée définie dans le fichier de configuration et injecte du bytecode au besoin pour permettre de dialoguer avec le serveur et d'effectuer les traitements que le serveur impose (mise à jour de l'état d'un ou plusieurs objets, pose ou libération de verrous sur les moniteurs, synchronisation des threads, ...).

L'instrumentation du bytecode permet donc entre autres de capturer les modifications faites sur un objet et de les envoyer au serveur. Ces modifications ne seront envoyées aux autres noeuds que lorsque ceux-ci auront besoin d'accéder à l'objet. Ainsi tous les noeuds ont accès aux objets mais ceux-ci ne sont mis à jour dans la JVM locale que si nécessaire.

Terracotta assure la cohérence d'une donnée gérée au travers du cluster.

Les objets gérés sont stockés dans le serveur Terracotta. Terracotta définit une racine pour un graphe d'objets qui sont partagés. Cette racine est identifiée par un champ nommé root dans le fichier de configuration.

Lorsqu'un objet racine est instancié, lui-même et toutes ses dépendances sont partagées par Terracotta. Toutes les données de ces objets sont stockées sur le serveur Terracotta.

Lors d'une modification sur l'état d'un objet géré, celle-ci est automatiquement répercutée au besoin dans les autres noeuds du cluster par le serveur Terracotta.

Terracotta utilise la notion de transaction pour gérer les accès concurrents aux objets gérés dans le cluster. Cette gestion repose sur les mêmes mécanismes que ceux utilisés dans une JVM mais de façon distribuée entre les noeuds du cluster : cette distribution se fait par l'intermédiaire du serveur.

Ainsi les méthodes synchronized, les blocs synchronized et les méthodes déclarées locked dans le fichier de configuration de Terracotta sont utilisés comme délimiteurs pour ces transactions.

A la fin de la transaction, les modifications sont envoyées au serveur pour les diffuser aux autres noeuds du cluster.

Un objet géré par le cluster vit dans la JVM d'un noeud de la même façon qu'un objet non géré sauf que son état est synchronisé par le serveur :

- lorsque des modifications sont faites en locales, l'objet est directement modifié et les modifications sont envoyées au serveur
- lorsque des modifications sont faites dans un noeud distant, celles-ci sont fournies à la JVM par le serveur et les modifications sont appliquées à l'objet local

Chaque noeud possède sa propre instance d'un objet géré par le cluster dont l'état est synchronisé grâce au serveur du cluster.

Aucune API liée à Terracotta n'est à mettre en oeuvre dans le code pour faire fonctionner le cluster. Cependant, certaines modifications dans le code sont parfois nécessaires notamment pour améliorer les performances : ces modifications ne sont pas directement liées à Terracotta mais à la bonne mise en oeuvre d'une programmation concurrente. La transparence de Terracotta est possible grâce à l'instrumentation du bytecode des classes à leur chargement dans la JVM.

Pour le développeur, la seule préoccupation est de s'assurer de la bonne gestion des accès concurrents sur les objets gérés.

69.1.2. La gestion des objets par le cluster

Terracotta utilise le concept de virtual heap qui permet de gérer dans le cluster de grandes quantités de données qui peuvent être largement supérieures à la mémoire disponible sur les clients. Ceci est possible car les objets gérés par le cluster ne sont fournis aux clients qu'au moment où ceux-ci en ont besoin.

L'instrumentation du bytecode côté client permet au moment de l'accès à un champ de demander sa valeur au serveur et de l'instancier dans la mémoire locale si celui-ci n'existe pas encore en local. Il est possible que ces références locales soient supprimées localement par le ramasse-miettes de la JVM. Si cette référence est de nouveau requise en local, elle sera de nouveau obtenue du serveur.

Terracotta garantit qu'une instance d'un objet géré par le cluster sera unique pour un classloader d'une JVM. Toutes les modifications sont répliquées sur les clients du cluster pour la même instance du classloader. Pour faciliter ce travail, Terracotta génère et attribue un identifiant unique qui lui est propre aux objets gérés dans le cluster.

Lors du chargement d'une classe, Terracotta utilise un classloader particulier pour enrichir le bytecode des classes à instrumenter définies dans le fichier de configuration. Les traitements ajoutés permettent à Terracotta de synchroniser l'état des objets gérés dans le cluster dans les différentes JVM qui le composent.

Les instances des objets gérés par le cluster sont créées grâce aux traitements ajoutés à la classe lors de son instrumentation :

- Une vérification de l'existence de l'instance sur le serveur est faite
- Si elle existe alors elle est créée localement à l'image de l'état de l'objet sur le serveur
- Si elle n'existe pas alors elle est créée localement et elle est répliquée sur le serveur

Les classes qui accèdent à des objets gérés par le cluster doivent aussi être instrumentées même si elles ne sont pas elle-même gérées par le cluster.

Les dépendances d'un objet géré sont également gérées par le cluster : ceci permet aux graphes d'objets des objets racines d'être gérés automatiquement par le cluster assurant ainsi de maintenir la cohérence de l'état de l'objet racine au travers du cluster.

Un objet peut être géré de deux façons dans le cluster Terracotta :

- Physically Managed : c'est la façon la plus courante dans laquelle les modifications faites dans chaque champ sont envoyées au serveur qui les reporte directement dans l'objet à chaque noeud qui possède une instance locale de l'objet.
- Logically Managed : pour certains objets particuliers (généralement proches de la JVM comme les classes qui utilisent un hashcode) les modifications sont propagées par l'enregistrement de l'invocation des méthodes avec leur paramètre et leur invocation distribuée sur les noeuds du cluster.

Certaines classes ne peuvent pas être gérées dans le cluster comme par exemple la classe Thread. Il en va de même pour les classes qui héritent de ces classes même si elles sont instrumentées. Si une classe non gérable est incluse dans le graphe d'objets gérés par le cluster alors une exception de type TCNonPortableObjectException est levée.

Il est possible de définir certains champs d'un objet gérés par le cluster comme transient. Ces champs ne seront alors pas gérés par le cluster.

Par défaut, les champs marqués avec le modificateur transient ne sont pas ignorés par Terracotta mais il est possible de demander qu'ils le soient grâce au fichier de configuration. N'importe quel champ peut aussi être ignoré grâce à une définition particulière dans le fichier de configuration.

Terracotta propose de définir des traitements exécutés à l'instanciation de la classe qui permettent une initialisation correcte des champs transient.

69.1.3. Les avantages de Terracotta

L'élégance de cette solution est de ne pas être intrusive dans le code de l'application mise en cluster :

- aucune API liée à Terracotta n'est à utiliser
- les objets gérés sont de simples POJO : aucune interface n'est à implémenter, pas même Serializable

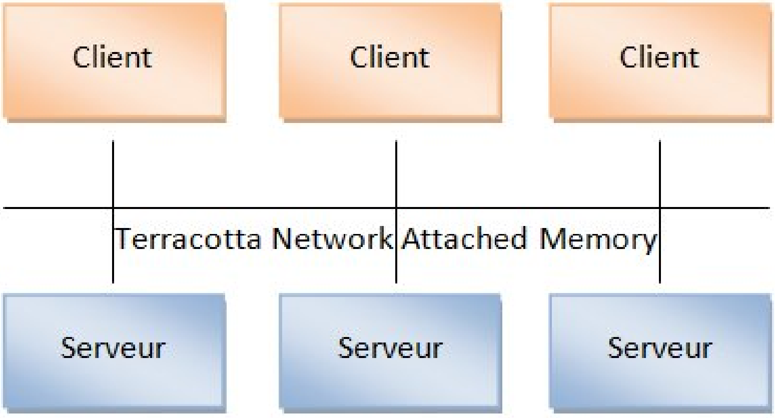
Le grand intérêt est de pouvoir utiliser une application de façon distribuée entre plusieurs instances dans des JVM dédiées sans avoir à modifier le code de l'application. Terracotta permet une mise en oeuvre d'un cluster de façon transparente pour le développeur : il n'y a pas de code intrusif à rajouter pour faire fonctionner l'application en cluster, sous réserve que le code soit déjà prévu pour fonctionner dans un mode multithread.

69.1.4. L'architecture d'un cluster Terracotta

Le projet Terracotta met en oeuvre le clustering au niveau de la JVM en utilisant des mécanismes de partage de mémoire au travers du réseau (NAM : Network Attached Memory) qui permettent de mettre en commun des instances d'objets entre plusieurs JVM.

L'architecture d'un cluster Terracotta est composée de deux types d'éléments : des noeuds clients et un ou plusieurs serveurs :

- les noeuds clients : les JVM qui exécutent l'application mise en cluster sont appelées des clients Terracotta ou des noeuds du cluster Terracotta
- un ou plusieurs serveurs Terracotta qui agissent comme un gestionnaire d'objets et de threads partagés entre tous les clients du cluster comme s'il ne s'agissait que d'une seule JVM. Le serveur Terracotta stocke les objets gérés, synchronise l'état de ces objets dans les différents noeuds et coordonne les threads entre les JVM.



Un serveur Terracotta assure plusieurs fonctionnalités :

- Gestion des objets gérés par le cluster et synchronisation de leur état avec les clients
- Gestion des verrous distribués et coordination des threads des différents clients
- Eventuellement persistance des objets sur disque pour assurer une persistance ou un stockage volatile (stockage temporaire sur disque des objets suite à un manque de place mémoire sur le serveur)

Le serveur gère les verrous posés ou demandés pour les différents threads des clients. Il coordonne aussi les notifications sur les threads concernés des clients lors de l'utilisation des méthodes notify() et notifyAll() appliquées à des objets gérés par le cluster.

Le serveur gère les objets dont il a la charge : ces objets sont instanciés sur un client qui en informe le serveur pour stockage et diffusion aux autres clients selon leurs besoins. Le serveur fournit aussi l'état des objets lors des sollicitations des clients pour qu'ils puissent créer une instance en local.

Il est possible de mettre en place plusieurs serveurs Terracotta en mode fail over. Un de ces serveurs est le serveur actif, les autres sont passifs (un des serveurs passifs prend le relais en devenant actif en cas d'arrêt du serveur actif). Ce sont les clients qui vont essayer de se connecter au serveur actif.

Le serveur Terracotta peut sauvegarder les données des objets partagés sur disque pour ne pas les perdre en cas d'arrêt du serveur.

69.2. Les concepts utilisés

Terracotta met en oeuvre plusieurs concepts :

- Un objet racine (root) : défini dans le fichier de configuration, il permet à Terracotta de savoir qu'un objet et ses dépendances sont gérés dans le cluster
- les transactions : elles permettent de garantir la cohérence des modifications faites sur des objets
- les verrous : ils permettent de gérer les accès concurrents faits sur les objets gérés par le cluster
- l'instrumentation des classes : au chargement de chaque classe précisée dans le fichier de configuration, une instrumentation de ces classes est réalisée pour permettre une communication avec le serveur.
- le ramasse-miettes distribué (Distributed Garbage Collector) : libère la mémoire des objets du serveur qui n'ont plus de référence ni sur le serveur ni sur les clients

69.2.1. Les racines (root)

Tous les objets d'un noeud d'un cluster Terracotta ne sont pas gérés par le cluster mais uniquement ceux qui sont déclarés comme étant la racine (root) d'un graphe d'objets gérés par le cluster.

Une racine permet d'identifier un objet comme devant être géré dans le cluster. La définition d'une racine se fait de manière déclarative dans le fichier de configuration. Une racine (root) constitue le sommet d'un graphe d'objets qui seront gérés et partagés par le cluster Terracotta. Le graphe est constitué de la racine et des objets qui sont accessibles depuis cette instance.

Une racine est un champ d'une classe dont le nom pleinement qualifié est défini dans le fichier de configuration.

Lorsqu'une racine est instanciée pour la première fois par un client du cluster, l'instance et ses éventuelles dépendances sont créées sur le serveur Terracotta.

Une fois qu'une instance d'une racine est créée sur le serveur, celle-ci ne peut plus être réaffectée : ceci n'empêche pas de modifier les autres instances du graphe d'objets de l'instance.

Une racine peut être une variable de type littéral :

- types primitifs : byte, short, int, long, char, float, double, boolean,
- leur Wrapper : Byte, Short, Integer, Long, Character, Float, Double, Boolean
- certains objets : String, Class, BigInteger, BigDecimal
- les énumérations

La valeur d'un objet racine de type littéral peut évoluer durant le cycle de vie du cluster. Les verrous sur un littéral sont posés par Terracotta sur leurs valeurs et non sur leur référence.

La classe qui encapsule la racine doit obligatoirement être instrumentée par Terracotta.

Des dépendances d'un objet géré par le cluster peuvent être configurées pour ne pas l'être par ce dernier.

69.2.2. Les transactions

Une transaction est un ensemble de modifications faites de façon atomique sur un ou plusieurs objets gérés par le cluster afin de garantir la cohérence de leurs états.

La portée d'une transaction est définie grâce à la pose et la libération d'un verrou. Lorsqu'un verrou est posé, Terracotta débute une transaction qui va enregistrer les modifications faites dans les objets gérés. Lorsque le verrou est libéré, la transaction est validée en reportant les modifications sur le serveur.

Chaque changement sur un objet géré par le cluster doit se faire dans une transaction : le thread qui veut faire la modification doit obligatoirement poser un verrou avant de changer l'état d'un objet géré par le cluster sinon une exception de type UnlockedSharedObjectException est levée.

Il faut donc obligatoirement :

- que les modifications d'un objet géré se fassent dans une transaction
- que l'objet soit géré par le cluster avant d'être modifié dans une transaction

Aucune transaction n'est définie pour une méthode :

- qui ne soit pas définie dans la partie lock du fichier de configuration
- qui n'est pas synchronized ou qui n'a pas l'attribut auto-synchronized à true

Important : il est impératif d'instrumenter toutes les classes qui peuvent modifier l'état d'un objet géré par le cluster. Dans le cas contraire, Terracotta ne verra pas les modifications et l'objet pourra se retrouver dans un état inconsistant.

69.2.3. Les verrous (locks)

Les modifications faites sur des objets gérés par le cluster doivent l'être dans le cadre d'une transaction. Une transaction enregistre les modifications sur les objets et les données primitives des objets. A la fin de la transaction, les modifications faites sur les objets sont envoyées au serveur.

Les verrous ont deux utilités essentielles dans le fonctionnement de Terracotta :

- Permettre de coordonner les accès aux sections critiques de code entre les threads des JVM
- Permettre de déterminer le début et la fin d'une transaction

Les verrous sont assez similaires au rôle du mot clé synchronized. D'ailleurs la définition d'un verrou peut se faire par le mot clé synchronized et/ou par définition dans le fichier de configuration.

Les verrous sont définis dans le fichier de configuration grâce à des expressions régulières qui définissent une méthode ou un ensemble de méthodes.

Il est impératif que les classes des méthodes sur lesquelles des verrous sont définis soient instrumentées.

Les verrous sont définis dans le fichier de configuration avec les éléments <autolock> parents de l'élément <locks>.

Ceci permet à Terracotta d'ajouter aux méthodes définies qui possèdent une synchronisation sur leur classe le code nécessaire à la pose de verrous au travers du cluster.

Exemple :

```
<locks>
  <autolock auto-synchronized="false">
    <method-expression>* fr.jmdoudoux.dej.MaClasse.*(..)</method-expression>
    <lock-level>write</lock-level>
  </autolock>
</locks>
```

La ou les méthodes qui seront verrouillées sont identifiées avec une expression dont la syntaxe est celle d'AspectWerkz.

Le type de verrou est précisé dans le fichier de configuration avec un tag <lock-level>.

Les verrous de Terracotta peuvent être posés selon plusieurs niveaux :

- Read : permet de s'assurer que les données partagées lues sont fraiches. Plusieurs threads peuvent poser un verrou mais uniquement pour de la lecture : aucune modification n'est possible sur les objets gérés par le cluster dans ce type de verrou. Aucun thread du cluster ne peut obtenir un verrou de type write si un verrou de type read est posé. Si un thread modifit un objet géré par le cluster dans une transaction avec un verrou de type read alors une exception de type UnlockedSharedObjectException est levée. Une exception est aussi levée si un thread pose un verrou de type write alors qu'un verrou de type read est déjà posé
- Synchronous write : un seul thread peut accéder à la portion de code. Le verrou sera conservé jusqu'à ce que toutes les modifications soient reportées sur le serveur et acquittées par celui-ci
- Write : agit comme les verrous Java. Un seul thread dans tout le cluster peut poser le verrou et accéder à la donnée
- Concurrent : verrou utilisé en interne par Terracotta

Le type et la portée d'un verrou peuvent avoir de gros impacts sur les performances de l'application exécutée dans le cluster.

Il n'est pas toujours possible de modifier le code existant pour ajouter des portions de code synchronized. Terracotta permet de les ajouter lors de l'instrumentation des classes grâce à deux fonctionnalités :

- auto-synchronized : permet d'ajouter le mot clé synchronized à la méthode concerné. Le verrou ainsi défini peut cependant être de type read ou write selon la définition réalisée dans le fichier de configuration
- named lock : permet de définir un verrou global qui va s'appliquer à toutes les instances de la classe au travers du cluster

Il peut être tentant de mettre auto-synchronized à toutes les méthodes mais l'effet sur les performances serait similaire à mettre le mot clé synchronized sur toutes les méthodes : les performances sont alors catastrophiques car il y a énormément de contention. Les verrous doivent donc être posés judicieusement et de préférence dans le code.

La pose de verrous est ajoutée par Terracotta sur les méthodes synchronized et leurs portions de code synchronized qui sont identifiés dans un autolock.

Une méthode définie dans un autolock qui n'a pas de synchronized n'a aucun effet. Dans ce cas, il est possible d'utiliser l'attribut auto-synchronized pour que Terracotta ajoute dynamiquement le mot clé synchronized à la méthode.

La mise en oeuvre des autolock de Terracotta se fait lors de l'instrumentation du bytecode (MONITORENTER et MONITOREXIT) en ajoutant du bytecode pour poser et lever un verrou au niveau du cluster.

Le verrou doit donc se faire sur un objet géré par le cluster car l'acquisition du verrou se fait en utilisant l'identifiant unique attribué par Terracotta. Pour une méthode synchronized qui n'est pas définit dans la partie autolock, le verrou est posé uniquement dans la machine virtuelle locale.

Terracotta propose, en plus des autolocks, des named locks qui permettent de définir des verrous identifiés par un nom. Il est préférable d'utiliser des autolocks plutôt que des named locks lorsque cela est possible car ces derniers peuvent avoir de gros impacts sur les performances de l'application.

Exemple :

```
<named-lock>
   <lock-name>MonVerrou</lock-name>
   <method-expression>* fr.jmdoudoux.dej.MaClasse.*(..)</method-expression>
   <lock-level>read</lock-level>
</named-lock>
```

69.2.4. L'instrumentation des classes

Les traitements relatifs au clustering sont ajoutés par instrumentation du bytecode de l'application : ce code est injecté au chargement de la classe par un classloader dédié. Il permet notamment de poser les verrous et échanger l'état des objets gérés par le cluster.

Les classes à instrumenter sont définies dans le fichier de configuration. Il est inutile d'instrumenter toutes les classes mais uniquement celles qui doivent être gérées par le cluster ou qui manipulent des objets gérés par le cluster. Il est d'autant plus important de limiter le nombre de classes à instrumenter que cette instrumentation est coûteuse au chargement de la classe et à l'exécution du fait des échanges réseau avec le serveur.

La définition des classes à instrumenter dans le fichier de configuration est indépendante de la définition des racines (roots) et des verrous (locks) : la définition de l'un ou l'autre n'implique pas une instrumentation explicite.

Terracotta doit instrumenter certaines classes par le classloader de boot. Ces classes ne peuvent pas être instrumentées dynamiquement puisque chargées avant Terracotta.

Celles-ci doivent donc être pré-instrumentées et placée dans un fichier jar particulier qui sera ajouté dans le classpath de boot. Ce fichier jar est nommé "boot jar" par Terracotta.

Terracotta propose un outil dédié pour générer ce boot jar. Une classe qui est chargée par le classloader de boot ne peut pas être gérée par le cluster : cette classe doit être instrumentée dans le boot jar.

La bibliothèque boot jar est précisée au lancement de chaque JVM cliente avec l'option :

-Xbootclasspath/p:chemin_du_fichier_jar_de_boot_terracotta.jar

69.2.5. L'invocation distribuée de méthodes

L'invocation distribuées de méthodes (DMI : Distributed Method Invocation) permet l'invocation d'une méthode d'un objet géré par le cluster dans tous les noeuds du cluster.

L'objet sur lequel la méthode sera invoquée doit être instrumenté et être géré par le cluster.

69.2.6. Le ramasse-miettes distribué

Un serveur Terracotta dispose d'un ramasse-miettes distribué (DGC Distributed Garbage Collector) dont le rôle est de purger les objets gérés par le cluster qui ne sont plus référencés ni dans le serveur ni dans aucun client.

Malgré son nom, ce n'est pas un processus distribué : il supprime uniquement des objets côté serveur (en mémoire et éventuellement dans le système de persistance des données du cluster) après s'être assuré qu'ils ne sont plus référencés par aucun objet gérés sur le serveur et qu'aucun client n'en possède encore une référence.

Un objet est géré par le cluster de son instanciation jusqu'à sa libération par le ramasse-miettes distribué (DGC Distributed Garbage Collector) de Terracotta.

Les objets racines ne sont pas traités par le DGC. Il y a plusieurs façons d'exécuter le DGC :

- périodiquement selon le paramétrage dans le fichier de configuration
- à la demande grâce à JMX
- à la demande par la console d'administration de Terracotta
- la demande en exécutant le script run-dgc

L'activité du DGC est journalisée dans le fichier de log du serveur. Cette activité peut aussi être surveillée en utilisant la console d'administration de Terracotta.

69.3. La mise en oeuvre des fonctionnalités

Cette section va fournir quelques informations pour mettre en oeuvre Terracotta.

69.3.1. L'installation

Terracotta peut être téléchargé gratuitement sur le site terracotta.org après un enregistrement obligatoire.

Il faut télécharger le fichier terracotta-3.2.0-installer.jar et l'exécuter :

```
C:\java>java -jar terracotta-3.2.0-installer.jar
```

Un assistant guide l'installation qui se fait par défaut dans le répertoire C:\Program Files\terracotta\terracotta-3.2.0

Pour faciliter son utilisation, il est possible d'ajouter le sous-répertoire bin, issu de l'installation, à la variable système PATH.

Le répertoire d'installation contient plusieurs sous-répertoires :

Nom	Contenu
bin	les scripts de commande
config-examples	les exemples de fichier de configuration
distributed-cache	ehcache
docs	la javadoc et un lien vers la doc en ligne
hibernate	
icons	
lib	les bibliothèques de Terracotta et de ses dépendances

modules	les modules permettant l'intégration de différentes technologies
quartz-1.7.0	l'api de scheduling Quartz
samples	des exemples
schema	le schéma du fichier de configuration et sa documentation
tools	des outils notamment le sessions-configurator
vendors	des outils tiers préconfigurés avec Terracotta

69.3.2. Les modules d'intégration

Pour faciliter la mise en oeuvre sans une connaissance approfondie de certains outils ou bibliothèques, Terracotta propose des modules d'intégration (TIM : Terracotta Integration Module).

Ces modules proposent une configuration out of the box :

- pour certains produits : Spring, Tomcat, GlassFish,
- pour certaines API : collections, ...
- pour des frameworks courants : EHCache, Hibernate, Spring, Struts, ...

La commande tim-get permet de gérer les modules installés avec Terracotta.

La syntaxe de cette commande est de la forme :

```
tim-get.bat [command] [arguments] {options}
```

Elle possède en premier argument une commande en fonction de l'action à réaliser :

Option	Rôle
list	obtenir une liste des TIM utilisables
install	installer un module
install-for	installer les modules précisés dans le fichier de configuration
info	afficher des informations détaillées sur un module
update	installer la dernière version d'un module
upgrade	mettre à jour le fichier de configuration et installer les dernières versions des modules
help	afficher une aide sur les commandes

L'option -h ou --help de chaque commande permet d'afficher une aide sur les options de la commande.

Cette commande requiert un accès à internet.

Certaines propriétés de la commande peuvent être modifiées dans le fichier tim-get.properties du sous-répertoire lib/resources d'installation de Terracotta. C'est notamment le cas si l'accès à internet passe par un proxy.

Les modules disposent d'un fichier terracotta.xml qui contient un fragment de la configuration qui sera fusionné avec le fichier de configuration de Terracotta.

69.3.3. Les scripts de commande

Terracotta propose un ensemble de scripts permettant de mettre en oeuvre le cluster.

Ces scripts sont livrés pour Unix (*.sh) ou windows (*.bat).

Script	Rôle
archive-tool	collecter des informations sur l'environnement pour les fournir au support de Terracotta
boot-jar-path	utiliser l'outil dso-env pour déterminer le chemin de la JVM. Cet outil ne devrait pas être utilisé directement
dev-console	lancer la console du développeur qui est un outil graphique pour surveiller et piloter le cluster

dso-env	permet de configurer un environnement client en valorisant une variable d'environnement système TC_JAVA_OPTS à partir des informations fournies par les variables JAVA_HOME, TC_INSTALL_DIR et TC_CONFIG_PATH
	Microsoft Windows
	`set TC_INSTALL_DIR="C:\Program Files\terracotta\terracotta-3.2.0"`
	`set TC_CONFIG_PATH="localhost:9510"`
	`call "%TC_INSTALL_DIR%\bin\dso-env.bat" -q`
	`set JAVA_OPTS=%TC_JAVA_OPTS% %JAVA_OPTS%`
	`call "%JAVA_HOME%\bin\java" %JAVA_OPTS% ...`
	UNIX/Linux (bash)
	`TC_INSTALL_DIR="/usr/local/terracotta-3.2.0"`
	`export  TC_INSTALL_DIR`
	`TC_CONFIG_PATH="localhost:9510"`
	`export TC_CONFIG_PATH`
	`. ${TC_INSTALL_DIR}/bin/dso-env.sh -q`
	`JAVA_OPTS="${JAVA_OPTS} ${TC_JAVA_OPTS}"`
	`${JAVA_HOME}/bin/java ${JAVA_OPTS} ...`

dso-java	permet de lancer un environnement d'exécution pour un client du cluster
make-boot-jar	permet de créer si nécessaire le boot jar qui devra être ajouté au classpath de boot. Le boot jar est spécifique à une plate-forme, une JVM et à la version de cette JVM
run-dgc	demander l'exécution du ramasse-miettes distribué
scan-boot-jar	permet de vérifier la cohérence entre le boot jar et le fichier de configuration
server-stat	(server status tool) permet de vérifier le statut courant du cluster
start-tc-server	démarrer un serveur du cluster
stop-tc-server	arrêter un serveur du cluster
tc-stats	(Terracotta cluster statistics recorder) permet de configurer et de gérer l'enregistrement des statistiques dans le cluster
tim-get	gérer les modules installés de Terracotta (TIM)
version	afficher la version de Terracotta

69.3.4. Les limitations

Tous les éléments du cluster (clients et serveurs) doivent utiliser la même JVM (vendeur et version) et la même version de Terracotta.

Attention : toutes les JVM ne sont pas supportées. Vu le mode de fonctionnement impliquant des interactions de bas niveau avec la JVM, il n'est pas surprenant que Terracotta ne fonctionne qu'avec certaines implémentations de la JVM ou même certaines versions de ces implémentations.

Terracotta peut gérer dans le cluster la plupart des objets Java mais il y a cependant des restrictions notamment des classes qui encapsulent une ressource externe (fichier, socket réseau, connexion vers des ressources, ...) ou spécifique à la JVM (Runtime, Thread, ...)

Un objet ne peut donc être géré par le cluster que s'il est portable. Toute tentative de faire gérer par le cluster un objet non portable lève une exception de type TCNonPortableObjectException.

69.4. Les cas d'utilisation

Terracotta propose du clustering d'objets Java au niveau de la JVM dans un but généraliste. Les cas d'utilisation sont donc nombreux parmi lesquels :

- partage de données entre les instances d'une même application
- avoir un singleton sur un cluster
- cache distribué de niveau 1 ou 2
- réplication de sessions http de conteneurs web de façon efficace
- partage de données entre plusieurs applications
- heap virtuel pour le partage de grandes quantités de données avec lazy loading
- répartition de charge grâce au partage d'une queue des traitements à effectuer par les différentes JVM
- utilisation comme système de messaging, sans avoir à mettre en oeuvre une solution de type MOM, en permettant l'échange d'objets directement entre JVM
- partage de très grandes quantités de données, supérieures à la capacité des clients : seules celles utilisées seront copiées localement
- ...

Certains de ces cas sont détaillés dans les sections suivantes.

69.4.1. La réplication de sessions HTTP

Chaque conteneur web propose sa propre implémentation concernant le stockage des sessions. Les données d'une session particulière sont retrouvées grâce à un identifiant unique nommé jsessionid.

La tendance est aux applications web stateless côté serveur mais cela implique que l'état soit stocké côté client (dans une application de type RIA) ou soit stocké dans la base de données mais ces deux solutions ne sont pas toujours possibles à mettre en oeuvre. Si le contexte est stocké dans une session, ce qui est le cas dans beaucoup d'applications web, il est nécessaire de répliquer ces sessions dans les différents noeuds du cluster ne serait-ce que pour garantir le fail-over.

Dans la réplication des sessions d'un cluster de conteneurs web, Terracotta peut être plus efficace que les mécanismes proposés par ces conteneurs qui utilisent généralement la sérialisation. Le mode de fonctionnement de Terracotta peut aussi permettre l'utilisation d'une affinité de session : tant que le serveur répond, il n'y pas de réplication des données sur les autres noeuds du cluster. Dès que le serveur tombe, un autre noeud répond et va obtenir les données de la session du serveur Terracotta. Les échanges réseaux sont donc limités.

De plus les objets stockés dans la session n'ont pas l'obligation d'implémenter l'interface Serializable.

Fréquemment pour optimiser la réplication des sessions fournies par les conteneurs, les objets de la session sont répartis dans différents attributs. Le mécanisme optimisé de la réplication des objets proposés par Terracotta évite d'avoir à sérialiser le graphe d'objets.

Pour faciliter la configuration, Terracotta propose des configurations par défaut pour conteneurs (Tomcat, Jetty, JBoss, GlassFish, ...) qui vont permettre de gérer les objets de stockages de la session dans le cluster Terracotta.

La configuration est très simple puisqu'il suffit

1. d'utiliser le module adéquat pour le conteneur utilisé
2. de fournir le nom de la webapp dont les sessions doivent être gérées
3. de déclarer l'instrumentation des types d'objets qui seront mis dans la session

Exemple :

```
<tc:tc-config xmlns:tc="http://www.terracotta.org/config">

  ...

  <application>
    <dso>
```

```
      <instrumented-classes>
        <include>
          <class>fr.jmdoudoux.dej.terracotta.MaClasse</class>
        </include>
      </instrumented-classes>
      <web-applications>
        <web-application>MaWebApp</web-application>
      </web-applications>
    </dso>
  </application>
</tc:tc-config>
```

Pour limiter les échanges de données entre les noeuds du cluster, il peut être utile d'utiliser l'affinité de session au niveau du load balancer pour permettre que cela soit toujours le même noeud qui réponde à un même client dans des conditions d'utilisation normale.

69.4.2. Un cache distribué

La mise en cluster d'un graphe d'objets se prête particulièrement bien à une utilisation sous la forme d'un cache distribué. Ainsi les données du cache sont répliquées dans les différents noeuds du cluster avec un chargement unique dans un noeud et la réplication dans les autres évitant ainsi un chargement à chaque noeud.

Terracotta peut distribuer un cache dont il propose un TIM par exemple EhCache ou une solution maison plus simple utilisant par exemple une collection de type Map gérant les accès concurrents comme ConcurrentHashMap.

Si le cache est fréquemment mis à jour, il faut être vigilent sur le positionnement des verrous pour ne pas dégrader les performances d'accès au cache ce qui annihilerait l'intérêt de sa mise en oeuvre.

69.4.3. La répartition de charge de traitements

Ce cas d'utilisation met en oeuvre le motif de conception master-worker. Un unique master crée des tâches et les empiles dans une collection partagée généralement de type Queue. Plusieurs workers se chargent du traitement des tâches placées dans la file.

Ceci permet de paralléliser les traitements de ces tâches dans les différentes JVM qui composent le cluster.

69.4.4. Le partage de données entre applications

Terracotta permet de partager des objets entre plusieurs instances d'une même application mais peut aussi permettre le partage d'objets entre différentes applications. Bien sûr, ces objets doivent avoir des caractéristiques communes notamment celles définies comme racine.

Ceci peut être très pratique pour par exemple partager des données entre une version standalone et une version web d'une application.

69.5. Quelques exemples de mise en oeuvre

Terracotta est fourni avec plusieurs applications d'exemples notamment une application de type chat, de partage de données, d'édition graphique, de type Queue, ...

Cette section va proposer quelques exemples simples de mise en oeuvre de Terracotta. Dans un contexte concret, les principes utilisés sont les mêmes mais sont généralement plus complexes notamment en ce qui concerne la définition du contenu du fichier de configuration.

69.5.1. Un exemple simple avec une application standalone

L'application utilisée dans cet exemple incrémente simplement un compteur static.

Exemple :

```
package fr.jmdoudoux.dej.terracotta;

public class MainTest {

  private static int compteur;

  public static void main(String[] args) {
      compteur++;
      System.out.println("Compteur = " + compteur);
  }

}
```

Remarque : cet exemple est basique car il ne gère pas les accès concurrents à la variable compteur. Il ne peut donc fonctionner correctement que si une seule instance de l'application est en cours d'exécution.

Chaque exécution de cette application affiche toujours la valeur 1 pour le compteur puisque sa valeur est initialisée à chaque lancement d'une JVM.

Résultat :

```
C:\eclipse34\workspace\TestTerracotta\bin>java -cp . fr.jmdoudoux.dej.terracot
ta.MainTest
Compteur = 1

C:\eclipse34\workspace\TestTerracotta\bin>java -cp . fr.jmdoudoux.dej.terracot
ta.MainTest
Compteur = 1
```

La mise en cluster de cette application va permettre de maintenir la valeur du compteur dans le cluster et permettre ainsi son incrémentation à chaque exécution.

Le fichier de configuration de Terracotta contient uniquement la définition d'une racine qui correspond au compteur.

Exemple :

```
<tc:tc-config xmlns:tc="http://www.terracotta.org/config">
  <application>
    <dso>
      <roots>
        <root>
          <field-name>fr.jmdoudoux.dej.terracotta.MainTest.compteur</field-name>
        </root>
      </roots>
    </dso>
  </application>
</tc:tc-config>
```

Il faut lancer le serveur Terracotta dans une boîte de commandes dédiée.

Résultat :

```
C:\>start-tc-server.bat
2010-04-25 21:29:58,555 INFO - Terracotta 3.2.0, as of 20100107-130122 (Revision
 14244 by cruise@su10mo5 from 3.2)
2010-04-25 21:30:00,040 INFO - Configuration loaded from the Java resource at '/
```

```
com/tc/config/schema/setup/default-config.xml', relative to class com.tc.config.
schema.setup.StandardXMLFileConfigurationCreator.
2010-04-25 21:30:01,009 INFO - Log file: 'C:\Documents and Settings\jmd\terracot
ta\server-logs\terracotta-server.log'.
2010-04-25 21:30:04,383 INFO - Available Max Runtime Memory: 504MB
2010-04-25 21:30:08,305 INFO - JMX Server started. Available at URL[service:jmx:
jmxmp://0.0.0.0:9520]
2010-04-25 21:30:10,399 INFO - Terracotta Server instance has started up as ACTI
VE node on 0.0.0.0:9510 successfully, and is now ready for work.
```

Il faut ensuite exécuter l'application dans une JVM configurée pour être utilisable dans le cluster. Pour une application standalone, Terracotta propose le script dso-java qui lance une JVM configurée pour devenir un client du cluster.

Par défaut, l'outil dso-java utilise le fichier tc-config.xml présent dans le répertoire courant. Il est possible de préciser un autre fichier en utilisant l'option -Dtc.config

Résultat :

```
C:\eclipse34\workspace\TestTerracotta\bin>dso-java -cp . fr.jmdoudoux.dej.terr
acotta.MainTest
 Starting Terracotta client...
2010-04-25 21:42:00,615 INFO - Terracotta 3.2.0, as of 20100107-130122 (Revision
 14244 by cruise@su10mo5 from 3.2)
2010-04-25 21:42:01,891 INFO - Configuration loaded from the file at 'C:\eclipse
34\workspace\TestTerracotta\bin\tc-config.xml'.
2010-04-25 21:42:02,343 INFO - Log file: 'C:\eclipse34\workspace\TestTerracotta\
bin\logs-172.16.0.50\terracotta-client.log'.
2010-04-25 21:42:13,204 INFO - Connection successfully established to server at
127.0.0.1:9510
Compteur = 1

C:\eclipse34\workspace\TestTerracotta\bin>dso-java -cp . fr.jmdoudoux.dej.terr
acotta.MainTest
 Starting Terracotta client...
2010-04-25 21:42:18,167 INFO - Terracotta 3.2.0, as of 20100107-130122 (Revision
 14244 by cruise@su10mo5 from 3.2)
2010-04-25 21:42:18,728 INFO - Configuration loaded from the file at 'C:\eclipse
34\workspace\TestTerracotta\bin\tc-config.xml'.
2010-04-25 21:42:18,852 INFO - Log file: 'C:\eclipse34\workspace\TestTerracotta\
bin\logs-172.16.0.50\terracotta-client.log'.
2010-04-25 21:42:20,906 INFO - Connection successfully established to server at
127.0.0.1:9510
Compteur = 2

C:\eclipse34\workspace\TestTerracotta\bin>dir
 Volume in drive C has no label.
 Volume Serial Number is 1B46-3C32

 Directory of C:\eclipse34\workspace\TestTerracotta\bin

25/04/2010  21:42    <DIR>          .
25/04/2010  21:42    <DIR>          ..
25/04/2010  21:22    <DIR>          com
25/04/2010  21:42    <DIR>          logs-127.0.0.1
25/04/2010  21:23               283 tc-config.xml

C:\eclipse34\workspace\TestTerracotta\bin>
```

L'état de la variable gérée par le cluster est automatiquement fourni au client une fois créé, ce qui permet une incrémentation sans réinitialisation par les clients.

69.5.2. Un second exemple avec une application standalone

Ce second exemple va utiliser une application qui stocke sa date/heure d'exécution dans une collection, attend 10 secondes et affiche le contenu de la collection.

Exemple :

```
package fr.jmdoudoux.dej.terracotta;

import java.util.*;

public class MainTest1 {

  private List<String> donnees = new ArrayList<String>();

  public void traiter() {
    synchronized (donnees) {
      donnees.add("Traitement du " + new Date());
      try {
        Thread.sleep(10000);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
      for (String donnee : donnees) {
        System.out.println(donnee);
      }
    }
  }

  public static void main(String[] args) {
    System.out.println("Demarrage de l'application");
    new MainTest1().traiter();
    System.out.println("Arret de l'application");
  }

}
```

A chaque exécution de cette application, la collection ne contient que l'occurrence du traitement courant puisque la collection est recrééе dans chaque JVM.

Il est possible avec Terracotta de maintenir l'état de la collection sur le serveur Terracotta et ainsi de partager l'objet entre différentes instances de JVM qui exécutent l'application en concomitance ou non.

Ce qui est intéressant avec Terracotta c'est que le code de l'application n'est pas modifié : aucune API de Terracotta n'est utilisée. Seul l'environnement d'exécution est différent.

Il faut tout d'abord créer un fichier de configuration pour Terracotta nommé tc-config.xml

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<tc:tc-config xmlns:tc="http://www.terracotta.org/config">
  <application>
    <dso>
      <roots>
        <root>
          <field-name>fr.jmdoudoux.dej.terracotta.MainTest1.donnees
          </field-name>
        </root>
      </roots>
      <locks>
        <autolock>
          <method-expression>
            * fr.jmdoudoux.dej.terracotta.MainTest1*.*(..)
          </method-expression>
            <lock-level>write</lock-level>
        </autolock>
      </locks>
      <instrumented-classes>
        <include>
          <class-expression>
            fr.jmdoudoux.dej.terracotta.MainTest1
          </class-expression>
        </include>
      </instrumented-classes>
    </dso>
  </application>
</tc:tc-config>
```

Ce fichier permet de préciser les classes qui sont prises en charge par Terracotta.

Il faut lancer le serveur Terracotta en exécutant la commande start-tc-server

Résultat :

```
C:\Documents and Settings\jmd>start-tc-server.bat
2010-02-12 21:00:08,599 INFO - Terracotta 3.2.0, as of 20100107-130122 (Revision
 14244 by cruise@su10mo5 from 3.2)
2010-02-12 21:00:09,193 INFO - Configuration loaded from the Java resource at '/
com/tc/config/schema/setup/default-config.xml', relative to class com.tc.config.
schema.setup.StandardXMLFileConfigurationCreator.
2010-02-12 21:00:09,490 INFO - Log file: 'C:\Documents and Settings\jmd\terracot
ta\server-logs\terracotta-server.log'.
2010-02-12 21:00:10,068 INFO - Available Max Runtime Memory: 504MB
2010-02-12 21:00:11,130 INFO - JMX Server started. Available at URL[service:jmx:
jmxmp://0.0.0.0:9520]
2010-02-12 21:00:12,287 INFO - Terracotta Server instance has started up as ACTI
VE node on 0.0.0.0:9510 successfully, and is now ready for work.
```

Il faut exécuter l'application en utilisant la commande dso-java

Résultat :

```
C:\eclipse34\workspace\TestTerracotta\bin>dso-java fr.jmdoudoux.dej.terracotta.MainTest1
 Starting Terracotta client...
2010-02-12 21:04:20,410 INFO - Terracotta 3.2.0, as of 20100107-130122 (Revision
 14244 by cruise@su10mo5 from 3.2)
2010-02-12 21:04:21,019 INFO - Configuration loaded from the file at 'C:\eclipse
34\workspace\TestTerracotta\bin\tc-config.xml'.
2010-02-12 21:04:21,144 INFO - Log file: 'C:\eclipse34\workspace\TestTerracotta\
bin\logs-127.0.0.1\terracotta-client.log'.
2010-02-12 21:04:24,097 INFO - Connection successfully established to server at 127.0.0.1:9510
Demarrage de l'application
Hello, World Fri Feb 12 21:04:24 CET 2010
Arret de l'application
```

Pour tirer avantages de Terracotta, il faut ouvrir deux boîtes de commandes et exécuter deux fois l'application en simultanée.

Si l'application est exécutée une troisième fois, les données des deux premières exécutions sont toujours présentes tant que le serveur Terracotta n'est pas arrêté.

69.5.3. Un exemple avec une application web

L'exemple de cette section va exécuter une application sur deux instances d'un serveur Tomcat version 6.0 mises en cluster.

Cette webapp contient une unique servlet stockant sa date/heure d'invocation dans un singleton qui encapsule une collection et affiche le contenu de la collection.

Le code du singleton est plutôt basique.

Exemple :

```
package fr.jmdoudoux.dej.terracotta.web;

```

```
import java.util.*;

public class MonCache {
  private static MonCache instance = new MonCache();
  private List<String> maListe;

  private MonCache() {
    maListe = new ArrayList<String>();
  }

  public static MonCache getInstance() {
    return instance;
  }

  public List<String> getDonnees() {
    return Collections.unmodifiableList(maListe);
  }

  public synchronized void ajouter(String occurence) {
    maListe.add(occurence);
  }

  public synchronized void effacer() {
    maListe.clear();
  }
}
```

Cette classe gère les accès concurrents notamment au niveau des méthodes qui opèrent des mises à jour dans la collection et elle renvoie une version immuable de lcelle-ci. Ces éléments sont importants pour la bonne mise en oeuvre de Terracotta.

Il faut obtenir et installer le TIM dédié à Tomcat 6.0 en utilisant la commande tim-get avec l'option install suivie du module à installer et de sa version.

Résultat :

```
C:\Users\Jean Michel>tim-get.bat list
Terracotta 3.2.0, as of 20100107-130122 (Revision 14244 by cruise@su10mo5 from 3
.2)

*** Terracotta Integration Modules for TC 3.2.0 ***

(+) ehcache-terracotta 1.8.0 [net.sf.ehcache]
(+) terracotta-hibernate-cache 1.1.0 [org.terracotta.hibernate]
(-) pojoizer 1.0.4
(-) tim-annotations 1.5.0
(-) tim-apache-collections-3.1 1.2.0
...
(-) tim-tomcat-5.0 2.1.0
(-) tim-tomcat-5.5 2.1.0
(-) tim-tomcat-6.0 2.1.0
(-) tim-vector 2.6.1
(-) tim-wan-collections 1.1.0
(-) tim-weblogic-10 2.1.0
(-) tim-weblogic-9 2.1.0
(-) tim-wicket-1.3 1.4.0
(+) quartz-terracotta 1.0.0 [org.terracotta.quartz]
(-) simulated-api 1.2.0 [org.terracotta]

(+) Installed  (-) Not installed  (!) Installed but newer version exists

C:\Users\Jean Michel>tim-get install tim-tomcat-6.0 2.1.0
Terracotta 3.2.0, as of 20100107-130122 (Revision 14244 by cruise@su10mo5 from 3
.2)

Installing tim-tomcat-6.0 2.1.0 and dependencies...
   INSTALLED: tim-tomcat-6.0 2.1.0 - Ok
   INSTALLED: tim-tomcat-5.5 2.1.0 - Ok
   INSTALLED: tim-tomcat-common 2.1.0 - Ok
   SKIPPED: tim-session-common 2.1.0 - Already installed
```

```
Done. (Make sure to update your tc-config.xml with the new/updated version if ne
cessary)

C:\Users\Jean Michel>
```

Il faut fournir en paramètre de la JVM les informations utiles à l'agent de Terracotta. Le plus simple est d'utiliser le script dso-env fourni par Terracotta :

- Définir la variable d'environnement TC_INSTALL_DIR
- Définir la variable d'environnement TC_CONFIG_PATH
- Invoquer le script dso-env contenu dans le répertoire bin de Terracotta
- Ajouter la variable TC_JAVA_OPTS à la variable JAVA_OPTS dans le script catalina de Tomcat lorsque celui-ci est utilisé pour démarrer Tomcat.

Exemple sous Linux/Unix :

Résultat :

```
...
export TC_INSTALL_DIR=<chemin_rep_Terracotta>
export TC_CONFIG_PATH=<chemin_fichier_tc-config.xml>
. $TC_INSTALL_DIR/bin/dso-env.sh -q
export JAVA_OPTS="$TC_JAVA_OPTS $JAVA_OPTS"
...
```

Exemple sous Windows :

Résultat :

```
...
set TC_INSTALL_DIR=< chemin_rep_Terracotta >
set TC_CONFIG_PATH=<path_to_local_tc-config.xml>
%TC_INSTALL_DIR%\bin\dso-env.bat -q
set JAVA_OPTS=%TC_JAVA_OPTS%;%JAVA_OPTS%
...
```

Il faut lancer le serveur Terracotta en lançant le script start-tc-server puis les serveurs du cluster.

69.5.4. Le partage de données entre deux applications

Cet exemple va permettre à deux applications d'incrémenter une variable commune partagée par Terracotta. Cette variable est encapsulée dans un objet propre à chacune des applications.

Exemple :

```
package fr.jmdoudoux.dej.terracotta.appli_a;

public class MonObjetA {

  private static int compteur;

  public static synchronized int incrementer() {
    compteur++;
    return compteur;
  }
}
```

La première application incrémente et affiche la variable deux fois.

Exemple :

```
package fr.jmdoudoux.dej.terracotta.appli_a;

public class AppliA {

  public static void main(String[] args) {

    System.out.println("Appli_A compteur="+MonObjetA.incrementer());

    System.out.println("Appli_A compteur="+MonObjetA.incrementer());
  }

}
```

Le fichier de configuration de Terracotta pour l'application ne dispose que d'une seule particularité : la racine du compteur partagé est définie avec un nom qui permettra d'y faire référence dans le fichier de configuration de l'autre application. Ce nom permettra à Terracotta d'identifier les deux objets comme étant le même : toutes les modifications dans l'un seront reportées dans l'autre et vice versa.

Exemple :

```
<tc:tc-config xmlns:tc="http://www.terracotta.org/config">
  <application>
    <dso>
      <roots>
        <root>
          <field-name>fr.jmdoudoux.dej.terracotta.appli_a.MonObjetA.compteur</field-name>
          <root-name>MonCompteur</root-name>
        </root>
      </roots>
      <locks>
        <autolock>
          <method-expression>
            * fr.jmdoudoux.dej.terracotta.appli_a.MonObjetA.*()
          </method-expression>
            <lock-level>write</lock-level>
        </autolock>
      </locks>
      <instrumented-classes>
        <include>
          <class-expression>
            fr.jmdoudoux.dej.terracotta.appli_a.*
          </class-expression>
        </include>
      </instrumented-classes>
    </dso>
  </application>
</tc:tc-config>
```

La seconde application encapsule aussi une variable statique de type int dans un objet dédié.

Exemple :

```
package fr.jmdoudoux.dej.terracotta.appli_b;

public class AppliB {
  public static void main(String[] args) {

    System.out.println("Appli_B compteur="+MonObjetB.incrementer());
  }
}
```

La seconde application va incrémenter le compteur et afficher la valeur.

Exemple :

```
package fr.jmdoudoux.dej.terracotta.appli_b;

public class MonObjetB {

  private static int compteur;

  public static synchronized int incrementer() {
    compteur++;
    return compteur;
  }
}
```

Le fichier de configuration est similaire à celui de la première application en utilisant ses propres objets.

Exemple :

```
<tc:tc-config xmlns:tc="http://www.terracotta.org/config">
  <application>
    <dso>
      <roots>
        <root>
          <field-name>fr.jmdoudoux.dej.terracotta.appli_b.MonObjetB.compteur</field-name>
          <root-name>MonCompteur</root-name>
        </root>
      </roots>
      <locks>
        <autolock>
          <method-expression>
            * fr.jmdoudoux.dej.terracotta.appli_b.MonObjetB.*()
          </method-expression>
            <lock-level>write</lock-level>
        </autolock>
      </locks>
      <instrumented-classes>
        <include>
          <class-expression>
            fr.jmdoudoux.dej.terracotta.appli_b.*
          </class-expression>
        </include>
      </instrumented-classes>
    </dso>
  </application>
</tc:tc-config>
```

Avant de lancer les applications, il faut lancer le serveur Terracotta.

Exemple :

```
C:\>start-tc-server.bat
2010-04-25 14:38:03,139 INFO - Terracotta 3.2.0, as of 20100107-130122 (Revision
 14244 by cruise@su10mo5 from 3.2)
2010-04-25 14:38:03,733 INFO - Configuration loaded from the Java resource at '/
com/tc/config/schema/setup/default-config.xml', relative to class com.tc.config.
schema.setup.StandardXMLFileConfigurationCreator.
2010-04-25 14:38:04,154 INFO - Log file: 'C:\Documents and Settings\jmd\terracot
ta\server-logs\terracotta-server.log'.
2010-04-25 14:38:06,701 INFO - Available Max Runtime Memory: 504MB
2010-04-25 14:38:09,279 INFO - JMX Server started. Available at URL[service:jmx:
jmxmp://0.0.0.0:9520]
2010-04-25 14:38:10,264 INFO - Terracotta Server instance has started up as ACTI
VE node on 0.0.0.0:9510 successfully, and is now ready for work.
```

Pour vérifier le partage de la donnée entre les deux applications, le test suivant est exécuté :

- Lancer l'application A
- Lancer l'application B
- Lancer l'application B

- Lancer l'application A

Résultat :

```
C:\eclipse34\workspace\TestTerracottaA\bin>dso-java -cp . fr.jmdoudoux.dej.ter
racotta.appli_a.AppliA
Starting Terracotta client...
2010-04-25 14:40:12,821 INFO - Terracotta 3.2.0, as of 20100107-130122 (Revision
14244 by cruise@su10mo5 from 3.2)
2010-04-25 14:40:13,414 INFO - Configuration loaded from the file at 'C:\eclipse
34\workspace\TestTerracottaA\bin\tc-config.xml'.
2010-04-25 14:40:13,586 INFO - Log file: 'C:\eclipse34\workspace\TestTerracottaA
\bin\logs-172.16.0.50\terracotta-client.log'.
2010-04-25 14:40:15,805 INFO - Connection successfully established to server at
172.16.0.50:9510
Appli_A compteur=1
Appli_A compteur=2
C:\eclipse34\workspace\TestTerracottaB\bin>dso-java -cp . fr.jmdoudoux.dej.ter
racotta.appli_b.AppliB
Starting Terracotta client...
2010-04-25 14:40:28,226 INFO - Terracotta 3.2.0, as of 20100107-130122 (Revision
14244 by cruise@su10mo5 from 3.2)
2010-04-25 14:40:28,820 INFO - Configuration loaded from the file at 'C:\eclipse
34\workspace\TestTerracottaB\bin\tc-config.xml'.
2010-04-25 14:40:28,960 INFO - Log file: 'C:\eclipse34\workspace\TestTerracottaB
\bin\logs-172.16.0.50\terracotta-client.log'.
2010-04-25 14:40:30,976 INFO - Connection successfully established to server at
172.16.0.50:9510
Appli_B compteur=3

C:\eclipse34\workspace\TestTerracottaB\bin>dso-java -cp . fr.jmdoudoux.dej.ter
racotta.appli_b.AppliB
Starting Terracotta client...
2010-04-25 14:40:35,757 INFO - Terracotta 3.2.0, as of 20100107-130122 (Revision
14244 by cruise@su10mo5 from 3.2)
2010-04-25 14:40:36,351 INFO - Configuration loaded from the file at 'C:\eclipse
34\workspace\TestTerracottaB\bin\tc-config.xml'.
2010-04-25 14:40:36,491 INFO - Log file: 'C:\eclipse34\workspace\TestTerracottaB
\bin\logs-172.16.0.50\terracotta-client.log'.
2010-04-25 14:40:38,476 INFO - Connection successfully established to server at
172.16.0.50:9510
Appli_B compteur=4
C:\eclipse34\workspace\TestTerracottaA\bin>dso-java -cp . fr.jmdoudoux.dej.ter
racotta.appli_a.AppliA
Starting Terracotta client...
2010-04-25 14:42:27,986 INFO - Terracotta 3.2.0, as of 20100107-130122 (Revision
14244 by cruise@su10mo5 from 3.2)
2010-04-25 14:42:28,580 INFO - Configuration loaded from the file at 'C:\eclipse
34\workspace\TestTerracottaA\bin\tc-config.xml'.
2010-04-25 14:42:28,752 INFO - Log file: 'C:\eclipse34\workspace\TestTerracottaA
\bin\logs-172.16.0.50\terracotta-client.log'.
2010-04-25 14:42:30,768 INFO - Connection successfully established to server at
172.16.0.50:9510
Appli_A compteur=5
Appli_A compteur=6
```

Le compteur est incrémenté correctement selon les invocations des applications.

69.6. La console développeur

Terracotta propose un outil particulièrement utile pour obtenir des informations sur le cluster et le surveiller : la console développeur (Developer Console).

Cet outil permet de voir de nombreuses informations sur l'état d'un cluster, notamment :

- Consulter l'état des objets gérés par le cluster

- Obtenir des métriques sur les activités du cluster (nombre d'objets instanciés, nombre de transactions réalisées, ...)

A partir de ces informations, il est possible de déterminer des actions pour améliorer les performances notamment sur les verrous.

69.7. Le fichier de configuration

Comme Terracotta ne propose aucune API, la configuration se fait dans un fichier de configuration qui est utilisé par le serveur et les noeuds du cluster.

Le fichier de configuration de Terracotta est un fichier XML qui permet de décrire les caractéristiques et le comportement du ou des serveurs Terracotta et des clients. Ce fichier explicite :

- la liste et les paramètres des serveurs
- la configuration des clients
- la liste des classes à instrumenter
- les racines des graphes d'objets gérées par le cluster
- les verrous

Au lancement d'un serveur, celui-ci recherche le fichier de configuration qui peut être :

- le fichier de configuration par défaut fourni avec Terracotta : celui-ci est utilisé si aucun fichier n'est précisé et si aucun fichier tc-config.xml n'est présent dans le répertoire courant au lancement du serveur
- un fichier local : soit précisé explicitement avec l'option -f de la commande start-tc-server soit le fichier tc-config.xml dans le répertoire courant où le serveur est lancé
- un fichier distant : l'url doit être précisée explicitement avec l'option -f de la commande start-tc-server

Au lancement d'un client, celui-ci recherche le fichier de configuration qui peut être :

- un fichier local : soit précisé explicitement avec l'option -f de la commande start-tc-server soit le fichier tc-config.xml dans le répertoire courant où le serveur est lancé
- un fichier distant : l'url doit être précisée explicitement avec l'option -f de la commande start-tc-server
- obtenu à partir d'un serveur : cette solution permet de garantir la synchronisation du fichier de configuration entre le serveur et le client puisque le fichier n'est déployé que sur le serveur

Le contenu du fichier de configuration chargé par un serveur ou un client est inséré dans le fichier de log. Il est aussi consultable grâce à la console du développeur.

Le fichier XML de configuration est composé de plusieurs éléments principaux facultatifs :

- system : permet de configurer des éléments de tous les composants du cluster
- servers : permet de configurer le ou les serveurs du cluster
- clients : permet de configurer le ou les noeuds du cluster
- application : permet de configurer les éléments qui seront gérés par le cluster

69.7.1. La configuration de la partie system

Cette partie est définie dans le tag <system> qui possède un tag fils <configuration-model> permettant de préciser un modèle de configuration : les valeurs possibles sont development ou production.

Par défaut, c'est le modèle development qui est utilisé. Il permet à chaque client d'avoir son propre fichier de configuration.

Avec le modèle production, les clients obtiennent le fichier de configuration d'un serveur, ce qui assure une cohérence entre les configurations des éléments du cluster (serveurs et clients). La JVM d'un noeud client doit avoir la propriété tc.config valorisée avec le nom du serveur et son port : -Dtc.config=host:port

69.7.2. La configuration de la partie serveur

Cette partie permet de configurer le ou les serveurs du cluster. Elle est définie dans le tag <servers>.

Si cette section est omise, alors le cluster contiendra un seul noeud avec les valeurs par défaut.

Chaque serveur du cluster est défini dans un tag fils <server>. Ce tag possède plusieurs attributs :

Attribut	Rôle
host	host du serveur Valeur par défaut : l'adresse IP de la machine locale
name	nom servant d'identifiant du serveur
bind	Valeur par défaut : 0.0.0.0

Si plusieurs serveurs sont définis dans le fichier de configuration, chaque serveur doit être démarré en précisant son nom.

Le tag fils <data> permet de préciser le répertoire qui va contenir les données du serveur stockées sur le système de fichiers.

Le tag fils <logs> permet de préciser le répertoire qui va contenir les logs du serveur.

Chaque serveur doit avoir un répertoire de logs distinct qui par défaut est le sous-répertoire terracotta/server-logs du répertoire de l'utilisateur.

Le tag fils <statistics> permet de préciser le répertoire qui va contenir les données statistiques du serveur.

Le tag fils <dso-port> permet de préciser le port d'écoute du serveur qui par défaut est le port 9510.

Le tag fils <jmx-port> permet de préciser le port JMX du serveur qui par défaut est le port 9520.

Le tag fils <l2-group-port> permet de préciser le port utilisé pour la communication entre les serveurs en mode actif-passif qui par défaut est le port 9530.

Le tag <dso> permet de configurer le service DSO du serveur.

Le tag fils <client-reconnect-window> permet de définir un temps de reconnexion pour un client exprimé en secondes. La valeur par défaut est 120.

Le tag fils <persistence> permet de préciser si les données gérées par le cluster sont persistantes ou non. Les valeurs possibles sont :

- temporary-swap-only : les données peuvent être temporairement écrites sur le système de fichiers mais elles ne survivent pas à un arrêt du cluster
- permanent-store : toutes les données gérées par le cluster sont stockées sur le système de fichiers ce qui permet de mettre en oeuvre un failover sur le cluster

Le tag fils <garbage-collection> permet de configurer le ramasse-miettes distribué. Il possède plusieurs tags fils :

Tag	Rôle
enabled	Activer ou non le DGC. La désactivation n'est utile que si aucune des instances gérées par le cluster n'est supprimée Par défaut : true
verbose	Activer ou non l'inclusion des informations relatives aux activités du DGC dans la log Par défaut : false

interval	Préciser le temps d'attente, en secondes, entre deux exécutions du DGC Par défaut : 3600

Le tag <ha> permet de configurer le mode de fonctionnement des serveurs du cluster : il faut donc qu'il y ait au moins 2 serveurs de configurés.

Le tag fils <mode> peut prendre deux valeurs :

- networked-active-passive :
- disk-based-active-passive :

Le tag fils <mirror-groups> permet de définir des groupes de serveurs.

69.7.3. La configuration de la partie cliente

Cette partie permet de configurer le ou les clients du cluster. Elle est définie dans le tag <clients>.

Exemple :

```
    <clients>
        <logs>/home/logs/terracotta/client-logs
        </logs>
        <statistics>/home/logs/terracotta/client-stats
        </statistics>
        <modules>
            <module name="tim-tomcat-6.0" version="2.1.0" />
            <module name="tim-vector" version="2.6.0" />
            <module name="tim-hashtable" version="2.6.0" />
        </modules>
    </clients>
```

Le tag fils <logs> permet de préciser où seront stockés les fichiers de logs.

Chaque client doit avoir un répertoire de logs distinct qui par défaut est le sous-répertoire terracotta/client-logs du répertoire de l'utilisateur.

Le tag fils <statistics> permet de préciser le répertoire qui va contenir les données statistiques du client.

Le tag fils <modules> permet de définir les TIM qui seront utilisés par les clients. Le tag <modules> contient un tag fils <module> pour chaque TIM utilisé.

Le tag <module> possède plusieurs attributs :

- name : contient le nom du module
- version : contient le numéro de version du module
- group-id : contient le nom du package du module (requis si le module n'est pas dans le répertoire par défaut des modules)

Par défaut, Terracotta recherche un jar correspondant au module à partir des informations fournies dans le sous-répertoire modules du répertoire d'installation de Terracotta. Il est possible de préciser des répertoires supplémentaires en utilisant le tag <repository> fils du tag <modules>. Si un module est stocké dans un de ces répertoires, l'attribut group-id du module doit être précisé.

69.7.4. La configuration de la partie applicative

Cette partie permet de configurer le ou les éléments à gérer par le cluster. Elle est définie dans le tag <dso> du tag fils <application>.

La configuration de ces éléments se fait au travers de trois entités principales :

- les racines (roots)
- les verrous (locks)
- les classes à instrumenter (instrumented classes)

69.7.4.1. La définition des racines

Les racines permettent de définir l'objet qui sera géré par le cluster en tant qu'objet père d'une hiérarchie d'objets composés de ses dépendances.

Les racines sont précisées dans un tag <roots> fils du tag application/dso.

Chaque objet racine est défini dans un tag <root>.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<tc:tc-config xmlns:tc="http://www.terracotta.org/config">
...
  <application>
    <dso>
      <roots>
        <root>
          <field-name>fr.jmdoudoux.dej.terracotta.MaClasse.monChamp
          </field-name>
        </root>
      </roots>
...
    </dso>
  </application>
</tc:tc-config>
```

Le tag fils <field-name> permet de préciser le champ d'une classe qui sera la racine du graphe d'objets gérés par le cluster. Ce graphe contient les dépendances de l'objet racine.

Le tag fils <root-name> permet de fournir un identifiant à la racine permettant d'y faire référence.

69.7.4.2. La définition des verrous

Les verrous sont précisés dans un tag <autolock> fils du tag application/dso/locks.

Il existe deux types de verrous :

- Autolock : permet de propager les instructions de synchronisation (méthodes ou blocs synchronized, wait(), notify(), ...) au travers du cluster
- Named lock : permet de définir un verrou nommé

Ces verrous définissent les méthodes qui posent des verrous afin de permettre à Terracotta de les propager à tous les noeuds du cluster, assurant ainsi une gestion sécurisée et distribuée des accès concurrents.

Cette propagation est assurée par des traitements ajoutés lors de l'instrumentation : les verrous indiquent à Terracotta qu'il faut instrumenter le code de la méthode pour prendre en charge des verrous distribués permettant ainsi de gérer les accès concurrents sur les objets.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<tc:tc-config xmlns:tc="http://www.terracotta.org/config">
...
```

```
  <application>
...
    <dso>
      <locks>
        <autolock>
          <method-expression>* fr.jmdoudoux.dej.terracotta.MaClasse.get*(..)
          </method-expression>
             <lock-level>read</lock-level>
        </autolock>
        <autolock>
          <method-expression>* fr.jmdoudoux.dej.terracotta.MaClasse.set*(..)
          </method-expression>
            <lock-level>write</lock-level>
        </autolock>
      </locks>
...
    </dso>
  </application>
</tc:tc-config>
```

Le tag fils <autolock> permet de définir un autolock.

L'attribut auto-synchronized permet de rajouter dynamiquement, au chargement de la classe, le modificateur synchronized sur la méthode. Ceci est pratique pour éviter de modifier le code source ou si le code source n'est pas modifiable. L'utilisation de cette option est cependant à utiliser avec parcimonie et pertinence.

Un autolock sur une méthode ou un ensemble de méthodes est défini grâce au tag fils <method-expression> en utilisant une syntaxe particulière empruntée à AspectWerkz.

Résultat :

```
* fr.jmdoudoux.dej.terracotta.MaClasse.get*(..)
* fr.jmdoudoux.dej.terracotta.MaClasse.set*(..)
void fr.jmdoudoux.dej.terracotta.MaClasse.setChamp(java.lang.String)
void *..*(..)
```

Important : il faut limiter le nombre de méthodes à instrumenter pour ne pas dégrader les performances au chargement des classes et au runtime.

Le tag fils <lock-level> permet de définir le niveau du verrou. Quatre niveaux sont définis : write, read, concurrent et synchronous-write. Le niveau par défaut est write.

69.7.4.3. La définition des classes à instrumenter

Les classes à instrumenter sont celles :

- Dont les instances sont gérées par le cluster (objets racines ou dépendances)
- Qui utilisent des instances gérées par le cluster
- Qui posent des verrous distribués

L'instrumentation des classes est définie dans le tag fils <instrumented-classes>.

Le tag fils <include> permet de définir les classes à instrumenter et le tag fils <exclude> permet de définir les classes à ne pas instrumenter.

Le tag <class-expression> fils du tag <include> permet de définir un ensemble de classes en utilisant une syntaxe particulière empruntée à AspectWerkz.

Résultat :

```
fr.jmdoudoux.dej.terracotta.MaClasse
fr.jmdoudoux.dej.terracotta.Ma*
fr.jmdoudoux.dej.terracotta.*
fr.jmdoudoux*.terracotta.*
fr.jmdoudoux.dej..
```

Le tag <exclude> utilise la même syntaxe.

69.7.4.4. La définition des champs qui ne doivent pas être gérés

Le tag fils <transient-fields> permet de définir des champs d'objets gérés par le cluster et à ne pas traiter.

C'est notamment pratique pour des champs qui ne sont pas marqués avec le modificateur transient dans le code source.

Chaque champ est défini grâce à un tag <field-name>.

69.7.4.5. La définition des méthodes dont l'invocation doit être distribuée

Le tag fils <distributed-methods> permet de définir des méthodes d'objets gérés par le cluster qui doivent être invoquées dans tous les noeuds du cluster dès qu'elles sont invoquées dans un des noeuds.

Chaque méthode est définie grâce à un tag <method-expression>.

L'attribut run-on-all-nodes permet de préciser dans quel noeud l'invocation sera faite : true (valeur par défaut) demande l'invocation dans tous les noeuds, false demande l'invocation dans tous les noeuds qui possèdent déjà une référence sur l'objet.

69.7.4.6. La définition des webapps dont la session doit être gérée

Le tag fils <web-applications> permet de définir une ou plusieurs applications de type web dont le contenu de la session sera géré par le cluster

Chaque application web doit être ajoutée avec un tag <web-application> qui contient son nom.

69.8. La fiabilisation du cluster

Terracotta propose plusieurs fonctionnalités qui peuvent être combinées selon les besoins pour assurer la fiabilité, la disponibilité et la montée en charge du cluster :

- La fiabilité grâce à la persistance des données
- La haute disponibilité avec la redondance des serveurs
- La montée en charge avec la définition de groupes de serveurs

A son lancement, une JVM cliente se connecte au serveur Terracotta actif pour interagir avec lui. Dans un cluster avec un serveur actif, il est possible de définir un ou plusieurs serveurs passifs. Si le serveur actif est arrêté, un des serveurs passifs est promu actif. Les clients du cluster tentent alors de se connecter au serveur actif parmi les serveurs configurés.

Si un serveur passif est configuré, l'arrêt du serveur actif est transparent pour les clients puisque ce serveur prend le relais en tant que serveur actif : les clients tentent de se connecter à ce serveur une fois qu'il est devenu actif. De nouveaux clients peuvent toujours rejoindre le cluster.

Si aucun serveur passif n'est configuré, le cluster est inutilisable tant que le serveur n'est pas redémarré. Les clients connectés avant l'arrêt attendent de pouvoir se reconnecter. Si le serveur était en mode persistant, les données sont restaurées et les clients peuvent se reconnecter. Si le serveur était en mode non persistant, les données sont perdues et les clients ne peuvent pas se reconnecter, ils doivent être arrêtés et démarrés pour se connecter au cluster.

Si un client ne peut se connecter à aucun serveur, il reste en attente tant qu'il n'arrive pas à se reconnecter à un serveur du cluster dans un temps configurable.

Un serveur peut être configuré pour être :

- Non persistant : les données des objets gérés par le cluster peuvent être écrites sur disque en cas de manque de mémoire sur le serveur. Les données sont perdues en cas d'arrêt et de redémarrage du cluster.
- Persistant : toutes les modifications sont écrites sur disque. Les données sont restaurées en cas d'arrêt et de redémarrage du cluster. Ce mode garantit que le redémarrage du cluster se fasse sans perte de données.

Si plusieurs serveurs sont utilisés en mode persistant, ils doivent être configurés pour écrire leurs données dans un système de fichiers partagés qui soit capable de gérer les accès concurrents en posant des verrous.

Il est possible d'utiliser et de configurer plusieurs serveurs Terracotta afin d'assurer une haute disponibilité du cluster Terracotta. Celle-ci repose sur plusieurs fonctionnalités selon le niveau de sûreté souhaité :

- Fail over avec un serveur actif et un ou plusieurs serveurs passifs
- Persistance des données du cluster sur disque
- La définition de groupes composés chacun d'un serveur actif et d'un ou plusieurs serveurs passifs ce qui permet de partitionner et répliquer des données (cette fonctionnalité n'est pas supportée dans la version open source)

La mise en oeuvre de ces fonctionnalités est effectuée dans le fichier de configuration.

L'état du cluster suite à l'arrêt du serveur actif (de façon volontaire ou non) dépend de deux facteurs :

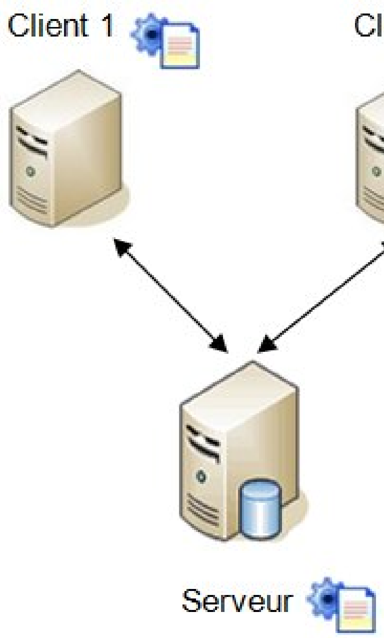
- Le serveur est dans le mode persistent ou non
- Une autre instance du serveur est configurée en mode passif

Serveur actif	Serveur passif	Etat du cluster
non persistent	non	Le cluster est inutilisable jusqu'au rédémarrage du serveur. A ce moment : • Les données des objets gérés sont perdues • Les clients déjà connecté ne peuvent pas se reconnecter et doivent être arrêtés et redémarrés
non persistent	oui	Le cluster continue de fonctionner car le serveur passif devient actif
persistent	non	Le cluster est inutilisable jusqu'au redémarrage du serveur. A ce moment : • Les données des objets gérés sont retrouvées • Les clients déjà connectés peuvent se reconnecter
persistent	oui	Le cluster continue de fonctionner car le serveur passif devient actif

69.8.1. La configuration minimale

Cette configuration ne propose ni fiabilité, ni haute disponibilité, ni montée en charge.

Cette configuration est pratique dans un environnement de développement mais n'est absolument pas recommandée en production.



La configuration minimale du cluster doit utiliser :

- un seul serveur Terracotta
- en mode non persistant

Exemple :

```
<?xml version="1.0" encoding="UTF-8" ?>
<tc:tc-config xmlns:tc="http://www.terracotta.org/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-5.xsd">
  <servers>
    <server name="noeud1">
...
          <dso>
              <persistence>
                  <mode>temporary-swap-only</mode>
              </persistence>
          </dso>
  </server>
...
  </servers>
  ...
</tc:tc-config>
```

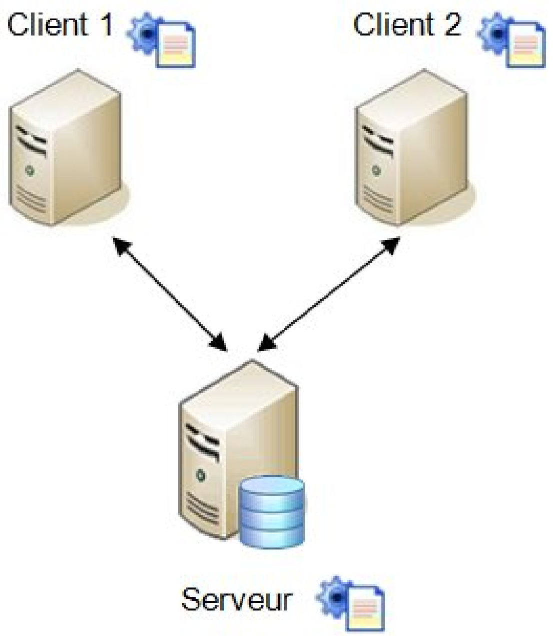
Pour démarrer le serveur, il suffit d'exécuter le script start-tc-server. L'option -f permet de préciser la localisation du fichier de configuration tc-config.xml.

69.8.2. La configuration pour la fiabilité

Cette configuration propose la fiabilité mais ne propose ni la haute disponibilité ni la montée en charge.

La configuration du cluster pour la fiabilité doit utiliser :

- Au moins un serveur Terracotta
- en mode persistant



Le mode persistant des données gérées par le cluster le rend plus fiable dans la mesure où les données sont restaurées si le cluster est redémarré.

Exemple :

```
<?xml version="1.0" encoding="UTF-8" ?>
<tc:tc-config xmlns:tc="http://www.terracotta.org/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-5.xsd">
  <servers>
    <server name="noeud1">
...
          <dso>
              <persistence>
                  <mode>permanent-store</mode>
              </persistence>
          </dso>
  </server>
...
  </servers>
  ...
</tc:tc-config>
```

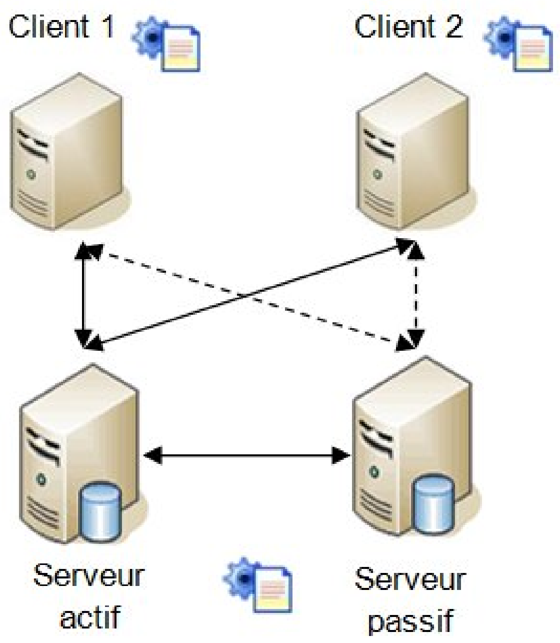
Cette configuration n'est généralement pas souhaitable telle quelle ni dans un environnement de développement (la persistance des données n'est pas nécessaire et est même parfois peu pratique car la purge des données est manuelle), ni dans un environnement de production (car elle n'assure pas la haute disponibilité).

69.8.3. La configuration pour une haute disponibilité

En production, il est nécessaire d'assurer la haute disponibilité du cluster notamment en mettant en place un failover sur le serveur. La configuration du cluster pour une haute disponibilité doit utiliser :

- au moins deux serveurs Terracotta
- le mode actif-passif

Dans cette configuration, un serveur actif communique avec les clients du cluster. Au moins un serveur passif attend de prendre le relai pour devenir le serveur actif en cas de défaillance. Dans cette configuration, un seul serveur peut être actif.



Terracotta synchronise automatiquement les serveurs pour permettre aux serveurs passifs d'être dans le même état que le serveur actif et ainsi de pouvoir prendre sa place en cas de défaillance.

Si les serveurs sont démarrés simultanément, l'un d'entre-eux est choisi pour être le serveur actif, les autres sont passifs. Lors de démarrage d'un serveur, si un serveur actif est en cours d'exécution, son état est synchronisé avec le serveur démarré.

En cas d'arrêt du serveur actif, un des serveurs passifs est promu actif.

Le mode actif-passif est configuré dans le tag <mode> fils du tag servers/ha avec la valeur networked-active-passive qui est celle recommandée car elle assure la synchronisation par le réseau.

Exemple :

```
<?xml version="1.0" encoding="UTF-8" ?>
<tc:tc-config xmlns:tc="http://www.terracotta.org/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-5.xsd">
  <servers>
    <server host="host2" name="noeud1">
...
    </server>
    <server host="host1" name="noeud2">
...
    </server>
...
     <ha>
        <mode>networked-active-passive</mode>
           <networked-active-passive>
               <election-time>5</election-time>
           </networked-active-passive>
       </ha>
  </servers>
  ...
</tc:tc-config>
```

Chaque serveur doit être défini dans un tag <server> avec obligatoirement pour chacun un attribut name unique.

Le tag <election-time> permet de préciser une durée en secondes pour déterminer le nouveau serveur actif. La valeur par défaut est 5 secondes.

Dans cette configuration, il est important que les répertoires de données précisés dans le tag <data> de chaque serveur soient différents et de préférence sur la machine locale pour améliorer les performances lorsque la persistance est requise.

Les répertoires des tags <logs> et <statistics> doivent aussi être différents.

Pour démarrer le serveur, il suffit d'exécuter le script start-tc-server avec l'option -n suivie du nom du serveur à démarrer. L'option -f permet de préciser la localisation du fichier de configuration tc-config.xml.

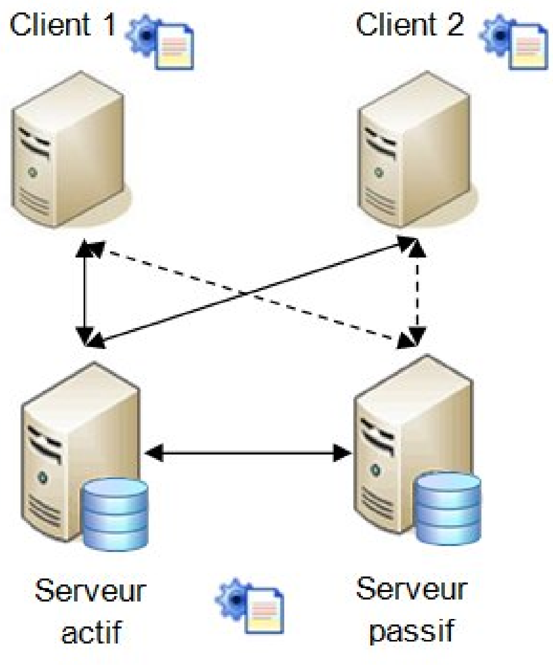
69.8.4. La configuration pour une haute disponibilité et la fiabilité

Cette configuration propose la fiabilité et la haute disponibilité. En production, il est nécessaire d'assurer la haute disponibilité et la fiabilité du cluster notamment en mettant en place un failover sur le serveur et la persistance des données.

La configuration du cluster pour une haute disponibilité et la fiabilité doit utiliser :

- au moins deux serveurs Terracotta
- en mode persistance des données
- le mode actif-passif

Cette configuration est une combinaison des configurations haute disponibilité et fiabilité.



Exemple :

```
<?xml version="1.0" encoding="UTF-8" ?>
<tc:tc-config xmlns:tc="http://www.terracotta.org/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-5.xsd">
  <servers>
    <server host="host2" name="noeud1">
...
          <data>repertoire_partage_des_donnees</data>
          <dso>
              <persistence>
                  <mode>permanent-store</mode>
              </persistence>
          </dso>
    </server>
    <server host="host1" name="noeud2">
...
          <data>repertoire_partage_des_donnees</data>
          <dso>
              <persistence>
                  <mode>permanent-store</mode>
              </persistence>
          </dso>
    </server>
...
     <ha>
        <mode>networked-active-passive</mode>
           <networked-active-passive>
               <election-time>5</election-time>
           </networked-active-passive>
       </ha>
  </servers>
  ...
</tc:tc-config>
```

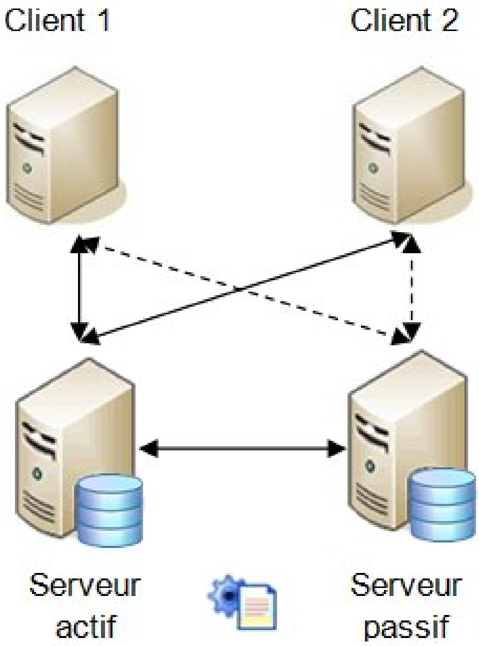
Il est possible d'utiliser plus de deux serveurs, l'un étant le serveur actif qui communique avec les clients, les autres serveurs étant passifs.

Les serveurs passifs peuvent être configurés en mode persistant ou non persistant mais il est préférable de les configurer dans le même mode que le serveur actif.

Si le serveur actif en mode persistant s'arrête et que le nouveau serveur actif est non persistant, il faudra obligatoirement purger les données présentes sur le disque du serveur stoppé avant de le relancer. Le serveur actif ne conservant plus les données, il y aurait un risque d'incohérence entre les données en mémoires et celles sur disque.

69.8.5. La configuration pour un environnement de production

En production, il est préférable de n'avoir qu'un seul fichier de configuration pour tous les éléments du cluster. Ceci n'est pas obligatoire mais cela facilite la gestion et la maintenance de placer le fichier de configuration à un seul endroit plutôt que de le répliquer pour chaque élément du cluster.



Le fichier de configuration est accessible aux différents serveurs en étant stocké dans un répertoire partagé. Lorsque le client se connecte à un serveur, il lui demande le contenu du fichier de configuration.

Dans le fichier de configuration, il faut que chaque serveur soit précisément identifié par son host et un nom unique.

Terracotta propose que les clients obtiennent le fichier de configuration du serveur auquel ils sont connectés. Pour cela, la variable d'environnement TC_CONFIG_PATH ne prend pas pour valeur le chemin du fichier de configuration mais contient le host suivi de deux-points et du port du serveur Il est possible de préciser les serveurs du cluster en les séparant par des virgules.

69.8.6. La configuration pour la montée en charge

Cette configuration propose la fiabilité, la haute disponibilité et la montée en charge.

La montée en charge est proposée au travers de la fonctionnalité mirror groups qui permet de définir des groupes de serveurs. Chacun de ces groupes possède un serveur actif et un ou plusieurs serveurs passifs.

Cette fonctionnalité n'est pas prise en charge dans la version open source de Terracotta mais elle est disponible dans la version Enterprise.

69.9. Quelques recommandations

Il est important pour le développeur de garder à l'esprit que pour que le cluster mis en oeuvre avec Terracotta fonctionne, l'application doit être codée en gérant correctement les accès concurrents.

Il est aussi nécessaire de prendre plusieurs facteurs en compte :

- s'assurer que tous les types d'objets gérés par Terracotta sont instrumentés dans le fichier de configuration
- correctement configurer les différents éléments dans le fichier de configuration
- ...

En cas de problème, Terracotta est assez verbeux dans l'exception levée et propose même une ou plusieurs pistes de corrections qui sont assez pertinentes.

Partie 10 : Le développement d'applications d'entreprises

Cette partie traite d'une utilisation de Java côté serveur avec la plate-forme basée sur le Java SE et orientée entreprise : Java EE (Java Entreprise Edition).

Cette partie regroupe plusieurs chapitres :

- J2EE / Java EE : introduit la plate-forme Java Entreprise Edition
- JavaMail : traite de l'API qui permet l'envoi et la réception d'e-mails
- JMS (Java Message Service) : indique comment utiliser cette API qui permet l'échange de données entre applications grâce à un système de messages
- Les EJB (Entreprise Java Bean) : propose une présentation de l'API et les spécifications pour des objets chargés de contenir les règles métiers
- Les EJB 3 : ce chapitre détaille la version 3 des EJB qui est une évolution majeure de cette technologie car elle met l'accent sur la facilité de développement sans sacrifier les fonctionnalités qui font la force des EJB.
- Les EJB 3.1 : ce chapitre détaille la version 3.1 des EJB utilisée par Java EE 6
- Les services web de type Soap : permettent l'appel de services distants en utilisant un protocole de communication et une structuration des données échangées avec XML de façon standardisée
- Les WebSockets : présente le procotole WebSocket
- L'API WebSocket : détaille l'utilisation de l'API Java API for WebSocket spécifiée par la JSR 356

70. J2EE / Java EE

Chapitre 70

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

J2EE est l'acronyme de Java 2 Entreprise Edition. Cette édition est dédiée à la réalisation d'applications pour entreprises. J2EE est basé sur J2SE (Java 2 Standard Edition) qui contient les API de base de Java. Depuis sa version 5, J2EE est renommée Java EE (Enterprise Edition).

Ce chapitre contient plusieurs sections :

- La présentation de J2EE : présente rapidement la plate-forme J2EE
- Les API de J2EE / Java EE : présente rapidement les différentes API qui composent J2EE
- L'environnement d'exécution des applications J2EE : présente les différents éléments qui composent l'environnement d'exécution des applications J2EE
- L'assemblage et le déploiement d'applications J2EE : décrit le mode d'assemblage et de déploiement des applications J2EE
- J2EE 1.4 SDK : installation et prise en main du J2EE 1.4 SDK
- La présentation de Java EE 5.0 : présente rapidement la version 5 de la plate-forme Java EE
- La présentation de Java EE 6
- La présentation de Java EE 7
- La présentation de Java EE 8/Jakarta EE 8

70.1. La présentation de J2EE

J2EE est une plate-forme fortement orientée serveur pour le développement et l'exécution d'applications distribuées. Elle est composée de deux parties essentielles :

- un ensemble de spécifications pour une infrastructure dans laquelle s'exécutent les composants écrits en Java : un tel environnement se nomme serveur d'applications.
- un ensemble d'API qui peuvent être obtenues et utilisées séparément. Pour être utilisées, certaines nécessitent une implémentation de la part d'un fournisseur tiers.

Sun propose une implémentation minimale des spécifications de J2EE : le J2EE SDK. Cette implémentation permet de développer des applications respectant les spécifications mais n'est pas prévue pour être utilisée dans un environnement de production. Ces spécifications doivent être respectées par les outils développés par des éditeurs tiers.

L'utilisation de J2EE pour développer et exécuter une application offre plusieurs avantages :

- une architecture d'applications basée sur les composants qui permet un découpage de l'application et donc une séparation des rôles lors du développement
- la possibilité de s'interfacer avec le système d'information existant grâce à de nombreuses API : JDBC, JNDI, JMS, JCA ...
- la possibilité de choisir les outils de développement et le ou les serveurs d'applications utilisés qu'ils soient commerciaux ou libres

J2EE permet une grande flexibilité dans le choix de l'architecture de l'application en combinant les différents composants. Ce choix dépend des besoins auxquels doit répondre l'application mais aussi des compétences dans les différentes API de J2EE. L'architecture d'une application se découpe idéalement en au moins trois tiers :

- la partie cliente : c'est la partie qui permet le dialogue avec l'utilisateur. Elle peut être composée d'une application standalone, d'une application web ou d'applets
- la partie métier : c'est la partie qui encapsule les traitements (dans des EJB ou des JavaBeans)
- la partie données : c'est la partie qui stocke les données

La suite de ce chapitre sera développée dans une version future de ce document

70.2. Les API de J2EE / Java EE

J2EE / Java EE regroupe un ensemble d'API pour le développement d'applications d'entreprise.

API	Rôle	version de l'API dans J2EE/Java EE						
		1.2	1.3	1.4	5	6	7	8
Entreprise Java Bean (EJB)	Composants serveurs contenant la logique métier	1.1	2.0	2.1	3.0	3.1	3.2	3.2
Remote Method Invocation (RMI) et RMI-IIOP	RMI permet l'utilisation d'objets Java distribués. RMI-IIOP est une extension de RMI pour une utilisation avec CORBA.	1.0						
Java Naming and Directory Interface (JNDI)	Accès aux services de nommage et aux annuaires d'entreprises	1.2	1.2	1.2.1				
Java Database Connectivity (JDBC)	Accès aux bases de données. J2EE intègre une extension de cette API	2.0	2.0	3.0				
Java Transaction API (JTA) Java Transaction Service (JTS)	Support des transactions	1.0	1.0	1.0	1.1	1.1	1.2	1.2
Java Message service (JMS)	Support de messages avec des MOM (Messages Oriented Middleware)	1.0	1.0	1.1	1.1	1.1	2.0	2.0
Servlets	Composants basés sur le concept C/S pour ajouter des fonctionnalités à un serveur. Pour le moment, principalement utilisé pour étendre un serveur web	2.2	2.3	2.4	2.5	3.0	3.1	4.0
Java Server Pages (JSP)		1.1	1.2	2.0	2.1	2.2	2.3	2.3
Java Server Faces (JSF)					1.2	2.0	2.2	2.3
Expression Language (EL)						2.2	3.0	3.0
Java Server Pages Standard Tag Libray (JSTL)					1.2	1.2	1.2	1.2
JavaMail	Envoi et réception d'e-mails	1.1	1.2	1.3	1.4	1.4	1.4	1.6
J2EE Connector Architecture (JCA)	Connecteurs pour accéder à des ressources du système d'information de l'entreprise telles que CICS, TUXEDO, SAP ...		1.0	1.5	1.5	1.6	1.6	1.7
Java API for XML Parsing (JAXP)	Analyse et exploitation de données au format XML		1.1	1.2				

Java Authentication and Authorization Service (JAAS)	Echange sécurisé de données		1.0					
JavaBeans Activation Framework	Utilisé par JavaMail : permet de déterminer le type MIME		1.0.2	1.0.2				
Java API for XML-based RPC (JAXP-RPC)				1.1	1.1	1.1	1.1	1.1
SOAP with Attachments API for Java (SAAJ)				1.2	1.3			
Java API for XML Registries (JAXR)				1.0	1.0	1.0	1.0	1.0
Java Management Extensions (JMX)				1.2				
Java Authorization Service Provider Contract for Containers (JACC)				1.0	1.1	1.4	1.4	1.5
Java API for XML-Based Web Services (JAX-WS)					2.0	2.2	2.2	2.2
Java Architecture for XML Binding (JAXB)					2.0	2.2		
Streaming API for XML (StAX)					1.0			
Java Persistence API (JPA)					1.0	2.0	2.1	2.2
Java API for RESTful Web Services (JAX-RS)						1.1	2.0	2.1
Web Services					1.2	1.3	1.3	1.3
Web Services Metadata for the Java Platform						2.1	2.1	2.1
Java APIs for XML Messaging (JAXM)						1.3		
Context and Dependency Injection (CDI)						1.0	1.1	1.1
Dependancy Injection (DI)						1.0	1.0	1.0
Bean Validation						1.0	1.1	2.0
Managed beans						1.0	1.0	1.0
Interceptors						1.1	1.2	1.2
Common Annotations						1.1	1.2	1.3
Java API for WebSocket							1.0	1.1
Java API for JSON Processing (JSON-P)							1.0	1.1
Concurrency Utilities for Java EE							1.0	1.0
Batch Applications for the Java Platform							1.0	1.0
Java API for JSON Binding (JSON-B)								1.0

Ces API peuvent être regroupées en trois grandes catégories :

- les composants : Servlet, JSP, EJB
- les services : JDBC, JTA/JTS, JNDI, JCA, JAAS
- la communication : RMI-IIOP, JMS, Java Mail

70.3. L'environnement d'exécution des applications J2EE

J2EE propose des spécifications pour une infrastructure dans laquelle s'exécutent les composants. Ces spécifications décrivent les rôles de chaque élément et précisent un ensemble d'interfaces pour permettre à chacun de ces éléments de communiquer.

Ceci permet de séparer les applications et l'environnement dans lequel elles s'exécutent. Les spécifications précisent à l'aide des API un certain nombre de fonctionnalités que doit implémenter l'environnement d'exécution. Ces fonctionnalités permettent aux développeurs de se concentrer sur la logique métier.

Pour exécuter ces composants de natures différentes, J2EE définit des conteneurs pour chacun d'eux. Il définit pour chaque composant des interfaces qui leur permettront de dialoguer avec les composants lors de leur exécution. Les conteneurs permettent aux applications d'accéder aux ressources et aux services en utilisant les API.

Les appels aux composants se font par des clients en passant par les conteneurs. Les clients n'accèdent pas directement aux composants mais sollicitent le conteneur pour les utiliser.

70.3.1. Les conteneurs

Les conteneurs assurent la gestion du cycle de vie des composants qui s'exécutent en eux. Les conteneurs fournissent des services qui peuvent être utilisés par les applications lors de leur exécution.

Il existe plusieurs conteneurs définis par J2EE:

- conteneur web : pour exécuter les servlets et les JSP
- conteneur d'EJB : pour exécuter les EJB
- conteneur client : pour exécuter des applications standalone sur les postes qui utilisent des composants J2EE

Les serveurs d'applications peuvent fournir un conteneur web uniquement (exemple : Tomcat) ou un conteneur d'EJB uniquement (exemple : JBoss, Jonas, ...) ou les deux (exemple : Websphere, Weblogic, ...).

Pour déployer une application dans un conteneur, il faut lui fournir deux éléments :

- l'application avec tous les composants (classes compilées, ressources ...) regroupés dans une archive ou module. Chaque conteneur possède son propre format d'archive.
- un fichier descripteur de déploiement contenu dans le module qui précise au conteneur des options pour exécuter l'application

Il existe trois types d'archives :

Archive / module	Contenu	Extension	Descripteur de déploiement
bibliothèque	Regroupe des classes	jar	
application client	Regroupe les ressources nécessaires à leur exécution (classes, bibliothèques, images, ...)	jar	application-client.jar
web	Regroupe les servlets et les JSP ainsi que les ressources nécessaires à leur exécution (classes, bibliothèques de balises, images, ...)	war	web.xml
EJB	Regroupe les EJB et leurs composants (classes)	jar	ejb-jar.xml

Une application est un regroupement d'un ou plusieurs modules dans un fichier EAR (Entreprise ARchive). L'application est décrite dans un fichier application.xml lui-même contenu dans le fichier EAR

70.3.2. Le conteneur web

Le conteneur web est une implémentation des spécifications servlets et par extension des spécifications des JSP. Ce type de conteneur est composé de deux éléments majeurs : un moteur de servlets (servlet engine) et un moteur de JSP (JSP engine).

Les conteneurs web peuvent généralement utiliser leur propre serveur web et être utilisés en tant que plug-in d'un serveur web dédié (Apache, IIS, ...).

L'implémentation de référence pour ce type de conteneur est le projet open source Tomcat du groupe Apache.

Les API spécifiquement mises en oeuvre dans un conteur web sont détaillées dans les chapitres «Les servlets»" et "JSP".

70.3.3. Le conteneur d'EJB

Les EJB sont détaillées dans le chapitre «Les EJB (Entreprise Java Bean)».

70.3.4. Les services proposés par la plate-forme J2EE

Une plate-forme d'exécution J2EE complète implémentée dans un serveur d'applications propose les services suivants :

- service de nommage (naming service)
- service de déploiement (deployment service)
- service de gestion des transactions (transaction service)
- service de sécurité (security service)

Ces services sont utilisés directement ou indirectement par les conteneurs mais aussi par les composants qui s'exécutent dans les conteneurs grâce à leurs API respectives.

70.4. L'assemblage et le déploiement d'applications J2EE

J2EE propose une spécification pour décrire le mode d'assemblage et de déploiement d'une application J2EE.

Une application J2EE peut regrouper différents modules : modules web, modules EJB ... Chacun de ces modules possède son propre mode de packaging. J2EE propose de regrouper ces différents modules dans un module unique sous la forme d'un fichier EAR (Entreprise ARchive).

Le format de cette archive est très semblable à celui des autres archives :

- un contenu : les différents modules qui composent l'application (module web, EJB, fichier RAR, ...)
- un fichier descripteur de déploiement

Les serveurs d'applications extraient chaque module du fichier EAR et les déploient séparément un par un.

70.4.1. Le contenu et l'organisation d'un fichier EAR

Le fichier EAR est composé au minimum :

- d'un ou plusieurs modules

- d'un répertoire META-INF contenant un fichier descripteur de déploiement nommé application.xml

Les modules ne doivent pas obligatoirement être insérés à la racine du fichier EAR : ils peuvent être mis dans un des sous-répertoires pour organiser le contenu de l'application. Il est par exemple pratique de créer un répertoire lib qui contient les fichiers .jar des bibliothèques communes aux différents modules.

70.4.2. La création d'un fichier EAR

Pour créer un fichier EAR, il est possible d'utiliser un outil graphique fourni par le vendeur du serveur d'applications ou de créer le fichier manuellement en suivant les étapes indiquées ci-dessous :

1. créer l'arborescence des répertoires qui vont contenir les modules
2. insérer dans cette arborescence les différents modules à inclure dans le fichier EAR
3. créer le répertoire META-INF (en respectant la casse)
4. créer le fichier application.xml dans ce répertoire
5. utiliser l'outil jar pour créer le fichier EAR en précisant les options cvf, le nom du fichier ear avec son extension et les différents éléments qui composent le fichier (modules, répertoire dont le répertoire META-INF).

70.4.3. Les limitations des fichiers EAR

Actuellement les fichiers EAR ne servent qu'à regrouper différents modules pour former une seule entité. Rien n'est actuellement prévu pour prendre en compte la configuration des objets permettant l'accès aux ressources par l'application telle qu'une base de données (JDBC pour DataSource, pool de connexions ...), un système de messages (JMS), etc ...

Pour pallier une partie de ces limites, les serveurs d'applications commerciaux proposent souvent des mécanismes propriétaires supplémentaires en attendant une évolution des spécifications.

70.5. J2EE 1.4 SDK

La version 1.4 de J2EE a été diffusée en novembre 2003.

La grande nouveauté de la version 1.4 est le support des services web. Deux nouvelles API ont été ajoutées pour normaliser le déploiement (J2EE deployment API 1.1) et la gestion des applications (J2EE management API 1.0 qui utilise JMX). Une nouvelle API permet de standardiser l'authentification (Java ACC : Java Authorization Contract for Container). Plusieurs API déjà présentes dans les précédentes versions de J2EE ont été mises à jour (EJB, JSP, Servlet, ...) :

- J2EE Connector Architecture 1.5
- Enterprise JavaBeans (EJB) 2.1
- JavaServer Pages (JSP) 2.0
- Java Servlet 2.4
- JavaMail 1.3
- Java Message Service 1.1
- Java API for XML parsing (JAXP) 1.2
- Java API for XML-based RPC (JAX-RPC) 1.1
- SOAP with Attachments API for Java (SAAJ) 1.2
- Java API for XML Registries (JAXR) 1.0
- Java Management Extensions (JMX) 1.2
- Java Authorization Service Provider Contract for Containers (JACC) 1.0

Le J2EE SDK 1.4 qui est l'implémentation de référence inclus le J2SE SDK 1.4.2 et J2EE 1.4 application server.

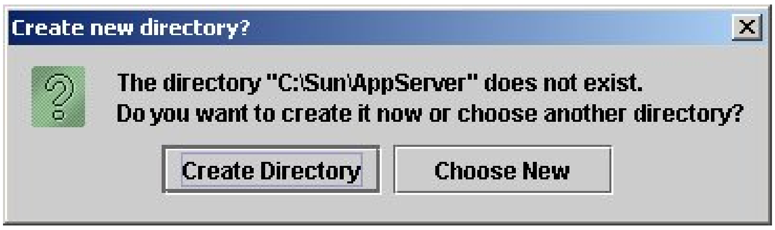
70.5.1. L'installation de l'implémentation de référence sous Windows

Le J2EE SDK 1.4 peut être installé sur les systèmes Microsoft suivants : Windows 2000 pro avec un service pack SP2, Windows XP Pro avec un service pack SP1 et Windows Server 2003.

Il existe plusieurs packages d'installation : celui utilisé ci-dessous ne contient que le serveur d'applications puisque le J2SE 1.4.2 était déjà présent sur la machine.

Lancer le programme j2eesdk-1_4-dr-windows-eval-app.exe. L'application extrait les fichiers, lance le J2RE et exécute un assistant qui va guider l'installation :

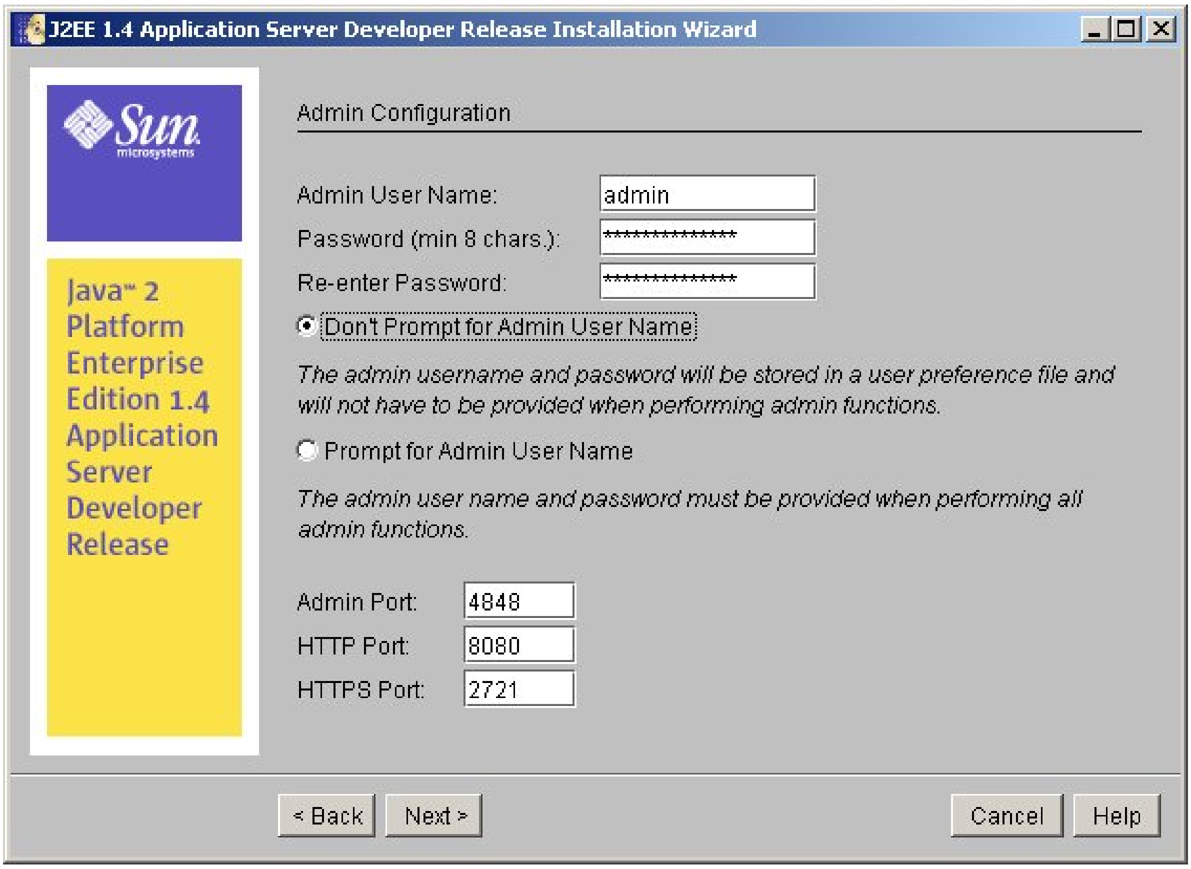
- la première page est la page d'accueil (welcome) : cliquez sur le bouton « Next »
- La page suivante permet de lire et d'accepter la licence d'utilisation (software licence agreement) : lire la licence et si vous l'acceptez, cliquez sur le bouton « Yes » puis sur le bouton « Next »
- la page suivante permet de sélectionner le répertoire d'installation du produit (Select Installation Directory) : sélectionnez le répertoire d'installation et cliquez sur « Next »



The directory "C:\Sun\AppServer" does not exist.
Do you want to create it now or choose another directory?

Cliquez sur « Create directory »

- la page suivante permet de préciser l'emplacement du J2SDK nécessaire au produit (Java 2 SDK Required) : sélectionnez l'emplacement du J2SE SDK (s'il est présent sur la machine, son chemin est proposé par défaut), puis cliquez sur le bouton « Next »
- la page suivante permet de préciser les informations nécessaires à la configuration du serveur


J2EE 1.4 Application Server Developer Release Installation Wizard
Sun microsystems
Java™ 2 Platform Enterprise Edition 1.4 Application Server Developer Release
< Back
Next >
Cancel
Help

Admin Configuration

Admin User Name: admin
Password (min 8 chars.): ***************
Re-enter Password: ***************

● Don't Prompt for Admin User Name

The admin username and password will be stored in a user preference file and will not have to be provided when performing admin functions.

○ Prompt for Admin User Name

The admin user name and password must be provided when performing all admin functions.

Admin Port: 4848
HTTP Port: 8080
HTTPS Port: 2721

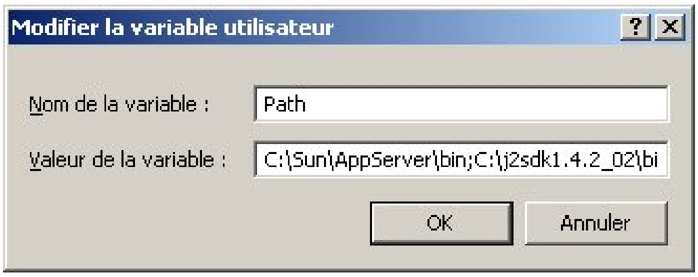
Il faut saisir des paramètres de configuration : saisir le mot de passe de l'administrateur et sélectionner l'option pour l'authentification ou non lors d'actions d'administration.

Il est aussi possible de définir les ports pour le module d'administration et pour les serveurs web HTTP et HTTPS.

Une fois les informations saisies, cliquez sur le bouton « Next ».

- La page suivante permet de sélectionner le type d'installation à réaliser (Installation Options) : pour une première installation, il suffit de conserver l'option par défaut proposée puis cliquez sur le bouton « Next »
- La page suivante synthétise les différentes informations avant l'installation (Ready to Install) : cliquez sur « Install Now »
- La dernière page indique que l'installation s'est bien terminée et donne quelques informations sommaires sur les différents outils

Il est utile de rajouter le répertoire bin du répertoire d'installation de J2EE SDK 1.4 à la variable PATH du système d'exploitation.


Modifier la variable utilisateur

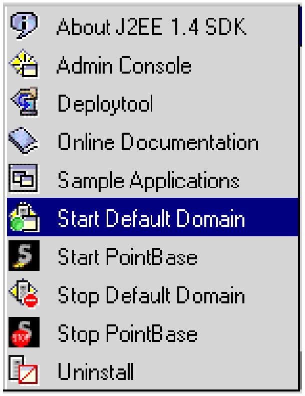
Nom de la variable : Path

Valeur de la variable : C:\Sun\AppServer\bin;C:\j2sdk1.4.2_02\bi

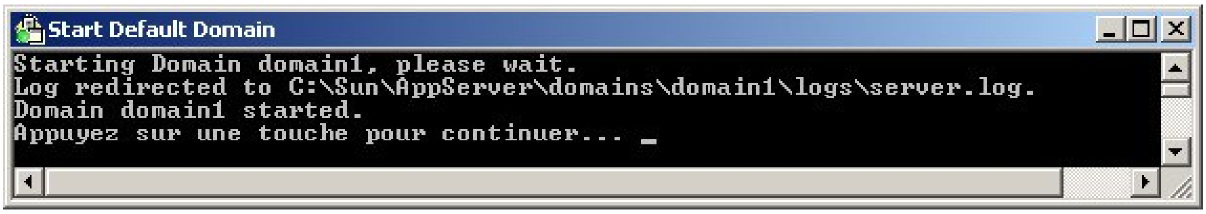
70.5.2. Le démarrage et l'arrêt du serveur

Un domaine permet de regrouper des applications qui s'exécutent avec une configuration particulière sur une instance donnée du serveur. Lors de l'installation un domaine par défaut est créé : domain1

Le programme d'installation a créé plusieurs entrées dans le menu « Démarrer/Programmes/Sun Microsystems/J2EE 1.4 SDK/ »



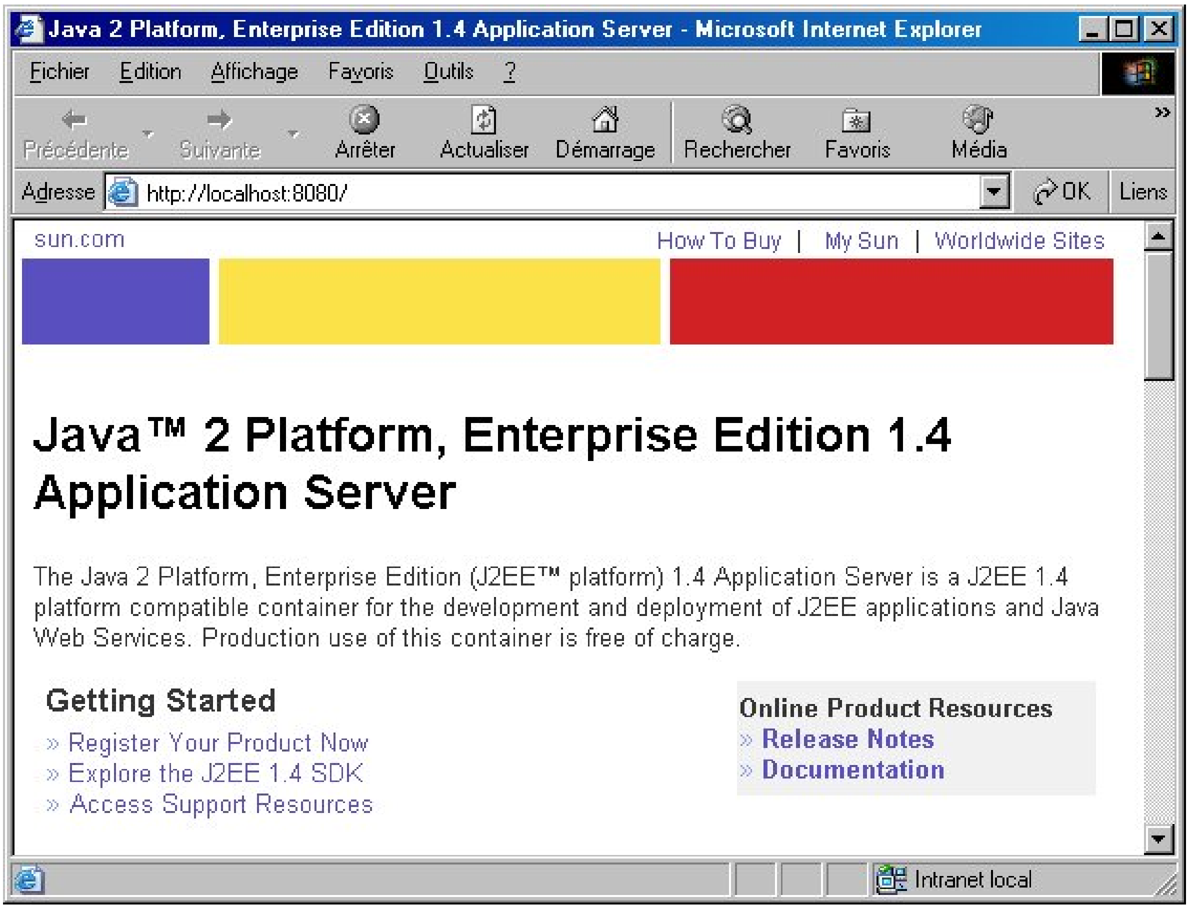
L'option « Start default domain » permet de démarrer le domaine domain1.


Start Default Domain

```
Starting Domain domain1, please wait.
Log redirected to C:\Sun\AppServer\domains\domain1\logs\server.log.
Domain domain1 started.
Appuyez sur une touche pour continuer... _
```

Il suffit d'appuyer sur une touche pour fermer la fenêtre.

Pour vérifier la bonne exécution du serveur, il suffit d'appeler l'URL http://localhost:nnnn/ dans un navigateur où nnnn représente le port http précisé dans les paramètres lors de l'installation.


Java 2 Platform, Enterprise Edition 1.4 Application Server - Microsoft Internet Explorer
Fichier Edition Affichage Favoris Outils ?
Précédente Suivante Arrêter Actualiser Démarrage Rechercher Favoris Média
Adresse http://localhost:8080/
OK Liens
sun.com
How To Buy | My Sun | Worldwide Sites
Intranet local

Java™ 2 Platform, Enterprise Edition 1.4 Application Server

The Java 2 Platform, Enterprise Edition (J2EE™ platform) 1.4 Application Server is a J2EE 1.4 platform compatible container for the development and deployment of J2EE applications and Java Web Services. Production use of this container is free of charge.

Getting Started

» Register Your Product Now
» Explore the J2EE 1.4 SDK
» Access Support Resources

Online Product Resources

» Release Notes
» Documentation

L'arrêt du domaine par défaut peut être obtenu en utilisant l'option « Stop default domain ».

70.5.3. L'outil asadmin

J2EE application server est livré avec une application nommée asadmin, utilisable sur une ligne de commandes, pour administrer le serveur.

Cette application utilise deux modes de fonctionnement :

- la réception de commandes par la console après son lancement
- le passage des commandes en argument de la commande

Les commandes possèdent des noms bien définis en fonction de leurs actions et nécessitent souvent un ou plusieurs paramètres.

Exemple : démarrage d'un domaine

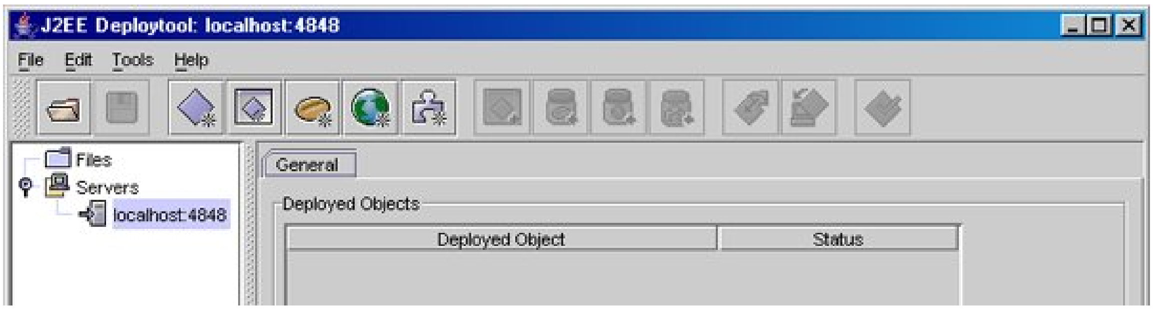
```
C:\>asadmin start-domain domain1

Starting Domain domain1, please wait.

Log redirected to C:\Sun\AppServer\domains\domain1\logs\server.log.

Domain domain1 started.
```

Exemple : arrêt d'un domaine

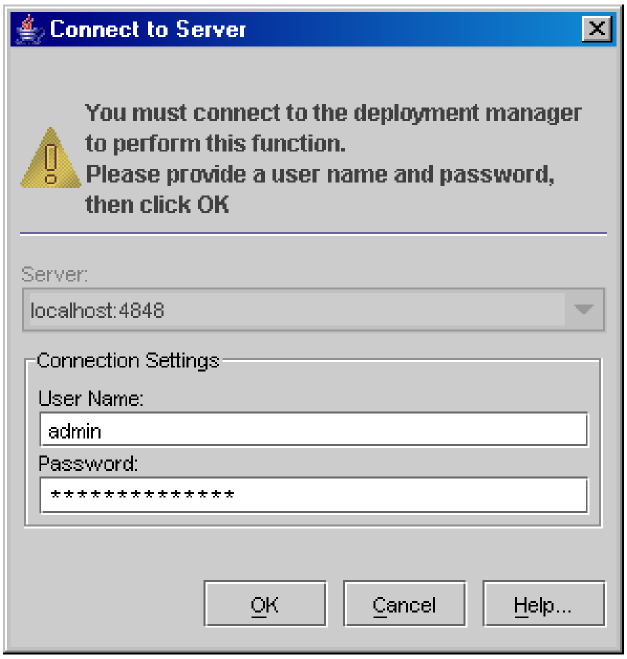
```
C:\>asadmin stop-domain domain1

Domain domain1 stopped.
```

70.5.4. Le déploiement d'applications

Pour déployer une application sous la forme d'un fichier war ou ear il suffit de copier le fichier dans le sous-répertoire domains/nom_du_domaine/autodeploy.



Il est nécessaire de s'authentifier auprès du serveur d'applications pour certaines opérations.



You must connect to the deployment manager to perform this function.
Please provide a user name and password, then click OK

Server: localhost:4848

Connection Settings

User Name: admin

Password: **************

70.5.5. La console d'administration

La console d'administration est une application web qui permet de configurer le serveur.

Pour l'utiliser, le serveur doit être lancé et il suffit de saisir dans un navigateur l'URL http://localhost:4848/asadmin

70.6. La présentation de Java EE 5.0

Le nom de la cinquième version de la plate-forme Java pour Entreprise a été simplifié : au lieu de se nommer J2EE (Java 2 Enterprise Edition) version 1.5, la plate-forme a été renommée Java EE 5 (Java Enterprise Edition).

L'accent est mis dans cette version sur la simplification des développements tout en conservant et en faisant évoluer les fonctionnalités proposées par la plate-forme J2EE.

J2EE est réputée pour sa complexité et pour certaines lourdeurs essentiellement liées aux nombreuses entités à développer (classes, interfaces, fichiers de configuration, ...). La version 5 de la plate-forme repose sur la version 5 de la plate-forme Java Standard Edition et profite donc de ses améliorations notamment les generics et les annotations.

L'utilisation intensive de ces dernières dans la version 5 de la plate-forme Java EE permet de simplifier les développements et ainsi de réduire le temps nécessaire à leur réalisation.

L'un des principaux buts de Java EE 5 est de conserver les fonctionnalités et la puissance de la plate-forme tout en simplifiant grandement le code à produire. Cette simplification repose essentiellement sur :

- L'utilisation intensive des annotations afin de réduire le volume de code et le nombre de fichiers à créer
- L'utilisation de valeurs et de comportements par défaut
- Les descripteurs de déploiement ne sont plus nécessaires que pour des besoins très particuliers

La simplification concerne aussi des sujets plus précis tels que l'utilisation des POJO ou l'injection de ressources.

Cette nouvelle version de la plate-forme, spécifiée dans la JSR 244, propose donc d'énormes simplifications dans le code à écrire par les développeurs.

Elle intègre aussi de nouvelles API :

- Java Server Faces pour le développement d'applications web avec la JSTL et un support pour AJAX
- Une nouvelle API pour la persistance des données reposant sur les POJO et les annotations : Java Persistence API
- La version 3.0 des EJB simplifie grandement l'utilisation de cette technologie
- Le support des dernières versions des API concernant les services web et permettant une mise en oeuvre d'une architecture de type SOA

Elle intègre aussi les dernières versions de la plupart des API qui formaient la version précédente de la plate-forme. Ainsi la version 5 de l'édition Entreprise de Java inclut de nombreuses spécifications :

Technologies pour les services web	
Implementing Enterprise Web Services	JSR 109
Java API for XML-Based Web Services (JAX-WS) 2.0	JSR 224
Java API for XML-Based RPC (JAX-RPC) 1.1	JSR 101
Java Architecture for XML Binding (JAXB) 2.0	JSR 222
SOAP with Attachments API for Java (SAAJ)	JSR 67
Web Service Metadata for the Java Platform	JSR 181
Technologies pour les composants	
Enterprise JavaBeans 3.0	JSR 220
J2EE Connector Architecture 1.5	JSR 112
Java Servlet 2.5	JSR 154
JavaServer Faces 1.2	JSR 252
JavaServer Pages 2.1	JSR 245
JavaServer Pages Standard Tag Library	JSR 52
Technologies de gestion et déploiement	
J2EE Management	JSR 77
J2EE Application Deployment	JSR 88
Java Authorization Contract for Containers	JSR 115
Autres technologies	
Common Annotations for the Java Platform	JSR 250
Java Transaction API (JTA)	JSR 907
JavaBeans Activation Framework (JAF) 1.1	JSR 925

JavaMail	JSR 919
Streaming API for XML (StAX) 1.0	JSR 173

70.6.1. La simplification des développements

La version 5 de la plate-forme Java EE fait un usage important des annotations introduites dans la plate-forme Java SE 5.0. Les annotations sont des métadonnées qui seront utilisées par le conteneur. Le code à écrire est ainsi réduit car certaines entités à définir ou règles à respecter sont simplement remplacées par l'utilisation d'une ou plusieurs annotations.

Historiquement, des tags étaient déjà utilisés notamment par Javadoc.

Une annotation commence par le caractère @ suivi par le nom de l'annotation éventuellement suivi d'une liste, entourée de parenthèses, de paramètres sous la forme de paires clé/valeur.

Les annotations précèdent par convention les modificateurs des entités qu'elles caractérisent. Elles correspondent à des classes particulières.

Les annotations n'ont aucune influence sur la logique des traitements du code mais elles influent sur la façon dont certains outils vont exécuter le code.

Java EE 5 propose des annotations pour de nombreux rôles :

- définition et utilisation des services web
- définition des EJB
- mapping objets / XML
- mapping objet / relationnel
- précision sur les informations de déploiement
- ...

L'utilisation des annotations n'est pas obligatoire : la configuration peut aussi être faite dans un descripteur de déploiement.

Le packaging a aussi été simplifié : il est maintenant possible d'écrire des EJB ou des services web sans devoir écrire de descripteur de déploiement sauf pour des besoins particuliers. La configuration est déduite par le conteneur à partir des annotations.

De nombreux attributs d'annotations possèdent des valeurs par défaut que le développeur pourra utiliser sans avoir à les préciser.

70.6.2. La version 3.0 des EJB

La version 3.0 des EJB propose une simplification de leur développement. Le travail des développeurs est réduit au profit d'une augmentation des traitements pris en charge par le conteneur :

- Le nombre de classes et d'interfaces à écrire est réduit
- Le descripteur de déploiement est optionnel car il n'est plus utile que pour des besoins spécifiques. Pour cela, la définition des composants utilise les annotations et l'injection de dépendance.
- Les EJB entités sont plus faciles à développer en faisant usage de la nouvelle API Java Persistence API pour le mapping O/R
- Les intercepteurs permettent de proposer des traitements avant ou après l'appel de méthodes à l'image de ce que peuvent proposer certaines fonctionnalités de l'AOP

Dans les versions antérieures des spécifications, les interactions entre le bean et le conteneur pour la gestion de son cycle de vie étaient réalisées par les méthodes ejbRemove(), setMessage(), setSessionContext(), ejbActivate() et ejbPassivate() des classes javax.ejb.SessionBean et javax.ejb.MessageDrivenBean. Même inutiles, ces méthodes devaient être écrites.

Dans la version 3.0, il suffit simplement d'utiliser les annotations définies dans les spécifications de Java EE dont voici les principales :

tag	Rôle
@Stateless	annote une classe qui est un composant de type EJB Session Stateless
@Stateful	annote une classe qui est un composant de type EJB Session Stateful
@PostConstruct	
@PreDestroy	
@PostActivate	
@PrePassivate	
@EJB	annote un EJB qui sera injecté par le conteneur
@WebServiceRef	annote un service web qui sera injecté par le conteneur
@Resource	annote une ressource différente d'un EJB ou d'un service web qui sera injectée par le conteneur
@MessageDriven	annote une classe qui est un composant de type EJB Message Driven
@TransactionAttribute	annote une classe (dans ce cas toutes ses méthodes) ou une méthode pour préciser les attributs d'appartenance à une transaction
@TransactionManagement	
@RolesAllowed, @PermitAll @DenyAll	annote une méthode pour indiquer ses permissions d'utilisation
@RolesReferenced	
@RunAs	

70.6.3. Un accès facilité aux ressources grâce à l'injection de dépendance

Le motif de conception injection de dépendance permet à une entité extérieure à un objet de lui fournir toutes les références sur les objets dont il dépend.

Avec Java EE 5, l'injection de dépendance peut être mise en oeuvre sur plusieurs types de ressources utilisés par un composant :

- EJB
- Services web
- DataSource
- EntityManager
- Queue et Topic
- SessionContext
- UserTransaction
- TimerService

Trois annotations permettent de mettre en oeuvre l'injection de dépendance :

- @EJB : pour les EJB
- @WebServiceRef : pour les services web
- @Ressource : pour tous les autres types de ressources supportés

Ces annotations peuvent être utilisées dans tout objet dont le cycle de vie est géré par un conteneur du serveur d'applications (EJB, service web, servlet, bean entité, ...).

L'injection de dépendance peut donc être mise en oeuvre dans les trois conteneurs de la plate-forme : EJB, web et client.

Ceci permet d'éviter l'utilisation directe de l'API JNDI pour obtenir une instance de la ressource stockée dans l'annuaire.

70.6.4. JPA : le nouvelle API de persistance

La nouvelle API Java Persistence API, développée sous la JSR 220, est ajoutée à la version 5.0 de la plate-forme Java EE. Cette API peut être utilisée dans les EJB mais aussi dans toutes les autres applications même celles utilisant Java SE. Son utilisation n'est ainsi pas réservée qu'à des développements avec la plate-forme Java EE.

Cette API propose les fonctionnalités suivantes :

- standardisation du mapping O/R
- utilisation de POJO
- mise en oeuvre des opérations de type CRUD au travers de l'objet EntityManager
- support de l'héritage et du polymorphisme
- requêtes en EJB Query Langage

Les entités sont de simples POJO enrichis d'annotations dédiées. Ces annotations permettent de préciser comment est réalisé le mapping entre une ou plusieurs tables d'une base de données et l'entité.

Exemple :

```
package fr.jmdoudoux.dej.jpa;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Personne implements Serializable {
  @Id
  @GeneratedValue
  private int id;

  private String prenom;

  private String nom;

  private static final long serialVersionUID = 1L;

  public Personne() {
    super();
  }

  public int getId() {
    return this.id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public String getPrenom() {
    return this.prenom;
  }

  public void setPrenom(String prenom) {
    this.prenom = prenom;
  }

  public String getNom() {
    return this.nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }
}
```

JPA propose une API pour manipuler ces entités notamment en utilisant un objet de type EntityManager.

Exemple : rechercher et supprimer une occurrence

```
    private EntityManager em;
...
    Personne personne = em.find(Personne.class, 4);
    if (personne == null) {
      System.out.println("Personne non trouvée");
    } else {
      em.remove(personne);
    }
```

La mise en oeuvre de JPA est détaillée dans le chapitre «JPA (Java Persistence API)»

70.6.5. Des services web plus simples à écrire

La simplification de JAVA EE 5 concerne aussi les services web : les services web sont plus simples à développer et le nombre de standards supportés a augmenté. Cette simplification est largement assurée par l'utilisation des annotations et de comportements par défaut.

Exemple :

```
package fr.jmdoudoux.dej.jaxws;

import javax.jws.WebService;

@WebService
public class MonService {

  public String saluer(String param) {
    return "Bonjour " + param;
  }
}
```

Avec Java EE 5, l'utilisation d'annotations a grandement simplifié le développement de services web.

Java EE 5 propose aussi plusieurs API concernant les services web : Java API for XML-Based Web Services (JAX-WS) 2.0 (JSR 224), Java Architecture for XML Binding (JAXB) 2.0 (JSR 222) et Web Services Metadata for the Java Platform (JSR 181).

JAX-WS 2.0 est la nouvelle API pour le développement de services web. Elle succède à JAX-RPC 1.1. Cette nouvelle version propose :

- l'utilisation des annotations
- le binding des données grâce à JAXB 2.0
- le support de SOAP 1.1 et 1.2
- le support de MTOM/XOP pour l'encodage optimisé des attachments
- le support des services web de type REST

La définition d'un service web consiste à utiliser l'annotation @WebService sur une classe.

Par défaut, toutes les méthodes publiques sont exposées en tant qu'opérations du service web. L'utilisation des annotations permet de ne pas avoir à définir de fichier de déploiement.

Pour modifier le mapping par défaut, certaines annotations peuvent être utilisées.

Exemple :

```
package fr.jmdoudoux.dej.jaxws;

import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService(name="MonServiceWS")
public class MonService {

  @WebMethod(operationName="direBonjour")
  public String saluer(String param) {
    return "Bonjour " + param;
  }
}
```

Par défaut, le WSDL d'un service web sera généré dynamiquement par le conteneur selon les spécifications de JAX-WS 2.

JAX-WS 2.0 propose aussi une API pour permettre l'appel de services web par un client de façon asynchrone. Cet appel asynchrone peut être utilisé avec n'importe quel service web : l'invocation de services web de façon asynchrone n'implique aucun traitement particulier côté serveur. C'est simplement l'invocation côté client qui est différente.

70.6.6. Le développement d'applications Web

Le framework Java Server Faces version 1.2 est inclus dans la plate-forme Java EE 5.

La bibliothèque JavaServer Pages Standard Tag Library (JSTL) est incluse dans la plate-forme Java EE 5. JSTL propose un langage d'expression pour faciliter la manipulation d'entités et un ensemble de tags personnalisés.

Les incompatibilités entre les langages d'expressions de la JSTL et des JSF ont été corrigées, ce qui leur permet d'être utilisés simultanément grâce à UEL (Unified EL).

70.6.7. Les autres fonctionnalités

La version 2.0 de JAXB offre un support complet des schémas XML.

L'API Streaming API for XML (StAX) défini une méthode pour parser un document XML à partir d'événements. Son mode de fonctionnement est différent de SAX : avec SAX le développeur écrit du code pour répondre à des événements émis par le parser, avec StAX c'est le programme qui pilote le parser.

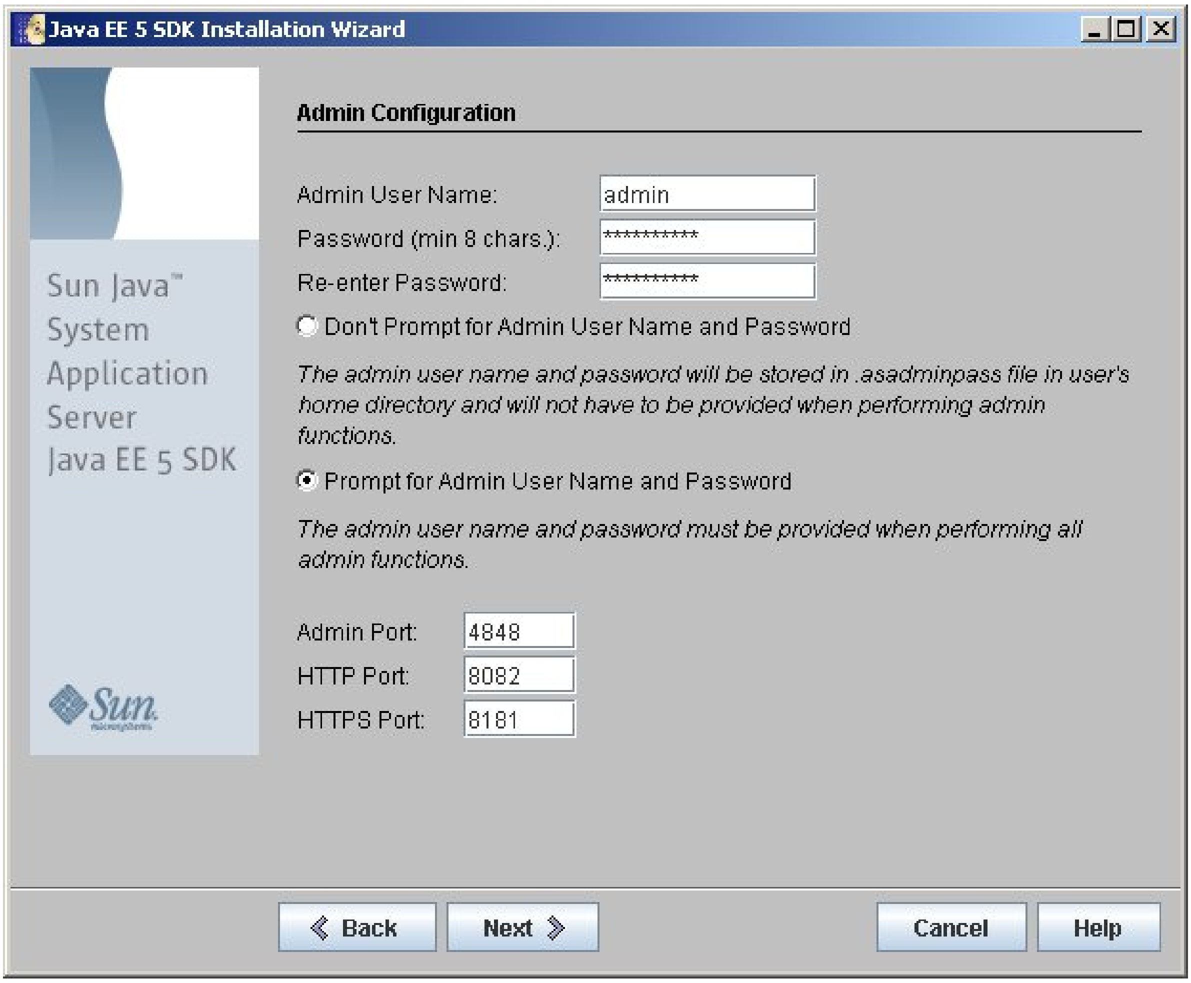
Remarque : ces deux API ont été ajoutées à la version 6 de Java SE.

70.6.8. L'installation du SDK Java EE 5 sous Windows

Téléchargez le fichier java_ee_sdk-5-windows.exe sur le site de Sun et exécutez le.

Les fichiers d'installation sont extraits puis le programme d'installation est lancé pour guider l'utilisateur avec un assistant :

- Sur la page « Welcome », cliquez sur le bouton « Next »
- Sur la page « Software Licence Agreement », lisez la licence et si vous l'acceptez, cliquez sur « Yes » puis sur le bouton « Next »
- Sur la page « select Installation Directory, modifiez si nécessaire le répertoire d'installation puis cliquez sur le bouton « Next »
- Sur la page « Admin Configuration », saisissez le mot de passe, sélectionnez la saisie ou non du mot de passe, et modifiez les ports au besoin puis cliquez sur le bouton « Next »



Admin Configuration

Admin User Name: admin

Password (min 8 chars.): **********

Re-enter Password: **********

○ Don't Prompt for Admin User Name and Password

The admin user name and password will be stored in .asadminpass file in user's home directory and will not have to be provided when performing admin functions.

● Prompt for Admin User Name and Password

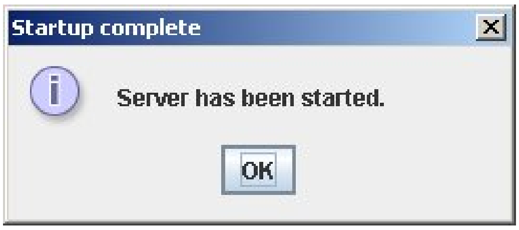
The admin user name and password must be provided when performing all admin functions.

Admin Port: 4848

HTTP Port: 8082

HTTPS Port: 8181

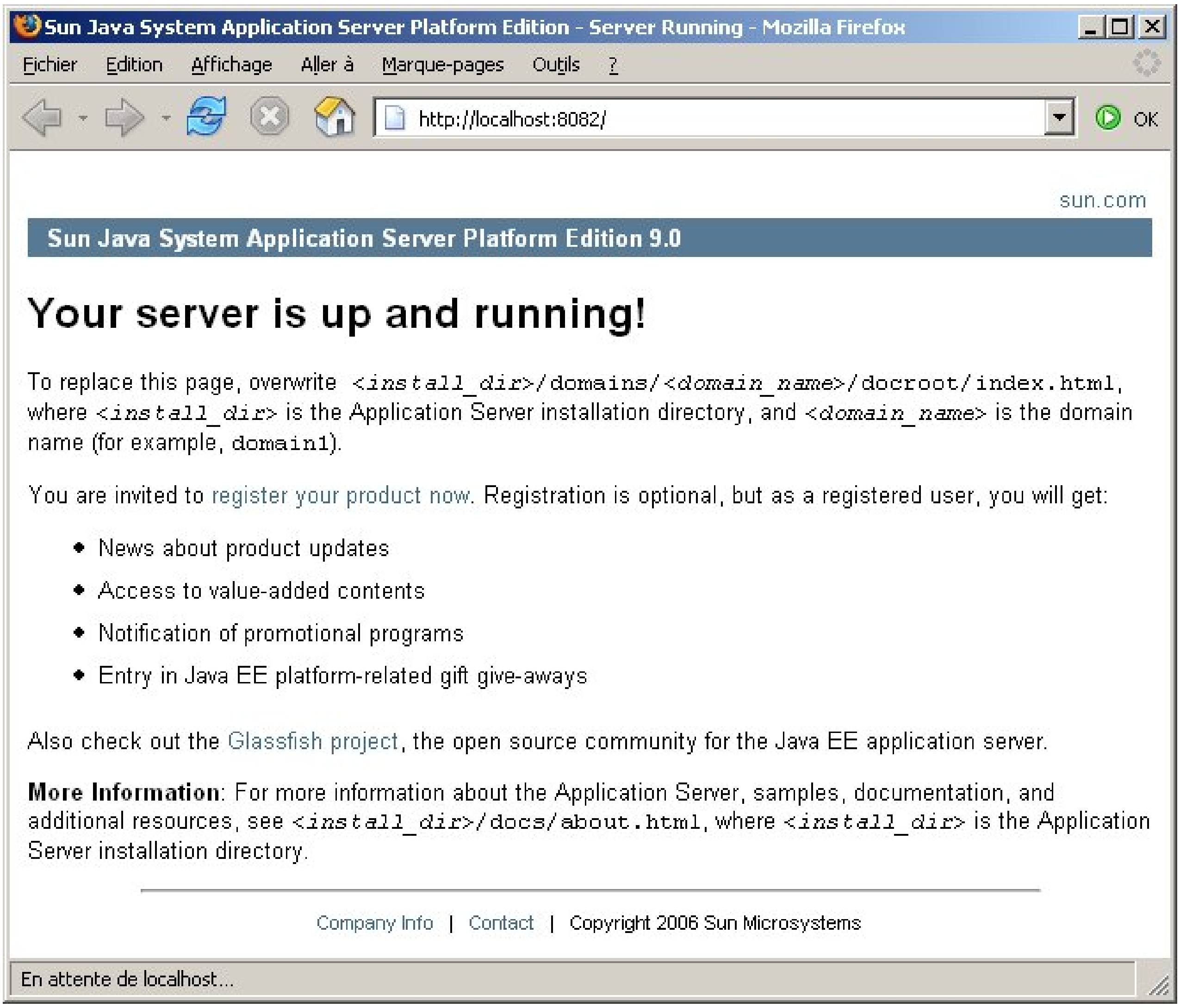
- Sur la page « Installation options », cochez «Create desktop shortcut to autodeploy directory» et «create windows service» en plus des options déjà cochées, puis cliquez sur le bouton «Next». Le programme d'installation vérifie l'espace disque requis.
- Sur la page « Ready to Install », cliquez sur le bouton «Install Now».
- Le programme d'installation copie les fichiers.
- Sur la page « Installation Complete », cliquez sur le bouton « Start server »



Server has been started.

Cliquez sur le bouton «OK» puis sur le bouton « Finish »

Ouvrez un navigateur sur l'url http://localhost:8082



Sun Java System Application Server Platform Edition - Server Running - Mozilla Firefox
Fichier
Edition
Affichage
Aller à
Marque-pages
Outils
?
http://localhost:8082/
OK
sun.com

Sun Java System Application Server Platform Edition 9.0

Your server is up and running!

To replace this page, overwrite `<install_dir>/domains/<domain_name>/docroot/index.html`, where `<install_dir>` is the Application Server installation directory, and `<domain_name>` is the domain name (for example, `domain1`).

You are invited to register your product now. Registration is optional, but as a registered user, you will get:

- News about product updates
- Access to value-added contents
- Notification of promotional programs
- Entry in Java EE platform-related gift give-aways

Also check out the Glassfish project, the open source community for the Java EE application server.

More Information: For more information about the Application Server, samples, documentation, and additional resources, see `<install_dir>/docs/about.html`, where `<install_dir>` is the Application Server installation directory.

Company Info | Contact | Copyright 2006 Sun Microsystems

En attente de localhost...

70.7. La présentation de Java EE 6

La version 6 de la plate-forme Java EE est diffusée depuis décembre 2009. Elle s'appuie sur la plate-forme Java SE 6 dont elle utilise de nombreuses API (JDBC, JNDI, JAXB, JAXP, RMI, JMX, ...).

Cette nouvelle version de la plate-forme a plusieurs caractéristiques :

- Plus riche : de nouvelles spécifications sont ajoutées et les spécifications majeures sont enrichies
- Plus facile : généralisation de l'utilisation des POJO et des annotations notamment dans le tiers web
- Plus légère : création de la notion de profiles et de pruning, EJB Lite
- Toujours aussi robuste après 10 ans d'existence

L'implémentation de référence est le projet open source GlassFish version 3.

70.7.1. Les spécifications de Java EE 6

C'est la version de la plate-forme avec le plus grand nombre de spécifications. Elle contient 28 spécifications :

Technologies	JSR
Java Platform, Enterprise Edition 6 (Java EE 6) (avec Managed Beans 1.0)	JSR 316

Technologies relatives aux services web	
Java API for RESTful Web Services (JAX-RS) 1.1	JSR 311
Implementing Enterprise Web Services 1.3	JSR 109
Java API for XML-Based Web Services (JAX-WS) 2.2	JSR 224
Java Architecture for XML Binding (JAXB) 2.2	JSR 222
Web Services Metadata for the Java Platform	JSR 181
Java API for XML-Based RPC (JAX-RPC) 1.1	JSR 101
Java APIs for XML Messaging 1.3	JSR 67
Java API for XML Registries (JAXR) 1.0	JSR 93
Technologies relatives aux développements web	
Java Servlet 3.0	JSR 315
JavaServer Faces 2.0	JSR 314
JavaServer Pages 2.2/Expression Language 2.2	JSR 245
Standard Tag Library for JavaServer Pages (JSTL) 1.2	JSR 52
Debugging Support for Other Languages 1.0	JSR 45
Technologies relatives aux développements d'applications	
Contexts and Dependency Injection for Java (Web Beans 1.0)	JSR 299
Dependency Injection for Java 1.0	JSR 330
Bean Validation 1.0	JSR 303
EnterpriseJava Beans 3.1 (avec Interceptors 1.1)	JSR 318
Java EE Connector Architecture 1.6	JSR 322
Java Persistence 2.0	JSR 317
Common Annotations for the Java Platform 1.1	JSR 250
Java Message Service API 1.1	JSR 914
Java Transaction API (JTA) 1.1	JSR 907
JavaMail 1.4	JSR 919
Gestion et sécurité	
Java Authentication Service Provider Interface for Containers	JSR 196
Java Authorization Contract for Containers 1.3	JSR 115
Java EE Application Deployment 1.2	JSR 88
J2EE Management 1.1	JSR 77

70.7.2. Une plate-forme plus légère

Une critique récurrente de la plate-forme Java EE est qu'elle est très riche et donc complexe. La plupart des applications de petite taille ou de taille moyenne n'ont pas besoin de toutes les technologies proposées par la plate-forme.

Cette richesse est aussi liée au fait que certaines technologies sont obsolètes car remplacées par une nouvelle technologie : ces anciennes versions sont cependant conservées pour des raisons de compatibilité.

70.7.2.1. La notion de profile

Java EE 6 définit la notion de profile qui est un sous-ensemble et/ou un sur-ensemble de Java EE 6 pour des besoins particuliers.

Les profiles sont conçus pour proposer une solution technique particulière pour des besoins spécifiques. Cela permet à un fournisseur de n'avoir à proposer que le support des technologies incluses dans le profile plutôt que d'avoir à implémenter toutes celles de la plate-forme Java EE.

Java EE 6 définit un premier profile : le web profile qui se concentre sur le développement d'applications de type web. Il doit pouvoir être exécuté dans un simple conteneur web car le packaging se fait dans une archive de type war.

Le web profile est composé de plusieurs spécifications pour définir un sous-ensemble de Java EE :

Servlet 3.0	JSP 2.2	EL 1.2
JSTL 1.2	JSF 2.0	EJB Lite 3.1
JTA 1.1	JPA 2.0	Bean Validation 1.0
DI 1.0	CDI 1.0	Interceptors 1.1

D'autres profiles devraient être définis ultérieurement de façon indépendante des évolutions de la plate-forme Java EE.

Un fournisseur n'a plus l'obligation de fournir une implémentation complète de la spécification Java EE mais peut simplement fournir une implémentation d'un profile. Bien sûr dans ce cas, les fonctionnalités utilisables sont uniquement définies dans ce profile.

La notion de profile a fait l'objet de nombreux débats au sein des membres de la JSR notamment sur sa définition, la compatibilité et sur la confusion et les interrogations qu'elle peut susciter chez les développeurs.

70.7.2.2. La notion de pruning

La plate-forme Java EE est devenue au fil des versions imposante en terme de nombre de spécifications, de fonctionnalités et d'API. Aucunes des précédentes versions n'a supprimé de fonctionnalités même si certaines sont obsolètes car remplacées, rarement adoptées ou utilisées. Cela pose plusieurs problèmes :

- Pour le développeur : la taille du SDK augmente avec le nombre d'API
- Pour les fournisseurs de serveurs d'applications : l'obligation de support pour ces fonctionnalités avec un accroissement de la taille des serveurs, de leur consommation en ressources et de leur temps de démarrage

Certaines de ces fonctionnalités sont de plus obsolètes car remplacées par une plus récente ou ne sont que peu ou pas utilisées.

Java EE 6 entame donc une cure d'amaigrissement de la plate-forme en définissant un ensemble d'API déclarées comme pruned.

Les fonctionnalités déclarées pruned dans Java EE 6 sont :

- Entity CMP 2.x : cette API est avantageusement remplacée par JPA
- JAX-RPC : cette API est avantageusement remplacée par JAX-WS
- JAX-R : cette API est très peu utilisée car l'utilisation de UDDI n'est pas trés répandue
- JSR 88 (Java EE Application Deployment) : cette API est très peu mise en oeuvre par les fournisseurs de serveurs d'applications

Elles doivent toujours être disponibles dans une implémentation de Java EE 6 mais elles seront certainement amenées à disparaître dans la prochaine version de la plate-forme Java EE et leur support ne sera plus obligatoire dans les implémentations des serveurs d'applications. Le concept de pruned est plus fort que le concept de deprecated de la plate-forme Java SE.

Il est donc raisonnable de ne plus utiliser ces fonctionnalités dans de nouveaux développements et de migrer progressivement les applications existantes qui les utilisent.

Le but est de simplifier le développement des prochaines versions de conteneurs des serveurs d'applications qui n'auront plus l'obligation de fournir une implémentation des fonctionnalités déclarées pruned.

70.7.3. Les évolutions dans les spécifications existantes

Plusieurs spécifications existantes dans la version 5 de Java EE 5 ont été mises à jour dans la plate-forme Java EE 6.

70.7.3.1. Servlet 3.0

Cette nouvelle version de l'API servlet a pour but de simplifier son utilisation. Comme pour la plupart des API de Java EE, cette simplification repose en grande partie sur l'utilisation d'annotations telles que @WebServlet, @ServletFilter, @WebServletContextListener, @InitParam, @WebFilter, ... pour déclarer des entités ce qui permet de rendre le descripteur de déploiement web.xml plus léger voire optionnel.

Bien qu'une servlet puisse être annotée avec @WebServlet, elle doit toujours hériter de la classe HttpServlet notamment pour permettre d'identifier de façon unique les méthodes à invoquer selon le type de requête http à traiter.

Même si la plupart des développements n'utilise plus directement cette API au profit de frameworks, les développeurs de ces derniers vont pouvoir utiliser les nouvelles fonctionnalités de la spécification.

Le fichier web.xml est rendu modulaire et peut être rédigé sous la forme de fragments qui seront agrégés et fusionnés par le conteneur au moment du déploiement.

Un fragment est une portion du descripteur de déploiement dont le tag racine est <web-fragment> qui peut contenir la définition de tout ou partie des entités configurables dans le descripteur de déploiement.

Ceci pourra ainsi permettre d'éviter d'avoir à déclarer des servlets ou des filtres d'un framework utilisés dans une webapp. Le framework pourra simplement définir le fragment pour que ces déclarations soient prises en compte par le conteneur.

Le conteneur recherche des fragments du fichier web.xml dans le classpath de la webapp (WEB-INF/classes et dans les fichiers jar du répertoire WEB-INF/lib) pour les agréger et composer le fichier web.xml.

Servlet 3.0 propose un support des invocations asynchrones en utilisant l'attribut asyncSupported=True de l'annotation @WebServlet. Une api dédiée est proposée pour gérer l'état d'une requête.

L'interface ServletContext propose des méthodes pour permettre d'enregistrer dynamiquement des servlets, des filtres et des listeners.

L'implémentation de référence est GlassFish v3.

70.7.3.2. JSF 2.0

Cette nouvelle version de JSF a pour but de simplifier son utilisation. Comme pour la plupart des API de Java EE, cette simplification repose en grande partie sur l'utilisation d'annotations telles que @ManagedBean, @ManagedProperty, @ApplicationScoped, @SessionScoped, @FacesValidator, @FacesConverter ... qui permettent de rendre le fichier faces-config.xml beaucoup plus petit.

JSF 2.0 utilise le projet open source Facelets comme technologie pour la partie vue : l'organisation du contenu des pages et des composants est facilitée grâce à l'utilisation de Facelets. Le développement de composants a été grandement simplifié grâce à Facelets en utilisant la notion de composition qui met en oeuvre XHTML et un tag JSF.

JSF propose en standard un support de fonctionnalités mettant en oeuvre Ajax sur des fonctions JavaScript standardisées contenues dans le fichier jsf.js que chaque implémentation doit fournir. Le cycle de vie de traitement d'une requête JSF a dû être adapté pour les traitements Ajax en incorporant la notion de page partielle (partial page).

L'implémentation de référence est Mojorra.

70.7.3.3. Les EJB 3.1

La version 3.1 des EJB poursuit l'effort de simplification de la version 3.0 tout en apportant de nouvelles fonctionnalités et un enrichissement des fonctionnalités existantes :

- Les interfaces Local sont optionnelles pour les EJB Session
- EJB Singleton : le conteneur garantit qu'une seule instance de l'EJB sera accessible par défaut de façon thread-safe. Il est cependant possible d'avoir un contrôle sur la gestion de la concurrence des accès.
- Les invocations asynchrones des session beans
- EJB Timer
- Packaging dans un war qui est utilisé notamment dans le web profile
- Les noms JNDI portables pour faciliter le déploiement sur plusieurs serveurs d'applications.
- Le conteneur embarqué pour exécuter des EJB sur la plate-forme Java SE
- EJB Lite qui est utilisé, en autres, dans le web profile en ne proposant qu'une petite partie des fonctionnalités proposées par les EJB (Session Bean, transactions et sécurité) en occultant notamment les appels distants, les MDB et le scheduling.

L'implémentation de référence est GlassFish v3.

Le chapitre «Les EJB 3.1» contient une description détaillée de cette API.

70.7.3.4. JPA 2.0

JPA 2.0 a fait l'objet d'une spécification dédiée.

Les possibilités de mapping sont enrichies avec

- le support des collections qui ne représentent pas de relations avec des entités grâce à l'annotation @ElementCollection (mappe une collection d'éléments ou une collection de type Map dans une table dédiée)
- le support des relations unidirectionnelles one-to-many

La version 2.0 de JPA propose de nouvelles fonctionnalités manquantes dans la version précédente :

- Une gestion plus fine des verrous (locks) notamment avec le support des verrous pessimistes
- Une API pour coder les critères de recherche de façon dynamique sous la forme d'un graphe d'objets
- Une API simple pour la gestion du cache

L'implémentation de référence est EclipseLink.

70.7.3.5. JAX-WS 2.2

Cette API permet la mise en oeuvre de services web de type Soap. La version 2.2 est incluse dans Java EE 6.

L'implémentation de référence est Metro.

70.7.3.6. Interceptors 1.1

Ils peuvent être utilisés sur les EJB et les Managed Beans.

70.7.4. Les nouvelles API

De nouvelles API ont été ajoutées à la version 6 de la plate-forme Java EE.

70.7.4.1. JAX-RS 1.1

L'API JAX-RS permet de mettre en oeuvre des services web de type RestFul.

L'inclusion de JAX-RS 1.1 dans la plate-forme Java EE suit la tendance à l'adoption grandissante des services web de type REST.

JAX-RS utilise des annotations sur des POJO pour masquer la complexité de traitement des requêtes et de génération des réponses ce qui permet de simplifier leurs mises en oeuvre.

Plusieurs annotations sont définies :

- @Path permet de déterminer l'URL d'accès à la ressource
- @GET, @POST, @PUT and @DELETE permet de déterminer la méthode http utilisée pour accéder à la ressource
- @QueryParam, @PathParam, @CookieParam et @HeaderParam permettent d'extraire les valeurs de la requête http (paramètres, cookies, header)
- @Produces, @Consumes permet de préciser le format de restitution ou de consommation de la ressource

Cette API repose sur l'utilisation de ces annotations sur un POJO.

La version utilisée dans Java EE 6 est la 1.1.

Exemple :

```
@Path("/helloworld")
public class HelloWorldRS {
  @GET
  @Produces("text/plain")
  public String saluer() {
    return "Hello World";
  }
}
```

Cette version permet une utilisation de l'API avec les EJB.

L'implémentation de référence est Jersey.

70.7.4.2. Contexte and Dependency Injection (WebBeans) 1.0

Le but de cette spécification est de faciliter les interactions entre la couche de présentation, la couche métier et la couche de persistance notamment à l'aide de beans qui pourront être utilisés par plusieurs couches.

Cette spécification a été influencée par plusieurs projets open source notamment JBoss Seam, Google Guice et Spring.

CDI a pour objectif de fournir une glue entre les couches mettant en oeuvre JSF, EJB et JPA. Elle permet notamment d'enregistrer et de gérer des EJB, des entités JPA et des ManagedBeans sous la forme de composants qui seront injectables et utilisables grâce à EL (Expression Language).

CDI peut remplacer les backing beans de JSF.

La gestion du cycle de vie des composants par CDI se fait par rapport à un contexte (requête, session et application mais aussi deux nouveaux contextes nommés dependent et conversation).

L'implémentation de référence est JBoss Seams.

70.7.4.3. Dependency Injection 1.0

Le but de cette JSR est de proposer la standardisation d'un ensemble d'annotations utilisables avec n'importe quel moteur d'injection : le but n'est pas de spécifier un tel moteur.

Elle définit plusieurs annotations :

- @Inject : identifier un constructeur, une méthode ou un champ injectable
- @Named : permet de qualifier une dépendance avec une chaîne de caractères
- @Qualifier : permet de qualifier une dépendance
- @Scope : permet de définir la portée de l'injection
- @Singleton : injection d'une instance unique

Google Guice et Spring 3.0 implémentent cette spécification.

70.7.4.4. Bean Validation 1.0

Cette API standardise la validation de données.

Les contraintes sont définies dans les beans avec des annotations (@NotNull, @Size, @Past, ...)

Bean Validation propose une API pour valider des données, définir ses propres contraintes et rechercher des contraintes.

Cette API est utilisée notamment par JSF 2.0 et JPA 2.0

L'implémentation de référence est Hibernate Validator 4.0.

Le chapitre «La validation des données» contient une description détaillée de cette API.

70.7.4.5. Managed Beans 1.0

C'est un modèle de composants légers : ce sont des POJO gérés par le conteneur.

Les managed beans supportent plusieurs services :

- Injection de dépendances : @Resource, @Inject
- Gestion du cycle de vie : @PostConstruct, @PreDestroy
- Intercepteurs : @Interceptor, @AroundInvoke

Un managed bean est un POJO annoté avec l'annotation @javax.annotation.ManagedBean. Cette annotation est issue de la JSR 250 (Commons annotations).

70.8. La présentation de Java EE 7

Les spécifications de Java EE 7 sont définies dans la JSR 342. Cette spécification ne définit pas directement d'API mais elle liste celles qui sont incluses dans la plate-forme et comment celles-ci interagissent entre-elles. Elle définit aussi des éléments précis de la plate-forme comme les transactions, la sécurité, l'assemblage, le déploiement, ...

Java EE 7 a plusieurs objectifs :

- poursuivre l'amélioration de la productivité et la simplification de l'utilisation de la plate-forme
- le support de HTML 5 (WebSocket, Json, HTML5 forms, ...)

- l'ajout de nouvelles fonctionnalités : Batch API (modèle de programmation pour les applications batch) et Concurrency Utilities API (exécution de traitements asynchrones sans avoir à utiliser JMS)

Les spécifications de Java EE 7 furent officiellement diffusées en juin 2013. Java EE 7 s'appuie sur Java SE 7. Cette spécification contient 14 nouvelles JSR et 9 versions de maintenance (Maintenance Release) de JSR.

Java EE 7 inclut plusieurs nouvelles API :

- Java API for JSON Processing 1.0 (JSR 353)
- Java API for WebSocket 1.0 (JSR 356)
- Batch Applications for the Java Platform 1.0 (JSR 352)
- Concurrency utilities for Java EE 1.0 (JSR 236)

Java EE 7 inclut des API ayant une mise à jour majeure :

- Java Message Service 2.0 (JSR 343)
- Expression Language 3.0 (JSR 341)
- JAX-RS : Java API for Restful Web Services 2.0 (JSR 339)

Java EE 7 inclut des API mises à jour :

- JPA 2.1 (JSR 338)
- EJB 3.2 (JSR 345)
- Servlet 3.1 (JSR 340)
- Java Server Faces 2.2 (JSR 344)
- Contexts and Dependency Injection for Java EE 1.1 (JSR 346)
- Bean Validation 1.1 (JSR 349)
- Interceptors 1.2 (JSR 318)
- Java EE Connector Architecture 1.7 (JSR 322)

Les JSR mises à jour (Maintenance Release) sont :

- Web Services for Java EE 1.4 (JSR 109)
- Java Authorization Service Provider Contract for Containers 1.5 (JACC 1.5) (JSR 115)
- Java Authentication Service Provider Interface for Containers 1.1 (JASPIC 1.1) (JSR 196)
- JavaServer Pages 2.3 (JSR 245)
- Common Annotations for the Java Platform 1.2 (JSR 250)
- Java Transaction API 1.2 (JSR 907)
- JavaMail 1.5 (JSR 919)

Initialement Java EE 7 devait intégrer la spécification Java Temporary Caching API 1.0 (JSR 107) et des fonctionnalités relatives au Cloud (par exemple State Management (JSR 350)) mais elles sont repoussées dans la prochaine version de Java EE.

Java EE 7 poursuit la simplification de l'utilisation de la plate-forme entamée depuis Java EE 5 notamment :

- la nouvelle version 2.0 de l'API JMS
- des changements dans la configuration : ressources de type fabrique de connexions par défaut pour la base de données et le broker JMS, amélioration dans la déclaration des permissions de sécurité, ...
- les technologies déclarées pruned dans Java EE 6 sont optionnelles dans Java EE 7 : EJB Entity Beans, JAX-RPC 1.1, JAXR 1.0 et la JSR-88

JAX-RS 2.0 est ajoutée dans le Web Profile.

L'implémentation de référence est la version 4.0 du serveur d'applications Glassfish.

70.8.1. Java API for JSON Processing 1.0 (JSR 353)

La JSR 353 définit les spécifications d'une nouvelle API qui permet de produire et consommer des documents JSON. La version 1.0 ne propose rien concernant le binding entre un document et des objets Java.

Elle propose deux API pour réaliser ces tâches :

- streaming API qui est une API de bas niveau reposant sur le traitement d'événements émis lors du parcours du document JSON et une API de type fluent pour générer un document
- model API qui permet d'obtenir ou de créer un graphe d'objets qui encapsule les éléments du document.

Les principales interfaces de l'API Streaming, contenues dans le package javax.json.stream sont :

- JsonParser : pour le parcours séquentiel d'un document JSON avec l'émission d'événements
- JsonGenerator : pour la création séquentielle des éléments d'un document JSON

Les principales interfaces de l'API Model, contenues dans le package javax.json sont :

- JsonBuilder : pour créer des éléments d'un document JSON (objets et tableaux)
- JsonReader : pour lire un élément d'un document JSON à partir d'un flux
- JsonWriter : pour écrire un élément d'un document JSON

Pour permettre une utilisation de différentes implémentations, l'API propose des fabriques.

L'utilisation de cette API est détaillée dans le chapitre «JSON-P».

70.8.2. Java API for WebSocket 1.0 (JSR 356)

WebSocket est un protocole de communication bi-directionnel en mode full duplex

Les WebSockets sont standardisés par l'IETF sous la RFC 6455

Les WebSockets font parties des spécifications HTML 5 : l'API Javascript définie par le W3C est à l'état Candidat Recommendation

La JSR 356 définit les spécifications de l'API « Java API for WebSocket 1.0 » qui propose une utilisation côté client et serveur.

Elle propose deux modèles de développement :

- utilisation d'annotations sur des POJO
- l'utilisation de l'API notamment en implémentant l'interface Endpoint

L'API WebSocket 1.0 définit plusieurs annotations

Annotation	S'applique sur	Rôle
@ServerEndPoint	Classe	Déclarer la classe comme étant un endpoint client de la websocket
@OnOpen	Méthode	Intercepter l'ouverture de la websocket
@OnMessage	Méthode	Intercepter un message de la websocket
@PathParam	Paramètre d'une méthode	Définir un paramètre dont la valeur sera extraite de l'URI
@OnError	Méthode	Intercepter les erreurs durant les échanges
@OnClose	Méthode	Intercepter la fermeture du websocket
@ClientEndPoint	Classe	Déclarer la classe comme étant un client du websocket

Elle consiste essentiellement à réagir à certains événements liés au cycle de vie d'une WebSocket.

La mise en oeuvre repose sur plusieurs concepts :

- Endpoint : encapsule un endpoint et contient les traitements des différents événements

- Session : encapsule une conversation
- MessageHandler : pour traiter un message reçu au format texte ou binaire
- Encoder/Decoder : pour permettre l'encodage et le décodage d'un objet

L'utilisation de cette API est détaillée dans le chapitre «L'API Websocket».

70.8.3. Batch Applications for the Java Platform 1.0 (JSR 352)

La JSR 352 propose une standardisation d'une API pour le développement de traitements de type batch. Aussi appelé traitements par lots, ce type d'applications existe depuis toujours dans l'informatique : elle permet de traiter grâce à différentes étapes de grandes quantités de données avec des algorithmes plus ou moins complexes sans interaction si tout se passe bien.

Les traitements batch ou traitements par lot (batch processing) sont des traitements automatisés sur un ensemble de données. Généralement ces traitements effectuent des opérations complexes ou lourdes sur un volume conséquent de données : leurs temps d'exécution est généralement long, ce qui les empêchent d'être exécutés en temps réels.

Les traitements métier d'un batch sont généralement composés d'une ou plusieurs étapes.

La spécification ne précise rien concernant la planification de l'exécution des batches car différentes solutions existent déjà notamment les EJB Timer dans la plate-forme Java EE.

La JSR 352 est une spécification qui définit un modèle de programmation pour des traitements batch.

Ce modèle repose sur plusieurs concepts :

- Batch Job : un traitement batch composé d'un ou plusieurs steps exécutés séquentiellement.
- Step : une étape particulière d'un Job
- Job Specification Langage : description d'un Job en utilisant XML
- ItemReader : lecture d'un élément à traiter dans un Step
- ItemProcessor : traitements à réaliser sur un élément
- ItemWriter : écriture d'un élément traité dans un Step
- JobOperator : permet un accès au JobRepository

Lors de l'exécution d'un batch, plusieurs concepts sont utilisés :

- JobRepository : encapsule les jobs executés et ceux en cours d'exécution. Il contient notamment des instances de type Job, JobExecution et StepExecution
- JobInstance : encapsule une exécution logique d'un job
- JobExecution : encapsule le résultat d'une exécution d'un job (réussie ou échouée)
- JobParameters : encapsule les paramètres fournis à l'exécution d'un job
- StepExecution : encapsule une tentative d'exécution d'un step

Les classes et interfaces de l'API Batch processing sont contenues dans le package javax.batch.

Un job est un traitement qui peut être composé d'une ou plusieurs étapes (Step).

Un Step peut être de deux types :

- Chunked qui est orienté traitement d'un élément
- Batchlet qui est orienté traitement d'une tâche

Un step de type chunked met en oeuvre le modèle de conception lecture/traitement/écriture (read/process/write) d'un élément. Pour cela, trois types de classes sont définies : ItemReader, ItemProcessor et ItemWriter. Chaque élément est lu et traité.

Exemple :

```
package fr.jmdoudoux.dej.javaee7.batch;

import java.io.Serializable;
import java.util.Iterator;
import java.util.List;
import javax.batch.api.chunk.AbstractItemReader;
import javax.ejb.EJB;
import javax.inject.Named;

@Named("MonItemReader")
public class MonItemReader extends AbstractItemReader
{
    @EJB
    private ElementBean elementBean;
    Iterator<Element> elementsIterator;

    @Override
    public void open(Serializable e) throws Exception {
        List<Element> elements = elementBean.getElements();
        elementsIterator = elements.iterator();
    }

    @Override
    public Object readItem() throws Exception {
        return elementsIterator.hasNext() ? elementsIterator.next() : null;
    }
}
```

Exemple :

```
package fr.jmdoudoux.dej.javaee7.batch;

import javax.batch.api.chunk.ItemProcessor;
import javax.inject.Named;

@Named("MonItemProcessor")
public class MonItemProcessor implements ItemProcessor {

    @Override
    public Object processItem(Object obj) throws Exception {
        Element element = (Element) obj;
        int valeur = element.getValeur();
        element.setValeur(valeur + 10);
        return element;
    }
}
```

L'écriture des éléments se fait par paquets : la gestion des checkpoints dans les transactions est configurable dans ce modèle ce qui permet d'avoir des commit à intervalles réguliers.

Un step de type Bachlet ne repose sur aucun modèle prédéfini : les traitements sont invoqués une seule fois dans leur intégralité et se terminent.

Le contrôle des Jobs se fait grâce au JSL (Job Specification Language). Le JSL est un document XML qui permet de décrire les fonctionnalités d'un Job.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<job id="monJob" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
version="1.0">
    <step id="traitement">
        <chunk item-count="1">
            <reader ref="MonItemReader"></reader>
            <processor ref="MonItemProcessor"></processor>
            <writer ref="MonItemWriter"></writer>
        </chunk>
    </step>
</job>
```

L'API propose un support de fonctionnalités avancées notamment :

- les checkpoints : pour permettre la définition de points de reprise qui peuvent permettre de reprendre l'exécution d'un step qui se serait arrêté prématurément. Les checkpoints s'utilisent sur des step de type chunked
- la gestion des exceptions pour permettre une retentative du step ou de le passer
- une définition de la séquence de traitements d'un job (décision, transition, fail, end, stop, ...)
- l'exécution en parallèle d'un même step en partitionnant les données à traiter

70.8.4. Concurrency utilities for Java EE (JSR 236)

Il ne faut pas utiliser l'API Concurrency de Java SE dans une application Java EE. Le but de la spécification Concurrency Utilities définit par la JSR 236 est de permettre la mise en oeuvre des traitements concurrents dans un conteneur Java EE. Elle enrichit la JSR 166.

Les spécifications de cette API ont plusieurs objectifs :

- permettre à des composants Java EE de réaliser des traitements concurrents tout en préservant l'intégrité des containers Java EE dans lequel le composant peut s'exécuter
- assurer une cohérence entre le modèle de programmation concurrente de Java SE et Java EE (notamment en étant une extension de l'API Concurrent Utilities (JSR 166)

Les classes et interfaces de cette spécifications sont dans le package javax.enterprise.concurrent.

Chaque serveur d'applications Java EE propose une implémentation de l'interface ManagedExecutorService qui hérite de l'interface java.util.concurrent.ExecutorService. Pour obtenir une instance, il faut soit se la faire injecter sous la forme d'une ressource soit l'obtenir grâce à un lookup JNDI (dans ce dernier cas, il est nécessaire de déclarer la ressource dans un descripteur de déploiement).

Exemple (code Java 7) :

```
@Resource(name="concurrent/monExecutor")
ManagedExecutorService managedExecutor;
```

Exemple dans le description de déploiement web.xml d'une webapp

Exemple :

```
<resource-env-ref>
  <resource-env-ref-name>concurrent/monExecutor</resource-env-ref-name>
  <resource-env-ref-type>javax.enterprise.concurrent.ManagedExecutorService
</resource-env-ref-type>
</resource-env-ref>
```

Il est possible de définir plusieurs ManagedExecutorService avec différentes configurations dans le serveur d'applications, chacun ayant un nom permettant de les identifier, en les associant au sous contexte JNDI java:comp/env/concurrent.

Les traitements qui seront exécutés par le ManagedExecutorService doivent être encapsulés dans une classe qui doit implémenter l'interface java.lang.Runnable ou java.util.concurrent.Callable

Exemple (code Java 7) :

```
public class MaTache implements Runnable {
  public void run() {
    // Traitements a effectuer
  }
}
```

Le plus simple pour demander l'exécution d'une tâche est de passer son instance en paramètre de la méthode submit(). Elle renvoie une instance de type java.util.concurrent.Future qui permet de gérer et de déterminer le résultat de manière asynchrone.

La méthode invokeAll() permet de demander l'exécution de plusieurs tâches passées en paramètres sous la forme d'une collection de type List<Callable<T>>.

Exemple (code Java 7) :

```
package fr.jmdoudoux.dej.concurrency;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.annotation.Resource;
import javax.enterprise.concurrent.ManagedExecutorService;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(name = "MaServlet", urlPatterns = {"/MaServlet"})
public class MaServlet extends HttpServlet {

  @Resource(name = "concurrent/monExecutor")
  ManagedExecutorService mes;

  @Override
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {

    response.setContentType("text/html;charset=UTF-8");
    try (PrintWriter out = response.getWriter()) {
      out.println("<!DOCTYPE html>");
      out.println("<html>");
      out.println("<head>");
      out.println("<title>Test JSR 236</title>");
      out.println("</head>");
      out.println("<body>");
      out.println("<p>Tache lancée à " + new Date() + "</p>");

      MaTache maTache = new MaTache();
      Future maTacheFuture = mes.submit(maTache);
      while (!(maTacheFuture.isDone() || maTacheFuture.isCancelled())) {
        try {
          Thread.sleep(500);
        } catch (InterruptedException ex) {
        }
      }

      out.println("<p>Tache terminée à " + new Date() + "</p>");
      out.println("<p>Taches lancées à " + new Date() + "</p>");
      ArrayList<Callable<Date>> taches = new ArrayList<>();
      taches.add(new MaSecondeTache(2000));
      taches.add(new MaSecondeTache(5000));

      List<Future<Date>> res = null;
      try {
        res = mes.invokeAll(taches);
        out.println("<p>Tache 1 terminée à " + res.get(0).get() + "</p>");
        out.println("<p>Tache 2 terminée à " + res.get(1).get() + "</p>");
      } catch (InterruptedException | ExecutionException ex) {
        Logger.getLogger(MaServlet.class.getName()).log(Level.SEVERE, null, ex);
      }
      out.println("</body>");
      out.println("</html>");
    }
  }
}
```

Chaque implémentation du serveur d'applications Java EE propose ses propres possibilités de configuration et les fonctionnalités qui leur sont associées.

L'interface javax.enterprise.concurrent.ManagedScheduledExecutorService hérite des interfaces javax.enterprise.concurrent.ManagedExecutorService et java.util.concurrent.ScheduledExecutorService pour permettre de décaler l'exécution de tâches ou permettre leur exécution de manière répétitive selon un certain délai.

Pour soumettre l'exécution d'une tâche, il est possible d'utiliser une des méthodes submit(), invokeXXX() ou scheduleXXX(). Deux surcharges de la méthode schedule() attendent en paramètres un objet de type javax.enterprise.concurrent.Trigger qui permet de définir les conditions de déclenchement de l'exécution.

Exemple (code Java 7) :

```
@Resource(name="concurrent/monScheduledExecutor")
    ManagedScheduledExecutorService executor;
    // ...
    Future future = executor.schedule(new MaTache(), 5, TimeUnit.SECONDS);
```

Une classe qui implémente une tâche peut optionnellement implémenter l'interface ManagedTask pour permettre de fournir des informations d'exécution et enregistrer un ManagedTaskListener qui recevra et traitera des événements sur le cycle de vie de la tâche.

L'interface ManagedThreadFactory qui hérite de l'interface java.util.concurrent.ThreadFactory définit les fonctionnalités d'une fabrique de threads gérés par le conteneur. Les threads obtenus en invoquant la méthode newThread() doivent implémenter l'interface ManageableThread.

Une instance de type ManagedThreadFactory doit être définie dans le contexte JNDI et une référence vers cette instance doit être déclarée dans le descripteur de déploiement pour permettre un lookup JNDI.

Exemple :

```
<resource-env-ref>
  <resource-env-ref-name>concurrent/maThreadFactory</resource-env-ref-name>
  <resource-env-ref-type>javax.enterprise.concurrent.ManagedThreadFactory
</resource-env-ref-type>
</resource-env-ref>
```

Il est aussi possible de demander l'injection de l'instance et de l'utiliser dans le code.

Exemple (code Java 7) :

```
    @Resource(name = "concurrent/maThreadFactory")
    ManagedThreadFactory threadFactory;
    // ...
    MaTache maTache = new MaTache();
    Thread thread = threadFactory.newThread(maTache);
```

70.8.5. Java Message Service 2.0 (JSR 343)

La précédente version, la 1.1, a été publiée en 2003. Pourtant cette API est largement utilisée.

Les spécifications de la version 2.0 de JMS sont dans la JSR 343.

La version 2.0 de l'API JMS a pour but de simplifier l'utilisation de l'API en réduisant la quantité de code nécessaire à sa mise en oeuvre.

La version 2.0 de JMS définit une nouvelle API et simplifie l'API existante notamment :

- en réduisant la quantité de code superflu

- de faciliter l'injection des ressources et ainsi renforcer l'intégration avec CDI
- en facilitant la gestion des erreurs
- les principaux objets implémentent l'interface AutoClosable qui permet leur utilisation dans une instruction try avec ressources
- une utilisation avec Java EE et Java SE
- les méthodes pour créer une session existent toujours. Une nouvelle méthode pour Java SE attend le mode de session en paramètre. Une nouvelle méthode pour Java EE ne possède aucun paramètre.

Le but de ces évolutions est de maintenir la compatibilité ascendante tout en simplifiant le code à écrire pour utiliser JMS.

L'utilisation de l'API JMS 1.1, pour par exemple envoyer un message, présente plusieurs inconvénients :

- il est nécessaire de créer plusieurs instances d'objets de types différents (connection, Session, Producer/Consumer, TextMessage)
- la connexion doit être fermée explicitement, particulièrement si la méthode createSession() de l'interface Connection est utilisée
- les paramètres de certaines méthodes peuvent être sources d'erreurs
- les exceptions doivent être gérées puisqu'elles sont de type checked.

Exemple (code Java 7) :

```
@Resource(lookup = "java:global/jms/mqConnectionFactory")
ConnectionFactory connectionFactory;
@Resource(lookup = "java:global/jms/maQueue")
Queue maQueue;

public void sendMessage(String contenu) {
  try {
    Connection connection = connectionFactory.createConnection();
    try {
      Session session = connection.createSession(false,Session.AUTO_ACKNOWLEDGE);
      MessageProducer messageProducer = session.createProducer(maQueue);
      TextMessage textMessage = session.createTextMessage(contenu);
      messageProducer.send(textMessage);
    } finally {
      connection.close();
    }
  } catch (JMSException ex) {
    // traitement de l'exception
  }
}
```

Pour conserver la compatibilité, les modifications dans l'API existante ont été limitées.

En plus des méthodes existantes qui sont conservées, la classe javax.jms.Connection possède de nouvelles méthodes :

- creationSession() essentiellement pour Java EE
- creationSession(sessionMode) essentiellement pour Java SE

Les principaux objet de JMS (Connection, Session, MessageProducer, MessageConsumer, QueueBrowser) implémentent AutoCloseable pour permettre leur utilisation avec l'instruction try with ressources.

Exemple (code Java 7) :

```
@Resource(lookup = "jms/mqConnectionFactory")
ConnectionFactory cf;
@Resource(lookup="jms/maQueue")
Destination dest;

public void sendMessage (String contenu) throws JMSException {
  try ( Connection conn = connectionFactory.createConnection();
    Session session = conn.createSession();
    MessageProducer producer = session.createProducer(dest););{
    Message mess = sess.createTextMessage(payload);
    producer.send(mess);
  } catch(JMSException e){
    // traitement de l'exception
  }
}
```

Pour faciliter l'utilisation de JMS, la version 2.0 introduit de nouveaux concepts dans une API simplifiée.

Envoi d'un message avec l'API simplifiée de JMS 2.0

Exemple (code Java 7) :

```
@Resource(lookup = "jms/mqConnectionFactory")
ConnectionFactory connectionFactory;
@Resource(lookup = "jms/maQueue")
Queue maQueue;

public void sendMessage (String contenu) {
  try (JMSContext context = connectionFactory.createContext();){
    context.createProducer().send(maQueue, contenu);
  } catch (JMSRuntimeException ex) {
    // traitement de l'exception
  }
}
```

L'interface JMSContext définit les fonctionnalités d'objets qui encapsulent une connexion et une session et ainsi facilite la création d'objets de l'API JMS.

Pour obtenir une instance, il suffit d'invoquer la méthode createContext() de la classe ConnectionFactory ou de demander au conteneur d'en injecter une instance.

L'interface JMSProducer facilite la définition des propriétés et l'envoi de messages en utilisant un JMSContext. Pour en obtenir une instance, il faut invoquer la méthode createProducer().

Les méthodes de ces classes ne lèvent plus d'exceptions de type checked mais des exceptions de type unchecked.

Exemple (code Java 7) :

```
@Inject
JMSContext context;
@Resource(mappedName="maQueue")
Queue maQueue;

public void sendMessage(String contenu) {
  context.createProducer().send(maQueue, contenu);
}
```

70.8.6. BeanValidation 1.1 (JSR 349)

La JSR 349 spécifie la version 1.1 de l'API BeanValidation qui propose notamment :

- une meilleure intégration avec les spécifications JAX-RS et JAXB
- l'application de contraintes sur les paramètres et la valeur de retour d'une méthode
- une API pour valider les contraintes sur les paramètres et la valeur de retour d'une méthode

Exemple (code Java 7) :

```
public void MaMethode(@NotNull String contenu) {
  context.createProducer().send(maQueue, contenu);
}

@Future
public Date determinerProchaineEcheance() {
  // ...
}
```

70.8.7. Java API for RESTful Web Services 2.0 (JSR 339)

La JSR 339 définit les spécifications de la version 2.0 de l'API JAX-RS.

La version 2.0 propose notamment :

- une API client standardisée
- l'utilisation de l'API Bean Validation pour effectuer des vérifications sur les données obtenues d'une requête HTTP
- le traitement de requêtes de manière asynchrone côtés serveur et client
- l'utilisation de filtres grâce aux intercepteurs
- le support d'Hypermedia qui permet de créer et traiter des liens associés à des ressources

70.8.8. Servlets 3.1 (JSR 340)

La JSR 340 définit les spécifications de la version 3.1 de l'API Servlets.

Cette version apporte plusieurs améliorations à l'API :

- meilleur support des appels IO asynchrones avec NIO en utilisant les listeners ReadListener et WriteListener enregistrés sur les classes ServletInputStream et ServletOutputStream
- simplification des invocations des servlets de manière asynchrone
- utilisation de l'API Java EE concurrency
- sécurité enrichie
- prise en charge de l'option upgrade du protocole HTTP pour permettre la mise en oeuvre des WebSockets

La prise en charge de l'option upgrade du protocole HTTP se fait grâce à la méthode upgrade() de la classe HttpServletRequest qui attend en paramètre le type d'une classe implémentant l'interface HttpUpgradeHandler.

L'interface HttpUpgradeHandler définit deux méthodes :

Méthode	Rôle
void destroy()	Méthode invoquée lorsque la connexion est fermée
void init(WebConnection wc)	Méthode invoquée lors la connexion est prête à utiliser le nouveau protocole

70.8.9. JPA 2.1 (JSR 338)

La version 2.1 de JPA, spécifiée dans la JSR 338, ajoute quelques fonctionnalités manquantes dans les versions précédentes, notamment :

- le support de l'invocation des procédures stockées en utilisant l'annotation @NamedStoredProcedureQuery et l'interface StoredProcedureQuery
- la possibilité de faire des mises à jour et des suppressions par lots en utilisant l'API Criteria respectivement avec les interfaces CriteriaUpdate et CriteriaDelete
- la possibilité de générer le schéma de la base de données en utilisant les métadonnées de mapping et de nouvelles annotations : @Index permet de préciser des index additionnels et @ForeignKey permet de préciser des clés étrangères
- l'utilisation de fonctions définies par l'utilisateur avec le mot clé FUNCTION en JPQL
- la définition de convertisseurs en utilisant l'annotation @Converter
- le support des jointures ouvertes en utilisant le mot clé ON en JPQL
- les PersistenceContext non synchronisés tant que la méthode joinTransaction() n'est pas invoquée

70.8.10. JTA 1.2

La JSR 907 définit les spécifications de la version 1.2 de JTA.

Cette version propose le support de la déclaration de transactions en dehors des EJB grâce à l'utilisation des intercepteurs de CDI.

Cette déclaration se fait en utilisant une nouvelle annotation : javax.transaction.Transactional.

Exemple (code Java 7) :

```
public class MaClasse {

  // ...
  @Transactional
  public void maMethode() {

    // ...
  }
  // ...
}
```

Elle peut avoir en attribut une énumération de type TxType qui par défaut vaut TxType.REQUIRED.

L'énumération TxType définit plusieurs valeurs : REQUIRED, REQUIRED_NEW, MANDATORY, SUPPORTS, NOT_SUPPORTED, NEVER

70.9. La présentation de Java EE 8/Jakarta EE 8

Les spécifications de Java EE 8 sont définies dans la JSR 366. Cette spécification ne définit pas directement d'API mais elle liste celles qui sont incluses dans la plate-forme et comment celles-ci interagissent entre-elles. Elle définit aussi des éléments précis de la plate-forme comme les transactions, la sécurité, l'assemblage, le déploiement, ...

Les spécifications de Java EE 8 furent officiellement diffusées en août 2017. Java EE 8 s'appuie sur Java SE 8. Cette spécification contient 2 nouvelles API, 8 mises à jour majeures et des versions de maintenance (Maintenance Release) de JSR.

La version 8 est une version qui propose :

- Support de Java SE 8 (Date&Time API, repeatable annotations, Completable Future, ...)
- Support de standards web (HTTP/2, SSE, ...)

Java EE 8 inclut deux nouvelles API :

- Java API for JSON Bindong 1.0 (JSR 367)
- Java EE Security API (JSR 375)

Java EE 8 inclut des API ayant une mise à jour majeure :

- Contexts and Dependency Injection for Java EE 2.0 (JSR 365)
- Bean Validation 2.0 (JSR 380)
- Servlet 4.0 (JSR 369)

Java EE 8 inclut des API mises à jour :

- JAX-RS : Java API for Restful Web Services 2.1 (JSR 370)
- JSON-P : Java API for JSON Processing 1.1 (JSR 374)
- JPA 2.2 (JSR 338)
- Web Sockets 1.1 (JSR 338)
- Java Server Faces 2.3 (JSR 372)

- Common Annotations for the Java Platform 1.3 (JSR 250)

L'implémentation de référence est la version 5.0 du serveur d'applications open source Glassfish.

71. JavaMail

Chapitre 71

Niveau : 🟩🟩🟨⬜⬜⬜ Intermédiaire

Le courrier électronique repose sur le concept du client/serveur. Ainsi, l'utilisation d'e-mails requiert deux composants :

- un client de mails (Mail User Agent : MUA) tel que Outlook, Messenger, Eudora, ...
- un serveur de mails (Mail Transport Agent : MTA) tel que SendMail

Les clients de mails s'appuient sur un serveur de mails pour obtenir et envoyer des messages. Les échanges entre clients et serveurs sont normalisés par des protocoles particuliers.

JavaMail est une API qui permet d'utiliser le courrier électronique (e-mail) dans une application écrite en Java (application cliente, applet, servlet, EJB, ...). Son but est d'être facile à utiliser, de fournir une souplesse qui permette de la faire évoluer et de rester le plus indépendant possible des protocoles utilisés.

JavaMail est une extension au JDK qui n'est donc pas fournie avec Java SE. Elle est intégrée à Java EE.

Les classes et interfaces sont regroupées dans quatre packages : javax.mail, javax.mail.event, javax.mail.internet, javax.mail.search.

Il existe deux versions de cette API :

- 1.1.3 : version fournie avec J2EE 1.2
- 1.2 : version courante

Les deux versions fonctionnent avec un JDK dont la version est au moins 1.1.6.

Cette API repose sur une abstraction assez forte de tout système de mails, ce qui lui permet d'ajouter des protocoles non gérés en standard. Pour ces différents protocoles, il faut utiliser une implémentation particulière pour chacun d'eux, implémentation fournie par des tiers. En standard, JavaMail 1.2 implémente les protocoles SMTP, POP3 et IMAP4. JavaMail 1.1.3 ne fournit une implémentation que pour les protocoles SMTP et IMAP : l'implémentation pour le protocole POP3 doit être téléchargée séparément.

Ce chapitre contient plusieurs sections :

- ♦ Le téléchargement et l'installation
- ♦ Les principaux protocoles
- ♦ Les principales classes et interfaces de l'API JavaMail
- ♦ L'envoi d'un e-mail par SMTP
- ♦ La récupération des messages d'un serveur POP3
- ♦ Les fichiers de configuration

71.1. Le téléchargement et l'installation

Pour le J2SE, il est nécessaire de télécharger les fichiers utiles et de les installer.

Pour les deux versions de l'API, il faut télécharger la version correspondante, décompresser le fichier dans un répertoire et ajouter le fichier mail.jar dans le CLASSPATH.

Ensuite il faut aussi installer le framework JAF (Java Activation Framework) : télécharger le fichier, décompresser et ajouter le fichier activation.jar dans le CLASSPATH

Pour pouvoir utiliser le protocole POP3 avec JavaMail 1.1.3, il faut télécharger en plus l'implémentation de ce protocole et inclure le fichier POP3.jar dans le CLASSPATH.

Pour le J2EE 1.2.1, l'API version 1.1.3 est intégrée à la plate-forme. Elle ne contient donc pas l'implémentation pour le protocole POP3. Il faut la télécharger et l'installer en plus comme avec le J2SE.

Pour le J2EE 1.3, il n'y a rien de particulier à faire puisque l'API version 1.2 est intégrée à la plate-forme.

71.2. Les principaux protocoles

71.2.1. Le protocole SMTP

SMTP est l'acronyme de Simple Mail Transport Protocol. Ce protocole défini par la recommandation RFC 821 permet l'envoi de mails vers un serveur de mails qui supporte ce protocole.

71.2.2. Le protocole POP

POP est l'acronyme de Post Office Protocol. Ce protocole défini par la recommandation RFC 1939 permet la réception de mails à partir d'un serveur de mails qui implémente ce protocole. La version courante de ce protocole est 3. C'est un protocole très populaire sur Internet. Il définit une boîte aux lettres unique pour chaque utilisateur. Une fois que le message est reçu par le client, il est effacé du serveur.

71.2.3. Le protocole IMAP

IMAP est l'acronyme de Internet Message Acces Procol. Ce protocole défini par la recommandation RFC 2060 permet aussi la réception de mails à partir d'un serveur de mails qui implémente ce protocole. La version courante de ce protocole est 4. Ce protocole est plus complexe car il apporte des fonctionnalités supplémentaires : plusieurs répertoires par utilisateur, partage de répertoires entre plusieurs utilisateurs, maintien des messages sur le serveur, etc ...

71.2.4. Le protocole NNTP

NNTP est l'acronyme de Network News Transport Protocol. Ce protocole est utilisé par les forums de discussions (news).

71.3. Les principales classes et interfaces de l'API JavaMail

JavaMail propose des classes et interfaces qui encapsulent ou définissent les objets liés à l'utilisation des mails et les protocoles utilisés pour les échanger.

71.3.1. La classe Session

La classe Session encapsule pour un client donné sa connexion avec le serveur de mails. Cette classe encapsule les données liées à la connexion (options de configuration et données d'authentification). C'est à partir de cet objet que toutes les actions concernant les mails sont réalisées.

Les paramètres nécessaires sont fournis dans un objet de type Properties. Un objet de ce type est utilisé pour contenir les variables d'environnements : placer certaines informations dans cet objet permet de partager des données.

Une session peut être unique ou partagée par plusieurs entités.

Exemple :

```
// creation d'une session unique
Session session = Session.getInstance(props,authenticator);
// creation d'une session partagee
Session defaultSession = Session.getDefaultInstance(props,authenticator);
```

Pour obtenir une session, deux paramètres sont attendus :

- un objet Properties qui contient les paramètres d'initialisation. Un tel objet est obligatoire
- un objet Authenticator optionnel qui permet d'authentifier l'utilisateur auprès du serveur de mails

La méthode setDebug() qui attend en paramètre un booléen est très pratique pour debugger car avec le paramètre true, elle affiche des informations lors de l'utilisation de la session notamment le détail des commandes envoyées au serveur de mails.

71.3.2. Les classes Address, InternetAddress et NewsAddress

La classe Address est une classe abstraite dont héritent toutes les classes qui encapsulent une adresse dans un message.

Deux classes filles sont actuellement définies :

- InternetAddress
- NewsAddress

Le classe InternetAddress encapsule une adresse email respectant le format de la RFC 822. Elle contient deux champs : address qui contient l'adresse e-mail et personal qui contient le nom de la personne. La classe possède des constructeurs, des getters et des setters pour utiliser ces attributs.

Le plus simple pour créer un objet InternetAddress est d'appeler le constructeur en lui passant en paramètre une chaîne de caractères contenant l'adresse e-mail.

Exemple :

```
InternetAddress vInternetAddresses = new InternetAddress("moi@chez-moi.fr");
```

Un second constructeur permet de préciser l'adresse e-mail et un nom en clair.

La méthode getLocalAddress(Session) permet de déterminer si possible l'objet InternetAddress encapsulant l'adresse e-mail de l'utilisateur courant, sinon elle renvoie null.

La méthode parse(String) permet de créer un tableau d'objets InternetAddress à partir d'une chaîne contenant les adresses e-mail séparées par des virgules.

Un objet InternetAddress est nécessaire pour chaque émetteur et destinataire du mail. L'API ne vérifie pas l'existence des adresses fournies. C'est le serveur de mails qui vérifiera les destinataires et éventuellement les émetteurs selon son paramétrage.

La classe NewsAddress encapsule une adresse news (forum de discussion) respectant le format RFC1036. Elle contient deux champs : host qui contient le nom du serveur et newsgroup celui du nom du forum.

La classe possède des constructeurs, des getters et des setters pour utiliser ces attributs.

71.3.3. L'interface Part

Cette interface définit un certain nombre d'attributs commun à la plupart des systèmes de mails et un contenu.

Le contenu peut être renvoyé sous trois formes : DataHandler, InputStream et Object.

Cette interface définit plusieurs méthodes principalement des getters et des setters dont les principaux sont :

Méthode	Rôle
int getSize()	Renvoyer la taille du contenu ou -1 si elle ne peut être déterminée
int getLineCount()	Renvoyer le nombre de lignes du contenu ou -1 s'il ne peut être déterminé
String getContentType()	Renvoyer le type du contenu sinon null
String getDescription()	Renvoyer la description
void setDescription(String)	Mettre à jour la description
InputStream getInputStream()	Renvoyer le contenu sous la forme d'un flux
DataHandler getDataHandler()	Renvoyer le contenu sous la forme d'un objet DataHandler
Object getContent()	Renvoyer le contenu sous la forme d'un objet. Un cast est nécessaire selon le type du contenu.
void setText(String)	Mettre à jour le contenu sous forme d'une chaîne de caractères fournie en paramètre

71.3.4. La classe Message

La classe abstraite Message encapsule un Message. Le message est composé de deux parties :

- une en-tête qui contient des attributs
- un corps qui contient les données à envoyer

Pour la plupart de ces données, la classe Message implémente l'interface Part qui encapsule les attributs nécessaires à la distribution du message (auteur, destinataire, sujet ...) et le corps du message.

Le contenu du message est stocké sous forme d'octets. Pour y accéder, il faut utiliser un objet du JavaBean Activation Framework (JAF) : DataHandler. Ceci permet une séparation des données nécessaires à la transmission et du contenu du message qui peut ainsi prendre n'importe quel format. La classe Message ne connaît pas directement le type du contenu du corps du message.

JavaMail fournit en standard une classe fille nommée MimeMessage qui implémente la recommandation RFC 822 pour les messages possédant un type MIME.

Il y a deux façons d'obtenir un objet de type Message : instancier une classe fille pour créer un nouveau message ou utiliser un objet de type Folder pour obtenir un message existant.

La classe Message définit deux constructeurs en plus du constructeur par défaut :

Constructeur	Rôle
Message(session)	Créer un nouveau message

Message(Folder, int)	Créer un message à partir d'un message existant

La classe MimeMessage est la seule classe fille qui hérite de la classe Message. Elle dispose de plusieurs constructeurs.

Exemple :

```
      MimeMessage message = new MimeMessage(session);
```

Elle possède de nombreuses méthodes pour initialiser les données du message :

Méthode	Rôle
void addFrom(Address[])	Ajouter des émetteurs au message
void addRecipient(RecepientType, Address[])	Ajouter des destinataires à un type (direct, en copie ou en copie cachée)
Flags getFlags()	Renvoyer les états du message
Adress[] getFrom()	Renvoyer les émetteurs
int getLineCount()	Renvoyer le nombre de lignes du message
Address[] getRecipients(RecepientType)	Renvoyer les destinataires du type fourni en paramètre
Address getReplyTo()	Renvoyer l'adresse e-mail pour la réponse
int getSize()	Renvoyer la taille du message
String getSubject()	Renvoyer le sujet
Message reply(boolean)	Créer un message pour la réponse : le booléen indique si la réponse ne doit être faite qu'à l'émetteur
void setContent(Object, String)	Mettre à jour le contenu du message en précisant son type MIME
void setFrom(Address)	Mettre à jour l'émetteur
void setRecipients(RecepientsType, Address[])	Mettre à jour les destinataires d'un type
void setSendDate(Date)	Mettre à jour la date d'envoi
void setText(String)	Mettre à jour le contenu du message avec le type MIME « text/plain »
void setReply(Address)	Mettre à jour le destinataire de la réponse
void writeTo(OutputStream)	Envoyer le message au format RFC 822 dans un flux. Très pratique pour visualiser le message sur la console en passant en paramètre (System.out)

La méthode addRecipient() permet d'ajouter un destinataire et le type d'envoi.

Le type d'envoi est précisé grâce à une constante pour chaque type :

- destinataire direct : Message.RecipientType.TO
- copie conforme : Message.RecipientType.CC
- copie cachée : Message.RecipientType.BCC

La méthode setText() permet de facilement mettre une chaîne de caractères dans le corps du message avec un type MIME « text/plain ». Pour envoyer un message dans un format différent, par exemple HTML, il faut utiliser la méthode setContent() qui attend en paramètres un objet et une chaîne de caractères indiquant son type MIME.

Exemple :

```
    String texte = "<H1>bonjour</H1><a href=\"mailto:moi@moi.fr\">mail</a>";
    message.setContent(texte, "text/html");
```

Il est possible de joindre avec le mail des ressources sous forme de pièces jointes (attachments). Pour cela, il faut :

- instancier un objet de type MimeMessage
- renseigner les éléments qui composent l'en-tête : émetteur, destinataire, sujet ...
- instancier un objet de type MimeMultiPart
- instancier un objet de type MimeBodyPart et alimenter le contenu de l'élément
- ajouter cet objet à l'objet MimeMultiPart grâce à la méthode addBodyPart()
- répéter l'instanciation et l'alimentation pour chaque ressource à ajouter
- utiliser la méthode setContent() du message en passant en paramètre l'objet MimeMultiPart pour associer le message et les pièces jointes au mail

Exemple :

```
    Multipart multipart = new MimeMultipart();

    // creation partie principale du message
    BodyPart messageBodyPart = new MimeBodyPart();
    messageBodyPart.setText("Test");
    multipart.addBodyPart(messageBodyPart);

    // creation et ajout de la piece jointe
    messageBodyPart = new MimeBodyPart();
    DataSource source = new FileDataSource("image.gif");
    messageBodyPart.setDataHandler(new DataHandler(source));
    messageBodyPart.setFileName("image.gif");
    multipart.addBodyPart(messageBodyPart);

    // ajout des éléments au mail
    message.setContent(multipart);
```

71.3.5. Les classes Flags et Flag

La classe Flags encapsule un ensemble d'états pour un message.

Il existe deux types d'états : les états prédéfinis (System Flag) et les états particuliers définis par l'utilisateur (User Defined Flag)

Un état prédéfini est encapsulé par la classe Internet Flags.Flag. Cette classe définit plusieurs états statiques :

Etat	Rôle
Flags.Flag.ANSWERED	
Flags.Flag.DELETED	Le message est marqué pour la suppression
Flags.Flag.DRAFT	Le message est un brouillon
Flags.Flag.FLAGGED	Le message est marqué dans un état qui n'a pas de définition particulière
Flags.Flag.RECENT	Le message est arrivé récemment. Le client ne peut pas modifier cet état
Flags.Flag.SEEN	Le message a été visualisé : positionné à l'ouverture du message
Flags.Flag.USER	Le client a la possibilité d'ajouter des états particuliers

Tous ces états ne sont pas obligatoirement supportés par le serveur.

La classe Message possède plusieurs méthodes pour gérer les états d'un message. La méthode getFlags() renvoie un objet de type Flags qui contient les états du message. Les méthodes setFlag(Flag, boolean) permettent d'ajouter un état du message. La méthode contains(Flag) vérifie si l'état fourni en paramètre est positionné pour le message.

La classe Flags possède plusieurs méthodes pour gérer les états. Les principales sont :

Méthode	Rôle
void add(Flags.Flag)	Permet d'ajouter un état
void add(Flags)	Permet d'ajouter un ensemble d'états
void remove(Flags.Flag)	Permet d'enlever un état
void remove(Flags)	Permet d'enlever un ensemble d'états
boolean contains(Flags.Flag)	Permet de savoir si un état est positionné

71.3.6. La classe Transport

La classe Transport se charge de réaliser l'envoi du message avec le protocole adéquat. C'est une classe abstraite qui contient la méthode static send() pour envoyer un mail.

Il est possible d'obtenir un objet Transport dédié au protocole particulier utilisé par la session en utilisant la méthode getTransport() d'un objet Session. Dans ce cas, il faut :

1. établir la connexion en utilisant la méthode connect() avec le nom du serveur, le nom de l'utilisateur et son mot de passe
2. envoyer le message en utilisant la méthode sendMessage() avec le message et les destinataires. La méthode getAllRecipients() de la classe Message permet d'obtenir ceux contenus dans le message.
3. fermer la connexion en utilisant la méthode close()

Il est préférable d'utiliser une instance de Transport tel qu'expliqué ci-dessus lorsqu'il y a plusieurs mails à envoyer car on peut maintenir la connexion avec le serveur ouverte pendant les envois.

La méthode static send() ouvre et ferme la connexion à chacun de ses appels.

71.3.7. La classe Store

La classe abstraite Store représente un système de stockage de messages. Pour obtenir une instance de cette classe, il faut utiliser la méthode getStore() d'un objet de type Session en lui donnant comme paramètre le protocole utilisé.

Pour pouvoir dialoguer avec le serveur de mails, il faut appeler la méthode connect() en lui précisant le nom du serveur, le nom d'utilisateur et le mot de passe de l'utilisateur.

La méthode close() permet de libérer la connexion avec le serveur.

71.3.8. La classe Folder

La classe abstraite Folder représente un répertoire dans lequel les messages sont stockés. Pour obtenir une instance de cette classe, il faut utiliser la méthode getFolder() d'un objet de type Store en lui précisant le nom du répertoire.

Avec le protocole POP3 qui ne gère qu'un seul répertoire, le seul possible est « INBOX ».

Pour pouvoir être utilisé, il faut appeler la méthode open() de la classe Folder en lui précisant le mode d'utilisation : READ_ONLY ou READ_WRITE.

Pour obtenir les messages contenus dans le répertoire, il faut appeler la méthode getMessages(). Cette méthode renvoie un tableau de Message qui peut être null si aucun message n'est renvoyé.

Une fois les opérations terminées, il faut fermer le répertoire en utilisant la méthode close().

71.3.9. Les propriétés d'environnement

JavaMail utilise des propriétés d'environnement pour recevoir certains paramètres de configuration. Ils sont stockés dans un objet de type Properties.

L'objet Properties peut contenir un certain nombre de propriétés qui possèdent des valeurs par défaut :

Propriété	Rôle	Valeur par défaut
mail.store.protocol	Protocole de stockage du message	le premier protocole concerné dans le fichier de configuration
mail.transport.protocol	Protocole de transport par défaut	le premier protocole concerné dans le fichier de configuration
mail.host	Serveur de mails par défaut	localhost
mail.user	Nom de l'utilisateur pour se connecter au serveur de mails	user.name
mail.protocol.host	Serveur de mails pour un protocole dédié	mail.host
mail.protocol.user	Nom de l'utilisateur pour se connecter au serveur de mails pour un protocole dédié	
mail.from	adresse par défaut de l'expéditeur	user.name@host
mail.debug	mode de débogage par défaut	

Attention : l'utilisation de JavaMail dans une applet implique de fournir explicitement toutes les valeurs des propriétés utiles car une applet ne peut pas accéder à ces valeurs.

L'usage de certains serveurs de mails nécessite l'utilisation d'autres propriétés.

71.3.10. La classe Authenticator

Authenticator est une classe abstraite qui propose des méthodes de base pour permettre d'authentifier un utilisateur. Pour l'utiliser, il faut créer une classe fille qui se chargera de collecter les informations. Plusieurs méthodes sont à redéfinir selon les besoins :

Méthode	Rôle
String getDefaultUserName()	
PasswordAuthentication getPasswordAuthentication()	
int getRequestingPort()	
String getRequestingPort()	
String getRequestingProtocol()	
InetAddress getRequestingSite()	

Par défaut, la méthode getPasswordAuthentication() de la classe Authentication renvoie null. Cette méthode renvoie un objet PasswordAuthentication à partir d'une source de données (boîte de dialogue pour saisie, base de données, ...).

Une instance d'une classe fille de la classe Authenticator peut être fournie à la session. L'appel à Authenticator sera fait selon les besoins par la session.

71.4. L'envoi d'un e-mail par SMTP

Pour envoyer un e-mail via SMTP, il faut suivre les principales étapes suivantes :

- Positionner les variables d'environnement nécessaires
- Instancier un objet Session
- Instancier un objet Message
- Mettre à jour les attributs utiles du message
- Appeler la méthode send() de la classe Transport

Exemple :

```
import javax.mail.internet.*;
import javax.mail.*;
import java.util.*;

/**
 * Classe permettant d'envoyer un mail.
 */
public class TestMail {
  private final static String MAILER_VERSION = "Java";
  public static boolean envoyerMailSMTP(String serveur, boolean debug) {
    boolean result = false;
    try {
      Properties prop = System.getProperties();
      prop.put("mail.smtp.host", serveur);
      Session session = Session.getDefaultInstance(prop,null);
      Message message = new MimeMessage(session);
      message.setFrom(new InternetAddress("moi@chez-moi.fr"));
      InternetAddress[] internetAddresses = new InternetAddress[1];
      internetAddresses[0] = new InternetAddress("moi@chez-moifr");
      message.setRecipients(Message.RecipientType.TO,internetAddresses);
      message.setSubject("Test");
      message.setText("test mail");
      message.setHeader("X-Mailer", MAILER_VERSION);
      message.setSentDate(new Date());
      session.setDebug(debug);
      Transport.send(message);
      result = true;
    } catch (AddressException e) {
      e.printStackTrace();
    } catch (MessagingException e) {
      e.printStackTrace();
    }
    return result;
  }

  public static void main(String[] args) {
    TestMail.envoyerMailSMTP("10.10.50.8",true);
  }
}
```

javac -classpath activation.jar;mail.jar;smtp.jar TestMail.java

java -classpath .;activation.jar;mail.jar;smtp.jar TestMail

71.5. La récupération des messages d'un serveur POP3

Cette section sera développée dans une version future de ce document

71.6. Les fichiers de configuration

Ces fichiers permettent d'enregistrer des implémentations de protocoles supplémentaires et des valeurs par défaut. Il existe 4 fichiers répartis en deux catégories :

- javamail.providers et javamail.default.providers
- javamail.address.map et javamail.default.address.map

JavaMail recherche les informations contenues dans ces fichiers dans l'ordre suivant :

1. $JAVA_HOME/lib

2. META-INF/javamail.xxx dans le fichier jar de l'application

3. MATA-INF/javamail.default.xxx dans le fichier jar de javamail

Il est aussi possible d'utiliser son propre fichier sans faire de modification dans le fichier jar de JavaMail. Cette utilisation peut se faire sur le poste client ou dans le fichier jar de l'application, ce qui offre une grande souplesse.

71.6.1. Les fichiers javamail.providers et javamail.default.providers

Ce sont deux fichiers au format texte qui contiennent la liste et la configuration des protocoles pour lesquels le système dispose d'une implémentation. L'application peut ainsi rechercher la liste des protocoles utilisables.

Chaque protocole est défini en utilisant des attributs avec la forme nom=valeur suivi d'un point virgule. Cinq attributs sont définis (leurs noms doivent être en minuscules) :

Nom de l'attribut	Rôle	Présence
protocol	nom du protocole	obligatoire
type	type protocole : « store » ou « transport »	obligatoire
class	nom de la classe contenant l'implémentation du protocole	obligatoire
vendor	nom du fournisseur	optionnelle
version	numéro de version	optionnelle

Exemple : le contenu du fichier META-INF/javamail.default.providers

```
# JavaMail IMAP provider Sun Microsystems, Inc
protocol=imap; type=store; class=com.sun.mail.imap.IMAPStore; vendor=Sun Microsystems, Inc;
# JavaMail SMTP provider Sun Microsystems, Inc
protocol=smtp; type=transport; class=com.sun.mail.smtp.SMTPTransport; vendor=Sun Microsystems,
 Inc;
# JavaMail POP3 provider Sun Microsystems, Inc
protocol=pop3; type=store; class=com.sun.mail.pop3.POP3Store; vendor=Sun Microsystems, Inc;
```

71.6.2. Les fichiers javamail.address.map et javamail.default.address.map

Ce sont deux fichiers au format texte qui permettent d'associer un type de transport avec un protocole. Cette association se fait sous la forme nom=valeur suivie d'un point virgule.

Exemple : le contenu du fichier META-INF/javamail.default.address.map

```
rfc822=smtp
```

72. JMS (Java Message Service)

Chapitre 72

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

JMS, acronyme de Java Message Service, est une API pour permettre un dialogue standard entre des applications ou des composants grâce àdes brokers de messages ou MOM (Message-Oriented Middleware). Elle permet donc d'utiliser des services de messaging dans des applications Java comme le fait l'API JDBC pour les bases de données.

La page officielle de JMS est à l'URL : https://www.oracle.com/java/technologies/java-message-service.html.

Ce chapitre contient plusieurs sections :

- La présentation de JMS
- Les services de messages
- Le package javax.jms
- L'utilisation du mode point à point (queue)
- L'utilisation du mode publication/abonnement (publish/subscribe)
- La gestion des erreurs
- JMS 1.1
- Les ressources relatives à JMS

72.1. La présentation de JMS

JMS a été intégré à la plate-forme J2EE à partir de la version 1.3. Il n'existe pas d'implémentation officielle de cette API avant cette version. JMS est utilisable avec les versions antérieures mais elle oblige à utiliser un outil externe qui implémente l'API.

Chaque fournisseur (provider) doit fournir une implémentation de ses spécifications. Il existe un certain nombre d'outils qui implémentent JMS dont la majorité sont des produits commerciaux.

Dans la version 1.3 du J2EE, JMS peut être utilisé dans un composant web ou un EJB, un type d'EJB particulier a été ajouté pour traiter les messages et des échanges JMS peuvent être intégrés dans une transaction gérée avec JTA (Java Transaction API).

JMS définit plusieurs entités :

- Un provider JMS : outil qui implémente l'API JMS pour échanger les messages : ce sont les brokers de messages
- Un client JMS : composant écrit en Java qui utilise JMS pour émettre et/ou recevoir des messages.
- Un message : données échangées entre les composants

Différents objets utilisés avec JMS sont généralement stockés dans l'annuaire JNDI du serveur d'applications ou du provider du MOM :

- La fabrique de connexions (ConnectionFactory)
- Les destinations à utiliser (Queue et Topic)

JMS définit deux modes pour la diffusion des messages :

- Point à point (Point to point) : dans ce mode un message est envoyé par un producteur et est reçu par un unique consommateur. Le support utilisé pour la mise en oeuvre de ce mode est la file (queue). Le message émis est stocké dans la file jusqu'à ce que le consommateur le lise et envoie une notification de réception du message. A ce moment là le message est supprimé de la file. Le message a généralement une date d'expiration.
- Publication / souscription (publish/subscribe) : dans ce mode un message est envoyé par un producteur et est reçu par un ou plusieurs consommateurs. Le support utilisé pour la mise en oeuvre de ce mode est le sujet (topic). Chaque consommateur doit s'abonner à un sujet (souscription). Seuls les messages émis à partir de cet abonnement sont accessibles par le consommateur.

Dans la version 1.0 de JMS, ces modes utilisent des interfaces distinctes.
Dans la version 1.1 de JMS, ces interfaces sont toujours utilisables mais il est aussi possible d'utiliser des interfaces communes à ces modes ce qui les rend interchangeables.

Les messages sont asynchrones mais JMS définit deux modes pour consommer un message :

- Mode synchrone : ce mode nécessite l'appel de la méthode receive() ou d'une de ses surcharges. Dans ce cas, l'application est arrêtée jusqu'à l'arrivée du message. Une version surchargée de cette méthode permet de rendre la main après un certain timeout.
- Mode asynchrone : il faut définir un listener qui va lancer un thread attendant les messages et exécutant une méthode à leur arrivée.

JMS propose un support pour différents types de messages : texte brut, flux d'octets, objets Java sérialisés, ...

72.2. Les services de messages

Les brokers de messages ou MOM (Message-Oriented Middleware) permettent d'assurer l'échange de messages entre deux composants nommés clients. Ces échanges peuvent se faire dans un contexte interne (pour l'EAI) ou un contexte externe (pour le B2B).

Les deux clients n'échangent pas directement des messages : un client envoie un message et le client destinataire doit demander la réception du message. Le transfert du message et sa persistance sont assurés par le broker.

Les échanges de message sont :

- asynchrones :
- fiables : les messages ne sont délivrés qu'une et une seule fois

Les MOM représentent le seul moyen d'effectuer un échange de messages asynchrones. Ils peuvent aussi être très pratiques pour l'échange synchrone de messages plutôt que d'utiliser d'autres mécanismes plus compliqués à mettre en oeuvre (sockets, RMI, CORBA ...).

Les brokers de messages peuvent fonctionner selon deux modes :

- le mode point à point (point to point)
- le mode publication/abonnement (publish/subscribe)

Le mode point à point (point to point) repose sur le concept de files d'attente (queues). Le message est stocké dans une file d'attente puis il est lu dans cette file ou dans une autre. Le transfert du message d'une file à l'autre est réalisé par le broker de message.

Chaque message est envoyé dans une seule file d'attente. Il y reste jusqu'à ce qu'il soit consommé par un client et un seul. Le client peut le consommer ultérieurement : la persistance est assurée par le broker de message.

Le mode publication/abonnement repose sur le concept de sujets (Topics). Plusieurs clients peuvent envoyer des messages dans ce topic. Le broker de message assure l'acheminement de ce message à chaque client qui se sera préalablement abonné à ce topic. Le message possède donc potentiellement plusieurs destinataires. L'émetteur du message ne connaît pas les destinataires qui se sont abonnés.

72.3. Le package javax.jms

Ce package et ses sous-packages contiennent plusieurs interfaces qui définissent l'API.

- Connection
- Session
- Message
- MessageProducer
- MessageListener

72.3.1. La fabrique de connexions

Un objet de type Factory produit une connexion permettant l'accès au broker de messages. Il faut fournir plusieurs paramètres à l'objet de type Factory.

Il existe deux types de fabriques : QueueConnectionFactory et TopicConnectionFactory selon le type d'échange que l'on fait. Ce sont des interfaces que le broker de messages doit implémenter pour fournir des objets.

Pour obtenir un objet de ce type, il faut soit instancier directement un tel objet soit faire appel à JNDI pour l'obtenir. Cette dernière solution est préférable car elle est plus portable.

La fabrique de type ConnectionFactory permet d'obtenir une instance de l'interface Connection. Cette instance est du type de l'implémentation fournie par le provider, ce qui permet de proposer une manière unique d'obtenir une instance de chaque implémentation.

Chaque provider fournit sa propre solution pour gérer les objets contenus dans l'annuaire JNDI.

72.3.2. L'interface Connection

Cette interface définit des méthodes pour la connexion au broker de messages.

Cette connexion doit être établie en fonction du mode utilisé :

- l'interface QueueConnection pour le mode point à point
- l'interface TopicConnection pour le mode publication/abonnement

Pour obtenir l'un ou l'autre, il faut utiliser un objet de type Factory correspondant au type QueueConnectionFactory ou TopicConnectionFactory avec la méthode correspondante : createQueueConnection() ou createTopicConnection().

La classe qui implémente cette interface se charge du dialogue avec le broker de messages.

La méthode start() permet de démarrer la connexion.

Exemple :

```
          connection.start();
```

La méthode stop() permet de suspendre temporairement la connexion.

La méthode close() permet de fermer la connexion.

Remarque : il est important de fermer explicitement la connexion lorsqu'elle devient inutile en utilisant la méthode close().

72.3.3. L'interface Session

Elle représente un contexte transactionnel de réception et d'émission pour une connexion donnée.

C'est d'ailleurs à partir d'un objet de type Connection que l'on crée une ou plusieurs sessions.

La session est monothread : si l'application utilise plusieurs threads qui échangent des messages, il faut définir une session pour chaque thread.

C'est à partir d'un objet session que l'on crée des messages et des objets à envoyer et à recevoir.

Comme pour la connexion, la création d'un objet de type Session dépend du mode de fonctionnement. L'interface Session possède deux interfaces filles :

- l'interface QueueSession pour le mode point à point
- l'interface TopicSession pour le mode publication/abonnement

Pour obtenir l'un ou l'autre, il faut utiliser un objet Connection correspondant de type QueueConnection ou TopicConnection avec sa méthode associée : createQueueSession() ou createTopicSession().

Ces deux méthodes demandent deux paramètres : un booléen qui indique si la session gère une transaction et une constante qui précise le mode d'accusé de réception des messages.

Les messages sont considérés comme traités par le MOM à la réception d'un accusé de réception. Celui-ci est fourni au MOM selon le mode utilisé. Il existe trois modes d'accusés de réception (trois constantes sont définies dans l'interface Session) :

- AUTO_ACKNOWLEDGE : l'accusé de réception est automatique, le MOM reçoit l'accusé de réception à la réception du message que ce dernier soit traité ou non par l'application
- CLIENT_ACKNOWLEDGE : le MOM reçoit explicitement l'acquittement de la part de l'application, c'est le client qui envoie l'accusé grâce à l'appel de la méthode acknowledge() du message
- DUPS_OK_ACKNOWLEDGE : ce mode permet d'indiquer au MOM qu'il peut envoyer plusieurs fois le message à une même destination. Ce mode peut améliorer les performances de certains MOM notamment avec un nombre de messages très important.

L'interface Session définit plusieurs méthodes dont les principales sont :

Méthode	Rôle
void close()	Fermer la session
void commit()	Valider la transaction
XXX createXXX()	Créer un Message dont le type est XXX
void rollback()	Invalider la transaction

72.3.4. Les messages

Les messages sont encapsulés dans un objet de type javax.jms.Message : ils doivent obligatoirement implémenter l'interface Message ou l'une de ses sous-classes.

Un message est constitué de trois parties :

- L'en-tête (header) : contient des données techniques
- Les propriétés (properties) : contient des données fonctionnelles
- Le corps du message (body) : contient les données du message

L'interface Session propose plusieurs méthodes createXXXMessage() pour créer des messages contenant des données au format XXX.

Il existe aussi pour chaque format des interfaces filles de l'interface Message :

- BytesMessage : message composé d'octets
- MapMessage : message composé de paires clé/valeur
- ObjectMessage : message contenant un objet sérialisé
- StreamMessage : message issu d'un flux
- TextMessage : message contenant du texte

72.3.4.1. L'en-tête

Cette partie du message est constituée d'un certain nombre de champs prédéfinis qui contiennent des données pour identifier et acheminer le message.

La plupart de ces données sont renseignées lors de l'appel à la méthode send() ou publish().

L'en-tête contient des données standardisées dont le nom commence par JMS (JMSDestination, JMSDeliveryMode, JMSExpiration, JMSPriority, JMSMessageID, JMSTimestamp, JMSRedelivered, JMSCorrelationID, JMSReplyTo et JMSType)

Les champs les plus importants sont :

Nom	Rôle
JMSMessageID	Identifiant unique du message
JMSDestination	File d'attente ou topic destinataire du message
JMSCorrelationID	Utilisé pour synchroniser de façon applicative deux messages de la forme requête/réponse. Dans ce cas, dans le message réponse, ce champ contient le messageID du message requête

Les propriétés contiennent des données fonctionnelles sous la forme de paires clé/valeur. Certaines propriétés peuvent aussi être positionnées par l'implémentation du provider. Leurs noms commencent par JMS_ suivis du nom du provider.

72.3.4.2. Les propriétés

Ce sont des champs supplémentaires : certains sont définis par JMS mais il est possible d'ajouter ses propres champs.

Cette partie du message est optionnelle. Les propriétés permettent de définir des champs qui seront utilisées pour fournir des données supplémentaires ou pour filtrer le message.

72.3.4.3. Le corps du message

Il contient les données du message et est formaté selon son type.

Cette partie du message est optionnelle. Les messages peuvent être de plusieurs types, définis dans les interfaces suivantes :

type	Interface	Rôle
bytes	BytesMessage	échange d'octets
texte	TextMessage	échange de données texte (XML par exemple)
object	ObjectMessage	échange d'objets Java qui doivent être sérialisables
Map	MapMessage	échange de données sous la forme clé/valeur. La clé doit être une chaîne de caractères et la valeur de type primitive

Stream	StreamMessage	échange de données en provenance d'un flux

Il est possible de définir son propre type qui doit obligatoirement implémenter l'interface Message.

C'est un objet de type Session qui contient les méthodes nécessaires à la création d'un message selon son type.

Lors de la réception d'un message, celui-ci est toujours de type Message : il faut effectuer un transtypage en fonction de son type en utilisant l'opérateur instanceof. A ce moment, il faut utiliser le getter correspondant pour obtenir les données.

Exemple :

```
Message message = ...

if (message instanceof TextMessage) {
   TextMessage textMessage = (TextMessage) message;
   System.out.println("message: " + textMessage.getText());
}
```

72.3.5. L'envoi de messages

L'interface MessageProducer est la super-interface des interfaces qui définissent des méthodes pour l'envoi de messages.

Il existe deux interfaces filles selon le mode de fonctionnement pour envoyer un message : QueueSender et TopicPublisher.

Ces objets sont créés à partir d'un objet représentant la session :

- la méthode createSender() pour obtenir un objet de type QueueSender
- la méthode createPublisher() pour obtenir un objet de type TopicPublisher

Ces objets peuvent être liés à une entité physique par exemple une file d'attente particulière pour un objet de type QueueSender. Si ce n'est pas le cas, cette entité devra être précisée lors de l'envoi du message en utilisant une version surchargée de la méthode chargée de l'émission du message.

72.3.6. La réception de messages

L'interface MessageConsumer est la super-interface des interfaces qui définissent des méthodes pour la réception de messages.

Il existe des interfaces selon le mode de fonctionnement pour recevoir un message QueueReceiver et TopicSubscriber.

La réception d'un message peut se faire avec deux modes :

- synchrone : dans ce cas, l'attente d'un message bloque l'exécution du reste du code
- asynchrone : dans ce cas, un thread est lancé qui attend le message et appelle une méthode (callback) à son arrivée. L'exécution de l'application n'est pas bloquée.

L'interface MessageConsumer définit plusieurs méthodes dont les principales sont :

Méthode	Rôle
close()	Fermer l'objet qui reçoit les messages pour le rendre inactif
Message receive()	Attendre et retourner le message à son arrivée
Message receive(long)	Attendre durant le nombre de millisecondes précisé en paramètre et renvoyer le message s'il arrive durant ce laps de temps
Message receiveNoWait()	Retourner le prochain message s'il y en a un d'immédiatement disponible

setMessageListener(MessageListener)	Associer un Listener pour traiter les messages de façon asynchrone

Pour obtenir un objet qui implémente l'interface QueueReceiver, il faut utiliser la méthode createReceiver() d'un objet de type QueueSession.

Pour obtenir un objet qui implémente l'interface TopicSubscriber, il faut utiliser la méthode createSubscriber() d'un objet de type TopicSession.

72.4. L'utilisation du mode point à point (queue)

72.4.1. La création d'une fabrique de connexions : QueueConnectionFactory

L'objet QueueConnectionFactory permet d'obtenir une QueueConnection pour se connecter au broker de messages.

Pour obtenir un objet de ce type, il faut soit instancier directement un tel objet soit faire appel à JNDI pour l'obtenir.

Exemple : avec MQSeries

```
String qManager = ...
String hostName = ...
String channel = ...

MQQueueConnectionFactory factory = new MQQueueConnectionFactory();
factory.setQueueManager(qManager);
factory.setHostName(hostName);
factory.setChannel(channel);
factory.setTransportType(JMSC.MQJMS_TP_CLIENT_MQ_TCPIP);
```

Il est cependant préférable de faire appel à JNDI pour obtenir un objet de type QueueConnectionFactory. Une instance de cet objet est stockée dans un annuaire par le broker et il suffit de se connecter à cet annuaire avec JNDI pour obtenir l'instance de la fabrique.

72.4.2. L'interface QueueConnection

Cette interface hérite de l'interface Connection.

Pour obtenir un objet qui implémente cette interface, il faut utiliser un objet de type QueueConnectionFactory avec la méthode correspondante : createQueueConnection().

Exemple :

```
            QueueConnection connection = factory.createQueueConnection();
            connection.start();
```

L'interface QueueConnection définit plusieurs méthodes dont la principale est :

Méthode	Rôle
QueueSession createQueueSession(boolean, int)	Renvoyer un objet qui définit la session. Le booléen précise si la session gère une transaction. L'entier précise le mode d'accusé de réception.

72.4.3. La session : l'interface QueueSession

Elle hérite de l'interface Session.

Pour obtenir un objet qui implémente cette interface, il faut utiliser la méthode createQueueSession() d'un objet de type QueueConnection.

Exemple :

```
QueueSession session = connection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
```

L'interface QueueSession définit plusieurs méthodes dont les principales sont :

Méthode	Rôle
QueueReceiver createQueueReceiver(Queue)	Renvoyer un objet qui définit une file d'attente de réception
QueueSender createQueueSender(Queue)	Renvoyer un objet qui définit une file d'attente d'émission

72.4.4. L'interface Queue

Un objet qui implémente cette interface encapsule une file d'attente particulière.

Pour obtenir un objet qui implémente cette interface, il faut utiliser la méthode createQueue() d'un objet de type QueueSession.

Exemple avec MQseries :

```
   Queue fileEnvoi = session.createQueue(
          "queue:///file.out?expiry=0&persistence=1&targetClient=1");
```

72.4.5. La création d'un message

Pour créer un message, il faut utiliser une méthode createXXXMessage() d'un objet QueueSession où XXX représente le type du message.

Exemple :

```
            String message = "bonjour";
            TextMessage textMessage = session.createTextMessage();
            textMessage.setText(message);
```

72.4.6. L'envoi de messages : l'interface QueueSender

Cette interface hérite de l'interface MessageProducer.

Pour obtenir un objet qui implémente cette interface, il faut utiliser la méthode createQueueSender() d'un objet de type QueueSession.

Exemple :

```
           QueueSender queueSender = session.createSender(fileEnvoi);
```

Il est possible de fournir un objet de type Queue qui représente la file d'attente : dans ce cas, l'objet QueueSender est lié à cette file d'attente. Si l'on ne précise pas de file d'attente (null fourni en paramètre), il faudra obligatoirement utiliser une version surchargée de la méthode send() lors de l'envoi pour indiquer celle à utiliser.

Avec un objet de type QueueSender, la méthode send() permet l'envoi d'un message dans la file d'attente. Cette méthode possède plusieurs surcharges :

Méthode	Rôle
void send(Message)	Envoyer le message dans la file d'attente définit dans l'objet de type QueueSender
void send(Queue, Message)	Envoyer le message dans la file d'attente fournie en paramètre

Exemple :

```
          queueSender.send(textMessage);
```

72.4.7. La réception de messages : l'interface QueueReceiver

Cette interface hérite de l'interface MessageConsumer.

Pour obtenir un objet qui implémente cette interface, il faut utiliser la méthode createQueueReceiver() à partir d'un objet de type QueueSession.

Exemple :

```
          QueueReceiver queueReceiver = session.createReceiver(fileReception);
```

Il est possible de fournir un objet de type Queue qui représente la file d'attente : dans ce cas, l'objet QueueReceiver est lié à cette file d'attente. Si l'on ne précise pas de file d'attente (null fourni en paramètre), dans ce cas, il faudra obligatoirement utiliser une version surchargée de la méthode receive() pour l'indiquer.

Cette interface ne définit qu'une seule méthode supplémentaire :

Méthode	Rôle
Queue getQueue()	Renvoyer la file d'attente associée à l'objet

La réception de messages peut se faire dans le mode synchrone ou asynchrone.

72.4.7.1. La réception dans le mode synchrone

Dans ce mode, le programme est interrompu jusqu'à l'arrivée d'un nouveau message. Il faut utiliser la méthode receive() héritée de l'interface MessageConsumer. Il existe plusieurs méthodes et surcharges de ces méthodes qui permettent de répondre à plusieurs utilisations :

- receiveNoWait() : renvoie un message présent sans attendre
- receive(long) : renvoie un message arrivé durant le délais fourni en paramètre
- receive() : renvoie le message dès qu'il arrive

Exemple :

```
          Message message = null;
          message = queueReceiver.receive(10000);
```

72.4.7.2. La réception dans le mode asynchrone

Dans ce mode, le programme n'est pas interrompu mais un objet écouteur va être enregistré auprès de l'objet de type QueueReceiver. Cet objet qui implémente l'interface MessageListener va être utilisé comme gestionnaire d'événements lors de l'arrivée d'un nouveau message.

L'interface MessageListener ne définit qu'une seule méthode qui reçoit en paramètre le message : onMessage(). C'est cette méthode qui sera appelée lors de la réception d'un message.

72.4.7.3. La sélection de messages

Une version surchargée de la méthode createReceiver() d'un objet de type QueueSession permet de préciser dans ses paramètres une chaîne de caractères qui va servir de filtre sur les messages à recevoir.

Dans ce cas, le filtre est appliqué par le broker de message plutôt que par le programme.

Cette chaîne de caractères contient une expression qui doit avoir une syntaxe proche d'une condition SQL. Les critères de la sélection doivent porter sur des champs inclus dans l'en-tête ou dans les propriétés du message. Il n'est pas possible d'utiliser des données du corps du message pour effectuer le filtre.

Exemple : envoi d'un message requête et attente de sa réponse. Dans ce cas, le champ JMSCorrelationID du message réponse contient le JMSMessageID de la requête

```
String messageEnvoi = "bonjour";
TextMessage textMessage = session.createTextMessage();
textMessage.setText(messageEnvoi);
queueSender.send(textMessage);

int correlId = textMessage.getJMSMessageID();
QueueReceiver queueReceiver = session.createReceiver(
  fileEnvoi, "JMSCorrelationID = '" + correlId +"'");
Message message = null;
message = queueReceiver.receive(10000);
```

72.5. L'utilisation du mode publication/abonnement (publish/subscribe)

72.5.1. La création d'une fabrique de connexions : TopicConnectionFactory

Un objet TopicConnectionFactory permet d'obtenir une TopicConnection pour se connecter au broker de messages.

Pour obtenir un objet de ce type, il faut soit instancier directement un tel objet soit faire appel à JNDI pour l'obtenir.

72.5.2. L'interface TopicConnection

Cette interface hérite de l'interface Connection.

Pour obtenir un objet qui implémente cette interface, il faut utiliser un objet factory correspondant de type TopicConnectionFactory avec la méthode associée : createTopicConnection().

Exemple :

```
TopicConnection connection = factory.createTopicConnection();
connection.start();
```

L'interface TopicConnection définit plusieurs méthodes dont la principale est :

Méthode	Rôle
TopicSession createTopicSession(boolean, int)	Renvoyer un objet qui définit la session. Le booléen précise si la session gère une transaction. L'entier précise le mode d'accusé de réception.

72.5.3. La session : l'interface TopicSession

Elle hérite de l'interface Session.

Pour obtenir un objet qui implémente cette interface, il faut utiliser la méthode createTopicSession() d'un objet connexion de type TopicConnection.

Exemple :

```
      TopicSession session = connection.createTopicSession(false,Session.AUTO_ACKNOWLEDGE);
```

L'interface TopicSession définit plusieurs méthodes dont les principales sont :

Méthode	Rôle
TopicSubscriber createSubscriber(Topic)	Renvoyer un objet qui permet la réception de messages dans un topic
TopicPublisher createPublisher(Topic)	Renvoyer un objet qui permet l'envoi de messages dans un topic
Topic createTopic(String)	Créer un topic correspondant à la désignation fournie en paramètre

72.5.4. L'interface Topic

Un objet qui implémente cette interface encapsule un sujet.

Pour obtenir un objet qui implémente cette interface, il faut utiliser la méthode createTopic() d'un objet de type TopicSession.

72.5.5. La création d'un message

Pour créer un message, il faut utiliser une méthode createXXXMessage() d'un objet TopicSession où XXX représente le type du message.

Exemple :

```
            String message = "bonjour";
            TextMessage textMessage = session.createTextMessage();
            textMessage.setText(message);
```

72.5.6. L'émission de messages : l'interface TopicPublisher

Cette interface hérite de l'interface MessageProducer.

Avec un objet de type TopicPublisher, la méthode publish() permet l'envoi du message. Cette méthode possède plusieurs surcharges :

Méthode	Rôle
void publish(Message)	Envoyer le message dans le topic défini dans l'objet de type TopicPublisher
void publish(Topic, Message)	Envoyer le message dans le topic fourni en paramètre

72.5.7. La réception de messages : l'interface TopicSubscriber

Cette interface hérite de l'interface MessageProducer.

Pour obtenir un objet qui implémente cette interface, il faut utiliser la méthode createSubscriber() à partir d'un objet de type TopicSession.

Exemple :

```
      TopicSubscriber topicSubscriber = session.createSubscriber(topic);
```

Il est possible de fournir un objet de type Topic qui représente le topic : dans ce cas, l'objet TopicSubscriber est lié à ce topic. Si l'on ne précise pas de topic (null fourni en paramètre), il faudra obligatoirement utiliser une version surchargée de la méthode receive() lors de l'envoi pour l'indiquer.

Cette interface ne définit qu'une seule méthode supplémentaire :

Méthode	Rôle
Topic getTopic()	Renvoyer le topic associé à l'objet

72.6. La gestion des erreurs

Les erreurs d'exécution liées à l'utilisation de JMS sont rapportées sous la forme d'exceptions. La plupart des méthodes des objets JMS peuvent lever une exception de type JMSException.

Lors de l'utilisation d'un message asynchrone, il est possible d'enregistrer un listener de type ExceptionListener. Une instance de ce listener redéfinit la méthode onException() qui attend en paramètre une instance de type JMSException

72.6.1. Les exceptions de JMS

Plusieurs exceptions sont définies par l'API JMS. La classe mère de toutes ces exceptions est la classe JMSException.

Les exceptions définies sont :

- IllegalStateException,
- InvalidClientIDException,
- InvalidDestinationException,
- InvalidSelectorException,
- JMSSecurityException,
- MessageEOFException,
- MessageFormatException,
- MessageNotReadableException,
- MessageNotWriteableException,
- ResourceAllocationException,
- TransactionInProgressException,
- TransactionRolledBackException

La méthode getErrorCode() permet d'obtenir le code erreur spécifique du produit sous la forme d'une chaîne de caractères.

72.6.2. L'interface ExceptionListener

Ce listener permet d'être informé des exceptions levées par le provider JMS (exemple : arrêt du serveur, problème réseau, ..).

Cette interface définit la méthode OnException() qui doit être implémentée pour contenir les traitements en cas d'erreur.

Exemple :

```
import javax.jms.ExceptionListener;
import javax.jms.JMSException;

public class MonExceptionListener implements ExceptionListener {

    public void onException(JMSException jmse) {
        jmse.printStackTrace(System.err);
    }
}
```

72.7. JMS 1.1

La version 1.1 de JMS propose une utilisation de l'API indépendamment du domaine utilisé et ainsi d'unifier l'API pour les modes d'utilisation point à point et publication/souscription. Avec cette version, l'utilisation d'un mode ou l'autre ne nécessite plus d'interfaces spécifiques au mode utilisé, ce qui rend l'API plus simple à utiliser.

JMS 1.1 contient toujours toutes les interfaces dépendantes du domaine utilisé mais propose aussi l'enrichissement des interfaces communes pour permettent leur utilisation indépendamment du domaine utilisé.

La version 1.0.2 de l'API définit trois familles d'interfaces : commune, Queue et Topic. Pour chaque mode, une interface spécifique est définie pour la fabrique, la connexion, la session, la production et la consommation de messages.

Interfaces communes	Interfaces point à point	Interfaces publication/souscription
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Session	QueueSession	TopicSession
Destination	Queue	Topic
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver QueueBrowser	TopicSubscriber

JMS propose aussi 9 interfaces supplémentaires pour le support des transactions distribuées avec XA.

Avec JMS 1.1 il est possible de généraliser l'utilisation des interfaces communes que ce soit pour une utilisation dans le mode point à point ou publication/souscription. Ces interfaces communes ont été enrichies pour rendre les interfaces filles polymorphes. Par exemple, l'interface MessageProducer possède une méthode send() pour permettre à un client d'envoyer un message dans un mode ou un autre. Ainsi l'interface MessageProducer permet de réaliser les actions des interfaces QueueSender et TopicPublisher.

Le code devient donc plus simple, plus générique et plus réutilisable.

Ceci permet de rendre le code indépendant de la solution utilisée : l'utilisation d'une instance de type Destination se fait pour une Queue ou pour un Topic.

Ceci permet aussi dans une même session d'utiliser une queue et un topic simultanément alors que dans les versions précédentes de JMS, il était nécessaire de définir deux sessions.

JMS 1.1 permet l'utilisation de destinations qui n'ont pas besoin de savoir si celles-ci concernent une Queue ou un Topic : le code écrit peut utiliser indifféremment l'un ou l'autre.

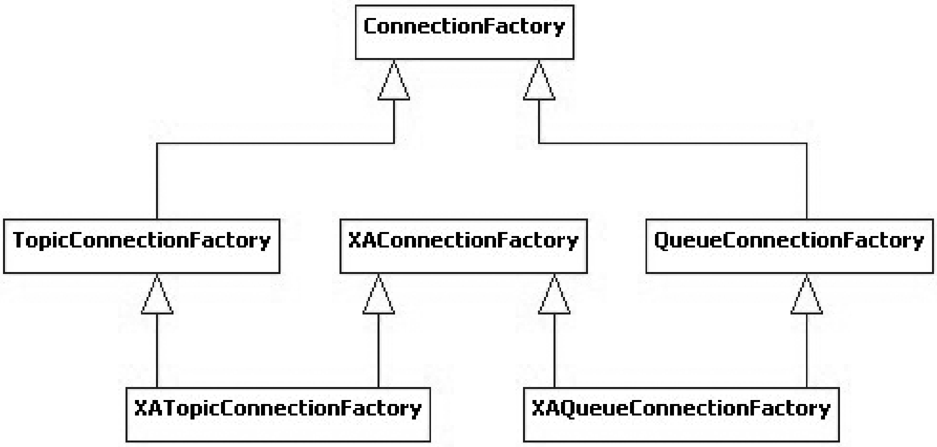
La version 1.1 a été diffusée en avril 2002. Cette version est intégrée à J2EE 1.4 : c'est un pré-requis pour la version 2.1 des EJB.

72.7.1. L'utilisation de l'API JMS 1.0 et 1.1

Point à point	Publication/souscription	Point à point ou Publication/souscription
JMS 1.0 ou 1.1	JMS 1.0 ou 1.1	JMS 1.1 uniquement
Obtenir une instance de la fabrique de type connectionFactory		
Obtenir une instance de QueueConnectionFactory à partir de JNDI	Obtenir une instance de TopicConnectionFactory à partir de JNDI	Obtenir une instance de ConnectionFactory à partir de JNDI
Créer une instance de Connection		
Appel de la méthode createQueueConnection() de la fabrique	Appel de la méthode createTopicConnection() de la fabrique	Appel de la méthode createConnection() de la fabrique
Créer une instance de Session		
Appel de la méthode createQueueSession() de la connexion	Appel de la méthode createTopicSession() de la connexion	Appel de la méthode createSession de la connexion
Créer une instance de MessageProducer		
Créer une instance de QueueSender en utilisant la méthode createSender() de la session	Créer une instance de TopicPublisher en utilisant la méthode createPublisher() de la session	Créer une instance de TopicPublisher en la méthode createProducer() de la session
Envoyer un message		
Utiliser la méthode send() de la classe QueueSender	Utiliser la méthode publish() de la classe TopicPublisher	Utiliser la méthode send() de la classe MessageProducer
Créer une instance de MessageConsumer		
Créer une instance de la classe QueueReceiver en utilisant la méthode createReceiver() de la session	Créer une instance de la classe QueueReceiver en utilisant la méthode createReceiver() de la session	Créer une instance de la classe MessageConsumer en utilisant la méthode createConsumer() de la session
Recevoir un message de façon synchrone		
Appel de la méthode receive() de l'instance de QueueReceiver	Appel de la méthode receive() de l'instance de TopicSubscriber	Appel de la méthode receive() de l'instance de MessageConsumer
Recevoir un message de façon asynchrone		
Implémenter l'interface MessageListerner et l'enregistrer avec la méthode setMessageListener() de la classe QueueReceiver	Implémenter l'interface MessageListerner et l'enregistrer avec la méthode setMessageListener() de la classe TopicSubscriber	Implémenter l'interface MessageListerner et l'enregistrer avec la méthode setMessageListener() de la classe MessageConsumer

72.7.2. L'interface ConnectionFactory

Un objet de type ConnectionFactory est une fabrique qui permet d'obtenir une instance de l'interface Connection. Il faut interroger un annuaire JNDI pour obtenir une instance de cette fabrique.

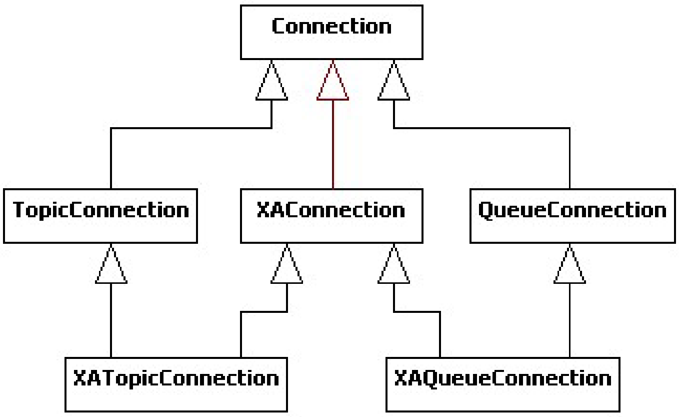


Avec JMS 1.1, il est maintenant possible d'utiliser une instance de ConnectionFactory directement : il n'est plus nécessaire comme dans les versions précédentes d'utiliser une fabrique dédiée à l'utilisation de Queue ou de Topic.

Remarque : Avec certaines implémentations, il est possible de créer manuellement une instance de ConnectionFactory mais cela nécessite de faire appel à des objets spécifiques à l'implémentation ce qui rend le code dépendant et donc moins portable.

72.7.3. L'interface Connection

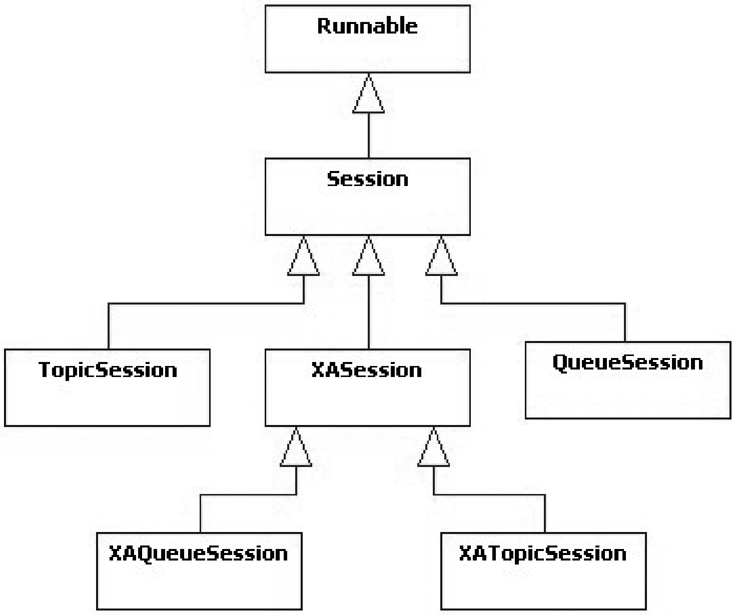
L'interface Connection permet de se connecter au serveur JMS. Avec JMS1.1, pour obtenir une instance du type Connection, il faut utiliser une des surcharges de la méthode createConnection() de l'interface ConnectionFactory.



La méthode start() de l'interface Connection permet de démarrer la connexion.

72.7.4. L'interface Session

Une session est une fabrique de messages et elle encapsule un contexte dans lequel les messages sont produits et consommés.



Une session JMS permet de créer les objets de type MessageProducer, MessageConsumer et Message.

Pour obtenir une instance de l'interface Session, il faut utiliser la méthode createSession(). Depuis JMS 1.1, cette méthode est disponible dans l'interface Connection.

Depuis JMS 1.1, de nouvelles méthodes ont été ajoutées à l'interface Session :

Méthode	Rôle
MessageProducer createProducer()	
MessageConsumer createConsumer()	
Queue createQueue()	
Topic createTopic()	
TopicSubscriber createDurableSubscriber()	
QueueBrowser createBrowser()	
TemporaryTopic createTemporaryTopic()	
TemporaryQueue createTemporaryQueue()	
void unsubscribe()	

Pour obtenir une session, il faut utiliser la méthode createSession() de l'interface Connection.

Cette méthode attend deux paramètres :

- Un booléen qui précise si la session est transactionnelle (true) ou non (false)
- Un entier qui précise le mode d'acquittement de la réception d'un message (Session.AUTO_ACKNOWLEDGE, Session.CLIENT_ACKNOWLEDGE, ou Session.DUPS_OK_ACKNOWLEDGE)

Une connexion JMS est thread-safe par contre la session JMS ne l'est pas : il faut donc utiliser une session par thread.

Une session JMS peut être transactionnelle en passant la valeur true au paramètre transacted des methodes createSession(), createQueueSession() ou createTopicSession().
Pour valider la transaction, il faut utiliser la méthode commit() de l'interface Session.

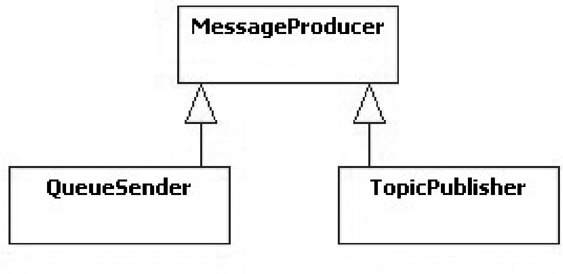
72.7.5. L'interface Destination

L'interface Destination est la super-interface des interfaces Queue et Topic.

Avec JMS 1.1, il est préférable d'utiliser cette interface plutôt que d'utiliser une interface dédiée au domaine utilisé.

72.7.6. L'interface MessageProducer

L'interface MessageProducer permet d'envoyer un message vers une destination indépendamment du domaine utilisé (queue ou topic)



Avec JMS 1.1, une instance est obtenue en utilisant la méthode createProducer() de l'interface Session avec en paramètre la destination. Il est aussi possible de créer un MessageProducer sans préciser la destination. Dans ce cas, cette dernière devra être indiquée lors de l'envoi du message.

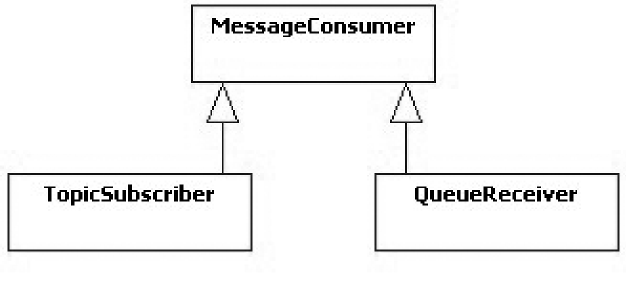
Depuis JMS 1.1, il est possible d'utiliser une instance de cette interface pour produire des messages. Avec JMS 1.0, il était nécessaire d'utiliser TopicPublisher ou QueueSender.

Depuis JMS 1.1, de nouvelles méthodes ont été ajoutées à l'interface MessageProducer notamment la méthode getDestination() et plusieurs surcharges de la méthode send().

Les surcharges de la méthode send() de l'interface MessageProducer permettent d'envoyer un message fourni en paramètre.

72.7.7. L'interface MessageConsumer

L'interface MessageConsumer permet la réception de messages d'une destination. Une instance est obtenue en utilisant la méthode createConsumer() de l'interface Session. Cette méthode attend en paramètre une destination.



72.7.7.1. La réception synchrone de messages

La méthode receive() de l'interface MessageConsumer permet d'attendre l'arrivée d'un nouveau message en bloquant le reste de l'application. Une version surchargée attend en paramètre un nombre de millisecondes indiquant la durée d'attente maximale.

La méthode receiveNoWait() permet de recevoir un éventuel nouveau message sans attendre.

Un message reçu est retourné par ces méthodes sous la forme d'un objet de type Message. Pour traiter le message, il faut caster ce résultat en fonction du type réel de l'objet.

72.7.7.2. La réception asynchrone de messages

La méthode receive() de la classe MessageConsumer permet de recevoir un message de façon synchrone. Lors de l'appel à cette méthode un message est obtenu ou non.

L'arrivée d'un message est cependant rarement prévisible et surtout ne doit pas bloquer l'exécution de l'application. Il est alors préférable de définir un listener et de l'enregistrer pour qu'il soit automatiquement exécuté à l'arrivée d'un message.

L'interface MessageListener permet de définir un listener pour la réception asynchrone de messages. Elle ne définit que la méthode onMessage() qui sera appelée lors de chaque réception d'un nouveau message de la destination.

La méthode onMessage() possède un paramètre de type Message encapsulant le message reçu. Il faut redéfinir cette méthode pour qu'elle exécute les traitements à réaliser sur les messages.

Le listener s'enregistre en utilisant la méthode setMessageListener() de la classe MessageConsumer().

Exemple :

```
  messageConsumer.setMessageListener(listener);
```

Remarque : il est important d'enregistrer le listener après que la connexion au serveur est réalisée (appel de la méthode start() de la Connection).

72.7.8. Le filtrage des messages

Il est possible de filtrer les messages reçus d'une destination au moyen d'un sélecteur (selector). Les fonctionnalités utilisables correspondent à un petit sous-ensemble de l'ensemble des fonctionnalités de SQL.

Le filtre ne peut s'appliquer que sur certaines données de l'en-tête : JMSDeliveryMode, JMSPriority, JMSMessageID, JMSCorrelationID, JMSType et JMSTimestamp

Le filtre peut aussi utiliser toutes les propriétés personnelles du message.

Exemple :

```
JMSPriority < 10
```

Lors de l'instanciation d'un objet de type MessageConsumer, il est possible de préciser le filtre des messages à recevoir sous la forme d'une chaîne de caractères. Cette chaîne est une expression qui précise le filtre à appliquer et est nommée selector..

Exemple :

```
        messageConsumer consumer = session.createConsumer(destination, "maPropriete = '1234' ") ;
```

Il est possible de définir ses propres propriétés et de les utiliser dans le filtre. Le nom de ces propriétés doit impérativement respecter les spécifications de JMS (par exemple, le nom ne peut pas commencer par JMSX ou JMS_).

La valeur d'une propriété peut être de type boolean, byte, short, int, long, float, double ou String.

Les valeurs des propriétés sont précisées avant l'envoi du message et ne peuvent plus être modifiées après l'envoi du message.

Les spécifications JMS ne précisent pas de règle pour l'utilisation d'une donnée sous la forme d'une propriété ou dans le corps du message. Il est cependant conseillé de réserver l'utilisation des propriétés pour des besoins spécifiques (filtre de messages par exemple).

Les filtres permettent à un client de ne recevoir que les messages dont les données de l'en-tête respectent le filtre précisé. Il n'est pas possible d'utiliser dans le filtre les données du corps du message.

Les messages retenus sont ceux dont l'évaluation de l'expression avec les valeurs de l'en-tête du message vaut true.

Le filtre ne peut pas être changé en cours d'exécution.

72.7.8.1. La définition du filtre

Le filtre, nommé selector est une chaîne de caractères définissant une expression dont la syntaxe est un sous-ensemble des expressions conditionnelles de la norme SQL 92.

Par défaut, le filtre est évalué de gauche à droite mais l'usage de parenthèses peut être mis en oeuvre pour modifier cet ordre.

Un selector peut contenir :

des séparateurs	espaces, tabulations, retour chariot, ...
des littéraux	des chaînes de caractères encodées en Unicode et entourées par de simples quotes des numériques entiers correspondant au type Java long des numériques flottants correspondant au type Java double des booléens qui peuvent avoir les valeurs true ou false
des identifiants	leur nom doit respecter ceux des identifiants Java et ne doivent pas correspondre à des mots clés (true, false, null, not, and, or, ...) ils ne doivent pas commencer par JMSX ou JMS_ ils sont sensibles à la casse ils ne peuvent pas correspondre aux propriétés d'en-tête prédéfinis : JMSDeliveryMode, JMSPriority, JMSMessageID, JMSTimestamp, JMSCorrelationID ou JMSType
des parenthèses	pour modifier l'ordre d'évaluation de l'expression
des expressions	arithmétiques conditionnelles
des opérateurs logiques	NOT, AND, OR
des opérateurs de comparaisons	=, >, >=, <, <=, <> (seuls = et <> sont utilisables avec des booléens et des chaînes de caractères)
des opérateurs arithmétiques	+, -, *, /
l'opérateur between	exemple : valeur between 5 and 9 est équivalent à valeur >= 5 et valeur <= 9, valeur not between 5 and 9 est équivalent à valeur < 5 ou valeur > 9
l'opérateur in	permet la comparaison parmi plusieurs chaînes de caractères (exemple : valeur in ("aa", "bb", "cc") est équivalant à (valeur = "aa") ou (valeur = "bb") ou (valeur = "cc"))
l'opérateur like	permet la comparaison par rapport à un motif : dans ce motif le caractère _ désigne un caractère quelconque, le caractère % désigne zéro ou plusieurs caractères, le caractères \ permet de déspécialiser les deux précédents caractères
L'opérateur is null	permet de tester la valeur null d'une propriété ou l'existence de sa définition

72.7.9. Des exemples de mise en oeuvre

Exemple : envoie d'un message dans une queue

```
package fr.jmdoudoux.dej.openjms;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class TestOpenJMS1 {

  public static void main(final String[] args) {
    Context context = null;
    ConnectionFactory factory = null;
    Connection connection = null;
    Destination destination = null;
    Session session = null;
    MessageProducer sender = null;
    try {
      context = new InitialContext();
      factory = (ConnectionFactory) context.lookup("ConnectionFactory");
      destination = (Destination) context.lookup("queue1");
      connection = factory.createConnection();
      session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
      sender = session.createProducer(destination);
      connection.start();

      final TextMessage message = session.createTextMessage();
      message.setText("Mon message");
      sender.send(message);
      System.out.println("Message envoye= " + message.getText());
    } catch (Exception e) {
      e.printStackTrace();
    } finally {
      if (context != null) {
        try {
          context.close();
        } catch (Exception e) {
          e.printStackTrace();
        }
      }

      if (connection != null) {
        try {
          connection.close();
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
    }
  }
}
```

Pour exécuter correctement l'application il faut qu'un broker de messages JMS soit installé et configuré. Il suffit alors de fournir les paramètres de connexion à ce serveur.

Exemple : le fichier jndi.properties avec OpenJMS

```
java.naming.provider.url=tcp://localhost:3035
java.naming.factory.initial=org.exolab.jms.jndi.InitialContextFactory
java.naming.security.principal=admin
java.naming.security.credentials=openjms
```

Résultat :

```
Message envoye= Mon message
```

Grâce à la version 1.1 de JMS, pour envoyer un message dans le topic1, il suffit simplement le remplacer le nom JNDI de la destination

Exemple :

```
...
      destination = (Destination) context.lookup("topic1");
...
```

Exemple : lecture d'un message dans une file d'attente

```
package fr.jmdoudoux.dej.openjms;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class TestOpenJMS2 {

  public static void main(String[] args) {
    Context context = null;
    ConnectionFactory factory = null;
    Connection connection = null;
    Destination destination = null;
    Session session = null;
    MessageConsumer receiver = null;

    try {
      context = new InitialContext();
      factory = (ConnectionFactory) context.lookup("ConnectionFactory");
      destination = (Destination) context.lookup("queue1");
      connection = factory.createConnection();
      session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
      receiver = session.createConsumer(destination);
      connection.start();

        Message message = receiver.receive();
        if (message instanceof TextMessage) {
          TextMessage text = (TextMessage) message;
          System.out.println("message recu= " + text.getText());
        } else if (message != null) {
          System.out.println("Aucun message dans la file");
        }
    } catch (Exception e) {
      e.printStackTrace();
    } finally {
      if (context != null) {
        try {
          context.close();
        } catch (NamingException e) {
          e.printStackTrace();
        }
      }

      if (connection != null) {
        try {
          connection.close();
        } catch (JMSException e) {
          e.printStackTrace();
          }
        }
      }
    }
  }
```

72.8. Les ressources relatives à JMS

Le site de JMS.

La Documentation de l'API JMS en version 1.0.2b et 1.1.

Pour mettre en oeuvre JMS, il faut une implémentation de l'API fournie soit par un serveur d'applications soit par une implémentation autonome.

Les principaux brokers de messages commerciaux sont :

Produit	Société	URL
Swift MQ		https://www.swiftmq.com
IBM MQ (Websphere MQ / MQ Series)	IBM	https://www.ibm.com/products/mq

Il existe quelques brokers de messages open source :

Outils	Description / URL
Apache ActiveMQ	MOM Open Source de la fondation Apache https://activemq.apache.org
OW2 Joram	implémentation open source des spécifications JMS par le consortium OW2 https://joram.ow2.io/
OpenJMS	implémentation Open Source des spécifications JMS http://openjms.sourceforge.net/

73. Les EJB (Entreprise Java Bean)

Chapitre 73

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Les Entreprise Java Bean ou EJB sont des composants serveurs donc non visuels qui respectent les spécifications d'un modèle éditées par Sun. Ces spécifications définissent une architecture, un environnement d'exécution et un ensemble d'API.

Le respect de ces spécifications permet d'utiliser les EJB de façon indépendante du serveur d'applications J2EE dans lequel ils s'exécutent, du moment que le code de mise en oeuvre n'utilise pas d'extensions proposées par un serveur d'applications particulier.

Le but des EJB est de faciliter la création d'applications distribuées pour les entreprises.

Une des principales caractéristiques des EJB est de permettre aux développeurs de se concentrer sur les traitements orientés métiers car les EJB et l'environnement dans lequel ils s'exécutent prennent en charge un certain nombre de traitements tel que la gestion des transactions, la persistance des données, la sécurité, ...

Physiquement, un EJB est un ensemble d'au moins deux interfaces et une classe regroupées dans un module contenant un descripteur de déploiement particulier.

Pour obtenir des informations complémentaires sur les EJB, il est possible de consulter le site : https://www.oracle.com/java/technologies/enterprise-javabeans-technology.html

Il existe plusieurs versions des spécifications des E.J.B. :

- 1.0 :
- 1.1 :
- 2.0 :
- 2.1 :
- 3.0 :

Remarque : dans ce chapitre, le mot bean sera utilisé comme synonyme d'EJB. Ce chapitre couvre essentiellement la version 2.x des EJB.

Ce chapitre contient plusieurs sections :

- ♦ La présentation des EJB
- ♦ Les EJB session
- ♦ Les EJB entité
- ♦ Les outils pour développer et mettre en oeuvre des EJB
- ♦ Le déploiement des EJB
- ♦ L'appel d'un EJB par un client
- ♦ Les EJB orientés messages

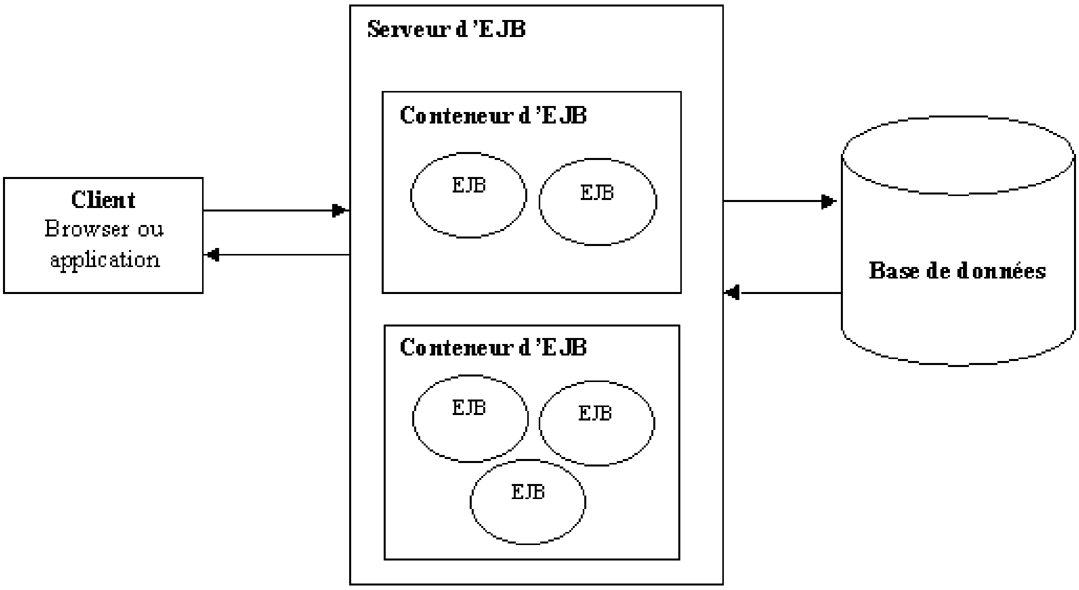
73.1. La présentation des EJB

Les EJB sont des composants et en tant que tels, ils possèdent certaines caractéristiques comme la réutilisabilité, la possibilité de s'assembler pour construire une application, etc ... Les EJB et les beans n'ont en commun que d'être des composants. Les JavaBeans sont des composants qui peuvent être utilisés dans toutes les circonstances. Les EJB doivent obligatoirement s'exécuter dans un environnement serveur dédié.

Les EJB sont parfaitement adaptés pour être intégrés dans une architecture trois tiers ou plus. Dans une telle architecture, chaque tier assure une fonction particulière :

- le client « léger » assure la saisie et l'affichage des données
- sur le serveur, les objets métiers contiennent les traitements. Les EJB sont spécialement conçus pour constituer de telles entités.
- une base de données assure la persistance des informations

Les EJB s'exécutent dans un environnement particulier : le serveur d'EJB. Celui-ci fournit un ensemble de fonctionnalités utilisées par un ou plusieurs conteneurs d'EJB qui constituent le serveur d'EJB. En réalité, c'est dans un conteneur que s'exécute un EJB et il lui est impossible de s'exécuter en dehors.



Le conteneur d'EJB propose un certain nombre de services qui assurent la gestion :

- du cycle de vie du bean
- de l'accès au bean
- de la sécurité d'accès
- des accès concurrents
- des transactions

Les entités externes au serveur qui appellent un EJB ne communiquent pas directement avec celui-ci. Les accès aux EJB par un client se font obligatoirement par le conteneur. Un objet héritant de la classe EJBObject assure le dialogue entre ces entités et les EJB en passant par le conteneur. L'avantage c'est que l'EJB peut utiliser les services proposés par le conteneur et libérer ainsi le développeur de cette charge de travail. Ceci permet au développeur de se concentrer sur les traitements métiers proposés par le bean.

Il existe de plusieurs serveurs d'EJB commerciaux : BEA Weblogic, IBM Webpshere, Sun IPlanet, Macromedia JRun, Borland AppServer, etc ... Il existe aussi des serveurs d'EJB open source : Glassfish, Wildfly, Jonas, ...

73.1.1. Les différents types d'EJB

Il existe deux types d'EJB : les beans de session (session beans) et les beans entité (les entity beans). Depuis la version 2.0 des EJB, il existe un troisième type de bean : les beans orientés message (message driven beans). Ces trois types de bean possèdent des points communs notamment celui de devoir être déployés dans un conteneur d'EJB.

Les session beans peuvent être de deux types : sans état (stateless) ou avec état (stateful).

Les beans de session sans état peuvent être utilisés pour traiter les requêtes de plusieurs clients. Les beans de session avec état ne sont accessibles que lors d'un ou plusieurs échanges avec le même client. Ce type de bean peut conserver des données entre les échanges avec le client.

Les beans entités assurent la persistance des données. Il existe deux types d'entity bean :

- persistance gérée par le conteneur (CMP : Container Managed Persistence)
- persistance gérée par le bean (BMP : Bean Managed Persistence).

Avec un bean entité CMP (container-managed persistence), c'est le conteneur d'EJB qui assure la persistance des données. Un bean entité BMP (bean-managed persistence), assure lui-même la persistance des données grâce à du code inclus dans le bean.

La spécification 2.0 des EJB définit un troisième type d'EJB : les beans orientés messages (message-driven beans).

73.1.2. Le développement d'un EJB

Le cycle de développement d'un EJB comprend :

- la création des interfaces et des classes du bean
- le packaging du bean sous forme de fichier archive jar
- le déploiement du bean dans un serveur d'EJB
- le test du bean

La création d'un bean nécessite la création d'au minimum deux interfaces et une classe pour respecter les spécifications de Sun : la classe du bean, l'interface remote et l'interface home.

L'interface remote permet de définir l'ensemble des services fournis par le bean. Cette interface étend l'interface EJBObject. Dans la version 2.0 des EJB, l'API propose une interface supplémentaire, EJBLocalObject, pour définir les services fournis par le bean qui peuvent être appelés en local par d'autres beans. Ceci permet d'éviter de mettre en oeuvre toute une mécanique longue et coûteuse en ressources pour appeler des beans s'exécutant dans le même conteneur.

L'interface home permet de définir l'ensemble des services qui vont assurer la gestion du cycle de vie du bean. Cette interface étend l'interface EJBHome.

La classe du bean contient l'implémentation des traitements du bean. Cette classe implémente les méthodes déclarées dans les interfaces home et remote. Les méthodes définissant celles de l'interface home sont obligatoirement préfixées par "ejb".

L'accès aux fonctionnalités du bean se fait obligatoirement par les méthodes définies dans les interfaces home et remote.

Il existe un certain nombre d'API qu'il n'est pas possible d'utiliser dans un EJB :

- les threads
- flux pour des entrées/sorties
- du code natif
- AWT et Swing

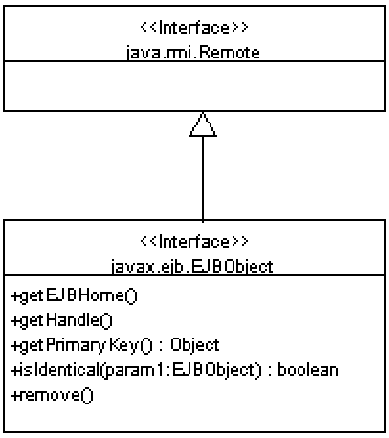
73.1.3. L'interface remote

L'interface remote permet de définir les méthodes qui contiendront les traitements proposés par le bean. Cette interface doit étendre l'interface javax.ejb.EJBObject.

Exemple :

```
package fr.jmdoudouxejb;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;

public interface MonPremierEJB extends EJBObject {
  public String message() throws RemoteException;
}
```

Toutes les méthodes définies dans cette interface doivent obligatoirement respecter les spécifications de RMI et déclarer qu'elles peuvent lever une exception de type RemoteException.



L'interface javax.ejb.EJBObject définit plusieurs méthodes qui seront donc présentes dans tous les EJB :

- EJBHome getEJBHome() throws java.rmi.RemoteException : renvoie une référence sur l'objet Home
- Handle getHandle() throws java.rmi.RemoteException : renvoie un objet permettant de sérialiser le bean
- Object getPrimaryKey() throws java.rmi.RemoteException : renvoie une référence sur l'objet qui encapsule la clé primaire d'un bean entité
- boolean isIdentical(EJBObject) throws java.rmi.RemoteException : renvoie un boolean qui précise si le bean est identique à l'instance du bean fournie en paramètre. Pour un bean session sans état, cette méthode renvoie toujours true. Pour un bean entité, la méthode renvoie true si les clés primaires des deux beans sont identiques
- void remove() throws java.rmi.RemoteException, javax.ejb.RemoveException : cette méthode demande la destruction du bean. Pour un bean entité, elle provoque la suppression des données correspondantes dans la base de données

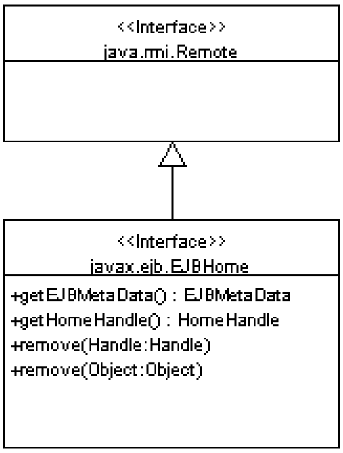
73.1.4. L'interface home

L'interface home permet de définir des méthodes qui vont gérer le cycle de vie du bean. Cette interface doit étendre l'interface EJBHome.

La création d'une instance d'un bean se fait grâce à une ou plusieurs surcharges de la méthode create(). Chacune de ces méthodes renvoie une instance d'un objet du type de l'interface remote.

Exemple :

```
package fr.jmdoudouxejb;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface MonPremierEJBHome extends EJBHome {
  public MonPremierEJB create() throws CreateException, RemoteException;
}
```



L'interface javax.ejb.EJBHome définit plusieurs méthodes :

- EJBMetaData getEJBMetaData() throws java.rmi.RemoteException
- HomeHandle getHomeHandle() throws java.rmi.RemoteException : renvoie un objet qui permet de sérialiser l'objet implémentant l'interface EJBHome
- void remove(Handle) throws java.rmi.RemoteException, javax.ejb.RemoveException : supprime le bean
- void remove(Object) throws java.rmi.RemoteException, javax.ejb.RemoveException : supprime le bean entité dont l'objet encapsulant la clé primaire est fourni en paramètre

La ou les méthodes à définir dans l'interface home dépendent du type d'EJB:

Type de bean	Méthodes à définir
bean session sans état	une seule méthode create() sans paramètre
bean session avec état	une ou plusieurs méthodes create()
bean entité	aucune ou plusieurs méthodes create() et une ou plusieurs méthodes finder()

73.2. Les EJB session

Les EJB session sont des EJB de service dont la durée de vie correspond à un échange avec un client. Ils contiennent les règles métiers de l'application.

Il existe deux types d'EJB session : sans état (stateless) et avec état (stateful).

Les EJB session stateful sont capables de conserver l'état du bean dans des variables d'instance durant toute la conversation avec un client. Mais ces données ne sont pas persistantes : à la fin de l'échange avec le client, l'instance de l'EJB est détruite et les données sont perdues.

Les EJB session stateless ne peuvent pas conserver de telles données entre chaque appel du client.

Il ne faut pas faire appel directement aux méthodes create() et remove() de l'EJB. C'est le conteneur d'EJB qui se charge de la gestion du cycle de vie de l'EJB et qui appelle ces méthodes. Le client décide simplement du moment de la création et de la suppression du bean en passant par le conteneur.

Une classe qui encapsule un EJB session doit implémenter l'interface javax.ejb.SessionBean. Elle ne doit pas implémenter les interfaces home et remote mais définir les méthodes déclarées dans ces deux interfaces.

La classe qui implémente le bean doit définir les méthodes de l'interface remote. La classe doit aussi définir les méthodes ejbCreate(), ejbRemove(), ejbActivate(), ejbPassivate et setSessionContext().

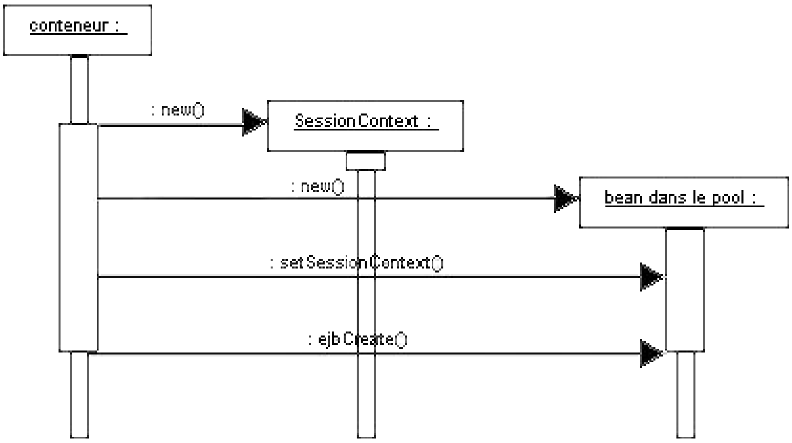
La méthode ejbRemove() est appelée par le conteneur lors de la suppression de l'instance du bean.

Pour permettre au serveur d'applications d'assurer la montée en charge des différentes applications qui s'exécutent dans ses conteneurs, celui-ci peut momentanément libérer de la mémoire en déchargeant un ou plusieurs beans. Cette action consiste à sérialiser le bean sur le système de fichiers et à le désérialiser pour sa remontée en mémoire. Lors de ces deux actions, le conteneur appelle respectivement les méthodes ejbPassivate() et ejbActivate().

73.2.1. Les EJB session sans état

Ce type de bean propose des services sous la forme de méthodes. Il ne peut pas conserver de données entre deux appels de méthodes. Les données nécessaires aux traitements d'une méthode doivent obligatoirement être fournies par le client en paramètre de la méthode.

Les services proposés par ces beans peuvent être gérés dans un pool par le conteneur pour améliorer les performances puisqu'ils sont indépendants du client qui les utilise. Le pool contient un certain nombre d'instances du bean. Toutes ces instances étant "identiques", il suffit au conteneur d'ajouter ou de supprimer de nouvelles instances dans le pool selon les variations de la charge du serveur d'applications. Il est donc inutile au serveur de sérialiser un EJB session sans état. Il suffit simplement de déclarer les méthodes ejbActivate() et ejbPassivate() sans traitements.



Le conteneur s'assure qu'un même bean ne recevra pas d'appel de méthode de la part de deux clients différents en même temps.

Exemple :

```
package fr.jmdoudouxejb;

import java.rmi.RemoteException;
import javax.ejb.EJBException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;


public class MonPremierEJBBean implements SessionBean {

  public String message() {
    return "Bonjour";
  }

  public void ejbActivate() {
  }

  public void ejbPassivate() {
  }

  public void ejbRemove() {
  }
  public void setSessionContext(SessionContext arg0) throws EJBException, RemoteException {
  }

  public void ejbCreate() {
  }
}
```

73.2.2. Les EJB session avec état

Ce type de bean fournit aussi un ensemble de traitements grâce à ses méthodes mais il a la possibilité de conserver des données entre les différents appels de méthodes d'un même client. Une instance particulière est donc dédiée à chaque client qui sollicite ses services et ce tout au long du dialogue entre les deux entités.

Les données conservées par le bean sont stockées dans les variables d'instances du bean. Les données sont donc conservées en mémoire. Généralement, les méthodes proposées par le bean permettent de consulter et mettre à jour ces données.

Dans un EJB session avec état il est possible de définir plusieurs méthodes permettant la création d'un tel EJB. Ces méthodes doivent obligatoirement commencer par ejbCreate.

Les méthodes ejbPassivate() et ejbActivate() doivent définir et contenir les éventuels traitements lors de leur appel par le conteneur. Celui-ci appelle ces deux méthodes respectivement lors de la sérialisation du bean et sa désérialisation. La méthode ejbActivate() doit contenir les traitements nécessaires à la restitution du bean dans un état utilisable après la désérialisation.

Le cycle de vie d'un ejb avec état est donc identique à celui d'un bean sans état avec un état supplémentaire lorsque celui-ci est sérialisé. La fin du bean peut être demandée par le client lorsque celui-ci utilise la méthode remove(). Le conteneur invoque la méthode ejbRemove() du bean avant de supprimer sa référence.

Certaines méthodes métiers doivent permettre de modifier les données stockées dans le bean.

73.3. Les EJB entité

Ces EJB permettent de représenter et de gérer des données enregistrées dans une base de données. Ils implémentent l'interface EntityBean.

L'avantage d'utiliser un tel type d'EJB plutôt que d'utiliser JDBC ou de développer sa propre solution pour mapper les données est que certains services sont pris en charge par le conteneur.

Les beans entités assurent la persistance des données en représentant tout au partie d'une table ou d'une vue. Il existe deux types de bean entité :

- persistance gérée par le conteneur (CMP : Container Managed Persistence)
- persistance gérée par le bean (BMP : Bean Managed Persistence).

Avec un bean entité CMP (container-managed persistence), c'est le conteneur d'EJB qui assure la persistance des données grâce aux paramètres fournis dans le descripteur de déploiement du bean. Il se charge de toute la logique des traitements de synchronisation entre les données du bean et les données dans la base de données.

Un bean entité BMP (bean-managed persistence), assure lui-même la persistance des données grâce à du code inclus dans les méthodes du bean.

Plusieurs clients peuvent accéder simultanément à un même EJB entity. La gestion des transactions et des accès concurrents est assurée par le conteneur.

73.4. Les outils pour développer et mettre en oeuvre des EJB

La mise en oeuvre des EJB requiert un conteneur d'EJB généralement inclus dans un serveur d'applications et un IDE pour être productif.

73.4.1. Les outils de développement

Plusieurs EDI (Environnement de Développement Intégré) open source permettent de développer et de tester des EJB notamment Eclipse et Netbeans. Netbeans est d'ailleurs celui qui propose le plus rapidement une implémentation pour mettre en oeuvre la dernière version des spécifications relatives aux EJB.

73.4.2. Les conteneurs d'EJB

Il existe plusieurs conteneurs d'EJB commerciaux mais aussi d'excellents conteneurs d'EJB open source notamment Glassfish, JBoss ou Jonas.

73.5. Le déploiement des EJB

Un EJB doit être déployé sous forme d'une archive jar contenant un fichier qui est le descripteur de déploiement et toutes les classes qui composent chaque EJB (interfaces home et remote, les classes qui implémentent ces interfaces et toutes les autres classes nécessaires aux EJB).

Une archive ne doit contenir qu'un seul descripteur de déploiement pour tous les EJB de l'archive. Ce fichier au format XML doit obligatoirement être nommé ejb-jar.xml.

L'archive doit contenir un répertoire META-INF (attention au respect de la casse) qui contiendra lui-même le descripteur de déploiement.

Le reste de l'archive doit contenir les fichiers .class avec toute l'arborescence des répertoires des packages.

Le jar des EJB peut être inclus dans un fichier de type EAR.

73.5.1. Le descripteur de déploiement

Le descripteur de déploiement est un fichier au format XML qui permet de fournir au conteneur des informations sur les beans à déployer. Le contenu de ce fichier dépend du type de beans à déployer.

73.5.2. La mise en package des beans

Une fois toutes les classes et le fichier de déploiement écrits, il faut les rassembler dans une archive .jar afin de pouvoir les déployer dans le conteneur.

73.6. L'appel d'un EJB par un client

Un client peut être une entité de toute forme : application avec ou sans interface graphique, un bean, une servlet, une JSP ou un autre EJB.

Un EJB étant un objet distribué, son appel utilise RMI.

Le stub est une représentation locale de l'objet distant. Il implémente l'interface remote mais contient une connexion réseau pour accéder au skeleton de l'objet distant.

Le mode d'appel d'un EJB suit toujours la même logique :

- obtenir une référence qui implémente l'interface home de l'EJB grâce à JNDI
- créer une instance qui implémente l'interface remote en utilisant la référence précédemment acquise
- appel de la ou des méthodes de l'EJB

73.6.1. Un exemple d'appel d'un EJB session

L'appel d'un EJB session avec ou sans état suit la même logique.

Il faut tout d'abord utiliser un objet du type InitialContext pour pouvoir interroger JNDI. Cet objet nécessite qu'on lui fournisse des informations dont le nom de la classe à utiliser comme fabrique et l'url du serveur JNDI.

Cet objet permet d'obtenir une référence sur le bean enregistré dans JNDI. A partir de cette référence, il est possible de créer un objet qui implémente l'interface home. Un appel à la méthode create() sur cet objet permet de créer un objet du type de l'EJB. L'appel des méthodes de cet objet entraîne l'appel des méthodes de l'objet EJB qui s'exécute dans le conteneur.

Exemple :

```
package testEJBClient;

import java.util.*;
import javax.naming.*;

public class EJBClient {

  public static void main(String[] args) {
    Properties ppt = null;
    Context ctx = null;
    Object ref = null;
    MonPremierBeanHome home = null;
    MonPremierBean bean = null;

    try {
      ppt = new Properties();
      ppt.put(Context.INITIAL_CONTEXT_FACTORY, "org.jnp.interfaces.NamingContextFactory");
      ppt.put(Context.PROVIDER_URL, "localhost:1099");
      ctx = new InitialContext(ppt);
      ref = ctx.lookup("MonPremierBean");
      home = (MonPremierBeanHome) javax.rmi.PortableRemoteObject.narrow(ref,
        MonPremierBeanHome.class);
      bean = home.create();
      System.out.println("message = " + bean.message());
      bean.remove();
    }
    catch (Exception e) {
       e.printStackTrace();
    }
  }
}
```

73.7. Les EJB orientés messages

Ces EJB sont différents des deux types d'EJB précédents car ils répondent à des invocations de façon asynchrone. Ils permettent de réagir à l'arrivée de messages fournis par un M.O.M. (Middleware Oriented Messages).

74. Les EJB 3

Chapitre 74

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Les EJB (Entreprise Java Bean) sont un des éléments très importants de la plate-forme Java EE pour le développement d'applications distribuées.

La plate-forme Java EE propose de mettre en oeuvre les couches métiers et persistance avec les EJB. Particulièrement intéressants dans des environnements fortement distribués, jusqu'à la version 3, leur mise en oeuvre est assez lourde sans l'utilisation d'outils tels que certains IDE ou XDoclet.

La version 3 des EJB vise donc à simplifier le développement et la mise en oeuvre des EJB qui sont fréquemment jugés trop complexes et trop lourds à mettre en oeuvre.

Cette nouvelle version majeure des EJB propose une simplification de leur développement tout en conservant une compatibilité avec sa précédente version. Elle apporte de très nombreuses fonctionnalités dans le but de simplifier la mise en oeuvre des EJB.

Cette simplification est rendue possible notamment par :

- l'utilisation des annotations
- la mise en oeuvre de valeurs par défaut qui répondent à la plupart des besoins (configuration par exception)
- le descripteur de déploiement est facultatif
- l'utilisation de POJO et de JPA pour les beans de type entity
- l'injection de dépendances côté serveur mais aussi côté client (l'interface Home qui gérait le cycle de vie est abandonnée) qui remplace l'utilisation directe de JNDI
- ...

Tous ces éléments délèguent une partie du travail du développeur au conteneur d'EJB.

Ce chapitre contient plusieurs sections :

- L'historique des EJB
- Les nouveaux concepts et fonctionnalités utilisés
- EJB 2.x vs EJB 3.0
- Les conventions de nommage
- Les EJB de type Session
- Les EJB de type Entity
- Un exemple simple complet
- L'utilisation des EJB par un client
- L'injection de dépendances
- Les intercepteurs
- Les EJB de type MessageDriven
- Le packaging des EJB
- Les transactions
- La mise en oeuvre de la sécurité

74.1. L'historique des EJB

EJB 1.1 publié en décembre 1999, intégré dans J2EE 1.2 :

- Session beans (stateless/stateful)
- Entity Beans (CMP / BMP)
- Interface Remote uniquement

EJB 2.0 publié en septembre 2001, intégré à J2EE 1.3 :

- Message-Driven Beans
- Entity 2.x reposant sur EJB QL
- Interface Local pour améliorer les performances des appels dans la même JVM

EJB 2.1 publié en novembre 2003, intégré à J2EE 1.4 :

- EJB Timer Service
- EJB Web Service Endpoints via JAX-RPC
- Amélioration du langage EJB QL

EJB 3.0, intégré à Java EE 5 :

- utilisation de POJO et POJI, plus d'interface Home
- utilisation des annotations, le descripteur de déploiement est optionnel
- utilisation de JPA pour les beans de type entity

EJB 3.1, intégré à Java EE 6

74.2. Les nouveaux concepts et fonctionnalités utilisés

Dans les versions antérieures à la version 3.0 des EJB, le développeur était contraint de créer de nombreuses entités pour respecter l'API EJB (par exemple, l'implémentation d'interfaces engendrant la création de plusieurs méthodes ou le descripteur de déploiement), ce qui rendait l'écriture relativement lourde même avec l'assistance de certains IDE ou outils (XDoclet notamment). Dans la version 3.0, ceci est remplacé par l'utilisation d'annotations.

La mise en oeuvre de l'interface EJBHome n'est plus requise : un EJB de type session est maintenant une simple classe, qui peut implémenter une interface métier.

La seule annotation obligatoire dans un EJB est celle qui précise le type d'EJB (@javax.ejb.Stateless, @javax.ejb.Stateful ou @javax.ejb.MessageDriven).

Les annotations possèdent des valeurs par défaut qui répondent à une majorité de cas typiques d'utilisations. L'utilisation de ces annotations n'est alors requise que si les valeurs par défaut ne répondent pas au besoin. Ceci permet de réduire la quantité de code à écrire.

L'utilisation des annotations et de valeurs par défaut pour la plupart de ces dernières rend optionnelle la nécessité de créer un descripteur de déploiement sauf pour des besoins très particuliers.

Le conteneur obtient des informations sur la façon de mettre en oeuvre un EJB par trois moyens :

- Des valeurs par défaut pour la plupart des annotations ce qui évite d'avoir à les déclarer explicitement dans le code
- Les annotations utilisées dans le code
- Le descripteur de déploiement

L'ordre d'utilisation par le conteneur est : le descripteur de déploiement, les annotations, les valeurs par défaut.

L'utilisation des annotations est plus simple à mettre en oeuvre mais le descripteur de déploiement permet de centraliser les informations.

La nouvelle API Java Persistence remplace la persistance assurée par le conteneur : cette API assure la persistance des données grâce à un mapping O/R reposant sur des POJO.

Le conteneur a la possibilité d'injecter des dépendances d'objets dont il assure la gestion.

Les intercepteurs permettent d'offrir des fonctionnalités proches de celles proposées par l'AOP : ceci permet de définir des traitements lors de l'invocation de méthodes des EJB ou d'invoquer des méthodes particulières liées au cycle de vie de l'EJB.

74.2.1. L'utilisation de POJO et POJI

Les classes et les interfaces des EJB 3.0 sont de simples POJO ou POJI : ceci simplifie le développement des EJB. Par exemple, l'interface Home n'est plus à déclarer.

Il est toujours possible d'implémenter les interfaces SessionBean, EntityBean et MessageDrivenBean mais le plus simple est d'utiliser les annotations définies : @Stateless, @Stateful, @Entity ou @MessageDriven

Exemple :

```
@Stateless
public class HelloWorldBean {
  public String saluer(String nom) {
    return "Bonjour "+nom;
  }
}
```

Il est possible de définir une interface métier pour l'EJB ou de laisser générer cette interface lors du déploiement.

Dans le premier cas, il n'est plus nécessaire qu'elle implémente l'interface EJBObject ou EJBLocalObject mais il faut simplement utiliser les annotations définies : @Remote ou @Local.

Exemple :

```
@Remote
@Stateless
public class HelloWorldBean {
  public String saluer(String nom)
  {
    return "Bonjour "+nom;
  }
}
```

Dans le second cas, ces annotations doivent être utilisées dans la classe d'implémentation pour permettre de déterminer l'interface générée.

Il est possible de définir une interface locale et/ou distante pour un même EJB.

Il n'est pas recommandé de laisser les interfaces être générées par le conteneur pour plusieurs raisons :

- les interfaces générées exposent par défaut toutes les méthodes de l'EJB
- l'interface est utilisée par le client pour invoquer l'EJB
- le nom des interfaces générées utilise le nom de l'implémentation de l'EJB

74.2.2. L'utilisation des annotations

La spécification 3.0 des EJB fait un usage intensif des annotations. Celles-ci sont issues de la JSR 175 et intégrées dans Java SE 5.0 qui constitue la base de Java EE 5.

Les annotations sont des attributs ou métadonnées à l'image de celles proposées par XDoclet.

Avec les EJB 3.0, les annotations sont utilisées pour générer des entités et remplacer tout ou partie du descripteur de déploiement.

De nombreuses annotations permettent de simplifier le développement des EJB.

La nature de l'EJB est précisée par une des annotations @Stateless, @Stateful, @Entity et @MessageDriven selon le type d'EJB à définir.

Le type d'accès est précisé par deux annotations

- @Remote : permet un accès à l'EJB depuis un client hors de la JVM
- @Local : permet un accès à l'EJB depuis un client dans la même JVM que celle de l'EJB

Par défaut, l'interface d'appel est locale si aucune annotation n'est indiquée.

Dans le cas d'un accès distant, il est inutile que chaque méthode précise qu'elle peut lever une exception de type RemoteException mais elles peuvent déclarer la levée d'exceptions métiers.

Jusqu'à la version 2.1 des EJB, il était obligatoire d'implémenter plusieurs méthodes relatives à la gestion du cycle de vie de l'EJB notamment ejbActivate, ejbLoad, ejbPassivate, ejbRemove, ... pour chaque EJB même si ces méthodes ne contenaient aucun traitement.

Avec les EJB 3.0, l'implémentation de ces méthodes est remplacée par l'utilisation facultative d'annotations sur les méthodes concernées. La signature de ces méthodes doit être de la forme public void nomMethode()

Par exemple, pour que le conteneur exécute automatiquement une méthode avant de retirer l'instance du bean, il faut annoter la méthode avec l'annotation @Remove.

Plusieurs annotations permettent ainsi de définir des méthodes qui interviendront dans le cycle de vie de l'EJB.

Annotation	Rôle
@PostConstruct	la méthode est invoquée après que l'instance est créée et que les dépendances sont injectées
@PostActivate	la méthode est invoquée après que l'instance de l'EJB est désérialisée du disque. C'est l'équivalent de la méthode ejbActivate() des EJB 2.x
@Remove	la méthode est invoquée avant que l'EJB ne soit retiré du conteneur
@PreDestroy	la méthode est invoquée avant que l'instance de l'EJB ne soit supprimée
@PrePassivate	la méthode est invoquée avant que de l'instance de l'EJB ne soit sérialisée sur disque. C'est l'équivalent de la méthode ejbPassivate() des EJB 2.x

L'utilisation facultative de ces annotations remplace la définition obligatoire des méthodes de gestion du cycle de vie utilisées jusqu'à la version 2.1 des EJB.

Le descripteur de déploiement n'est plus obligatoire puisqu'il peut être remplacé par l'utilisation d'annotations dédiées directement dans les classes des EJB.

Chaque attribut de déploiement possède une valeur par défaut qu'il ne faut définir que si cette valeur ne répond pas au besoin.

Plusieurs annotations sont définies par les spécifications des EJB pour permettre de déclarer le type de bean, le type de l'interface, des références vers des ressources qui seront injectées, la gestion des transactions, la gestion de la sécurité, ...

Chaque vendeur peut définir en plus ses propres annotations dans l'implémentation de son serveur d'applications. Leur utilisation n'est cependant pas recommandée car elle rend l'application dépendante du serveur d'applications utilisé.

L'utilisation des annotations va simplifier le développement des EJB mais la gestion de la configuration pourra devenir plus complexe puisqu'elle n'est plus centralisée.

74.2.3. L'injection de dépendances

L'EJB déclare les ressources dont il a besoin à l'aide d'annotations. Le conteneur va injecter ces ressources lorsqu'il va instancier l'EJB donc avant l'appel aux méthodes liées au cycle de vie du bean ou aux méthodes métiers. Ceci impose que l'injection de ressources se fasse sur des objets gérés par le conteneur.

Ces ressources peuvent être de diverses natures : référence vers un autre EJB, contexte de sécurité, contexte de persistance, contexte de transaction, ...

Plusieurs annotations sont définies pour mettre en oeuvre l'injection de dépendances :

- L'annotation @EJB permet d'injecter une ressource de type EJB.
- L'annotation @Resource permet d'injecter une ressource qui est obtenue par JNDI (EntityManager, UserTransaction, SessionContext, ...)
- ...

L'utilisation de l'injection de dépendances remplace l'utilisation implicite de JNDI.

L'injection peut aussi être définie dans le descripteur de déploiement.

74.2.4. La configuration par défaut

La suite de cette section sera développée dans une version future de ce document

74.2.5. Les intercepteurs

Les intercepteurs sont une fonctionnalité avancée similaire à celle proposée par l'AOP : ils permettent d'intercepter l'invocation de méthodes pour exécuter des traitements.

Ils sont définis grâce à des annotations dédiées notamment @Interceptors et @AroundInvoke ou dans le descripteur de déploiement.

Leur utilisation peut être variée : traces, logs, gestion de la sécurité, ...

74.3. EJB 2.x vs EJB 3.0

Le développement d'EJB n'a jamais été facile et est même devenu plus complexe au fur et à mesure des nouvelles spécifications.

Avant la version 3.0 des EJB, les EJB étaient relativement complexes et lourds à mettre en oeuvre :

- création de plusieurs interfaces et classes (deux interfaces et une classe au minimum)
- implémentation de méthodes callback généralement inutiles
- l'interface de l'EJB doit hériter de EJBObject ou de EJBLocalObject
- chaque méthode de l'EJB doit déclarer pouvoir lever l'exception RemoteException
- le descripteur de déploiement des EJB est complexe

- les EJB entité de type CMP présentent plusieurs limitations : complexes à développer et à maintenir, de nombreux problèmes de performance, le langage EJBQL est limité
- le support de la POO pour les EJB est très limité vis-à-vis de l'héritage
- les EJB doivent être testés dans un conteneur ce qui les rend difficiles à déboguer
- l'appel d'un EJB par un client nécessite obligatoirement une utilisation de JNDI

La version 3.0 des spécifications des EJB apporte une solution de simplification à tous les points précédemment cités.

- Il n'est plus nécessaire de déclarer d'interfaces (pour des raisons de bonne pratique, la déclaration d'une interface métier contenant les méthodes proposées est cependant fortement recommandée)
- Le descripteur de déploiement est optionnel sauf dans des cas particuliers
- L'utilisation de POJO et POJI
- L'injection de dépendances rend très facile l'obtention d'une instance d'une ressource gérée par le conteneur
- ...

Les principales différences entre les EJB 2.x et EJB 3.0 sont donc :

- les descripteurs de déploiement ne sont plus obligatoires grâce à l'utilisation d'annotations et de valeurs par défaut
- Les EJB sont de simples POJO annotés : ils n'ont plus besoin d'implémenter une interface de l'API EJB. De fait, il n'est plus nécessaire de définir des méthodes liées au cycle de vie de l'EJB. Si ces méthodes sont nécessaires, il suffit d'utiliser des annotations dédiées sur une méthode.
- Le type de l'interface de l'EJB est précisé avec l'annotation @Local ou @Remote
- L'interface métier est une simple POJI
- Les EJB de type Entity CMP et BMP sont remplacés par l'utilisation du modèle de persistance reposant sur l'API JPA

74.4. Les conventions de nommage

Il n'existe pas de règles imposées mais il est important de définir des conventions de nommage pour les différentes entités qui sont utilisées lors de la mise en oeuvre des EJB.

Exemple :

Nom du bean : CalculEJB

Nom de la classe métier : CalculBean

Interface locale : CalculLocal

Interface distante : CalculRemote

74.5. Les EJB de type Session

Les EJB Session sont généralement utilisés comme façade pour proposer des fonctionnalités qui peuvent faire appel à d'autres composants ou entités tels que des EJB session, des EJB Entity, des POJO, ...

La version 3.0 des EJB rend inutile l'implémentation d'une interface spécifique à l'API EJB. Mais même si cela n'est pas obligatoire, il est fortement recommandé (dans la mesure du possible) de définir une interface dédiée à l'EJB qui va notamment préciser son mode d'accès et les méthodes utilisables.

Cette interface est alors une simple POJI.

74.5.1. L'interface distante et/ou locale

Un EJB peut être invoqué :

- en local : le client appelant est exécuté dans la même JVM que celle de l'EJB. Ce type d'appel est le plus performant puisqu'il ne nécessite pas d'échanges réseaux et donc pas de mécanisme pour gérer ces échanges
- à distance : le client appelant est exécuté dans une autre JVM que celle de l'EJB

L'interface distante définit les méthodes qui peuvent être appelées par un client en dehors de la JVM du conteneur. L'interface ou le bean doit être marqué avec l'annotation @Remote implémentée dans la classe javax.ejb.Remote

L'interface locale définit les méthodes qui peuvent être appelées par un autre EJB s'exécutant dans la même JVM que le conteneur. Les performances sont ainsi accrues car les mécanismes de protocoles d'appels distants ne sont pas utilisés (sérialisation/désérialisation, RMI, ...).

L'utilisation de l'interface Local pour des appels à l'EJB dans un même JVM est fortement recommandée car cela améliore les performances de façon importante. L'interface Remote met en oeuvre des mécanismes de communication utilisant la sérialisation, ce qui dégrade les performances notamment de façon inutile si l'appel à l'EJB se fait dans une même JVM.

Un client ne dialogue jamais en direct avec une instance de l'EJB : le client utilise toujours l'interface pour accéder au bean grâce à un proxy généré par le conteneur. Même un client local utilise un proxy particulier dépourvu des accès réseau. Ce proxy permet au conteneur d'assurer certaines fonctionnalités comme la sécurité et les transactions.

74.5.2. Les beans de type Stateless

Les beans de type stateless sont les plus simples et les plus véloces car le conteneur gère un pool d'instances qui sont utilisées au besoin, ce qui évite des opérations d'instanciation et de destruction à chaque utilisation. Ceci permet une meilleure montée en charge de l'application.

L'annotation @javax.ejb.Stateless permet de préciser qu'un EJB session est de type stateless. Elle s'utilise sur une classe qui encapsule un EJB et possède plusieurs attributs :

Attribut	Rôle
String name	Nom de l'EJB (optionnel)
String mappedName	Nom sous lequel l'EJB sera mappé. La valeur par défaut est le nom non qualifié de la classe (optionnel)
String description	Description de l'EJB (optionnel)

Il faut définir l'interface de l'EJB avec l'annotation précisant le mode d'accès.

L'annotation @javax.ejb.Remote permet de préciser que l'EJB pourra être accédé par des clients distants. Elle s'utilise sur une classe qui encapsule un EJB ou l'interface qui décrit les fonctionnalités de l'EJB utilisables à distance. Cette annotation ne peut être utilisée que pour des EJB sessions.

Elle possède un seul attribut :

Attribut	Rôle
Class[] value	Préciser la liste des interfaces distantes de l'EJB. Son utilisation est obligatoire si la classe de l'EJB implémente plusieurs interfaces différentes java.io.Serializable, java.io.Externalizable ou une des interfaces du package javax.ejb (optionnel)

Exemple :

```
import javax.ejb.Remote;

```

```
@Remote
public interface CalculRemote {
  public long additionner(int valeur1, int valeur2);
}
```

Cette interface est marquée avec l'annotation @Remote, elle permet un appel distant et définit la méthode additionner.

Remarque : l'utilisation de l'annotation rend inutile l'utilisation de la clause throws RemoteException des versions antérieures des EJB.

L'annotation @javax.ejb.Local permet de préciser que l'EJB pourra être accédé par des clients locaux de la JVM. Elle s'utilise sur une classe qui encapsule un EJB ou l'interface qui décrit les fonctionnalités de l'EJB utilisables en local dans la JVM. Cette annotation ne peut être utilisée que pour des EJB sessions.

Elle possède un attribut :

Attribut	Rôle
Class[] value	Préciser la liste des interfaces distantes de l'EJB. Son utilisation est obligatoire si la classe de l'EJB implémente plusieurs interfaces différentes java.io.Serializable, java.io.Externalizable ou une des interfaces du package javax.ejb (optionnel)

Exemple :

```
import javax.ejb.Local;

@Local
public interface CalculLocal {
  public long additionner(int valeur1, int valeur2);
}
```

Cette interface est marquée avec l'annotation @Local pour permettre un appel local et définir la méthode additionner.

Il faut ensuite définir la classe de l'EJB qui va contenir les traitements métiers.

Exemple :

```
import javax.ejb.*;

@Stateless
public class CalculBean implements CalculRemote, CalculLocal {
  public long additionner(int valeur1, int valeur2) {
    return valeur1 + valeur2;
  }
}
```

Cette classe est marquée avec l'annotation @Stateless et implémente les interfaces distante et locale précédemment définies.

Il est préférable lorsque cela est possible d'utiliser l'interface Local car elle est beaucoup plus performante. L'interface Remote est à utiliser lorsque le client n'est pas dans la même JVM.

Les annotations @Local et @Remote peuvent être utilisées directement sur l'EJB mais il est préférable de définir une interface par mode d'accès et d'utiliser l'annotation adéquate sur chacune des interfaces.

La classe de l'EJB ne doit plus implémenter l'interface javax.ejb.SessionBean qui était obligatoire avec les EJB 2.x. Maintenant, les EJB Session de type stateless peuvent utiliser les callbacks d'évènements marqués avec les annotations suivantes :

- @PostConstruct
- @PreDestroy

74.5.3. Les beans de type Stateful

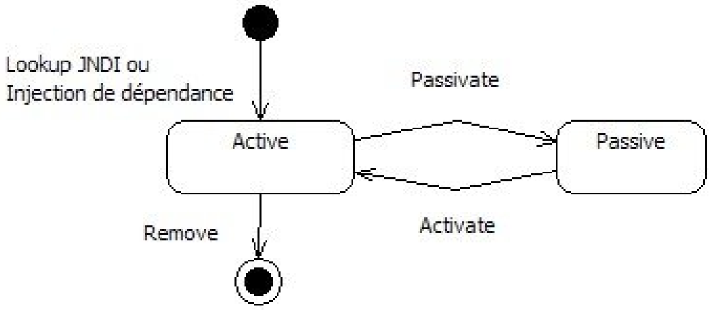
Les beans de type Stateful sont capables de conserver leur état durant toute leur utilisation par le client. Cet état n'est cependant pas persistant : les données sont perdues à la fin de son utilisation ou à l'arrêt du serveur. Un exemple type d'utilisation de ce type de bean est l'implémentation d'un caddie pour un site de vente en ligne.

L'annotation @javax.ejb.Stateful permet de préciser qu'un EJB Session est de type Stateful. Elle s'utilise sur une classe qui encapsule un EJB.

Elle possède plusieurs attributs :

Attribut	Rôle
String name	Nom de l'EJB (optionnel)
String mappedName	Nom sous lequel l'EJB sera mappé. La valeur par défaut est le nom non qualifié de la classe (optionnel)
String description	Description de l'EJB (optionnel)

Le conteneur EJB a la possibilité de sérialiser/désérialiser des EJB de type Stateful notamment dans le cas où la JVM du conteneur commence à manquer de mémoire. Dans ce cas, le conteneur peut sérialiser des EJB (passivate) qui ne sont pas en cours d'utilisation sur le disque. Dès qu'un de ces EJB sera sollicité, le conteneur va le déserialiser (activate) à partir du disque pour le remettre en mémoire et pouvoir l'utiliser.



Les EJB 3 n'ont plus à implémenter l'interface javax.ejb.SessionBean comme c'était le cas dans les versions antérieures. Maintenant, les EJB session de type stateful peuvent utiliser les callbacks d'évènements marqués avec les annotations suivantes :

- @PostConstruct
- @PostActivate
- @PreDestroy
- @PrePassivate
- @Remove

74.5.4. L'invocation d'un EJB Session par un service web

Le support des services web dans les EJB 3.0 repose essentiellement sur JAX-WS 2.0 et SAAJ qu'il faut privilégier au détriment de JAX-RPC qui est toujours supporté.

La suite de cette section sera développée dans une version future de ce document

74.5.5. L'utilisation des exceptions

Les exceptions personnalisées qui sont utilisées dans les interfaces métiers des EJB doivent être annotées avec @javax.ejb.ApplicationException qui s'utilise donc sur une classe encapsulant une exception métier.

Une exception annotée avec @ApplicationException sera directement envoyée au client par le conteneur.

Elle possède un seul attribut x:

Attribut	Rôle
boolean rollback	Préciser si le conteneur doit effectuer un rollback si cette exception est levée. La valeur par défaut est false (optionnel)

L'attribut rollback de l'annotation @ApplicationException de type booléen permet de préciser si la levée de l'exception va déclencher ou non un rollback de la transaction en cours. La valeur par défaut est false, signifiant qu'il n'y aura pas de rollback.

Exemple :

```
package fr.jmdoudoux.dej.domaine.ejb;

import javax.ejb.ApplicationException;

@ApplicationException
public class ErreurMetierException extends Exception {

    public ErreurMetierException() {
    }


    public ErreurMetierException(String msg) {
        super(msg);
    }
}
```

L'annotation @ApplicationException peut être utilisée avec des exceptions de type checked et unchecked.

74.6. Les EJB de type Entity

Dans les versions antérieures des EJB, les EJB de type Entity avaient la charge de la persistance des données. Les EJB de type Entity CMP (Container Managed Persistence) nécessitent simplement un fichier de description.

Les EJB 3.0 proposent d'utiliser l'API Java Persistence pour assurer la persistance des données dans les EJB : ils utilisent un modèle de persistance léger standard en remplacement des entity beans de type CMP.

JPA repose sur des beans entity qui sont de simples POJO enrichis d'annotations permettant de mettre en oeuvre les concepts de POO tels que l'héritage ou le polymorphisme.

Jusqu'à la version 3.0 des EJB, les Entity beans sont des composants qui dépendent pleinement du conteneur d'EJB du serveur d'applications dans lequel ils s'exécutent. L'utilisation de POJO avec l'API Java Persistence permet de rendre les beans entity indépendants du conteneur. Ceci présente plusieurs avantages dont celui de pouvoir facilement tester les beans puisqu'ils ne requièrent plus de conteneur pour leur exécution.

Avec la version 3.0 des EJB, les beans entity sont donc des POJO qui n'ont pas besoin d'implémenter une interface spécifique aux EJB, doivent posséder un constructeur sans argument et implémenter l'interface Serializable.

Les attributs persistants sont déclarés par des annotations soit au niveau de l'attribut soit au niveau de son getter/setter. De ce fait, ils peuvent être utilisés directement comme objets du domaine ; il n'y a plus l'obligation de définir un DTO.

74.6.1. La création d'un bean Entity

Les beans de type Entity sont dans la version 3.0 des spécifications de simple POJO utilisant les annotations de l'API Java Persistence (JPA) pour définir le mapping.

Les informations de mapping entre une table et un objet peuvent être définies grâce aux annotations mais aussi par un fichier de mapping qui permet d'externaliser les informations du POJO. Il est possible de mixer les deux (annotations et fichiers de mapping) mais les données incluses dans le fichier sont prioritaires sur les annotations.

Le bean entity doit être annoté avec l'annotation @Entity implémentée dans la classe javax.persistence.Entity.

L'annotation @Table implémentée dans la classe javax.persistence.Table permet de préciser le nom de la table vers laquelle le bean sera mappé. L'utilisation de cette annotation est facultative si le nom de la table correspond au nom de la classe.

Pour mapper un champ de la table avec une propriété du bean, il faut utiliser l'annotation @Column implémentée dans la classe javax.persistence.Column sur le getter de la propriété. L'utilisation de cette annotation est facultative si le nom du champ correspond au nom de la propriété.

Le champ correspondant à la clé primaire de la table doit être annoté avec l'annotation @Id implémentée dans la classe javax.persistence.Id. L'utilisation de cette annotation est obligatoire car un identifiant unique est obligatoire pour chaque occurrence et l'API n'a aucun moyen de déterminer le champ qui encapsule cette information.

Il peut être pratique pour un bean de type entity d'implémenter l'interface Serializable : le bean pourra être utilisé dans les paramètres et la valeur de retour des méthodes métiers d'un EJB. Le bean peut ainsi être utilisé pour la persistance et le transfert de données.

Exemple :

```
package fr.jmdoudoux.dej.domaine.entity;

import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;

@Entity
@Table(name = "PERSONNE")
@NamedQueries({@NamedQuery(name = "Personne.findById",
  query = "SELECT p FROM Personne p WHERE p.id = :id"),
               @NamedQuery(name = "Personne.findByNom",
  query = "SELECT p FROM Personne p WHERE p.nom = :nom"),
               @NamedQuery(name = "Personne.findByPrenom",
  query = "SELECT p FROM Personne p WHERE p.prenom = :prenom")})
public class Personne implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Column(name = "ID", nullable = false)
    private Integer id;
    @Column(name = "NOM")
    private String nom;
    @Column(name = "PRENOM")
    private String prenom;

    public Personne() {
    }
    public Personne(Integer id) {
        this.id = id;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }

    public String getPrenom() {
        return prenom;
    }

    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }

    @Override
    public String toString() {
        return "fr.jmdoudoux.dej.domaine.entity.Personne[id=" + id + "]";
    }

}
```

Remarque : il est préférable de définir tous les beans de type entity dans un package dédié.

La mise en oeuvre précise de l'API JPA est proposée dans le chapitre qui lui est consacré.

74.6.2. La persistance des entités

La version 3.0 propose une refonte complète des EJB entités afin de simplifier leur développement. Cette simplification est assurée en grande partie par la mise en oeuvre de JPA qui permet :

- la standardisation du mapping O/R
- l'utilisation de POJO annotés avec support de l'héritage et du polymorphisme
- la possibilité d'utiliser les EJB entités en dehors du conteneur d'EJB ce qui permet notamment la mise en oeuvre de tests automatisés

La persistance d'objets avec JPA repose sur plusieurs fonctionnalités :

- un ensemble d'entités annotées qui représente le modèle objet du domaine
- une API contenue dans le package javax.persistence
- un cycle de vie pour les entités

La classe EntityManager est responsable de la gestion des opérations sur une entité notamment grâce à plusieurs méthodes :

- persist()
- remove()
- merge()
- flush()
- find()

- refresh()
- ...

La suite de cette section sera développée dans une version future de ce document

74.6.3. La création d'un EJB Session pour manipuler le bean Entity

Le bean entity n'est utilisé que pour le mapping. Pour réaliser des opérations avec le bean entity, il faut développer un EJB Session qui va encapsuler la logique des traitements à réaliser avec le bean entity.

Il faut définir l'interface Local et/ou Remote des méthodes métiers de l'EJB.

Il faut ensuite définir l'EJB qui va utiliser l'API Java Persistence et le bean entity.

L'injection de dépendances est utilisée pour obtenir une instance de l'EntityManager par le conteneur.

Exemple :

```
        @PersistenceContext
        private EntityManager em;
```

L'annotation @PersistenceContext demande au conteneur d'injecter une instance de la classe EntityManager.

Le conteneur retrouve l'EntityManager grâce au nom de l'unité de persistance fourni comme valeur à la propriété unitName de l'annotation si plusieurs unités de persistance sont définies.

L'instance de type EntityManager peut être utilisée dans les méthodes métiers pour réaliser des traitements sur le bean entity.

Exemple :

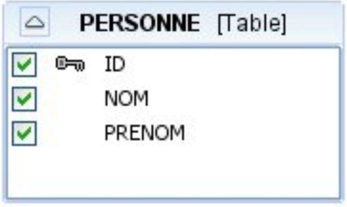
```
...
    public void create(Personne personne) {
        em.persist(personne);
    }

    public void edit(Personne personne) {
        em.merge(personne);
    }

    public void remove(Personne personne) {
        em.remove(em.merge(personne));
    }

    public Personne find(Object id) {
        return em.find(fr.jmdoudoux.dej.domaine.entity.Personne.class, id);
    }

...
```

74.7. Un exemple simple complet

L'exemple de cette section va développer un EJB métier effectuant des opérations de type CRUD sur une table nommée personne et permettant l'appel de cet EJB par un service web.

La table personne contient trois champs :

- id : identifiant unique de la personne
- nom : nom de la personne
- prenom : prénom de la personne


PERSONNE [Table]

☑ ID

☑ NOM

☑ PRENOM

Cette table est stockée dans une base de données de type JavaDB.

Une connexion vers la base de données est définie dans l'annuaire sous le nom MaTestDb

Remarque : les sources de cet exemple sont générées par l'IDE Netbeans.

74.7.1. La création de l'entité

La classe Personne encapsule une entité sur la table personne.

Exemple :

```
package fr.jmdoudoux.dej.domaine.entity;

import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;

@Entity
@Table(name = "PERSONNE")
@NamedQueries({@NamedQuery(name = "Personne.findById",
  query = "SELECT p FROM Personne p WHERE p.id = :id"),
               @NamedQuery(name = "Personne.findByNom",
  query = "SELECT p FROM Personne p WHERE p.nom = :nom"),
               @NamedQuery(name = "Personne.findByPrenom",
  query = "SELECT p FROM Personne p WHERE p.prenom = :prenom")})
public class Personne implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Column(name = "ID", nullable = false)
    private Integer id;
    @Column(name = "NOM")
    private String nom;
    @Column(name = "PRENOM")
    private String prenom;

    public Personne() {
    }

    public Personne(Integer id) {
        this.id = id;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }

    public String getPrenom() {
        return prenom;
    }

    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }

    @Override
    public String toString() {
        return "fr.jmdoudoux.dej.domaine.entity.Personne[id=" + id + "]";
    }

}
```

74.7.2. La création de la façade

L'interface métier locale est définie dans l'interface PersonneFacadeLocal

Exemple :

```
package fr.jmdoudoux.dej.domaine.ejb;

import fr.jmdoudoux.dej.domaine.entity.Personne;
import java.util.List;
import javax.ejb.Local;

@Local
public interface PersonneFacadeLocal {

    void create(Personne personne);

    void edit(Personne personne);

    void remove(Personne personne);

    Personne find(Object id);

    List<Personne> findAll();

}
```

L'interface métier distante est définie dans l'interface PersonneFacadeRemote

Exemple :

```
package fr.jmdoudoux.dej.domaine.ejb;

import fr.jmdoudoux.dej.domaine.entity.Personne;
import java.util.List;
import javax.ejb.Remote;

@Remote
public interface PersonneFacadeRemote {

    void create(Personne personne);

    void edit(Personne personne);

    void remove(Personne personne);

    Personne find(Object id);
    List<Personne> findAll();

}
```

La façade est implémentée sous la forme d'un EJB de type stateless.

Exemple :

```
package fr.jmdoudoux.dej.domaine.ejb;

import fr.jmdoudoux.dej.domaine.entity.Personne;
import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Stateless
public class PersonneFacade implements PersonneFacadeLocal, PersonneFacadeRemote {
    @PersistenceContext
    private EntityManager em;

    public void create(Personne personne) {
        em.persist(personne);
    }

    public void edit(Personne personne) {
        em.merge(personne);
    }

    public void remove(Personne personne) {
        em.remove(em.merge(personne));
    }

    public Personne find(Object id) {
        return em.find(fr.jmdoudoux.dej.domaine.entity.Personne.class, id);
    }

    public List<Personne> findAll() {
        return em.createQuery("select object(o) from Personne as o").getResultList();
    }

}
```

Les fonctionnalités offertes par l'EJB sont de type CRUD.

Le fichier persistence.xml demande simplement l'utilisation de la connexion définie dans l'annuaire.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
  <persistence-unit name="EnterpriseApplication3-ejbPU" transaction-type="JTA">
    <jta-data-source>MaTestDb</jta-data-source>
    <properties/>
  </persistence-unit>
</persistence>
```

74.7.3. La création du service web

Pour permettre une meilleure séparation des rôles de chaque classe, le service web est développé dans une classe dédiée.

Exemple :

```
package fr.jmdoudoux.dej.services;

import fr.jmdoudoux.dej.domaine.ejb.PersonneFacadeLocal;
import fr.jmdoudoux.dej.domaine.entity.Personne;
import java.util.List;
import javax.ejb.EJB;
import javax.jws.Oneway;
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.ejb.Stateless;

@WebService()
@Stateless()
public class PersonneWS {
    @EJB
    private PersonneFacadeLocal ejbRef;

    @WebMethod(operationName = "create")
    @Oneway
    public void create(Personne personne) {
        ejbRef.create(personne);
    }

    @WebMethod(operationName = "edit")
    @Oneway
    public void edit(Personne personne) {
        ejbRef.edit(personne);
    }

    @WebMethod(operationName = "remove")
    @Oneway
    public void remove(Personne personne) {
        ejbRef.remove(personne);
    }

    @WebMethod(operationName = "find")
    public Personne find(Object id) {
        return ejbRef.find(id);
    }

    @WebMethod(operationName = "findAll")
    public List<Personne> findAll() {
        return ejbRef.findAll();
    }

}
```

L'injection de dépendances est utilisée pour laisser le conteneur fournir au service web une référence sur l'instance de l'EJB.

74.8. L'utilisation des EJB par un client

La suite de cette section sera développée dans une version future de ce document

74.8.1. Pour un client de type application standalone

Un client distant est en mesure d'utiliser des EJB possédant une interface de type Remote : dans ce cas, plusieurs opérations sont à réaliser

- Connexion au serveur JNDI
- Recherche de l'interface Remote de l'EJB dans JNDI
- Récupération du proxy grâce à JNDI
- Utilisation de l'EJB au travers du proxy

Les informations nécessaires à la connexion à l'annuaire JNDI du serveur d'applications sont spécifiques à chaque implémentation du serveur.

Le nom de stockage de l'interface dans JNDI est aussi spécifique au serveur utilisé.

La suite de cette section sera développée dans une version future de ce document

74.8.2. Pour un client de type module Application Client Java EE

La suite de cette section sera développée dans une version future de ce document

74.9. L'injection de dépendances

Le conteneur peut être utilisé pour assurer l'injection de dépendances de certaines ressources requises par exemple par un contexte de persistance ou un autre EJB.

L'injection de dépendances est réalisée au moment de l'instanciation du bean par le conteneur.

L'injection de dépendances permet de simplifier le travail des développeurs : il n'est plus nécessaire d'invoquer l'annuaire du serveur par JNDI et de caster le résultat pour obtenir une instance de la dépendance. C'est le conteneur lui-même qui va s'en charger grâce à des annotations déchargeant le développeur de l'écriture du code utilisant JNDI ou un objet de type EJBContext.

Plusieurs annotations sont définies pour mettre en oeuvre cette injection de dépendances :

- @EJB : permet d'injecter une référence vers un autre EJB
- @Ressource : injecter une dépendance vers une ressource externe : DataSources JDBC, destinations JMS (queue ou topic), ... (annotation de Java 5)
- @PersistenceContext : injecter un objet de type EntityManager
- @WebServiceRef : injecter une référence vers un service web (annotation de JAX-WS)

Les annotations relatives à l'injection de dépendances peuvent être utilisées sur des variables d'instance ou sur des méthodes de type setter.

74.9.1. L'annotation @javax.ejb.EJB

L'annotation @EJB permet de demander au conteneur d'injecter une référence sur un EJB sans avoir à faire appel explicitement à l'annuaire du serveur avec JNDI.

Exemple :

```
    @EJB
    private PersonneFacadeLocal ejbReference;
```

Le conteneur utilise par défaut le type de la variable pour déterminer le type de l'instance de l'EJB qui sera injectée.

Elle s'utilise sur une classe, une méthode ou une propriété :

- sur une propriété : il est possible d'annoter un objet du type de l'EJB. L'EJB injecté sera du type de l'objet annoté.
- sur une méthode : il est possible d'annoter une méthode de type setter sur la classe de l'EJB. L'EJB injecté sera du type de l'objet en paramètre de la méthode.
- sur une classe : cela permet de déclarer que l'EJB sera utilisé à l'exécution

Elle possède plusieurs attributs :

Attribut	Rôle
Class beanInterface	Nom de l'interface de l'EJB
String beanName	Nom de l'EJB (correspond à l'attribut name des annotations @Stateless et @Stateful). Par défaut, c'est le nom de la classe de l'EJB
String description	Description de l'EJB
String mappedName	Nom JNDI de l'EJB. Cet attribut n'est pas portable
String name	Nom avec lequel l'EJB sera recherché

L'injection de dépendances est réalisée entre l'assignation de l'EJBContext et le premier appel d'une méthode de l'EJB.

S'il y a une ambiguïté pour déterminer le type de l'EJB à injecter, il est possible d'utiliser les attributs beanName et mappedName de l'annotation @EJB pour désigner l'EJB concerné.

74.9.2. L'annotation @javax.annotation.Resource

L'annotation @javax.annotation.Resource permet d'injecter des instances de ressources gérées par le conteneur telles qu'une datasource JDBC ou une destination JMS (queue ou topic) par exemple.

Cette annotation est utilisable sur une classe, une méthode ou un champ. Si l'annotation est utilisée sur une méthode ou un champ, le conteneur injecte les références au moment de l'initialisation du bean.

Elle possède plusieurs attributs :

Attribut	Rôle
String name	Nom de la ressource
Class type	Type de la ressource
AuthenticationType authenticationType	Type d'authentification à utiliser pour accéder à la ressource. Les valeurs possibles sont AuthenticationType.CONTAINER et AuthenticationType.APPLICATION
boolean shareable	Indiquer si la ressource peut être partagée entre cet EJB et d'autres EJB. Ne s'applique que sur certains types de ressources
String mappedName	Nom JNDI de la ressource
String description	Description de la ressource

74.9.3. Les annotations @javax.annotation.Resources et @javax.ejb.EJBs

La suite de cette section sera développée dans une version future de ce document

74.9.4. L'annotation @javax.xml.ws.WebServiceRef

L'annotation @WebServiceRef possède plusieurs attributs :

Attribut	Rôle
String name	Nom JNDI de la ressource
String wsdlLocation	URL pointant sur le WSDL du service web
Class type	Type Java
Class value	La classe du service qui doit obligatoirement hériter de javax.xml.ws.Service
String mappedName	

Pour définir les mêmes fonctionnalités dans le descripteur de déploiement, il faut utiliser le tag <service-ref>.

74.10. Les intercepteurs

Un intercepteur est une méthode qui sera exécutée selon deux types d'événements :

- intercepteur pour l'invocation de méthodes métiers
- intercepteur pour des événements liés au cycle de vie de l'EJB

Un intercepteur permet de définir des traitements, généralement transverses, qui seront exécutés lorsque ces événements surviendront. Leur rôle est similaire à certaines fonctionnalités de base de l'AOP (programmation orientée aspect)

Les intercepteurs sont utilisables avec des EJB Session et MessageDriven.

Les annotations dédiées, utilisées pour la mise en oeuvre des intercepteurs, sont regroupées dans le package javax.interceptor :

- AroundInvoke
- ExcludeClassInterceptors
- DefaultInterceptors
- Interceptors

74.10.1. Le développement d'un intercepteur

Un intercepteur permet de définir des traitements sous la forme de méthodes qui seront exécutées soit à l'invocation d'une méthode métier soit lors d'événements liés au cycle de vie de l'EJB. Son rôle est similaire à certaines fonctionnalités de base proposées par l'AOP.

Un intercepteur peut être défini soit :

- dans un EJB : dans ce cas il ne concerne que cet EJB
- dans une classe intercepteur : dans ce cas, il pourra être utilisé par tous les EJB qui en feront la demande en utilisant l'annotations @javax.interceptor.Interceptors

La signature des méthodes de callback diffère selon la nature de l'intercepteur :

- dans la classe d'un EJB : la signature est void nomMethode()
- dans la classe d'un intercepteur : la signature est void nomMethode(InvocationContext)

74.10.1.1. L'interface InvocationContext

L'interface javax.interceptor.InvocationContext définit les fonctionnalités pour permettre d'utiliser un contexte lors de l'invocation d'un ou plusieurs intercepteurs.

Cette interface définit plusieurs méthodes :

Méthode	Rôle
Object getTarget()	Renvoyer l'instance de l'EJB
Method getMethod()	Renvoyer la méthode métier de l'EJB qui a provoqué l'invocation de l'intercepteur. Si l'invocation est liée au cycle de vie de l'EJB alors cette méthode renvoie null
Object[] getParameters()	Renvoyer un tableau des paramètres de la méthode du bean pour laquelle l'intercepteur a été invoqué
void setParameters(Object[])	Modifier les paramètres qui seront utilisés pour l'invocation de la méthode
Map<String,Object> getContextData()	Obtenir une collection des données associées à l'invocation du callback
Object proceed()	Invoquer le prochain intercepteur de la chaîne ou de la méthode métier de l'EJB si tous les intercepteurs ont été invoqués

Une instance de type InvocationContext est passée en paramètre des intercepteurs.

Il est ainsi possible d'échanger des données entre les invocations des intercepteurs définis pour une même méthode.

Attention : une instance d'InvocationContext n'est pas partageable entre un intercepteur pour des méthodes métiers et un intercepteur pour des événements liés au cycle de vie des EJB.

74.10.1.2. La définition d'un intercepteur lié aux méthodes métiers

L'annotation @AroundInvoke permet de marquer une méthode qui sera exécutée lors de l'invocation des méthodes métiers d'un EJB. Cette annotation ne peut être utilisée qu'une seule fois dans une même classe d'un intercepteur ou d'un EJB. Il n'est pas possible d'annoter une méthode métier avec l'annotation @AroundInvoke.

La signature d'une méthode annotée avec @AroundInvoke doit être de la forme :

```
Object nomMethode(InvocationContext) throws Exception
```

Une méthode annotée avec @AroundInvoke doit toujours invoquer la méthode proceed() de l'instance de type InvocationContext fournie en paramètre pour permettre l'invocation d'éventuels autres intercepteurs assossiés à la méthode.

Exemple :

```
package fr.jmdoudoux.dej.domaine.ejb;

import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;

/**
 * Intercepteur qui calcule le temps d'exécution d'une méthode métier
 * @author jmd
 */
public class MesurePerfIntercepteur {

  @AroundInvoke
  public Object mesurerPerformance(InvocationContext ic) throws Exception {
    long debutExec = System.currentTimeMillis();
    try {
      return ic.proceed();
    } finally {
      long tempsExec = System.currentTimeMillis() - debutExec;
      System.out.println("[PERF] Temps d'execution de la methode " + ic.getClass()
        + "." + ic.getMethod() + " : " + tempsExec + " ms");
    }
  }
}
```

L'exemple ci-dessus permet de définir un intercepteur qui logguera les temps d'exécutions des méthodes métiers des EJB.

74.10.1.3. La définition d'un intercepteur lié au cycle de vie

Un intercepteur peut être exécuté lorsque certains événements liés au cycle de vie de l'EJB tels que la création, la destruction, la passivation ou la réactivation surviennent.

Les EJB 2.x imposaient l'implémentation de méthodes d'une interface telles que ejbCreate(), ejbPassivate(), ... Avec les EJB 3.x, ces méthodes peuvent avoir un nom quelconque du moment qu'elles sont annotées avec une annotation liée à un événement du cycle de vie de l'EJB. Les annotations pour définir des callbacks sur des invocations de méthodes liées au cycle de vie de l'EJB sont :

- @javax.annotation.PostConstruct : méthode invoquée par le conteneur lorsqu'il a terminé les injections de dépendances pour un EJB et avant le premier appel à une des méthodes métiers du bean
- @javax.annotation.PreDestroy : méthode invoquée par le conteneur juste avant que l'EJB ne soit définitivement détruit
- @javax.ejb.PrePassivate : méthode invoquée par le conteneur lorsqu'un EJB de type session stateful va être rendu inactif (EJB session de type stateful uniquement)
- @javax.ejb.PostActivate : méthode invoquée par le conteneur lorsqu'un EJB de type session stateful va être réactivé (EJB session de type stateful uniquement)

Il est possible dans une même classe d'utiliser plusieurs de ces annotations mais il n'est pas possible d'utiliser plusieurs fois la même dans une même classe.

Une méthode annotée avec une annotation liée au cycle de vie dans une classe d'un intercepteur doit invoquer la méthode proceed() de l'instance de type InvocationContext fournie en paramètre pour permettre l'invocation des traitements liés à l'état courant du cycle de vie de l'EJB.

74.10.1.4. La mise en oeuvre d'une classe d'un intercepteur

Une classe d'intercepteurs est un simple POJO qui doit obligatoirement avoir un constructeur sans paramètre et dont certaines méthodes sont annotées avec l'annotation @AroundInvoke ou avec une annotation liée au cycle de vie de l'EJB.

Exemple :

```
package fr.jmdoudoux.dej.domaine.ejb;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;

public class MonIntercepteur {

    @AroundInvoke
    public Object audit(InvocationContext ic)
            throws Exception {
        System.out.println("MonIntercepteur Invocation de la methode : " + ic.getMethod());
        return ic.proceed();
    }

    @PreDestroy
    public void preDestroy(InvocationContext ic) {
        System.out.println("MonIntercepteur suppression du bean : " + ic.getTarget());
    }

    @PostConstruct
    public void postConstruct(InvocationContext ic) {
        System.out.println("MonIntercepteur Creation du bean : " + ic.getTarget());
    }
}
```

Les intercepteurs peuvent avoir accès par injection de dépendances aux ressources gérées par le conteneur (EJB, EntityManager, destination JMS, ...).

Un intercepteur métier peut lever une exception applicative puisque les méthodes métiers peuvent lever une exception dans leur clause throws.

Les intercepteurs définis dans la classe de l'EJB sont exécutés après les intercepteurs précisés par l'annotation @Interceptors.

74.10.2. Les intercepteurs par défaut

Il est possible de définir des intercepteurs par défaut qui seront appliqués à tous les EJB d'un même jar.

La définition d'un intercepteur par défaut ne peut se faire que dans le descripteur de déploiement ejb-jar.xml. Ils ne peuvent pas être définis par des annotations.

Pour déclarer un intercepteur par défaut, il faut modifier le descripteur de déploiement ejb-jar.xml en utilisant un tag <interceptor-binding> fils du tag <assembly-descriptor>.

Le tag <interceptor-binding> peut avoir deux tags fils :

- <ejb-name> dont la valeur précise un filtre sur les ejb-name qui indique les EJB concernés. La valeur * permet d'indiquer que tous les EJB sont concernés.
- <interceptor-class> permet de préciser la classe pleinement qualifiée de l'intercepteur

L'intercepteur doit être déclaré dans un tag <interceptor> fils du tag <interceptors>. Le tag fils <interceptor-class> permet de préciser le nom pleinement qualifié de la classe de l'intercepteur.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns = "http://java.sun.com/xml/ns/javaee"
         version = "3.0"
         xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation = "http://java.sun.com/xml/ns/javaee
           http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd">
 <interceptors>
  <interceptor>
   <interceptor-class>fr.jmdoudoux.dej.domaine.ejb.MesurePerfIntercepteur</interceptor-class>
  </interceptor>
 </interceptors>
 <assembly-descriptor>
  <interceptor-binding>
   <ejb-name>*</ejb-name>
   <interceptor-class>fr.jmdoudoux.dej.domaine.ejb.MesurePerfIntercepteur</interceptor-class>
  </interceptor-binding>
 </assembly-descriptor>
</ejb-jar>
```

Les intercepteurs par défaut sont toujours invoqués avant les autres intercepteurs.

Pour empêcher l'invocation d'un intercepteur par défaut pour un EJB, il faut l'annoter avec l'annotation @javax.interceptor.excludeDefaultInterceptors.

Ces intercepteurs offrent donc deux avantages :

- ils peuvent s'appliquer à tout ou partie des EJB contenus dans le même jar que l'intercepteur
- la description de leur application est centralisée et configurable dans le descripteur de déploiement ce qui permet facilement de la modifier (exemple : désactiver un intercepteur par défaut)

74.10.3. Les annotations des intercepteurs

Plusieurs annotations peuvent être utilisées lors de la mise en oeuvre des intercepteurs. Celles-ci sont soit des annotations dédiées contenues dans le package javax.interceptor soit des annotations standards de l'API Java contenues dans le package javax.annotation.

74.10.3.1. L'annotation @javax.annotation.PostContruct

L'annotation @javax.annotation.Postconstruct permet de définir un intercepteur qui est lié à l'événement de création du cycle de vie de l'EJB.

La méthode annotée avec cette annotation sera invoquée par le conteneur après l'initialisation de l'EJB et l'injection des dépendances mais avant l'appel de la première méthode.

Elle peut être utilisée dans la classe d'un intercepteur ou dans la classe d'un EJB mais dans les deux cas, une seule méthode peut être annotée avec cette annotation.

Elle s'utilise sur une méthode dont la signature doit respecter quelques contraintes :

- elle ne doit pas avoir de valeur de retour
- elle ne peut pas lever d'exception de type checked
- elle ne peut pas être ni static ni final
- elle ne possède aucun attribut.

74.10.3.2. L'annotation @javax.annotation.PreDestroy

L'annotation @javax.annotation.PreDestroy permet de définir un intercepteur qui est lié à l'événement de suppression du cycle de vie de l'EJB.

La méthode annotée avec cette annotation sera invoquée par le conteneur avant que l'EJB ne soit détruit du conteneur. Celle-ci pourra par exemple procéder à la libération de ressources.

Elle peut être utilisée dans la classe d'un intercepteur ou dans la classe d'un EJB mais dans les deux cas, une seule méthode peut être annotée avec cette annotation.

Elle s'utilise sur une méthode dont la signature doit respecter quelques contraintes :

- elle ne doit pas avoir de valeur de retour
- elle ne peut pas lever d'exception de type checked
- elle ne peut pas être ni static ni final
- elle ne possède aucun attribut.

74.10.3.3. L'annotation @javax.interceptor.AroundInvoke

L'annotation @javax.interceptor.AroundInvoke permet de définir un intercepteur qui est lié à l'exécution de méthodes métiers. Cette annotation peut être utilisée dans la classe d'un intercepteur ou dans la classe d'un EJB mais dans les deux cas, une seule méthode peut être annotée avec cette annotation.

Elle s'utilise sur une méthode. Elle ne possède aucun attribut.

74.10.3.4. L'annotation @javax.interceptor.ExcludeClassInterceptors

L'annotation @javax.interceptor.ExcludeClassInterceptors permet de demander d'inhiber l'invocation des intercepteurs pour une méthode. L'inhibition ne concerne pas les intercepteurs par défaut.

Elle s'utilise sur une méthode. Elle ne possède aucun attribut

74.10.3.5. L'annotation @javax.interceptor.ExcludeDefaultInterceptors

L'annotation @javax.interceptor.ExcludeDefaultInterceptors permet d'inhiber l'invocation des intercepteurs par défaut. Utilisée sur la classe d'un bean, cette annotation inhibe l'invocation des intercepteurs par défaut pour toutes les méthodes métiers du bean. Utilisée sur une méthode d'un bean, l'inhibition se limite à cette méthode.

Cette annotation s'utilise sur une classe ou une méthode.

74.10.3.6. L'annotation @javax.interceptor.Interceptors

L'annotation @javax.interceptor.Interceptors permet de définir les classes d'intercepteurs qui seront invoquées par le conteneur. Si plusieurs classes d'intercepteurs sont définies alors elles seront invoquées dans l'ordre de leur définition dans l'annotation.

Si l'annotation est utilisée sur la classe du bean alors les intercepteurs seront invoqués pour chaque méthode du bean. Si l'annotation est utilisée sur une méthode alors les intercepteurs seront invoqués uniquement pour la méthode.

Les intercepteurs sont invoqués dans un ordre précis :

- les intercepteurs par défaut
- les intercepteurs au niveau classe
- les intercepteurs au niveau méthode

Elle s'utilise sur une classe ou une méthode. Elle possède un seul attribut :

Attribut	Rôle
Class[] value	Préciser un tableau de classes d'intercepteurs. L'ordre des intercepteurs dans le tableau définit leur ordre d'invocation (obligatoire)

74.10.4. L'utilisation d'un intercepteur

Chaque EJB qui souhaite utiliser un intercepteur devra l'ajouter grâce à l'annotation @javax.interceptor.Interceptors.

Exemple :

```
package fr.jmdoudoux.dej.domaine.ejb;

import fr.jmdoudoux.dej.domaine.entity.Personne;
import java.util.List;
import javax.ejb.Stateless;
import javax.interceptor.Interceptors;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Stateless
@Interceptors({ fr.jmdoudoux.dej.domaine.ejb.MonIntercepteur.class})
public class PersonneFacade implements PersonneFacadeLocal, PersonneFacadeRemote {
    @PersistenceContext
    private EntityManager em;

    public void create(Personne personne) {
        em.persist(personne);
    }

...

    public List<Personne> findAll() {
        return em.createQuery("select object(o) from Personne as o").getResultList();
    }
}
```

Lors du lancement du serveur d'applications et de l'appel de cet EJB, les traces suivantes sont affichées dans la console du serveur d'applications :

Résultat :

```
...
Creation du bean
: fr.jmdoudoux.dej.domaine.ejb.PersonneFacade@1697e2a
...
Invocation de la
methode : public java.util.List
fr.jmdoudoux.dej.domaine.ejb.PersonneFacade.findAll()
...
```

Plusieurs intercepteurs peuvent être indiqués par cette annotation : leur ordre d'exécution sera celui dans lequel ils sont précisés dans l'annotation.

Un intercepteur est toujours exécuté dans la même transaction et le même contexte de sécurité que la méthode qui est à l'origine de son invocation.

Par défaut, si l'intercepteur est défini au niveau de la classe de l'EJB, toutes les méthodes concernées de l'EJB provoqueront l'invocation de l'intercepteur par le conteneur.

Si l'intercepteur est défini au niveau d'une méthode, l'intercepteur ne sera exécuté qu'à l'invocation de la méthode annotée.

L'annotation @javax.interceptor.ExcludeClassInterceptors sur une méthode permet de demander que l'exécution des intercepteurs de type @AroundInvoke précisés dans l'annotation @Interceptors soit ignorée pour la méthode.

L'annotation @javax.interceptorExcludeDefaultInterceptors sur une classe ou une méthode permet de demander que l'exécution des intercepteurs par défaut soit ignorée.

Il est possible d'associer un intercepteur à un EJB dans le descripteur de déploiement, donc sans utiliser l'annotation @Interceptors.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns = "http://java.sun.com/xml/ns/javaee"
         version = "3.0"
         xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation = "http://java.sun.com/xml/ns/javaee
           http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd">
 <interceptors>
  <interceptor>
   <interceptor-class>fr.jmdoudoux.dej.domaine.ejb.MesurePerfIntercepteur</interceptor-class>
  </interceptor>
 </interceptors>

 <assembly-descriptor>
  <interceptor-binding>
   <ejb-name>PersonneFacade</ejb-name>
   <interceptor-class>fr.jmdoudoux.dej.domaine.ejb.MesurePerfIntercepteur</interceptor-class>
  </interceptor-binding>
 </assembly-descriptor>
</ejb-jar>
```

74.11. Les EJB de type MessageDriven

Les EJB de type MessageDriven permettent de réaliser des traitements asynchrones exécutés à la réception d'un message dans une queue JMS.

Ils ne proposent pas d'interface locale ou distante et ne peuvent pas être utilisés comme un service web. Pour connecter le bean à une queue JMS, il faut que le bean implémente l'interface javax.jms.MessageListener.

Cette interface définit la méthode onMessage(Message).

74.11.1. L'annotation @ javax.ejb.MessageDriven

L'annotation @ javax.ejb.MessageDriven permet de préciser qu'un EJB est de type MessageDriven. Elle s'utilise sur une classe qui encapsule un EJB.

L'annotation @MessageDriven possède plusieurs attributs optionnels :

Attribut	Rôle
ActivationConfigProperty[] activationConfig	Préciser les informations de configuration (type de endpoint, destination (queue ou topic), mode d'aquittement des messages, ...) sous la forme d'un tableau d'annotations de type @javac.ejb.ActivationConfigProperty (optionnel)
String description	Description de l'EJB (optionnel)
String mappedName	Nom sous lequel l'EJB sera mappé. Peut aussi être utilisé pour désigner le nom JNDI de la destination utilisée (optionnel)
Class messageListenerInterface	Préciser l'interface de type message Listener. Il faut utiliser cet attribut si l'EJB n'implémente pas d'interface ou implémente plusieurs interfaces différentes de java.io.Serializable, java.io.Externalizable ou une ou plusieurs interfaces du package javax.ejb. La valeur par défaut est Object.class (optionnel)
String name	Nom de l'EJB. La valeur par défaut est le nom non qualifié de la classe (optionnel)

74.11.2. L'annotation @javax.ejb.ActivationConfigProperty

Les paramètres nécessaires à la configuration de l'EJB notamment le type et la destination sur laquelle le bean doit écouter doivent être précisés grâce à l'attribut activationConfig. Cet attribut est un tableau d'objets de type ActivationConfigProperty.

L'annotation @javax.ejb.ActivationConfigProperty permet de préciser le nom et la valeur d'une propriété de la configuration des EJB de type MessageDriven. Elle s'utilise dans la propriété activationConfig d'une annotation de type javax.ejb.MessageDriven.

Elle possède plusieurs attributs :

Attribut	Rôle
String propertyName	Préciser le nom de la propriété (obligatoire)
String propertyValue	Préciser la valeur de la propriété (obligatoire)

Exemple :

```
@MessageDriven(mappedName = "jms/MonEJBQueue", activationConfig = {
  @ActivationConfigProperty(propertyName="acknowledgeMode", propertyValue="Auto-acknowledge"),
  @ActivationConfigProperty(propertyName="destinationType", propertyValue="javax.jms.Queue")
})
```

74.11.3. Un exemple d'EJB de type MDB

L'exemple ci-dessous est un EJB de type MessageDriven qui écoute sur une queue nommée jms/MonEJBQueue et qui affiche sur la console le contenu des messages de type texte reçus dans la queue.

Exemple :

```
package fr.jmdoudoux.dej.domaine.ejb;

import javax.annotation.Resource;
import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.ejb.MessageDrivenContext;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

@MessageDriven(mappedName = "jms/MonEJBQueue", activationConfig = {
  @ActivationConfigProperty(propertyName="acknowledgeMode", propertyValue="Auto-acknowledge"),
  @ActivationConfigProperty(propertyName="destinationType", propertyValue="javax.jms.Queue")
})
public class MonEJBMessageBean implements MessageListener {

  @Resource
  private MessageDrivenContext mdc;

  public MonEJBMessageBean() {
  }

  public void onMessage(Message message) {

    TextMessage msg = null;
    try {
      if (message instanceof TextMessage) {
        msg = (TextMessage) message;
        System.out.println("Message recu = " + msg.getText());
      }
    } catch (JMSException e) {
      e.printStackTrace();
      mdc.setRollbackOnly();
    } catch (Throwable te) {
      te.printStackTrace();
    }
  }
}
```

Il est possible d'écrire un client de test par exemple sous la forme d'une servlet. Cette servlet peut utiliser l'injection de dépendances si elle s'exécute dans le même serveur d'applications.

Exemple :

```
package fr.jmdoudoux.dej.servlet;

import java.io.*;
import java.net.*;

import javax.annotation.Resource;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.servlet.*;
import javax.servlet.http.*;

public class EnvoyerMessage extends HttpServlet {

  @Resource(mappedName = "jms/MonEJBQueue")
  Queue queue = null;

  @Resource(mappedName = "jms/MonEJBQueueFactory")
  QueueConnectionFactory factory = null;

  protected void processRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    try {
      out.println("<html>");
      out.println("<head>");
      out.println("<title>Servlet EnvoyerMessage</title>");
      out.println("</head>");
      out.println("<body>");
      out.println("<h1>Servlet EnvoyerMessage</h1>");
      out.println("<form>");
      out.println("Message : <input type='text' name='msg'><br/>");
      out.println("<input type='submit'><br/>");
      out.println("</form>");
      out.println("</body>");
      out.println("</html>");

      String msg = request.getParameter("msg");

      if (msg != null) {
        QueueConnection connection = null;
        QueueSession session = null;
        MessageProducer messageProducer = null;
        try {
          connection = factory.createQueueConnection();

          session = connection.createQueueSession(false,
                    QueueSession.AUTO_ACKNOWLEDGE);
          messageProducer = session.createProducer(queue);

          TextMessage message = session.createTextMessage();
          message.setText(msg);
          messageProducer.send(message);
          messageProducer.close();
          connection.close();
        } catch (JMSException ex) {
          ex.printStackTrace();
        }
      }
    } finally {
      out.close();
    }
  }

  protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
    processRequest(request, response);
  }

  protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
    processRequest(request, response);
  }

  public String getServletInfo() {
    return "Envoi d'un message pour test EJB de type MDB";
  }
}
```

Si le client ne s'exécute pas dans le même serveur d'applications, il faut utiliser JNDI pour obtenir la queue et la fabrique de connexions.

Exemple :

```
package fr.jmdoudoux.dej.servlet;

import java.io.*;
import java.net.*;

import javax.annotation.Resource;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.servlet.*;
import javax.servlet.http.*;

public class EnvoyerMessage extends HttpServlet {

    Queue queue = null;
    QueueConnectionFactory factory = null;

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet EnvoyerMessage</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Servlet EnvoyerMessage</h1>");
            out.println("<form>");
            out.println("Message : <input type='text' name='msg'><br/>");
            out.println("<input type='submit'><br/>");
            out.println("</form>");
            out.println("</body>");
            out.println("</html>");

            String msg = request.getParameter("msg");

            if (msg != null) {
                QueueConnection connection = null;
                QueueSession session = null;
                MessageProducer messageProducer = null;
                try {
                    InitialContext ctx = new InitialContext();
                    queue = (Queue) ctx.lookup("jms/MonEJBQueue");
                    factory = (QueueConnectionFactory) ctx.lookup("jms/MonEJBQueueFactory");

                    connection = factory.createQueueConnection();
                    session = connection.createQueueSession(false,
                            QueueSession.AUTO_ACKNOWLEDGE);
                    messageProducer = session.createProducer(queue);

                    TextMessage message = session.createTextMessage();
                    message.setText(msg);
                    messageProducer.send(message);
                    messageProducer.close();
                    connection.close();

                } catch (JMSException ex) {
                    ex.printStackTrace();
                } catch (NamingException ex) {
                    ex.printStackTrace();
                }
            }

        } finally {
            out.close();
        }
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);
    }

    public String getServletInfo() {
        return "Envoi d'un message pour test EJB de type MDB";
    }
}
```

Il est important que la queue et la fabrique de connexions soient définies dans l'annuaire du serveur d'applications.

Exemple avec GlassFish : extrait du fichier sun-resources.xml

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE resources PUBLIC "-//Sun Microsystems, Inc.
  //DTD Application Server 9.0 Resource Definitions
  //EN" "http://www.sun.com/software/appserver/dtds/sun-resources_1_3.dtd">
<resources>
...
  <admin-object-resource enabled="true" jndi-name="jms/MonEJBQueue"
    object-type="user" res-adapter="jmsra" res-type="javax.jms.Queue">
    <description/>
    <property name="Name" value="PhysicalQueue"/>
  </admin-object-resource>
  <connector-resource enabled="true" jndi-name="jms/MonEJBQueueFactory"
    object-type="user" pool-name="jms/MonEJBQueueFactoryPool">
    <description/>
  </connector-resource>
...
</resources>
```

Les EJB de type MessageDriven peuvent exploiter toutes les fonctionnalités de JMS : utilisation d'une queue ou d'un topic comme destination, utilisation des différents types de messages (TextMessage, ObjectMessage, ...)

74.12. Le packaging des EJB

Les EJB doivent être packagés dans une archive de type jar qui contiendra tous les éléments nécessaires à leur exécution.

Le fichier jar peut lui-même être incorporé dans une archive de type EAR (Enterprise Archive) qui regroupe plusieurs modules dans une même entité (EJB, application web, application client lourde, ...).

La suite de cette section sera développée dans une version future de ce document

74.13. Les transactions

Une transaction exécute une succession d'unités de traitements qui utilisent une ou plusieurs ressources, le plus souvent une base de données. Ces unités de traitements forment un ensemble d'activités qui interagit pour former un tout fonctionnel : leurs exécutions doivent toutes réussir ou aucune ne doit être exécutée.

Le but d'une transaction est de s'assurer que toutes les unités de traitements qu'elle inclut seront correctement exécutées ou qu'aucune ne le sera si un problème survient.

Une transaction permet d'assurer l'intégrité des données car soit elle s'exécute correctement dans son intégralité soit elle ne fait aucune modification.

Une transaction possède quatre caractéristiques connues sous l'acronyme ACID :

- Atomic : l'exécution doit être correcte dans son intégralité ou ne pas avoir d'effet. Chaque unité de traitement doit être exécutée sans erreur : si une erreur survient alors toutes les modifications réalisées dans les précédentes unités d'exécution doivent être annulées pour revenir à l'état initial
- Consistent : le développeur doit s'assurer que les modifications réalisées dans une transaction doivent être consistantes. Par exemple, lors d'une opération bancaire de transfert de fond entre deux comptes, le montant du débit et du crédit sur chacun des comptes doit être identique
- Isolated : les données mises à jour dans la transaction ne doivent pas être modifiées en dehors de la transaction durant son execution
- Durable : le résultat de l'exécution correcte de la transaction doit être rendu persistant

L'abandon d'une transaction ne doit donc pas simplement se limiter à son arrêt, il est aussi obligatoire d'annuler toutes les mises à jour déjà réalisées par la transaction pour permettre de laisser le système dans son état initial au lancement de la transaction.

74.13.1. La mise en oeuvre des transactions dans les EJB

Le conteneur d'EJB propose un support des transactions par déclaration ce qui évite d'avoir à mettre en oeuvre explicitement une API de gestion des transactions dans le code.

Dans les EJB, une transaction concerne une méthode d'un EJB : cette transaction inclut tous les traitements contenus dans la méthode. Ceci inclut donc aussi les appels aux méthodes d'autres EJB sous réserve de leur déclaration de participation à une transaction.

La transaction est aussi propagée au contexte de persistance assuré par les EntityManager. Si la transaction est validée, alors le contexte de persistance va rendre persistante les modifications effectuées durant la transaction.

Tous les traitements inclus dans la transaction définissent la portée de la transaction.

Lorsque la transaction est gérée par le conteneur, la décision de valider ou d'abandonner la transaction est prise par le conteneur. Une transaction est abandonnée si une exception est levée dans les traitements de la méthode ou par une des méthodes de l'EJB appelées dans ses traitements.

74.13.2. La définition de transactions

L'annotation @TransactionAttribute implémentée dans la classe javax.ejb.TransactionAttribute ou le descripteur des EJB permet de mettre en oeuvre les transactions par déclaration. Ceci transforme les caractéristiques de la transaction sans avoir à modifier le code des traitements dans la méthode de l'EJB.

74.13.2.1. La définition du mode de gestion des transactions dans un EJB

L'annotation javax.ejb.TransactionManagement permet de préciser le mode de gestion des transactions dans un EJB de type session ou message driven. Ce mode peut prendre deux valeurs :

- gestion par le container (valeur par défaut)
- gestion par le code de l'EJB

Elle s'utilise sur une classe d'un EJB session ou message driven.

Elle possède un attribut :

Attribut	Rôle
TransactionManagementType value	Préciser le mode de gestion des transactions dans l'EJB. Cet attribut peut prendre deux valeurs : • TransactionManagementType.CONTAINER (valeur par défaut) • TransactionManagementType.BEAN

Dans le cas où le mode précisé est BEAN, il est nécessaire de coder la gestion des transations dans les méthodes qui en ont besoin en utilisant l'API JTA.

74.13.2.2. La définition de transactions avec l'annotation @TransactionAttribute

L'annotation @javax.ejb.TransactionAttribute permet de préciser dans quel contexte transactionnel une méthode d'un EJB sera invoquée. Cette annotation est incompatible avec la valeur BEAN de l'annotation TransactionManagement.

Elle s'utilise sur une classe d'un EJB ou sur une méthode d'un EJB session. Utilisée sur une classe, l'annotation s'applique à toutes les méthodes de l'EJB.

Elle possède un attribut :

Attribut	Rôle
TransactionAttributeType value	Préciser le contexte transactionnel d'invocation d'une méthode de l'EJB. Cet attribut peut prendre plusieurs valeurs : • TransactionAttributeType.MANDATORY • TransactionAttributeType.REQUIRED (valeur par défaut) • TransactionAttributeType.REQUIRES_NEW • TransactionAttributeType.SUPPORTS • TransactionAttributeType.NOT_SUPPORTED • TransactionAttributeType.NEVER

L'annotation @TransactionAttribute peut prendre différentes valeurs :

- NOT_SUPPORTED : suspend la propagation de la transaction aux traitements de la méthode et des appels aux autres EJB de ces traitements. Une éventuelle transaction démarrée avant l'appel d'une méthode marquée avec cet attribut est suspendue jusqu'à la sortie de la méthode.
- SUPPORTS : la méthode est incluse dans une éventuelle transaction démarrée avant son appel. Cet attribut permet à la méthode d'être incluse ou non dans une transaction
- REQUIRED : la méthode doit obligatoirement être incluse dans une transaction. Si une transaction est démarrée avant l'appel de cette méthode, alors la méthode est incluse dans la portée de la transaction. Si aucune transaction n'est définie à l'appel de la méthode, le conteneur va créer une nouvelle transaction dont la portée concernera les traitements de la méthode et les appels aux EJB de ces traitements. La transaction prend fin à la sortie de la méthode (valeur par défaut lorsque l'annotation n'est pas utilisée ou définie dans le fichier de déploiement)
- REQUIRES_NEW : une nouvelle transaction est systématiquement démarrée même si une transaction est démarrée lors de l'appel de la méthode. Dans ce cas, la transaction existante est suspendue jusqu'à la fin de l'exécution de la méthode
- MANDATORY : la méthode doit obligatoirement être incluse dans la portée d'une transaction existante avant son appel. Aucune transaction ne sera créée et elle doit obligatoirement être fournie par le client appelant. L'appel de la méthode non incluse dans la portée d'une transaction lève une exception de type javax.ejb.EJBTransactionRequiredException
- NEVER : la méthode ne doit jamais être appelée dans la portée d'une transaction. Si c'est le cas, une exception de type EJBException est levée

Cette annotation peut être utilisée au niveau de l'EJB (dans ce cas, toutes les méthodes de l'EJB utilisent la même déclaration des attributs de transaction) ou au niveau de chaque méthode.

74.13.2.3. La définition de transactions dans le descripteur de déploiement

Les déclarations des attributs relatives aux transactions peuvent aussi être faites dans le descripteur de déploiement. Le tag <container-transaction> est utilisé pour préciser les attributs de transaction d'une ou plusieurs méthodes d'un EJB.

Pour un EJB, le tag fils <method> permet de préciser la ou les méthodes concernées. Le tag fils <ejb-name> indique l'EJB. Le tag <method-name> permet de préciser la méthode concernée ou toutes les méthodes de l'EJB en mettant * comme valeur du tag.

Le tag fils <trans-attribute> permet de préciser l'attribut de transaction à utiliser.

74.13.2.4. Des recommandations sur la mise en oeuvre des transactions

Il est fortement recommandé d'utiliser un contexte de persistance (EntityManager) dans la portée d'une transaction afin de s'assurer que tous les accès à la base de données se font dans un contexte transactionnel. Ceci implique d'utiliser les attributs de transaction Required, Requires_New ou Mandatory.

Un EJB de type MessageDriven ne peut utiliser que les attributs de transaction NotSupported et Required. L'attribut NotSupported précise que les messages ne seront pas traités dans une transaction. L'attribut Required précise que les messages seront traités dans une transaction créée par le conteneur.

Il n'est pas possible d'utiliser l'attribut Mandatory avec un EJB qui est proposé sous la forme d'un service web.

La gestion des attributs d'une transaction est importante car l'utilisation d'un EJB dans un contexte transactionnel est coûteuse en ressources. Il faut bien tenir compte du fait que la valeur par défaut des attributs de transaction est utilisée si aucun attribut n'est précisé et que cet attribut par défaut est REQUIRED, ce qui place automatiquement l'EJB dans un contexte transactionnel.

Il est donc fortement recommandé d'utiliser un attribut de transaction NotSupported lorsqu'aucune transaction n'est requise.

74.14. La mise en oeuvre de la sécurité

Les autorisations reposent en Java EE sur la notion de rôle. Un ou plusieurs rôles sont affectés à un utilisateur. L'attribution des autorisations se fait donc au niveau rôle et non au niveau utilisateur.

Même s'il est possible d'utiliser une API dédiée, généralement la mise en oeuvre de la sécurité dans les EJB se fait de manière déclarative.

Seuls les EJB de type Session peuvent être sécurisés.

Pour définir des restrictions, il faut utiliser le descripteur de déploiement ou les annotations dédiées. Ces restrictions reposent sur la notion de rôle.

Lorsqu'une méthode est invoquée et que le conteneur détecte une violation des restrictions d'accès alors ce dernier lève une exception de type javax.ejb.EJBAccessException qui devra être traitée par le client appelant.

74.14.1. L'authentification et l'identification de l'utilisateur

Lorsqu'un client utilise des fonctionnalités du conteneur d'EJB, il possède un identifiant de sécurité durant sa connexion. L'authentification de l'utilisateur est à la charge de l'application cliente.

La façon dont l'utilisateur est fourni au conteneur est dépendante de l'implémentation des EJB utilisés. Généralement cela se fait en passant des propriétés lors de la recherche du contexte JNDI.

Certains serveurs d'applications utilisent des mécanismes plus complexes et plus riches fonctionnellement en mettant en oeuvre l'API JAAS par exemple.

Lors de l'invocation des méthodes des EJB, cet identifiant de sécurité est passé implicitement à chaque appel pour permettre au conteneur de vérifier les autorisations d'utilisation par l'utilisateur.

74.14.2. La définition des restrictions

La définition des restrictions d'accès permet la mise en oeuvre des mécanismes d'autorisation.

Lorsqu'un utilisateur invoque un EJB, le conteneur contrôle les autorisations d'exécution de la méthode invoquée en comparant le ou les rôles de l'utilisateur avec le ou les rôles autorisés à exécuter cette méthode.

Le mécanisme d'autorisations est précisement défini dans les spécifications des EJB. La définition des autorisations peut être déclarée de deux façons différentes :

- utilisation des annotations dans le code de classe des EJB
- utilisation du descripteur de déploiement

74.14.2.1. La définition des restrictions avec les annotations

Par défaut, toutes les méthodes publiques d'un EJB peuvent être invoquées sans restriction de sécurité.

La définition de restrictions d'accès à un EJB se fait principalement grâce à l'annotation @javax.annotation.security.RolesAllowed qui permet de préciser les rôles qui seront autorisés à invoquer la méthode de l'EJB.

L'annotation @RolesAllowed peut s'utiliser :

- sur la classe de l'EJB : dans ce cas, cela définit les restrictions par défaut pour toutes les méthodes de l'EJB
- sur une méthode de l'EJB : dans ce cas, cela définit les restrictions pour la méthode en remplaçant les

restrictions par défaut déjà définies

L'annotation @PermitAll permet l'invocation par tout le monde : c'est l'annotation par défaut si aucune restriction n'est définie.

L'annotation @DenyAll permet d'empêcher l'invocation d'une méthode quel que soit le rôle de l'utilisateur qui l'invoque.

L'annotation @RunAs permet de forcer le rôle sous lequel l'EJB est exécuté dans le conteneur. Cette annotation ne fait aucun contrôle d'accessibilité.

74.14.2.2. La définition des restrictions avec le descripteur de déploiement

La définition de la configuration de sécurité incluant les rôles et les restrictions d'accès peut être réalisée en tout ou partie dans le descripteur de déploiement.

Les restrictions d'accès sont définies dans un tag <method-permission>

Le tag <method-permission> peut avoir plusieurs tags fils :

- un ou plusieurs tags <role-name> qui permettent de préciser un rôle autorisé à utiliser la méthode
- le tag <unchecked> qui est équivalent à l'annotation @PermitAll
- un tag <method> qui précise la ou les méthodes concernées

Le tag <method> possède plusieurs tags fils :

- <ejb-name> qui précise le nom de l'EJB concerné
- <method-name> qui précise la méthode ou toutes les méthodes en utilisant le caractère *. Remarque : il n'est pas possible de combiner l'utilisation de caractères avec le caractère *
- <method-params> qui est optionnel permet de déterminer les méthodes concernées en cas de surcharge. Chacun des paramètres est défini avec un tag <method-param> qui contient le type du paramètre
- <method-intf> qui est optionnel permet de préciser l'interface d'accès. Les valeurs possibles sont : Remote, Local, Home, LocalHome et ServicePoint
- <description> qui est optionnel permet de fournir une description

Pour empêcher l'accès à certaines méthodes, il faut utiliser le tag <exclude-list> fils du tag <assembly-descriptor>. Le tag <exclude-list> a un rôle équivalent à l'annotation @DenyAll. Chaque méthode concernée est décrite avec un tag fils <method>.

74.14.3. Les annotations pour la sécurité

Les spécifications des EJB 3.0 définissent plusieurs annotations pour gérer et mettre en oeuvre la sécurité dans les accès réalisés sur les EJB.

74.14.3.1. javax.annotation.security.DeclareRoles

L'annotation @DeclareRoles permet de définir la liste des rôles qui sont utilisés par un EJB pour sécuriser l'invocation de ses méthodes.

Cette annotation est utile pour préciser les rôles au conteneur dans le cas où les restrictions d'accès sont définies par programmation. Elle peut aussi être utilisée pour fournir explicitement au conteneur la liste des rôles implicitement définis dans les annotations @RolesAllowed.

L'annotation @DeclareRoles s'applique uniquement sur une classe. Elle ne possède qu'un seul attribut :

Attribut	Rôle

String[] value	Préciser le ou les rôles utilisés lors du contrôle d'accès à l'EJB (obligatoire)

74.14.3.2. javax.annotation.security.DenyAll

Aucun client ne peut invoquer la méthode de l'EJB qui est marquée avec cette annotation.

L'annotation @DenyAll s'applique uniquement sur une méthode. Elle ne possède pas d'attribut.

74.14.3.3. javax.annotation.security.PermitAll

L'annotation @PermitAll permet de préciser que la ou les méthodes de l'EJB n'ont aucune restriction d'accès.

L'annotation @PermitAll s'applique sur une classe ou une méthode. Elle ne possède pas d'attribut.

Cette annotation est l'annotation par défaut pour un EJB si aucune restriction d'accès n'est explicitement définie.

74.14.3.4. javax.annotation.security.RolesAllowed

L'annotation @RolesAllowed permet de préciser les rôles qui seront autorisés à invoquer une ou plusieurs méthodes d'un EJB.

L'annotation @RolesAllowed s'applique sur une classe ou une méthode.

Appliquée à une classe, cette annotation définit les restrictions d'accès par défaut de toutes les méthodes de l'EJB

Appliquée à une méthode, cette annotation définit les restrictions d'accès pour la méthode en remplaçant les éventuelles restrictions par défaut.

Elle ne possède qu'un seul attribut :

Attribut	Rôle
String[] value	Préciser le ou les rôles qui peuvent invoquer la ou les méthodes (obligatoire)

74.14.3.5. javax.annotation.security.RunAs

L'annotation @RunAs permet de préciser le rôle sous lequel un EJB va être executé dans le conteneur indépendemment du rôle de l'utilisateur qui invoque l'EJB.

L'annotation @RunAs s'utilise sur la classe d'un EJB. Elle possède un seul attribut :

Attribut	Rôle
String value	Préciser le rôle sous lequel l'EJB s'exécute (obligatoire)

Cette annotation peut être utilisée sur un EJB de type Session ou Message Driven.

74.14.4. La mise en oeuvre de la sécurité par programmation

L'interface EJBContext propose des fonctionnalités relatives à la mise en oeuvre de la sécurité.

La méthode javax.security.Principal getCallerPrincipal() permet de connaître l'utilisateur qui invoque l'EJB.

L'interface javax.security.Principal encapsule l'utilisateur qui invoque un EJB. Sa méthode getName() permet de connaître le nom de l'utilisateur.

La méthode boolean isCallerInRole() renvoie un booléen qui vaut true si l'utilisateur qui invoque l'EJB possède le rôle founi en paramètre.

Lorsque cette méthode est utilisée, il faut utiliser l'annotation @DeclareRoles en lui précisant en paramètre les rôles qui sont utilisés avec la méthode isCallerInRole(). Autrement, il faut effectuer la déclaration équivalente dans le descripteur de déploiement. Ceci permet au conteneur de savoir que ces rôles sont utilisés par l'EJB.

L'utilisation de ces méthodes permet de mettre en oeuvre des fonctionnalités d'autorisations plus pointues que la simple vérification vis-à-vis d'un rôle.

75. Les EJB 3.1

Chapitre 75

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

La versions 3.1 des EJB comme la version précédente permet le développement rapide d'objets métiers pour des applications distribuées, sécurisées, transactionnelles et portables.

Cette version apporte de nouvelles fonctionnalités (Les interfaces Local sont optionnelles pour les EJB Session, EJB Singleton, les invocations asynchrones, EJB Lite, packaging simplifié, ...) et un enrichissement de fonctionnalités existantes (Service Timer, noms JNDI portables, ...) qui permettent aux développeurs et aux architectes de répondre aux besoins de leurs applications.

Leur facilité de développement et les nouvelles fonctionnalités des EJB 3.1 leur permettent de devenir très intéressants même pour des applications de tailles moyennes voire petites.

Les EJB 3.1 sont issus des spécifications de la JSR 318.

Ce chapitre contient plusieurs sections :

- ♦ Les interfaces locales sont optionnelles
- ♦ Les EJB Singleton
- ♦ EJB Lite
- ♦ La simplification du packaging
- ♦ Les améliorations du service Timer
- ♦ La standardisation des noms JNDI
- ♦ L'invocation asynchrone des EJB session
- ♦ L'invocation d'un EJB hors du conteneur

75.1. Les interfaces locales sont optionnelles

Au moins une interface (locale ou distante) est requise pour les EJB Session 3.0

Les interfaces sont un excellent moyen pour limiter le couplage et assurer la testabilité : cependant dans certains cas, elles ne sont pas toujours nécessaires notamment si les deux points précédents ne sont pas une grande préoccupation.

Avec la version 3.1, il n'est plus nécessaire de définir une interface Local pour les EJB session : la classe de l'EJB session peut être directement annotée avec @Stateless ou @Stateful.

Rendre les interfaces pour les EJB session optionnelles permet à un EJB session d'être un simple POJO.

Exemple :

```
@Stateless
public class MonEJBBean {
}
```

Les EJB Session n'ont plus l'obligation de définir explicitement une interface Local : le conteneur peut simplement utiliser le bean qui par défaut expose toutes les méthodes publiques de la classe et de ses classes mères. Un client peut obtenir une référence sur ce bean en utilisant l'injection de dépendance ou une recherche dans l'annuaire JNDI comme pour les interfaces Local ou Remote.

Contrairement aux interfaces Local et Remote avec lesquelles la référence obtenue est du type de son interface, c'est le type du bean qui est directement obtenu en tant que référence.

L'exemple ci-dessous définit un EJB session.

Exemple :

```
package fr.jmdoudoux.dej.ejb31.domaine;

import javax.ejb.Stateless;

@Stateless
public class MonBean {

  public String saluer() {
    return "Bonjour";
  }
}
```

Ce bean ne définit aucune interface particulière. Pour l'utiliser, par exemple dans une servlet, il suffit d'utiliser l'injection de dépendance avec l'annotation @EJB sur un objet du type de la classe d'implémentation de l'EJB.

Exemple :

```
package fr.jmdoudoux.dej.ejb31.servlets;

import fr.jmdoudoux.dej.ejb31.domaine.MonBean;
import java.io.IOException;
import java.io.PrintWriter;
import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(name="MaServlet", urlPatterns={"/MaServlet"})
public class MaServlet extends HttpServlet {

  @EJB
  private MonBean monBean;

  protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    try {
      out.println("<html>");
      out.println("<head>");
      out.println("<title>Servlet MaServlet</title>");
      out.println("</head>");
      out.println("<body>");
      out.println("<h1>" + monBean.saluer() + "</h1>");
      out.println("</body>");
      out.println("</html>");
    } finally {
      out.close();
    }
  }

  @Override
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    processRequest(request, response);
  }
  @Override
  protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    processRequest(request, response);
  }

  @Override
  public String getServletInfo() {
    return "Ma servlet de test";
  }
}
```

Le fait d'utiliser le type de l'implémentation du bean comme référence impose quelques contraintes :

- lorsqu'un EJB ne définit aucune interface (ni Local ni Remote) alors le conteneur doit proposer une vue de type no-interface
- il ne faut pas utiliser l'opérateur new pour obtenir une référence sur le bean mais toujours l'obtenir par injection de dépendances
- une exception de type EJBException est levée avec le message «Illegal non-business method access on no-interface view » si une méthode non publique est invoquée

Toutes les méthodes publiques du bean et de ses classes mères sont exposées dans la vue no-interface. Ceci expose donc les méthodes de gestion du cycle de vie, ce qui peut ne pas être souhaité.

Seules les interfaces Local sont optionnelles : les interfaces Remote sont toujours obligatoires.

Il ne faut cependant pas abuser de cette fonctionnalité et la réserver à des cas d'applications simples : avec les IDE, le coût de création et de maintenance d'une interface sont négligeables et cela renforce le découplage.

75.2. Les EJB Singleton

Plusieurs fournisseurs de serveurs d'applications permettaient de n'avoir qu'une seule instance d'un EJB en offrant de préciser le nombre maximum d'instances à créer dans leur descripteur de déploiement. Cette solution n'est malheureusement pas portable puisque dépendante de l'implémentation du fournisseur.

La version 3.1 des EJB propose un nouveau type d'EJB Session nommé Singleton pour résoudre ce problème : il est possible de définir un EJB qui aura les caractéristiques du design pattern singleton : le conteneur garantit qu'une seule instance de cet EJB sera utilisable et partagée dans le conteneur.

C'est un nouveau composant qui ressemble à un EJB Session mais qui ne peut avoir qu'une seule instance dans un conteneur pour une application.

Un EJB singleton est utilisé principalement pour partager ou mettre en cache des données dans l'application. L'avantage des EJB Singleton c'est qu'ils offrent tous les services d'un EJB : sécurité, transaction, injection de dépendances, gestion du cycle de vie et intercepteurs, ...

Un EJB singleton se définit avec l'annotation @Singleton. Par défaut, toutes les méthodes d'un EJB Singleton sont thread-safe et transactionnelles.

Les EJB de type Singleton permettent d'ajouter de nouvelles fonctionnalités aux EJB :

- Exécution de code au lancement ou à l'arrêt de l'application
- Partage de données avec gestion des accès concurrents

Exemple :

```
package fr.jmdoudoux.dej.ejb31;

import java.util.HashMap;
import java.util.Map;
import javax.annotation.PostConstruct;
import javax.ejb.Singleton;
import javax.ejb.LocalBean;

@Singleton
@LocalBean
public class MonCache {

  private Map<String, Object> cache;

  @PostConstruct
  public void initialiser(){
    this.cache = new HashMap<String, Object>();
  }

  public Object get(String cle){
    return this.cache.get(cle);
  }

  public void put(String cle, Object valeur){
    this.cache.put(cle, valeur);
  }

  public void clear(){
    this.cache.clear();
  }
}
```

Le conteneur garantit qu'une seule instance sera accessible à l'application : les accès à cette instance pourront être effectués par plusieurs threads.

Il est possible d'annoter certaines méthodes pour gérer le cycle de vie notamment en utilisant les annotations @PostConstruct et @PreDestroy. Ceci peut permettre de réaliser des opérations liées au cycle de vie de l'application : ces traitements étaient uniquement réalisables avant avec l'API Servlet grâce à un ServletContextListener.

L'annotation @Startup demande l'initialisation du Singleton au lancement de l'application. Cette annotation ne permet cependant pas de préciser un ordre de lancement.

Il est toutefois possible de définir un ordre de démarrage des EJB Singleton en utilisant l'annotation @DependsOn. Le conteneur garantira alors que les EJB dépendants sont démarrés avant l'EJB annoté.

Le cycle de vie d'un EJB Singleton est géré par le conteneur. Par défaut, c'est le conteneur qui décide de l'instanciation et de l'initialisation d'un EJB Singleton. L'annotation @Startup permet de demander au conteneur d'initialiser l'EJB à l'initialisation de l'application.

Exemple :

```
package fr.jmdoudoux.dej.ejb31;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.ejb.DependsOn;
import javax.ejb.Singleton;
import javax.ejb.Startup;

@Singleton
@Startup
@DependsOn({"MonSecondBean"})
public class MonBean {

  @PostConstruct
  public void initialiser() {
    System.out.println("Initialisation MonBean");
  }

  @PreDestroy
  public void Detruire() {
    System.out.println("Destruction MonBean");
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.ejb31;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.ejb.Singleton;
import javax.ejb.LocalBean;

@Singleton
@LocalBean
public class MonSecondBean {

  @PostConstruct
  public void initialiser() {
    System.out.println("Initialisation MonSecondBean");
  }

  @PreDestroy
  public void Detruire() {
    System.out.println("Destruction MonSecondBean");
  }
}
```

Durant l'arrêt de l'application, le conteneur va supprimer l'EJB après avoir éventuellement exécuté les méthodes marquées avec l'annotation @PreDestroy.

L'état de l'EJB est maintenu par le conteneur durant toute la durée de vie de l'application : cet état n'est pas persistant à l'arrêt de l'application ou de la JVM.

La gestion des accès concurrents peut utiliser deux stratégies :

- Container Managed Concurrency (CMC) : c'est le conteneur qui gère les accès concurrents au bean. C'est la stratégie par défaut.
- Bean Managed Concurrency (BMC) : la gestion des accès concurrents est à la charge du développeur en utilisant les fonctionnalités ou les API de la plate-forme.

La stratégie est précisée par l'annotation @ConcurrencyManagement qui peut prendre deux valeurs : ConcurrencyManagementType.CONTAINER ou ConcurrencyManagementType.BEAN.

La stratégie CMC répond à la plupart des besoins : elle utilise des métadonnées pour gérer les verrous. Chaque méthode possède un verrou de type read ou write précisé par une annotation.

Un verrou de type read indique que la méthode peut être accédée par plusieurs threads en simultané. Un verrou de type write indique que la méthode ne peut être accédée que par un seul thread : les invocations des autres threads sont mises en attente jusqu'à la fin de l'exécution de la méthode et réactivées une par une.

L'annotation @Lock permet de préciser le type de verrou à utiliser : elle attend en paramètre une valeur qui peut être LockType.READ ou LockType.WRITE.

Cette annotation peut être utilisée sur une classe, une interface ou une méthode. Appliquée sur une classe, cette annotation agit comme valeur par défaut pour toutes les méthodes de la classe sauf pour les méthodes qui sont annotées avec @Lock. Le type de verrou par défaut est write.

La stratégie BMC laisse au développeur le soin de gérer par programmation la gestion des accès concurrents en utilisant notamment les opérateurs synchronized et volatile ou en utilisant l'API contenue dans le package java.util.concurrent.

Par défaut, le temps d'attente d'un thread pour invoquer une méthode de l'EJB Singleton est infini. Il est possible de définir un timeout avec l'annotation @AccessTimeout qui permet de préciser un délai maximum d'attente en millisecondes. Si ce délai est atteint sans que l'invocation ne soit réalisée alors une exception de type ConcurrentAccessTimeoutException est levée.

Exemple :

```
package fr.jmdoudoux.dej.ejb31;

import javax.ejb.AccessTimeout;
import javax.ejb.ConcurrencyManagement;
import javax.ejb.ConcurrencyManagementType;
import javax.ejb.DependsOn;
import javax.ejb.Lock;
import javax.ejb.LockType;
import javax.ejb.Singleton;
import javax.ejb.Startup;

@Singleton
@Startup
@DependsOn({"MonSecondBean"})
@ConcurrencyManagement(ConcurrencyManagementType.BEAN)
@Lock(LockType.READ)
@AccessTimeout(15000)
public class MonBean {
...
}
```

La spécification ne prend pas en compte le clustering : elle n'apporte donc aucune précision sur le support des singletons dans un cluster et sauf implémentation particulière du serveur d'applications, il y aura une instance du singleton dans chaque JVM ou l'application est déployée.

Le conteneur doit maintenir actif un EJB Singleton durant la durée de vie de l'application même s'il lève une exception dans une de ses méthodes.

75.3. EJB Lite

Le but d'EJB Lite est de proposer une version légère d'un conteneur d'EJB utilisable par exemple dans une application Java SE ou un conteneur web comme Tomcat.

L'utilisation des EJB est souvent associée avec des serveurs d'applications Java EE mais il existe des conteneurs d'EJB open source qui peuvent être embarqués comme OpenEJB, EasyBeans ou Embedded JBoss. Le concept est maintenant proposé en standard avec les EJB 3.1.

Le but d'EJB Lite est de permettre de standardiser un conteneur d'EJB embarquable et utilisable avec Java SE. Ceci doit permettre, par exemple, de réaliser des tests unitaires ou d'utiliser des EJB dans des applications desktop ou dans un conteneur web.

Une application web typique n'a pas forcément besoin des EJB de type MDB, des services Timer ou de l'appel distant d'EJB. La plupart des applications utilisent des EJB Session locaux, la persistance, l'injection et les transactions. EJB Lite tente d'offrir une solution en proposant une implémentation allégée.

Les EJB Lite sont un sous-ensemble de l'API EJB qui permet une utilisation des EJB locaux en dehors d'un conteneur EJB comme dans le Web Profile ou une application standalone. EJB Lite propose les fonctionnalités suivantes :

- support des EJB de type session (stateless, stateful et singleton)
- support des EJB avec interface local ou sans interface
- l'injection
- les intercepteurs
- la sécurité et les transactions (Container Managed Transactions et Bean Managed Transactions)

Les fonctionnalités non prises en charge dans les EJB Lite sont :

- les EJB 2.x
- l'invocation par RMI/IIOP
- les session beans avec interface Remote
- les EJB de type MDB

- le support des endpoints pour les services web
- le service Timer
- CMP / BMP

Le conteneur d'EJB emarqué propose donc un ensemble réduit de fonctionnalités qui permet à un client d'utiliser des EJB de type session sans avoir besoin d'un serveur d'applications Java EE.

Un conteneur embarqué doit au minimum supporter les API définies dans EJB Lite mais les fournisseurs peuvent ajouter à ce support tout ou partie des fonctionnalités des EJB 3.1.

Une API est proposée pour :

- initialiser et exécuter le conteneur
- obtenir le contexte du conteneur

La classe EJBContainer permet une utilisation d'un conteneur d'EJB embarqué. Elle possède plusieurs méthodes :

Méthode	Rôle
static EJBContainer createEJBContainer()	créer une nouvelle instance du conteneur et l'initialiser
Context getContext()	renvoyer un objet de type javax.naming.Context qui permet un accès à l'annuaire pour rechercher des ressources de type EJB Session
void close()	demander l'arrêt du conteneur

Généralement, il suffit d'ajouter un ou plusieurs jar dans le classpath et d'utiliser l'API pour permettre la mise en oeuvre du conteneur EJB Lite.

Les usages possibles sont nombreux notamment intégrer le conteneur dans une application standalone ou web, faciliter l'exécution de tests, ...

L'exemple suivant utilise GlassFish V3 pour mettre en oeuvre un conteneur d'EJB embarqué dans une application standalone avec des tests unitaires de l'EJB.

L'exemple contient un EJB de type Session Stateless.

Exemple :

```
package fr.jmdoudoux.dej.ejb.embedded.domaine;

import java.util.logging.Level;
import java.util.logging.Logger;
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.ejb.Stateless;

@Stateless
public class MonBean {

  private static Logger logger = Logger.getLogger(MonBean.class.getName());

  public long ajouter(int a, int b) {
    return a + b;
  }

  @PostConstruct
  public void initialiser() {
    logger.log(Level.INFO, "Initialisation instance de MonBean");
  }

  @PreDestroy
  public void detruire() {
    logger.log(Level.INFO, "Destruction instance de MonBean");
  }
}
```

Pour compiler la classe, il faut ajouter deux bibliothèques au classpath du projet :

- javax.ejb.jar contenu dans le sous-répertoire glassfish/modules de GlassFish (C:\Program Files\sges-v3\glassfish\modules par défaut)
- glassfish-embedded-static-shell contenu dans le sous-répertoire glassfish/lib/embedded de GlassFish (C:\Program Files\sges-v3\glassfish\lib\embedded par défaut)

L'application crée une instance du conteneur d'EJB embarqué qui va rechercher et déployer les EJB contenus dans le classpath. Une instance du bean est obtenue à partir de son nom JNDI et utilisée pour invoquer la méthode ajouter().

Exemple :

```
package fr.jmdoudoux.dej.ejb.embedded;

import fr.jmdoudoux.dej.ejb.embedded.domaine.MonBean;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.ejb.embeddable.EJBContainer;
import javax.naming.Context;
import javax.naming.NamingException;

public class Main {

  public static void main(String[] args) {
    EJBContainer container = EJBContainer.createEJBContainer();
    Context context = container.getContext();
    MonBean monBean;
    try {
      monBean = (MonBean) context.lookup("java:global/bin/MonBean");
      System.out.println("3+2=" + monBean.ajouter(3, 2));
    } catch (NamingException ex) {
      Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
    }
    container.close();
  }
}
```

Résultat :

```
27 déc. 2009 22:57:58 com.sun.enterprise.v3.server.AppServerStartup run
INFO: GlassFish v3 (74.2) startup time : Embedded(2508ms) startup services(316ms) total(2824ms)
27 déc. 2009 22:57:58 org.glassfish.admin.mbeanserver.JMXStartupService$JMXConnectorsStarter
Thread run
INFO: JMXStartupService: JMXConnector system is disabled, skipping.
27 déc. 2009 22:57:58 com.sun.enterprise.transaction.JavaEETransactionManagerSimplified init
Delegates
INFO: Using com.sun.enterprise.transaction.jts.JavaEETransactionManagerJTSDelegate as the
delegate
27 déc. 2009 22:57:59 AppServerStartup run
INFO: [Thread[GlassFish Kernel Main Thread,5,main]] started
27 déc. 2009 22:57:59 com.sun.enterprise.security.SecurityLifecycle <init>
INFO: security.secmgroff
27 déc. 2009 22:57:59 com.sun.enterprise.security.SecurityLifecycle onInitialization
INFO: Security startup service called
27 déc. 2009 22:57:59 com.sun.enterprise.security.PolicyLoader loadPolicy
INFO: policy.loading
27 déc. 2009 22:57:59 com.sun.enterprise.security.auth.realm.Realm doInstantiate
INFO: Realm admin-realm of classtype com.sun.enterprise.security.auth.realm.
file.FileRealm successfully created.
27 déc. 2009 22:57:59 com.sun.enterprise.security.auth.realm.Realm doInstantiate
INFO: Realm file of classtype com.sun.enterprise.security.auth.realm.file.FileRealm
successfully created.
27 déc. 2009 22:57:59 com.sun.enterprise.security.auth.realm.Realm doInstantiate
INFO: Realm certificate of classtype com.sun.enterprise.security.auth.realm.
certificate.CertificateRealm successfully created.
27 déc. 2009 22:57:59 com.sun.enterprise.security.SecurityLifecycle onInitialization
INFO: Security service(s) started successfully....
```

```
27 déc. 2009 22:58:00 com.sun.ejb.containers.BaseContainer initializeHome
INFO: Portable JNDI names for EJB MonBean : [java:global/bin/MonBean,
java:global/bin/MonBean!fr.jmdoudoux.dej.ejb.embedded.domaine.MonBean]
27 déc. 2009 22:58:00 fr.jmdoudoux.dej.ejb.embedded.domaine.MonBean initialiser
INFO: Initialisation instance de MonBean
3+2=5
27 déc. 2009 22:58:00 fr.jmdoudoux.dej.ejb.embedded.domaine.MonBean detruire
INFO: Destruction instance de MonBean
27 déc. 2009 22:58:00 org.glassfish.admin.mbeanserver.JMXStartupService shutdown
INFO: JMXStartupService and JMXConnectors have been shut down.
27 déc. 2009 22:58:00 com.sun.enterprise.v3.server.AppServerStartup stop
INFO: Shutdown procedure finished
27 déc. 2009 22:58:00 AppServerStartup run
INFO: [Thread[GlassFish Kernel Main Thread,5,main]] exiting
```

Une application Java SE peut utiliser un conteneur d'EJB embarqué qui s'exécute dans la même JVM et utilise le même classloader que celui de l'application.

Le test unitaire de l'EJB utilise également le conteneur d'EJB embarqué pour obtenir une instance de l'EJB et invoquer sa méthode ajouter() pour vérifier sa bonne exécution.

Exemple :

```
package fr.jmdoudoux.dej.ejb.embedded.domaine;

import javax.ejb.embeddable.EJBContainer;
import javax.naming.Context;
import javax.naming.NamingException;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class MonBeanTest {

  private EJBContainer container;
  private Context context;
  private MonBean monBean;

  @Before
  public void setUp() throws NamingException {
    container = EJBContainer.createEJBContainer();
    context = container.getContext();
    monBean = (MonBean) context.lookup("java:global/bin/MonBean");
  }

  @After
  public void tearDown() {
    container.close();
  }

  @Test
  public void testAjouter() throws Exception {
    int a = 3;
    int b = 2;
    long attendu = 5L;
    long resultat = monBean.ajouter(a, b);
    assertEquals("", attendu, resultat);
  }
}
```

Résultat :

```
27 déc. 2009 22:55:03 com.sun.enterprise.v3.server.AppServerStartup run
************************************************************************************
INFO: GlassFish v3 (74.2) startup time : Embedded(2711ms) startup services(331ms) total(3042ms)
27 déc. 2009 22:55:03 org.glassfish.admin.mbeanserver.JMXStartupService$JMXConnectorsStarter
Thread run
INFO: JMXStartupService: JMXConnector system is disabled, skipping.
27 déc. 2009 22:55:03 com.sun.enterprise.transaction.JavaEETransactionManagerSimplified init
```

```
Delegates
INFO: Using com.sun.enterprise.transaction.jts.JavaEETransactionManagerJTSDelegate as the
delegate
27 déc. 2009 22:55:03 AppServerStartup run
INFO: [Thread[GlassFish Kernel Main Thread,5,main]] started
27 déc. 2009 22:55:04 com.sun.enterprise.security.SecurityLifecycle <init>
INFO: security.secmgroff
27 déc. 2009 22:55:04 com.sun.enterprise.security.SecurityLifecycle onInitialization
INFO: Security startup service called
27 déc. 2009 22:55:04 com.sun.enterprise.security.PolicyLoader loadPolicy
INFO: policy.loading
27 déc. 2009 22:55:04 com.sun.enterprise.security.auth.realm.Realm doInstantiate
INFO: Realm admin-realm of classtype com.sun.enterprise.security.auth.
realm.file.FileRealm successfully created.
27 déc. 2009 22:55:04 com.sun.enterprise.security.auth.realm.Realm doInstantiate
INFO: Realm file of classtype com.sun.enterprise.security.auth.realm.file.FileRealm
successfully created.
27 déc. 2009 22:55:04 com.sun.enterprise.security.auth.realm.Realm doInstantiate
INFO: Realm certificate of classtype com.sun.enterprise.security.auth.
realm.certificate.CertificateRealm successfully created.
27 déc. 2009 22:55:04 com.sun.enterprise.security.SecurityLifecycle onInitialization
INFO: Security service(s) started successfully....
27 déc. 2009 22:55:04 com.sun.ejb.containers.BaseContainer initializeHome
INFO: Portable JNDI names for EJB MonBean : [java:global/bin/MonBean,
java:global/bin/MonBean!fr.jmdoudoux.dej.ejb.embedded.domaine.MonBean]
27 déc. 2009 22:55:05 fr.jmdoudoux.dej.ejb.embedded.domaine.MonBean initialiser
INFO: Initialisation instance de MonBean
27 déc. 2009 22:55:05 fr.jmdoudoux.dej.ejb.embedded.domaine.MonBean detruire
INFO: Destruction instance de MonBean
27 déc. 2009 22:55:05 org.glassfish.admin.mbeanserver.JMXStartupService shutdown
INFO: JMXStartupService and JMXConnectors have been shut down.
27 déc. 2009 22:55:05 com.sun.enterprise.v3.server.AppServerStartup stop
INFO: Shutdown procedure finished
27 déc. 2009 22:55:05 AppServerStartup run
INFO: [Thread[GlassFish Kernel Main Thread,5,main]] exiting
```

Le conteneur embarqué recherche les EJB à déployer dans le classpath :

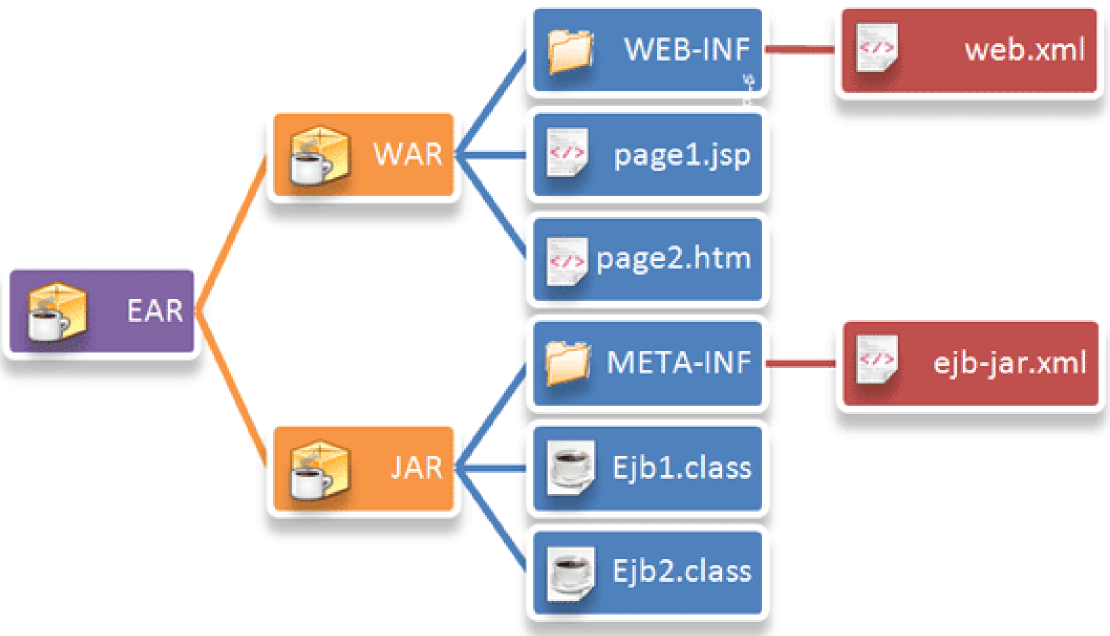
- des EJB sous la forme de classes annotées packagées dans une archive de type jar
- des EJB sous la forme de classes annotées
- le fichier ejb-jar.xml dans le sous-répertoire META-INF

L'environnement dans lequel un EJB s'exécute est transparent pour lui : le code de l'EJB est le même dans un conteneur embarqué et dans un serveur d'applications Java EE.

75.4. La simplification du packaging

Avant leur version 3.1, les EJB devaient être packagés dans une archive de type jar dédiée. Comme une application d'entreprise est généralement composée d'une partie IHM sous la forme d'une webapp packagée dans une archive de type war, il fallait regrouper les archives war et jar dans une archive de type ear.

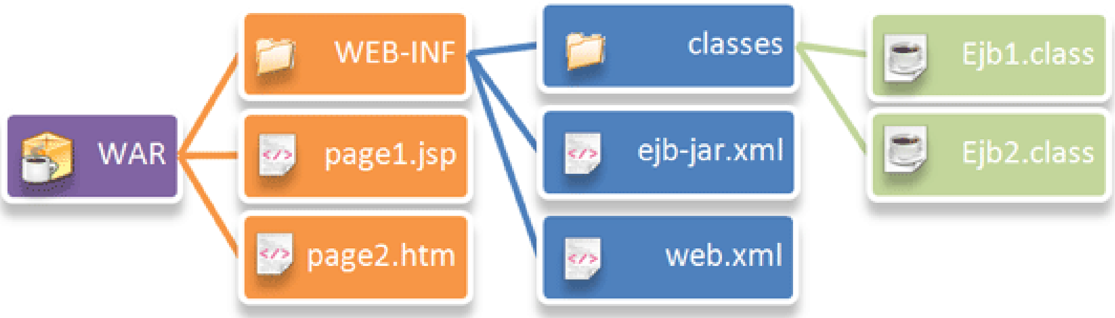
Le packaging des EJB avait été simplifié dans la version 3.0 en rendant le descripteur de déploiement optionnel. Cependant, le packaging devait toujours être fait de façon modulaire : un pour la partie web dans une archive de type war et un pour la partie EJB dans une archive de type jar, le tout regroupé dans une archive de type ear.



Ce packaging est intéressant pour rendre modulaire une grosse application mais il est complexe pour une simple application web qui utilise directement des services métiers et dont les composants n'ont pas besoin d'être partagés par plusieurs clients ou d'autres modules Java EE.

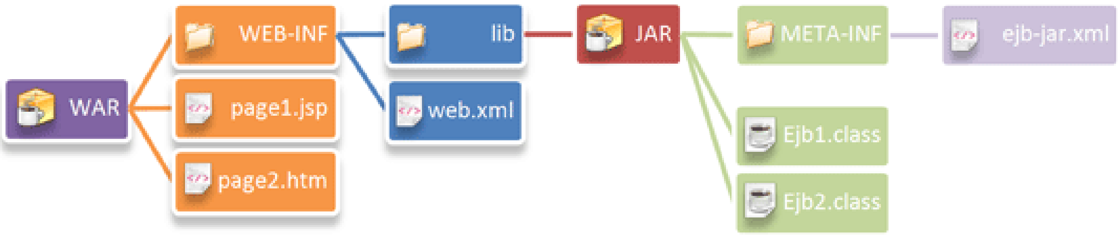
La version 3.1 propose de pouvoir intégrer les EJB directement dans la webapp sans avoir à créer un module dédié aux EJB. Les EJB qui sont des POJO annotés peuvent être mis directement dans le sous-répertoire WEB-INF/classes de la webapp et donc packagés directement dans l'archive de type war.

Si le descripteur de déploiement ejb-jar.xml doit être utilisé, il doit être placé dans le sous-répertoire WEB-INF avec le fichier web.xml.



Il est aussi possible d'ajouter un jar contenant les EJB dans le sous-répertoire WEB-INF/lib.

Une archive war ne peut contenir qu'un seul fichier ejb-jar.xml soit directement dans le sous-répertoire WEB-INF de la webapp soit dans le sous-répertoire META-INF d'une des archives jar contenues dans le sous-répertoire WEB-INF/lib



Comme les composants web et EJB sont packagés dans le même module, les ressources définies dans le war peuvent être partagées au travers de l'espace de nommage java:comp/env.

Ainsi, il est possible de définir une source de données dans le descripteur de déploiement web.xml et d'obtenir une référence par le contexte JNDI dans un EJB packagé dans le war.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
  <resource-ref>
    <description>Ma source de données</description>
    <res-ref-name>jdbc/bdd</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
    <res-sharing-scope>Shareable</res-sharing-scope>
  </resource-ref>
</web-app>
```

Il suffit alors de définir la source de données dans le conteneur ou le serveur d'applications dans lequel le war est déployé et d'utiliser JNDI pour obtenir une référence sur cette source de données.

Exemple :

```
package fr.jmdoudoux.dej.ejb31.domaine;

import java.util.logging.Level;
import java.util.logging.Logger;
import javax.ejb.Stateless;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;

@Stateless
public class MonBean {

  public String saluer() {
    Context ctx = null;
    DataSource ds = null;
    try {
      ctx = new InitialContext();
      ds = (DataSource) ctx.lookup("java:comp/env/jdbc/bdd");
      Logger.getLogger(MonBean.class.getName()).log(Level.INFO,
        "*****"+ds.getClass().getName());
      // utilisation de la source de données
      // ...
    } catch (NamingException ex) {
      Logger.getLogger(MonBean.class.getName()).log(Level.SEVERE, null, ex);
    }
    return "Bonjour";
  }
}
```

Cette possibilité est intéressante pour intégrer des EJB dans des applications existantes.

Le nouveau modèle de déploiement pourra pleinement être utilisé avec la fonctionnalité EJB Lite qui peut être mise en oeuvre dans un simple conteneur web comme Tomcat ou Jetty. C'est d'ailleurs ce qui est proposé par le Web Profile.

Cette facilité de packaging favorise l'utilisation des EJB dans les petites et moyennes applications web.

Il est cependant recommandé de réserver ce type de packaging pour des applications simples et de conserver l'usage des archives de type ear pour des applications complexes.

75.5. Les améliorations du service Timer

Il est fréquent dans une application d'entreprise d'avoir besoin de fonctionnalités pilotées par des contraintes temporelles permettant leurs déclenchements de façons régulières ou périodiques.

La version 2.1 des EJB propose le service Timer qui permet l'invocation de callbacks dans un contexte transactionnel selon des contraintes temporelles spécifiées. Ce service présente cependant quelques limitations :

- les timers doivent être créés par programmation
- la spécification des contraintes manque de flexibilité

La version 3.1 des EJB enrichit le service Timer avec :

- la possibilité de créer un timer par déclaration en utilisant l'annotation @Schedule ou le descripteur de déploiement
- l'enrichissement de l'interface TimerService pour créer un timer par programmation avec les mêmes fonctionnalités que par déclaration

Le service EJB Timer du conteneur permet de planifier l'exécution de callbacks en spécifiant un temps, une période ou un intervalle.

Le service EJB Timer propose un support de la configuration de la planification d'un timer de deux façons :

- soit de façon déclarative grâce à l'annotation @Schedule sur une méthode d'un EJB Session ou dans le descripteur de déploiement.
- soit par programmation

L'annotation @Schedule met en oeuvre une expression de façon similaire à l'utilitaire cron sous Unix pour déclarer un timer qui va exécuter les traitements de la méthode qu'elle annote à chaque fois que le timer expire.

75.5.1. La définition d'un timer

Un timer peut être défini soit automatiquement par le conteneur en utilisant des annotations ou le descripteur de déploiement soit par programmation. Les timers définis par déclaration sont automatiquement créés par le conteneur au déploiement de l'EJB.

L'annotation @Schedule s'utilise sur une méthode qui sera le callback invoqué à chaque fois que la contrainte temporelle est activée.

La méthode de callback invoquée lorsque le timeout d'un Timer est atteint peut être de deux types :

- méthode associée à un Timer instanciée via une instance de TimerService
- méthode annotée avec @Schedule ou définie dans le descripteur de déploiement

Pour définir le callback d'un timer par programmation, il y a deux solutions :

- le bean implémente l'interface javax.ejb.TimedObject
- utiliser l'annotation @Timeout

Exemple :

```
package fr.jmdoudoux.dej.ejb31;

import java.text.DateFormat;
import java.util.Date;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.annotation.PostConstruct;
import javax.ejb.Timer;
import javax.annotation.Resource;
import javax.ejb.LocalBean;
import javax.ejb.ScheduleExpression;
import javax.ejb.Singleton;
import javax.ejb.Startup;
import javax.ejb.Timeout;
import javax.ejb.TimerService;

@Singleton
@Startup
@LocalBean
public class TraitementsPeriodiques3 {

  @Resource
  TimerService timerService;

  private DateFormat mediumDateFormat =
    DateFormat.getDateTimeInstance(DateFormat.MEDIUM,DateFormat.MEDIUM);

  @PostConstruct
  public void creerTimer() {
    Logger.getLogger(TraitementsPeriodiques3.class.getName()).log(Level.INFO,
      "Creation du Timer");
    ScheduleExpression scheduleExp =
      new ScheduleExpression().second("*/10").minute("*").hour("*");
    Timer timer = timerService.createCalendarTimer(scheduleExp);
  }

  @Timeout
  public void executerTraitement(Timer timer) {
    Logger.getLogger(TraitementsPeriodiques3.class.getName()).log(Level.INFO,
      "Execution du traitement toutes les 10 secondes "+mediumDateFormat.format(new Date()));
  }
}
```

Les méthodes annotées avec @Timeout ne peuvent pas lever d'exception.

L'interface TimedObject ne définit qu'une seule méthode ejbTimeout() qui attend en paramètre un objet de type Timer encapsulant l'invocation de la méthode de callback.

Dans ce cas, une seule méthode de callback peut être définie, celle de l'interface.

Exemple :

```
package fr.jmdoudoux.dej.ejb31;

import java.text.DateFormat;
import java.util.Date;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.annotation.PostConstruct;
import javax.ejb.Timer;
import javax.annotation.Resource;
import javax.ejb.LocalBean;
import javax.ejb.ScheduleExpression;
import javax.ejb.Singleton;
import javax.ejb.Startup;
import javax.ejb.TimedObject;
import javax.ejb.Timeout;
import javax.ejb.TimerService;

@Singleton
@Startup
@LocalBean
public class TraitementsPeriodiques4 implements TimedObject {

  @Resource
  TimerService timerService;

  private DateFormat mediumDateFormat =
    DateFormat.getDateTimeInstance(DateFormat.MEDIUM,DateFormat.MEDIUM);

  @PostConstruct
  public void creerTimer() {
    Logger.getLogger(TraitementsPeriodiques4.class.getName()).log(Level.INFO,
      "Creation du Timer");
    ScheduleExpression scheduleExp =
      new ScheduleExpression().second("*/5").minute("*").hour("*");
    Timer timer = timerService.createCalendarTimer(scheduleExp);
  }

  public void ejbTimeout(Timer timer) {
    Logger.getLogger(TraitementsPeriodiques4.class.getName()).log(Level.INFO,
    "Execution du traitement toutes les 5 secondes "+mediumDateFormat.format(new Date()));
  }
}
```

Les méthodes de callbacks des Timers créés automatiquement sont soit annotées avec @Schedule ou @Schedules ou définies dans l'élément timeout-method du descripteur de déploiement.

Ces méthodes annotées de callbacks peuvent avoir deux signatures (où xxx est le nom de la méthode) :

- void xxx()
- void xxx(Timer timer)

Elles peuvent avoir n'importe quel modificateur d'accès mais ne peuvent pas être déclarées ni final ni static. Elles ne peuvent pas lever d'exception.

Comme le callback est interne à l'EJB, il ne possède aucun contexte de sécurité.

Le conteneur doit créer une nouvelle transaction si l'attribut de transaction est REQUIRED ou REQUIRED_NEW. Si la transaction échoue ou si elle est abandonnée, le conteneur doit retenter au moins une fois l'exécution du callback.

75.5.2. L'annotation @Schedule

L'annotation @Schedule permet de créer un timer dont les caractéristiques sont fournies sous la forme d'attributs de l'annotation.

La syntaxe de déclaration se fait sous la forme d'une expression dont la syntaxe est inspirée de l'outil Unix cron. Cette expression peut utiliser huit attributs :

Attribut	Valeurs possibles	Exemple
Hour	0 à 23 (heure)	hour = "23"
Minute	0 à 59 (minute)	minute = "59"
Second	0 à 59 (seconde)	second = "59"

dayOfMonth	1 à 31 : jour du mois
	last : dernier jour du mois
	-1 à -7 : nombre de jours avant la fin du mois
	{"1st", "2nd", "3rd", "4th", "5th", "Last"}
	{"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"} : identifier un jour précis dans le mois
		dayOfMonth = "1
		dayOfMonth = "last"
		dayOfMonth = "-1"
		dayOfMonth = "1st Mon"

dayOfWeek	0 à 7 : jour de la semaine (0 et 7 représentent le dimanche) {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"}	dayOfWeek = "1"
Month	1 à 12 : le mois de l'année	month = "1"

	{"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"} : le mois selon 3 première lettres	month = "Jan"
Year	Une année sur 4 chiffres	year = "2010"
timezone		

La valeur fournie à chaque attribut peut prendre différentes formes :

Forme	Description	Exemple
Une valeur simple	Une valeur unique correspondant à une des valeurs possibles de l'attribut	hour = "20"
Une étoile	Représente toutes les valeurs possibles de l'attribut	dayOfMonth = "*"
Une Liste	Représente un ensemble de valeurs possibles pour l'attribut séparées par des virgules	dayOfWeek = "Mon, Wed, Thu"
Une plage	Représente une plage de valeurs consécutives possibles pour l'attribut dont les deux bornes incluses sont séparées par un tiret	year = "2010-2019"
Une incrémentation	Définie une expression de la forme x/y où la valeur est incrémentée de y dans la plage de valeurs possibles en commençant à la valeur x. Elle ne peut être appliquée que sur heure, minute et seconde. Une fois la valeur maximale atteinte, l'incrémentation s'arrête	minute= "*/10" (toutes les 10 minutes)

Les expressions possèdent des règles et des contraintes :

- la valeur par défaut des attributs hour, minute et second est 0
- la valeur par défaut des attributs dayOfWeek, dayOfMonth, month et year est « * »
- les chaines de caractères constantes sont insensibles à la casse
- les valeurs en double dans les listes sont ignorées

Voici quelques exemples :

Expression	Description
@Schedule(hour="6", dayOfMonth="1")	Le premier de chaque mois à 6 heure du matin
@Schedule(dayOfWeek="Mon-Fri", hour="22")	Du lundi au vendredi à 10 heure du soir
@Schedule(hour = "22", minute = "30", dayOfWeek = "Fri")	Tous les vendredis à 22 heure 30
@Schedule(hour = "10, 14, 18", dayOfWeek = "Mon-Fri")	Du lundi au vendredi à 10, 14 et 18 heure
@Schedule(hour = "*", dayOfWeek = "1")	Toutes les heures de chaque lundi
@Schedule(hour = "23", dayOfMonth = "Last Fri", month="*")	Le dernier vendredi de chaque mois à 23 heure
@Schedule(hour = "22", dayOfMonth = "-3")	Trois jours avant la fin de chaque mois à 22 heure
@Schedule(minute = "*/15", hour = "12/1")	Tous les quart d'heure à partir de midi

Exemple :

```
package fr.jmdoudoux.dej.ejb31;

import java.text.DateFormat;
import java.util.Date;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.ejb.LocalBean;
import javax.ejb.Schedule;
import javax.ejb.Stateless;
```

```
@Stateless
@LocalBean
public class TraitementsPeriodiques2 {

  DateFormat mediumDateFormat =
    DateFormat.getDateTimeInstance(DateFormat.MEDIUM,DateFormat.MEDIUM);

  @Schedule(dayOfWeek="Mon")
  public void traiterHebdomadaires() {
    Logger.getLogger(TraitementsPeriodiques2.class.getName()).log(Level.INFO,
      "Execution du traitement hebdomadaire");
  }

  @Schedule(minute="*/1", hour="*")
  public void traiterMinutes() {
    Logger.getLogger(TraitementsPeriodiques2.class.getName()).log(Level.INFO,
      "Execution du traitement chaque minute "+mediumDateFormat.format( new Date()));
  }

  @Schedule(second="*/30", minute="*", hour="*")
  public void traiterTrenteSecondes() {
    Logger.getLogger(TraitementsPeriodiques2.class.getName()).log(Level.INFO,
      "Execution du traitement toutes les 30 secondes "+mediumDateFormat.format(new Date()));
  }
}
```

Résultat :

```
INFO: Execution du traitement chaque minute 31 janv. 2010 16:50:00
INFO: Execution du traitement toutes les 30 secondes 31 janv. 2010 16:50:00
INFO: Execution du traitement toutes les 30 secondes 31 janv. 2010 16:50:30
INFO: Execution du traitement toutes les 30 secondes 31 janv. 2010 16:51:00
INFO: Execution du traitement chaque minute 31 janv. 2010 16:51:00
INFO: Execution du traitement toutes les 30 secondes 31 janv. 2010 16:51:30
```

L'annotation @Schedule possède un attribut info qui permet de fournir une description du timer. Ces informations peuvent être retrouvées grâce à la méthode getInfo() de l'instance de type Timer.

Par déclaration, il est possible d'associer plusieurs timers à une même méthode de callback en utilisant l'annotation @Schedules qui agit comme un conteneur d'annotations @Schedule

Exemple :

```
@Schedules(
{ @Schedule(hour="20", dayOfWeek="Mon-Thu"),
  @Schedule(hour="18", dayOfWeek="Fri")
})
public void envoyerRapport() {
...
}
```

75.5.3. La persistance des timers

Un timer peut être persistant ou non : les timers non persistants ne survivent pas à un arrêt du conteneur.

La durée de vie d'un timer non persistant est liée à la durée de vie de la JVM qui l'a créé et dans laquelle il s'exécute : il est considéré comme supprimé en cas d'arrêt de l'application ou d'arrêt volontaire ou non de la JVM.

Les timers définis par déclaration sont par défaut persistants : le conteneur les réactive automatiquement en cas d'arrêt puis de relance.

Exemple : log de démarrage d'un serveur Glassfish v3

```
INFO: [TimerBeanContainer] Created  TimerBeanContainer: TimerBean
INFO: Portable JNDI names for EJB TimerBean : [java:global/ejb-timer-service-app/TimerBean,
java:global/ejb-timer-service-app/TimerBean!com.sun.ejb.containers.TimerLocal]
```

```
INFO: EJB5109:EJB Timer Service started successfully for datasource [jdbc/__TimerPool]
INFO: ==> Restoring Timers ...
INFO: <== ... Timers Restored.
INFO: Loading application ejb-timer-service-app at /ejb-timer-service-app
```

Un timer non persistant peut être créé de deux façons :

- par déclaration : en utilisant l'attribut persistent=false de l'annotation @Schedule
- par programmation : en utilisant la classe TimerConfig passée en paramètre de la méthode createTimer() de l'interface TimerService. Il faut fournir en paramètre de la surcharge de la méthode de création une instance de type TimerConfig pour laquelle la méthode setPersistent() a été invoquée avec la valeur false.

75.5.4. L'interface Timer

L'interface javax.ejb.Timer propose des méthodes pour annuler un timer ou obtenir des informations sur lui.

Méthode	Rôle
void cancel()	Demande la suppression du timer et de toutes ses notifications au conteneur
long getTimeRemaining()	Obtenir le nombre de millisecondes avant la prochaine notification d'expiration du Timer
Date getNextTimeout()	Obtenir la date/heure programmée de la prochaine notification d'expiration du Timer
ScheduleExpression getSchedule()	Obtenir l'objet qui définit l'expression de planification
TimerHandle getHandle()	Obtenir une version sérialisable du Timer
Serializable getInfo()	Obtenir les informations complémentaires fournies lors de la création du Timer
boolean isPersistent()	Déterminer si le Timer est persistant ou non
boolean isCalendar()	Déterminer si le Timer est basé sur un calendrier

Exemple :

```
package fr.jmdoudoux.dej.ejb31;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.GregorianCalendar;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.annotation.PostConstruct;
import javax.ejb.Timer;
import javax.annotation.Resource;
import javax.ejb.LocalBean;
import javax.ejb.Singleton;
import javax.ejb.Startup;
import javax.ejb.TimedObject;
import javax.ejb.TimerConfig;
import javax.ejb.TimerService;

@Singleton
@Startup
@LocalBean
public class TraitementsPeriodiques7 implements TimedObject {

  @Resource
  TimerService timerService;
  private DateFormat mediumDateFormat =
    DateFormat.getDateTimeInstance(DateFormat.MEDIUM, DateFormat.MEDIUM);
  @PostConstruct
  public void creerTimer() {
    Logger.getLogger(TraitementsPeriodiques7.class.getName()).log(Level.INFO,
      "Creation du Timer" + mediumDateFormat.format(new Date()));

    TimerConfig config = new TimerConfig();
    config.setInfo("donnees complementaires");

    Timer timer = timerService.createSingleActionTimer(60000, config);
  }

  public void ejbTimeout(Timer timer) {
    Logger.getLogger(TraitementsPeriodiques7.class.getName()).log(Level.INFO,
      "Execution du traitement après 60s d'attente ("+timer.getInfo()+") "
      + mediumDateFormat.format(new Date()));
  }
}
```

75.5.5. L'interface TimerService

L'interface TimerService définit les méthodes pour permettre un accès au service Timer du conteneur. Elle a été enrichie pour permettre de définir des timers par programmation.

Pour obtenir une instance de type TimerService, il faut utiliser la méthode getTimerService() de l'interface EJBContext ou demander l'injection d'une ressource de type TimerService.

Grâce aux différentes surcharges de la méthode createTimer(), l'interface TimerService propose plusieurs méthodes pour créer des instances de type Timer qui sont déclenchées :

- de façon unique en utilisant la méthode createSingleActionTimer()
- selon un intervalle en utilisant la méthode createIntervalTimer()
- ou planifiées selon un calendrier spécifié avec une expression en utilisant la méthode createCalendarTimer()

La méthode createSingleActionTimer() crée un Timer qui sera supprimé dès que son callback sera invoqué. Une version surchargée permet de préciser un délai d'attente.

Exemple :

```
package fr.jmdoudoux.dej.ejb31;

import java.text.DateFormat;
import java.util.Date;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.annotation.PostConstruct;
import javax.ejb.Timer;
import javax.annotation.Resource;
import javax.ejb.LocalBean;
import javax.ejb.Singleton;
import javax.ejb.Startup;
import javax.ejb.TimedObject;
import javax.ejb.TimerConfig;
import javax.ejb.TimerService;

@Singleton
@Startup
@LocalBean
public class TraitementsPeriodiques5 implements TimedObject {

  @Resource
  TimerService timerService;

  private DateFormat mediumDateFormat =
    DateFormat.getDateTimeInstance(DateFormat.MEDIUM,DateFormat.MEDIUM);

  @PostConstruct
  public void creerTimer() {
    Logger.getLogger(TraitementsPeriodiques5.class.getName()).log(Level.INFO,
      "Creation du Timer"+mediumDateFormat.format(new Date()));
    Timer timer = timerService.createSingleActionTimer(60000, new TimerConfig());
  }

  public void ejbTimeout(Timer timer) {
    Logger.getLogger(TraitementsPeriodiques5.class.getName()).log(Level.INFO,
      "Execution du traitement après 60s d'attente "+mediumDateFormat.format(new Date()));
  }
}
```

Une autre version surchargée permet de préciser une date/heure.

Exemple :

```
package fr.jmdoudoux.dej.ejb31;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.GregorianCalendar;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.annotation.PostConstruct;
import javax.ejb.Timer;
import javax.annotation.Resource;
import javax.ejb.LocalBean;
import javax.ejb.Singleton;
import javax.ejb.Startup;
import javax.ejb.TimedObject;
import javax.ejb.TimerConfig;
import javax.ejb.TimerService;

@Singleton
@Startup
@LocalBean
public class TraitementsPeriodiques6 implements TimedObject {

  @Resource
  TimerService timerService;
  private DateFormat mediumDateFormat =
    DateFormat.getDateTimeInstance(DateFormat.MEDIUM, DateFormat.MEDIUM);

  @PostConstruct
  public void creerTimer() {
    Logger.getLogger(TraitementsPeriodiques6.class.getName()).log(Level.INFO,
      "Creation du Timer" + mediumDateFormat.format(new Date()));
    GregorianCalendar calend =
      new GregorianCalendar(2010, GregorianCalendar.FEBRUARY, 7, 16, 45, 0);
    Timer timer = timerService.createSingleActionTimer(calend.getTime(), new TimerConfig());
  }

  public void ejbTimeout(Timer timer) {
    Logger.getLogger(TraitementsPeriodiques6.class.getName()).log(Level.INFO,
      "Execution du traitement" + mediumDateFormat.format(new Date()));
  }
}
```

La classe createCalendarTimer() permet de créer un Timer dont les conditions d'exécution son précisées par une instance de la classe ScheduleExpression fournie en paramètre.

La classe ScheduleExpression encapsule l'expression qui définit le déclenchement des traitements des Timers programmés par un calendrier.

La méthode getTimers() retourne une collection des Timers associés avec l'EJB. Il est ainsi possible d'accéder à chaque Timer pour obtenir des informations ou pour les effacer.

75.6. La standardisation des noms JNDI

Tous les EJB de type Session sont enregistrés dans un annuaire avec un nom unique accessible par un client dans un contexte JNDI que ce soit en utilisant directement le contexte (hors du conteneur) ou en utilisant l'injection de dépendance (dans le conteneur).

Ce nom JNDI est automatiquement défini par le conteneur à l'enregistrement de chaque EJB, chaque fournisseur utilisant sa propre nomenclature puisque les spécifications leur en laisse la latitude.

Cela pose des problèmes de portabilité entre différents conteneurs. C'est encore plus gênant avec l'injection de dépendances puisque le conteneur doit être capable de déterminer le nom JNDI à partir des métadonnées de l'annotation @EJB.

Cette liberté laissée au fournisseur d'implémentations sur le nom JNDI sous lequel l'EJB est désigné limite la portabilité de l'application sur différents serveurs d'applications.

Ainsi, les EJB Session d'une même application déployée dans différents conteneurs se voient déployés avec un nom JNDI différents, ce qui va nécessairement être un problème lors des invocations par le ou les clients. Ceci va à l'encontre de la philosophie de Java EE.

La spécification standardise le nom global JNDI et deux autres espaces de nommages relatifs aux différentes portées d'une application Java EE.

La standardisation du nom JNDI par la spécification permet de définir clairement comment le nom global JNDI (global JNDI name) doit être défini, résolvant ainsi les problèmes de portabilité pour retrouver des références vers des composants ou des ressources.

Ce nom JNDI est composé de façon à le rendre unique dans une instance d'un conteneur en utilisant le préfixe java:global, le nom de l'application, le nom du module, le nom du bean et le nom de l'interface sous la forme.

java:global[/<application-name>]/<module-name>/<bean-name>!<interface-name>

Partie du nom	Description	Obligatoire
application-name	Nom de l'application dans lequel l'EJB est packagé. Par défaut c'est le nom de l'archive de type ear sans son extension sauf si le nom de l'application est précisée dans le descripteur de déploiement application.xml.	Non
module-name	Nom du module dans lequel l'EJB est packagé. Par défaut, c'est le nom de l'archive de type jar ou war sans son extension sauf si le nom du module est précisé dans le fichier ejb-jar.xml par un élément module-name	Oui
bean-name	Nom du bean Par défaut, c'est le nom de la classe d'implémentation de l'EJB sauf si le nom est précisé par l'attribut name de l'annotation @Stateless, @ Stateful et @Singleton ou par l'élément bean-name du descripteur de déploiement	Oui
interface-name	Nom pleinement qualifié de l'interface sous laquelle l'EJB est exposé. Si l'EJB ne possède aucune interface (no interface view) alors c'est le nom pleinement qualifié de la classe d'implémentation qui est utilisé.	Oui

Le nom de l'application est optionnel car il n'est connu que si l'application est packagée dans une archive de type ear.

Le nom du module est déterminé à partir de l'archive jar ou war selon le format de l'archive dans laquelle l'EJB est packagé.

Le nom de l'interface n'est utile que si l'EJB implémente plusieurs interfaces (Locale et Remote) : il est inutile si l'EJB n'implémente qu'une seule interface ou aucune interface. Dans ce cas, le conteneur doit aussi associer l'EJB avec un nom

JNDI court sous la forme :

java:global[/<application-name>]/<module-name>/<bean-name>

Le conteneur a aussi l'obligation d'enregistrer l'EJB dans deux autres espaces de nommages du contexte : java:app et java:module.

L'espace de nommage java:app concerne l'application. La syntaxe est la suivante :

java:app/<module-name>/<bean-name>[!<interface-name>]

L'espace de nommage java:module concerne le module. La syntaxe est la suivante :

java:module/<bean-name>[!<interface-name>]

Ceci devrait améliorer la portabilité des applications Java EE entre différents conteneurs.

75.7. L'invocation asynchrone des EJB session

L'invocation de traitements asynchrones est relativement fréquente dans les applications d'entreprises mais jusqu'à la version 3.0 incluse des EJB aucune solution standard n'était proposée pour ce besoin.

Comme les threads ne peuvent pas être utilisés dans les EJB, une façon couramment employée pour permettre une invocation asynchrone d'un EJB est de passer par un message JMS traité par un EJB de type MDB. Cependant, le rôle principal de JMS est l'échange de messages et pas l'invocation de fonctionnalités de façon asynchrone.

De plus, cette solution n'est pas idyllique car elle ne permet pas facilement d'avoir un retour à la fin des traitements réalisés.

75.7.1. L'annotation @Asynchronous

La version 3.1 des EJB propose un support pour l'invocation asynchrone des EJB de type Session en utilisant l'annotation @Asynchronous sur la méthode de l'EJB qui contient les traitements.

Cette méthode peut retourner :

- void : dans ce cas, il n'y aura aucun retour à la fin de l'exécution des traitements et la méthode ne doit lever aucune exception puisque celles-ci ne pourraient pas être traitées
- Future<T> : dans ce cas, le client pourra avoir un contrôle sur l'état de l'exécution et obtenir la valeur de retour ou une exception levée par les traitements

L'invocation asynchrone d'EJB de type Session peut être utilisée sur tous les types d'EJB Session et avec toutes les interfaces de ces EJB.

L'annotation @Asynchronous peut être utilisée sur une méthode, une classe ou une interface.

Si l'annotation @Asynchronous est utilisée sur des méthodes alors seules ces méthodes sont invocables de façon asynchrone.

Exemple :

```
package fr.jmdoudoux.dej.ejb31;

import java.util.concurrent.Future;
import javax.ejb.AsyncResult;
import javax.ejb.Asynchronous;
import javax.ejb.Stateless;

@Stateless
```

```
public class MailEJB {

  @Asynchronous
  public Future<Boolean> envoyerAsync() {
    return new AsyncResult<Boolean>(true);
  }

  public Boolean envoyer() {
    return true;
  }
}
```

Si l'annotation @Asynchronous est utilisée sur la classe, toutes les méthodes exposées sont invocables de façon asynchrone.

Exemple :

```
package fr.jmdoudoux.dej.ejb31;

import java.util.concurrent.Future;
import javax.ejb.AsyncResult;
import javax.ejb.Asynchronous;
import javax.ejb.Stateless;

@Stateless
@Asynchronous
public class MailEJB {

  public Future<Boolean> envoyer() {
    return new AsyncResult<Boolean>(true);
  }

  public Future<Boolean> envoyerAvecCopie() {
    return new AsyncResult<Boolean>(true);
  }
}
```

Il est aussi possible d'utiliser l'annotation @Asynchronous sur une interface. Dans ce cas, les méthodes invocables de façon asynchrone seront celles précisées par l'interface puisque l'EJB sera invoqué au travers de son interface.

Exemple :

```
package fr.jmdoudoux.dej.ejb31;

import javax.ejb.Local;

@Local
public interface MailEJBLocal {

  Boolean envoyer();
}
```

Exemple :

```
package fr.jmdoudoux.dej.ejb31;

import java.util.concurrent.Future;
import javax.ejb.Asynchronous;
import javax.ejb.Remote;

@Remote
public interface MailEJBRemote {

  @Asynchronous
  Future<Boolean> envoyerAsync();
}
```

Exemple :

```
package fr.jmdoudoux.dej.ejb31;

import java.util.concurrent.Future;
import javax.ejb.AsyncResult;
import javax.ejb.Stateless;

@Stateless
public class MailEJB implements MailEJBRemote, MailEJBLocal {

  public Future<Boolean> envoyerAsync() {
    return new AsyncResult<Boolean>(true);
  }

  public Boolean envoyer() {
    return true;
  }
}
```

L'annotation @Asynchronous peut aussi être utilisée sur un EJB de type Singleton.

75.7.2. L'invocation d'une méthode asynchrone

Lors de l'invocation de la méthode annotée avec @Asynchronous, le client poursuit l'exécution de ses traitements sans attendre la fin de l'exécution de l'invocation.

C'est le conteneur qui garantit que les traitements seront exécutés de façon asynchrone.

L'invocation asynchrone est faite par le client (EJB, application standalone, ...) de façon transparente pour lui.

Exemple :

```
package fr.jmdoudoux.dej.ejb31;

import java.util.logging.Level;
import java.util.logging.Logger;
import javax.ejb.EJB;
import javax.ejb.Stateless;
import javax.jws.WebMethod;
import javax.jws.WebService;

@Stateless
@WebService
public class CommandeEJB implements CommandeEJBLocal, CommandeEJBRemote {

  @EJB
  MailEJBLocal mailEJB;

  @WebMethod
  public void valider(int id) {

    // traitement de validation de la commande
    Logger.getLogger(CommandeEJB.class.getName()).log(Level.INFO,
      "validation de la commande numero "+id);

    // envoie d'un mail de prise en compte
    mailEJB.envoyerAsync(id);

    Logger.getLogger(CommandeEJB.class.getName()).log(Level.INFO,
      "fin de la validation de la commande");
  }
}
```

La classe java.util.concurrent.Future<T>, disponible depuis la version 5 de Java SE, permet d'avoir un contrôle sur l'invocation asynchrone d'un traitement. Elle est typée avec le type de la valeur de retour à l'issue de l'exécution des traitements.

L'interface Future<V> définit plusieurs méthodes :

Méthode	Rôle
boolean cancel(boolean)	Demander une tentative d'annulation de l'exécution des traitements. Le conteneur va tenter d'annuler l'invocation si celle-ci n'a pas encore commencé. La méthode renvoie true si l'invocation a pu être annulée. Le paramètre permet de demander au conteneur d'informer le bean de la demande d'annulation si celui-ci est déjà en cours d'exécution
V get() V get(long, TimeUnit)	Renvoyer la valeur de retour des traitements Cette méthode possède deux surcharges : • sans paramètres : attend jusqu'à la fin des traitements • avec un timeout en paramètre : attend jusqu'à la durée du timeout puis tente de récupérer la valeur de retour
boolean isCancelled()	Préciser si l'exécution des traitements a été annulée
boolean isDone()	Préciser si l'exécution des traitements est terminée

La classe javax.ejb.AsyncResult<V> est une implémentation fournie en standard de l'interface Future<V> qui propose notamment un constructeur attendant la valeur de retour de type V en paramètre.

La méthode wasCancelCalled() de l'interface SessionContext renvoie true si le client a invoqué la méthode Future.cancel() avec la valeur true en paramètre.

Exemple :

```
package fr.jmdoudoux.dej.ejb31;

import java.util.Date;
import java.util.concurrent.Future;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.annotation.Resource;
import javax.ejb.AsyncResult;
import javax.ejb.Asynchronous;
import javax.ejb.SessionContext;
import javax.ejb.Stateless;

@Stateless
public class MailEJB implements MailEJBRemote, MailEJBLocal {

  @Resource
  SessionContext ctx;

  @Asynchronous
  public Future<Boolean> envoyerAsync(int valeur) {
    boolean resultat = ( (valeur % 2) == 0);

    Logger.getLogger(MailEJB.class.getName()).log(Level.INFO,
      "debut de l'envoi du mail "+new Date());

    long i = 0;
    while ( i < 300000000 && !ctx.wasCancelCalled() ) {
      // code des traitements a executer
      i++;
    }
    if (ctx.wasCancelCalled()) {
      resultat = false;
    }

    Logger.getLogger(MailEJB.class.getName()).log(Level.INFO,
      "fin de l'envoi du mail "+new Date());

    return new AsyncResult<Boolean>(resultat);
  }

  public Boolean envoyer() {
    return true;
  }
}
```

L'exemple ci-dessus effectue un traitement qui peut être interrompu par le client. La méthode envoyerAsync() renvoie un booléen qui indique le succès des traitements : elle renvoie false si l'id fournie est impaire ou si les traitements ont été interrompus par le client.

La méthode get() de l'interface Future peut lever une exception de type ExecutionException qui va encapsuler une éventuelle exception levée par la méthode exécutée de façon asynchrone. L'exception originale est chaînée et donc accessible en utilisant la méthode getCause().

Exemple :

```
...
    @Action
    public void invocationOk() {
      executerTraitement(2, false);
    }

    @Action
    public void InvocationKo() {
      executerTraitement(1, false);
    }

    @Action
    public void invocationCancel() {
      executerTraitement(2, true);
    }

    public void executerTraitement(int valeur, boolean arret) {
      try {
        Future<Boolean> future = bean.envoyerAsync(valeur);
        if (arret) {
          Thread.sleep(1000);
          future.cancel(true);
        }
        Boolean resultat = future.get();
        jTextArea1.setText("resultat="+resultat+
          " isCancelled="+future.isCancelled());
        Logger.getLogger(Main.class.getName()).log(Level.SEVERE,
          "résultat="+resultat+ " isCancelled="+future.isCancelled());
      } catch (InterruptedException ex) {
        Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
      } catch(ExecutionException ee) {
        Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ee);
      } catch (Exception e) {
        Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, e);
      }
    }
...
```

L'exemple ci-dessus est un extrait de code d'une application cliente Swing qui permet d'invoquer l'EJB de façon asynchrone et de tester les différents cas d'utilisation.

Le contexte de sécurité est utilisé comme lors de l'appel synchrone de la méthode.

Par contre, le contexte transactionnel n'est pas propagé à l'invocation asynchrone d'une méthode. Si la méthode invoquée est marquée avec l'attribut REQUIRES alors une nouvelle transaction est créée (comme si l'attribut REQUIRES_NEW avait été utilisé). Si la méthode invoquée est marquée avec l'attribut SUPPORT alors aucune transaction n'est utilisée. Si la méthode invoquée est marquée avec l'attribut MANDATORY alors une exception de type TransactionRequiredException est toujours levée.

75.8. L'invocation d'un EJB hors du conteneur

Dans l'exemple ci-dessous, une application standalone va invoquer un EJB déployé dans un serveur d'applications GlassFish v3.

Exemple :

```
package fr.jmdoudoux.dej.ebj31.client;

import fr.jmdoudoux.dej.ejb31.CommandeEJBRemote;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Main {

  public static void main(String[] args) {
    Context ctx = null;
    CommandeEJBRemote bean = null;
    try {
      ctx = new InitialContext();
      bean = (CommandeEJBRemote) ctx.lookup("fr.jmdoudoux.dej.ejb31.CommandeEJBRemote");
      bean.valider(1234);
    } catch (NamingException ex) {
      Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
      System.exit(1);
    }
    Logger.getLogger(Main.class.getName()).log(Level.INFO, "Fin de l'application");
  }
}
```

Le nom JNDI de l'EJB est indiqué dans les logs au moment du déploiement de l'EJB dans le conteneur : il est impératif de prendre son interface Remote puisque le client ne s'exécute pas dans le contexte du serveur d'applications.

Il faut ajouter au classpath la bibliothèque qui contient l'interface de l'EJB et ajouter le fichier gf-client.jar contenu dans le sous-répertoire modules du répertoire d'installation de GlassFish v3.

La bibliothèque gf-client.jar contient les valeurs des paramètres par défaut pour permettre un accès à l'annuaire en utilisant JNDI.

Si une exception de type java.net.ConnectException est levée à l'exécution, il faut préciser le port sur lequel l'application peut contacter l'annuaire grâce à JNDI : le plus simple est de définir la propriété org.omg.CORBA.ORBInitialPort de la JVM

Résultat :

```
-Dorg.omg.CORBA.ORBInitialPort=42382
```

La valeur à utiliser est contenue dans les logs de démarrage du serveur.

76. Les services web de type Soap

Chapitre 76

Niveau : 🟩🟩🟨🟨🟥⬜ Confirmé

Les services web de type Soap permettent l'appel d'une méthode d'un objet distant en utilisant un protocole web pour le transport (http en général) et XML pour formater les échanges. Les services web fonctionnent sur le principe client / serveur :

- un client appelle les services web
- le serveur traite la demande et renvoie le résultat au client
- le client utilise le résultat

L'appel de méthodes distantes n'est pas une nouveauté mais la grande force des services web est d'utiliser des standards ouverts et reconnus notamment HTTP et XML. L'utilisation de ces standards permet d'écrire des services web dans plusieurs langages et de les utiliser sur des systèmes d'exploitation différents.

Les services web de type Soap utilisent des messages au format XML pour permettre l'appel de méthodes ou l'échange de messages.

Certaines fonctionnalités complémentaires mais généralement utiles des services web ne sont pas encore complètement matures à cause de la jeunesse des technologies utilisées pour les mettre en oeuvre. Il reste encore de nombreux domaines à enrichir (sécurité, gestion des transactions, workflow, ...). Des technologies pour répondre à ces besoins sont en cours de développement mais généralement plusieurs solutions sont en concurrence.

Initialement, Sun a proposé un ensemble d'outils et d'API pour permettre le développement de services web avec Java. Cet ensemble se nomme JWSDP (Java Web Services Developer Pack) dont il existe plusieurs versions.

Depuis, Sun a intégré la plupart de ces API permettant le développement de services web dans les spécifications de J2EE version 1.4.

La version 5 de Java EE et la version 6 de Java SE utilisent JAX-WS 2.0 pour faciliter le développement de services web en utilisant des annotations.

Ce chapitre contient plusieurs sections :

- La présentation des services web
- Les standards
- Les différents formats de services web SOAP
- Des conseils pour la mise en oeuvre
- Les API Java pour les services web
- Les implémentations des services web
- Inclure des pièces jointes dans SOAP
- WS-I
- Les autres spécifications

76.1. La présentation des services web

Les services web sont des composants distribués qui offrent des fonctionnalités aux applications au travers du réseau en utilisant des standards ouverts. Ils peuvent donc être utilisés par des applications écrites dans différents langages et exécutées dans différentes plates-formes sur différents systèmes.

Les services Web utilisent une architecture distribuée composée de plusieurs ordinateurs et/ou systèmes différents qui communiquent sur le réseau. Ils mettent en oeuvre un ensemble de normes et standards ouverts qui permettent aux développeurs d'implémenter des applications distribuées internes ou externes en utilisant des outils différents fournis par les fournisseurs.

Un service web permet généralement de proposer une ou plusieurs fonctionnalités métiers qui seront invoquées par un ou plusieurs consommateurs.

Il existe deux grandes familles de services web :

- les services web de type SOAP
- les services web de type REST

Ce chapitre va se concentrer sur les services web de type SOAP.

76.1.1. La définition d'un service web

Il existe plusieurs définitions pour les services web mais la plus simple pourrait être "fonctionnalité utilisable au travers du réseau en mettant en oeuvre un format standard utilisant généralement XML".

Les services web ne sont donc qu'une nouvelle forme d'échanges de type RPC (Remote Procedure Call). Leur grand intérêt est de reposer sur des standards plutôt que sur des protocoles propriétaires. Par exemple, le transport repose généralement sur le protocol HTTP mais il est possible d'utiliser d'autres protocoles tels que JMS, FTP ou SMTP.

Les services web de type Soap font un usage intensif de XML, des namespaces XML et des schémas XML. Ces technologies font la force des services web pour permettre leur utilisation par des clients et des serveurs hétérogènes. XML est notamment utilisé pour stocker et organiser les informations de la requête et de la réponse mais aussi pour décrire le service web. L'utilisation de XML pour le format des messages rend les échanges indépendants du système d'exploitation, de la plate-forme et du langage.

Il est ainsi possible de développer des services web avec une plate-forme (par exemple Java) et d'utiliser ces services web avec une autre plate-forme (par exemple .Net ou PHP) : c'est une des grandes forces des services web même si cela reste parfois quelque peu théorique, essentiellement à cause des implémentations des moteurs utilisés pour mettre en oeuvre les services web.

Un service web est donc une fonctionnalité accessible au travers du réseau grâce à des messages au format XML. Le format de ces messages est généralement SOAP bien que d'autres formats existent (REST, XML-RPC, ...).

Les services web peuvent prendre plusieurs formes :

- métier
- technique
- ...

Lors de la mise en place de services web, plusieurs problématiques interviennent tôt ou tard :

- choix des spécifications mises en oeuvre
- choix des outils
- traitements proposés par les services
- administration des services
- orchestration des services
- ...

L'appel à un service web de type SOAP suit plusieurs étapes :

1. le client instancie une classe de type proxy encapsulant le service Web XML.
2. le client invoque une méthode du proxy.
3. le moteur SOAP sur le client crée le message à partir des paramètres utilisés pour invoquer la méthode
4. le moteur SOAP envoie le message SOAP au serveur généralement en utilisant le protocol HTTP
5. le moteur SOAP du serveur réceptionne et analyse le message SOAP
6. le moteur fait appel à la méthode de l'objet correspondant à la requête SOAP
7. le moteur SOAP sur le serveur crée le message réponse à partir de la valeur de retour
8. le moteur SOAP envoie le message SOAP contenant la réponse au client généralement en utilisant le protocole http
9. le moteur SOAP du client réceptionne et analyse le message SOAP
10. le moteur SOAP du client instancie un objet à partir du message SOAP contenant la réponse

Un des intérêts des services web est de masquer aux développeurs la complexité de l'utilisation des standards sous-jacents. Ceci est réalisé grâce aux développements d'API et de moteurs pour la production et la consommation de services web.

Ces API sont dépendantes des plates-formes utilisées (Java, .Net, PHP, Perl, ...) mais elles mettent toutes en oeuvre avec plus ou moins de complétude les standards de l'industrie relatifs aux services web notamment SOAP et WSDL.

Ainsi les développeurs peuvent se concentrer sur l'écriture des traitements proposés par les services et par leur consommation sans se soucier de la tuyauterie sous-jacente. Un minimum de compréhension est cependant nécessaire pour bien appréhender les mécanismes mis en oeuvre.

76.1.2. Les différentes utilisations

Les services web proposent un mécanisme facilitant :

- la communication entre applications hétérogènes : un service web développé dans une technologie peut être consommé par une application développée dans une autre technologie. Ceci est possible car les services web reposent sur des standards ouverts
- l'exposition de fonctionnalités métiers aux applications internes mais aussi à des applications externes : dans ce dernier cas l'utilisation du protocole HTTP permet facilement de passer les pare-feux
- la mise en oeuvre d'une architecture SOA puisque les services web peuvent être une implémentation possible d'une telle architecture

L'utilisation de services web peut avoir plusieurs intérêts :

- l'exposition de fonctionnalités au travers du réseau : les traitements des opérations des services web peuvent être invoqués par une requête HTTP, ce qui peut permettre à plusieurs applications de consommer ces services web
- la communication entre des applications et des systèmes hétérogènes : l'utilisation de standards ouverts permet la production et la consommation des services web par différentes technologies sur différents systèmes d'exploitation
- la mise en oeuvre des protocoles standards de l'industrie au niveau des couches transport, messaging, description et recherche permet de choisir entre plusieurs implémentations proposées et ainsi de ne pas dépendre d'un seul fournisseur
- les échanges se font en utilisant l'infrastructure existante puisque les services web sont généralement invoqués en utilisant le protocole HTTP. Ceci permet de facilement passer un firewall pour permettre une invocation depuis l'extérieur
- les services web permettent un couplage faible entre les fonctionnalités exposées et les applications qui les utilisent à tel point que les consommateurs et les producteurs peuvent être écrits pour des plates-formes ou des langages différents (Java, .Net, PHP, ...).
- les services permettent de définir de nouvelles opportunités de business voire même de nouveaux modèles économiques en permettant de proposer des fonctionnalités à des partenaires par exemple

76.2. Les standards

L'intérêt des services web grandissant, des standards ont été développés pour assurer les besoins nécessaires à leur mise en oeuvre.

L'architecture des services web est composée de quatre grandes couches utilisant plusieurs technologies :

- découverte : cette couche représente un annuaire dans lequel il est possible de publier des services et de les rechercher (UDDI est le standard)
- description : cette couche normalise la description de l'interface publique d'un service web en utilisant WSDL (Web Service Description Language)
- communication : cette couche permet d'encoder les messages échangés (SOAP est le standard)
- transport : cette couche assure le transport des messages : généralement HTTP est mis en oeuvre mais d'autres protocoles peuvent être utilisés (SMTP, FTP, ...)

La description d'un service web permet à son consommateur de connaître l'interface du service.

La communication permet de formaliser le format des messages échangés.

En plus de SOAP, WSDL et UDDI, il existe de nombreuses autres spécifications plus ou moins standard pour permettre la mise en oeuvre de fonctionnalités manquantes dans ces standards comme la sécurité, la gestion des transactions, l'orchestration des services, ...

Ces spécifications sont en cours de développement ou d'évolution ce qui les rend généralement immatures. De plus, fréquemment, il existe plusieurs spécifications ayant trait à un même sujet qui sont donc concurrentes. La mise en oeuvre de ces spécifications n'est pas requise pour des services web basiques mais elle peut être nécessaire pour des besoins plus spécifiques.

76.2.1. SOAP

SOAP (acronyme de Simple Object Acces Protocol jusqu'à sa version 1.1) est un standard du W3C qui permet l'échange formaté d'informations entre un client et un serveur. SOAP peut être utilisé pour la requête et la réponse de cet échange.

SOAP assure la partie messaging dans l'architecture des services web : il est utilisé pour normaliser le format des messages échangés entre le consommateur et le fournisseur de services web. SOAP est donc un protocole qui est principalement utilisé pour dialoguer avec des objets distribués comme peut le proposer JRMP utilisé par RMI, DCOM ou IIOP.

Son grand intérêt est d'utiliser XML ce qui le rend ouvert contrairement aux autres protocoles qui sont propriétaires : cela permet la communication entre un client et un serveur utilisant des technologies différentes. SOAP fait un usage intensif des espaces de nommages (namespaces).

SOAP est défini pour être indépendant du protocole de transport utilisé pour véhiculer le message. Cependant, le protocole le plus utilisé avec SOAP est HTTP car c'est un des protocoles le plus répandu et utilisé du fait de sa simplicité. Son utilisation avec SOAP permet de rendre les services web plus interopérables. De plus, cela permet aux services web de facilement traverser les firewalls du côté producteur et consommateur notamment dans le cas d'échanges par internet.

D'autres protocoles peuvent être utilisés (par exemple SMTP ou FTP) mais leur configuration sera plus délicate car elle ne sera pas fournie en standard comme c'est le cas avec HTTP. En fait, tous les protocoles capables de véhiculer un flux d'octets peuvent être utilisés.

SOAP est aussi indépendant de tout système d'exploitation et de tout langage de programmation car il utilise XML. Ceci permet une exposition et une consommation de services web avec des outils et des OS différents.

SOAP peut être utilisé pour :

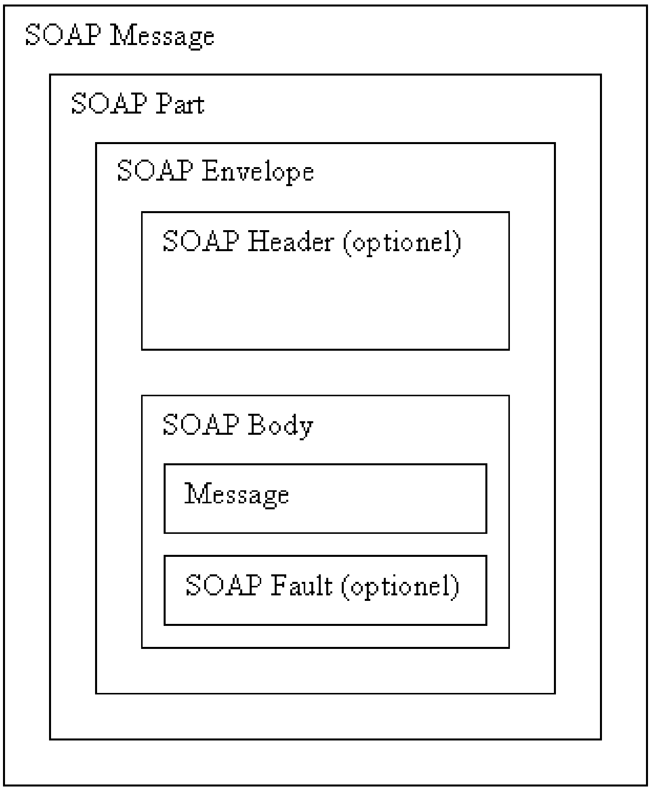
- appeler une méthode d'un service (SOAP RPC)
- echanger un message avec un service (SOAP Messaging)
- recevoir un message d'un service (selon la version de Soap)

76.2.1.1. La structure des messages SOAP

Un message SOAP est contenu dans une enveloppe, ainsi le tag racine d'un document SOAP est le tag <Envelope>.

La structure d'une enveloppe SOAP se compose de plusieurs parties :

- un en-tête optionnel composé d'un ou plusieurs headers : il contient des informations sur le traitement du message
- un corps (Body) : il contient les informations de la requête ou de la réponse
- une gestion d'erreurs optionnelle (Fault) contenue dans le corps
- des pièces jointes optionnelles (attachment) contenues dans le corps



L'en-tête contient des informations sur le traitement du message : ces informations sont contenues dans un tag <Header>. Pour des services web simples, cette partie peut être vide ou absente mais pour des services plus complexes elle peut contenir des informations concernant les transactions, la sécurité, le routage, etc ...

Le corps du message SOAP est contenu dans un tag <Body> obligatoire. Il contient les données échangées entre le client et le service sous la forme d'un fragment de document XML.

Tous ces éléments sont codés dans le message XML avec un tag particulier mettant en oeuvre un espace de nommage particulier défini dans les spécifications de SOAP.

Un message SOAP peut aussi contenir des pièces jointes placées chacune dans une partie optionnelle nommée AttachmentPart. Ces parties appartiennent à la partie SOAP Part.

SOAP définit aussi l'encodage pour les différents types de données qui est basé sur la technologie schéma XML du W3C. Les données peuvent être de type simple (chaine, entier, flottant, ...) ou de type composé.

Les types simples peuvent être

- un type de base : string, int, float, ...
- une énumération
- un tableau d'octets (array of bytes)

Les types composés

- une structure (Struct)
- un tableau (Array)

La partie SOAP Fault permet d'indiquer qu'une erreur est survenue lors des traitements du service web. Cette partie peut être composée de 4 éléments :

- faultcode : indique le type de l'erreur (VersionMismatch en cas d'incompatibilité avec la version de SOAP utilisée, MustUnderstand en cas de problème dans le header du message, Client en cas de manque d'informations de la part du client, Server en cas de problème d'exécution des traitements par le serveur)
- faultstring : message décrivant l'erreur
- faultactor : URI de l'élément ayant déclenché l'erreur
- faultdetail

76.2.1.2. L'encodage des messages SOAP

Deux formats de messages SOAP sont définis :

- remote procedure call : (RPC) permet l'invocation d'opérations qui peuvent retourner un résultat.
- message oriented (Document) : données au format XML définies dans un schéma XML

Les règles d'encodage (Encoding rules) précisent les mécanismes de sérialization des données dans un message. Il existe deux types :

- Encoded : les paramètres d'entrée de la requête et les données de la réponse sont encodées en XML dans le corps du message selon un format particulier à SOAP
- Literal : les données n'ont pas besoin d'être encodées de façon particulière : elles sont directement encodées en XML selon un schéma défini dans le WSDL

Le style et le type d'encodage permettent de définir comment les données seront sérialisées et désérialisées dans les requêtes et les réponses.

La combinaison du style et du type d'encodage peut prendre plusieurs valeurs :

- RPC/Encoded
- RPC/Literal
- Document Encoded : cette combinaison n'est pas implémentée
- Document/Literal
- Wrapped Document/Literal : extension du Document/Literal proposé par Microsoft

Le style RPC/Encoded a largement été utilisé au début des services web : actuellement ce style est en cours d'abandon par l'industrie au profit du style Document/Literal. C'est pour cette raison que le style RPC/Encoded n'est pas intégré dans le WS-I Basic Profile 1.1.

Le style et le type d'encodage sont précisés dans le WSDL. L'appel du service web doit obligatoirement se faire dans le style précisé dans le WSDL puisque celui-ci détermine le format des messages échangés.

76.2.1.3. Les différentes versions de SOAP

Les versions de SOAP

- 1.0 :
- 1.1 :
- 1.2 : permet l'utilisation de requêtes HTTP de type GET

Les spécifications de SOAP 1.2 sont composées de plusieurs parties :

- Part 0 : Primer
- Part 1 : Messaging Framework

- Part 2 : Adjuncts
- Specification Assertions and Test Collection

La version 1.2 des spécifications de SOAP est plus précise pour réduire les ambigüités qui pouvaient conduire à des problèmes d'interopérabilité entre différentes implémentations.

SOAP 1.2 propose un support pour des protocoles de transport différents de HTTP. La sérialisation de messages n'est pas obligatoirement en XML mais peut utiliser des formats binaires (XML Infoset par exemple).

76.2.2. WSDL

WSDL (acronyme de Web Service Description Language) est utilisé pour fournir une description d'un service web afin de permettre son utilisation. C'est une recommandation du W3C.

Pour permettre à un client de consommer un service web, ce dernier a besoin d'une description détaillée du service avant de pouvoir interagir avec lui. Un WSDL fournit cette description dans un document XML. WSDL joue un rôle important dans l'architecture des services en assurant la partie description : il contient toutes les informations nécessaires à l'invocation du service qu'il décrit.

La description WSDL d'un service web comprend une définition du service, les types de données utilisés notamment dans le cas de types complexes, les opérations utilisables, le protocole utilisé pour le transport et l'adresse d'appel.

C'est un document XML qui décrit un service web de manière indépendante de tout langage. Il permet l'appel de ses opérations et l'exploitation des réponses (les paramètres, le format des messages, le protocole utilisé, ...).

WSDL est conçu pour être indépendant de tout protocoles. Ceci rend le standard WSDL flexible mais aussi plus complexe à comprendre. Comme SOAP et HTTP sont les deux protocoles les plus couramment utilisés pour implémenter les services web, le standard WSDL intègre un support de ces deux protocoles.

L'utilisation de XML permet à des outils de différents systèmes, plates-formes et langages d'utiliser le contenu d'un WSDL pour générer du code permettant de consommer un service web. Les moteurs SOAP proposent en général un outil qui va lire le WSDL et générer les classes requises pour utiliser un service web avec la technologie du moteur SOAP. Le code généré utilise un moteur SOAP qui masque toute la tuyauterie du protocole utilisé et des messages échangés lors de la consommation de services web en agissant comme un proxy. Par exemple, Axis propose l'outil WSDL2Java pour la génération de ces classes à partir du WSDL.

Pour assurer une meilleure interopérabilité, WS-I Basic Profil 1.0 oblige à utiliser WSDL et les schémas XML pour la description des services web.

76.2.2.1. Le format général d'un WSDL

Un document WSDL définit plusieurs éléments :

- Type : la définition des types de données utilisés
- Message : la définition de la structure d'un message en lui attribuant un nom et en décrivant les éléments qui le composent avec un nom et un type
- PortType : la description de toutes les opérations proposées par le service web (interface du service) et identification de cet ensemble avec un nom
- Operation : la description d'une action proposée par le service web notamment en précisant les messages en entrée (input) et en sortie (output)
- Binding : la description du protocole de transport et d'encodage utilisé par un PortType afin de pouvoir invoquer un service web
- Port : référence un Binding (généralement cela correspond à l'url d'invocation du service web)
- Service : c'est un ensemble de ports

Un WSDL est un document XML dont le tag racine est <definitions> et qui utilise l'espace de nommage "http://schemas.xmlsoap.org/wsdl/".

Un WSDL est virtuellement composé de deux parties :

- des définitions abstraites : celles-ci concernent l'interface du service (types, message, portType). Ces informations sont exploitées dans le code du client
- des définitions concrètes : celles-ci concernent l'invocation du service (binding, service). Ces informations sont exploitées par le moteur SOAP.

Le contenu du WSDL d'un service nommé MonService est de la forme :

Exemple :

```
<!-Structure d'un WSDL -->
<definitions name="MonService"
targetNamespace="http://fr.jmdoudoux.dej.ws.monservice/"
xmlns="http://schemas.xmlsoap.org/wsdl/">
   <!-- Définitions abstraites -->
   <types> ... </types>
   <message> ... </message>
   <portType> ... </portType>

   <!-- Définitions concrètes -->
   <binding> ... </binding>
   <service> ... </service>
</definitions>
```

L'ordre de définition des informations dans un WSDL facilite les traitements de ce document par une machine. Pour une exploitation par un humain, il est plus facile de lire le WSDL en commençant par la fin.

Exemple :

```
<?xml version="1.0" encoding="UTF-8" ?>
 <wsdl:definitions targetNamespace="http://axis.test.jmdoudoux.com"
      xmlns:apachesoap="http://xml.apache.org/xml-soap"
      xmlns:impl="http://axis.test.jmdoudoux.com"
      xmlns:intf="http://axis.test.jmdoudoux.com"
      xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
      xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!--
WSDL created by Apache Axis version: 1.3
Built on Oct 05, 2005 (05:23:37 EDT)

  -->
  <wsdl:message name="additionnerRequest">
  <wsdl:part name="valeur1" type="xsd:int" />
  <wsdl:part name="valeur2" type="xsd:int" />
  </wsdl:message>
  <wsdl:message name="additionnerResponse">
  <wsdl:part name="additionnerReturn" type="xsd:long" />
  </wsdl:message>
 <wsdl:portType name="Calculer">
  <wsdl:operation name="additionner" parameterOrder="valeur1 valeur2">
  <wsdl:input message="impl:additionnerRequest" name="additionnerRequest" />
  <wsdl:output message="impl:additionnerResponse" name="additionnerResponse" />
  </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="CalculerSoapBinding" type="impl:Calculer">
  <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
  <wsdl:operation name="additionner">
  <wsdlsoap:operation soapAction="" />
  <wsdl:input name="additionnerRequest">
  <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    namespace="http://axis.test.jmdoudoux.com" use="encoded" />
  </wsdl:input>
  <wsdl:output name="additionnerResponse">
  <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    namespace="http://axis.test.jmdoudoux.com" use="encoded" />
  </wsdl:output>
  </wsdl:operation>
```

```
</wsdl:binding>
<wsdl:service name="CalculerService">
<wsdl:port binding="impl:CalculerSoapBinding" name="Calculer">
<wsdlsoap:address location="http://localhost:8080/TestWS/services/Calculer" />
</wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

Un document WSDL est un document XML dont le tag racine est <definitions>. Généralement, ce tag contient la définition des différents espaces de nommages qui seront utilisés dans le document XML.

L'espace de nommage du document WSDL est défini.

Exemple :

```
xmlns="http://schemas.xmlsoap.org/wsdl/"  (déclaration de l'espace de nommage par défaut)
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
```

L'espace de nommage de la norme Schema XML est défini.

Exemple :

```
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

Plusieurs autres espaces de nommages standard sont généralement définis selon les protocoles utilisés.

Exemple :

```
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
```

Il est possible de trouver la définition d'espaces de nommages propres à l'implémentation.

Exemple :

```
xmlns:apachesoap="http://xml.apache.org/xml-soap"
```

Un ou plusieurs espaces de nommages sont définis pour le service web lui-même.

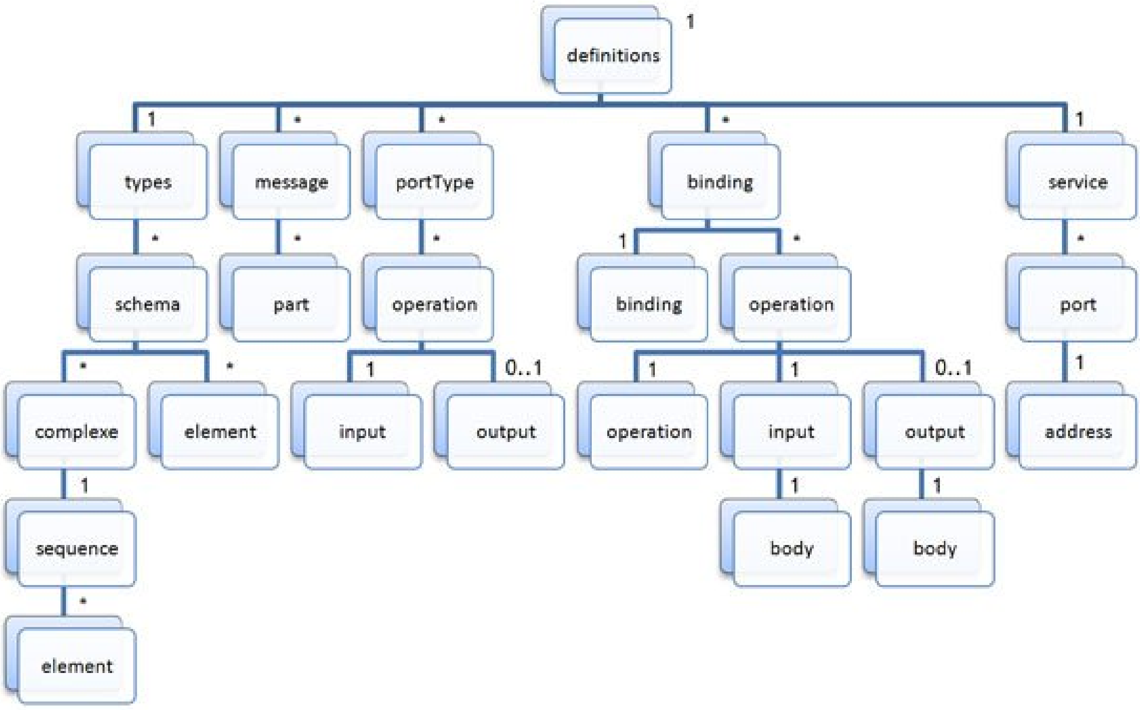
Remarque : les préfixes utilisés pour ces espaces de nommages peuvent être différents selon l'implémentation des services web mise en oeuvre

Dans un document WSDL, les différentes entités se référencent mutuellement grâce à leur nom complet (espace de nommage et nom).

L'élément racine d'un WSDL est le tag <definitions>.

Le tag <definitions> peut contenir plusieurs tags fils :

- <types> : description des types de données utilisés
- <message> : description des messages qui peuvent être composés de plusieurs types
- <portType> : description des opérations du endpoint sous la forme d'échanges de messages. Ceci correspond à l'interface du service
- <binding> : description du protocole et spécification du format des données pour un portType
- <service> : description des endpoints du service (binding et uri)



76.2.2.2. L'élément Types

Le tag <definitions> ne peut avoir qu'un seul tag fils <types>.

L'élément <types> contient une définition des différents types de données qui seront utilisés.

Cette définition peut être faite sous plusieurs formats mais l'utilisation des schémas XML est recommandée. Le WS-I Basic Profile impose que cette description soit faite avec des schémas XML.

L'élément <types> peut avoir aucun, un ou plusieurs éléments fils <schema> ayant pour espace de nommage "http://www.w3.org/2001/XMLSchema".

Chaque structure de données est décrite en utilisant la norme schéma XML.

L'élément <types> peut donc avoir plusieurs schémas XML comme éléments fils. Ces schémas peuvent décrire des types simples (element) ou complexes (complexType).

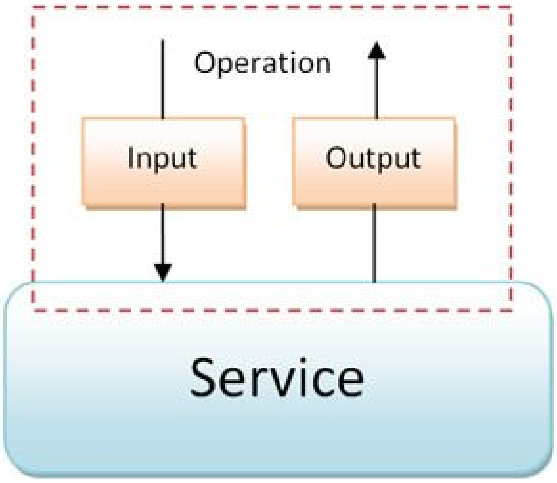
76.2.2.3. L'élément Message

Le tag <definitions> peut avoir plusieurs tags fils <message>. Le tag <message> décrit un message qui est utilisé en tant que requête ou réponse lors de l'invocation d'une opération : il contient une définition des paramètres pour un message échangé en entrée ou en sortie..

Le tag <message> peut avoir un ou plusieurs tags <part>. Le tag <part> possède un attribut "name" pour permettre d'y faire référence et utilise un attribut "element" (pour le style document qui représente l'élément XML inséré dans le body) ou un attribut "type" (pour le style RPC qui représente les paramètres de l'opération).

76.2.2.4. L'élément PortType/Interface

En WSDL, un échange de messages est une opération qui peut donc avoir une requête en entrée et une réponse en sortie.



Le tag <definitions> peut avoir un ou plusieurs tags fils <typePort>. Le tag <portType> décrit l'interface d'un service web.

Le terme typePort est particulièrement ambigu : il correspond à une description de l'interface du service. Le tag <interface> est utilisé à partir de la version 2.0 de WSDL.

Le tag <typePort> possède un attribut name qui permet d'y faire référence.

Il contient un ensemble d'opérations chacune définie dans un tag fils <operation> qui possède un attribut name permettant d'y faire référence.

Le tag <operation> peut avoir les tags fils <input>, <output> et <fault>. La présence et l'ordre des deux premiers tags définissent le mode d'invocation d'une opération nommé MEP (Message Exchance Pattern).

MEP	Description
One-way	L'endpoint reçoit un message sans fournir de réponse : <input> uniquement
Request-response	L'endpoint reçoit un message et envoie la réponse correspondante : <input> et <output>
Notification	L'endpoint envoie un message sans avoir de réponse : <output> uniquement
Solicit-response	L'endpoint envoie un message et reçoit la réponse correspondante : <output> et <input>

Le support de ces types d'opérations dépend de l'implémentation du moteur SOAP utilisé.

L'attribut message des tags <input> et <output> fait référence à un tag <message> par son nom.

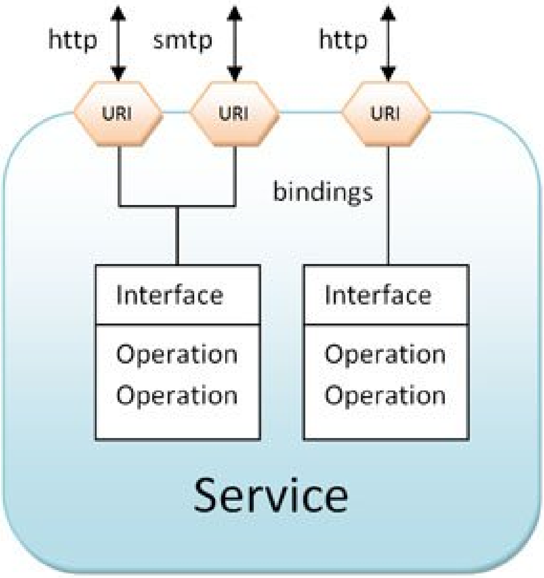
76.2.2.5. L'élément Binding

La description du service doit aussi fournir des informations pour invoquer le service :

- le protocole utilisé pour le transport du message
- l'encodage du message : style et mécanisme d'encodage

Un binding permet de fournir des détails sur la façon dont les données sont transportées.

Un service peut avoir plusieurs bindings mais chacun doit se faire sur une URI unique nommée endpoint.



Le tag <definitions> peut avoir un ou plusieurs tags fils <binding>.

Il permet de définir pour un portType le protocole de transport utilisé et le mode d'encodage des messages.

Le tag <binding> possède un attribut name qui permet d'y faire référence et un attribut type qui permet de faire référence au portType concerné de façon nominative.

Le détail des informations sur le protocole et le mode d'encodage sont des extensions spécifiques comme celle fournie en standard relative à SOAP.

Ainsi le tag fils <soap:binding> est utilisé pour préciser que c'est la version 1.1 de SOAP qui sera utilisée. Son attribut style permet de préciser le style du message : les valeurs possibles sont rpc ou document. Son attribut transport permet de préciser le procotole de transport à utiliser, généralement "http://schemas.xmlsoap.org/soap/http" pour le protocole http.

Le tag <binding> possède un tag fils <operation> pour chaque opération. Le tag <operation> peut avoir plusieurs tags fils.

Le tag fils <soap:operation> permet de définir par son attribut "soapAction" la valeur du header http correspondant. Selon les spécifications WS-I BP, l'attribut "soapAction" doit toujours avoir la valeur chaîne vide.

Les tags <input> et <output> permettent de fournir des précisions sur l'encodage du corps des messages.

Le tag fils <soap:body> permet, par son attribut "use", de préciser comment le corps de message sera encodé : les valeurs possibles sont encoded et document.

L'utilisation de l'encodage "rpc" précise que le corps du message sera une représentation XML des paramètres ou de la valeur de retour de la méthode invoquée.

L'utilisation de l'encodage "document" précise que le corps du message sera un message XML.

76.2.2.6. L'élément Service

Le tag <definitions> ne peut avoir qu'un seul tag fils <service>.

Un service possède un nom précisé dans la valeur de son attribut name.

Un service est composé d'un ou plusieurs ports qui en SOAP correspondent à des endpoints. Chaque port est associé à un binding en utilisant l'attribut binding qui a comme valeur le nom d'un binding défini.

Les tags fils du tag <port> sont spécifiques au binding utilisé : ce sont des extensions qui précisent le endpoint selon le binding. Par exemple pour préciser l'url d'un service web utilisant http, il faut utiliser le tag <address> en fournissant l'url du endpoint comme valeur de l'attribut location. Le tag <endpoint> est utilisé à partir de la version 2.0 de WSDL.

Un port permet de décrire la façon d'accéder au service, ce qui correspond généralement à l'url d'un endpoint et à un binding.

76.2.3. Les registres et les services de recherche

76.2.3.1. UDDI

UDDI, acronyme de Universal Description, Discovery and Integration, est utilisé pour publier et rechercher des services web. C'est un protocole et un ensemble de services pour utiliser un annuaire afin de stocker les informations concernant les services web et de permettre à un client de les retrouver. Les spécifications sont rédigées par l'Oasis.

UDDI est une spécification pour permettre la publication et la recherche d'informations sur des entreprises et les services web qu'elles proposent. UDDI permet à une entreprise de s'inscrire dans l'annuaire, d'y enregistrer et de publier ses services web. Il est alors possible d'accéder à l'annuaire et de rechercher un service particulier.

Le site web officiel d'UDDI est à l'url : http://uddi.xml.org

Un annuaire UDDI contient une description de services web mais aussi des entreprises qui les proposent. Ainsi, il est possible avec UDDI de faire une recherche par entreprise ou par activité.

Les données incluses dans un annuaire UDDI sont classées dans trois catégories :

- les pages blanches (white pages) : elles contiennent les informations générales sur une entreprise
- les pages jaunes (yellow pages) : elles permettent une catégorisation des entreprises
- les pages vertes (green pages) : elles contiennent les informations techniques sur les services proposés

Il est possible d'utiliser un annuaire UDDI en interne dans une entreprise mais il existe aussi des annuaires UDDI globaux nommés UBR (UDDI Business Registry).

Il existe plusieurs versions d'UDDI :

- version 1 :
- version 2 :
- version 3 : les spécifications de cette version sont consultables à l'url https://uddi.org/pubs/uddi_v3.htm

UDDI n'est pas un élément indispensable à la mise en oeuvre des services web comme peut l'être XML, WSDL ou SOAP.

UDDI est une spécification d'annuaire accessible sous la forme de services web de type SOAP que ce soit pour les recherches ou les mises à jour.

76.2.3.2. Ebxml

La suite de cette section sera développée dans une version future de ce document

76.3. Les différents formats de services web SOAP

Un message SOAP peut être formaté de plusieurs façons en fonction de son style et de son type d'encodage.

Il existe deux styles de services web reposant sur SOAP : RCP et Document.

En plus du style, il existe deux types d'encodages : Encoded et Literal. Cela permet de définir quatre combinaisons mais généralement les combinaisons utilisées sont RPC/Encoded et Document/Literal. La combinaison Document/Encoded n'est supportée par aucune implémentation.

De plus, Microsoft est à l'origine d'un cinquième format qui bien que non standardisé est largement utilisé car il est mis en oeuvre par défaut dans la plate-forme.Net et il offre un bon compromis entre performance et restrictions d'utilisation.

Style / Type d'encodage	Encoded	Literal
RPC	RPC / Encoded	RPC / Literal
Document	Document / Encoded	Document / Literal
		Document / Literal wrapped

Il y a deux façons de structurer un document SOAP : RPC et Document. Initialement, avant la diffusion de sa première version, SOAP ne permettait que le style RPC. La version 1.0 s'est vue ajouter le support pour le style Document.

Le style RPC est parfaitement structuré alors que le type Document n'a pas de structure imposée mais son contenu peut être facilement validé grâce à un schéma XML ou traité puisque c'est un document XML. Avec le style document, il est donc possible de structurer librement le corps du message grâce au schéma XML.

Les différents styles sont :

Style	Description
RPC	Les messages contiennent le nom de l'opération Paramètres en entrée multiples et valeur de retour
Document	Les messages ne contiennent pas le nom de l'opération Un document XML en entrée et en retour

Les deux désignations pour le style d'encodage (RPC et Document) peuvent être trompeuses car elles peuvent induire que le style RPC est utilisé pour l'invocation d'opérations distantes et que le style document est utilisé pour l'échange de messages. En fait, le style n'a rien à voir avec un modèle de programmation mais il permet de préciser comment le message SOAP est encodé.

Dans le style RPC, le corps du message (tag <soap:body>) contient un élément qui est le nom de l'opération du service. Cet élément contient un élément fils pour chaque paramètre.

Dans le style Document, le corps du message (tag <soap:body>) contient directement un document xml dont tous les composants doivent être décrits dans un ou plusieurs schémas XML. Le moteur Soap est alors responsable du mapping entre le contenu du message et les objets du serveur.

Les types d'encodage pour sérialiser les messages en XML sont :

Encodage	Description
Encoded	Aussi appelé SOAP encoding car l'encodage est spécifique à SOAP sans utiliser de schéma XML
Literal	L'encodage du message repose sur les schémas XML

Literal wrapped	Idem Literal avec en plus l'encapsulation de chaque message dans un tag qui contient le nom de l'opération. Ce format est défini par Microsoft qui l'utilise dans sa plate-forme .Net

Le type d'encodage Literal propose que le contenu du corps (body) soit validé par un schéma XML donné : chaque élément qui correspond à un paramètre ou à la valeur de retour est décrit dans un schéma XML.

Le type d'encodage Encoded utilise un ensemble de règles reposant sur les types de données des schémas XML pour encoder les données mais le message ne respecte pas de schéma particulier : chaque élément qui correspond à un paramètre ou à la valeur de retour contient la description de la donnée sous la forme d'attributs spécifiés dans la norme Soap (type, null ou pas, ...)

Le type d'encodage Encoded est particulièrement adapté lors de l'utilisation d'un graphe d'objets cyclique car chaque type d'objet ne sera défini qu'une seule fois. Avec l'encodage Literal, chaque élément est répété dans le document.

Le type d'encodage Literal permet une manipulation du document XML qui constitue le message (validation par un schéma XML, parsing du document, transformation à l'aide d'une feuille de style XSLT, ...)

Le format RPC encoded repose sur des types définis par SOAP alors que les formats RPC Literal et Document Literal repose sur les types du schéma XML.

Il existe donc plusieurs formats utilisables pour un message SOAP :

- RPC Encoded : c'est le premier format historiquement proposé par SOAP mais son utilisation est de moins en moins fréquente
- RPC Literal :
- Document Literal :
- Document Literal Wrapped :
- Document Encoded : ce format n'est pas supporté actuellement par les moteurs de services web

Ces différents styles et types d'encodages sont à l'origine de difficultés d'interopérabilité au début des services web car la plate-forme .Net utilisait le style Document par défaut et les implémentations sur la plate-forme Java utilisaient plutôt le style RPC.

Au début de l'utilisation de SOAP, c'est donc le format RPC encoding qui était utilisé. Depuis, c'est plutôt le format Document/Literal ou RPC/Literal qui est recommandé notamment par les spécifications WS-I Basic Profile.

Les sections suivantes vont utiliser la classe d'implémentation du service web ci-dessous avec Axis.

Exemple :

```
package fr.jmdoudoux.dej.axis;

public class Calculer {

  public long additionner(int valeur1, int valeur2) {
    return valeur1+valeur2;
  }
}
```

Les sections suivantes vont utiliser la classe d'implémentation du service web ci-dessous avec Metro

Exemple :

```
package fr.jmdoudoux.dej.ws;

import javax.jws.WebService;

@WebService
public class Calculer {

  public long additionner(int valeur1, int valeur2) {
    return valeur1+valeur2;
  }
}
```

76.3.1. Le format RPC Encoding

Ce format de messages est le plus ancien et le plus simple à mettre en oeuvre pour le développeur.

La structure du corps du message avec le style RPC est imposée. Par exemple, pour une requête, il doit contenir le nom de la méthode ainsi que ses paramètres. Cette structure est donc de la forme :

Exemple :

```
<soapenv:Body>
<ns0:nom_de_la_methode xmlns:ns0="uri_de_l_espace_de_nommage"
  soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <nom_param_1 xsi:type="xsd:type_param_1">valeur_parametre_1</nom_param_1>
  <nom_param_2 xsi:type="xsd:type_param_2">valeur_parametre_2</nom_param_2>
  </ns0: nom_de_la_methode>
  </soapenv:Body>
  </soapenv:Envelope>
```

Le message réponse a une forme similaire.

Le type des paramètres peut être simple ou plus complexe (par exemple un objet qui encapsule des données de types simples ou d'autres objets).

Attention : RPC Encoding n'est pas conforme à la spécification WS-I Basic Profile

Exemple : Le fichier WSDL

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://axis.test.jmdoudoux.com"
                  xmlns:apachesoap="http://xml.apache.org/xml-soap"
                  xmlns:impl="http://axis.test.jmdoudoux.com"
                  xmlns:intf="http://axis.test.jmdoudoux.com"
                  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!--WSDL created by Apache Axis version: 1.3
Built on Oct 05, 2005 (05:23:37 EDT)-->
   <wsdl:message name="additionnerRequest">
      <wsdl:part name="valeur1" type="xsd:int"/>
      <wsdl:part name="valeur2" type="xsd:int"/>
   </wsdl:message>
   <wsdl:message name="additionnerResponse">
      <wsdl:part name="additionnerReturn" type="xsd:long"/>
   </wsdl:message>
   <wsdl:portType name="Calculer">
      <wsdl:operation name="additionner" parameterOrder="valeur1 valeur2">
         <wsdl:input message="impl:additionnerRequest" name="additionnerRequest"/>
         <wsdl:output message="impl:additionnerResponse" name="additionnerResponse"/>
      </wsdl:operation>
   </wsdl:portType>
   <wsdl:binding name="CalculerSoapBinding" type="impl:Calculer">
      <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
      <wsdl:operation name="additionner">
         <wsdlsoap:operation soapAction=""/>
         <wsdl:input name="additionnerRequest">
            <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
              namespace="http://axis.test.jmdoudoux.com" use="encoded"/>
         </wsdl:input>
         <wsdl:output name="additionnerResponse">
            <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
              namespace="http://axis.test.jmdoudoux.com" use="encoded"/>
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>
   <wsdl:service name="CalculerService">
      <wsdl:port binding="impl:CalculerSoapBinding" name="Calculer">
         <wsdlsoap:address location="http://localhost:8080/TestWS/services/Calculer"/>
      </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

Exemple : descripteur Axis

Exemple :

```
<service name="Calculer" provider="java:RPC">
  <operation name="additionner" qname="ns1:additionner"
             returnQName="additionnerReturn"
             returnType="xsd:long"
             soapAction=""
             xmlns:ns1="http://axis.test.jmdoudoux.com"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <parameter name="valeur1" type="xsd:int"/>
   <parameter name="valeur2" type="xsd:int"/>
  </operation>
  <parameter name="allowedMethods" value="additionner"/>
  <parameter name="typeMappingVersion" value="1.2"/>
  <parameter name="wsdlPortType" value="Calculer"/>
  <parameter name="className" value="fr.jmdoudoux.dej.axis.Calculer"/>
  <parameter name="wsdlServicePort" value="Calculer"/>
  <parameter name="wsdlTargetNamespace" value="http://axis.test.jmdoudoux.com"/>
  <parameter name="wsdlServiceElement" value="CalculerService"/>
</service>
```

La requête SOAP

Exemple :

```
<?xml version="1.0" encoding="UTF-8" ?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns0:additionner
      xmlns:ns0="http://axis.test.jmdoudoux.com"
      soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <valeur1 xsi:type="xsd:int">10</valeur1>
      <valeur2 xsi:type="xsd:int">20</valeur2>
    </ns0:additionner>
  </soapenv:Body>
</soapenv:Envelope>
```

La réponse SOAP

Exemple :

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:additionnerResponse
      xmlns:ns1="http://axis.test.jmdoudoux.com"
      soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <additionnerReturn href="#id0" />
    </ns1:additionnerResponse>
    <multiRef xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
              id="id0" soapenc:root="0"
              soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
```

```
               xsi:type="xsd:long">30</multiRef>
  </soapenv:Body>
</soapenv:Envelope>
```

Avantages :

- ce format est facilement compréhensible par un être humain
- le nom de l'opération à invoquer est inclus dans le message ce qui rend le dispatching par le moteur SOAP vers la méthode correspondante très facile
- la gestion des valeurs null est supportée en standard

Inconvénients :

- il n'est pas possible de valider le message dans la mesure où seuls les paramètres sont définis dans un schéma. Le reste du contenu du corps de l'enveloppe est défini directement dans le WSDL.
- ce mode peut poser des problèmes d'intéropérabilité : il est d'ailleurs incompatible avec les spécifications WS-I Basic Profile. Son utilisation n'est donc plus recommandée
- c'est le type d'encodage qui a les moins bonnes performances et qui génère les messages les plus verbeux notamment à cause de l'indication du type de chaque donnée

76.3.2. Le format RPC Literal

Les messages de type RPC/Literal sont encodés comme des appels RPC avec une description des paramètres et des valeurs de retour décrites chacune avec son propre schéma XML.

Le moteur Soap utilisé pour l'exemple de cette section est Metro version 1.5.

Exemple :

```
import javax.jws.soap.SOAPBinding.Use;

@WebService
@SOAPBinding(style=Style.RPC, use=Use.LITERAL, parameterStyle=ParameterStyle.BARE)
public class Calculer {

  public long additionner(Valeurs valeurs) {
    return valeurs.getValeur1()+valeurs.getValeur2();
  }
}
```

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime"
      version="2.0">
  <endpoint name="CalculerWS" implementation="fr.jmdoudoux.dej.ws.Calculer"
      url-pattern="/services/CalculerWS" />
</endpoints>
```

Le fichier WSDL

Exemple :

```
<?xml version='1.0' encoding='UTF-8'?>
  <!--
    Published by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is
    JAX-WS RI 2.1.7-hudson-48-.
  -->
  <!--
    Generated by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is
    JAX-WS RI 2.1.7-hudson-48-.
  -->
```

```
<definitions
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http://ws.test.jmdoudoux.com/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://schemas.xmlsoap.org/wsdl/"
  targetNamespace="http://ws.test.jmdoudoux.com/" name="CalculerService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://ws.test.jmdoudoux.com/"
        schemaLocation="http://localhost:8089/TestMetro/services/CalculerWS?xsd=1" />
    </xsd:schema>
  </types>
  <message name="additionner">
    <part name="additionner" element="tns:additionner" />
  </message>
  <message name="additionnerResponse">
    <part name="additionnerResponse" element="tns:additionnerResponse" />
  </message>
  <portType name="Calculer">
    <operation name="additionner">
      <input message="tns:additionner" />
      <output message="tns:additionnerResponse" />
    </operation>
  </portType>
  <binding name="CalculerPortBinding" type="tns:Calculer">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
      style="rpc" />
    <operation name="additionner">
      <soap:operation soapAction="" />
      <input>
        <soap:body use="literal" namespace="http://ws.test.jmdoudoux.com/" />
      </input>
      <output>
        <soap:body use="literal" namespace="http://ws.test.jmdoudoux.com/" />
      </output>
    </operation>
  </binding>
  <service name="CalculerService">
    <port name="CalculerPort" binding="tns:CalculerPortBinding">
      <soap:address location="http://localhost:8089/TestMetro/services/CalculerWS" />
    </port>
  </service>
</definitions>
```

La description est faite dans un schéma dédié

Exemple :

```
<?xml version='1.0' encoding='UTF-8'?>
<xs:schema xmlns:tns="http://ws.test.jmdoudoux.com/"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           version="1.0" targetNamespace="http://ws.test.jmdoudoux.com/">
<xs:element name="additionner" nillable="true" type="tns:valeurs" />
  <xs:element name="additionnerResponse" type="xs:long" />
  <xs:complexType name="valeurs">
    <xs:sequence>
      <xs:element name="valeur1" type="xs:int" />
      <xs:element name="valeur2" type="xs:int" />
    </xs:sequence>
</xs:complexType>
</xs:schema>
```

Le premier élément du tag <body> du message désigne la méthode à invoquer

La requête SOAP

Exemple :

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:ws="http://ws.test.jmdoudoux.com/">
   <soapenv:Header/>
   <soapenv:Body>
      <ws:additionner>
         <ws:additionner>
            <valeur1>20</valeur1>
            <valeur2>30</valeur2>
         </ws:additionner>
      </ws:additionner>
   </soapenv:Body>
</soapenv:Envelope>
```

La réponse SOAP

Exemple :

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:additionnerResponse xmlns:ns2="http://ws.test.jmdoudoux.com/">
      50</ns2:additionnerResponse>
   </S:Body>
</S:Envelope>
```

Avantages :

- supporté par le WS-I Basic Profile
- le nom de la méthode invoquée est inclus dans le message

Inconvénient :

- il est difficile de valider le message

76.3.3. Le format Document encoding

Ce type d'encodage n'est supporté par aucun moteur de services web.

76.3.4. Le format Document Literal

Pour respecter le WS-I BP, le tag <soap:body> d'un message Soap encodé en document Literal ne peut avoir qu'un seul élément fils.

L'exemple de cette section va donc utiliser une version légèrement différente du service web qui attend en paramètre un bean encapsulant les données à calculer.

Exemple :

```
package fr.jmdoudoux.dej.ws.axis;

public class CalculerWS {

  public long additionner(Valeurs valeurs) {
    return valeurs.getValeur1()+valeurs.getValeur2();
  }
}
```

La classe Valeurs est un POJO qui encapsule les paramètres requis par la méthode.

Exemple :

```
package fr.jmdoudoux.dej.ws;
```

```
public class Valeurs {
  private int valeur1;
  private int valeur2;

  public Valeurs() {
    super();
  }

  public Valeurs(int valeur1, int valeur2) {
    super();
    this.valeur1 = valeur1;
    this.valeur2 = valeur2;
  }

  public synchronized int getValeur1() {
    return valeur1;
  }

  public synchronized void setValeur1(int valeur1) {
    this.valeur1 = valeur1;
  }

  public synchronized int getValeur2() {
    return valeur2;
  }

  public synchronized void setValeur2(int valeur2) {
    this.valeur2 = valeur2;
  }
}
```

Descripteur Axis 1.x

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
            xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
  <service name="CalculerWS" provider="java:RPC" style="document"
    use="literal">
    <parameter name="wsdlTargetNamespace" value="http://axis.ws.test.jmdoudoux.com" />
    <parameter name="wsdlServiceElement" value="CalculerWSService" />
    <parameter name="schemaQualified" value="http://axis.ws.test.jmdoudoux.com" />
    <parameter name="wsdlServicePort" value="CalculerWS" />
    <parameter name="className" value="fr.jmdoudoux.dej.ws.axis.CalculerWS" />
    <parameter name="wsdlPortType" value="CalculerWS" />
    <parameter name="typeMappingVersion" value="1.2" />
    <operation xmlns:retNS="http://axis.ws.test.jmdoudoux.com"
      xmlns:rtns="http://www.w3.org/2001/XMLSchema" name="additionner"
      qname="additionner" returnQName="retNS:additionnerReturn" returnType="rtns:long"
      soapAction="">
      <parameter xmlns:pns="http://axis.ws.test.jmdoudoux.com"
        xmlns:tns="http://axis.ws.test.jmdoudoux.com" qname="pns:valeurs"
        type="tns:Valeurs" />
    </operation>
    <parameter name="allowedMethods" value="additionner" />

    <typeMapping xmlns:ns="http://axis.ws.test.jmdoudoux.com"
      qname="ns:Valeurs" type="java:fr.jmdoudoux.dej.ws.axis.Valeurs"
      serializer="org.apache.axis.encoding.ser.BeanSerializerFactory"
      deserializer="org.apache.axis.encoding.ser.BeanDeserializerFactory"
      encodingStyle="" />
  </service>
</deployment>
```

Le fichier WSDL

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://axis.ws.test.jmdoudoux.com"
                  xmlns:apachesoap="http://xml.apache.org/xml-soap"
                  xmlns:impl="http://axis.ws.test.jmdoudoux.com"
                  xmlns:intf="http://axis.ws.test.jmdoudoux.com"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!--WSDL created by Apache Axis version: 1.4
Built on Apr 22, 2006 (06:55:48 PDT)-->
 <wsdl:types>
  <schema elementFormDefault="qualified"
          targetNamespace="http://axis.ws.test.jmdoudoux.com"
          xmlns="http://www.w3.org/2001/XMLSchema">
   <complexType name="Valeurs">
    <sequence>
     <element name="valeur1" type="xsd:int"/>
     <element name="valeur2" type="xsd:int"/>
    </sequence>
   </complexType>
   <element name="valeurs" type="impl:Valeurs"/>
   <element name="additionnerReturn" type="xsd:long"/>
  </schema>
 </wsdl:types>
   <wsdl:message name="additionnerResponse">
      <wsdl:part element="impl:additionnerReturn" name="additionnerReturn"/>
   </wsdl:message>
   <wsdl:message name="additionnerRequest">
      <wsdl:part element="impl:valeurs" name="valeurs"/>
   </wsdl:message>
   <wsdl:portType name="CalculerWS">
      <wsdl:operation name="additionner" parameterOrder="valeurs">
         <wsdl:input message="impl:additionnerRequest" name="additionnerRequest"/>
         <wsdl:output message="impl:additionnerResponse" name="additionnerResponse"/>
      </wsdl:operation>
   </wsdl:portType>
   <wsdl:binding name="CalculerWSSoapBinding" type="impl:CalculerWS">
      <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
      <wsdl:operation name="additionner">
         <wsdlsoap:operation soapAction=""/>
         <wsdl:input name="additionnerRequest">
            <wsdlsoap:body use="literal"/>
         </wsdl:input>
         <wsdl:output name="additionnerResponse">
            <wsdlsoap:body use="literal"/>
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>
   <wsdl:service name="CalculerWSService">
      <wsdl:port binding="impl:CalculerWSSoapBinding" name="CalculerWS">
         <wsdlsoap:address location="http://localhost:8089/TestsAxisWS/services/CalculerWS"/>
      </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

La requête SOAP

Exemple :

```
<soapenv:Envelope
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:axis="http://axis.ws.test.jmdoudoux.com">
   <soapenv:Header/>
   <soapenv:Body>
      <axis:valeurs>
         <axis:valeur1>20</axis:valeur1>
         <axis:valeur2>30</axis:valeur2>
      </axis:valeurs>
   </soapenv:Body>
</soapenv:Envelope>
```

La réponse SOAP

Exemple :

```
<soapenv:Envelope
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <soapenv:Body>
      <additionnerReturn
        xmlns="http://axis.ws.test.jmdoudoux.com">
        50
      </additionnerReturn>
   </soapenv:Body>
</soapenv:Envelope>
```

Avantages :

- le contenu du corps de l'enveloppe peut être validé puisque tous les éléments sont définis dans un schéma
- supporté par le WS-I Basic Profile avec quelques contraintes

Inconvénients :

- le fichier WSDL est plus compliqué à comprendre pour un être humain
- le nom de l'opération n'apparaît pas dans la requête SOAP : le mapping vers la méthode du service web à invoquer est donc plus limité puisqu'il doit se faire sur la séquence de paramètres
- il n'est donc pas possible dans un même service web d'avoir deux méthodes avec la même liste de paramètres
- le nom de l'opération n'est pas contenu dans le message ce qui impose des contraintes au niveau des signatures des interfaces des opérations
- pour respecter le WS-I BP le tag <soap:body> ne peut avoir qu'un seul tag fils

76.3.5. Le format Document Literal wrapped

Ce format a été défini par Microsoft pour la plate-forme .Net et il n'existe aucune spécification officielle mais c'est un standard de fait. Ce format reprend le format Document Literal mais le corps contient un élément qui précise le nom de l'opération.

Le tag <body> possède plusieurs caractéristiques en Document/Literal wrapped :

- le corps du message n'est composé que d'une seule partie
- cette partie est encapsulée dans un élément dont le nom correspond au nom de l'opération invoquée
- les éléments des paramètres ne possèdent aucun attribut

Le fichier WSDL

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://axis.test.jmdoudoux.com"
    xmlns:apachesoap="http://xml.apache.org/xml-soap"
    xmlns:impl="http://axis.test.jmdoudoux.com"
    xmlns:intf="http://axis.test.jmdoudoux.com"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!--WSDL created by Apache Axis version: 1.3
Built on Oct 05, 2005 (05:23:37 EDT)-->
 <wsdl:types>
  <schema elementFormDefault="qualified"
      targetNamespace="http://axis.test.jmdoudoux.com"
      xmlns="http://www.w3.org/2001/XMLSchema">
   <element name="additionner">
    <complexType>
     <sequence>
```

```
      <element name="valeur1" type="xsd:int"/>
      <element name="valeur2" type="xsd:int"/>
     </sequence>
    </complexType>
   </element>
   <element name="additionnerResponse">
    <complexType>
     <sequence>
      <element name="additionnerReturn" type="xsd:long"/>
     </sequence>
    </complexType>
   </element>
  </schema>
 </wsdl:types>
   <wsdl:message name="additionnerResponse">
      <wsdl:part element="impl:additionnerResponse" name="parameters"/>
   </wsdl:message>
   <wsdl:message name="additionnerRequest">
      <wsdl:part element="impl:additionner" name="parameters"/>
   </wsdl:message>
   <wsdl:portType name="Calculer">
      <wsdl:operation name="additionner">
         <wsdl:input message="impl:additionnerRequest" name="additionnerRequest"/>
         <wsdl:output message="impl:additionnerResponse" name="additionnerResponse"/>
      </wsdl:operation>
   </wsdl:portType>
   <wsdl:binding name="CalculerSoapBinding" type="impl:Calculer">
      <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
      <wsdl:operation name="additionner">
         <wsdlsoap:operation soapAction=""/>
         <wsdl:input name="additionnerRequest">
            <wsdlsoap:body use="literal"/>
         </wsdl:input>
         <wsdl:output name="additionnerResponse">
            <wsdlsoap:body use="literal"/>
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>
   <wsdl:service name="CalculerService">
      <wsdl:port binding="impl:CalculerSoapBinding" name="Calculer">
         <wsdlsoap:address location="http://localhost:8080/TestWS/services/Calculer"/>
      </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

Descripteur Axis

Exemple :

```
<service name="Calculer" provider="java:RPC" style="wrapped" use="literal">
  <operation name="additionner" qname="ns1:additionner"
      returnQName="ns1:additionnerReturn" returnType="xsd:long"
      soapAction="" xmlns:ns1="http://axis.test.jmdoudoux.com"
                    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <parameter qname="ns1:valeur1" type="xsd:int"/>
    <parameter qname="ns1:valeur2" type="xsd:int"/>
  </operation>
  <parameter name="allowedMethods" value="additionner"/>
  <parameter name="typeMappingVersion" value="1.2"/>
  <parameter name="wsdlPortType" value="Calculer"/>
  <parameter name="className" value="fr.jmdoudoux.dej.axis.Calculer"/>
  <parameter name="wsdlServicePort" value="Calculer"/>
  <parameter name="schemaQualified" value="http://axis.test.jmdoudoux.com"/>
  <parameter name="wsdlTargetNamespace" value="http://axis.test.jmdoudoux.com"/>
  <parameter name="wsdlServiceElement" value="CalculerService"/>
 </service>
```

La requête SOAP

Exemple :

```
<?xml version="1.0" encoding="UTF-8" ?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:q0="http://axis.test.jmdoudoux.com"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <q0:additionner>
      <q0:valeur1>20</q0:valeur1>
      <q0:valeur2>30</q0:valeur2>
    </q0:additionner>
  </soapenv:Body>
</soapenv:Envelope>
```

La réponse SOAP

Exemple :

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <additionnerResponse xmlns="http://axis.test.jmdoudoux.com">
      <additionnerReturn>50</additionnerReturn>
    </additionnerResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Avantages :

- le contenu du corps du message est défini par un schéma permettant ainsi sa validation
- le nom de l'opération est inclus dans la requête SOAP sous la forme d'un tag au nom de l'opération et placé entre le tag <body>et les tags contenant les paramètres.
- ce format est relativement proche du format RPC/Literal

Inconvénient :

- il n'est pas possible d'utiliser ce style avec des méthodes surchargées

76.3.6. Le choix du format à utiliser

Les différents formats (style et encodage) des services web ont tous des restrictions d'utilisation qui peuvent engendrer des limitations dans l'écriture des services ou forcer l'utilisation d'un format ou d'un autre.

76.3.6.1. L'utilisation de document/literal

Dans ce mode, le nom de l'opération n'est pas fourni dans le message : le mapping pour déterminer l'opération à invoquer repose donc sur les paramètres.

Il n'est donc pas possible d'invoquer le service ci-dessous avec le mode document/literal

Exemple :

```
public MonService {
  public void maMethode(int x, int y);
  public void maSecondeMethode(int x, int y);
}
```

76.3.6.2. L'utilisation de Document/Literal Wrapped

Il pourrait être tentant de toujours utiliser le mode Document/Literal Wrapped mais ce n'est pas forcément le meilleur choix :

- ce mode n'est pas supporté par tous les moteurs SOAP
- il n'est pas possible d'utiliser des opérations surchargées dans un service puisque le mapping de l'opération sur la méthode se fait sur le nom de la méthode. La classe ci-dessous ne peut pas être exposée sous la forme d'un service web invoqué par le mode Document/Literal Wrapped.

Exemple :

```
public MonService {
  public void maMethode(int x, int y);
  public void maMethode(int x);
}
```

Remarque : WSDL 2.0 interdit l'utilisation des opérations surchargées.

76.3.6.3. L'utilisation de RPC/Literal

Comme le mode Document/Literal ne contient pas le nom de l'opération à invoquer, il y a des cas où il faut utiliser le mode Document/Literal Wrapped ou un des deux modes RPC/Encoded ou RPC/Literal.

Exemple :

```
public MonService {
  public void maMethode(int x, int y);
  public void maMethode(int x);
  public void maSecondeMethode(int x, int y);
}
```

L'exemple ci-dessus ne peut pas être invoqué ni en Document/Literal ni en Document/Literal Wrapped.

Comme le mode RPC/encoded n'est pas WS-I Basic Profile compliant, il ne reste que le mode RPC/Literal

76.3.6.4. L'utilisation de RPC/Encoded

Le mode RPC/Encoded n'est pas WS-I Basic Profile compliant mais il est parfois nécessaire de l'utiliser. Ce mode est le seul qui puisse prendre en charge un graphe d'objets contenant plusieurs fois la même référence.

Exemple :

```
<complexType name="MonElement">
    <sequence>
        <element name="nom" type="xsd:string"/>
        <element name="partie1" type="MonElement" xsd:nillable="true"/>
        <element name="partie2" type="MonElement" xsd:nillable="true"/>
    </sequence>
</complexType>
```

RPC/Encoded utilise l'attribut id pour donner un identifiant à un élément et utilise un attribut href pour y faire référence.

Exemple :

```
<element1>
    <name>nom1</name>
    <partie1 href="1234"/>
    <partie2 href="1234"/>
</element1>
<element2 id="1234">
    <name>nom2</name>
    <partie1 xsi:nil="true"/>
    <partie2 xsi:nil="true"/>
</element2>
```

Dans le style Literal, il n'y pas de moyen de faire une référence sur un objet déjà présent dans le graphe : la seule solution c'est de le dupliquer, ce qui va poser des problèmes au consommateur du service.

76.4. Des conseils pour la mise en oeuvre

Avant de développer des services web, il faut valider la solution choisie avec un POC (Proof Of Concept) ou un prototype. Lors de ces tests, il est important de vérifier l'interopérabilité notamment si les services web sont consommés par différentes technologies.

Le choix du moteur SOAP est aussi très important notamment vis-à-vis du support des spécifications, des performances, de la documentation, ...

76.4.1. Les étapes de la mise en oeuvre

La mise en oeuvre de services web suit plusieurs étapes.

Etape 1 : définition des contrats des services métiers

Cette étape est une phase d'analyse qui va définir les fonctionnalités proposées par chaque service pour répondre aux besoins

Etape 2 : identification des services web

Cette étape doit permettre de définir les contrats techniques des services web à partir des services métiers définis dans l'étape précédente. Un service métier peut être composé d'un ou plusieurs services web.

La réalisation de cette étape doit tenir compte de plusieurs contraintes :

- Penser forte granularité / faible couplage
- Ternir compte de contraintes techniques
- Préférer les services web indépendants du contexte client

L'invocation d'un service est coûteuse notamment à cause du mapping objet/xml et xml/objet réalisé à chaque appel. Cette règle est vraie pour toutes les invocations de fonctionnalités distantes mais encore plus avec les services web. Il est donc préférable de limiter les invocations de méthodes d'un service web en proposant des fonctionnalités à forte granularité. Par exemple, il est préférable de définir une opération qui permet d'obtenir les données d'une entité plutôt que de proposer autant d'opérations que l'entité possède de champs. Ceci permet de réduire le nombre d'invocations du service web et réduit le couplage entre la partie front-end et back-end.

La définition des services web doit tenir compte de contraintes techniques liées aux performances ou à la consommation de ressources. Par exemple, si le temps de traitement d'un service web est long, il faudra prévoir son invocation de façon asynchrone ou si les données retournées sont des binaires de tailles importantes, il faudra envisager d'utiliser le mécanisme de pièces jointes (attachment).

Il est préférable de définir des services web qui soient stateless (ne reposant pas par exemple sur une utilisation de la session http). Ceci permet de déployer les services web dans un cluster où la réplication de session sera inutile.

Etape 3 : écriture des services web

Cette étape est celle du codage proprement dit des services web.

Deux approches sont possibles :

- écriture du WSDL en premier (contract first) : des outils du moteur Soap sont utilisés pour gérer le code des services web à partir du WSDL. Face à la complexité de la rédaction du WSDL, cette approche n'est pas toujours privilégiée.
- écriture de la classe et génération du WSDL (code first) : chaque service est implémenté sous la forme d'une ou plusieurs classes et c'est le moteur Soap utilisé qui va générer le WSDL correspondant en se basant sur la description de la classe et des métadonnées.

Etape 4 : déploiement et tests

Les services web doivent être packagés et déployés généralement dans un serveur d'applications ou un conteneur web.

Pour tester les services web, il est possible d'utiliser des outils fournis par l'IDE ou d'utiliser des outils tiers comme SoapUI qui propose de très nombreuses fonctionnalités pour les tests des services web allant de la simple invocation à l'invocation de scénarios complexes et de tests de charges.

Etape 5 : consommation des services web par les applications clientes

Il faut mettre en oeuvre les outils du moteur Soap utilisé par l'application cliente pour générer les classes nécessaires à l'invocation des services web et utiliser ces classes dans l'application. C'est généralement le moment de faire quelques adaptations pour permettre une bonne communication entre le client et le serveur.

76.4.2. Quelques recommandations

Afin de maximiser la portabilité d'un service web, il faut essayer de suivre quelques recommandations.

Il ne faut pas se lier à un langage de programmation :

- n'utiliser que des types communs : int, float, String, Date, ...
- ne pas utiliser de types spécifiques : Object, DataSet, ...
- éviter la composition d'objets
- utiliser un tableau ou des collections typées avec un generic plutôt qu'une collection non typée

Il faut éviter la surchage des méthodes.

Il faut éviter de transformer une classe en service web (notamment en utilisant des annotations) : il est recommandé de définir une interface qui va établir le contrat entre le service et son implémentation. Cette pratique venant de la POO doit aussi s'appliquer pour les services web.

76.4.3. Les problèmes liés à SOAP

SOAP est assez complexe et sa mise en oeuvre dépend de l'implémentation de la technologie utilisée côtés consommateur et fournisseur de services web. Il en résulte des problèmes d'interopérabilités alors qu'un des but de SOAP est pourtant de s'en affranchir.

Il existe plusieurs types de problèmes :

<u>Les problèmes liés aux versions de SOAP</u>

Les versions SOAP 1.1 et 1.2 étant incompatibles, cela peut entrainer des problèmes de compatibilité si les implémentations des moteurs SOAP utilisés supportent des versions différentes.

Ceci est notamment le cas si l'implémentation du moteur SOAP est assez ancienne.

<u>Les problèmes liés aux modèles de messages</u>

Un message Soap peut être encodé selon plusieurs modèles : le modèle le plus ancien (RPC) est abandonné au profit du modèle Document.

Cela peut introduire des problèmes d'incompatibilité notamment entre des services web existants et des consommateurs plus récents ou vice-versa.

76.5. Les API Java pour les services web

Java propose un ensemble d'API permettant la mise en oeuvre des services web.

API	Rôle
JAXP	API pour le traitement de documents XML : analyse en utilisant SAX ou DOM et transformation en utilisant XSLT.
JAX-RPC	API pour le développement de services web utilisant SOAP avec le style RPC
JAXM	API pour le développement de services utilisant des messages XML orientés documents
JAXR	API pour permettre un accès aux annuaires de référencement de services web
JAXB	API et outils pour automatiser le mapping d'un document XML avec des objets Java
StaX	API pour le traitement de documents XML
SAAJ	API pour permettre la mise en oeuvre des spécifications SOAP with Attachment
JAX-WS	API pour le développement grâce à des annotations de services web utilisant SOAP avec le style Document

L'API de base pour le traitement de document XML avec Java est JAXP. JAXP regroupe un ensemble d'API pour traiter des documents XML avec SAX et DOM et les modifier avec XSLT. Cette API est indépendante de tout parseur. JAXP est détaillée dans le chapitre «Java et XML».

D'autres API sont spécifiques au développement de services web :

- JAX-RPC (JSR-101) : permet l'appel de procédures distantes en utilisant SOAP (Remote Procedure Call)
- JAXM (JSR-67) : permet l'envoi de messages (en utilisant SAAJ)
- JAXR : permet l'accès au service de registre de façon standard (UDDI)
- SAAJ (SOAP with Attachment API for Java) : permet l'envoi et la réception de messages respectant les normes SOAP et SOAP with Attachment

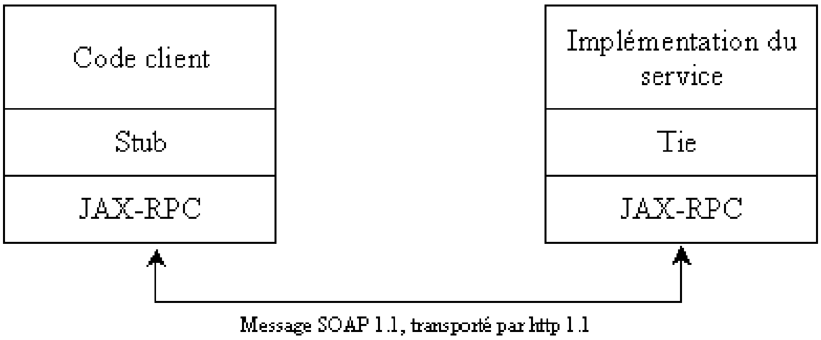
76.5.1. JAX-RPC

JAX-RPC est l'acronyme de Java API for XML-Based Remote Procedure Calls. Cette API permet la mise en oeuvre de services web utilisant SOAP aussi bien côté fournisseur que consommateur : elle permet l'appel de méthodes distantes et la réception de leurs réponses en utilisant SOAP 1.1 et HTTP 1.1.

Cette API a été développée par le JCP sous la JSR 101. Elle propose de masquer un grand nombre de détails de l'utilisation de SOAP notamment en ce qui concerne le codage en XML du message et ainsi de rendre cette API facile à utiliser.

L'utilisation de JAX-RPC est similaire à celle de RMI : le code du client appel les méthodes à partir d'un objet local nommé stub. Cet objet se charge de dialoguer avec le serveur et de coder et décoder les messages SOAP échangés.

Un objet similaire nommé tie permet de réaliser le même type d'opération côté serveur.


Code client
Stub
JAX-RPC
Implémentation du service
Tie
JAX-RPC
Message SOAP 1.1, transporté par http 1.1

La principale différence entre RMI et les services web est que RMI ne peut être utilisé qu'avec Java alors que les services web sont interopérables grâce à XML. Ainsi un client écrit en Java peut utiliser un service web développé avec .Net et vice-versa.

La spécification JAX-RPC définit précisément le mapping entre les types Schema et les types Java.

Type Schema	Type Java
xsd:boolean	boolean
xsd:short	short
xsd:int	int
xsd:long	long
xsd:integer	BigInteger
xsd:float	float
xsd:double	double
xsd:decimal	BigDecimal
xsd:date	java.util.calendar
xsd:time	java.util.calendar
xsd:datetime	java.util.calendar
xsd:base64Binary	byte[]
xsd:hexBinary	byte[]

Les types primitifs qui sont nillable sont mappés sur leurs wrappers Java correspondants.

Les types complexes sont mappés sur des Java beans.

L'API JAX-RPC est regroupée dans plusieurs sous-packages du package javax.xml.rpc

76.5.1.1. La mise en oeuvre côté serveur

L'écriture d'un service web avec JAX-RPC requiert plusieurs entités :

- une interface facultative qui décrit le endpoint
- une implémentation du endpoint
- un ou plusieurs fichiers de description et configuration
- le wdsl du service web

L'utilisation de JAX-RPC côté serveur se fait en plusieurs étapes :

1. Définition de l'interface du service (écrite manuellement ou générée automatiquement par un outil à partir de la description du service (WSDL)).

Exemple :

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface MonWS extends Remote {

  public String getMessage(String nom) throws RemoteException;
}
```

Cette interface doit étendre l'interface java.rmi.Remote.

Toutes les méthodes définies dans l'interface doivent au minimum déclarer la possibilité de lever une exception de type java.rmi.RemoteException. Chaque méthode peut aussi déclarer d'autres exceptions dans sa définition du moment que ces exceptions héritent de la classe java.lang.Exception.

Les méthodes peuvent sans restriction utiliser des types primitifs et l'objet String pour les paramètres et la valeur de retour. Pour les autres types, il existe dans les spécifications une liste minimale prédéfinie de ceux utilisables.

Une interface particulière peut cependant proposer le support d'autres types. Par exemple, l'implémentation de référence propose le support de la plupart des classes de l'API Collection : ArrayList, HashMap, HashTable, LinkedList, TreeMap, TreeSet, Vector, ... Dans ce cas toutefois, attention à la perte de la portabilité lors de l'utilisation d'une autre implémentation.

2. Ecriture de la classe d'implémentation du service

C'est une simple classe Java qui implémente l'interface définie précédemment et posséde un constructeur sans paramètre : dans l'exemple ci-dessous, celui-ci sera généré lors de la compilation car il n'y a pas d'autre constructeur défini.

Exemple :

```
public class MonWS_Impl implements MonWS {

  public String getMessage(String nom) {
    return new String("Bonjour " + nom);
  }
}
```

Il est inutile dans l'implémentation des méthodes de déclarer la levée de l'exception de type RemoteException. C'est lors de l'invocation de la méthode par JAX-RPC que cette exception pourra être levée.

3. Déploiement du service

Le déploiement dépend du moteur Soap utilisé et implique généralement la création d'un fichier de mapping entre l'url et la classe correspondante.

76.5.1.2. La mise en oeuvre côté client

JAX-RPC peut aussi être utilisée pour consommer un service web dans un client.

L'invocation de méthodes côté client se fait de manière synchrone avec JAX-RPC : le client fait appel au service et se met en attente jusqu'à la réception de la réponse

Cette invocation du service peut alors être faite selon trois modes :

- Un stub généré
- Un proxy dynamique

- Dynamic Invocation Interface

Un proxy dynamique met en oeuvre un mécanisme proche de celui utilisé par RMI : le client accède à un service distant en utilisant un stub. Le stub sert de proxy : il implémente l'interface du service et se charge des appels au service en utilisant le protocole SOAP lors de l'invocation de ses méthodes.

Ce proxy est généré par un compilateur dédié qui va utiliser le WSDL pour générer le proxy, notamment le portType pour définir l'interface de l'objet et les binding et port pour connaître les paramètres d'appel du service.

Le proxy généré est responsable de la transformation des invocations de méthodes en requêtes Soap et de la transformation des messages Soap en objets selon les indications fournies dans le WSDL. En cas d'erreur, le message Soap de type fault est transformé en une exception.

La suite de cette section sera développée dans une version future de ce document

76.5.2. JAXM

JAXM est l'acronyme de Java API for XML Messaging. Cette API permet le développement de services utilisant des messages XML orientés documents.

JAXM a été développée sous la JSR-067.

Les classes de cette API sont regroupées dans le package javax.xml.messaging.

JAXM met en oeuvre SOAP 1.1 et SAAJ

76.5.3. JAXR

L'API JAXR (Java API for XML Registries) propose de standardiser les accès aux registres dans lesquels sont recensés les services web. JAXR permet notamment un accès aux registres de type UDDI ou ebXML.

Une implémentation de cette spécification doit être proposée par un fournisseur.

Elle est incluse dans deux packages :

- javax.xml.registry : classes et interfaces de base (Connection, Query, LifeCycleManager, ...)
- javax.xml.registry.infomodel : interfaces qui décrivent les informations du modèle stockées dans un registre

Le support de l'accès aux registres de type ebXML est facultatif.

76.5.4. SAAJ

L'API SAAJ (SOAP with Attachment API for Java) permet l'envoi et la réception de messages respectant les normes SOAP 1.1 et SOAP with attachments : cette API propose un niveau d'abstraction assez élevé permettant de simplifier l'usage de SOAP.

Les classes de cette API sont regroupées dans le package javax.xml.soap.

Initialement, cette API était incluse dans JAXM. Depuis la version 1.1, elles ont été séparées.

SAAJ propose des classes qui encapsulent les différents éléments d'un message SOAP : SOAPMessage, SOAPPart, SOAPEnvelope, SOAPHeader et SOAPBody.

Tous les échanges de messages avec SOAP utilisent une connexion encapsulée dans la classe SOAPConnection. Cette classe permet la connexion directe entre l'émetteur et le receveur du ou des messages.

76.5.5. JAX-WS

JAX-WS (Java API for XML based Web Services) est une nouvelle API, mieux architecturée, qui remplace l'API JAX-RPC 1.1 mais n'est pas compatible avec elle. Il est fortement recommandé d'utiliser le modèle de programmation proposé par JAX-WS notamment pour les nouveaux développements.

Elle propose un modèle de programmation pour produire (côté serveur) ou consommer (côté client) des services web qui communiquent par des messages XML de type SOAP.

Elle a pour but de faciliter et simplifier le développement des services web notamment grâce à l'utilisation des annotations. JAX-WS fournit les spécifications pour le coeur du support des services web de la plate-forme Java SE et Java EE.

JAX-WS a été spécifié par la JSR 224 : Java API for XML-Based Web Services (JAX-WS) 2.0.

JAX-WS permet la mise en oeuvre de plusieurs spécifications :

- JAX-WS respecte le standard WS-I Basic Profile version 1.1.
- JAX-WS propose un support pour SOAP 1.1 et 1.2
- JAX-WS permet le développement de services web orientés RPC (literal) ou orientés documents (literal/encoded/literal wrapped)

JAX-WS repose sur plusieurs autres JSR :

- JSR 181 (Web Services MetaData for the Java Platform) : propose un ensemble d'annotations qui permettent de définir les services web
- JSR 109 et JSR 921 (Implementing Enterprise Web Services) : décrit comment déployer, gérer et accéder aux services web par un serveur d'applications
- JSR 183 (Web Services Message Security APIs) : décrit la sécurisation des messages SOAP

Le fournisseur de l'implémentation de JAX-WS utilise les spécifications de la JSR 921 pour générer les fichiers de configuration et de déploiement à partir des annotations et d'éventuelles métadonnées.

JAX-WS utilise JAXB 2.0 et SAAJ 1.3. JAXB propose une API et des outils pour automatiser le mapping d'un document XML et des objets Java. A partir d'une description du document XML (Schéma XML ou DTD), des classes sont générées pour effectuer automatiquement l'analyse du document ML et le mapping de ce dernier dans des objets Java.

JAX-WS peut être combiné avec d'autres spécifications comme les EJB 3 par exemple.

76.5.5.1. La mise en oeuvre de JAX-WS

JAX-WS est une spécification : pour la mettre en oeuvre, il faut utiliser une implémentation.

L'implémentation de référence de JAX-WS est le projet Metro développé par la communauté du projet GlassFish. Il existe d'autres implémentations notamment Axis 2 qui propose son propre modèle de programmation mais aussi un support de JAX-WS.

Le développement d'un service web en Java avec JAX-WS débute par la création d'une classe annotée avec @WebService du package javax.jws. La classe ainsi annotée définit le endpoint du service web.

Le service endpoint interface (SEI) est une interface qui décrit les méthodes du service : celles-ci correspondent aux opérations invocables par un client.

Il est possible de préciser explicitement le SEI en utilisant l'attribut endpointInterface de l'annotation @WebService

76.5.5.2. La production de service web avec JAX-WS

Par rapport à JAX-RPC, l'utilisation de JAX-WS est plus simple : un service web peut être basiquement défini en utilisant une classe de type POJO avec des annotations.

La classe d'implémentation du service est donc très simple : un simple POJO avec des annotations. Il n'y a pas besoin d'implémenter une interface particulière de l'API ni de déclarer une exception dans les méthodes.

Avec JAX-WS, la définition d'un service web et de ses opérations se fait en utilisant des annotations soit dans une interface qui décrit le service soit directement dans la classe d'implémentation.

Ni côté client ni côté serveur, le développeur n'a besoin de manipuler le contenu des messages Soap. Ceci est cependant possible pour des besoins très spécifiques.

Les annotations fournissent des métadonnées exploitées par le moteur Soap pour générer le code des traitements sous-jacents. Le développeur est ainsi déchargé de la plomberie et peut se concentrer sur les traitements métiers qui représentent la plus-value du service.

Le développement d'un service web avec JAX-WS requiert plusieurs étapes :

- coder la classe qui encapsule le service
- compiler la classe
- utiliser la commande wsgen pour générer les fichiers requis pour le déploiement (schémas, WSDL, classes, ...)
- packager le service dans un fichier.war
- déployer le war dans un conteneur

Pour définir un endpoint avec JAX-WS, il a plusieurs contraintes :

- la classe qui encapsule le endpoint doit être public, non static, non final, non abstract et être annotée avec @WebService
- elle doit avoir un constructeur par défaut (sans paramètre)
- il est recommandé de définir explicitement l'interface du SEI
- les méthodes exposées par le service web doivent être public, non static, non final et être annotées avec @WebMethod
- les types des paramètres et de la valeur de retour de ces méthodes doivent être supportés par JAXB

Exemple :

```
import javax.jws.WebService;

import javax.jws.WebMethod;

@WebService
public class MonService {

  @WebMethod
  public String saluer(){
    return "Bonjour";
  }
}
```

La classe qui encapsule le endpoint du service peut définir des méthodes annotées avec @PostConstruct et @PreDestroy pour définir des traitements liés au cycle de vie du service. Ces méthodes sont invoquées par le conteneur respectivement avant la première utilisation de la classe et avant le retrait du service.

Il faut compiler la classe et utiliser l'outil wsgen pour générer les classes et fichiers requis pour l'exécution du service web.

L'outil wsgen doit être utilisé pour générer les classes utiles à l'exposition du service web : celles-ci concernent essentiellement des classes qui utilisent JAXB pour mapper le contenu du message avec un objet et vice-versa. Il permet aussi de générer le WSDL et les schémas XML des messages.

La syntaxe est de la forme :

```
wsgen [options] <sei>
```

<sei> est le nom pleinement qualifié de classe d'implémentation du SEI.

Option	Rôle
-classpath <path> -cp <path>	Spécifier le classpath
-d <directory>	Préciser le répertoire qui va contenir les classes générées
-help	Afficher l'aide
-keep	Conserver les fichiers générés
-r <directory>	Préciser le répertoire qui va contenir les fichiers de ressources générés (WSDL, ...)
-s <directory>	Préciser le répertoire qui va contenir les fichiers sources générés
-verbose	Activer le mode verbeux
-version	Afficher la version
-wsdl[:protocol]	Demander la génération du WSDL : ce fichier n'est pas utilisé à l'exécution mais il peut être consulté par le développeur pour vérification. Le protocole est facultatif : il permet de préciser la version SOAP qui sera utilisée (les valeurs possibles sont soap1.1 et soap1.2)
-servicename <name>	Définir la valeur de l'attribut name du tag <wsdl:service> lorsque l'option -wsdl est utilisée
-portname <name>	Définir la valeur de l'attribut name du tag <wsdl:port> lorsque l'option -wsdl est utilisée

Une tâche Ant est proposée pour invoquer wsgen par cet outil de build.

Il faut packager le service dans une webapp avec les fichiers compilés.

Il faut déployer le service dans un conteneur web ou un serveur d'applications. Au déploiement, JAX-WS va créer les différentes classes requises pour l'utilisation du service web (celles encapsulant les messages) si celles-ci ne sont pas présentes.

Pour voir le WSDL du service il faut utiliser l'url :

```
http://localhost:8080/helloservice/hello?wsdl
```

JAX-WS utilise JAXB-2.0 pour le mapping entre les objets et XML : les objets échangés par les services web peuvent utiliser les annotations de JAXB pour paramétrer finement certains éléments du message SOAP.

Exemple :

```
package fr.jmdoudoux.dej.ws;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;

@WebService()
public class PersonneWS {

  @WebMethod(operationName = "Saluer")
  public String Saluer(@WebParam(name = "personne") final Personne personne) {
    return "Bonjour " + personne.getNom() + " "+personne.getPrenom();
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.ws;

import java.util.Date;

public class Personne {

  private String nom;
  private String prenom;
  private Date dateNaiss;

  public Personne() {
    super();
  }

  public Personne(String nom, String prenom, Date dateNaiss) {
    super();
    this.nom = nom;
    this.prenom = prenom;
    this.dateNaiss = dateNaiss;
  }

  public synchronized String getNom() {
    return nom;
  }

  public synchronized void setNom(String nom) {
    this.nom = nom;
  }

  public synchronized String getPrenom() {
    return prenom;
  }

  public synchronized void setPrenom(String prenom) {
    this.prenom = prenom;
  }

  public synchronized Date getDateNaiss() {
    return dateNaiss;
  }

  public synchronized void setDateNaiss(Date dateNaiss) {
    this.dateNaiss = dateNaiss;
  }
}
```

Le WSDL généré définit l'élément avec un nom dont la première lettre est en minuscule.

Exemple :

```
<?xml version='1.0' encoding='UTF-8'?>
<!-- Published by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is
JAX-WS RI 2.1.7-hudson-48-. -->
<xs:schema xmlns:tns="http://ws.test.jmdoudoux.com/"
              xmlns:xs="http://www.w3.org/2001/XMLSchema"
              version="1.0" targetNamespace="http://ws.test.jmdoudoux.com/">

  <xs:element name="Saluer" type="tns:Saluer" />

  <xs:element name="SaluerResponse" type="tns:SaluerResponse" />

  <xs:complexType name="Saluer">
    <xs:sequence>
      <xs:element name="personne" type="tns:personne" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="personne">
    <xs:sequence>
      <xs:element name="dateNaiss" type="xs:dateTime" minOccurs="0" />
      <xs:element name="nom" type="xs:string" minOccurs="0" />
      <xs:element name="prenom" type="xs:string" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="SaluerResponse">
    <xs:sequence>
      <xs:element name="return" type="xs:string" minOccurs="0" />
      </xs:sequence>
    </xs:complexType>
</xs:schema>
```

En utilisant l'annotation @XmlType, il est possible de forcer le nom de l'élément généré dans le schéma

Exemple :

```
package fr.jmdoudoux.dej.ws;

import java.util.Date;

import javax.xml.bind.annotation.XmlType;

@XmlType(name = "Personne")
public class Personne {

  private String nom;
  private String prenom;
  private Date dateNaiss;

   ...
}
```

Le WSDL généré définit l'élément avec un nom dont la première lettre est en majuscule.

Exemple :

```
<?xml version='1.0' encoding='UTF-8'?>
<!-- Published by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is
JAX-WS RI 2.1.7-hudson-48-. -->
<xs:schema xmlns:tns="http://ws.test.jmdoudoux.com/"
              xmlns:xs="http://www.w3.org/2001/XMLSchema"
              version="1.0" targetNamespace="http://ws.test.jmdoudoux.com/">

  <xs:element name="Saluer" type="tns:Saluer" />

  <xs:element name="SaluerResponse" type="tns:SaluerResponse" />

  <xs:complexType name="Saluer">
    <xs:sequence>
      <xs:element name="personne" type="tns:Personne" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="Personne">
    <xs:sequence>
      <xs:element name="dateNaiss" type="xs:dateTime" minOccurs="0" />
      <xs:element name="nom" type="xs:string" minOccurs="0" />
      <xs:element name="prenom" type="xs:string" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="SaluerResponse">
    <xs:sequence>
      <xs:element name="return" type="xs:string" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

76.5.5.3. La consommation de services web avec JAX-WS

Dans la partie cliente, un objet de type proxy est généré pour faciliter l'invocation et la consommation des services web. Les classes de ce proxy sont générées par l'outil wsimport à la demande du développeur à partir du wsdl.

Pour développer un client qui consomme le service web, il y a plusieurs étapes :

- utiliser l'outil wsimport pour générer les classes du proxy
- écrire le code des traitements en utilisant le proxy
- compiler toutes les classes
- exécuter le client

La suite de cette section sera développée dans une version future de ce document

76.5.5.4. Les handlers

Les handlers proposent un mécanisme de traitements particuliers exécutés par le moteur Soap pour permettre d'agir sur les messages de type requête ou réponse. JAX-WS propose deux types de handlers selon la source des données à obtenir ou modifier dans le message :

- Protocol handler : ce type de handler est dédié à un protocole particulier par exemple Soap. Il permet d'obtenir des informations ou d'en modifier dans toutes les parties du message
- Logical Handler : ce type de handler est indépendant du protocole utilisé par le message en permettant une modification du corps du message à partir de son contexte JAXB.

Les handlers sont généralement utilisés pour traiter des informations particulières du message Soap

Les handlers pour le protocole SOAP doivent hériter de la classe javax.xml.ws.handler.soap.SOAPHandler. La classe SOAPMessageContext propose des méthodes pour permettre un accès au contenu du message encapsulé dans un objet de type SOAPMessage. Le contenu du message peut alors être manipulé avec l'API SAAJ.

Les logical handlers doivent hériter de la classe javax.xml.ws.handler.LogicalHandler. Ils permettent un accès au contenu du message qui correspond pour un message de type SOAP au body. La classe LogicalMessageContext propose des méthodes pour permettre un accès au contenu du message encapsulé dans un objet de type LogicalMessage.

La suite de cette section sera développée dans une version future de ce document

76.5.6. La JSR 181 (Web Services Metadata for the Java Platform)

La JSR 181 propose une spécification pour permettre le développement de services web en utilisant des POJO et des annotations.

La JSR 181 a pour but de définir un modèle de programmation pour faciliter le développement des services web. Ce modèle repose essentiellement sur les annotations : ceci permet de définir les services web sans avoir à connaître les détails de l'implémentation qui sera mise en oeuvre.

Les annotations proposées permettent un contrôle assez fin sur la façon dont un service web va être exposé et invoqué.

La JSR 181 est une spécification dont le but est de fournir un standard pour la déclaration de services web en proposant :

- Un modèle standard pour le développement de services web en utilisant des annotations
- De masquer les détails de l'implémentation
- D'assurer la maintenabilité et l'interopérabilité

Chaque implémentation de cette JSR doit fournir des fonctionnalités pour permettre d'exécuter les classes annotées dans un environnement d'exécution pour les services web.

La mise en oeuvre suit plusieurs étapes :

- Ecriture de la classe qui contient les fonctionnalités à exposer
- Annoter la classe ou son interface
- Exploitation par une implémentation des annotations de la classe pour générer des schémas XML, le WSDL et d'autres fichiers requis pour le déploiement

Exemple :

```
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService public class MonService {

  @WebMethod
  public String saluer() {
    return "Bonjour";
  }
}
```

Il existe plusieurs contraintes dont il faut tenir compte lors de l'implémentation du service. La classe de l'implémentation doit :

- être public,
- non final,
- non abstract,
- avoir un constructeur par défaut

Par défaut, toutes les méthodes public sont exposées sous la forme d'une opération et ne doivent utiliser que des paramètres respectant ceux définis dans JAX-RPC 1.1. Les méthodes héritées sont aussi exposées sauf celles héritées de la classe Object.

76.5.6.1. Les annotations définies

Les annotations sont utilisées dans la classe d'implémentation ou dans l'interface d'un service web.

Toutes les annotations de la JSR 181 sont définies dans le package javax.jws.

Ces annotations sont exploitées au runtime.

Attention : plusieurs implémentations fournissent, en plus des annotations de la JSR 181, des annotations qui leur sont propres. Même si elles sont pratiques, elles limitent la portabilité des services web à s'exécuter dans un autre moteur Soap (exemple : @EnableMTOM, @ServiceProperty, @ServicesProperties dans XFire).

76.5.6.2. javax.jws.WebService

L'annotation javax.ws.WebService permet de définir une classe ou une interface comme étant l'interface du endpoint d'un service web.

L'annotation WebService est la seule annotation obligatoire pour développer un service web.

Si l'annotation est utilisée sur l'interface du service web (SEI), il faut aussi l'utiliser sur la classe d'implémentation en précisant l'interface avec l'attribut endpointInterface.

Cette annotation s'utilise sur une classe ou une interface uniquement.

Attribut	Rôle
String name	le nom du service web utilisé dans l'attribut name de l'élément wsdl:portType du WSDL Par défaut, c'est le nom non qualifié de la classe
String targetNamespace	espace de nommage utilisé dans le WSDL Par défaut c'est le nom du package
String serviceName	le nom du service utilisé dans l'attribut name de l'élément wsdl:service du WSDL Par défaut, c'est le nom de la classe suffixée par "Service"
String wsdlLocation	url relative ou absolue du WSDL prédéfini
String endpointInterface	nom pleinement qualifié de l'interface du endpoint (SEI), ce qui permet de séparer l'interface de l'implémentation
String portName	Nom du port du service web utilisé dans l'attribut name de l'élément wsdl:port du WSDL

Exemple :

```
@WebService(name = "BonjourWS", targetNamespace = "http://www.jmdoudoux.fr/ws/Bonjour")

public class BonjourServiceImpl {

  @WebMethod
  public String saluer() {
    return "Bonjour";
  }
}
```

76.5.6.3. javax.jws.WebMethod

L'annotation javax.ws.WebMethod permet de définir une méthode comme étant une opération d'un service web.

Cette annotation s'utilise sur une méthode uniquement. La méthode sur laquelle cette annotation est appliquée doit être public.

Elle possède plusieurs attributs.

Attribut	Rôle
String operationName	nom utilisé dans l'élément wsdl:operation du message Par défaut: le nom de la méthode
String action	action associée à l'opération : utilisé comme valeur du paramètre SOAPAction
boolean exclude	booléen qui précise si la méthode doit être exposée ou non dans le service web. Cette propriété n'est utilisable que dans une classe et doit être le seul attribut de l'annotation.

	Par défaut : false

L'annotation WebMethod ne peut être utilisée que dans une classe ou une interface annotée avec @WebService.

Les paramètres de la méthode, sa valeur de retour et les exceptions qu'elle peut lever doivent obligatoirement respecter les spécifications relatives à ces entités dans les spécifications JAX-RPC 1.1.

76.5.6.4. javax.jws.OneWay

L'annotation javax.ws.OneWay permet de définir une méthode comme étant une opération d'un service web qui ne fournit pas de réponse lors de son invocation. Elle permet une optimisation à l'exécution qui évite d'attendre une réponse qui ne sera pas fournie.

Cette annotation s'utilise sur une méthode uniquement : celle-ci ne doit pas avoir de valeur de retour ou lever une exception puisque dans ce cas, il y a une réponse de type SoapFault.

Cette annotation ne possède aucun attribut.

Exemple :

```
@WebService
public class MonService {

  @WebMethod
  @Oneway
  public void MonOperation() {
  }
}
```

76.5.6.5. javax.jws.WebParam

L'annotation javax.ws.WebParam permet de configurer comment un paramètre d'une opération sera mappé dans le message SOAP.

Cette annotation s'utilise uniquement sur un paramètre d'une méthode de l'implémentation du service.

Attribut	Rôle
String name	nom du paramètre utilisé dans le WSDL Par défaut: le nom du paramètre
Mode mode	mode d'utilisation du paramètre. Le type Mode est une énumération qui contient IN, OUT et INOUT Par défaut : IN
String targetNamespace	précise l'espace de nommage du paramètre dans les messages utilisant le mode document Par défaut : l'espace de nommage du service web
boolean header	booléen qui indique si la valeur du paramètre est contenue dans l'en-tête de la requête http plutôt que dans le corps Par défaut : false
String partName	Définit l'attribut name de l'élément <wsdl:part> des messages de type RPC et DOCUMENT/BARE

Cette annotation est pratique pour permettre d'utiliser le même paramètre dans plusieurs opérations d'un service web encodé en document literal.

76.5.6.6. javax.jws.WebResult

L'annotation javax.ws.WebResult permet de choisir comment une valeur de retour d'une opération sera mappée dans l'élément wsdl:part message SOAP.

Cette annotation s'utilise sur une méthode uniquement.

Attribut	Rôle
String name	nom de la valeur de retour utilisé dans le WSDL. Avec le style RPC, c'est l'attribut name de l'élément wsdl:part. Avec le style DOCUMENT, c'est le nom de l'élément dans la réponse Par défaut: return pour RPC et DOCUMENT/WRAPPED et le nom de la méthode suffixé par "Response" pour DOCUMENT/BARE
String targetNamespace	espace de nommage de la valeur de retour dans les messages utilisant le mode document Par défaut : l'espace de nommage du service web
boolean header	booléen qui indique si la valeur de retour est stockée dans l'en-tête de la requête http plutôt que dans le corps Par défaut : false
String partName	attribut name de l'élément wsdl:part des messages de type RPC et DOCUMENT/BARE Par défaut : la valeur de l'attribut name

Cette annotation est pratique pour permettre d'utiliser la même valeur de retour dans plusieurs opérations d'un service web encodé en Document Literal.

76.5.6.7. javax.jws.soap.SOAPBinding

L'annotation javax.jws.soap.SOAPBinding permet de déterminer l'encodage du message et de la réponse SOAP.

Cette annotation s'utilise sur une classe, une interface ou une méthode.

Attribut	Rôle
Style style	Définir le style d'encodage du message. Style est une énumération qui contient DOCUMENT et RPC. Par défaut: DOCUMENT
Use use	Définir le format du message. Use est une énumération qui contient ENCODED et LITERAL. Par défaut: LITERAL
ParameterStyle parameterStyle	Définir si les paramètres forment le contenu du message ou s'ils sont encapsulés par un tag du nom de l'opération à invoquer. ParameterStyle est une énumération qui contient BARE et WRAPPED. BARE ne peut être utilisé qu'avec le style DOCUMENT Par défaut: WRAPPED

Exemple :

```
@WebService
@SOAPBinding(style=Style.DOCUMENT, use=Use.LITERAL, parameterStyle=ParameterStyle.BARE)
public class MonService {

  @WebMethod

  public void MonOperation() {
  }
};
```

76.5.6.8. javax.jws.HandlerChain

La suite de cette section sera développée dans une version future de ce document

76.5.6.9. javax.jws.soap.SOAPMessageHandlers

La suite de cette section sera développée dans une version future de ce document

76.5.7. La JSR 224 (JAX-WS 2.0 Annotations)

La JSR 224 définit les annotations spécifiques à JAX-WS 2.0.

Toutes ces annotations sont dans le package javax.xml.ws.

76.5.7.1. javax.xml.ws.BindingType

L'annotation @BindingType permet de préciser le binding qui sera utilisé pour invoquer le service. Elle s'utilise sur une classe

Attribut	Rôle
value	Identifiant du binding à utiliser. Les valeurs possibles sont : SOAPBinding.SOAP11HTTP_BINDING, SOAPBinding.SOAP12HTTP_BINDING ou HTTPBinding.http_BINDING La valeur par défaut est SOAP11_HTTP_BINDING

76.5.7.2. javax.xml.ws.RequestWrapper

L'annotation @RequestWrapper permet de préciser la classe JAXB de binding qui sera utilisée dans la requête à l'invocation du service. Elle s'utilise sur une méthode

Attribut	Rôle
String className	Préciser le nom pleinement qualifié de la classe qui encapsule la requête (Obligatoire)
String localName	Définir le nom de l'élément dans le schéma qui encapsule la requête.

	Par défaut, c'est la valeur de l'attribut operationName de l'annotation WebMethod
String targetNamespace	l'espace de nommage. Par défaut, c'est le targetNamespace du SEI

76.5.7.3. javax.xml.ws.ResponseWrapper

L'annotation @ResponseWrapper permet de préciser la classe JAXB de binding qui sera utilisée dans la réponse à l'invocation du service. Elle s'utilise sur une méthode

Attribut	Rôle
String localName	Définir le nom de l'élément dans le schéma qui encapsule la réponse. Par défaut c'est le nom de l'opération définie par l'annotation @WebMethod concaténé à Response
String targetNamespace	Définir l'espace de nommage. Par défaut, c'est le targetNamespace du SEI
String ClassName	Préciser le nom pleinement qualifié de la classe qui encapsule la réponse (Obligatoire)

76.5.7.4. javax.xml.ws.ServiceMode

Cette annotation permet de préciser si le provider va avoir accès uniquement au payload du message (PAYLOAD) ou à l'intégralité du message (MESSAGE).

Elle s'utilise sur une classe qui doit obligatoirement implémenter un Provider.

Attribut	Rôle
Service.Mode value	Indiquer si le provider va avoir accès uniquement au payload du message (PAYLOAD) ou à l'intégralité du message (MESSAGE). La valeur par défaut est PAYLOAD

Exemple :

```
@ServiceMode(value=Service.Mode.PAYLOAD)
public class MonOperationProvider implements Provider<Source> {
  public Source invoke(Source source)
    throws WebServiceException {
    Source source = null;
    try {

      // code du traitement de la requete et generation de la reponse

    } catch(Exception e) {
      throw new WebServiceException("Erreur durant les traitements du Provider", e);
    }
    return source;
  }
}
```

76.5.7.5. javax.xml.ws.WebFault

Cette annotation s'utilise sur une classe qui encapsule une exception afin de personnaliser certains éléments de la partie Fault du message Soap. Elle s'utilise sur une exception levée par une opération.

Attribut	Rôle
String name	Le nom de l'élément fault Cet attribut est obligatoire

String targetNameSpace	Définir l'espace de nommage pour l'élément fault.
String faultBean	Nom pleinement qualifié de la classe qui encapsule l'exception

76.5.7.6. javax.xml.ws.WebEndpoint

Cette annotation permet de préciser le portName d'une méthode du SEI.

Elle s'utilise sur une méthode.

Attribut	Rôle
String name	Définir le nom qui va identifier de façon unique l'élément <wsdl:port> du tag <wsdl:service>

76.5.7.7. javax.xml.ws.WebServiceclient

La suite de cette section sera développée dans une version future de ce document

76.5.7.8. javax.xml.ws.WebServiceProvider

Cette annotation est à utiliser sur une implémentation d'un Provider

Elle s'utilise sur des classes qui héritent de la classe Provider.

Attribut	Rôle
String portName	nom du port du service (élément <wsdl:portName>)
String serviceName	nom du service (élément <wsdl:service>)
String targetNamespace	espace de nommage
String wsdlLocation	chemin du WSDL du service

Exemple :

```
@WebServiceProvider
public class MonOperationProvider implements Provider<Source> {
  public Source invoke(Source source) throws WebServiceException {
    Source source = null;
    try {

      // code du traitement de la requete et generation de la reponse

    } catch(Exception e) {
      throw new WebServiceException("Erreur durant les traitements du Provider", e);
    }
    return source;
  }
}
```

76.5.7.9. javax.xml.ws.WebServiceRef

L'annotation WebServiceRef permet de définir une référence sur un service web et éventuellement autorise son injection.

Cette annotation est à utiliser dans un contexte Java EE.

Elle s'utilise sur une classe, une méthode (getter ou setter) ou un champ.

Attribut	Rôle
String name	nom JNDI de la ressource. Par défaut sur un champ c'est le nom du champ. Par défaut sur un getter ou un setter, c'est le nom de la propriété
Class type	type de la ressource. Par défaut sur un champ, c'est le type du champ. Par défaut sur un getter ou un setter, c'est le type de la propriété
String mappedName	nom spécifique au conteneur sur lequel le service est mappé (non portable)
Class value	classe du service qui doit étendre javax.xml.ws.Service
String wsdlLocation	chemin du WSDL du service

76.6. Les implémentations des services web

Il existe de nombreuses implémentations possibles de moteurs SOAP permettant la mise en oeuvre de services web avec Java, notamment plusieurs solutions open source :

- Intégrées à la plate-forme Java EE 5.0 et Java SE 6.0
- JWSDP de Sun
- Axis et Axis 2 du projet Apache
- CXF du projet Apache
- JBoss WS
- Metro du projet GlassFish
- ...

A cause d'un effort de spécification tardif de JAX-WS, plusieurs implémentations utilisent une approche spécifique pour la mise en oeuvre et le déploiement de services web, ce qui rend le choix d'une de ces solutions délicat. Heureusement, toutes tendent à proposer un support de JAX-WS.

Même si les concepts sous-jacents sont équivalents, quelle que soit l'implémentation utilisée, sa mise en oeuvre est très différente d'une implémentation à l'autre.

De plus, la plupart des solutions historiques sont relativement complexes à mettre en oeuvre car certains points techniques ne sont pas assez masqués par les outils (code à écrire, fichiers de configuration, descripteurs de déploiement, ...). Avec ces solutions, le développeur doit consacrer une part non négligeable de son temps à du code technique pour développer le service web.

JAX-WS propose une solution pour simplifier grandement le développement des services grâce à l'utilisation d'annotations qui évitent d'avoir à écrire du code ou des fichiers pour la plomberie. JAX-WS, en tant que spécification, est implémentée dans plusieurs solutions.

76.6.1. Axis 1.0

Axis (Apache eXtensible Interaction System) est un projet open-source du groupe Apache diffusé sous la licence Apache 2.0 qui propose une implémentation d'un moteur de service web implémentant le protocole SOAP : il permet de créer, déployer et consommer des services web.

Son but est de proposer un ensemble d'outils pour faciliter le développement, le déploiement et l'utilisation des services web écrits en Java. Axis propose de simplifier au maximum les tâches pour la création et l'utilisation des services web. Il permet notamment de générer automatiquement le fichier WSDL à partir d'une classe Java et le code nécessaire à l'appel du service web.

Pour son utilisation, Axis 1.0 nécessite un J.D.K. 1.3 minimum et un conteneur de servlets (les exemples de cette section utilise Tomcat).

Le site officiel est à l'url https://ws.apache.org/axis/

C'est un projet open source d'implémentation du protocole SOAP. Il est historiquement issu du projet Apache SOAP.

C'est un outil populaire qui de fait est la référence des moteurs de services web Open Source implémentant JAX-RPC en Java : son utilisation est répandue notamment dans des produits open source ou commerciaux.

La version 1.2 diffusé en mai 2005 apporte le support de l'encodage de type Document/Literal pour être compatible avec les spécifications WS-I Basic Profile 1.0 et JAX-RPC 1.1.

La version 1.3 est diffusée en octobre 2005

La version la plus récente est la 1.4, diffusée en avril 2006.

Attention : Axis 1.x n'est plus supporté au profit de Axis 2 qui possède lui aussi des numéros de versions 1.x.

Axis implémente plusieurs spécifications :

- JSR 101 : Java API for XML-Based RPC (JAX-RPC) 1.1
- JSR 67 : SOAP with Attachments API for Java Specification (SAAJ) 1.2
- Java API for XML Registries Specification (JAXR) 1.0

Axis permet donc la mise en oeuvre de :

- SOAP 1.1 et 1.2
- WSDL 1.1
- XML-RPC
- WS-I Basic Profile 1.1

Attention : Axis 1.0 n'est pas compatible avec

- JSR 109 Web Services for EE (WS4EE) 1.0
- JSR 224 Java API for XML-Based Web Services (JAX-WS) 2.0
- JSR 181 Web Service Metadata for the Java Platform
- JSR 222 Java Architecture for XML Binding (JAXB) 2.0

Axis génère le document wsdl du service web : pour accéder à ce document il suffit d'ajouter ?wsdl à l'url d'appel du service web.

L'interopérabilité entre Axis et .Net 1.x est assurée tant que les types utilisés se limitent aux primitives, aux chaînes de caractères, aux tableaux des types précédents et aux Java Beans composés uniquement des types précédents ou d'autres Java Beans.

L'interopérabilité entre Axis 1.4 et .Net 2.0 est bien meilleure. Par exemple, la gestion des objets Nullable dans .Net 2.0 est prise en compte (notamment pour les dates et types primitifs) : il n'est donc plus nécessaire d'utiliser une gestion particulière pour ces objets.

Les extensions sont mises en oeuvre au travers du mécanisme de handlers.

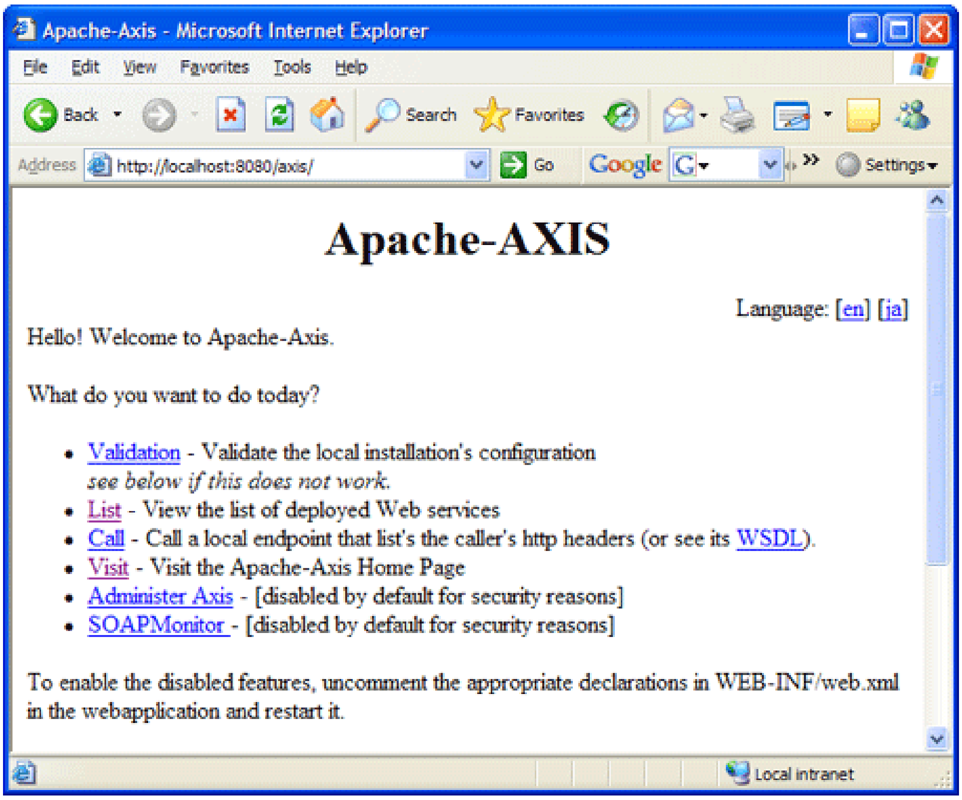
76.6.1.1. Installation

Il faut télécharger Axis 1.x (par exemple le fichier axis-bin-1_4.zip pour la version 1.4) sur le site et décompresser le contenu de l'archive dans un répertoire du système.

Axis s'utilise en tant qu'application web dans un conteneur web. Pour un environnement de développement avec Tomcat, le plus simple est de copier le répertoire axis contenu dans le sous-répertoire webapps issu de la décompression dans le répertoire des applications web du conteneur (le répertoire webapps pour le serveur Tomcat) et de redémarrer le serveur.

Pour vérifier la bonne installation, il suffit d'ouvrir un navigateur sur l'url de l'application web axis :

```
http://localhost:8080/axis/index.html
```



Apache-AXIS

Language: [en] [ja]

Hello! Welcome to Apache-Axis.

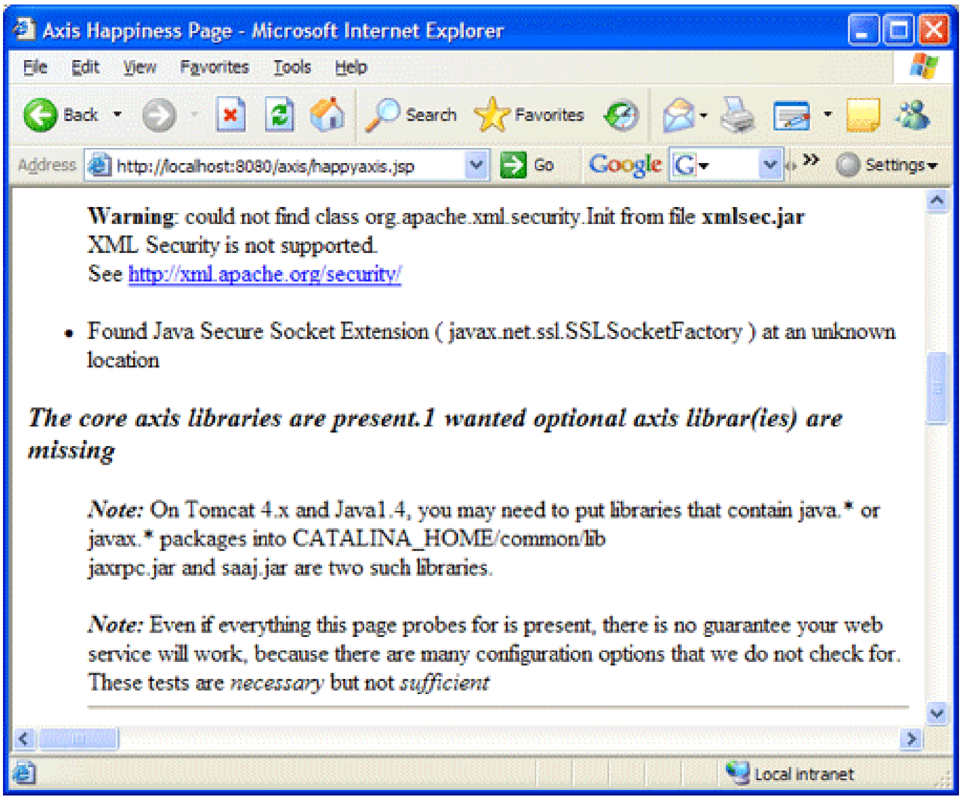
What do you want to do today?

- Validation - Validate the local installation's configuration
  see below if this does not work.
- List - View the list of deployed Web services
- Call - Call a local endpoint that list's the caller's http headers (or see its WSDL).
- Visit - Visit the Apache-Axis Home Page
- Administer Axis - [disabled by default for security reasons]
- SOAPMonitor - [disabled by default for security reasons]

To enable the disabled features, uncomment the appropriate declarations in WEB-INF/web.xml in the webapplication and restart it.

Un clic sur le lien « List » permet de voir les services web qui sont installés.

Un clic sur le lien « Validation » permet d'exécuter une JSP qui fait un état des lieux de la configuration du conteneur et des API nécessaires et optionnelles accessibles.


Axis Happiness Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:8080/axis/happyaxis.jsp Go
Google Settings
Local intranet

Warning: could not find class org.apache.xml.security.Init from file **xmlsec.jar**
XML Security is not supported.
See http://xml.apache.org/security/

- Found Java Secure Socket Extension (javax.net.ssl.SSLSocketFactory) at an unknown location

The core axis libraries are present.1 wanted optional axis librar(ies) are missing

Note: On Tomcat 4.x and Java1.4, you may need to put libraries that contain java.* or javax.* packages into CATALINA_HOME/common/lib
jaxrpc.jar and saaj.jar are two such libraries.

Note: Even if everything this page probes for is present, there is no guarantee your web service will work, because there are many configuration options that we do not check for. These tests are *necessary* but not *sufficient*

76.6.1.2. La mise en oeuvre côté serveur

Axis 1.x propose deux méthodes pour déployer un service web :

- le déploiement automatique d'une classe Java dont l'extension est .jws
- l'utilisation d'un fichier WSDD avec la classe d'implémentation

76.6.1.2.1. Mise en oeuvre côté serveur avec JWS

Axis propose une solution pour facilement et automatiquement déployer une classe Java en tant que service web. Il suffit simplement d'écrire la classe, de remplacer l'extension .java en .jws (java web service) et de copier le fichier dans le répertoire de la webapp axis.

Remarque : il ne faut pas compiler le fichier .jws

76.6.1.2.2. Mise en oeuvre côté serveur avec un descripteur de déploiement

Cette solution est un peu moins facile à mettre en oeuvre mais elle permet d'avoir un meilleur contrôle sur le déploiement du service web.

Il faut écrire la classe Java qui va contenir les traitements proposés par le service web.

Exemple :

```
public class MonServiceWebAxis2{

  public String message(String msg){
    return "Bonjour "+msg;
  }
}
```

Il faut compiler cette classe et mettre le fichier .class dans le répertoire WEB-INF/classes de la webapps axis.

Il faut créer le fichier WSDD qui va contenir la description du service web.

Exemple : le fichier deployMonServiceWebAxis2.wsdd

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
            xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
  <service name="monServiceWebAxis2" provider="java:RPC">
    <parameter name="className" value="MonServiceWebAxis2"/>
    <parameter name="allowedMethods" value="*"/>
  </service>
</deployment>
```

Il faut ensuite déployer le service web en utilisant l'application AdminClient fournie par Axis.

Résultat :

```
C:\java\jwsdp-1.1\webapps\axis\WEB-INF\classes>java org.apache.axis.client.Admin
Client deployMonServiceWebAxis2.wsdd
- Processing file deployMonServiceWebAxis2.wsdd
- <Admin>Done processing</Admin>
```

L'extension wsdd signifie WebService Deployment Descriptor.

C'est un document xml dont le tag racine est deployment.

Les informations relatives au service web sont définies dans le tag service qui possède plusieurs attributs notamment :

- name :
- provider :

Plusieurs informations doivent être fournies avec un tag parameter qui possède les attributs name et value :

- className : le nom pleinement qualifié de la classe d'implémentation du service web
- allowedMethods : précise les méthodes qui sont exposées. Le caractère étoile permet d'indiquer toutes les méthodes

76.6.1.3. Mise en oeuvre côté client

Pour faciliter l'utilisation d'un service web, Axis propose l'outil WSDL2Java qui génère automatiquement à partir d'un document WSDL des classes qui encapsulent l'appel à un service web. Grace à ces classes, l'appel d'un service web par un client ne nécessite que quelques lignes de code.

Résultat :

```
C:\java\jwsdp-1.1\webapps\axis\WEB-INF\classes>java org.apache.axis.wsdl.WSDL2Ja
va
http://localhost:8080/axis/services/monServiceWebAxis2?wsdl
```

L'utilisation de l'outil WSDL2Java nécessite une url vers le document WSDL qui décrit le service web. Il génère à partir de ce fichier plusieurs classes dans le package localhost. Ces classes sont utilisées dans le client pour appeler le service web.

Résultat :

```
C:\java\jwsdp-1.1\webapps\axis\WEB-INF\classes>java org.apache.axis.wsdl.WSDL2Ja
va http://localhost:8080/axis/services/monServiceWebAxis2?wsdl
```

Il faut utiliser les classes générées pour appeler le service web.

Exemple :

```
import localhost.MonServiceWebAxis2;
import localhost.*;

public class MonServiceWebAxis2Client{

  public static void main(String[] args) throws Exception{
    MonServiceWebAxis2Service locator = new MonServiceWebAxis2ServiceLocator();
    MonServiceWebAxis2 monsw = locator.getmonServiceWebAxis2();
    String s = monsw.message("Jean Michel");
    System.out.println(s);
  }
}
```

Résultat :

```
C:\java\jwsdp-1.1\webapps\axis\WEB-INF\classes>javac MonServiceWebAxis2client.java
C:\java\jwsdp-1.1\webapps\axis\WEB-INF\classes>java MonServiceWebAxis2Client

Bonjour Jean Michel
```

Axis propose une API regroupée dans le package org.apache.axis.client pour faciliter l'appel de services web par un client.

Exemple :

```
package fr.jmdoudoux.dej.axis;

import java.rmi.RemoteException;

import javax.xml.namespace.QName;
import javax.xml.rpc.ServiceException;

import org.apache.axis.client.Call;
import org.apache.axis.client.Service;

public class TestCalculerManuel {

  public static void main(String[] args) {
    Service service = new Service();
    Call call;

    try {
      call = (Call) service.createCall();
      String endpoint = "http://localhost:8080/TestWS/services/Calculer";

      call.setTargetEndpointAddress(endpoint);
      call.setOperationName(new QName("additionner"));
      long resultat = (Long) call.invoke(new Object[] { 10, 20 });

      System.out.println("resultat = " + resultat);
    } catch (ServiceException e) {
      e.printStackTrace();
    } catch (RemoteException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
26 déc. 2006 11:22:32 org.apache.axis.utils.JavaUtils isAttachmentSupported
ATTENTION: Unable to find required classes (javax.activation.DataHandler
and javax.mail.internet.MimeMultipart). Attachment support is disabled.
resultat = 30
```

La classe Call permet l'invocation d'une méthode d'un service web. Une instance de cette classe est obtenue en utilisant la méthode createCall() d'un objet de type Service.

La méthode setTargetEndpointAddress() permet de préciser l'url du service web à invoquer.

La méthode setOperationName() permet de préciser le nom de l'opération à invoquer.

La méthode invoke() permet de réaliser l'invocation du service web proprement dit.

Pour faciliter cette mise en oeuvre, Axis fournit l'outil wsdl2java qui génère des classes et interfaces à partir du WSDL du service qui sera à invoquer. Ces classes implémentent un proxy qui facilite l'invocation du service web.

Exemple : code client mettant en oeuvre le proxy généré

```
package fr.jmdoudoux.dej.axis;

import java.rmi.RemoteException;
import javax.xml.rpc.ServiceException;

public class TestCalculerGenere {

  public static void main(String[] args) {
    CalculerServiceLocator locator = new CalculerServiceLocator();
    long resultat;
    Calculer service;

    try {
      service = locator.getCalculer();
      resultat = service.additionner(10, 20);
      System.out.println("resultat = " + resultat);
    } catch (ServiceException e) {
      e.printStackTrace();
    } catch (RemoteException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
26 déc. 2006 11:22:32 org.apache.axis.utils.JavaUtils isAttachmentSupported
ATTENTION: Unable to find required classes (javax.activation.DataHandler
and javax.mail.internet.MimeMultipart). Attachment support is disabled.
resultat = 30
```

Le proxy généré encapsule toute la mécanique d'appel. Un objet de type ServiceLocator facilite l'obtention du endpoint. L'utilisation du proxy rend le code plus simple, plus compréhensible et plus évolutif puisqu'il est généré.

76.6.1.4. L'outil TCPMonitor

Cet outil agit comme un proxy qui permet de visualiser les requêtes http échangées entre un client et un serveur.

Résultat :

```
C:\java\axis-1_4\lib>java -cp ./axis.jar org.apache.axis.utils.tcpmon 1234 localhost 8080
```

Les paramètres optionnels pouvant être fournis sont :

- le port écouté sur le client
- le hostname du serveur
- le port du serveur

Si aucun paramètre n'est fourni, l'outil affiche une première fenêtre qui permet de saisir les informations requises.

L'outil écoute les requêtes faites sur un port local, les affiche puis ces requêtes sont envoyées au serveur. Les réponses suivent le chemin inverse pour permettre leur affichage.

Cet outil est pratique pour afficher le contenu des requêtes et réponses http échangées lors des invocations.

76.6.2. Apache Axis 2

Axis 2 est le successeur du projet Axis : le projet a été complètement réécrit pour proposer une architecture plus modulaire.

Il propose un modèle de déploiement spécifique : les services web peuvent être packagés dans un fichier ayant l'extension.aar (Axis ARchive) ou contenus dans un sous-répertoire du répertoire WEB-INF/services. La configuration se fait dans le fichier META-INF/services.xml

Le runtime d'Axis 2 est une application web qui peut être utilisée dans n'importe quel serveur d'applications Java EE et même un conteneur web comme Apache Tomcat.

Des modules complémentaires permettent d'enrichir le moteur en fonctionnalités notamment le support de certaines spécifications WS-*. Chaque module est packagé dans un fichier avec l'extension .mar

Axis 2 permet de choisir le framework de binding XML/Objets.

76.6.3. Xfire

XFire était un projet open source initié par la communauté CodeHaus

Ce projet n'est plus maintenu car il a été repris par le projet CXF d'Apache.

76.6.4. Apache CXF

Apache CXF est né de la fusion des projets XFire et Celtix.

L'url du projet est **https://cxf.apache.org/**

CXF propose un support de plusieurs standards des services web notamment, SOAP 1.1 et 1.2, WSDL 1.1 et 1.2, le WS-I Basic Profile, MTOM, WS-Addressing, WS-Policy, WS-ReliableMessaging, et WS-Security.

CXF utilise une api propriétaire mais implémente aussi les spécifications de JAX-WS. CXF propose plus qu'une implémentation d'un moteur SOAP en proposant un framework complet pour le développement de services

Ses principaux objectifs sont la facilité d'utilisation, les performances, l'extensibilité et l'intégration dans d'autres systèmes. CXF utilise le framework Spring.

CXF est utilisé dans d'autres projets notamment ServiceMix et Mule.

76.6.5. JWSDP (Java Web Service Developer Pack)

Le Java Web Services Developer Pack (JWSDP) est un ensemble d'outils et d'API fournis par Sun qui permet de faciliter le développement, le déploiement et le test des services web et des applications web avec Java.

Le JWSDP contient les outils suivants :

- Apache Tomcat
- Java WSDP Registry Server (serveur UDDI)
- Web application development tool
- Apache Ant
- wscompile, wsdeploy,
- ...

La plupart de ces éléments peuvent être installés manuellement séparément. Le JWSDP propose un pack qui les regroupe en une seule installation et propose en plus des outils spécifiquement dédiés au développement de services web.

Le JWSDP contient les API particulières suivantes :

- Java XML Pack : Java API for XML Processing (JAXP), Java API for XML-based RPC (JAX-RPC), Java API for XML Messaging (JAXM), Java API for XML Registries (JAXR)
- Java Architecture for XML Binbing (JAXB)
- Java Secure Socket (JSSE)
- SOAP with Attachments API for Java (SAAJ)

JWSDP fournit aussi toutes les APIs nécessaires aux développements d'applications Web notamment les API Servlet/JSP, JSTL et JSF.

Remarque : le projet GlassFish remplace le JWSDP.

76.6.5.1. L'installation du JWSDP 1.1

Pour pouvoir l'utiliser, il faut au minimum un jdk 1.3.1. Il faut télécharger sur le site de Sun le fichier jwsdp-1_1-windows-i586.exe et l'exécuter.

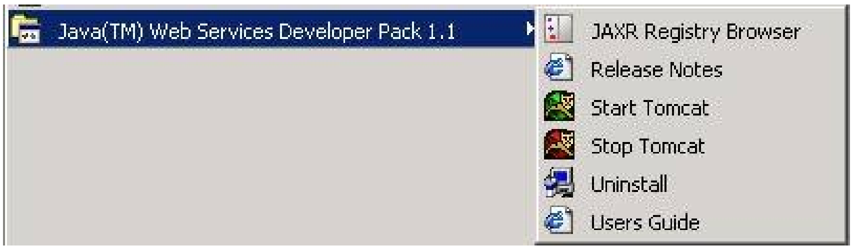
Un assistant guide l'installation :

- Cliquer sur "Suivant".
- Lire le contrat de licence, sélectionner "Approve" et cliquer sur "Suivant".
- Sélectionner le JDK à utiliser et cliquer sur "Suivant"
- Dans le cas de l'utilisation d'un proxy, il faut renseigner les informations le concernant. Cliquer sur "Suivant".
- Sélectionner le répertoire d'installation et cliquer sur "Suivant".
- Sélectionner le type d'installation et cliquer sur "Suivant".
- Il faut saisir un nom d'utilisation qui sera l'administrateur et son mot de passe et cliquer sur "Suivant".
- L'assistant affiche un récapitulatif des options choisies. Cliquer sur "Suivant".

- Cliquer sur "Suivant".
- Cliquer sur "Suivant".
- Cliquer sur "Fin".

76.6.5.2. L'exécution du serveur

L'installation a créé une entrée dans le menu "Démarrer/Programmes".

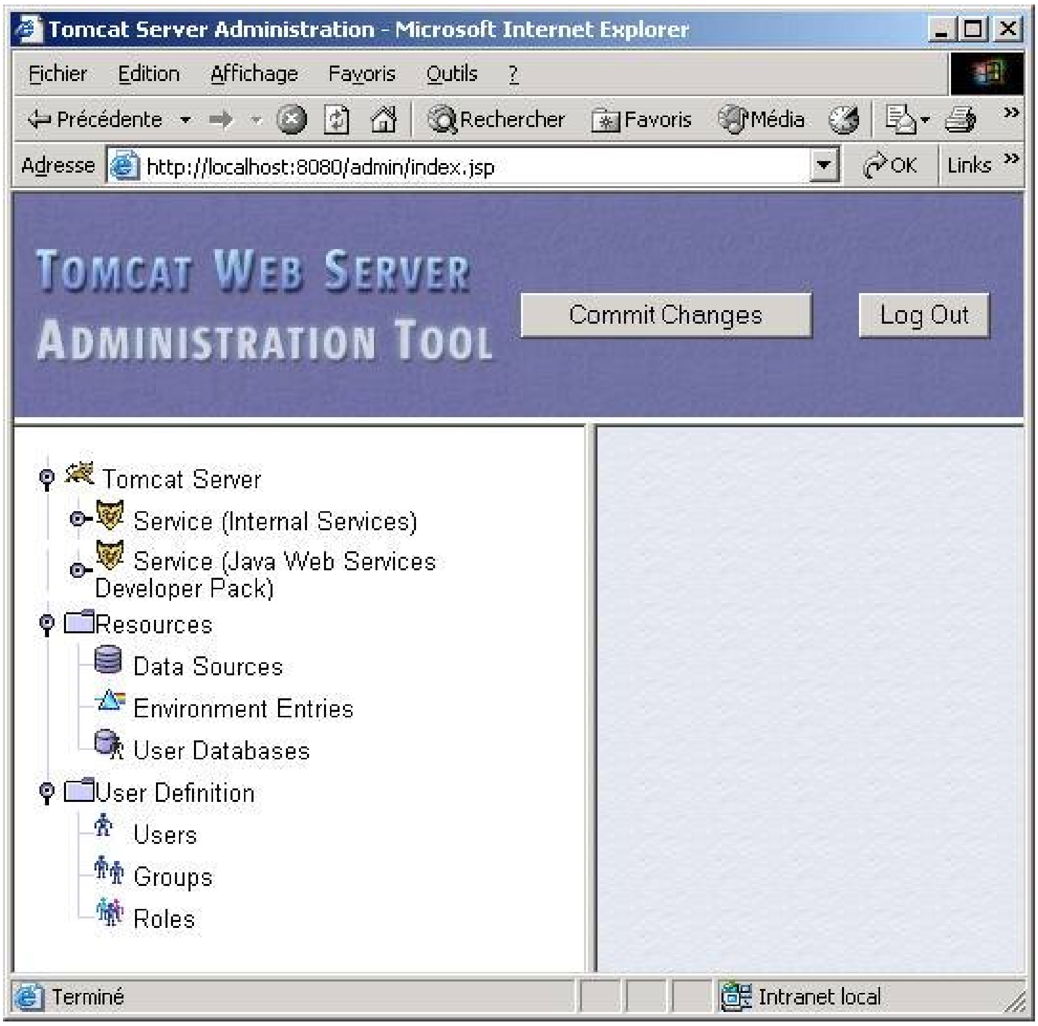


Pour lancer le serveur d'applications Tomcat, il faut utiliser l'option Start Tomcat.

Attention, les ports 8080 et 8081 ne doivent pas être occupés par un autre serveur.

Pour accéder à la console d'administration, il faut lancer un navigateur sur l'url http://localhost:8081/admin. Si la page ne s'affiche pas, il faut aller voir dans le fichier catalina.out contenu dans le répertoire logs où a été installé le JWSDP.

Il faut saisir le nom de l'utilisateur et le mot de passe défini lors de l'installation de JWSDP.



Cette console permet de modifier les paramètres du JWSDP.

76.6.5.3. L'exécution d'un des exemples

Il faut créer un fichier build.properties dans le répertoire home (c:\document and settings\user_name) qui contient :

```
username=
password=
```

Il faut s'assurer que le chemin C:\java\jwsdp-1_0_01\bin est en premier dans le classpath surtout si une autre version de Ant est déjà installée sur la machine

Il faut lancer Tomcat puis suivre les étapes proposées ci-dessous :

Résultat :

```
C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hello>dir
Le volume dans le lecteur C s'appelle SYSTEM
Le numéro de série du volume est 18AE-3A71
Répertoire de C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hello
03/01/2003  13:37       <DIR>          .
03/01/2003  13:37       <DIR>          ..
01/08/2002  14:16                  309 build.properties
01/08/2002  14:17                  496 build.xml
01/08/2002  14:17                  222 config.xml
01/08/2002  14:16                2 342 HelloClient.java
01/08/2002  14:17                1 999 HelloIF.java
01/08/2002  14:16                1 995 HelloImpl.java
01/08/2002  14:17                  545 jaxrpc-ri.xml
01/08/2002  14:17                  421 web.xml
               8 fichier(s)            8 329 octets
               2 Rép(s)     490 983 424 octets libres
C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hello>ant compile-server
Buildfile: build.xml
prepare:
     [echo] Creating the required directories....
    [mkdir] Created dir: C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hell
o\build\client\hello
    [mkdir] Created dir: C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hell
o\build\server\hello
    [mkdir] Created dir: C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hell
o\build\shared\hello
    [mkdir] Created dir: C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hell
o\build\wsdeploy-generated
    [mkdir] Created dir: C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hell
o\dist
    [mkdir] Created dir: C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hell
o\build\WEB-INF\classes\hello
compile-server:
     [echo] Compiling the server-side source code....
    [javac] Compiling 2 source files to C:\java\jwsdp-1_0_01\docs\tutorial\examp
les\jaxrpc\hello\build\shared
BUILD SUCCESSFUL
Total time: 7 seconds
C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hello>ant setup-web-inf
Buildfile: build.xml
setup-web-inf:
     [echo] Setting up build/WEB-INF....
   [delete] Deleting directory C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrp
c\hello\build\WEB-INF
     [copy] Copying 2 files to C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrp
c\hello\build\WEB-INF\classes\hello
     [copy] Copying 1 file to C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc
\hello\build\WEB-INF
     [copy] Copying 1 file to C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc
\hello\build\WEB-INF
BUILD SUCCESSFUL
Total time: 2 seconds
C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hello>ant package
Buildfile: build.xml
package:
     [echo] Packaging the WAR....
```

```
      [jar] Building jar: C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hel
lo\dist\hello-portable.war
BUILD SUCCESSFUL
Total time: 2 seconds
C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hello>ant process-war
Buildfile: build.xml
set-ws-scripts:
process-war:
     [echo] Running wsdeploy....
     [exec] info: created temporary directory: C:\java\jwsdp-1_0_01\docs\tutoria
l\examples\jaxrpc\hello\build\wsdeploy-generated\jaxrpc-deploy-b5e49c
     [exec] info: processing endpoint: MyHello
     [exec] Note: sun.tools.javac.Main has been deprecated.
     [exec] 1 warning
     [exec] info: created output war file: C:\java\jwsdp-1_0_01\docs\tutorial\ex
amples\jaxrpc\hello\dist\hello-jaxrpc.war
     [exec] info: removed temporary directory: C:\java\jwsdp-1_0_01\docs\tutoria
l\examples\jaxrpc\hello\build\wsdeploy-generated\jaxrpc-deploy-b5e49c
BUILD SUCCESSFUL
Total time: 15 seconds
C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hello>ant deploy
Buildfile: build.xml
deploy:
   [deploy] OK - Installed application at context path /hello-jaxrpc
   [deploy]
BUILD SUCCESSFUL
Total time: 7 seconds
C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hello>ant generate-stubs
Buildfile: build.xml
set-ws-scripts:
prepare:
     [echo] Creating the required directories....
generate-stubs:
     [echo] Running wscompile....
     [exec] Note: sun.tools.javac.Main has been deprecated.
     [exec] 1 warning
BUILD SUCCESSFUL
Total time: 14 seconds
C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hello>ant compile-client
Buildfile: build.xml
prepare:
     [echo] Creating the required directories....
compile-client:
     [echo] Compiling the client source code....
    [javac] Compiling 1 source file to C:\java\jwsdp-1_0_01\docs\tutorial\exampl
es\jaxrpc\hello\build\client
BUILD SUCCESSFUL
Total time: 4 seconds
C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hello>ant jar-client
Buildfile: build.xml
jar-client:
     [echo] Building the client JAR  file....
      [jar] Building jar: C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hel
lo\dist\hello-client.jar
BUILD SUCCESSFUL
Total time: 2 seconds
C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hello>ant run
Buildfile: build.xml
run:
     [echo] Running the hello.HelloClient program....
     [java] Hello Duke!
BUILD SUCCESSFUL
Total time: 5 seconds

C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hello>
```

76.6.6. Java EE 5

Java EE 5 utilise une nouvelle API pour le développement de services web : JAX-WS (Java API for XML Web Services).

76.6.7. Java SE 6

Java SE 6 fournit en standard une implémentation de JAX-WS 2.0 permettant ainsi de consommer mais aussi de produire des services web uniquement avec la plate-forme SE.

L'écriture et le déploiement d'un service web suit plusieurs étapes.

Il faut écrire la classe du service web en utilisant les annotations de JAX-WS.

Exemple :

```
package fr.jmdoudoux.dej.ws;

import javax.jws.WebService;

@WebService
public class TestWS {

  public String Saluer(final String nom) {
    return "Bonjour " + nom;
  }
}
```

La classe javax.xml.ws.Endpoint encapsule le endpoint d'un service web permettant ainsi son accès.

La méthode publish() permet de publier un endpoint associé à l'url fournie en paramètre.

Exemple :

```
package fr.jmdoudoux.dej.ws;

import javax.xml.ws.Endpoint;

public class Main {

  public static void main(String[] args) {

    System.out.println("Lancement du serveur web");

    Endpoint.publish("http://localhost:8080/ws/TestWS", new TestWS());
  }
}
```

Il faut compiler la classe.

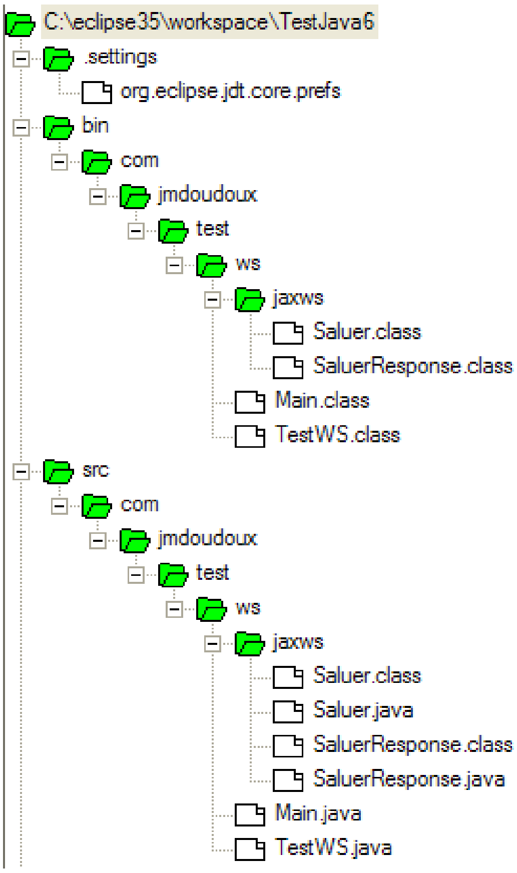
Il faut ensuite utiliser l'outil wsgen pour générer les classes JAXB qui vont mapper les requêtes et réponses des messages Soap.

Résultat :

```
C:\eclipse35\workspace\TestJava6\bin>wsgen -cp . -d ../src fr.jmdoudoux.dej.ws.TestWS
```

Dans l'exemple, deux classes sont générées dans le package fr.jmdoudoux.dej.ws.jaxws :

- Saluer : pour encapsuler la requête
- SaluerResponse : pour encapsuler la réponse



```
C:\eclipse35\workspace\TestJava6
  .settings
    org.eclipse.jdt.core.prefs
  bin
    com
      jmdoudoux
        test
          ws
            jaxws
              Saluer.class
              SaluerResponse.class
            Main.class
            TestWS.class
  src
    com
      jmdoudoux
        test
          ws
            jaxws
              Saluer.class
              Saluer.java
              SaluerResponse.class
              SaluerResponse.java
            Main.java
            TestWS.java
```

Il faut alors exécuter la classe Main : un serveur web minimaliste est lancé et le service web y est déployé.

Attention, l'environnement d'exécution doit être un JDK.

Il suffit alors d'ouvrir l'url http://localhost:8080/ws/TestWS?wsdl dans un navigateur

Le navigateur affiche alors le contenu du WSDL qui décrit le service web.

Le service web peut alors être consommé par un client, tant que l'application est en cours d'exécution.

Exemple : Le message Soap de la requête

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ws="http://ws.test.jmdoudoux.com/">
   <soapenv:Header/>
   <soapenv:Body>
      <ws:Saluer>
         <arg0>JM</arg0>
      </ws:Saluer>
   </soapenv:Body>
</soapenv:Envelope>
```

Exemple : Le message Soap en réponse

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:SaluerResponse xmlns:ns2="http://ws.test.jmdoudoux.com/">
         <return>Bonjour JM</return>
      </ns2:SaluerResponse>
   </S:Body>
</S:Envelope>
```

76.6.8. Le projet Metro et WSIT

Le projet Metro est une pile pour services web utilisée dans le serveur d'applications GlassFish V2 et V3. Metro est l'implémentation de référence de JAX-WS. Metro est livré avec GlassFish mais il est possible de l'utiliser dans d'autres serveurs d'applications ou conteneurs web, par exemple Tomcat. Dans ce dernier cas, il faut ajouter les bibliothèques de Metro et JAXB.

Metro est composé de deux éléments :

- Une implémentation de JAX-WS pour le support des services web
- Le projet Tango qui est une implémentation de certaines spécifications WS-*

Le projet Tango est une implémentation open source des spécifications Reliability, Security et Transaction des spécifications WS-*, ce qui facilite l'interopérabilité avec le framework WCF (Windows Communication Foundation) de Microsoft .Net versions 3.0 et ultérieures.

WSIT (Web Service Interoperability) est un projet commun entre Sun et Microsoft pour garantir l'intéropérabilité des piles de services web des plates-formes Java et .Net (avec le Windows Communication Framework).

Cette interopérabilité est assurée car Metro et WCF supportent tous les deux plusieurs spécifications WS-* :

- WS-Addressing
- WS-Policy
- WS-Security
- WS-Transaction
- WS-Reliable Messaging
- WS-Trust
- WS-SecureConversation

La mise en oeuvre de ces spécifications via WSIT repose sur une configuration dans un fichier XML. Le contenu de ce fichier peut être fastidieux à créer ou à modifier : Netbeans propose des assistants graphiques qui facilitent grandement leur mise en oeuvre.

76.7. Inclure des pièces jointes dans SOAP

Pour inclure des données binaires importantes dans un message SOAP, il faut utiliser le mécanisme des pièces jointes (attachment).

Malheureusement, ce mécanisme est implémenté par plusieurs standards :

- SOAP With Attachments : définis par le W3C dans la version 1.1 de SOAP
- XOP/MTOM : définis par le W3C dans la version 1.2 de SOAP

MTOM devient le standard utilisé par Java (JAX-WS) et .Net (WSE 3.0)

76.8. WS-I

Les nombreuses spécifications concernant les services web sont fréquemment incomplètes ou peu claires : il en résulte plusieurs incompatibilités lors de leur mise en oeuvre.

Le consortium WS-I (Web Service Interoperability) **http://www.ws-i.org/** a été créé pour définir des profiles qui sont des recommandations dont le but est de faciliter l'interopérabilité des services web entre plateformes, systèmes d'exploitation et langages pour promouvoir ces normes.

Le WS-I a définit plusieurs spécifications :

- WS-I Basic Profile
- WS-I Basic Security Profile
- Simple Soap Binding Profile
- ...

Le site web est à l'url : www.ws-i.org

76.8.1. WS-I Basic Profile

WS-I Basic Profile est un ensemble de recommandations dont le but est d'améliorer l'interopérabilité entre les différents moteurs SOAP.

La suite de cette section sera développée dans une version future de ce document

76.9. Les autres spécifications

Les spécifications SOAP et WSDL permettent de réaliser des échanges de messages basiques. L'accroissement de l'utilisation des services web a fait émerger la nécessité de fonctionnalités supplémentaires telles que la gestion de la sécurité, des transactions, de la fiabilité des messages, ...

Les spécifications désignées sous l'acronyme WS-* concernent les spécifications de seconde génération des services web (elles étendent les spécifications de la première génération de spécifications constituée par SOAP, WSDL, UDDI). L'abréviation WS-* est communément utilisée car la majorité de ces spécifications commence par WS-.

De nombreuses autres spécifications sont en cours d'élaboration et de tentatives de standardisation ou de reconnaissance par le marché.

Fréquemment ces spécifications sont complémentaires ou dépendantes voire même dans quelques cas concurrentes car elles sont soutenues par des acteurs du marché ou des organismes de standardisation différents. Il est généralement nécessaire d'utiliser plusieurs de ces spécifications pour permettre de répondre aux besoins notamment en terme de sécurité, fiabilité, ...

Il est aussi très important de tenir compte de la maturité d'une spécification avant de la mettre en oeuvre.

Ces spécifications permettent de mettre en oeuvre des scénarios complexes impliquant l'utilisation de services web.

Toutes ces spécifications requièrent l'utilisation de SOAP.

La suite de cette section sera développée dans une version future de ce document

77. Les WebSockets

Chapitre 77

Niveau : 🟩🟩⬜⬜⬜⬜ Elémentaire

Une WebSocket est une spécification d'un protocole permettant une communication bidirectionnelle et full duplex sur une seule socket TCP entre un client et un serveur.

Initialement développé pour HTML 5, WebSocket a été normalisé par l'IETF et le W3C. Tous les navigateurs récents implémentent et supportent les WebSockets. Ce protocole permet notamment d'implémenter facilement et de manière standard l'envoi de données en mode Push à l'initiative du serveur.

Ce chapitre contient plusieurs sections :

♦ Les limitations du protocole HTTP
♦ La spécification du protocole WebSocket
♦ La connexion à une WebSocket
♦ La mise en oeuvre des WebSockets

77.1. Les limitations du protocole HTTP

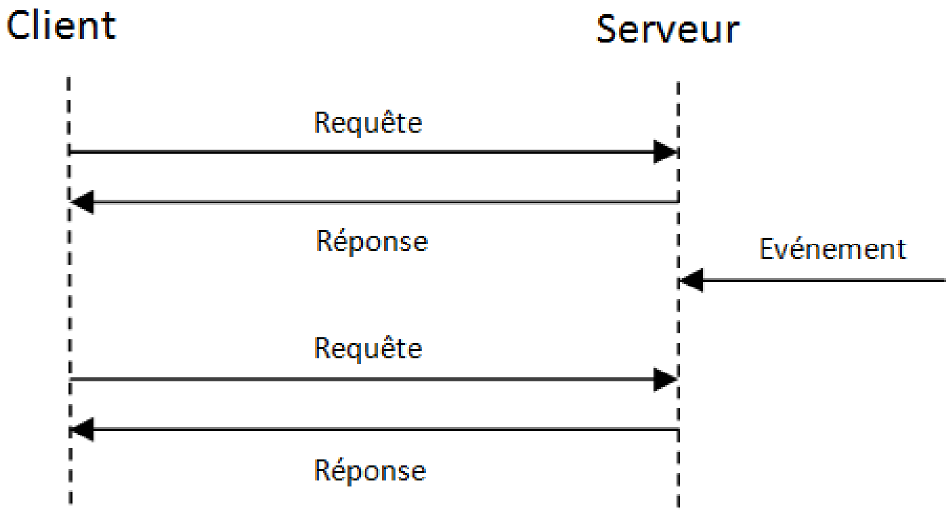
HTTP est un protocole sans état qui fonctionne sur le modèle requête/réponse. Avant la version 1.1 d'HTTP, chaque requête faite au serveur utilise une nouvelle connexion. A partir d'HTTP 1.1, il est possible d'utiliser des connexions persistantes qui permettent au client d'utiliser la même connexion pour obtenir les autres éléments de la page.

HTTP est le protocole standard utilisé pour le Web : il a été conçu pour obtenir des éléments du web. Il répond à de nombreux besoins mais il possède plusieurs inconvénients notamment pour une utilisation dans une application web interactive :

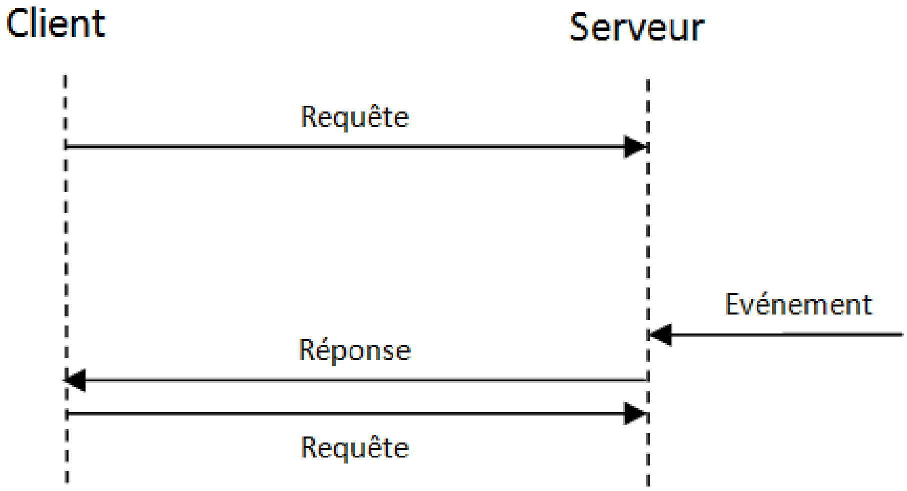
- half duplex : le protocole repose sur le modèle requête/réponse. Le client envoie une requête au serveur qui répond en lui renvoyant une réponse. Le client doit attendre la réponse. La transmission de données ne peut se faire que dans une direction en même temps.
- verbeux : chaque requête et réponse HTTP doit avoir un en-tête (header) contenant plus au moins d'informations qui fait parti des données échangées, ce qui augmente le trafic sur le réseau.
- il n'est pas possible d'utiliser un mode push de la part du serveur (le serveur envoie à son initiative des données au client).

Plusieurs techniques ont été développées pour contourner cette limitation :

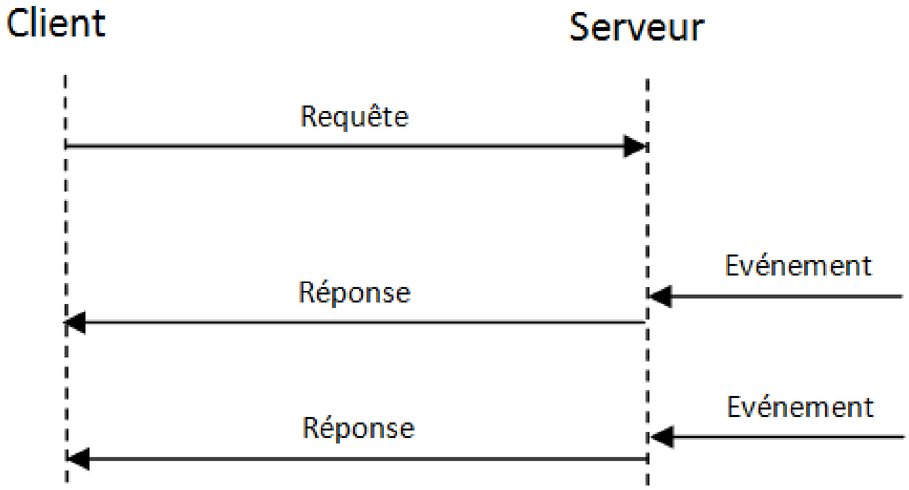
- polling : le client effectue périodiquement des requêtes synchrones au serveur pour obtenir des données ou pas selon qu'il y en ait de disponible. Cette technique est simple mais peu efficace car elle nécessite beaucoup de connexions selon la fréquence utilisée par le client pour obtenir potentiellement peu de données. Cette technique peut être intéressante si les données sont périodiquement modifiées côté serveur, ce qui permet de synchroniser les requêtes sur les modifications. Malheureusement ce cas de figure est plutôt rare et généralement de nombreuses requêtes sont inutiles.



- long polling : le client ouvre une connexion et envoie une requête HTTP au serveur qui ne renvoie la réponse que si un événement force l'envoi de données au client ou après un certain timeout. Le nombre de requêtes/réponses peut ainsi être réduit sauf si le nombre d'événements est très important



- Streaming : le client envoie une requête au serveur qui maintient le flux de la réponse ouvert en y envoyant des données au besoin. La durée du maintient de la réponse ouverte pour être limitée par un timeout ou infini. Cette technique reposant sur HTTP, elle pose généralement des soucis avec certains éléments réseaux comme les firewalls ou les proxys



- Server Side Event : cette technologie permet à un navigateur de recevoir des mises à jour de la part d'un serveur. Elle est supportée par la majorité des navigateurs sauf Internet Explorer. HTML 5 propose de standardiser une API pour utiliser SSE.

Comet est un concept dont le but est de permettre à un serveur d'envoyer à son initiative des données à un navigateur. Plusieurs techniques sont utilisées pour répondre au concept Comet (streaming, hidden iframe, Ajax avec long polling, ...).

Cependant, il était nécessaire de définir un standard qui permette la communication entre les clients et le serveur de manière bi-directionnelle utilisant un canal en mode full duplex. Le mode full-duplex indique qu'une WebSocket permet d'envoyer des messages du côté client et serveur indépendamment l'un de l'autre.

En 2011, l'IETF a défini le protocole WebSocket sous la RFC 6455. Depuis, les principaux navigateurs implémentent le protocole WebSocket et plusieurs implémentations sont disponibles pour la plate-forme Java.

77.2. La spécification du protocole WebSocket

Une WebSocket permet l'échange de données entre un client et un serveur de manière asynchrone, bidirectionnelle en mode full duplex utilisant une connections TCP.

Les WebSockets sont typiquement utilisées pour envoyer de petits messages.

La spécification du protocole WebSocket est définie dans la RFC 6455, publiée en décembre 2011.

L'utilisation d'une WebSocket dans une page web peut se faire avec l'API JavaScript dédiée proposée par HTML 5 : ceci facilite son adoption dans les applications web.

La demande d'interactivité des pages HTML se trouve limitée par le protocole HTTP :

- HTTP s'utilise en mode half duplex : il repose sur un modèle requête/réponse
- HTTP est verbeux notamment car chaque requête et réponse contient un en-tête contenant un certain nombre d'informations

Les WebSockets sont plus efficaces et sont plus performantes que les autres solutions :

- elles requièrent moins de bande passante car elles ne requièrent pas d'en-tête dans chaque message
- la latence est réduite.
- elles permettent de mettre en place des solutions quasi temps réel pour recevoir des données

Une WebSocket est un protocole réseau reposant sur TCP. Le protocole est composé de deux phases :

- handshake : c'est une requête/réponse utilisant HTTP avec l'option upgrade du protocole qui permet d'établir une connexion entre un client et un serveur
- data transfer : échange de données au format texte ou binaire en mode bidirectionnel, full duplex. Le format et le contenu des données échangées entre le client et le serveur est libre : les deux parties doivent donc connaître le format utilisé pour pouvoir exploiter les données.

Les données de type texte reçues d'une websocket sont encodées en UTF-8.

La mise en oeuvre des WebSockets requière plusieurs étapes :

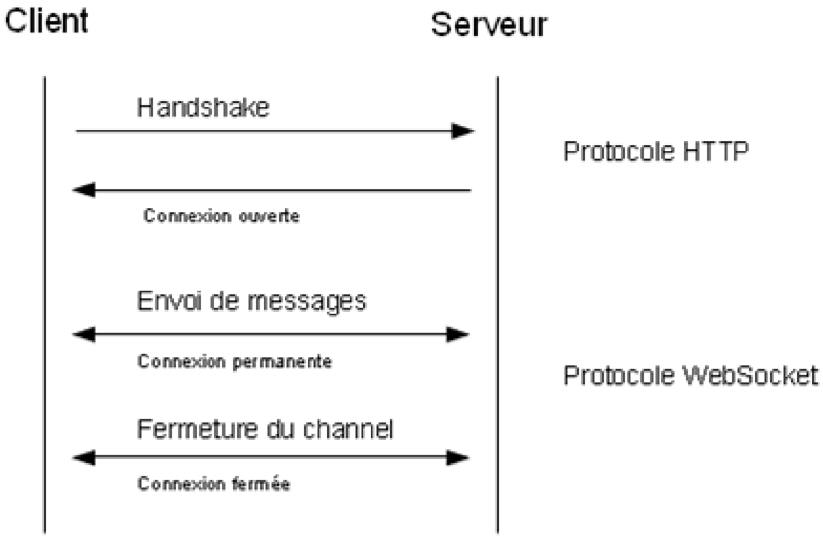
- établir une connexion
- envoyer des messages côté client et serveur (bi-directionnel) indépendamment les uns des autres (full duplex)
- fin de la connexion

Les cas d'utilisation des WebSockets sont nombreux : elles sont utilisables dès que des données doivent être envoyées du serveur vers le ou les clients.

77.3. La connexion à une WebSocket

Une connexion WebSocket est initialisée en utilisant le protocole HTTP : chaque connexion à une WebSocket débute par une requête HTTP qui utilise l'option upgrade dans son en-tête. Cette option permet de préciser que le client souhaite que la connexion utilise un autre protocole, en l'occurrence le protocole WebSocket. Cette requête HTTP s'appelle handshake dans le cas de l'utilisation d'une WebSocket.

Lorsque le serveur répond, la connexion est établie et le client et le serveur peuvent envoyer et recevoir des messages.



Le protocole HTTP n'est utilisé que pour établir la connexion d'une WebSocket : une fois la connexion établie le protocole HTTP n'est plus utilisé au profit du protocole WebSocket.

C'est toujours le client qui initie une demande de connexion : le serveur ne peut pas initier de connexions mais il est à l'écoute des clients qui le contacte pour créer une connexion.

Une WebSocket est identifiée par une URI particulière définie dans la RFC dont la syntaxe générale est :

ws(s)://host[:port]path[?param]

L'étape de connexion (Opening Handshake) requiert un unique échange HTTP (requête/réponse) entre le client qui initie la connexion et le serveur. La requête HTTP utilise l'option Upgrade qui permet de demander le changement du protocole utilisé pour les échanges.

La version 1.1 du protocole HTTP doit être utilisée car c'est à partir de cette version que le changement de protocole est supporté.

Exemple : la requête HTTP

```
GET /MaWebApp/echo HTTP/1.1
Cache-Control: no-cache
Connection: Upgrade
Host: localhost:8080
Origin: http://localhost:8080
Pragma: no-cache
Sec-WebSocket-Extensions: x-webkit-deflate-frame
Sec-WebSocket-Key: LwsTSMPv4TKzQscBprG1Iw==
Sec-WebSocket-Version: 13
Upgrade: websocket
User-Agent: Mozilla/5.0 (Windows NT 5.1)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/30.0.1599.101 Safari/537.36
```

La réponse HTTP contient le code 101 pour indiquer que le serveur a changé de protocole pour utiliser le protocole WebSocket.

Exemple : la réponse HTTP

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Sec-WebSocket-Accept:
9JUSDZQDUFa0yLScZ26xQdyzFy4=
Server: GlassFish Server Open Source
Edition  4.0
Upgrade: websocket
X-Powered-By: Servlet/3.1 JSP/2.3
(GlassFish Server Open Source Edition
4.0  Java/Oracle Corporation/1.7)
```

Une fois que le serveur a validé l'utilisation du protocole WebSocket, il n'est plus possible d'utiliser le protocole HTTP et tous les échanges suivants doivent utiliser le protocole WebSocket.

Si la connexion réussie, l'état de la WebSocket passe à l'état connected. Des données peuvent alors être échangées entre les deux endpoints de manière bi-directionnelle en mode full-duplex.

La fermeture de la connexion peut être à l'initiative du endpoint client ou serveur pour permettre de passer la WebSocket à l'état deconnected.

77.4. La mise en oeuvre des WebSockets

Le protocole WebSocket possède de nombreuses implémentations pour permettre sa mise en oeuvre côté client et serveur.

Plusieurs implémentations des WebSockets sont disponibles pour la plate-forme Java :

- Grizzly : https://grizzly.java.net/websockets.html
- WebSocket SDK : https://java.net/projects/websocket-sdk
- Apache Tomcat 7 : http://tomcat.apache.org/tomcat-7.0-doc/web-socket-howto.html
- Webbit : https://github.com/webbit/webbit
- Atmosphere : https://github.com/Atmosphere
- Jetty : http://wiki.eclipse.org/Jetty/Feature/WebSockets
- Netty : https://netty.io/news/2011/11/17/websockets.html
- jWebSocket : https://jwebsocket.org/
- jWamp : https://github.com/ghetolay/jwamp

Les WebSocket ont été standardisées dans la plate-forme Java EE dans les spécifications de la JSR 356. La JSR 356 est ajoutée au Web Profile de Java EE 7. Plusieurs implémentations sont disponibles notamment Tyrus qui est l'implémentation de référence.

Le client peut utiliser n'importe quelle technologie qui propose un support des WebSockets, par exemple:

- un navigateur web grâce à l'API WebSocket en JavaScript développée dans le cadre de HTML 5 par le W3C
- une application développée en Java EE 7 ou ultérieure
- une application standalone par exemple développée en Java SE avec une implémentation de la JSR 356 ou une implémentation qui propose un support des WebSockets
- une application développée dans une technologie qui propose un support des WebSockets (.Net, PHP, ...)

78. L'API WebSocket

Chapitre 78

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Comme fréquemment avec une nouvelle technologie, la communauté Java propose différentes solutions commerciales ou open source (notamment avec le framework Atmosphere), chacune avec une API différente. Une fois que la technologie est suffisamment mature, une spécification standard est développée par le JCP.

Les Websockets n'échappent pas à cette situation : la JSR 356 est une spécification pour la mise en oeuvre des Websockets dans la plate-forme Java aussi bien côté serveur que côté client.

Cette spécification propose les caractéristiques principales suivantes :

- développer des WebSocket Endpoint qui sont des composants Java capable d'utiliser le protocole WebSocket
- utilisation d'annotations ou de l'API pour développer les endpoints
- envoi et consommation de messages au format texte ou binaire
- utilisation de messages entiers ou d'un ensemble de morceaux
- envoi de messages de manière synchrone ou asynchrone
- utilisation d'encodeurs ou décodeurs pour mapper des objets en messages en vice versa

L'API WebSocket est de type event-driven : selon les différents événements qui surviennent durant le cycle de vie de la websocket, des callbacks à implémenter sont invoqués par le conteneur.

Un client Java peut utiliser une implémentation de la JSR 356 pour communiquer par Websockets avec un serveur.

La mise en oeuvre des WebSockets côté client et serveur peut se faire de deux manières :

- utilisation d'annotations sur des POJO
- utilisation d'une API

La JSR 356 est incluse dans les spécifications de la plate-forme Java EE 7 : chaque serveur d'applications doit fournir une implémentation de cette spécification. D'autres implémentations peuvent être utilisées en dehors d'un contexte Java EE, par exemple Tomcat 8 propose un support de la JSR 356.

L'implémentation de référence est le projet Tyrus dont la page officielle est à l'url : https://tyrus.java.net/

Ce chapitre contient plusieurs sections :

♦ Les principales classes et interfaces
♦ Le développement d'un endpoint
♦ Les encodeurs et les décodeurs
♦ Le débogage des WebSockets
♦ Des exemples d'utilisation
♦ L'utilisation d'implémentations

78.1. Les principales classes et interfaces

La communication entre un client et un serveur se fait au moyen d'endpoints : un endpoint côté client et un endpoint côté serveur.

Avec une WebSocket, l'endpoint client est celui qui initialise la connexion. Une fois la connexion établie, les deux endpoints possèdent les mêmes fonctionnalités.

Un endpoint est géré par un conteneur encapsulé dans une instance de type WebSocketContainer. Ce conteneur encapsule plusieurs paramètres relatifs à la communication (timeout par défaut, buffer, ...) et permet la gestion des extensions.

L'interface ServerContainer hérite de l'interface WebSocketContainer. Elle ajoute deux surcharges de la méthode add() qui permettent d'enregistrer des endpoints dans le conteneur.

Il ne doit y avoir qu'une seule instance de type ServerContainer pour une même application.

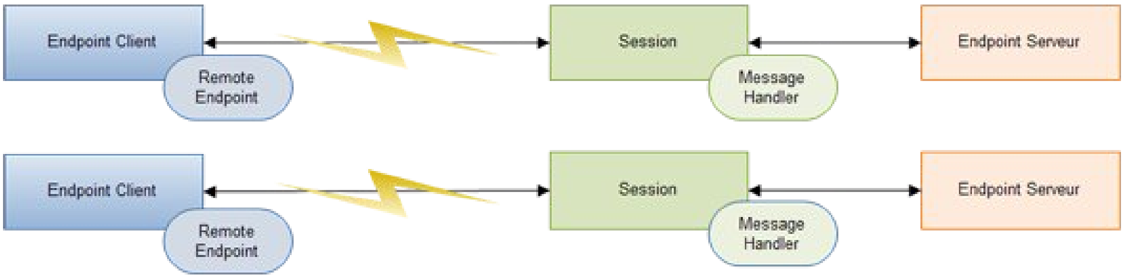
Une implémentation de la JSR 356 qui puisse être exécutée en dehors d'un conteneur web doit fournir sa propre solution pour obtenir une instance de type ServerContainer.

L'ouverture d'un channel pour une socket permet d'obtenir une instance de type javax.websocket.Session qui permet de gérer la communication : enregistrement de handler pour traiter les messages reçus, fermer un channel, obtenir une instance de type RemoteEndpoint pour envoyer des messages, obtenir et modifier des paramètres de configuration, obtenir des informations sur l'état, ...

La conception de l'API assure une séparation des rôles entre le Endpoint qui encapsule le cycle de vie du endpoint (open, close, error) et la gestion des messages reçus qui est assurée par un objet de type MessageHandler.

Les principales classes du package javax.net.websocket sont :

Classe	Rôle
MessageHandler	Gestion des messages entrant d'un endpoint
RemoteEndpoint	Envoie de messages à l'autre endpoint
Session	Encapsule la conversation
Endpoint	Encapsule un endpoint



Les différences entre l'API client et l'API serveur sont minimes : l'API client est un sous-ensemble de l'API serveur.

78.1.1. L'interface javax.websocket.Session

L'interface javax.websocket.Session définit les fonctionnalités relatives à une conversation entre un endpoint et son équivalent distant.

Une instance de type Session est valide tant que la connexion n'est pas fermée : si la session est fermée, une invocation d'une de ses méthodes lève une exception de IllegalStateException

Dès que la connexion est créée lors du handshake, l'implémentation associe au endpoint une instance de type Session. Plusieurs solutions sont utilisables pour obtenir cette instance selon le mode de développement du endpoint :

- en utilisant les annotations : les méthodes annotées @OnOpen, @OnMessage, @OnClose et @OnError peuvent avoir un paramètre de type Session
- en utilisant l'API : les méthodes onOpen(), onClose() et OnError() possèdent un paramètre de type Session

Dans ces cas, l'implémentation se charge de passer l'instance de la Session en paramètre lors de l'invocation de ces méthodes.

La classe Session propose plusieurs méthodes permettant d'obtenir des informations sur la connexion :

Méthode	Rôle
void addMessageHandler(MessageHandler handler)	Enregistrer un objet qui va gérer les messages entrants
void close()	Fermer la conversation avec un code de status normal et sans description de la raison
void close(CloseReason closeReason)	Fermer la conversation avec un code de status normal en précisant la description de la raison
RemoteEndpoint.Async getAsyncRemote()	Obtenir une référence sur un objet qui encapsule l'autre partie de la conversation et permet de lui envoyer des messages de manière asynchrone
RemoteEndpoint.Basic getBasicRemote()	Obtenir une référence sur un objet qui encapsule l'autre partie de la conversation et permet de lui envoyer des messages
WebSocketContainer getContainer()	Obtenir le conteneur qui gère la session
String getId()	Obtenir l'identifiant unique de la session
int getMaxBinaryMessageBufferSize()	Obtenir la taille maximale du buffer d'un message binaire gérable par la session
long getMaxIdleTimeout()	Obtenir le timeout en millisecondes avant que la conversation puisse être fermée par le conteneur si la session est inactive
int getMaxTextMessageBufferSize()	Obtenir la taille maximale du buffer d'un message texte gérable par la session
Set<MessageHandler> getMessageHandlers()	Obtenir une copie immuable de l'ensemble des objets qui gèrent les messages entrants
List<Extension> getNegotiatedExtensions()	Obtenir une liste des extensions utilisées pour la conversation
String getNegotiatedSubprotocol()	Obtenir le sous protocole demandé lors du handshake de la connexion
Set<Session> getOpenSessions()	Obtenir une copie de l'ensemble des sessions ouvertes sur le même endpoint que la session
Map<String,String> getPathParameters()	Obtenir une collection des paramètres (nom/valeur) utilisés dans la requête pour ouvrir la session
String getProtocolVersion()	Obtenir la version du protocole WebSocket utilisée par la conversation
String getQueryString()	Obtenir la requête utilisée pour ouvrir la session
Map<String,List<String>> getRequestParameterMap()	Obtenir une collection des paramètres utilisés dans la requête pour ouvrir la session
URI getRequestURI()	Obtenir l'URI et ses paramètres utilisés pour ouvrir la session
Principal getUserPrincipal()	Obtenir l'utilisateur identifié pour cette session s'il est défini sinon renvoie null
Map<String,Object> getUserProperties()	Obtenir une collection des propriétés spécifiques à la conversation

boolean isOpen()	Renvoyer un booléen qui précise si la socket sous-jacente est ouverte ou non
boolean isSecure()	Renvoyer un booléen qui précise si la socket sous-jacente utilise un protocole sécurisé
void removeMessageHandler(MessageHandler handler)	Retirer l'objet qui gère les messages entrants de ceux associés à la conversation
void setMaxBinaryMessageBufferSize(int length)	Définir la taille maximale du buffer d'un message binaire gérable par la session
void setMaxIdleTimeout(long milliseconds)	Définir le timeout en millisecondes avant que la conversation puisse être fermée par le conteneur si la session est inactive. La valeur fournie en paramètre doit être supérieure à zéro.
void setMaxTextMessageBufferSize(int length)	Définir la taille maximale du buffer d'un message binaire gérable par la session

La collection de type Map<String, Object> retournée par la méthode getUserProperties() permet de stocker des informations spécifiques à la session et à l'application qui pourront ainsi être partagées par les différents échanges de la conversation.

La classe CloseReason encapsule la raison de la fermeture de la websocket. Elle ne possède qu'un seul constructeur qui attend en paramètre une valeur de type CloseReason.CloseCodes et une chaîne de caractères qui décrit la raison de la fermeture.

L'énumération CloseReason.CloseCodes contient les codes de fermeture définit par la spécification.

Exemple (code Java 7) :

```
session.close(new CloseReason(CloseCodes.NORMAL_CLOSURE, "Fin de la conversation"));
```

Il est important de s'assurer de la fermeture d'une session lorsque celle-ci n'est plus utilisée pour permettre de libérer les ressources consommées par la WebSocket.

78.1.2. Les interfaces RemoteEndpoint

L'interface javax.websocket.RemoteEndpoint définit les fonctionnalités utilisables sur l'endpoint distant de la conversation notamment l'envoi de messages.

Il existe deux types de RemoteEndpoint :

- RemoteEndpoint.Basic pour l'envoi synchrone d'un message
- RemoteEndpoint.Async pour l'envoi asynchrone d'un message

Pour obtenir une instance de type RemoteEndpoint, il faut invoquer la méthode getBasicRemote() ou la méthode getAsyncRemote() de la classe Session.

Il n'y a pas de garantie sur la livraison du message au endpoint.

L'interface RemoteEndpoint définit plusieurs méthodes :

Méthode	Rôle
void flushBatch()	Indiquer à l'implémentation que tous les messages peuvent être envoyés au endpoint
boolean getBatchingAllowed()	Renvoyer un booléen qui précise si l'implémentation peut utiliser le mode batching

void sendPing(ByteBuffer data)	Envoyer un message de type ping contenant les données fournies en paramètres
void sendPong(ByteBuffer data)	Envoyer un message de type pong contenant les données fournies en paramètres
void setBatchingAllowed(boolean allowed)	Préciser à l'implémentation si elle peut utiliser le mode batching pour envoyer un message

Le mode batching permet à l'implémentation de traiter par lots les messages à envoyer. Toutes les implémentations ne proposent pas un support du mode batch qui est désactivé par défaut.

Lorsque le mode batching est utilisé, il est nécessaire d'invoquer explicitement la méthode flushBatch() pour s'assurer que tous les messages ont été envoyés.

L'interface RemoteEndpoint.async définit les fonctionnalités pour l'envoi de messages de manière asynchrone. Elle définit plusieurs méthodes :

Méthode	Rôle
long getSendTimeout()	Retourner le nombre de millisecondes durant laquelle l'implémentation peut attendre pour envoyer le message
Future<Void> sendBinary(ByteBuffer data)	Envoyer des données binaires de manière asynchrone
Future<Void> sendBinary(ByteBuffer data, SendHandler handler)	Envoyer des données binaires de manière asynchrone
Future<Void> sendObject(Object data)	Envoyer un objet de manière asynchrone
Future<Void> sendObject(Object data, SendHandler handler)	Envoyer un objet de manière asynchrone
Future<Void> sendText(String data)	Envoyer des données binaires de manière asynchrone
Future<Void> sendText(String data, SendHandler handler)	Envoyer des données binaires de manière asynchrone
void setSendTimeout(long timeout)	Définir le nombre de millisecondes que peut attendre l'implémentation pour envoyer un message

Exemple (code Java 7) :

```
public void envoyerMessagePartiel(Session session, String message, Boolean isLast) throws
  IOException {
  session.getBasicRemote().sendText(message, isLast);
}
```

L'interface RemoteEndpoint.Basic définit les fonctionnalités pour l'envoi de messages de manière synchrone. Elle définit plusieurs méthodes :

Méthode	Rôle
OutputStream getSendStream()	Obtenir un flux pour envoyer des données binaires
Writer getSendWriter()	Obtenir un flux pour envoyer des données textuelles
void sendBinary(ByteBuffer data)	Envoyer des données binaires
void sendBinary(ByteBuffer partialData, boolean isLast)	Envoyer un morceau des données binaires. Le booléen indique si c'est le dernier morceau
void sendBinary(Object data)	Envoyer un objet
void sendText(String data)	Envoyer des données textuelles

void sendText(String partialData, boolean isLast)	Envoyer un morceau des données textuelles. Le booléen indique si c'est le dernier morceau

L'envoi de message est bloquant jusqu'à ce que tout le message ait été envoyé à la connexion sous-jacente.

Une exception de type IllegalStateException peut être levée si deux messages sont envoyés en même temps sur la même connexion.

78.1.3. Les interfaces MessageHandler

L'interface MessageHandler définit les fonctionnalités relatives à la réception d'un message dans une conversation.

L'interface MessageHandler est utilisée :

- directement par le développeur pour traiter les messages reçus par un endpoint développé avec l'API
- indirectement par l'implémentation pour les méthodes des endpoints annotés avec @OnMessage

Les spécifications du protocole WebSocket précise qu'un message peut être envoyé dans son intégralité ou de manière partielle. L'interface MessageHandler possède deux interfaces imbriquées pour supporter ces fonctionnalités :

- MessageHandler.Partial : utilisée par l'implémentation pour traiter un message partiel
- MessageHandler.Whole : utilisée par l'implémentation pour traiter un message complet

L'interface MessageHandler.Partial<T> est un handler qui sera invoqué par l'implémentation lors de la réception d'une partie d'un message.

Le type T peut être :

- String dans le cas d'un message de type texte
- ByteBuffer ou Byte[] dans le cas d'un message de type binaire

Il ne faut pas utiliser l'instance de type ByteBuffer après l'invocation de la méthode onMessage() car l'implémentation peut recycler cette instance.

Elle ne définit qu'une seule méthode :

Méthode	Rôle
void onMessage(T partialMessage, boolean last)	Traiter un message partiel qui a été reçu par l'implémentation

L'interface MessageHandler.Whole<T> est un handler qui sera invoqué par l'implémentation lors de la réception d'un message.

Le type T peut être :

- String, Reader ou un objet pour lequel un Decoder.Text ou Decoder.TextStream est enregistré dans le cas d'un message de type texte
- ByteBuffer, Byte[], InputStream ou un objet pour lequel un Decoder.Binary ou Decoder.BinaryStream est enregistré dans le cas d'un message de type binaire
- PongMessage pour les messages de type pong

Il ne faut pas utiliser l'instance de type ByteBuffer, Reader et InputStream après l'invocation de la méthode onMessage() car l'implémentation peut recycler cette instance.

Elle ne définit qu'une seule méthode

Méthode	Rôle

void onMessage(T message)	Traiter un message qui a été reçu par l'implémentation

Il est nécessaire d'enregistrer un MessageHandler à la Session.

Exemple (code Java 7) :

```
public class MonEndpoint extends Endpoint {

    @Override
    public void onOpen(Session session, EndpointConfig EndpointConfig) {
        session.addMessageHandler(new MessageHandler.Whole<String>() {
            @Override
            public void onMessage(String message) {
                System.out.println("Message recu: "+message);
            }
        });
    }
}
```

Une instance de type MessageHandler ne sera invoquée que par un seul thread d'une session à la fois. Si une même instance de type MessageHandler est associée à plusieurs sessions, alors la gestion des accès concurrents doit être prise en charge dans l'implémentation du MessageHandler.

Il n'est possible de n'enregistrer qu'un seul MessageHandler complet ou partiel pour un même type de données sur une même session. Par exemple, il n'est possible d'enregistrer un MessageHandler.Whole de type texte et un MessageHandler.Partial de type texte sur la même session.

Exemple (code Java 7) :

```
    @Override
    public void onOpen(Session session, EndpointConfig config) {
        final RemoteEndpoint.Basic remote = session.getBasicRemote();
        session.addMessageHandler(new MessageHandler.Whole<String>() {
            @Override
            public void onMessage(String text) {
                try {
                    remote.sendText(text);
                } catch (IOException ioe) {
                   LOGGER.log(Level.SEVERE, "Erreur durant l'envoi",ioe);
                }
            }
        });
        session.addMessageHandler(new MessageHandler.Partial<String>() {
            @Override
            public void onMessage(String text, boolean last) {
                try {
                    remote.sendText(text);
                } catch (IOException ioe) {
                    LOGGER.log(Level.SEVERE, "Erreur durant l'envoi",ioe);
                }
            }
        });
    }
```

Résultat :

```
janv. 18, 2014 6:00:37 PM
fr.jmdoudoux.dej.websockets.server.MonEchoEndpoint onError
Grave: onError java.lang.IllegalStateException: Text
MessageHandler already registered.
```

Il n'est donc possible d'enregistrer qu'un seul MessageHandler pour des données de texte, un seul pour des données binaires et un seul pour des messages de type pong.

L'implémentation peut proposer de gérer elle-même les messages partiels en les stockant jusqu'à la réception du dernier morceau et invoquer le MessageHandler pour message complet avec l'intégralité du message. Inversement, si seulement un MessageHandler pour message partiel est enregistré et qu'un message complet est envoyé par l'endpoint distant alors l'implémentation peut invoquer le handler en lui passant le message.

78.2. Le développement d'un endpoint

La JSR 356 définit une API qui permet d'utiliser les WebSockets dans une application aussi bien côté serveur que côté client.

Les WebSockets fonctionnent en mode client/serveur. Le client est toujours responsable de l'initialisation de la communication en demandant l'ouverture de la connexion. Côté serveur, l'endpoint est publié pour attendre une demande de connexion d'un client. Une fois la connexion établie, le client et le serveur peuvent agir de manière symétrique : les API pour la partie cliente et serveur sont similaires.

Lors de l'utilisation de WebSockets, plusieurs événements peuvent survenir :

- un client initialise une connexion HTTP de type handshake vers le serveur
- le serveur répond pour établir la connexion en changeant le protocole de HTTP à WebSocket
- le client et le serveur peuvent alors émettre et recevoir des messages de manière symétrique
- la connexion peut être fermée par le client ou le serveur

Dans un endpoint, plusieurs méthodes sont définies comme des callbacks qui seront invoquées selon les événements de la communication : onOpen, onMessage, onError et onClose.

Pour répondre à ses événements, la JSR 356 propose deux modèles de programmation pour définir des méthodes qui seront les callbacks sur les événements liés aux conversations avec la WebSocket :

- utilisation d'annotations sur des POJO et les méthodes de ses POJO
- utilisation de l'API en implémentant l'interface Endpoint

78.2.1. Le développement d'un endpoint avec les annotations

La manière la plus simple de définir un endpoint est d'utiliser les annotations.

La JSR 356 définit plusieurs annotations :

Annotation	Application	Utilité
@javax.websocket.server.ServerEndpoint	Classe	Définir un POJO comme étant un endpoint côté serveur
@javax.websocket.OnOpen	Méthode	Déclarer une méthode comme étant un callback lors d'un événement de type Open
@javax.websocket.OnClose	Méthode	Déclarer une méthode comme étant un callback lors d'un événement de type Close
@javax.websocket.OnMessage	Méthode	Déclarer une méthode comme étant un callback lors d'un événement de type Message
@javax.websocket.server.PathParam	Paramètre d'une méthode	Permet de mapper un paramètre d'une méthode avec un marqueur défini dans le template de l'uri associé du endpoint. A la réception d'une requête HTTP qui satisfasse le template, le conteneur va extraire la valeur est la passé comme valeur du paramètre de la méthode annotée
@javax.websocket.OnError	Méthode	Déclarer une méthode comme étant un callback lors d'une erreur

78.2.1.1. L'annotation @ServerEndpoint

Le développement d'un endpoint serveur peut se faire en utilisant un simple POJO annoté avec l'annotation @javax.websocket.server.ServerEndpoint : ceci permet de préciser au conteneur que la classe est un endpoint pour WebSocket côté serveur.

L'annotation @ServerEndpoint possède plusieurs attributs :

Nom	Rôle
value	URI relative ou template pour l'URI relative à l'url du contexte de la webapp. Obligatoire
decoders	Enregistrer des décodeurs
encoders	Enregistrer des encodeurs
subprotocols	Définir la liste des noms des sous-protocoles qui sont supportés Exemple : soap, wamp (websocket application message processing), ... Le premier nom de la liste qui correspond à celui fourni par l'endpoint client sera utilisé
configurator	Fournir le type d'une classe de type ServerEndpointConfig.Configurator qui permettra de configurer les connexions au endpoint.

Exemple (code Java 7) :

```
@ServerEndpoint(
    value = "/monendpoint",
    decoders = MonDecoder.class,
    encoders = MonEncoder.class,
    subprotocols = {"sousprotocole1", "sousprotocole2"},
    configurator = MonConfigurator.class)
public class MonServeurEndpoint {
}
```

L'annotation @ServerEndpoint attend une valeur d'attribut obligatoire qui précise l'URI associée au endpoint. La valeur de l'URI doit obligatoirement commencer par un caractère slash et peut se terminer ou pas par un caractère slash.

Exemple (code Java 7) :

```
@ServerEndpoint("/echo")
public class EchoEndpoint { }
```

Le chemin précisé comme URI peut contenir des paramètres dont les valeurs correspondantes seront extraites à l'exécution de l'URL utilisée. L'obtention de la valeur se fait en utilisant l'annotation @PathParam.

Exemple (code Java 7) :

```
@ServerEndpoint("/personnes/{pers-id}")
public class PersonneEndpoint {
  @OnMessage
  public void traiter(@PathParam("pers-id")String id) {
  }
}
```

L'URL complète pour utiliser la WebSocket sera composée de plusieurs éléments :

- le hostname et le port du conteneur
- l'uri de la webapp
- l'uri de la WebSocket

Exemple :

```
ws://localhost:8080/MaWebApp/echo
```

78.2.1.2. L'annotation @OnMessage

L'annotation @javax.websocket.OnMessage permet de définir une méthode qui sera invoquée chaque fois qu'un message est reçu pour le endpoint.

Le message peut être de plusieurs types :

- String
- byte[]
- ByteBuffer
- toute classe pour lequel il existe un décodeur

Lorsque l'endpoint reçoit un message, la méthode annotée avec l'annotation @OnMessage est invoquée. Cette méthode peut avoir en paramètre :

- le contenu du message
- un objet de type javax.websocket.Session qui encapsule la session courante
- zéro ou plusieurs paramètres de type String annotés avec @PathParam qui contiendront les valeurs des paramètres utilisés dans la requête

Une seule méthode d'une classe annotée avec @ServerEndpoint ou @ClientEndpoint peut être annotée avec @OnMessage

Exemple (code Java 7) :

```
@OnMessage
public void traiterOnMessage(String message) {
   System.out.println("Message recu par WebSocket : "+message);
}
```

La méthode annotée peut avoir comme type de retour :

- void
- String, byte[], ByteBuffer, toute classe pour lequel il existe un encodeur

Si la méthode annotée avec @OnMessage retourne une valeur alors l'implémentation enverra un message contenant cette valeur au endpoint client.

Exemple (code Java 7) :

```
  @OnMessage
  public String traiterOnMessage(String message) {
    return message.toUpperCase();
  }
```

Il est aussi possible d'envoyer un message en utilisant un objet de type RemoteEndPoint.Basic obtenu en invoquant la méthode getBasicRemote() de la session courante.

Exemple (code Java 7) :

```
RemoteEndpoint.Basic remoteEndpoint = session.getBasicRemote();
remoteEndpoint.sendText ("contenu du message");
```

Pour obtenir une instance de la session courante, il faut ajouter un paramètre de type javax.websocket.Session à la méthode. L'implémentation fournira alors en paramètre l'instance de la Session.

78.2.1.3. L'annotation @OnOpen

L'annotation @javax.websocket.OnOpen permet de définir une méthode qui sera invoquée lorsque la connexion de la WebSocket est ouverte. Chaque connexion est associée à une session. La méthode annotée ne sera invoquée qu'une seule fois pour une même connexion d'une WebSocket.

Lorsque la connexion de la WebSocket est établie, une instance de type Session est créée et la méthode annotée avec @OnOpen est invoquée. Cette méthode peut avoir optionnellement en paramètre :

- un objet de type javax.websocket.EndpointConfig qui encapsule les informations utiles lors du handshake
- un objet de type javax.websocket.Session qui encapsule la session courante
- zéro ou plusieurs paramètres de type String annoté avec @PathParam qui contiendront les valeurs des paramètres utilisées dans la requête

Une seule méthode d'une classe annotée avec @ServerEndpoint ou @ClientEndpoint peut être annotée avec @OnOpen.

Exemple (code Java 7) :

```
private Map<String, Object> userProperties;

@OnOpen
public void traiterOnOpen (Session session, EndpointConfig config) {
  System.out.println ("WebSocket ouverte : "+session.getId());
  properties = config.getUserProperties();
}
```

78.2.1.4. L'annotation @OnClose

L'annotation @javax.websocket.OnClose permet de définir une méthode qui sera invoquée lorsque la connexion de la WebSocket est fermée.

Lorsque la connexion est fermée, la méthode annotée avec @OnClose est invoquée. Cette méthode peut avoir optionnellement en paramètre :

- un objet de type javax.websocket.Session qui encapsule la session courante. Celle-ci ne pourra plus être utilisée à la fin de l'invocation de la méthode
- un objet de type javax.websocket.CloseReason qui précise la raison de la fermeture de la WebSocket
- zéro ou plusieurs paramètres de type String annotés avec @PathParam qui contiendront les valeurs des paramètres utilisés dans la requête

Une seule méthode d'une classe annotée avec @ServerEndpoint ou @ClientEndpoint peut être annotée avec @OnClose.

Exemple (code Java 7) :

```
@OnClose
public void traiterOnClose (CloseReason reason) {
  System.out.println("Fermeture de la WebSocket a cause de : "+reason.getReasonPhrase());
}
```

78.2.1.5. L'annotation @OnError

Lorsqu'une erreur survient durant la conversation la méthode annotée avec @javax.websocket.OnError est invoquée. Cette méthode doit avoir un objet de type Throwable qui encapsule l'exception en paramètre et peut avoir optionnellement en paramètre :

- un objet de type javax.websocket.Session qui encapsule la session courante
- zéro ou plusieurs paramètres de type String annotés avec @PathParam qui contiendront les valeurs des paramètres utilisés dans la requête

Une seule méthode d'une classe annotée avec @ServerEndpoint ou @ClientEndpoint peut être annotée avec @OnError.

Exemple (code Java 7) :

```
  @OnError
  public void onError(Session session, Throwable t) {
    t.printStackTrace();
  }
```

78.2.1.6. L'annotation @ClientEndpoint

Pour créer un endpoint côté client, il faut créer une classe qui soit annotée avec l'annotation @javax.websocket.ClientEndpoint.

L'annotation @ClientEndpoint possède plusieurs attributs :

Nom	Rôle
decoders	Enregistrer des décodeurs qui permettent de transformer un message texte ou binaire en un objet. La valeur est un tableau de noms de classes qui implémentent l'interface Decoder
encoders	Enregistrer des encodeurs qui permettent de transformer un objet en un message texte ou binaire. La valeur est un tableau de noms de classes qui implémentent l'interface Encoder
subprotocols	Tableau des noms des sous-protocoles supportés par le client Exemple : soap, wamp (websocket application message processing), ...
Configurator	Préciser la classe de type ClientEndpointConfig.Configurator qui sera utilisée

Exemple (code Java 7) :

```
@ClientEndpoint (
    decoders = MonDecoder.class,
    encoders = MonEncoder.class,
    subprotocols = {"sousprotocole1", "sousprotocole2"},
    configurator = MonConfigurator.class)
public class MonClientEndpoint {}
```

Un endpoint client ne peut pas recevoir de requêtes pour créer une connexion.

Il est possible de définir une classe fille de la classe ClientEndpointConfiguration.Configurator qui permet de modifier certaines parties de la requête réponse lors du traitement d'un handshake.

La méthode beforeRequest() permet de modifier les éléments du header avant que la requête ne soit envoyée.

La méthode afterResponse() permet de modifier les traitements de la réponse du handshake.

Exemple (code Java 7) :

```
public class MonConfigurator {
  public void beforeRequest(Map<String, List<String>> headers) {
  }

  public void afterResponse(HandshakeResponse hr) {
    // traitements de la réponse du handshake
  }
}
```

78.2.2. Le développement d'un endpoint sans annotations

Il est possible de développer un endpoint en utilisant l'API WebSocket.

Il faut alors écrire une classe qui héritent de la classe javax.websocket.Endpoint et redéfinir selon les besoins les méthodes onOpen(), onClose() et onError().

La gestion des messages reçus se fait en enregistrant un handler de messages, qui est une instance de type MessageHandler, à la session dans les traitements de la méthode onOpen() : cette enregistrement se fait en invoquant la méthode addMessageHandler() de la session.

L'émission d'un message se fait en utilisant une instance de type RemoteEndpoint.

78.2.2.1. La développement d'un endpoint serveur

La classe abstraite javax.websocket.EndPoint encapsule un endpoint d'une WebSocket. Pour développer un endpoint en utilisant l'API, il faut créer une classe fille qui hérite de la classe EndPoint et redéfinir les méthodes utiles pour traiter les événements des conversations (open, error et close).

La classe Endpoint définit possède plusieurs méthodes :

Méthode	Rôle
void onClose(Session session, CloseReason closeReason)	Callback lors de la fermeture de la WebSocket
void onError(Session session, Throwable t)	Callback lorsqu'une erreur survient
abstract void onOpen(Session session, EndpointConfig config)	Callback lorsqu'une nouvelle conversation débute

Pour permettre au endpoint de traiter les messages reçus, il faut ajouter une implémentation de type MessageHandler à la session courante. Cette opération doit être faite dans la redéfinition de la méthode onOpen() du endpoint.

Exemple (code Java 7) :

```
package fr.jmdoudoux.dej.websockets.server;

import java.io.IOException;
import java.util.Date;
import java.util.logging.Logger;
import javax.websocket.CloseReason;
import javax.websocket.EndpointConfig;
import javax.websocket.MessageHandler;
import javax.websocket.RemoteEndpoint;
import javax.websocket.Session;

public class MonEchoEndpoint extends javax.websocket.Endpoint  {

    private static final Logger LOGGER = Logger.getLogger(MonEchoEndpoint.class.getName());

    public MonEchoEndpoint() {
        super();
        LOGGER.info("invocation constructeur MonEchoEndPoint");
    }

    @Override
    public void onOpen(Session session, EndpointConfig config) {
        final RemoteEndpoint.Basic remote = session.getBasicRemote();
        session.addMessageHandler(new MessageHandler.Whole<String>() {
            @Override
            public void onMessage(String text) {
                try {
                    remote.sendText(ThreadSafeFormatter.getDateFormatter().format(new Date())
+ " (MonEchoEndPoint) " + text);
                } catch (IOException ioe) {
                    LOGGER.severe("Could not send the message", ioe);
                }
            }
        });
    }

    @Override
    public void onClose(Session session, CloseReason closeReason) {
        LOGGER.info("onClose : "+closeReason);
    }

    @Override
    public void onError(Session session, Throwable throwable) {
        LOGGER.severe("onError", throwable);
    }
}
```

Par défaut, le Configurator associé au ServerEndPointConfig assure qu'une instance de type EndPoint ne sera invoquée que par un seul thread pour une même connexion.

78.2.2.2. Le développement d'un endpoint client

Pour qu'un client se connecte à un serveur, il faut obtenir une instance de type javax.websocket.WebSocketContainer en invoquant la méthode getWebSocketContainer() de la classe javax.websocket.ContainerProvider.

Il faut invoquer la méthode connectToServer() de l'instance de type WebSocketContainer qui attend en paramètre :

- la classe du endpoint client
- l'URI de la websocket

Exemple (code Java 7) :

```
package fr.jmdoudoux.dej.websocket.client;

import java.io.IOException;
import java.net.URI;
import java.net.URISyntaxException;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.websocket.ClientEndpoint;
import javax.websocket.DeploymentException;
import javax.websocket.OnMessage;
import javax.websocket.Session;

@ClientEndpoint
public class TestClientWebSocket {
    private static final Logger LOGGER = Logger.getLogger(TestClientWebSocket.class.getName());

    @OnMessage
    public void onMessage(String message, Session session) {
       LOGGER.log(Level.INFO, message);
    }

    public static void main(String[] args) {
        LOGGER.log(Level.INFO, "Lancement client");
        javax.websocket.WebSocketContainer container =
            javax.websocket.ContainerProvider.getWebSocketContainer();
        try {
            container.connectToServer(TestClientWebSocket.class,
                new URI("ws://localhost:8080/MaWebApp/valeurs"));
        } catch (DeploymentException | IOException | URISyntaxException ex) {
            LOGGER.log(Level.SEVERE, null, ex);
        }

       while(true) {}
    }
}
```

Résultat :

```
nov. 15, 2013 9:43:43 PM
fr.jmdoudoux.dej.websocket.client.TestClientWebSocket main
Infos: Lancement client
nov. 15, 2013 9:43:45 PM
fr.jmdoudoux.dej.websocket.client.TestClientWebSocket onMessage
Infos: 103,90

```

```
nov. 15, 2013 9:43:46 PM fr.jmdoudoux.dej.websocket.client.TestClientWebSocket
onMessage
Infos: 104,21
nov. 15, 2013 9:43:47 PM
fr.jmdoudoux.dej.websocket.client.TestClientWebSocket onMessage
Infos: 104,56
nov. 15, 2013 9:43:48 PM
fr.jmdoudoux.dej.websocket.client.TestClientWebSocket onMessage
Infos: 104,62
```

Pour exécuter le code ci-dessus, il faut ajouter plusieurs bibliothèques au classpath notamment l'API Java EE 7 et les bibliothèques requises par l'implémentation WebSocket utilisée.

L'interface javax.websocket.WebSocketContainer définit les fonctionnalités d'une classe qui permet un accès au conteneur qui exécute la websocket.

Elle permet notamment de configurer certains paramètres des endpoints et de se connecter à un serveur de websockets.

Méthode	Rôle
Session connectToServer(Class<?> annotatedEndpointClass, URI path)	Connecter à la websocket définie par l'URI fournie en paramètre la classe annotée qui encapsule l'endpoint
Session connectToServer(Class<? extends Endpoint> endpointClass, ClientEndpointConfig cec, URI path)	Connecter à la websocket définie par l'URI fournie en paramètre la classe qui encapsule l'endpoint en utilisant la configuration fournie
Session connectToServer(Endpoint endpointInstance, ClientEndpointConfig cec, URI path)	Connecter à la websocket définie par l'URI fournie en paramètre la classe qui encapsule l'endpoint en utilisant la configuration fournie
Session connectToServer(Object annotatedEndpointInstance, URI path)	Connecter à la websocket définie par l'URI fournie en paramètre la classe annotée qui encapsule l'endpoint
long getDefaultAsyncSendTimeout()	Définir le nombre de millisecondes que l'implémentation peut attendre lors de l'envoi d'un message pour tous les RemoteEndpoints associés au container
int getDefaultMaxBinaryMessageBufferSize()	Obtenir la taille maximale du buffer que le conteneur peut utiliser pour stocker un message binaire
long getDefaultMaxSessionIdleTimeout()	Obtenir le nombre de millisecondes après lequel une session inactive sera fermée
Int getDefaultMaxTextMessageBufferSize()	Définir la taille maximale du buffer que le conteneur peut utiliser pour stocker un message texte
Set<Extension> getInstalledExtensions()	Renvoyer une collection contenant les extensions installée dans le conteneur
setAsyncSendTimeout(long timeoutmillis)	Définir le nombre de millisecondes que l'implémentation peut attendre lors de l'envoi d'un message pour tous les RemoteEndpoints associés au container
void setDefaultMaxBinaryMessageBufferSize(int max)	Définir la taille maximale du buffer que le conteneur peut utiliser pour stocker un message binaire
void setDefaultMaxSessionIdleTimeout(long timeout)	Définir le nombre de millisecondes après lequel une session inactive sera fermée
void setDefaultMaxTextMessageBufferSize(int max)	Définir la taille maximale du buffer que le conteneur peut utiliser pour stocker un message texte

La classe ContainerProvider permet d'obtenir une instance de type WebSocketContainer en utilisant le ServiceLoader : le type de l'implémentation est précisé dans le fichier META-INF/services/javax.websocket.ContainerProvider contenu dans le jar de l'implémentation de la JSR 356.

Méthode	Rôle
protected abstract WebSocketContainer getContainer()	Charger l'implémentation du conteneur
static WebSocketContainer getWebSocketContainer()	Obtenir une instance de type WebsocketContainer

Exemple :

```
package fr.jmdoudoux.dej.websocket.client;

import java.io.IOException;
import java.net.URI;
import java.net.URISyntaxException;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.websocket.DeploymentException;

public class TestClientWebSocketMain {

    private static final Logger LOGGER =
        Logger.getLogger(TestClientWebSocketMain.class.getName());

    public static void main(String[] args) {
        LOGGER.log(Level.INFO, "Lancement client");
        try {
            long compteur = 0L;
            final ValeursClientEndpoint clientEndPoint = new ValeursClientEndpoint(
                   new URI("ws://localhost:8080/MaWebApp/monbean"));
            while (true) {
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException ex) {
                }
                compteur++;
                clientEndPoint.sendMessage(""+compteur);
            }
        } catch (DeploymentException | IOException | URISyntaxException ex) {
            LOGGER.log(Level.SEVERE, null, ex);
        }
    }
}
```

Exemple :

```
package fr.jmdoudoux.dej.websocket.client;

import java.io.IOException;
import java.io.StringReader;
import java.net.URI;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.json.Json;
import javax.json.JsonObject;
import javax.websocket.ClientEndpoint;
import javax.websocket.CloseReason;
import javax.websocket.ContainerProvider;
import javax.websocket.DeploymentException;
import javax.websocket.OnClose;
import javax.websocket.OnError;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.WebSocketContainer;

@ClientEndpoint
public class ValeursClientEndpoint {

    private static final Logger LOGGER =
        Logger.getLogger(ValeursClientEndpoint.class.getName());

    Session session = null;

    public ValeursClientEndpoint(URI endpointURI) throws DeploymentException, IOException {
         WebSocketContainer container = ContainerProvider
               .getWebSocketContainer();
        container.connectToServer(this, endpointURI);
    }

    @OnOpen
    public void onOpen(Session session) {
        LOGGER.log(Level.INFO, "Client endpoint open");
        this.session = session;
    }

    @OnClose
    public void onClose(Session session, CloseReason reason) {
        LOGGER.log(Level.INFO, "Client endpoint close");
        this.session = null;
    }

    @OnError
    public void onError(Throwable t) {
        LOGGER.log(Level.SEVERE, "Client endpoint error ", t);
    }

    @OnMessage
    public void onMessage(String message, Session session) {
        LOGGER.log(Level.INFO, "message recu="+message);
        JsonObject jsonObject = Json.createReader(new StringReader(message)).readObject();
        String nom = jsonObject.getString("nom");
        String valeur = jsonObject.getString("valeur");
        LOGGER.log(Level.INFO, "nom="+nom+" valeur="+valeur);
    }

    public void sendMessage(String message) {
        if (session != null) {
            this.session.getAsyncRemote().sendText(message);
        }
    }
}
```

Résultat :

```
janv. 30, 2014 9:49:27 PM
fr.jmdoudoux.dej.websocket.client.TestClientWebSocketMain main
Infos: Lancement client
janv. 30, 2014 9:49:28 PM
fr.jmdoudoux.dej.websocket.client.ValeursClientEndpoint onOpen
Infos: Client endpoint open
janv. 30, 2014 9:49:33 PM
fr.jmdoudoux.dej.websocket.client.ValeursClientEndpoint onMessage
Infos: message
recu={"nom":"nom1","valeur":"valeur1"}
janv. 30, 2014 9:49:33 PM
fr.jmdoudoux.dej.websocket.client.ValeursClientEndpoint onMessage
Infos: nom=nom1 valeur=valeur1
janv. 30, 2014 9:49:38 PM
fr.jmdoudoux.dej.websocket.client.ValeursClientEndpoint onMessage
Infos: message
recu={"nom":"nom2","valeur":"valeur2"}
janv. 30, 2014 9:49:38 PM
fr.jmdoudoux.dej.websocket.client.ValeursClientEndpoint onMessage
Infos: nom=nom2 valeur=valeur2
janv. 30, 2014 9:49:43 PM
fr.jmdoudoux.dej.websocket.client.ValeursClientEndpoint onMessage
Infos: message
recu={"nom":"nom3","valeur":"valeur3"}
janv. 30, 2014 9:49:43 PM
fr.jmdoudoux.dej.websocket.client.ValeursClientEndpoint onMessage
Infos: nom=nom3 valeur=valeur3
```

78.2.3. La configuration des endpoints sans annotations

Lorsque des endpoints sont développés en utilisant l'API, il est nécessaire d'écrire du code pour permettre de définir leur configuration qui sera utilisée lors de leur enregistrement.

Cette configuration peut impliquer l'utilisation de différentes classes et interfaces selon les besoins notamment des objets de type EndpointConfig.

78.2.3.1. L'interface EndpointConfig

L'interface javax.websocket.EndpointConfig définit les fonctionnalités de base pour la configuration d'un endpoint client ou serveur.

Cette interface définit plusieurs méthodes :

Méthode	Rôle
List<Class<? extends Decoder>> getDecoders()	Obtenir la liste des décodeurs associés au endpoint. L'implémentation va créer une instance de chacun des décodeurs lors de l'enregistrement du endpoint
List<Class<? extends Encoder>> getEncoders()	Obtenir la liste des encodeurs associés au endpoint. L'implémentation va créer une instance de chacun des encodeurs lors de l'enregistrement du endpoint
Map<String,Object> getUserProperties()	Obtenir une collection de type Map qui permet de gérer des données relatives au endpoint. Il est préférable que les clés et valeurs contenues dans cette collection soient sérialisables

Elle possède deux interfaces filles :

- ClientEndpointConfig pour la configuration d'un endpoint client
- ServerEndpointConfig pour la configuration d'un endpoint serveur

78.2.3.2. La configuration des endpoints serveur

Lors de l'écriture de endpoints serveur avec l'API, il est nécessaire d'écrire une classe qui implémente l'interface ServerApplicationConfig. Elle va permettre de fournir les informations de configuration pour chaque endpoint à enregistrer dans le serveur.

78.2.3.2.1. L'interface ServerEndpointConfig

L'interface javax.websocket.server.ServerEndpointConfig définit les méthodes d'un objet qui encapsule la configuration d'un endpoint pour son déploiement dans un serveur.

Elle définit plusieurs méthodes :

Méthode	Rôle
ServerEndpointConfig.Configurator getConfigurator()	Renvoyer l'instance de type ServerEndpointConfig.Configurator utilisée par cette configuration
Class<?> getEndpointClass()	Renvoyer le type de classe du endpoint
List<Extension> getExtensions()	Renvoyer la liste des extensions
String getPath()	Renvoyer l'uri associée au endpoint
List<String> getSubprotocols()	Renvoyer la liste des sous-protocoles

Pour faciliter la création d'une instance de type ServerEndpointConfig, il faut utiliser la classe ServerEndpointConfig.Builder.

Pour permettre de personnaliser certaines opérations notamment le handshake, il est possible d'utiliser une instance de type ServerEndpointConfig.Configurator.

78.2.3.2.2. La classe ServerEndpointConfig.Builder

La classe javax.websocket.server.ServerEndpointConfig.Builder permet de créer une instance de type ServerEndpointConfig en utilisant le motif de conception builder.

Elle propose donc plusieurs méthodes qui permettent de fournir les différents éléments qui seront encapsulés dans l'instance, une méthode pour renvoyer l'instance du Builder et une pour obtenir l'instance construite :

Méthode	Rôle
ServerEndpointConfig build()	Créer et renvoyer l'instance de type ServerEndpointConfigBuilds encapsulant les attributs fournis au builder
ServerEndpointConfig.builder configurator(ServerEndpointConfig.Configurator serverEndpointConfigurator)	Fournir à la configuration le Configurator qui sera utilisé
static ServerEndpointConfig.builder create(Class<?> endpointClass, String path)	Créer et retourner une instance de type Builder en lui passant en paramètres les informations obligatoires (la classe du endpoint et l'url relative associée au endpoint)
ServerEndpointConfig.builder decoders(List<Class<? extends Decoder>> decoders)	Fournir à la configuration les décodeurs qui seront utilisés
ServerEndpointConfig.builder encoders(List<Class<? extends Encoder>> encoders)	Fournir à la configuration les encodeurs qui seront utilisés
ServerEndpointConfig.builder extensions(List<Extension> extensions)	Fournir à la configuration les extensions qui seront utilisées
ServerEndpointConfig.Builder subprotocols(List<String> subprotocols)	Fournir à la configuration les sous-protocoles qui seront utilisés

Exemple (code Java 7) :

```
ServerEndpointConfig config =
ServerEndpointConfig.Builder.create(MonEndpoint.class,"/monendpoint").
    decoders(Arrays.<Class<? extends Decoder>>asList(MonDecoder.class)).
    encoders(Arrays.<Class< extends Encoder>>asList(MonEncoder.class)).build();
```

78.2.3.2.3. La classe ServerEndpointConfig.Configurator

La classe javax.websocket.server.ServerEndpointConfig.Configurator permet de personnaliser certaines opérations lors de demandes de connexion par des clients.

Elle possède plusieurs méthodes :

Méthode	Rôle
boolean checkOrigin(String originHeaderValue)	Vérifier la valeur de l'attribut Origin Header fournie par le client lors d'une demande de connexion. La valeur booléenne renvoyée indique le succès de la vérification
<T> T getEndpointInstance(Class<T> endpointClass)	

	Méthode invoquée par le conteneur pour obtenir une instance du endpoint. L'implémentation par défaut renvoie une nouvelle instance à chaque invocation.
List<Extension> getNegotiatedExtensions(List<Extension> installed, List<Extension> requested)	Renvoyer une collection des extensions supportées par le serveur par rapport à celles fournies en paramètres. La liste doit être vide si aucune des extensions n'est supportée
String getNegotiatedSubprotocol(List<String> supported, List<String> requested)	Renvoyer le sous-protocole sélectionné par le serveur parmi ceux fournis en paramètres : ils correspondent à ceux supportés par un client qui se connecte. Renvoi une chaîne vide si aucun n'est supporté
void modifyHandshake(ServerEndpointConfig sec, HandshakeRequest request, HandshakeResponse response)	Callback invoqué par le serveur permettant de personnaliser la réponse du handshake

78.2.3.2.4. L'interface ServerApplicationConfig

L'interface javax.websocket.server.ServerApplicationConfig permet d'encapsuler la configuration des endpoints à enregistrer dans un conteneur.

Elle définit plusieurs méthodes :

Méthode	Rôle
Set<ServerEndpointConfig> getEndpointConfigs(Set<Class<? extends Endpoint>> endpointClasses)	Renvoyer une collection de type ServerEndpointConfig contenant la configuration des endpoints développés avec l'API
Set<Class<?>> getAnnotatedEndpointClasses(Set<Class<?>> scanned)	Renvoyer une collection des endpoints utilisant des annotations devant être enregistrés dans le serveur

Lors de développement de endpoints sans utiliser les annotations, il faut écrire une classe qui implémente l'interface ServerApplicationConfig pour définir la configuration des endpoints.

L'exemple ci-dessous configure un seul endpoint, défini programmatiquement, associé à l'uri /monecho.

Exemple (code Java 7) :

```
package fr.jmdoudoux.dej.websockets.server;

import java.util.Arrays;
import java.util.Collections;
import java.util.HashSet;
import java.util.Set;
import java.util.logging.Logger;
import javax.websocket.Endpoint;
import javax.websocket.server.ServerApplicationConfig;
import javax.websocket.server.ServerEndpointConfig;

public class MonServerApplicationConfig implements ServerApplicationConfig {

  private static final Logger LOGGER =
    Logger.getLogger(MonServerApplicationConfig.class.getName());

  @Override
  public Set<ServerEndpointConfig> getEndpointConfigs(
    Set<Class<? extends Endpoint>> endpointClasses) {
    return new HashSet<ServerEndpointConfig>(Arrays.asList(
      ServerEndpointConfig.Builder.create(MonEchoEndpoint.class, "/monecho").build()));
  }

  @Override
  public Set<Class<?>> getAnnotatedEndpointClasses(Set<Class<?>> scanned) {
    return Collections.emptySet();
  }
}
```

La manière dont cette classe sera passée au serveur est dépendante de l'implémentation du serveur car cette fonctionnalité n'est pas spécifiée par l'API.

78.2.3.3. La configuration d'un endpoint client

L'interface javax.websocket.ClientEndpointConfig définit les fonctionnalités pour encapsuler la configuration d'un endpoint client. Elle hérite de l'interface Endpoint

Une instance de type ClientEndpointConfig est utilisée par un WebSocketContainer pour enregistrer un endpoint client.

Elle définit plusieurs méthodes :

Méthode	Rôle
ClientEndpointConfig.Configurator getConfigurator()	Renvoyer l'instance de type ClientEndpointConfig.Configurator utilisée par cette configuration
List<Extension> getExtensions()	Renvoyer la liste des extensions
List<String> getSubprotocols()	Renvoyer la liste des sous-protocoles

Pour faciliter la création d'une instance de type ClientEndpointConfig, il faut utiliser la classe ClientEndpointConfig.Builder.

Pour permettre de personnaliser certaines opérations notamment le handshake, il est possible d'utiliser une instance de type ClientEndpointConfig.Configurator.

78.2.3.3.1. La classe ClientEndpointConfig.Builder

La classe javax.websocket.ClientEndpointConfig.Builder permet de créer une instance de type ClientEndpointConfig en utilisant le motif de conception builder.

Elle propose donc plusieurs méthodes qui permettent de fournir les différents éléments qui seront encapsulés dans l'instance, une méthode pour renvoyer l'instance du builder et une pour obtenir l'instance construite :

Méthode	Rôle
ClientEndpointConfig build()	Créer et renvoyer l'instance de type ClientEndpointConfigBuilds encapsulant les attributs fournis au builder
ClientEndpointConfig.builder configurator(ClientEndpointConfig.Configurator clientEndpointConfigurator)	Fournir à la configuration le Configurator qui sera utilisé
static ClientEndpointConfig.builder create()	Créer et retourner une instance de type Builder
ClientEndpointConfig.builder decoders(List<Class<? extends Decoder>> decoders)	Fournir à la configuration les décodeurs qui seront utilisés
ClientEndpointConfig.builder encoders(List<Class<? extends Encoder>> encoders)	Fournir à la configuration les encodeurs qui seront utilisés
ClientEndpointConfig.builder extensions(List<Extension> extensions)	Fournir à la configuration les extensions qui seront utilisées

ServerEndpointConfig.Builder preferredSubprotocols(List<String> preferedSubprotocols)	Fournir à la configuration les sous-protocoles préférés qui seront utilisés

Exemple (code Java 7) :

```
  ClientEndpointConfig maConfig = ClientEndpointConfig.Builder.create()
    .encoders(Arrays.<Class<? extends Encoder>>asList(MonBeanEncoder.class))
    .decoders(Arrays.<Class<? extends Decoder>>asList(MonBeanDecoder.class))
    .preferredSubprotocols(Arrays.asList("sub1", "sub2")).build();
```

78.3. Les encodeurs et les décodeurs

Les messages échangés entre deux endpoints peuvent être du texte ou des données binaires. Les Encoders et les Decoders permettent respectivement de transformer un objet Java en un format sérialisé utilisable dans un message et vice versa.

Un objet Java quelconque peut être encodé sous une forme textuelle ou binaire pour être envoyé au endpoint à l'extrémité de la conversation. Dans ce cas, l'endpoint qui reçoit le message doit posséder un décodeur capable à partir du message de recréer l'objet correspondant. Généralement se sont des formats standards comme XML ou Json qui sont utilisés, ce qui évite d'avoir à réinventer son propre format.

L'API permet d'utiliser trois types de messages :

- texte
- binaire
- pong qui est utilisé par le protocole pour gérer la connexion

Lors du développement d'un endpoint en utilisant les annotations, une méthode permettant de gérer un message pour chacun des trois types peut être annotée avec @OnMessage.

Chaque type est associé à plusieurs types Java correspondant selon les besoins.

Pour les messages de type texte :

- String : pour recevoir l'intégralité du message
- String et un booléen : pour recevoir des morceaux de texte
- un type primitif ou son wrapper correspondant pour recevoir le résultat de la conversion du message dans ce type
- Reader : pour pouvoir traiter le contenu du message sous la forme d'un flux
- une classe quelconque pour laquelle un Decoder pour données textuelles (Decoder.Text ou Decoder.TextStream) est enregistré

Pour les messages de type binaire :

- byte[] ou ByteBuffer : pour recevoir l'intégralité du message
- byte[] ou ByteBuffer et un booléen : pour recevoir des morceaux de message
- InputStream : pour pouvoir traiter le contenu du message sous la forme d'un flux
- une classe quelconque pour laquelle un Decoder pour des données binaires (Decoder.Binary ou Decoder.BinaryStream) est enregistré

Pour les messages de type Pong :

- PongMessage

Lors du développement d'un endpoint en utilisant l'API, un seul MessageHandler peut être enregistré pour chacun des types.

78.3.1. Les encodeurs

Un encodeur permet de transformer un objet Java dans un format texte ou binaire qui pourra être envoyé comme réponse dans un message.

Un encodeur doit implémenter l'interface javax.websocket.Encoder qui définit plusieurs méthodes.

Méthode	Rôle
void destroy()	Cette méthode est invoquée lorsque l'encodeur va être retiré : elle doit permettre de libérer des ressources et de terminer l'encodeur de manière propre
void init(EndpointConfig config)	Cette méthode est invoquée lorsque l'encodeur est associé au endpoint : elle doit permettre d'initialiser l'encodeur

L'interface Encoder possède plusieurs sous-interfaces :

- Encoder.Text pour convertir des objets Java en texte
- Encoder.TextStream pour convertir des objets Java en flux de texte
- Encoder.Binary pour convertir des objets Java en binaire
- Encoder.BinaryStream pour convertir des objets Java en flux binaire

Ces interfaces définissent chacune une méthode encode() avec différentes valeurs de retour et paramètres.

L'exemple ci-dessous va écrire un encodeur pour un bean.

Exemple (code Java 7) :

```java
package fr.jmdoudoux.dej.websockets;

public class MonBean {

    private String nom;
    private String valeur;

    public MonBean() {
    }

    public MonBean(String nom, String valeur) {
        this.nom = nom;
        this.valeur = valeur;
    }

    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }

    public String getValeur() {
        return valeur;
    }

    public void setValeur(String valeur) {
        this.valeur = valeur;
    }
}
```

Les traitements sont réalisés en redéfinissant la méthode encode().

Exemple (code Java 7) :

```java
package fr.jmdoudoux.dej.websockets;

```

```
import java.io.StringWriter;
import javax.json.Json;
import javax.json.stream.JsonGenerator;
import javax.websocket.EncodeException;
import javax.websocket.Encoder;
import javax.websocket.EndpointConfig;

public class MonBeanEncoder implements Encoder.Text<MonBean> {
    @Override
    public String encode(MonBean monBean) throws EncodeException {
        StringWriter writer = new StringWriter();
        JsonGenerator generator = Json.createGenerator(writer);
        generator.writeStartObject()
            .write("nom", monBean.getNom())
            .write("valeur", monBean.getValeur())
            .writeEnd();
        generator.close();
        return writer.toString();
    }

    @Override public void init(EndpointConfig config) {
    }

    @Override public void destroy() {
    }
}
```

Dans l'exemple ci-dessus, l'objet est sérialisé en un message de type texte utilisant le format JSON.

78.3.2. Les décodeurs

Un décodeur permet de transformer le contenu texte ou binaire d'un message en un objet ou un graphe d'objets.

Un décodeur est une classe qui doit implémenter l'interface javax.websocket.Decoder qui définit plusieurs méthodes :

Méthode	Rôle
void destroy()	Cette méthode est invoquée lorsque le décodeur va être retiré : elle doit permettre de libérer des ressources et de terminer le décodeur de manière propre
void init(EndpointConfig config)	Cette méthode est invoquée lorsque le décodeur est associé au endpoint : elle doit permettre d'initialiser le décodeur

L'interface Decoder possède plusieurs sous interfaces :

- Decoder.Text pour convertir du texte en objets Java
- Decoder.TextStream pour convertir un flux de texte en objets Java
- Decoder.Binary pour convertir des données binaires en objets Java
- Decoder.BinaryStream pour convertir un flux binaire en objets Java

Ces interfaces définissent chacune une méthode decode() avec différentes valeurs de retour et paramètres.

L'exemple ci-dessous va écrire un décodeur pour un bean.

Exemple (code Java 7) :

```
package fr.jmdoudoux.dej.websockets;

import java.io.StringReader;
import javax.json.Json;
import javax.json.JsonObject;
import javax.websocket.DecodeException;
import javax.websocket.Decoder;
import javax.websocket.EndpointConfig;

public class MonBeanDecoder implements Decoder.Text<MonBean> {

    @Override
    public MonBean decode(String message) throws DecodeException {
        MonBean resultat = null;
        JsonObject jsonObject = Json.createReader(new StringReader(message)).readObject();
        String nom = jsonObject.getJsonString("nom").getString();
        String valeur = jsonObject.getJsonString("valeur").getString();
        resultat = new MonBean(nom, valeur);
        return resultat;
    }

    @Override
    public boolean willDecode(String message) {
        return message.startsWith("{");
    }

    @Override
    public void init(EndpointConfig config) {
    }

    @Override
    public void destroy() {
    }
}
```

78.3.3. L'enregistrement des encodeurs et des décodeurs

Un endpoint doit connaître l'ensemble des encodeurs/décodeurs qu'il peut utiliser.

En utilisant les annotations, il faut utiliser les attributs encoders et decoders des annotations @ClientEndpoint et @ServerEndpoint.

Exemple (code Java 7) :

```
@ServerEndpoint(value="/monendpoint", encoders
  = MonMessageEncoder.class, decoders= MonMessageDecoder.class)
public class MonEndpoint {
  // ...
}
```

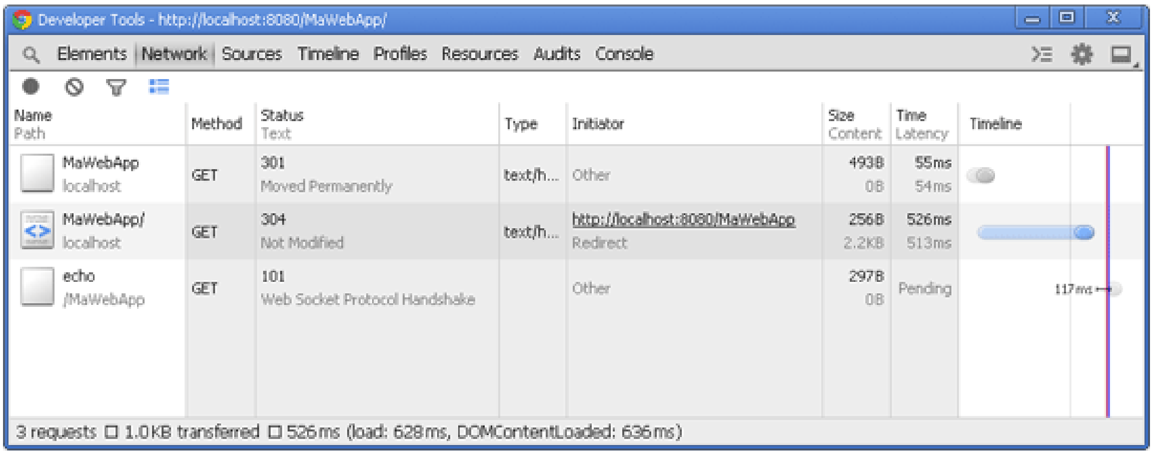
En utilisant l'API, les encodeurs et décodeurs sont obtenus en invoquant les méthodes getEncoders() et getDecoders() de l'interface EndpointConfig.

Les méthodes encoders() et decoders() des classes ClientEndpointConfig.Builder et ServerEndpointConfig.Builder permettent de fournir les encodeurs et décodeurs à l'instance de type EndpointConfig qui sera créée.

78.4. Le débogage des WebSockets

Pour visualiser les requêtes échangées avec le protocole WebSocket, il est possible d'utiliser l'application WireShark.

A partir de la version 20 de Chrome, l'outil Chrome Dev Tools permet de visualiser l'activité d'une WebSocket. Pour l'activer, il suffit d'utiliser l'option « Outils de développement » du menu « Outils »

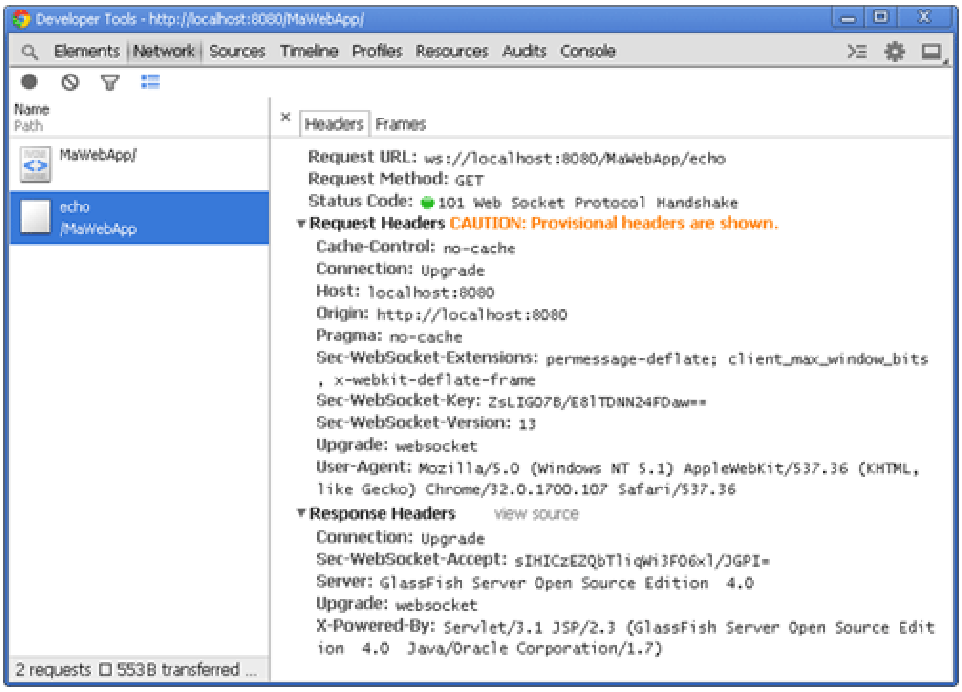

Developer Tools - http://localhost:8080/MaWebApp/
Elements Network Sources Timeline Profiles Resources Audits Console

Name Path	Method	Status Text	Type	Initiator	Size Content	Time Latency	Timeline
MaWebApp localhost	GET	301 Moved Permanently	text/h...	Other	493B 0B	55ms 54ms	
MaWebApp/ localhost	GET	304 Not Modified	text/h...	http://localhost:8080/MaWebApp Redirect	256B 2.2KB	526ms 513ms	
echo /MaWebApp	GET	101 Web Socket Protocol Handshake		Other	297B 0B	Pending	117ms

3 requests ◻ 1.0KB transferred ◻ 526ms (load: 628ms, DOMContentLoaded: 636ms)

Il faut sélectionner la requête qui correspond à la WebSocket.

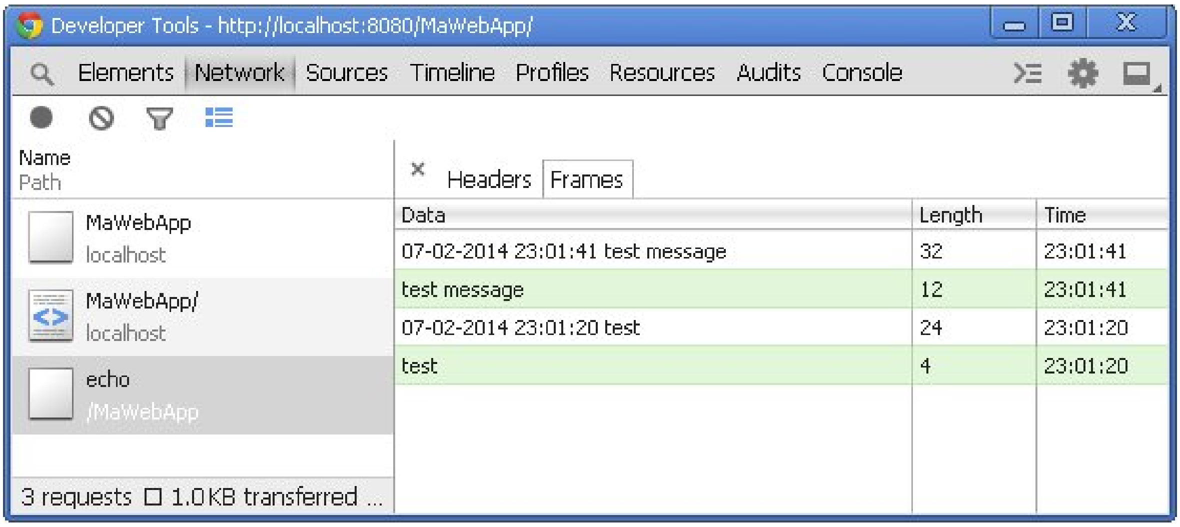
L'onglet « Header » permet de voir la requête et la réponse du handshake


Developer Tools - http://localhost:8080/MaWebApp/
Elements Network Sources Timeline Profiles Resources Audits Console
Name
Path
MaWebApp/
echo
/MaWebApp
Headers
Frames

```
Request URL: ws://localhost:8080/MaWebApp/echo
Request Method: GET
Status Code: ● 101 Web Socket Protocol Handshake
▼Request Headers CAUTION: Provisional headers are shown.
  Cache-Control: no-cache
  Connection: Upgrade
  Host: localhost:8080
  Origin: http://localhost:8080
  Pragma: no-cache
  Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits
  , x-webkit-deflate-frame
  Sec-WebSocket-Key: ZsLIGO7B/E8lTDNN24FDaw==
  Sec-WebSocket-Version: 13
  Upgrade: websocket
  User-Agent: Mozilla/5.0 (Windows NT 5.1) AppleWebKit/537.36 (KHTML,
  like Gecko) Chrome/32.0.1700.107 Safari/537.36
▼Response Headers      view source
  Connection: Upgrade
  Sec-WebSocket-Accept: sIHICzEZQbTliqWi3FO6xl/JGPI=
  Server: GlassFish Server Open Source Edition  4.0
  Upgrade: websocket
  X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server Open Source Edit
  ion  4.0  Java/Oracle Corporation/1.7)
```

2 requests ◻ 553B transferred ...

L'onglet « Frames » permet de visualiser les messages échangés.

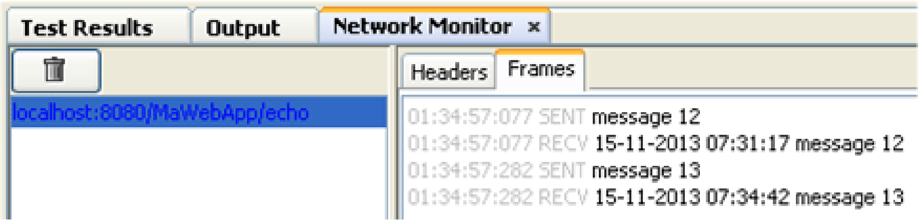

Developer Tools - http://localhost:8080/MaWebApp/
Elements Network Sources Timeline Profiles Resources Audits Console
Name
Path
MaWebApp
localhost
MaWebApp/
localhost
echo
/MaWebApp
Headers
Frames

Data	Length	Time
07-02-2014 23:01:41 test message	32	23:01:41
test message	12	23:01:41
07-02-2014 23:01:20 test	24	23:01:20
test	4	23:01:20

3 requests ◻ 1.0KB transferred ...

L'ouverture de l'url « chrome://net-internals/ » dans un nouvel onglet permet d'obtenir des informations de bas niveau sur les échanges réseaux.

Netbeans, à partir de sa version 7.4, propose la fonctionnalité Network Monitor qui utilise un plug-in Chrome pour afficher les échanges des conversations (handshake HTTP dans l'onglet Headers et messages échangés dans l'onglet Frames).



localhost:8080/MaWebApp/echo

```
01:34:57:077 SENT message 12
01:34:57:077 RECV 15-11-2013 07:31:17 message 12
01:34:57:282 SENT message 13
01:34:57:282 RECV 15-11-2013 07:34:42 message 13
```

Cette fonctionnalité requiert l'utilisation de Chrome comme navigateur et l'installation du plugin correspondant.

78.5. Des exemples d'utilisation

Cette section va proposer plusieurs exemples de mise en oeuvre des WebSockets.

78.5.1. Un premier cas simple

Ce premier cas n'exploite pas toutes les possibilités des Websockets mais il permet de mettre en place un exemple simple. Il va simplement renvoyer la chaîne de caractères reçue en la faisant précéder de la date/heure.

La partie serveur est une webapp qui contient un POJO annoté

Exemple (code Java 7) :

```
package fr.jmdoudoux.dej.websockets;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;
import javax.websocket.OnMessage;
import javax.websocket.server.ServerEndpoint;

@ServerEndpoint("/echo")
public class EchoEndPoint {
    @OnMessage
    public String echo(String message) {
        return ThreadSafeFormatter.getDateFormatter().format(new Date()) + " "
            +  message;
    }
}

class ThreadSafeFormatter {
    private static final ThreadLocal<SimpleDateFormat> formatter =
        new ThreadLocal<SimpleDateFormat>() {
        @Override
        protected SimpleDateFormat initialValue() {
            return new SimpleDateFormat("dd-MM-yyyy HH:mm:ss");
        }
    };

    public static DateFormat getDateFormatter() {
        return formatter.get();
    }
}
```

La partie cliente est une page HTML 5 qui utilise l'API Javascript Websocket.

Exemple :

```
<!DOCTYPE html>
<html>
    <head>
        <title>Test WebSockets</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width">
        <script language="javascript" type="text/javascript">
            var wsUri = getRootUri() + "/EchoWebApp/echo";
            function getRootUri() {
                return "ws://" + (document.location.hostname == "" ?
                    "localhost" : document.location.hostname) + ":" +
                    (document.location.port == "" ? "8080" : document.location.port);
            }

            function init() {
                messageDiv = document.getElementById("messageDivId");
                websocket = new WebSocket(wsUri);
                websocket.onopen = function(evt) {
                    onOpen(evt)
                };
                websocket.onmessage = function(evt) {
                    onMessage(evt)
                };
                websocket.onerror = function(evt) {
                    onError(evt)
                };
            }

            function onOpen(evt) {
                afficher("CONNECTE");
            }

            function onMessage(evt) {
                afficher("RECU : " + evt.data);
            }

            function onError(evt) {
                afficher('<span style="color: red;">ERREUR:</span> ' + evt.data);
            }

            function envoyer() {
                var message = textId.value;
                afficher("ENVOYE : " + message);
                websocket.send(message);
            }

            function afficher(message) {
                var ligne = document.createElement("p");
                ligne.innerHTML = message;
                messageDiv.appendChild(ligne);
            }

            window.addEventListener("load", init, false);
</script>
    </head>
    <body>
        <h2 style="text-align: center;">Client WebSocket Echo</h2>
        <div style="text-align: center;">
            <form action="">
                <input id="textId" name="message" value="" type="text">&nbsp;
                <input onclick="envoyer()" value="Envoyer" type="button">
            </form>
        </div>
        <div id="messageDivId"></div>
    </body>
</html>
```

78.5.2. La mise à jour périodique d'un graphique

Un POJO est défini comme étant l'endpoint d'une WebSocket. Elle conserve une collection des sessions ouvertes sur l'endpoint. La classe contient une méthode statique send() qui envoie la valeur reçue en paramètre à tous les clients connectés à la WebSocket.

Exemple (code Java 7) :

```
package fr.jmdoudoux.dej.websockets;

import java.io.IOException;
import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedQueue;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.websocket.OnClose;
import javax.websocket.OnError;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;

@ServerEndpoint("/valeurs")
public class ValeursEndPoint {
    private static final Logger logger = Logger.getLogger(ValeursEndPoint.class.getName());
    static Queue<Session> queue = new ConcurrentLinkedQueue<>();

    public static void send(double valeur) {
        String message = String.format("%.2f", valeur);
        try {
            for (Session session : queue) {
                session.getBasicRemote().sendText(message);
                logger.log(Level.INFO, "Send: {0} ", message + " to " + session.getId());
            }
        } catch (IOException e) {
            logger.log(Level.WARNING, e.toString());
        }
    }

    @OnOpen
    public void open(Session session) {
        queue.add(session);
    }

    @OnClose
    public void close(Session session) {
        queue.remove(session);
    }

    @OnError
    public void error(Session session, Throwable t) {
        queue.remove(session);
    }
}
```

Un EJB Timer est utilisé pour déterminer une nouvelle valeur aléatoirement toutes les secondes et les envoyer par WebSocket aux différents clients connectés.

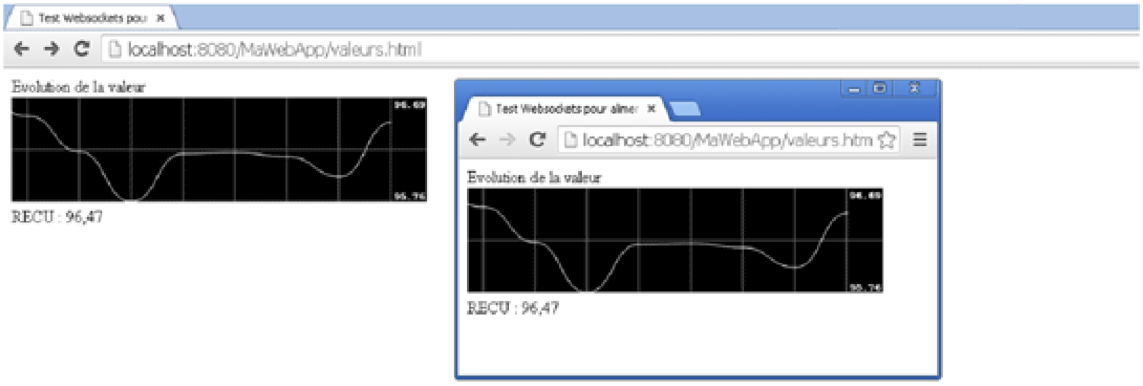
Exemple (code Java 7) :

```
package fr.jmdoudoux.dej.websockets;

import java.util.Random;
import java.util.logging.Logger;
import javax.annotation.PostConstruct;
import javax.annotation.Resource;
import javax.ejb.LocalBean;
import javax.ejb.Schedule;
import javax.ejb.Singleton;
import javax.ejb.Startup;
import javax.ejb.Stateless;
import javax.ejb.TimerConfig;
import javax.ejb.TimerService;

@Singleton
@Startup
@LocalBean
public class ValeursBean {
    private static final Logger logger = Logger.getLogger(ValeursBean.class.getName());
    private Random random;
    private volatile double valeur = 100.0;

    @PostConstruct
    public void init() {
        random = new Random();
    }

    @Schedule(second="*/1", minute="*", hour="*", persistent = false)
    public void timeoutx() {
        valeur += 1.0 * (random.nextInt(100) -50) / 100.0;
        logger.info("nouvelle valeur "+valeur);
        ValeursEndPoint.send(valeur);
    }
}
```

La page web utilise la bibliothèque Javascript Smoothie pour afficher les données reçues sous la forme d'un graphique. Les données sont reçues par la WebSocket.

Exemple :

```
<!DOCTYPE html>
<html>
    <head>
        <title>Test Websockets pour alimenter un graphique</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width">
        <script type="text/javascript" src="smoothie.js"></script>
        <script language="javascript" type="text/javascript">
            var wsUri = getRootUri() + "/MaWebApp/valeurs";
            var line1 = new TimeSeries();
            function getRootUri() {
                return "ws://" + (document.location.hostname == "" ?
                    "localhost" : document.location.hostname) + ":" +
                    (document.location.port == "" ? "8080" : document.location.port);
            }

            function init() {
                messageDiv = document.getElementById("messageDivId");
                websocket = new WebSocket(wsUri);
                websocket.onopen = function(evt) {
                    onOpen(evt)
                };

                websocket.onmessage = function(evt) {
                    onMessage(evt)
                };

                websocket.onerror = function(evt) {
                    onError(evt)
                };
                var smoothie = new SmoothieChart();
                smoothie.streamTo(document.getElementById("graphCanvas"));
                smoothie.addTimeSeries(line1);
            }
            function onOpen(evt) {
                afficher("CONNECTE");
            }
            function onMessage(evt) {
                afficher("RECU : " + evt.data);
                line1.append(new Date().getTime(), parseFloat(evt.data.replace(',', '.')));
            }
            function onError(evt) {
                afficher('<span style="color: red;">ERREUR:</span> ' + evt.data);
            }
            function afficher(message) {
                var ligne = document.createElement("p");
                ligne.innerHTML = message;
                messageDiv.innerHTML = ligne.innerHTML ;
            }
            window.addEventListener("load", init, false);
        </script>
    </head>
    <body>
        <div>Evolution de la valeur</div>
        <canvas id="graphCanvas" width="400" height="100"></canvas>
        <div id="messageDivId"></div>
    </body>
</html>
```


Test Websockets pou
localhost:8080/MaWebApp/valeurs.html
Test Websockets pour alime
localhost:8080/MaWebApp/valeurs.htm

Evolution de la valeur

RECU : 96,47

Evolution de la valeur

RECU : 96,47

78.6. L'utilisation d'implémentations

Une JSR ne définit que des spécifications : il est nécessaire d'utiliser une implémentation de ces spécifications. Cette implémentation peut être :

- l'implémentation de référence
- l'implémentation fournie par l'environnement (un serveur Java EE par exemple)
- une implémentation proposée par un tiers

78.6.1. L'utilisation de Tyrus

Tyrus est l'implémentation de référence de la JSR 356. Elle propose une API dédiée qui permet de lancer un serveur HTTP avec un conteneur pour les websockets qui peut s'exécuter dans une application standalone Java SE.

78.6.1.1. L'utilisation de Tyrus côté client

Le développement d'un endpoint côté client avec Tyrus se fait en utilisant l'API de la JSR 356.

Exemple (code Java 7) :

```
package fr.jmdoudoux.dej.websockets.client;

import java.io.IOException;
import java.net.URI;
import java.net.URISyntaxException;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.websocket.ClientEndpoint;
import javax.websocket.DeploymentException;
import javax.websocket.OnMessage;
import javax.websocket.Session;

@ClientEndpoint
public class TestClientWebSocketTyrus {
  private static final Logger LOGGER =
    Logger.getLogger(TestClientWebSocketTyrus.class.getName());

  @OnMessage
  public void onMessage(String message, Session session) {
    LOGGER.log(Level.INFO, message);
  }

  public static void main(String[] args) {
    LOGGER.log(Level.INFO, "Lancement client ba");
    javax.websocket.WebSocketContainer container
       = javax.websocket.ContainerProvider.getWebSocketContainer();
    try {
      Session session = container.connectToServer(TestClientWebSocketTyrus.class,
        URI.create("ws://localhost:8098/websockets/monecho"));
      while (true) {
        try {
          Thread.sleep(5000);
        } catch (InterruptedException ex) {
          LOGGER.log(Level.SEVERE, null, ex);
        }
        session.getBasicRemote().sendText("hello");
      }
    } catch (DeploymentException | IOException ex) {
      LOGGER.log(Level.SEVERE, "Impossible de se connecter au serveur", ex);
    }
  }
}
```

Dans l'exemple ci-dessus, la classe principale est aussi l'endpoint client. Elle se connecte à un endpoint serveur et lui envoie toutes les cinq secondes le message « hello ». L' endpoint client qu'elle implémente affiche simplement la réponse du serveur.

Pour exécuter un client WebSocket, il faut ajouter au classpath plusieurs bibliothèques : tyrus-client.jar, tyrus-server.jar, tyrus-core.jar, tyrus-websocket-core.jar, tyrus-spi.jar, tyrus-container-servlet.jar et tyrus-container-grizzly.jar

Ceci peut être fait en ajoutant plusieurs dépendances si l'application est un projet Maven : javax.websocket:javax.websocket-api:1.0, org.glassfish.tyrus:tyrus-server:1.1 (l'implémentation de la JSR 356), org.glassfish.tyrus:tyrus-client:1.1 (l'implémentation de la partie cliente) et org.glassfish.tyrus:tyrus-container-grizzly:1.1 (l'implémentation standalone du conteneur)

Exemple :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.jmdoudoux.dej.websockets</groupId>
  <artifactId>TestClientTyrus</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>TestClientTyrus</name>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <dependencies>
    <dependency>
      <groupId>javax.websocket</groupId>
      <artifactId>javax.websocket-api</artifactId>
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>org.glassfish.tyrus</groupId>
      <artifactId>tyrus-server</artifactId>
      <version>1.1</version>
    </dependency>
    <dependency>
      <groupId>org.glassfish.tyrus</groupId>
      <artifactId>tyrus-client</artifactId>
      <version>1.1</version>
    </dependency>
    <dependency>
      <groupId>org.glassfish.tyrus</groupId>
      <artifactId>tyrus-container-grizzly</artifactId>
      <version>1.1</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
    <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.1</version>
        <configuration>
          <compilerVersion>1.7</compilerVersion>
          <source>1.7</source>
          <target>1.7</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Résultat :

```
janv. 06, 2014 09:46:52 PM
fr.jmdoudoux.dej.websockets.client.TestClientWebSocketTyrus main
Infos: Lancement client
janv. 06, 2014 09:46:58 PM
fr.jmdoudoux.dej.websockets.client.TestClientWebSocketTyrus onMessage
Infos: 06-01-2014 09:46:58 (MonEchoEndPoint) hello
janv. 06, 2014 09:47:03 PM
fr.jmdoudoux.dej.websockets.client.TestClientWebSocketTyrus onMessage
Infos: 06-01-2014 09:47:03 (MonEchoEndPoint) hello
janv. 06, 2014 09:47:08 PM
fr.jmdoudoux.dej.websockets.client.TestClientWebSocketTyrus onMessage
Infos: 06-01-2014 09:47:08 (MonEchoEndPoint) hello
```

78.6.1.2. L'utilisation de Tyrus côté serveur

Tyrus propose une API qui permet de démarrer un conteneur qui sera la partie serveur dans une application standalone.

La classe org.glassfish.tyrus.server.Server est une implémentation d'un serveur de WebSockets.

Plusieurs surcharges du constructeur permettent de préciser un ou plusieurs endpoints à enregistrer dans le serveur. Une première surcharge attend en paramètre la classe d'un endpoint défini en utilisant l'annotation @ServerEndpoint.

Exemple (code Java 7) :

```
package fr.jmdoudoux.dej.websockets.server;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.HashSet;
import java.util.Set;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.websocket.DeploymentException;
import org.glassfish.tyrus.server.Server;

public class TestServerWebSocketTyrus {
    private static final Logger LOGGER =
    Logger.getLogger(TestServerWebSocketTyrus.class.getName());

    public static void main(String[] args) {
        Server server = new Server("localhost", 8098, "/websockets", EchoEndpoint.class);
        try {
            LOGGER.log(Level.INFO, "Lancement du serveur");
            server.start();
            BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
            System.out.print("Appuyer sur Entree pour arreter le serveur.");
            reader.readLine();
        } catch (IOException | DeploymentException e) {
            throw new RuntimeException(e);
        } finally {
            LOGGER.log(Level.INFO, "Arret du serveur");
            server.stop();
        }
    }
}
```

Il est possible de passer en troisième paramètres, une classe de type ServerApplicationConfig

Exemple (code Java 7) :

```
    Server server = new Server("localhost", 8098, "/websockets",
      MonServerApplicationConfig.class);
```

La troisième surcharge permet de fournir une collection de type Set contenant les endpoints définis avec l'annotation @ServerEndpoint à enregistrer dans le serveur.

Pour compiler et exécuter cette application, plusieurs dépendances sont requises :

Exemple :

```
<projectxmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.jmdoudoux.dej.websockets</groupId>
  <artifactId>TestServerWebSocketTyrus</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>TestServerWebSocketTyrus</name>
  <url>http://maven.apache.org</url>
    <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
    <dependencies>
      <dependency>
        <groupId>javax.websocket</groupId>
        <artifactId>javax.websocket-api</artifactId>
        <version>1.0</version>
      </dependency>
      <dependency>
        <groupId>org.glassfish.tyrus</groupId>
        <artifactId>tyrus-server</artifactId>
        <version>1.1</version>
      </dependency>
      <dependency>
        <groupId>org.glassfish.tyrus</groupId>
        <artifactId>tyrus-client</artifactId>
        <version>1.1</version>
      </dependency>
      <dependency>
```

```
        <groupId>org.glassfish.tyrus</groupId>
        <artifactId>tyrus-container-grizzly</artifactId>
        <version>1.1</version>
      </dependency>
      <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>3.8.1</version>
        <scope>test</scope>
      </dependency>
    </dependencies>

    <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
            <version>3.1</version>
            <configuration>
              <compilerVersion>1.7</compilerVersion>
              <source>1.7</source>
              <target>1.7</target>
            </configuration>
          </plugin>
        </plugins>
    </build>
</project>
```

L'exécution lance un serveur dédié directement dans l'application standalone.

Résultat :

```
janv. 06, 2014 09:40:38 PM fr.jmdoudoux.dej.websockets.server.TestServerWebSocketTyrus
main
Infos: Lancement du serveur
janv. 06, 2014 09:40:38 PM
org.glassfish.tyrus.server.ServerContainerFactory create
Infos: Provider class loaded:
org.glassfish.tyrus.container.grizzly.GrizzlyEngine
janv. 06, 2014 09:40:39 PM
org.glassfish.grizzly.http.server.NetworkListener start
Infos: Started listener bound to [0.0.0.0:8098]
janv. 06, 2014 09:40:39 PM
org.glassfish.grizzly.http.server.HttpServer start
Infos: [HttpServer] Started.
janv. 06, 2014 09:40:39 PM org.glassfish.tyrus.server.Server start
Infos: WebSocket Registered apps: URLs all start with ws://localhost:8098
Appuyer sur Entree pour arreter le serveur.
janv. 06, 2014 09:40:39 PM org.glassfish.tyrus.server.Server start
Infos: WebSocket server started.
janv. 06, 2014 09:42:31 PM
fr.jmdoudoux.dej.websockets.server.TestServerWebSocketTyrus main
Infos: Arret du serveur
janv. 06, 2014 09:42:32 PM
org.glassfish.grizzly.http.server.NetworkListener stop
Infos: Stopped listener bound to [0.0.0.0:8098]
janv. 06, 2014 09:42:32 PM org.glassfish.tyrus.server.Server stop
Infos: Websocket Server stopped.
```

78.6.2. L'utilisation de Javascript côté client

HTML 5 propose une API Javascript pour permettre d'utiliser les Websockets dans les pages Web.

La majorité des navigateurs récents supportent le protocole WebSocket : il est possible de consulter l'url https://caniuse.com/#feat=websockets ou https://caniuse.com/websockets pour s'assurer du support pour une version donnée d'un navigateur.

La classe Javascript WebSocket est l'élément principal pour utiliser les websockets dans une page Web.

Il faut créer un objet de type WebSocket

var socket = new WebSocket(url, [sub-protocol]);

Il attend en paramètre l'URL de la websocket : le protocole de l'URL doit être ws:// ou wss:///

Le second paramètre, optionnel, permet de préciser le ou les sous-protocoles utilisables pour la communication avec le serveur. Lors de la connexion, le serveur sélectionnera un de ceux-ci s'il le supporte. Dès que la connexion est établie, la propriété protocol de la classe WebSocket permet de connaître le protocol choisi par le serveur.

Résultat :

```
var socket = new WebSocket("ws://localhost:8080/websockets/monbean",
  ["procotole1","protocole2"]);
```

La classe WebSocket possède plusieurs attributs :

Attribut	Rôle
readyState	Fournir l'état de la connexion • 0 : la connexion n'est pas encore établie • 1 : la connexion est établie • 2 : la connexion est en cours de fermeture • 3 : la connexion est fermée
bufferedAmount	Nombre d'octets

La classe WebSocket possède plusieurs événements liés au cycle de vie de la websocket :

Evénement	Rôle
onopen	La connexion est établie
onmessage	Un message est reçu. Les données sont stockées dans la propriété data du paramètre
onerror	Une erreur est survenue
onclose	La connexion est fermée

Avant de se connecter au endpoint, il faut associer des gestionnaires sur les événements utiles liés au cycle de vie de la websocket.

La classe WebSocket possède deux méthodes :

Méthode	Rôle
send()	Envoyer un message par la websocket
close()	Fermer la connexion

Exemple :

```
<!DOCTYPE html>
<html>
    <head>
        <title>Test Websockets pour obtenir des données</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width">
        <script language="javascript" type="text/javascript">
            var wsUri = getRootUri() + "/MaWebApp/monbean";
            var websocket;
            var id = 0;
            function getRootUri() {
                return "ws://" + (document.location.hostname == "" ?
                        "localhost" : document.location.hostname) + ":" +
                        (document.location.port == "" ? "8080" : document.location.port);
            }
            function init() {
                messageDiv = document.getElementById("messageDivId");
                websocket = new WebSocket(wsUri);
                websocket.onopen = function(evt) {
                    onOpen(evt)
                };
                websocket.onmessage = function(evt) {
                    onMessage(evt)
                };
                websocket.onerror = function(evt) {
                    onError(evt)
                };
                setInterval(sendMessage,5000);
            }
            function close() {
                websocket.close();
            }
            function onOpen(evt) {
                afficher("CONNECTE");
            }
            function onMessage(evt) {
                afficher("RECU : " + evt.data);
                afficherDonnees(evt.data)
            }
            function onError(evt) {
                afficher('<span style="color: red;">ERREUR:</span> ' + evt.data);
            }
            function afficherDonnees(message) {
                donnees = JSON.parse(message);
                var nom = document.getElementById("nom");
                var valeur = document.getElementById("valeur");
                nom.value = donnees.nom;
                valeur.value = donnees.valeur;
            }
            function afficher(message) {
                var ligne = document.createElement("p");
                ligne.innerHTML = message;
                messageDiv.innerHTML = ligne.innerHTML ;
            }

           function sendMessage() {
              id = id + 1;
              websocket.send(id);
            }

            window.addEventListener("load", init, false);
            window.addEventListener("unload", close, false);
        </script>
    </head>
    <body>
      <div id="donnees">
        <input id="nom" class="text-field" type="text" placeholder="Nom" readonly />
        <input id="valeur" class="text-field" type="text" placeholder="Valeur" readonly />
        <input class="button" type="submit" value="Send" onclick="sendMessage();" />
      </div>
      <div id="messageDivId"></div>
    </body>
</html>
```

Il est important de gérer correctement les événements du cycle de vie de la websocket. La connexion est généralement bien gérée car sinon aucun messages n'est émis ou reçus. Par contre, la déconnexion doit être correctement gérée notamment pour permettre de libérer les ressources côtés serveur : ce traitement peut par exemple se faire dans l'événement onunload() du tag <body> de la page.

Il est possible de tester si le navigateur est capable de mettre en oeuvre les websockets.

Résultat :

```
if(window.WebSocket) {
  // utilisation de la websocket
} else {
  alert('Le navigateur ne supporte pas les websockets');
}
```

Le protocole WebSocket est développé pour être utilisé par différents types de clients mais tous n'ont pas un support équivalent des fonctionnalités. Ainsi, il n'est pas possible d'échanger des données binaires avec un client utilisant Javascript.

Partie 11 : Le développement d'applications web

Partie 11 : Développement d'applications web

Plusieurs frameworks standard de Java EE ou open source peuvent être utilisés pour développer des applications web.

Cette partie contient plusieurs chapitres :

- Les servlets : plonge au coeur de l'API servlet qui est un des composants de base pour le développement d'applications Web
- Les JSP (Java Server Pages) : poursuit la discussion sur les servlets en explorant un mécanisme basé sur celles-ci pour réaliser facilement des pages web dynamiques
- JSTL (Java server page Standard Tag Library) : est un ensemble de bibliothèques de tags personnalisés communément utilisé dans les JSP
- Struts : présente et détaille la mise en oeuvre du framework open source de développement d'applications web le plus populaire
- JSF (Java Server Faces) : détaille l'utilisation de la technologie Java Server Faces (JSF) dont le but est de proposer un framework qui facilite et standardise le développement d'applications web avec Java.
- D'autres frameworks pour les applications web : présente rapidement quelques frameworks open source pour le développement d'applications web

79. Les servlets

Chapitre 79

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

A la base, les serveurs web sont seulement capables de renvoyer des fichiers présents sur le serveur en réponse à une requête d'un client. Cependant, pour permettre l'envoi d'une page HTML contenant par exemple une liste d'articles répondant à différents critères, il faut créer dynamiquement cette page HTML. Plusieurs solutions existent pour ces traitements. Les servlets Java sont une de ces solutions.

Mais les servlets peuvent aussi servir à d'autres usages.

Des informations sur les servlets sont disponibles sur la page :
https://www.oracle.com/java/technologies/servlet-technology.html

Ce chapitre contient plusieurs sections :

- La présentation des servlets
- L'API servlet
- Le protocole HTTP
- Les servlets http
- Les informations sur l'environnement d'exécution des servlets
- L'utilisation des cookies
- Le partage d'informations entre plusieurs échanges HTTP
- Packager une application web
- L'utilisation de Log4J dans une servlet

79.1. La présentation des servlets

Une servlet est un programme qui s'exécute côté serveur en tant qu'extension du serveur. Elle reçoit une requête du client, elle effectue des traitements et renvoie le résultat. La liaison entre la servlet et le client peut être directe ou passer par un intermédiaire comme par exemple un serveur http.

Même si pour le moment la principale utilisation des servlets est la génération de pages html dynamiques utilisant le protocole http et donc un serveur web, n'importe quel protocole reposant sur le principe de requête/réponse peut faire usage d'une servlet.

Ecrite en Java, une servlet en retire ses avantages : la portabilité, l'accès à toutes les API de Java dont JDBC pour l'accès aux bases de données, ...

Une servlet peut être invoquée plusieurs fois en même temps pour répondre à plusieurs requêtes simultanées.

Dans une architecture Client/Serveur trois tiers, la servlet se positionne dans le tiers du milieu entre le client léger chargé de l'affichage et la source de données.

Il existe plusieurs versions des spécifications de l'API Servlets :

Version	
2.0	1997
2.1	Novembre 1998, partage d'informations grâce au Servletcontext La classe GenericServlet implémente l'interface ServletConfig une méthode log() standard pour envoyer des informations dans le journal du conteneur objet RequestDispatcher pour le transfert du traitement de la requête vers une autre ressource ou inclure le résultat d'une autre ressource
2.2	Aout 1999, format war pour un déploiement standard des applications web mise en buffer de la réponse inclus dans J2EE 1.2
2.3	Septembre 2001, JSR 053 : nécessite Java 1.2 minimum ajout d'un mécanisme de filtre ajout de méthodes pour la gestion d'événements liés à la création et la destruction du context et de la session inclus dans J2EE 1.3
2.4	Novembre 2003, JSR 154 inclus dans J2EE 1.4
2.5	Septembre 2005, JSR 154 inclus dans Java EE 5, nécessite Java SE 5 minimum
3.0	Décembre 2009, JSR 315 inclus dans Java EE 6, nécessite Java SE 6 minimum
3.1	Mai 2013, JSR 340 inclus dans Java EE 7
4.0	Septembre 2017, JSR 369 inclus dans Java EE 8

79.1.1. Le fonctionnement d'une servlet (cas d'utilisation de http)

Un serveur d'applications permet de charger et d'exécuter les servlets dans une JVM. C'est une extension du serveur web. Ce serveur d'applications contient entre autres un moteur de servlets qui se charge de manager les servlets qu'il contient.

Pour exécuter une servlet, il suffit de saisir une URL qui désigne la servlet dans un navigateur.

1. Le serveur reçoit du navigateur la requête http qui a recours à une servlet
2. Si c'est la première sollicitation de la servlet, le serveur l'instancie. Les servlets sont stockées (sous forme de fichiers .class) dans un répertoire particulier du serveur. Ce répertoire dépend du serveur d'applications utilisé. La servlet reste en mémoire jusqu'à l'arrêt du serveur. Certains serveurs d'applications permettent aussi d'instancier des servlets dès le lancement du serveur.
   Au fil des requêtes, la servlet peut être appelée par plusieurs threads lancés par le serveur. Ce principe de fonctionnement évite d'instancier un objet de type servlet à chaque requête et permet de maintenir un ensemble de ressources actives telles qu'une connexion à une base de données.
3. Le serveur crée un objet qui représente la requête http ainsi que l'objet qui contiendra la réponse et les envoie à la servlet
4. La servlet crée dynamiquement la réponse sous forme de page html transmise par un flux dans l'objet contenant la réponse. La création de cette réponse utilise bien sûr la requête du client mais aussi un ensemble de ressources incluses sur le serveur telles que des fichiers ou des bases de données.
5. Le serveur récupère l'objet réponse et envoie la page html au client.

79.1.2. Les outils nécessaires pour développer des servlets

Initialement, pour développer des servlets avec le JDK Standard Edition, il fallait utiliser le Java Server Development Kit (JSDK) qui est une extension du JDK. Pour réaliser les tests, le JSDK fournissait, dans sa version 2.0 un outil nommé servletrunner et depuis sa version 2.1, il fournit un serveur http allégé.

Actuellement, pour exécuter des applications web, il faut utiliser un conteneur web ou un serveur d'applications : il existe de nombreuses versions commerciales telles que IBM Webpshere ou BEA WebLogic mais aussi des versions libres telles que Tomcat du projet GNU Jakarta.

Ce serveur d'applications ou ce conteneur web doit utiliser ou inclure un serveur http dont le plus utilisé est Apache.

Le choix d'un serveur d'applications ou d'un conteneur web doit tenir compte de la version du JSDK qu'il supporte pour être compatible avec celle utilisée pour le développement des servlets. Le choix entre un serveur commercial et un libre doit tenir compte principalement du support technique, des produits annexes fournis et des outils d'installation et de configuration.

Pour simplement développer des servlets, le choix d'un serveur libre se justifie pleinement de part sa gratuité et sa « légèreté ».

79.1.3. Le rôle du conteneur web

Un conteneur web est un moteur de servlets qui prend en charge et gère les servlets : chargement de la servlet, gestion de son cycle de vie, passage des requêtes et des réponses ... Un conteneur web peut être intégré dans un serveur d'applications qui va contenir d'autres conteneurs et éventuellement proposer d'autres services..

Le chargement et l'instanciation d'une servlet se font selon le paramétrage soit au lancement du serveur soit à la première invocation de la servlet. Dès l'instanciation, la servlet est initialisée une seule et unique fois avant de pouvoir répondre aux requêtes. Cette initialisation peut permettre de mettre en place l'accès à des ressources telles qu'une base de données.

79.1.4. Les différences entre les servlets et les CGI

Les programmes ou script CGI (Common Gateway Interface) sont aussi utilisés pour générer des pages HTML dynamiques. Ils représentent la plus ancienne solution pour réaliser cette tâche.

Un CGI peut être écrit dans de nombreux langages.

Il existe plusieurs avantages à utiliser des servlets plutôt que des CGI :

- la portabilité offerte par Java bien que certains langages de script tels que PERL tournent sur plusieurs plates-formes.
- la servlet reste en mémoire une fois instanciée ce qui permet de garder des ressources systèmes et gagner le temps de l'initialisation. Un CGI est chargé en mémoire à chaque requête, ce qui réduit les performances.
- les servlets possèdent les avantages de toutes les classes écrites en Java : accès aux API, aux JavaBeans, le garbage collector, ...

79.2. L'API servlet

Les servlets sont conçues pour agir selon un modèle de requête/réponse. Tous les protocoles utilisant ce modèle peuvent être utilisés : http, ftp, etc ...

L'API servlets est une extension du jdk de base, et en tant que telle elle est regroupée dans des packages préfixés par javax.

L'API servlet regroupe un ensemble de classes dans deux packages :

- javax.servlet : contient les classes pour développer des serlvets génériques indépendantes d'un protocole
- javax.servlet.http : contient les classes pour développer des servlets qui reposent sur le protocole http utilisé par

les serveurs web.

Le package javax.servlet définit plusieurs interfaces, méthodes et exceptions :

javax.servlet	Nom	Rôle
Les interfaces	RequestDispatcher	Définition d'un objet qui permet le renvoi d'une requête vers une autre ressource du serveur (une autre servlet, une JSP ...)
	Servlet	Définition de base d'une servlet
	ServletConfig	Définition d'un objet pour configurer la servlet
	ServletContext	Définition d'un objet pour obtenir des informations sur le contexte d'exécution de la servlet
	ServletRequest	Définition d'un objet contenant la requête du client
	ServletResponse	Définition d'un objet qui contient la réponse renvoyée par la servlet
	SingleThreadModel	Permet de définir une servlet qui ne répondra qu'à une seule requête à la fois
Les classes	GenericServlet	Classe définissant une servlet indépendante de tout protocole
	ServletInputStream	Flux permettant la lecture des données de la requête cliente
	ServletOutPutStream	Flux permettant l'envoi de la réponse de la servlet
Les exceptions	SevletException	Exception générale en cas de problème durant l'exécution de la servlet
	UnavailableException	Exception levée si la servlet n'est pas disponible

Le package javax.servlet.http définit plusieurs interfaces et méthodes :

Javax.servlet	Nom	Rôle
Les interfaces	HttpServletRequest	Hérite de ServletRequest : définit un objet contenant une requête selon le protocole http
	HttpServletResponse	Hérite de ServletResponse : définit un objet contenant la réponse de la servlet selon le protocole http
	HttpSession	Définit un objet qui représente une session
Les classes	Cookie	Classe représentant un cookie (ensemble de données sauvegardées par le brower sur le poste client)
	HttpServlet	Hérite de GenericServlet : classe définissant une servlet utilisant le protocole http
	HttpUtils	Classe proposant des méthodes statiques utiles pour le développement de servlets http

79.2.1. L'interface Servlet

Une servlet est une classe Java qui implémente l'interface javax.servlet.Servlet. Cette interface définit 5 méthodes qui permettent au conteneur web de dialoguer avec la servlet : elle encapsule ainsi les méthodes nécessaires à la communication entre le conteneur et la servlet.

Méthode	Rôle
void service (ServletRequest req, ServletResponse res)	Cette méthode est exécutée par le conteneur lorsque la servlet est sollicitée : chaque requête du client déclenche une seule exécution de cette méthode. Cette méthode pouvant être exécutée par plusieurs threads, il faut

	prévoir un processus d'exclusion pour l'utilisation de certaines ressources.
void init(ServletConfig conf)	Initialisation de la servlet. Cette méthode est appelée une seule fois après l'instanciation de la servlet. Aucun traitement ne peut être effectué par la servlet tant que l'exécution de cette méthode n'est pas terminée.
ServletConfig getServletConfig()	Renvoie l'objet ServletConfig passé à la méthode init
void destroy()	Cette méthode est appelée lors de la destruction de la servlet. Elle permet de libérer proprement certaines ressources (fichiers, bases de données ...). C'est le serveur qui appelle cette méthode.
String getServletInfo()	Renvoie des informations sur la servlet.

Les méthodes init(), service() et destroy() assurent le cycle de vie de la servlet en étant respectivement appelées lors de la création de la servlet, lors de son appel pour le traitement d'une requête et lors de sa destruction.

La méthode init() est appelée par le serveur juste après l'instanciation de la servlet.

La méthode service() ne peut pas être invoquée tant que la méthode init() n'est pas terminée.

La méthode destroy() est appelée juste avant que le serveur ne détruise la servlet : cela permet de libérer des ressources allouées dans la méthode init() telles qu'un fichier ou une connexion à une base de données.

79.2.2. La requête et la réponse

L'interface ServletRequest définit plusieurs méthodes qui permettent d'obtenir des données sur la requête du client :

Méthode	Rôle
ServletInputStream getInputStream()	Permet d'obtenir un flux pour les données de la requête
BufferedReader getReader()	Idem

L'interface ServletResponse définit plusieurs méthodes qui permettent de fournir la réponse faite par la servlet suite à ses traitements :

Méthode	Rôle
SetContentType	Permet de préciser le type MIME de la réponse
ServletOutputStream getOutputStream()	Permet d'obtenir un flux pour envoyer la réponse
PrintWriter getWriter()	Permet d'obtenir un flux pour envoyer la réponse

79.2.3. Un exemple de servlet

Une servlet qui implémente simplement l'interface Servlet doit évidemment redéfinir toutes les méthodes de l'interface.

Il est très utile lorsque que l'on crée une servlet qui implémente directement l'interface Servlet de sauvegarder l'objet ServletConfig fourni par le conteneur en paramètre de la méthode init() car c'est le seul moment où l'on a accès à cet objet.

Exemple (code Java 1.1) :

```
import java.io.*;
import javax.servlet.*;

public class TestServlet implements Servlet {
  private ServletConfig cfg;

  public void init(ServletConfig config) throws ServletException {
     cfg = config;
  }

  public ServletConfig getServletConfig() {
    return cfg;
  }

  public String getServletInfo() {
    return "Une servlet de test";
  }

  public void destroy() {
  }

  public void service (ServletRequest req,  ServletResponse res )
  throws ServletException, IOException  {
    res.setContentType( "text/html" );
    PrintWriter out = res.getWriter();
    out.println( "<HTML>" );
    out.println( "<HEAD>");
    out.println( "<TITLE>Page generee par une servlet</TITLE>" );
    out.println( "</HEAD>" );
    out.println( "<BODY>" );
    out.println( "<H1>Bonjour</H1>" );
    out.println( "</BODY>" );
    out.println( "</HTML>" );
    out.close();
  }
}
```

79.3. Le protocole HTTP

Le protocole HTTP est un protocole qui fonctionne sur le modèle client/serveur. Un client qui est une application (souvent un navigateur web) envoie une requête à un serveur (un serveur web). Ce serveur attend en permanence les requêtes sur un port particulier (par défaut le port 80). A la réception de la requête, le serveur lance un thread qui va la traiter pour générer la réponse. Le serveur renvoie la réponse au client une fois les traitements terminés.

Une particularité du protocole HTTP est de maintenir la connexion entre le client et le serveur uniquement durant l'échange de la requête et de la réponse.

Il existe deux versions principales du protocole HTTP : 1.0 et 1.1.

La requête est composée de trois parties :

- la commande
- la section en-tête
- le corps

La première ligne de la requête contient la commande à exécuter par le serveur. La commande est suivie éventuellement d'un argument qui précise la commande (par exemple l'url de la ressource demandée). Enfin la ligne doit contenir la version du protocole HTTP utilisé, précédée de HTTP/.

Exemple :

GET / index.html HTTP/1.0

Avec HTTP 1.1, les commandes suivantes sont définies : GET, POST, HEAD, OPTIONS, PUT, DELETE, TRACE et CONNECT. Les trois premières sont les plus utilisées.

Il est possible de fournir sur les lignes suivantes de la partie en-tête des paramètres supplémentaires. Cette partie en-tête est optionnelle. Les informations fournies peuvent permettre au serveur d'obtenir des informations sur le client. Chaque information doit être mise sur une ligne unique. Le format est nom_du_champ:valeur. Les champs sont prédéfinis et sont sensibles à la casse.

Une ligne vide doit précéder le corps de la requête. Le contenu du corps de la requête dépend du type de la commande.

La requête doit obligatoirement être terminée par une ligne vide.

La réponse est elle aussi composée des trois mêmes parties :

- une ligne de statuts
- un en-tête dont le contenu est normalisé
- un corps dont le contenu dépend totalement de la requête

La première ligne de l'en-tête contient un état qui est composé : de la version du protocole HTTP utilisé, du code de statut et d'une description succincte de ce code.

Le code de statut est composé de trois chiffres qui donnent des informations sur le résultat du traitement qui a généré cette réponse. Ce code peut être associé à une catégorie en fonction de sa valeur :

Plage de valeurs du code	Signification
100 à 199	Information
200 à 299	Traitement avec succès
300 à 399	La requête a été redirigée
400 à 499	La requête est incomplète ou erronée
500 à 599	Une erreur est intervenue sur le serveur

Plusieurs codes sont définis par le protocole HTTP dont les plus importants sont :

- 200 : traitement correct de la requête
- 204 : traitement correct de la requête mais la réponse ne contient aucun contenu (ceci permet au browser de laisser la page courante affichée)
- 404 : la ressource demandée n'est pas trouvée (sûrement le plus célèbre)
- 500 : erreur interne du serveur

L'en-tête contient des informations qui précisent le contenu de la réponse.

Le corps de la réponse est précédé par une ligne vide.

La suite de cette section sera développée dans une version future de ce document

79.4. Les servlets http

L'usage principal des servlets est la création de pages HTML dynamiques. L'API fournit une classe qui encapsule une servlet utilisant le protocole http. Cette classe est la classe HttpServlet.

Cette classe hérite de GenericServlet, donc elle implémente l'interface Servlet, et redéfinit toutes les méthodes nécessaires pour fournir un niveau d'abstraction permettant de développer facilement des servlets avec le protocole http.

Ce type de servlet n'est pas utile seulement pour générer des pages HTML bien que cela soit son principal usage, elle peut aussi réaliser un ensemble de traitements tels que mettre à jour une base de données. En réponse, elle peut générer une page html qui indique le succès ou non de la mise à jour. Une servlets peut aussi par exemple renvoyer une image qu'elle aura dynamiquement générée en fonction de certains paramètres.

Elle définit un ensemble de fonctionnalités très utiles : par exemple, elle contient une méthode service() qui appelle certaines méthodes à redéfinir en fonction du type de requête http (doGet(), doPost(), etc ...).

La requête du client est encapsulée dans un objet qui implémente l'interface HttpServletRequest : cet objet contient les données de la requête et des informations sur le client.

La réponse de la servlet est encapsulée dans un objet qui implémente l'interface HttpServletResponse.

Typiquement pour définir une servlet, il faut définir une classe qui hérite de la classe HttpServlet et redéfinir la méthode doGet() et/ou doPost() selon les besoins.

La méthode service() héritée de HttpServlet appelle l'une ou l'autre de ces méthodes en fonction du type de la requête http :

- une requête GET : c'est une requête qui permet au client de demander une ressource
- une requête POST : c'est une requête qui permet au client d'envoyer des informations issues par exemple d'un formulaire

Une servlet peut traiter un ou plusieurs types de requêtes grâce à plusieurs autres méthodes :

- doHead() : pour les requêtes http de type HEAD
- doPut() : pour les requêtes http de type PUT
- doDelete() : pour les requêtes http de type DELETE
- doOptions() : pour les requêtes http de type OPTIONS
- doTrace() : pour les requêtes http de type TRACE

La classe HttpServlet hérite aussi de plusieurs méthodes définies dans l'interface Servlet : init(), destroy() et getServletInfo().

79.4.1. La méthode init()

Si cette méthode doit être redéfinie, il est important d'invoquer la méthode héritée avec un appel à super.init(config), config étant l'objet fourni en paramètre de la méthode. Cette méthode définie dans la classe HttpServlet sauvegarde l'objet de type ServletConfig.

De plus, la classe GenericServlet implémente l'interface ServletConfig. Les méthodes redéfinies pour cette interface utilisent l'objet sauvegardé. Ainsi, la servlet peut utiliser sa propre méthode getInitParameter() ou utiliser la méthode getInitParameter() de l'objet de type ServletConfig. La première solution permet un usage plus facile dans toute la servlet.

Sans l'appel à la méthode héritée lors d'une redéfinition, la méthode getInitParameter() de la servlet lèvera une exception de type NullPointerException.

79.4.2. L'analyse de la requête

La méthode service() est la méthode qui est appelée lors de l'invocation de la servlet.

Par défaut dans la classe HttpServlet, cette méthode contient du code qui réalise une analyse de la requête client contenue dans l'objet HttpServletRequest. Selon le type de requête GET ou POST, elle appelle la méthode doGet() ou doPost(). C'est bien le type de requête qui indique la méthode à utiliser dans la servlet.

Ainsi, la méthode service() n'est pas à redéfinir pour ces requêtes et il suffit de redéfinir les méthodes doGet() et/ou doPost() selon les besoins.

79.4.3. La méthode doGet()

Une requête de type GET est utile avec des liens. Par exemple :

```
<A HREF="http://localhost:8080/examples/servlet/tomcat1.MyHelloServlet">test de
la servlet</A>
```

Dans une servlet de type HttpServlet, une telle requête est associée à la méthode doGet().

La signature de la méthode doGet() :

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException {
}
```

Le traitement typique de la méthode doGet() est d'analyser les paramètres de la requête, alimenter les données de l'en-tête de la réponse et d'écrire la réponse.

79.4.4. La méthode doPost()

Une requête POST n'est utilisable qu'avec un formulaire HTML.

Exemple : de code HTML

```
<FORM ACTION="http://localhost:8080/examples/servlet/tomcat1.TestPostServlet"
METHOD="POST">
<INPUT NAME="NOM">
<INPUT NAME="PRENOM">
<INPUT TYPE="ENVOYER">
</FORM>
```

Dans l'exemple ci-dessus, le formulaire comporte deux zones de saisies correspondant à deux paramètres : NOM et PRENOM.

Dans une servlet de type HttpServlet, une telle requête est associée à la méthode doPost().

La signature de la méthode doPost() :

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws IOException {
}
```

La méthode doPost() doit généralement recueillir les paramètres pour les traiter et générer la réponse. Pour obtenir la valeur associée à chaque paramètre il faut utiliser la méthode getParameter() de l'objet HttpServletRequest. Cette méthode attend en paramètre le nom du paramètre dont on veut la valeur. Ce paramètre est sensible à la casse.

Exemple :

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException {
   String nom = request.getParameter("NOM");
   String prenom = request.getParameter("PRENOM");
}
```

79.4.5. La génération de la réponse

La servlet envoie sa réponse au client en utilisant un objet de type HttpServetResponse. HttpServletResponse est une interface : il n'est pas possible d'instancier un tel objet mais le moteur de servlets instancie un objet qui implémente cette interface et le passe en paramètre de la méthode service.

Cette interface possède plusieurs méthodes pour mettre à jour l'en-tête http et la page HTML de retour.

Méthode	Rôle
void sendError (int)	Envoie une erreur avec un code retour et un message par défaut
void sendError (int, String)	Envoie une erreur avec un code retour et un message
void setContentType(String)	Héritée de ServletResponse, cette méthode permet de préciser le type MIME de la réponse
void setContentLength(int)	Héritée de ServletResponse, cette méthode permet de préciser la longueur de la réponse
ServletOutputStream getOutputStream()	Héritée de ServletResponse, elle retourne un flux pour l'envoi de la réponse
PrintWriter getWriter()	Héritée de ServletResponse, elle retourne un flux pour l'envoi de la réponse

Avant de générer la réponse sous forme de page HTML, il faut indiquer dans l'en-tête du message http, le type MIME du contenu du message. Ce type sera souvent « text/html » qui correspond à une page HTML mais il peut aussi prendre d'autres valeurs en fonction de ce que retourne la servlet (une image par exemple). La méthode à utiliser est setContentType().

Il est aussi possible de préciser la longueur de la réponse avec la méthode setContentLength(). Cette précision est optionnelle mais si elle est utilisée, la longueur doit être exacte pour éviter des problèmes.

Il est préférable de créer une ou plusieurs méthodes recevant en paramètre l'objet HttpServletResponse qui seront dédiées à la génération du code HTML afin de ne pas alourdir les méthodes doXXX().

Il existe plusieurs façons de générer une page HTML : elles utiliseront toutes soit la méthode getOutputStream() ou getWriter() pour obtenir un flux dans lequel la réponse sera envoyée.

- Utilisation d'un StringBuffer et getOutputStream

Exemple (code Java 1.1) :

```
protected void GenererReponse1(HttpServletResponse reponse) throws IOException {
   //creation de la reponse
   StringBuffer sb = new StringBuffer();
   sb.append("<HTML>\n");
   sb.append("<HEAD>\n");
   sb.append("<TITLE>Bonjour</TITLE>\n");
   sb.append("</HEAD>\n");
   sb.append("<BODY>\n");
   sb.append("<H1>Bonjour</H1>\n");
   sb.append("</BODY>\n");
   sb.append("</HTML>");

   // envoi des infos de l'en-tete
   reponse.setContentType("text/html");
   reponse.setContentLength(sb.length());

   // envoi de la réponse
   reponse.getOutputStream().print(sb.toString());
}
```

L'avantage de cette méthode est qu'elle permet facilement de déterminer la longueur de la réponse.

Dans l'exemple, l'ajout des retours chariot '\n' à la fin de chaque ligne n'est pas obligatoire mais facilite la compréhension du code HTML surtout s'il devient plus complexe.

- Utilisation directe de getOutputStream

Exemple :

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class TestServlet4 extends HttpServlet {

   public void doGet(HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException {
      res.setContentType("text/html");
      ServletOutputStream out = res.getOutputStream();
      out.println("<HTML>\n");
      out.println("<HEAD>\n");
      out.println("<TITLE>Bonjour</TITLE>\n");
      out.println("</HEAD>\n");
      out.println("<BODY>\n");
      out.println("<H1>Bonjour</H1>\n");
      out.println("</BODY>\n");
      out.println("</HTML>");
   }
}
```

- Utilisation de la méthode getWriter()

Exemple (code Java 1.1) :

```
protected void GenererReponse2(HttpServletResponse reponse) throws IOException {

   reponse.setContentType("text/html");

   PrintWriter out = reponse.getWriter();

   out.println("<HTML>");
   out.println("<HEAD>");
   out.println("<TITLE>Bonjour</TITLE>");
   out.println("</HEAD>");
   out.println("<BODY>");
   out.println("<H1>Bonjour</H1>");
   out.println("</BODY>");
   out.println("</HTML>");
}
```

Avec cette méthode, il faut préciser le type MIME avant d'écrire la réponse. L'emploi de la méthode println() permet d'ajouter un retour chariot en fin de chaque ligne.

Si un problème survient lors de la génération de la réponse, la méthode sendError() permet de renvoyer une erreur au client : un code retour est positionné dans l'en-tête http et le message est indiqué dans une simple page HTML.

79.4.6. Un exemple de servlet HTTP très simple

Toute servlet doit au moins importer trois packages : java.io pour la gestion des flux et deux packages de l'API servlet ; javax.servlet.* et javax.servlet.http.

Il faut déclarer une nouvelle classe qui hérite de HttpServlet.

Il faut redéfinir la méthode doGet() pour y insérer le code qui va envoyer dans un flux le code HTML de la page générée.

Exemple (code Java 1.1) :

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyHelloServlet extends HttpServlet {

   public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws IOException, ServletException {

      response.setContentType("text/html");

      PrintWriter out = response.getWriter();

      out.println("<html>");
      out.println("<head>");
      out.println("<title>Bonjour tout le monde</title>");
      out.println("</head>");
      out.println("<body>");
      out.println("<h1>Bonjour tout le monde</h1>");
      out.println("</body>");
      out.println("</html>");
   }
}
```

La méthode getWriter() de l'objet HttpServletResponse renvoie un flux de type PrintWriter dans lequel on peut écrire la réponse.

Si aucun traitement particulier n'est associé à une requête de type POST, il est pratique de demander dans la méthode doPost() d'exécuter la méthode doGet(). Dans ce cas, la servlet est capable de renvoyer une réponse pour les deux types de requête.

Exemple (code Java 1.1) :

```
public void doPost(HttpServletRequest request,HttpServletResponse response)
throws ServletException,IOException {

   this.doGet(request, response);

}
```

79.5. Les informations sur l'environnement d'exécution des servlets

Une servlet est exécutée dans un contexte particulier mis en place par le moteur de servlets.

La servlet peut obtenir des informations sur ce contexte.

La servlet peut aussi obtenir des informations à partir de la requête du client.

79.5.1. Les paramètres d'initialisation

Dès que de la servlet est instanciée, le moteur de servlets appelle sa méthode init() en lui donnant en paramètre un objet de type ServletConfig.

ServletConfig est une interface qui possède deux méthodes permettant de connaître les paramètres d'initialisation :

- String getInitParameter(String) : retourne la valeur du paramètre dont le nom est fourni en paramètre

Exemple :

```
String param;
```

```
public void init(ServletConfig config) {

   param = config.getInitParameter("param");

}
```

- Enumeration getInitParameterNames() : retourne une énumération des paramètres d'initialisation

Exemple (code Java 1.1) :

```
public void init(ServletConfig config) throws ServletException {

   cfg = config;

   System.out.println("Liste des parametres d'initialisation");

   for (Enumeration e=config.getInitParameterNames(); e.hasMoreElements();) {
      System.out.println(e.nextElement());
   }
}
```

La déclaration des paramètres d'initialisation dépend du serveur qui est utilisé.

79.5.2. L'objet ServletContext

La servlet peut obtenir des informations à partir d'un objet ServletContext retourné par la méthode getServletContext() d'un objet ServletConfig.

Il est important de s'assurer que cet objet ServletConfig, obtenu par la méthode init() est soit explicitement sauvegardé soit sauvegardé par l'appel à la méthode init() héritée qui effectue cette sauvegarde.

L'interface ServletContext contient plusieurs méthodes dont les principales sont :

méthode	Rôle	Deprecated
String getMimeType(String)	Retourne le type MIME du fichier en paramètre	
String getServletInfo()	Retourne le nom et le numéro de version du moteur de servlet	
Servlet getServlet(String)	Retourne une servlet à partir de son nom grâce au context	Ne plus utiliser depuis la version 2.1 du jsdk
Enumeration getServletNames()	Retourne une énumération qui contient la liste des servlets relatives au contexte	Ne plus utiliser depuis la version 2.1 du jsdk
void log(Exception, String)	Ecrit les informations fournies en paramètre dans le fichier log du serveur	Utiliser la nouvelle méthode surchargée log()
void log(String)	Idem	
void log (String, Throwable)	Idem	

Exemple : écriture dans le fichier log du serveur

```
public void init(ServletConfig config) throws ServletException {

   ServletContext sc = config.getServletContext();

   sc.log( "Demarrage servlet TestServlet" );

}
```

Le format du fichier log est dépendant du serveur utilisé :

Exemple : résultat avec tomcat

```
Context log path="/examples" :Demarrage servlet TestServlet
```

79.5.3. Les informations contenues dans la requête

De nombreuses informations en provenance du client peuvent être extraites de l'objet ServletRequest passé en paramètre par le serveur (ou de HttpServletRequest qui hérite de ServletRequest).

Les informations les plus utiles sont les paramètres envoyés dans la requête.

L'interface ServletRequest dispose de nombreuses méthodes pour obtenir ces informations :

Méthode	Rôle
int getContentLength()	Renvoie la taille de la requête, 0 si elle est inconnue
String getContentType()	Renvoie le type MIME de la requête, null s'il est inconnu
ServletInputStream getInputStream()	Renvoie un flux qui contient le corps de la requête
Enumeration getParameterNames()	Renvoie une énumération contenant le nom de tous les paramètres
String getProtocol()	Retourne le nom du protocole utilisé par la requête et sa version
BufferedReader getReader()	Renvoie un flux qui contient le corps de la requête
String getRemoteAddr()	Renvoie l'adresse IP du client
String getRemoteHost()	Renvoie le nom de la machine cliente
String getScheme	Renvoie le protocole utilisé par la requête (exemple : http, ftp ...)
String getServerName()	Renvoie le nom du serveur qui a reçu la requête
int getServerPort()	Renvoie le port du serveur qui a reçu la requête

Exemple (code Java 1.1) :

```
package tomcat1;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class InfoServlet extends HttpServlet {

   public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws IOException,ServletException {
      GenererReponse(request, response);

   }

   protected void GenererReponse(HttpServletRequest request, HttpServletResponse reponse)
      throws IOException {

      reponse.setContentType("text/html");

      PrintWriter out =reponse.getWriter();

      out.println("<html>");
      out.println("<body>");
      out.println("<head>");
      out.println("<title>Informations a disposition de la servlet</title>");
      out.println("</head>");
      out.println("<body>");
      out.println("<p>Type mime de la requête :"
        +request.getContentType()+"</p>");
      out.println("<p>Protocole de la requête :"
        +request.getProtocol()+"</p>");
      out.println("<p>Adresse IP du client :"
        +request.getRemoteAddr()+"</p>");
      out.println("<p>Nom du client : "
        +request.getRemoteHost()+"</p>");
      out.println("<p>Nom du serveur qui a reçu la requête :"
        +request.getServerName()+"</p>");
      out.println("<p>Port du serveur qui a reçu la requête :"
        +request.getServerPort()+"</p>");
      out.println("<p>scheme: "+request.getScheme()+"</p>");
      out.println("<p>liste des paramètres </p>");

      for (Enumeration e =request.getParameterNames() ; e.hasMoreElements() ; ) {
         Object p = e.nextElement();
         out.println("<p>&nbsp;&nbsp;nom : "+p+" valeur :"
            +request.getParameter(""+p)+"</p>");
      }

      out.println("</body>");
      out.println("</html>");
   }
}
```

Résultat : avec l'url http://localhost:8080/examples/servlet/tomcat1.InfoServlet?param1=valeur1&param2=valeur2 : Une page html s'affiche contenant :

```
Type mime de la requête : null

Protocole de la requête : HTTP/1.0

Adresse IP du client : 127.0.0.1

Nom du client : localhost

Nom du serveur qui a reçu la requête : localhost

Port du serveur qui a reçu la requête : 8080

scheme : http

liste des paramètres

   nom : param2 valeur :valeur2

   nom : param1 valeur :valeur1
```

79.6. L'utilisation des cookies

Les cookies sont des fichiers contenant des données au format texte. Ils sont créés à l'initiative du serveur et envoyés par le serveur sur le poste client pour leur stockage. Les données contenues dans le cookie sont ensuite renvoyées au serveur à chaque requête.

Les cookies peuvent être utilisés explicitement ou implicitement par exemple lors de l'utilisation d'une session.

Les cookies ne sont pas dangereux car ce sont uniquement des fichiers textes qui ne sont pas exécutés. De plus, les navigateurs posent des limites sur le nombre (en principe 20 cookies pour un même serveur) et la taille des cookies (4ko maximum). Par contre les cookies peuvent contenir des données plus ou moins sensibles. Il est capital de ne stocker dans les cookies que des données qui ne sont pas facilement exploitables par une intervention humaine sur le poste client et en tout cas de ne jamais les utiliser pour stocker des informations sensibles telles qu'un numéro de carte bleue.

79.6.1. La classe Cookie

La classe javax.servlet.http.Cookie encapsule un cookie.

Un cookie est composé d'un nom, d'une valeur et d'attributs.

Pour créer un cookie, il suffit d'instancier un nouvel objet de type Cookie. La classe Cookie ne possède qu'un seul constructeur qui attend deux paramètres de type String : le nom et la valeur associée.

La classe Cookie possède plusieurs getters et setters pour obtenir ou définir des attributs qui sont tous optionnels.

Attribut	Rôle
Comment	Commentaire associé au cookie
Domain	Nom de domaine (partiel ou complet) associé au cookie. Seuls les serveurs contenant ce nom de domaine recevront le cookie
MaxAge	Durée de vie en secondes du cookie. Une fois ce délai expiré, le cookie est détruit sur le poste client par le navigateur. Par défaut la valeur limite la durée de vie du cookie à la durée de vie de l'exécution du navigateur
Name	Nom du cookie
Path	Chemin du cookie. Ce chemin permet de renvoyer le cookie uniquement au serveur dont l'url contient également le chemin. Par défaut, cet attribut contient le chemin de l'url de la servlet. Par exemple, pour que le cookie soit renvoyé à toutes les requêtes du serveur, il suffit d'affecter la valeur "/" à cet attribut
Secure	Booléen qui précise si le cookie ne doit être envoyé que par une connexion SSL
Value	Valeur associée au cookie
Version	Version du protocole utilisé pour gérer le cookie

79.6.2. L'enregistrement et la lecture d'un cookie

Pour envoyer un cookie au browser, il suffit d'utiliser la méthode addCookie() de la classe HttpServletResponse.

Exemple :

```
vCookie monCookie = new Cookie("nom","valeur");
response.addCookie(monCookie);
```

Pour lire un cookie envoyé par le browser, il faut utiliser la méthode getCookies() de la classe HttpServletRequest. Cette méthode renvoie un tableau d'objets Cookie. Les cookies sont renvoyés dans l'en-tête de la requête http. Pour rechercher un cookie particulier, il faut parcourir le tableau et rechercher le cookie à partir de son nom grâce à la méthode getName() de l'objet Cookie.

Exemple :

```
Cookie[] cookies = request.getCookies();
String valeur = "";
for(int i=0;i<cookies.length;i++) {
  if(cookies[i].getName().equals("nom")) {
    valeur=cookies[i].getValue();
  }
}
```

79.7. Le partage d'informations entre plusieurs échanges HTTP

Cette section sera développée dans une version future de ce document

79.8. Packager une application web

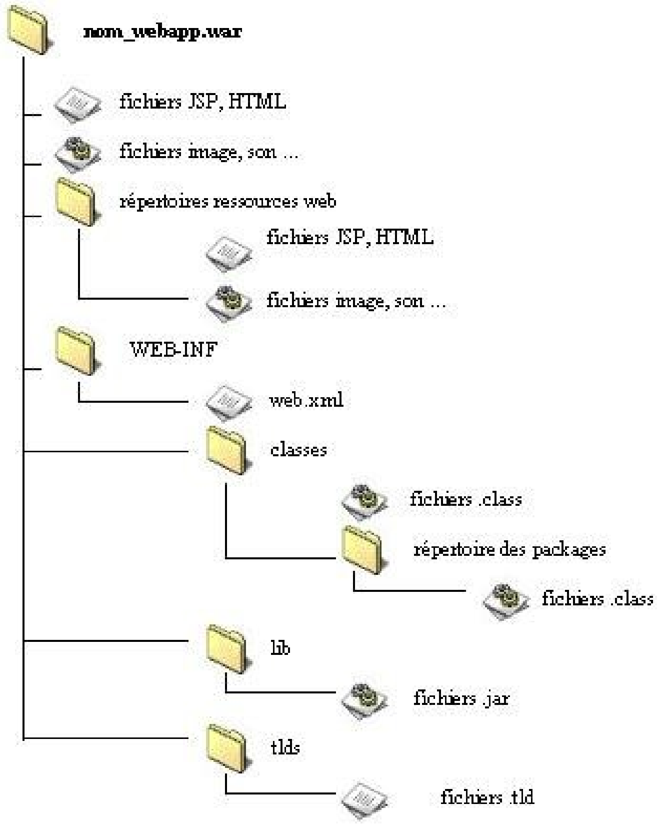
Le format war (Web Application Archive) permet de regrouper en un seul fichier tous les éléments d'une application web que ce soit pour le côté serveur (servlets, JSP, classes java, ...) ou pour le côté client (ressources HTML, images, son ...).

C'est une extension du format jar spécialement dédiée aux applications web qui a été introduite dans les spécifications de la version 2.2 des servlets. C'est un format indépendant de toute plate-forme et exploitable par tous les conteneurs web qui respectent a minima cette version des spécifications.

Le but principal est de simplifier le déploiement d'une application web et d'uniformiser cette action quel que soit le conteneur web utilisé.

79.8.1. La structure d'un fichier .war

Comme les fichiers jar, les fichiers war possèdent une structure particulière qui est incluse dans un fichier compressé de type "zip" possédant comme extension ".war".



Le nom du fichier .war est important car ce nom sera automatiquement associé dans l'url pour l'accès à l'application en concaténant le nom du domaine, un slash et le nom du fichier war. Par exemple, pour un serveur web sur le poste local avec un fichier test.war déployé sur le serveur d'applications, l'url pour accéder à l'application web sera http://localhost/test/.

Le répertoire WEB-INF et le fichier web.xml qu'il contient doivent obligatoirement être présents dans l'archive. Le fichier web.xml est le descripteur de déploiement de l'application web.

Le serveur web peut avoir accès par le serveur d'applications à toutes les ressources contenues dans le fichier .war hormis celles présentes dans le répertoire WEB-INF. Ces dernières ne sont accessibles qu'au serveur d'application.

Le répertoire WEB-INF/classes est automatiquement ajouté par le conteneur au CLASSPATH lors du déploiement de l'application web.

L'archive web peut être créée grâce à l'outil jar fourni avec le JDK ou avec un outil commercial. Avec l'outil jar, il suffit de créer l'arborescence de l'application, de se placer dans le répertoire racine de cette arborescence et d'exécuter la commande :

jar cvf nom_web_app.war .

Toute l'arborescence avec les fichiers qu'elle contient sera incluse dans le fichier nom_web_app.war.

79.8.2. Le fichier web.xml

Le fichier /WEB-INF/web.xml est un fichier au format XML qui est le descripteur de déploiement permettant de configurer : l'application, les servlets, les sessions, les bibliothèques de tags personnalisés, les paramètres de contexte, les types Mimes, les pages par défaut, les ressources externes, la sécurité de l'application et des ressources J2EE.

Le fichier web.xml commence par un prologue et une indication sur la version de la DTD à utiliser. Celle-ci dépend des spécifications de l'API servlet utilisée.

Exemple : servlet 2.2

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
```

Exemple : servlet 2.3

```
  <?xml version="1.0" encoding="ISO-8859-1"?>
  <!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
```

L'élément racine est le tag <web-app>. Cet élément peut avoir plusieurs tags fils dont l'ordre d'utilisation doit respecter celui défini dans la DTD utilisée.

Le tag <icon> permet de préciser une petite et une grande image qui pourront être utilisées par des outils graphiques.

Le tag <display-name> permet de donner un nom pour l'affichage dans les outils.

Le tag <description> permet de fournir un texte de description de l'application web.

Le tag <context-param> permet de fournir un paramètre d'initialisation de l'application. Ce tag peut avoir trois tags fils : <param-name>, <param-value> et <description>. Il doit y en avoir autant que de paramètres d'initialisation. Les valeurs fournies peuvent être retrouvées dans le code de la servlet grâce à la méthode getInitParameter() de l'objet ServletContext.

Le tag <servlet> permet de définir une servlet. Le tag fils <icon> permet de préciser une petite et une grande image pour les outils graphiques. Le tag <servlet-name> permet de donner un nom à la servlet qui sera utilisé pour le mapping avec l'URL par défaut de la servlet. Le tag <display-name> permet de donner un nom d'affichage. Le tag <description> permet de fournir une description de la servlet. Le tag <servlet-class> permet de préciser le nom complètement qualifié de la classe Java dont la servlet sera une instance. Le tag <init-param> permet de préciser un paramètre d'initialisation pour la servlet. Ce tag possède les tags fils <param-name>, <param-value>, <description>. Les valeurs fournies peuvent être retrouvées dans le code de la servlet grâce à la méthode getInitParameter() de la classe ServletConfig. Le tag <load-on-startup> permet de préciser si la servlet doit être instanciée lors de l'initialisation du conteneur. Il est possible de préciser dans le corps de ce tag un numéro de séquence qui permettra d'ordonner la création des servlets.

Exemple : servlet 2.2

```
<servlet>
  <servlet-name>MaServlet</servlet-name>
  <servlet-class>fr.jmdoudoux.dej.servlet.MaServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
  <init-param>
                <param-name>param1</param-name>
                      <param-value>valeur1</param-value>
  </init-param>

</servlet>
```

Le tag <servlet-mapping> permet d'associer la servlet à une URL. Ce tag possède les tags fils <servlet-name> et <servlet-mapping>.

Exemple : servlet 2.2

```
<servlet-mapping>
  <servlet-name>MaServlet</servlet-name>
        <url-pattern>/test</url-pattern>
</servlet-mapping>
```

Le tag <session-config> permet de configurer les sessions. Le tag fils <session-timeout> permet de préciser la durée maximum d'inactivité de la session avant sa destruction. La valeur fournie dans le corps de ce tag est exprimée en minutes.

Exemple : servlet 2.2

```
<session-config>
  <session-timeout>15</session-timeout>
      </session-config>
```

Le tag <mime-mapping> permet d'associer des extensions à un type MIME particulier.

Le tag <welcome-file-list> permet de définir les pages par défaut. Chacun des fichiers est défini grâce au tag fils <welcome-file>

Exemple : servlet 2.2

```
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
  <welcome-file>index.htm</welcome-file>
<welcome-file-list>
```

Le tag <error-page> permet d'associer une page web à un code d'erreur HTTP particulier ou a une exception java particulière. Le code erreur est précisé avec le tag fils <error-code>. L'exception Java est précisée avec le tag fils <exception-type>. La page web est précisée avec le tag fils <location>.

Le tag <tag-lib> permet de définir une bibliothèque de tags personnalisée. Le tag fils <taglib-uri> permet de préciser l'URI de la bibliothèque. Le tag fils <taglib-location> permet de préciser le chemin de la bibliothèque.

Exemple : déclaration de la biliothèque core de JSTL

```
<taglib>
        <taglib-uri>http://java.sun.com/jstl/core</taglib-uri>
        <taglib-location>/WEB-INF/tld/c.tld</taglib-location>
</taglib>
```

79.8.3. Le déploiement d'une application web

Le déploiement d'une archive web dans un serveur d'applications est très facile car il suffit simplement de copier le fichier .war dans le répertoire par défaut dédié aux applications web. Par exemple dans Tomcat, c'est le répertoire webapps. Attention cependant, si chaque conteneur qui respecte les spécifications 1.1 des JSP sait utiliser un fichier .war, son exploitation peut différer légèrement.

Par exemple avec Tomcat, il est possible de travailler directement dans le répertoire webapps avec le contenu de l'archive web décompressée. Cette fonctionnalité est particulièrement intéressante lors de la phase de développement de l'application car il n'est alors pas obligatoire de générer l'archive web pour tester chaque modification. Attention, si l'application est redéployée sous la forme d'une archive .war, il faut obligatoirement supprimer le répertoire qui contient l'ancienne version de l'application.

79.9. L'utilisation de Log4J dans une servlet

Log4J est un framework dont le but est de faciliter la mise en oeuvre de fonctionnalités de logging dans une application. Il est notamment possible de l'utiliser dans une application web. Pour plus de détails sur cette API, consultez la section qui lui est consacrée dans le chapitre «Le logging» de cet ouvrage.

Pour utiliser Log4J dans une application web, il est nécessaire d'initialiser Log4J avant utilisation. Le plus simple est d'écrire une servlet qui va réaliser cette initialisation et qui sera lancée automatiquement au chargement de l'application web.

Dans la méthode init() de la servlet, deux paramètres sont récupérés et utilisés pour :

- définir une variable d'environnement qui sera utilisée par Log4J dans son fichier de configuration pour établir le chemin du fichier journal utilisé
- initialiser Log4J en utilisant un fichier de configuration

Exemple :

```
package fr.jmdoudoux.dej.log4j;

import org.apache.log4j.PropertyConfigurator;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.*;

public class InitServlet extends HttpServlet {
        public void init() {

                String cheminWebApp = getServletContext().getRealPath("/");
                String cheminLogConfig = cheminWebApp + getInitParameter("log4j-fichier-config");
                String cheminLog = cheminWebApp + getInitParameter("log4j-chemin-log");

                File logPathDir = new File( cheminLog );
                System.setProperty( "log.chemin", cheminLog );

                if (cheminLogConfig != null) {
                        PropertyConfigurator.configure(cheminLogConfig);
                }
        }

        public void doGet(HttpServletRequest req, HttpServletResponse res) {
        }
}
```

Dans le fichier web.xml, il faut configurer les servlets utilisées et notamment la servlet définie pour initialiser Log4J. Celle-ci attend au moins deux paramètres :

- log4j-fichier-config : ce paramètre doit avoir comme valeur le chemin relatif du fichier de configuration de Log4J par rapport à la racine de l'application web

- log4j-chemin-log : ce paramètre doit avoir comme valeur le chemin du répertoire qui va contenir les fichiers journaux générés par Log4J par rapport à la racine de l'application web

Exemple : le fichier web.xml

```
<?xml version="1.0"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <servlet>
    <servlet-name>initservlet</servlet-name>
    <servlet-class>fr.jmdoudoux.dej.log4j.InitServlet</servlet-class>
    <init-param>
      <param-name>log4j-fichier-config</param-name>
      <param-value>WEB-INF/classes/log4j.properties</param-value>
    </init-param>
    <init-param>
      <param-name>log4j-chemin-log</param-name>
      <param-value>WEB-INF/log</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet>
    <servlet-name>TestServlet</servlet-name>
    <servlet-class>fr.jmdoudoux.dej.log4j.TestServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>TestServlet</servlet-name>
    <url-pattern>/test</url-pattern>
  </servlet-mapping>
</web-app>
```

Il est important de demander le chargement automatique de la servlet en donnant la valeur 1 à son tag <load-on-startup>.

Il faut définir le fichier de configuration nommé par exemple log4j.properties et le placer dans le répertoire WEB-INF/classes de l'application.

Exemple : le fichier log4j.properties

```
# initialisation de la racine du logger avec le niveau INFO
log4j.rootLogger=INFO, A1

# utilisation d'un fichier pour stocker les informations du journal
log4j.appender.A1=org.apache.log4j.FileAppender
log4j.appender.A1.file=${log.chemin}/application.log

# utilisation du layout de base
log4j.appender.A1.layout=org.apache.log4j.SimpleLayout
```

L'utilisation de Log4J dans une servlet est alors équivalente à celle d'une application standalone.

Exemple : une servlet qui utilise Log4J

```
package fr.jmdoudoux.dej.log4j;

import java.io.IOException;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.log4j.Logger;

public class TestServlet extends HttpServlet {

  private static final Logger logger = Logger.getLogger(TestServlet.class);

  public void init(ServletConfig config) throws ServletException {
    super.init(config);
    logger.info("initialisation de la servlet TestServlet");
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res) {
    StringBuffer sb = new StringBuffer();

    logger.debug("appel doGet de la servlet TestServlet");

    sb.append("<HTML>\n");
    sb.append("<HEAD>\n");
    sb.append("<TITLE>Bonjour</TITLE>\n");
    sb.append("</HEAD>\n");
    sb.append("<BODY>\n");
    sb.append("<H1>Bonjour</H1>\n");
    sb.append("</BODY>\n");
    sb.append("</HTML>");

    res.setContentType("text/html");
    res.setContentLength(sb.length());

    try {
      res.getOutputStream().print(sb.toString());
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Lors de l'exécution de l'application web, le journal est stocké dans le fichier /WEB-INF/log/application.log.

80. Les JSP (Java Server Pages)

Chapitre 80

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Les JSP (Java Server Pages) sont une technologie Java qui permet la génération de pages web dynamiques.

La technologie JSP permet de séparer la présentation sous forme de code HTML et les traitements écrits en Java sous la forme de JavaBeans ou de servlets. Ceci est d'autant plus facile que les JSP définissent une syntaxe particulière permettant d'appeler un bean et d'insérer le résultat de son traitement dans la page HTML dynamiquement.

Les informations fournies dans ce chapitre concernent les spécifications 1.0 et ultérieures des JSP.

Ce chapitre contient plusieurs sections :

- La présentation des JSP
- Les outils nécessaires
- Le code HTML
- Les Tags JSP
- Un exemple très simple
- La gestion des erreurs
- Les bibliothèques de tags personnalisés (custom taglibs)

80.1. La présentation des JSP

Les JSP permettent d'introduire du code Java dans des tags prédéfinis à l'intérieur d'une page HTML. La technologie JSP mélange la puissance de Java côté serveur et la facilité de mise en page d'HTML côté client.

La page officielle de cette technologie est à l'adresse suivante :
https://www.oracle.com/java/technologies/jspt.html.

Une JSP est habituellement constituée :

- de données et de tags HTML
- de tags JSP
- de scriptlets (code Java intégré à la JSP)

Les fichiers JSP possèdent par convention l'extension .jsp.

Concrètement, les JSP sont basées sur les servlets. Au premier appel de la page JSP, le moteur de JSP génère et compile automatiquement la servlet qui permet la génération de la page web. Le code HTML est repris intégralement dans la servlet. Le code Java est inséré dans la servlet.

La servlet générée est compilée et sauvegardée puis elle est exécutée. Les appels suivants de la JSP sont beaucoup plus rapides car la servlet, conservée par le serveur, est directement exécutée.

Il y a plusieurs manières de combiner les technologies JSP, les beans/EJB et les servlets en fonction des besoins pour développer des applications web.

Comme le code de la servlet est généré dynamiquement, les JSP sont relativement difficiles à déboguer.

Cette approche présente cependant plusieurs avantages :

- l'utilisation de Java par les JSP permet une indépendance de la plate-forme d'exécution mais aussi du serveur web utilisé.
- la séparation des traitements et de la présentation : la page web peut être écrite par un designer et les tags Java peuvent être ajoutés ensuite par le développeur. Les traitements peuvent être réalisés par des composants réutilisables (des Java beans).
- les JSP sont basées sur les servlets : tout ce qui est fait par une servlet pour la génération de pages dynamiques peut être fait avec une JSP.

Il existe plusieurs versions des spécifications JSP :

Version	
0.91	Première release
1.0	Juin 1999 : première version finale lié à l'API servlet 2.1
1.1	Décembre 1999 lié à l'API servlet 2.2
1.2	Octobre 2000, JSR 053 lié à l'API servlet 2.3
2.0	Novembre 2003, JSR 152 lié à l'API servlet 2.4
2.1	Mai 2006, JSR 245 lié à l'API servlet 2.5
2.2	Décembre 2009, Maintenance release de la JSR 245 lié à l'API servlet 3.0
2.3	Juin 2013, Maintenance release de la JSR 245 lié à l'API servlet 3.1

80.1.1. Le choix entre JSP et Servlets

Les servlets et les JSP ont de nombreux points communs puisqu'une JSP est finalement convertie en une servlet. Le choix d'utiliser l'une ou l'autre de ces technologies ou les deux doit être fait pour tirer le meilleur parti de leurs avantages.

Dans une servlet, les traitements et la présentation sont regroupés. L'aspect présentation est dans ce cas pénible à développer et à maintenir à cause de l'utilisation répétitive de méthodes pour insérer le code HTML dans le flux de sortie. De plus, une simple petite modification dans le code HTML nécessite la recompilation de la servlet. Avec une JSP, la séparation des traitements et de la présentation rend ceci très facile et automatique.

Il est préférable d'utiliser les JSP pour générer des pages web dynamiques.

L'usage des servlets est obligatoire si celles-ci doivent communiquer directement avec une applet ou une application et non plus avec un serveur web.

80.1.2. Les JSP et les technologies concurrentes

Il existe plusieurs technologies dont le but est similaire aux JSP notamment ASP, PHP et ASP.Net. Chacune de ces technologies possèdent des avantages et des inconvénients dont voici une liste non exhaustive.

	JSP	PHP	ASP	ASP.Net

langage	Java	PHP	VBScript ou JScript	Tous les langages supportés par .Net (C#, VB.Net, Delphi, ...)
mode d'exécution	Compilé en pseudo code (bytecode)	Interprété	Interprété	Compilé en pseudo code (MSIL)
principaux avantages	Repose sur la plate-forme Java dont elle hérite des avantages	Open source Nombreuses bibliothèques et sources d'applications libres disponibles Facile à mettre en oeuvre	Facile à mettre en oeuvre	Repose sur la plate-forme .Net dont elle hérite des avantages Wysiwyg et événementiel Code behind pour séparation affichage / traitements
principaux inconvénients	Débogage assez fastidieux Beaucoup de code à écrire	Débogage assez fastidieux Beaucoup de code à écrire support partiel de la POO en attendant la version 5	Débogage assez fastidieux Beaucoup de code à écrire Fonctionne essentiellement sur plates-formes Windows Pas de POO, objets métiers encapsulés dans des objets COM lourds à mettre en oeuvre	Fonctionne essentiellement sur plates-formes Windows. (Voir le projet Mono pour le support d'autres plates-formes)

80.2. Les outils nécessaires

Dans un premier temps, Sun a fourni un kit de développement pour les JSP : le Java Server Web Development Kit (JSWDK). Ensuite, Sun a chargé le projet Apache de développer l'implémentation de référence du moteur de JSP : le projet Tomcat. Depuis la version 2.2, l'implémentation de référence est le projet Glassfish.

En fonction des versions des API utilisées, il faut choisir un produit différent. Le tableau ci-dessous résume les implémentations de référence en fonction de la version de l'API mise en oeuvre.

Produit	Version	Version de l'API servlet implémentée	Version de l'API JSP implémentée
JSWDK	1.0.1	2.1	1.0
Tomcat	3.2	2.2	1.1
Tomcat	4.0	2.3	1.2
Tomcat	5.0	2.4	2.0
Tomcat	6.0	2.5	2.1
Glassfish	3.0	3.0	2.2
Glassfish	4.0	3.1	2.3

Il est aussi possible d'utiliser n'importe quel conteneur web compatible avec les spécifications de la plate-forme J2EE/Java EE. Une liste non exhaustive est fournie dans le chapitre «Les outils libres et commerciaux».

80.2.1. L'outil JavaServer Web Development Kit (JSWDK) sous Windows

Le JSWDK est proposé sous la forme d'un fichier zip nommé jswdk_1_0_1-win.zip.

Pour l'installer, il suffit de décompresser l'archive dans un répertoire du système. Le serveur se lance en exécutant le fichier startserver.bat.

Pour lancer le serveur :

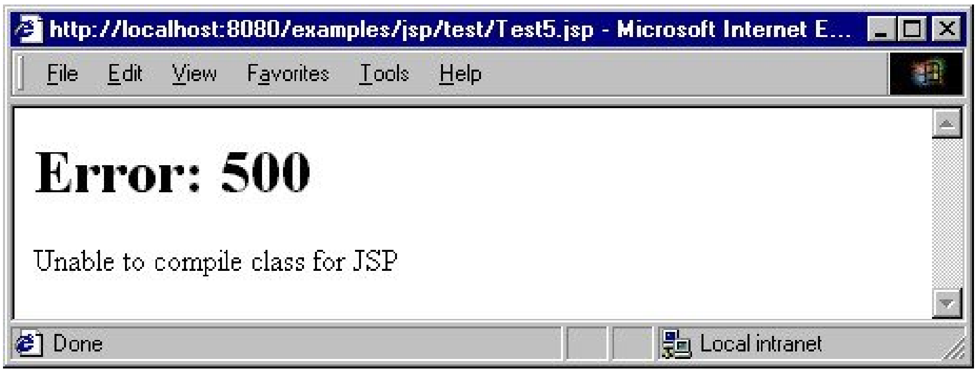
```
C:\jswdk-1.0.1>startserver.bat

Using classpath:.\classes;.\webserver.jar;.\lib\jakarta.jar;.\lib\servlet.jar;.
\lib\jsp.jar;.\lib\jspengine.jar;.\examples\WEB-INF\jsp\beans;.\webpages\WEB-INF
\servlets;.\webpages\WEB-INF\jsp\beans;.\lib\xml.jar;.\lib\moo.jar;\lib\tools.ja
r;C:\jdk1.3\lib\tools.jar;
C:\jswdk-1.0.1>
```

Le serveur s'exécute dans une console en tâche de fond. Cette console permet de voir les messages émis par le serveur.

Exemple : au démarrage

```
JSWDK WebServer Version 1.0.1
Loaded configuration from: file:C:\jswdk-1.0.1\webserver.xml
endpoint created: localhost/127.0.0.1:8080
```

Si la JSP contient une erreur, le serveur envoie une page d'erreur :


http://localhost:8080/examples/jsp/test/Test5.jsp - Microsoft Internet E...
File
Edit
View
Favorites
Tools
Help
Done
Local intranet

Error: 500

Unable to compile class for JSP

Une exception est levée et est affichée dans la fenêtre où le serveur s'exécute :

Exemple :

```
-- Commentaires de la page JSP --
^
1 error
at com.sun.jsp.compiler.Main.compile(Main.java:347)
at com.sun.jsp.runtime.JspLoader.loadJSP(JspLoader.java:135)
at com.sun.jsp.runtime.JspServlet$JspServletWrapper.loadIfNecessary(JspS
ervlet.java:77)
at com.sun.jsp.runtime.JspServlet$JspServletWrapper.service(JspServlet.j
ava:87)
at com.sun.jsp.runtime.JspServlet.serviceJspFile(JspServlet.java:218)
at com.sun.jsp.runtime.JspServlet.service(JspServlet.java:294)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:840)
at com.sun.web.core.ServletWrapper.handleRequest(ServletWrapper.java:155
)
at com.sun.web.core.Context.handleRequest(Context.java:414)
at com.sun.web.server.ConnectionHandler.run(ConnectionHandler.java:139)
HANDLER THREAD PROBLEM: java.io.IOException: Socket Closed
java.io.IOException: Socket Closed
at java.net.PlainSocketImpl.getInputStream(Unknown Source)
at java.net.Socket$1.run(Unknown Source)
at java.security.AccessController.doPrivileged(Native Method)
at java.net.Socket.getInputStream(Unknown Source)
at com.sun.web.server.ConnectionHandler.run(ConnectionHandler.java:161)
```

Le répertoire work contient le code et le bytecode des servlets générées à partir des JSP.

Pour arrêter le serveur, il suffit d'exécuter le script stopserver.bat.

A l'arrêt du serveur, le répertoire work qui contient les servlets générées à partir des JSP est supprimé.

80.2.2. Le serveur Tomcat

La mise en oeuvre et l'utilisation de Tomcat est détaillée dans une section du chapitre «Les servlets».

80.3. Le code HTML

Une grande partie du contenu d'une JSP est constituée de code HTML. D'ailleurs, le plus simple pour écrire une JSP est d'écrire le fichier HTML avec un outil dédié et d'ajouter ensuite les tags JSP pour ce qui concerne les parties dynamiques.

La seule restriction concernant le code HTML concerne l'utilisation des fragments " <% " et " %> " dans la page générée. Dans ce cas, le plus simple est d'utiliser les caractères spéciaux HTML &lt; et &gt;. Sinon l'analyseur syntaxique du moteur de JSP considère que ce sont des tags JSP et renvoie une erreur.

Exemple :

```
<HTML>
<HEAD>
<TITLE>Test</TITLE>
</HEAD>
<BODY>
<p>Plusieurs tags JSP commencent par &lt;% et se finissent par %&gt;</p>
</BODY>
</HTML>
```

80.4. Les Tags JSP

Il existe trois types de tags :

- tags de directives : ils permettent de contrôler la structure de la servlet générée
- tags de scripting: ils permettent d'insérer du code Java dans la servlet
- tags d'actions: ils facilitent l'utilisation de composants

Attention : Les noms des tags sont sensibles à la casse.

80.4.1. Les tags de directives <%@ ... %>

Les directives permettent de préciser des informations globales sur la page JSP. Les spécifications des JSP définissent trois directives :

- page : permet de définir des options de configuration
- include : permet d'inclure des fichiers statiques dans la JSP avant la génération de la servlet
- taglib : permet de définir des tags personnalisés

Leur syntaxe est la suivante :

<%@ directive attribut="*valeur*" ... %>

80.4.1.1. La directive page

Cette directive doit être utilisée dans toutes les pages JSP : elle permet de définir des options qui s'appliquent à toute la JSP.

Elle peut être placée n'importe où dans le source mais il est préférable de la mettre en début de fichier, avant même le tag <HTML>. Elle peut être utilisée plusieurs fois dans une même page mais elle ne doit définir la valeur d'une option qu'une seule fois, sauf pour l'option import.

Les options définies par cette directive sont de la forme option=valeur.

Option	Valeur	Valeur par défaut	Autre valeur possible
autoFlush	Une chaîne	«true»	«false»
buffer	Une chaîne	«8kb»	«none» ou «nnnkb» (nnn indiquant la valeur)
contentType	Une chaîne contenant le type MIME		
errorPage	Une chaîne contenant une URL		
extends	Une classe		
import	Une classe ou un package.*		
info	Une chaîne		
isErrorPage	Une chaîne	«false»	«true»
isThreadSafe	Une chaîne	«true»	«false»
langage	Une chaîne	«java»	
session	Une chaîne	«true»	«false»

Exemple :

```
<%@ page import="java.util.*" %>
<%@ page import="java.util.Vector" %>
<%@ page info="Ma premiere JSP"%>
```

Les options sont :

- autoFlush="**true**|false"

  Cette option indique si le flux en sortie de la servlet doit être vidé quand le tampon est plein. Si la valeur est false, une exception est levée dès que le tampon est plein. On ne peut pas mettre false si la valeur de buffer est none.

- buffer="none|**8kb**|*size*kb"

  Cette option permet de préciser la taille du buffer des données générées contenues par l'objet out de type JspWriter.

- contentType="*mimeType* [; charset=*characterSet*]" | "**text/html;charset=ISO-8859-1**"

  Cette option permet de préciser le type MIME des données générées.

  Cette option est équivalente à <% response.setContentType("mimeType"); %>

- errorPage=*"relativeURL"*

Cette option permet de préciser la JSP appelée au cas où une exception est levée

Si l'URL commence pas un '/', alors l'URL est relative au répertoire principale du serveur web sinon elle est relative au répertoire qui contient la JSP

- extends=*"package.class"*

  Cette option permet de préciser la classe qui sera la super classe de l'objet Java créé à partir de la JSP.

- import= "{ *package.class* | _package.*_ }, ..."

  Cette option permet d'importer des classes contenues dans des packages utilisées dans le code de la JSP. Cette option s'utilise comme l'instruction import dans un code source Java.

  Chaque classe ou package est séparée par une virgule.

  Cette option peut être présente dans plusieurs directives page.

- info="text"

  Cette option permet de préciser un petit descriptif de la JSP. Le texte fourni sera renvoyé par la méthode getServletInfo() de la servlet générée.

- isErrorPage="true|**false**"

  Cette option permet de préciser si la JSP génère une page d'erreur. La valeur true permet d'utiliser l'objet Exception dans la JSP

- isThreadSafe="**true**|false"

  Cette option indique si la servlet générée sera multithread : dans ce cas, une même instance de la servlet peut gérer plusieurs requêtes simultanément. En contrepartie, elle doit gérer correctement les accès concurrents aux ressources. La valeur false impose à la servlet générée d'implémenter l'interface SingleThreadModel.

- language="java"

  Cette option définit le langage utilisé pour écrire le code dans la JSP. La seule valeur autorisée actuellement est «java».

- session="**true**|false"

  Cette option permet de préciser si la JSP est incluse dans une session ou non. La valeur par défaut (true) permet l'utilisation d'un objet session de type HttpSession qui permet de gérer des informations dans une session.

80.4.1.2. La directive include

Cette directive permet d'inclure un fichier dans le code source JSP. Le fichier inclus peut être un fragment de code JSP, HTML ou Java. Le fichier est inclus dans la JSP avant que celle-ci ne soit interprétée par le moteur de JSP.

Ce tag est particulièrement utile pour insérer un élément commun à plusieurs pages tel qu'un en-tête ou un bas de page.

Si le fichier inclus est un fichier HTML, celui-ci ne doit pas contenir de tag <HTML>, </HTML>, <BODY> ou </BODY> qui ferait double emploi avec ceux présents dans le fichier JSP. Ceci impose d'écrire des fichiers HTML particuliers uniquement pour être inclus dans les JSP : ils ne pourront pas être utilisés seuls.

La syntaxe est la suivante :

<%@ include file="chemin relatif du fichier" %>

Si le chemin commence par un '/', alors le chemin est relatif au contexte de l'application, sinon il est relatif au fichier JSP.

Exemple : bonjour.htm

```
<p><table border="1" cellpadding="4" cellspacing="0" width="30%" align=center >
<tr bgcolor="#A6A5C2">
<td align="center">BONJOUR</Td>
</Tr>
</table></p>
```

Exemple : Test1.jsp

```
<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
<BODY>
<p align="center">Test d'inclusion d'un fichier dans la JSP</p>
<%@ include file="bonjour.htm"%>
<p align="center">fin</p>
</BODY>
</HTML>
```

Pour tester cette JSP avec le JSWDK, il suffit de placer ces deux fichiers dans le répertoire jswdk-1.0.1\examples\jsp\test.

Pour visualiser la JSP, il faut saisir l'url http://localhost:8080/examples/jsp/test/Test1.jsp dans un navigateur.

Attention : un changement dans le fichier inclus ne provoque pas une regénération et une compilation de la servlet correspondant à la JSP. Pour insérer un fichier dynamiquement à l'exécution de la servlet il faut utiliser le tag <jsp:include>.

80.4.1.3. La directive taglib

Cette directive permet de déclarer l'utilisation d'une bibliothèque de tags personnalisés. L'utilisation de cette directive est détaillée dans la section consacrée aux bibliothèques de tags personnalisés.

80.4.2. Les tags de scripting

Ces tags permettent d'insérer du code Java qui sera inclus dans la servlet générée à partir de la JSP. Il existe trois tags pour insérer du code Java :

- le tag de déclaration : le code Java est inclus dans le corps de la servlet générée. Ce code peut être la déclaration de variables d'instances ou de classes ou la déclaration de méthodes.
- le tag d'expression : évalue une expression et insère le résultat sous forme de chaîne de caractères dans la page web générée.
- le tag de scriptlets : par défaut, le code Java est inclus dans la méthode service() de la servlet.

Il est possible d'utiliser dans ces tags plusieurs objets définis par les JSP.

80.4.2.1. Le tag de déclarations <%! ... %>

Ce tag permet de déclarer des variables ou des méthodes qui pourront être utilisées dans la JSP. Il ne génère aucun caractère dans le fichier HTML de sortie.

La syntaxe est la suivante :

<%! declarations %>

Exemple :

```
<%! int i = 0; %>
<%! dateDuJour = new java.util.Date(); %>
```

Les variables ainsi déclarées peuvent être utilisées dans les tags d'expressions et de scriptlets.

Il est possible de déclarer plusieurs variables dans le même tag en les séparant avec des caractères ' ;'.

Ce tag permet aussi d'insérer des méthodes dans le corps de la servlet.

Exemple :

```
<HTML>
<HEAD>
<TITLE>Test</TITLE>
</HEAD>
<BODY>
<%!
int minimum(int val1, int val2) {
   if (val1 < val2) return val1;
   else return val2;
}
%>
<% int petit = minimum(5,3);%>
<p>Le plus petit de 5 et 3 est <%= petit %></p>
</BODY>
</HTML>
```

80.4.2.2. Le tag d'expressions <%= ... %>

Le moteur de JSP remplace ce tag par le résultat de l'évaluation de l'expression présente dans le tag.

Ce résultat est toujours converti en une chaîne. Ce tag est un raccourci pour éviter de faire appel à la méthode println() lors de l'insertion de données dynamiques dans le fichier HTML.

La syntaxe est la suivante :

<%= expression %>

Le signe '=' doit être collé au signe '%'.

Attention : il ne faut pas mettre de ' ;' à la fin de l'expression.

Exemple : Insertion de la date dans la page HTML

```
<%@ page import="java.util.*" %>
<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
<BODY>
<p align="center">Date du jour :
<%= new Date() %>
</p>
</BODY>
</HTML>
```

Résultat :

```
Date du jour : Thu Feb 15 11:15:24 CET 2001
```

L'expression est évaluée et convertie en chaîne avec un appel à la méthode toString(). Cette chaîne est insérée dans la page HTML en remplacement du tag. Il est ainsi possible que le résultat soit une partie ou la totalité d'un tag HTML ou même une JSP.

Exemple :

```
<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
<BODY>
<%="<H1>" %>Bonjour<%="</H1>" %>
</BODY>
</HTML>
```

Résultat : code HTML généré

```
<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
<BODY>
<H1>Bonjour</H1>
</BODY>
</HTML>
```

80.4.2.3. Les variables implicites

Les spécifications des JSP définissent plusieurs objets utilisables dans le code dont les plus utiles sont :

Object	Classe	Rôle
out	javax.servlet.jsp.JspWriter	Flux en sortie de la page HTML générée
request	javax.servlet.http.HttpServletRequest	Contient les informations de la requête
response	javax.servlet.http.HttpServletResponse	Contient les informations de la réponse
session	javax.servlet.http.HttpSession	Gère la session

80.4.2.4. Le tag des scriptlets <% ... %>

Ce tag contient du code Java nommé un scriptlet.

La syntaxe est la suivante : <% code Java %>

Exemple :

```
<%@ page import="java.util.Date"%>
<html>
<body>
<%! Date dateDuJour; %>
<% dateDuJour = new Date();%>
Date du jour : <%= dateDuJour %><BR>
</body>
</html>
```

Par défaut, le code inclus dans le tag est inséré dans la méthode service() de la servlet générée à partir de la JSP.

Ce tag ne peut pas contenir autre chose que du code Java : il ne peut pas par exemple contenir de tags HTML ou JSP.

Pour faire cela, il faut fermer le tag du scriptlet, mettre le tag HTML ou JSP puis de nouveau commencer un tag de scriptlet pour continuer le code.

Exemple :

```
<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
<BODY>
<% for (int i=0; i<10; i++) { %>
<%= i %> <br>
<% }%>
</BODY>
</HTML>
```

Résultat : la page HTML générée

```
<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
<BODY>
0 <br>
1 <br>
2 <br>
3 <br>
4 <br>
5 <br>
6 <br>
7 <br>
8 <br>
9 <br>
</BODY>
</HTML>
```

80.4.3. Les tags de commentaires

Il existe deux types de commentaires avec les JSP :

- les commentaires visibles dans le code HTML
- les commentaires invisibles dans le code HTML

80.4.3.1. Les commentaires HTML <!-- ... -->

Ces commentaires sont ceux définis par le format HTML. Ils sont intégralement reproduits dans le fichier HTML généré. Il est possible d'insérer, dans ce tag, un tag JSP de type expression qui sera exécuté.

La syntaxe est la suivante :

<!-- commentaires [<%= expression %>] -->

Exemple :

```
<%@ page import="java.util.*" %>
<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
<BODY>
<!-- Cette page a ete generee le <%= new Date() %> -->
<p>Bonjour</p>
</BODY>
</HTML>
```

Résultat :

```
<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
<BODY>
<!-- Cette page a ete generee le Thu Feb 15 11:44:25 CET 2001 -->
<p>Bonjour</p>
</BODY>
</HTML>
```

Le contenu d'une expression incluse dans des commentaires est dynamique : sa valeur peut changer à chaque génération de la page en fonction de son contenu.

80.4.3.2. Les commentaires cachés <%-- ... --%>

Les commentaires cachés sont utilisés pour documenter la page JSP. Leur contenu est ignoré par le moteur de JSP et ne sont donc pas reproduits dans la page HTML générée.

La syntaxe est la suivante :

<%-- commentaires --%>

Exemple :

```
<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
<BODY>
<%-- Commentaires de la page JSP --%>
<p>Bonjour</p>
</BODY>
</HTML>
```

Résultat :

```
<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
<BODY>
<p>Bonjour</p>
</BODY>
</HTML>
```

Ce tag peut être utile pour éviter l'exécution de code lors de la phase de débogage.

80.4.4. Les tags d'actions

Les tags d'actions permettent de réaliser des traitements couramment utilisés.

80.4.4.1. Le tag <jsp:useBean>

Le tag <jsp:useBean> permet de localiser une instance ou d'instancier un bean pour l'utiliser dans la JSP.

L'utilisation d'un bean dans une JSP est très pratique car il peut encapsuler des traitements complexes et être réutilisable par d'autre JSP ou composants. Le bean peut par exemple assurer l'accès à une base de données. L'utilisation des beans permet de simplifier les traitements inclus dans la JSP.

Lors de l'instanciation d'un bean, on précise la portée du bean. Si le bean demandé est déjà instancié pour la portée précisée alors il n'y pas de nouvelle instance du bean qui est créée mais sa référence est simplement renvoyée : le tag <jsp:useBean> n'instancie donc pas obligatoirement un objet.

Ce tag ne permet pas de traiter directement des EJB.

La syntaxe est la suivante :

<jsp:useBean
id="*beanInstanceNam*e"
scope="**page**|request|session|application"
{ class="*package.class*" |
type="*package.class*" |
class="*package.class*" type="*package.class*" |
beanName="{*package.class* | <%= *expression* %>}" type="*package.class*"
}
{ /> |
> ...
</jsp:useBean>
}

L'attribut id permet de donner un nom à la variable qui va contenir la référence sur le bean.

L'attribut scope permet de définir la portée sur laquelle le bean est défini et utilisable. La valeur de cet attribut détermine la manière dont le tag localise ou instancie le bean. Les valeurs possibles sont :

Valeur	Rôle
page	Le bean est utilisable dans toute la page JSP ainsi que dans les fichiers statiques inclus. C'est la valeur par défaut.
request	le bean est accessible durant la durée de vie de la requête. La méthode getAttribute() de l'objet request permet d'obtenir une référence sur le bean.
session	le bean est utilisable par toutes les JSP qui appartiennent à la même session que la JSP qui a instanciée le bean. Le bean est utilisable tout au long de la session par toutes les pages qui y participent. La JSP qui crée le bean doit avoir l'attribut session = « true » dans sa directive page.
application	le bean est utilisable par toutes les JSP qui appartiennent à la même application que la JSP qui a instanciée le bean. Le bean n'est instancié que lors du rechargement de l'application.

L'attribut class permet d'indiquer la classe du bean.

L'attribut type permet de préciser le type de la variable qui va contenir la référence du bean. La valeur indiquée doit obligatoirement être une super classe du bean ou une interface implémentée par le bean (directement ou par héritage).

L'attribut beanName permet d'instancier le bean grâce à la méthode instanciate() de la classe Beans.

Exemple :

```
<jsp:useBean id="monBean" scope="session" class="test.MonBean" />
```

Dans cet exemple, une instance de MonBean est créée une seule et unique fois lors de la session. Dans la même session, l'appel du tag <jsp:useBean> avec le même bean et la même portée ne feront que renvoyer l'instance créée. Le bean est ainsi accessible durant toute la session.

Le tag <jsp:useBean> recherche si une instance du bean existe avec le nom et la portée précisée. Si elle n'existe pas, alors une instance est créée. S'il y a instanciation du bean, alors les tags <jsp:setProperty> inclus dans le tag sont utilisés pour initialiser les propriétés du bean sinon ils sont ignorés. Les tags inclus entre les tags <jsp:useBean> et </jsp:useBean> ne sont exécutés que si le bean est instancié.

Exemple :

```
<jsp:useBean id="monBean" scope="session" class="test.MonBean" >
<jsp:setProperty name="monBean" property="*" />
</jsp:useBean>
```

Cet exemple a le même effet que le précédent avec une initialisation des propriétés du bean lors de son instanciation.

Exemple complet : TestBean.jsp

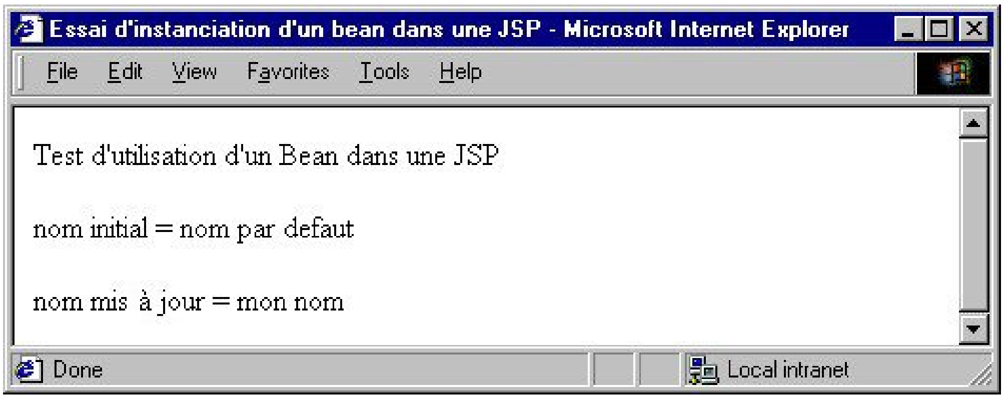
```
<html>
<HEAD>
<TITLE>Essai d'instanciation d'un bean dans une JSP</TITLE>
</HEAD>
<body>
<p>Test d'utilisation d'un Bean dans une JSP </p>
<jsp:useBean id="personne" scope="request" class="test.Personne" />
<p>nom initial = <%=personne.getNom() %></p>
<%personne.setNom("mon nom");%>
<p>nom mis à jour = <%= personne.getNom() %></p>
</body>
</html>
```

Exemple complet : Personne.java

```
package test;

public class Personne {
  private String nom;
  private String prenom;

  public Personne() {
    this.nom = "nom par defaut";
    this.prenom = "prenom par defaut";
  }

  public void setNom (String nom) {
    this.nom = nom;
  }

  public String getNom() {
    return (this.nom);
  }

  public void setPrenom (String prenom) {
    this.prenom = prenom;
  }

  public String getPrenom () {
    return (this.prenom);
  }
}
```

Selon le moteur de JSP utilisé, les fichiers du bean doivent être placés dans un répertoire particulier pour être accessibles par la JSP.

Pour tester cette JSP avec Tomcat, il faut compiler le bean Personne dans le répertoire c:\jakarta-tomcat\webapps\examples\web-inf\classes\test et placer le fichier TestBean.jsp dans le répertoire c:\jakarta-tomcat\webapps\examples\jsp\test.


Essai d'instanciation d'un bean dans une JSP - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Done
Local intranet

Test d'utilisation d'un Bean dans une JSP

nom initial = nom par defaut

nom mis à jour = mon nom

80.4.4.2. Le tag <jsp:setProperty >

Le tag <jsp:setProperty> permet de mettre à jour la valeur d'un ou plusieurs attributs d'un Bean. Le tag utilise le setter (méthode setXXX() où XXX est le nom de la propriété avec la première lettre en majuscule) pour mettre à jour la valeur. Le bean doit exister grâce à un appel au tag <jsp:useBean>.

Il existe trois façons de mettre à jour les propriétés soit à partir des paramètres de la requête soit avec une valeur :

- alimenter automatiquement toutes les propriétés avec les paramètres correspondants de la requête
- alimenter automatiquement une propriété avec le paramètre de la requête correspondant
- alimenter une propriété avec la valeur précisée

La syntaxe est la suivante :

<jsp:setProperty name="*beanInstanceName*"
{ property="*" |
property="*propertyName*" [param=" *parameterName*"] |
property="*propertyName*" value="{*string* | <%= *expression*%>}"
}
/>

L'attribut name doit contenir le nom de la variable qui contient la référence du bean. Cette valeur doit être identique à celle de l'attribut id du tag <jsp:useBean> utilisé pour instancier le bean.

L'attribut property= «*» permet d'alimenter automatiquement les propriétés du bean avec les paramètres correspondants contenus dans la requête. Le nom des propriétés et le nom des paramètres doivent être identiques.

Comme les paramètres de la requête sont toujours fournis sous forme de String, une conversion est réalisée en utilisant la méthode valueOf() du wrapper du type de la propriété.

Exemple :

```
<jsp:setProperty name="monBean" property="*" />
```

L'attribut property="*propertyName*" [param="*parameterName*"] permet de mettre à jour un attribut du bean. Par défaut, l'alimentation est faite automatiquement avec le paramètre correspondant dans la requête. Si le nom de la propriété et du paramètre sont différents, il faut préciser l'attribut property et l'attribut param contenant le nom du paramètre qui va alimenter la propriété du bean.

Exemple :

```
<jsp:setProperty name="monBean" property="nom"/>
```

L'attribut property="*propertyName*" value="{*string* | <%= *expression* %>}" permet d'alimenter la propriété du bean avec une valeur particulière.

Exemple :

```
<jsp:setProperty name="monBean" property="nom" value="toto" />
```

Il n'est pas possible d'utiliser param et value dans le même tag.

Exemple : Cette exemple est identique au précédent

```
<html>
<HEAD>
<TITLE>Essai d'instanciation d'un bean dans une JSP</TITLE>
</HEAD>
<body>
<p>Test d'utilisation d'un Bean dans une JSP </p>
<jsp:useBean id="personne" scope="request" class="test.Personne" />
<p>nom initial = <%= personne.getNom() %></p>
<jsp:setProperty name="personne" property="nom" value="mon nom" />
<p>nom mis à jour = <%= personne.getNom() %></p>
</body>
</html>
```

Ce tag peut être utilisé entre les tags <jsp:useBean> et </jsp:useBean> pour initialiser les propriétés du bean lors de son instanciation.

80.4.4.3. Le tag <jsp:getProperty>

Le tag <jsp:getProperty> permet d'obtenir la valeur d'un attribut d'un Bean. Le tag utilise le getter (méthode getXXX() où XXX est le nom de la propriété avec la première lettre en majuscule) pour obtenir la valeur et l'insérer dans la page HTML générée. Le bean doit exister grâce à un appel au tag <jsp:useBean>.

La syntaxe est la suivante :

<jsp:getProperty name="*beanInstanceName*" property=" *propertyName*" />

L'attribut name indique le nom du bean tel qu'il a été déclaré dans le tag <jsp:useBean>.

L'attribut property indique le nom de la propriété dont on veut la valeur.

Exemple :

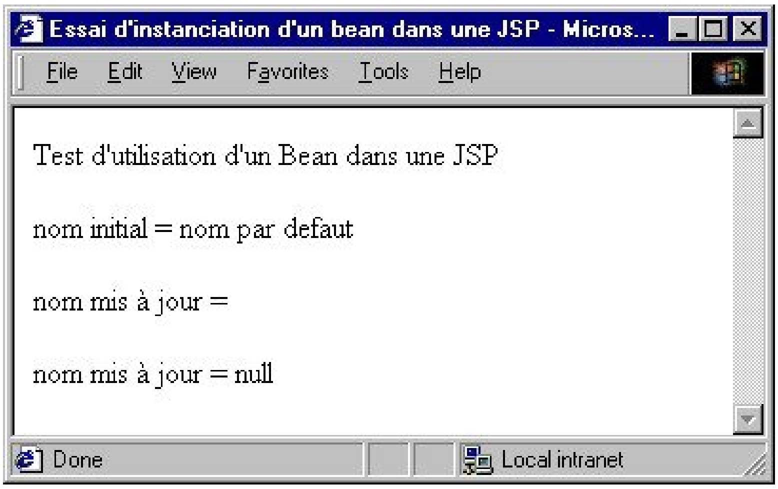
```
<html>
<HEAD>
<TITLE>Essai d'instanciation d'un bean dans une JSP</TITLE>
</HEAD>
<body>
<p>Test d'utilisation d'un Bean dans une JSP </p>
<jsp:useBean id="personne" scope="request" class="test.Personne" />
<p>nom initial = <jsp:getProperty name="personne" property="nom" /></p>
<jsp:setProperty name="personne" property="nom" value="mon nom" />
<p>nom mise à jour = <jsp:getProperty name="personne" property="nom" /></p>
</body>
</html>
```

Attention : ce tag ne permet pas d'obtenir la valeur d'une propriété indexée ni les valeurs d'un attribut d'un EJB.

Remarque : avec Tomcat 3.1, l'utilisation du tag <jsp:getProperty> sur un attribut dont la valeur est null n'affiche rien alors que l'utilisation d'un tag d'expression retourne « null ».

Exemple :

```
<html>
<HEAD>
<TITLE>Essai d'instanciation d'un bean dans une JSP</TITLE>
</HEAD>
<body>
<p>Test d'utilisation d'un Bean dans une JSP </p>
<jsp:useBean id="personne" scope="request" class="test.Personne" />
<p>nom initial = <jsp:getProperty name="personne" property="nom" /></p>
<% personne.setNom(null);%>
<p>nom mis à jour = <jsp:getProperty name="personne" property="nom" /></p>
<p>nom mis à jour = <%= personne.getNom() %></p>
</body>
</html>
```



Test d'utilisation d'un Bean dans une JSP

nom initial = nom par defaut

nom mis à jour =

nom mis à jour = null

80.4.4.4. Le tag de redirection <jsp:forward>

Le tag <jsp:forward> permet de rediriger la requête vers une autre URL pointant vers un fichier HTML, JSP ou une servlet.

Dès que le moteur de JSP rencontre ce tag, il redirige le requête vers l'URL précisée et ignore le reste de la JSP courante. Tout ce qui a été généré par la JSP est perdu.

La syntaxe est la suivante :

<jsp:forward page="{*relativeURL* | <%= *expression* %>}" />
ou
<jsp:forward page="{*relativeURL* | <%= *expression* %>}" >
<jsp:param name="*parameterName*" value="{ *parameterValue* | <%= *expression* %>}" /> +
</jsp:forward>

L'option page doit contenir la valeur de l'URL de la ressource vers laquelle la requête va être redirigée.

Cette URL est absolue si elle commence par un '/' sinon elle est relative à la JSP . Dans le cas d'une URL absolue, c'est le serveur web qui détermine la localisation de la ressource.

Il est possible de passer un ou plusieurs paramètres vers la ressource appelée grâce au tag <jsp :param>.

Exemple : Test8.jsp

```
<html>
<body>
<p>Page initiale appelée</p>
<jsp:forward page="forward.htm"/>
</body>
</html>
```

Exemple : forward.htm

```
<HTML>
<HEAD>
<TITLE>Page HTML</TITLE>
</HEAD>
<BODY>
<p><table border="1" cellpadding="4" cellspacing="0" width="30%" align=center >
<tr bgcolor="#A6A5C2">
<td align="center">Page HTML forwardée</Td>
</Tr>
</table></p>
</BODY>
</HTML>
```

Dans l'exemple, le fichier forward.htm doit être dans le même répertoire que la JSP. Lors de l'appel à la JSP, c'est la page HTML qui est affichée. Le contenu généré par la page JSP n'est pas affiché.

80.4.4.5. Le tag <jsp:include>

Ce tag permet d'inclure dynamiquement le contenu généré par une JSP ou une servlet au moment où la JSP est exécutée. C'est la différence avec la directive include pour laquelle le fichier est inséré dans la JSP avant la génération de la servlet.

La syntaxe est la suivante :

<jsp:include page="*relativeURL" flush="true"* />

L'attribut page permet de préciser l'URL relative de l'élément à insérer.

L'attribut flush permet d'indiquer si le tampon doit être envoyé au client et vidé. Si la valeur de ce paramètre est true, il n'est pas possible d'utiliser certaines fonctionnalités dans la servlet ou la JSP appelée : il n'est pas possible de modifier l'entête de la réponse (header, cookies) ou de faire suivre vers une autre page.

Exemple :

```
<html>
  <body>
    <jsp:include page="bandeau.jsp"/>
    <H1>Bonjour</H1>
    <jsp:include page="pied.jsp"/>
  </body>
</html>
```

Il est possible de fournir des paramètres à la servlet ou à la JSP appelée en utilisant le tag <jsp:param>.

80.4.4.6. Le tag <jsp:plugin>

Ce tag permet la génération du code HTML nécessaire à l'exécution d'une applet en fonction du navigateur : un tag HTML <Object> ou <Embed> est généré en fonction de l'attribut User-Agent de la requête.

Le tag <jsp:plugin> possède trois attributs obligatoires :

Attribut	Rôle
code	permet de préciser le nom de classe
codebase	contient une URL précisant le chemin absolu ou relatif du répertoire contenant la classe ou l'archive
type	les valeurs possibles sont applet ou bean

Il possède aussi plusieurs autres attributs optionnels dont les plus utilisés sont :

Attribut	Rôle
align	permet de préciser l'alignement de l'applet : les valeurs possibles sont bottom, middle ou top
archive	permet de préciser un ensemble de ressources (bibliothèques jar, classes, ...) qui seront automatiquement chargées. Le chemin de ces ressources tient compte de l'attribut codebase
height	précise la hauteur de l'applet en pixel ou en pourcentage
hspace	précise le nombre de pixels insérés à gauche et à droite de l'applet
jreversion	précise la version minimale du jre à utiliser pour faire fonctionner l'applet
name	précise le nom de l'applet
vspace	précise le nombre de pixels insérés en haut et en bas de l'applet
width	précise la longueur de l'applet en pixel ou en pourcentage

Pour fournir un ou plusieurs paramètres, il faut utiliser dans le corps du tag <jsp:plugin> le tag <jsp:params>. Chaque paramètre sera alors défini dans un tag <jsp:param>.

Exemple :

```
<jsp:plugin type="applet" code="MonApplet.class" codebase="applets"
          jreversion="1.1" width="200" height="200" >
  <jsp:params>
     <jsp:param name="couleur" value="eeeeee" />
  </jsp:params>
</jsp:plugin>
```

Le tag <jsp:fallback> dans le corps du tag <jsp:plugin> permet de préciser un message qui sera affiché dans les navigateurs ne supportant pas le tag HTML <Object> ou <Embed>.

80.5. Un exemple très simple

L'exemple de cette section est composé de deux pages.

La premiere page est une page HTML qui demande à l'utilisateur son nom et invoque une url vers une JSP.

Exemple : TestJSPIdent.html

```
<HTML>
<HEAD>
<TITLE>Identification</TITLE>
</HEAD>
<BODY>
<FORM METHOD=POST ACTION="jsp/TestJSPAccueil.jsp">
Entrez votre nom :
<INPUT TYPE=TEXT NAME="nom">
<INPUT TYPE=SUBMIT VALUE="SUBMIT">
</FORM>
</BODY>
</HTML>
```

La page JSP salue l'utilisateur en récupérant son nom.

Exemple : TestJSPAccueil.jsp

```
<HTML>
```

```
<HEAD>
<TITLE>Accueil</TITLE>
</HEAD>
<BODY>
<%
String nom = request.getParameter("nom");
%>
<H2>Bonjour <%= nom %></H2>
</BODY>
</HTML>
```

80.6. La gestion des erreurs

Lors de l'exécution d'une page JSP, des erreurs peuvent survenir. Chaque erreur se traduit par la levée d'une exception. Si cette exception est capturée dans un bloc try/catch de la JSP, celle-ci est traitée. Si l'exception n'est pas capturée dans la page, il y a deux possibilités selon qu'une page d'erreur est associée à la page JSP ou non :

- sans page d'erreur associée, la pile d'exécution de l'exception est affichée
- avec une page d'erreur associée, une redirection est effectuée vers cette JSP

La définition d'une page d'erreur permet de la préciser dans l'attribut errorPage de la directive page des autres JSP de l'application. Si une exception est levée dans les traitements d'une de ces pages, la JSP va automatiquement rediriger l'utilisateur vers la page d'erreur précisée.

La valeur de l'attribut errorPage de la directive page doit contenir l'URL de la page d'erreur. Le plus simple est de définir cette page à la racine de l'application web et de faire précéder le nom de la page par un caractère '/' dans l'url.

Exemple :

```
<%@ page errorPage="/mapagederreur.jsp" %>
```

80.6.1. La définition d'une page d'erreur

Une page d'erreur est une JSP dont l'attribut isErrorPage est égal à true dans la directive page. Une telle page dispose d'un accès à la variable implicite nommée exception de type Throwable qui encapsule l'exception qui a été levée.

Il est possible dans une telle page d'afficher un message d'erreur personnalisé mais aussi d'inclure des traitements liés à la gestion de l'exception : ajouter l'exception dans un journal, envoyer un mail pour son traitement, ...

Exemple :

```
<%@ page language="java" contentType="text/html" %>
%@ page isErrorPage="true" %>
<html>
  <body>
    <h1>Une erreur est survenue lors des traitements</h1>
    <p><%= exception.getMessage() %></p>
  </body>
</html>
```

80.7. Les bibliothèques de tags personnalisés (custom taglibs)

Les bibliothèques de tags (taglibs) ou tags personnalisés (custom tags) permettent de définir ses propres tags basés sur XML, de les regrouper dans une bibliothèque et de les réutiliser dans des JSP. C'est une extension de la technologie JSP apparue à partir de la version 1.1 des spécifications des JSP.

80.7.1. La présentation des tags personnalisés

Un tag personnalisé est un élément du langage JSP défini par un développeur pour des besoins particuliers qui ne sont pas traités en standard par les JSP. Ces dernières permettent de définir ses propres tags qui réaliseront des actions pour générer la réponse.

Le principal but est de favoriser la séparation des rôles entre le développeur Java et le concepteur de pages web. L'idée maitresse est de déporter le code Java contenu dans les scriplets de la JSP dans des classes dédiées et de les appeler dans le code source de la JSP en utilisant des tags particuliers.

Ce concept peut sembler proche de celui des javabeans dont le rôle principal est aussi de définir des composants réutilisables. Les javabeans sont particulièrement adaptés pour stocker et échanger des données entre les composants de l'application web en passant par la session.

Les tags personnalisés sont adaptés pour enlever du code Java inclus dans les JSP et le déporter dans une classe dédiée. Cette classe est physiquement un javabean qui implémente une interface particulière.

La principale différence entre un javabean et un tag personnalisé est que ce dernier tient compte de l'environnement dans lequel il s'exécute (notamment la JSP et le contexte de l'application web) et interagit avec lui.

Pour de plus amples informations sur les bibliothèques de tags personnalisés, il suffit de consulter le site qui leur est consacré :
https://www.oracle.com/java/technologies/java-server-tag-library.html.

Les tags personnalisés possèdent des fonctionnalités intéressantes :

- ils ont un accès aux objets de la JSP notamment l'objet de type HttpResponse. Ils peuvent donc modifier le contenu de la réponse générée par la JSP
- ils peuvent recevoir des paramètres envoyés à partir de la JSP qui les appelle
- ils peuvent avoir un corps qu'ils peuvent manipuler. Par extension de cette fonctionnalité, il est possible d'imbriquer un tag personnalisé dans un autre avec un nombre d'imbrications illimité

Les avantages des bibliothèques de tags personnalisés sont :

- une suppression du code Java dans la JSP remplacé par un tag XML facilement compréhensible ce qui simplifie grandement la JSP
- une API facile à mettre en oeuvre
- une forte et facile réutilisabilité des tags développés
- une maintenance des JSP facilitée

La définition d'une bibliothèque de tags comprend plusieurs entités :

- une classe dite "handler" pour chaque tag qui compose la bibliothèque
- un fichier de description de la bibliothèque

80.7.2. Les handlers de tags

Chaque tag est associé à une classe qui va contenir les traitements à exécuter lors de l'utilisation du tag. Une telle classe est nommée "handler de tag" (tag handler). Pour permettre son appel, une telle classe doit obligatoirement implémenter directement ou indirectement l'interface javax.servlet.jsp.tagext.Tag

L'interface Tag possède une interface fille BodyTag qui doit être utilisée dans le cas où le corps du tag est utilisé.

Pour plus de facilité, l'API JSP propose les classes TagSupport et BodyTagSupport qui implémentent respectivement l'interface Tag et BodyTag. Ces deux classes, contenues dans le package javax.servlet.jsp.tagext, proposent des implémentations par défaut des méthodes de l'interface. Ces deux classes proposent un traitement standard par défaut pour chacune des méthodes de l'interface qu'elles implémentent. Pour définir un handler de tag, il suffit d'hériter de l'une ou l'autre de ces deux classes.

Les méthodes définies dans les interfaces Tag et BodyTag sont appelées, par la servlet issue de la compilation de la JSP, au cours de l'utilisation du tag.

Le cycle de vie général d'un tag est le suivant :

- lors de la rencontre du début du tag, un objet du type du handler est instancié
- plusieurs propriétés sont initialisées (pageContext, parent, ...) en utilisant les setters correspondants
- si le tag contient des attributs, les setters correspondants sont appelés pour alimenter leurs valeurs
- la méthode doStartTag() est appelée
- si la méthode doStartTag() renvoie la valeur EVAL_BODYINCLUDE alors le contenu du corps du tag est évalué
- lors de la rencontre de la fin du tag, appel de la méthode doEndTag()
- si la méthode doEndTag() renvoie la valeur EVAL_PAGE alors l'évaluation de la page se poursuit, si elle renvoie la valeur SKIP_PAGE elle ne se poursuit pas

Toutes ces opérations sont réalisées par le code généré lors de la compilation de la JSP.

Un handler de tag possède un objet qui permet d'avoir un accès aux objets implicites de la JSP. Cet objet est du type javax.servlet.jsp.PageContext

Comme le code contenu dans la classe du tag ne peut être utilisé que dans le contexte particulier du tag, il peut être intéressant de sortir une partie de ce code dans une ou plusieurs classes dédiées qui peuvent être éventuellement des beans.

Pour compiler ces classes, il faut obligatoirement que le jar de l'API servlets (servlets.jar) soit inclus dans la variable CLASSPATH.

80.7.3. L'interface Tag

Cette interface définit les méthodes principales pour la gestion du cycle de vie d'un tag personnalisé qui ne doit pas manipuler le contenu de son corps.

Elle définit plusieurs constantes :

Constante	Rôle
EVAL_BODY_INCLUDE	Continuer avec l'évaluation du corps du tag
EVAL_PAGE	Continuer l'évaluation de la page
SKIP_BODY	Empêcher l'évaluation du corps du tag
SKIP_PAGE	Empêcher l'évaluation du reste de la page

Elle définit aussi plusieurs méthodes :

Méthode	Rôle
int doEndTag()	Traitements à la rencontre du tag de fin
int doStartTag()	Traitements à la rencontre du tag de début
setPageContext(Context)	Sauvegarde du contexte de la page

La méthode doStartTag() est appelée lors de la rencontre du tag d'ouverture et contient les traitements à effectuer dans ce cas. Elle doit renvoyer un entier prédéfini qui indique comment va se poursuivre le traitement du tag :

- EVAL_BODY_INCLUDE : poursuite du traitement avec évaluation du corps du tag
- SKIP_BODY : poursuite du traitement sans évaluation du corps du tag

La méthode doEndTag() est appelée lors de la rencontre du tag de fermeture et contient les traitements à effectuer dans ce cas. Elle doit renvoyer un entier prédéfini qui indique comment va se poursuivre le traitement de la JSP.

- EVAL_PAGE : poursuite du traitement de la JSP
- SKIP_PAGE : ne pas poursuivre le traitement du reste de la JSP

80.7.4. L'accès aux variables implicites de la JSP

Les tags ont accès aux variables implicites de la JSP dans laquelle ils s'exécutent grâce à un objet de type PageContext. La variable pageContext est un objet de ce type qui est initialisé juste après l'instanciation du handler.

Le classe PageContext est une classe abstraite dont l'implémentation des spécifications doit fournir une adaptation concrète.

Cette classe définit plusieurs méthodes :

Méthodes	Rôles
JspWriter getOut()	Permet un accès à la variable out de la JSP
Exception getException()	Permet un accès à la variable exception de la JSP
Object getPage()	Permet un accès à la variable page de la JSP
ServletRequest getRequest()	Permet un accès à la variable request de la JSP
ServletResponse getResponse()	Permet un accès à la variable response de la JSP
ServletConfig getServletConfig()	Permet un accès à l'instance de la variable de type ServletConfig
ServletContext getServletContext()	Permet un accès à l'instance de la variable de type ServletContext
HttpSession getSession()	Permet un accès à la session
Object getAttribute(String)	Renvoie l'objet associé au nom fourni en paramètre dans la portée de la page
setAttribute(String, Object)	Permet de placer dans la portée de la page un objet dont le nom est fourni en paramètre

80.7.5. Les deux types de handlers

Il existe deux types de handlers :

- les handlers de tags sans corps
- les handlers de tags avec corps

80.7.5.1. Les handlers de tags sans corps

Pour définir le handler d'un tag personnalisé sans corps, il suffit de définir une classe qui implémente l'interface Tag ou qui héritent de la classe TagSupport. Il faut définir ou redéfinir les méthodes doStartTag() et endStartTag().

La méthode doStartTag() est appelée à la rencontre du début du tag. Cette méthode doit contenir le code à exécuter dans ce cas et renvoyer la constante SKIP_BODY puisque le tag ne contient pas de corps.

80.7.5.2. Les handlers de tags avec corps

Le cycle de vie d'un tel tag inclut le traitement du corps si la méthode doStartTag() renvoie la valeur EVAL_BODY_TAG.

Dans ce cas, les opérations suivantes sont réalisées :

- la méthode setBodyContent() est appelée
- le contenu du corps est traité
- la méthode doAfterBody() est appelée. Si elle renvoie la valeur EVAL_BODY_TAG, le contenu du corps est de nouveau traité

80.7.6. Les paramètres d'un tag

Un tag peut avoir un ou plusieurs paramètres qui seront transmis à la classe par des attributs. Pour chacun des paramètres, il faut définir des getters et des setters en respectant les règles et conventions des Java beans. Il est impératif de définir un champ, un setter et éventuellement un accesseur pour chaque attribut.

La JSP utilisera le setter pour fournir à l'objet la valeur de l'attribut.

Au moment de la génération de la servlet par le moteur de JSP, celui-ci vérifie par introspection la présence d'un setter pour l'attribut concerné.

80.7.7. La définition du fichier de description de la bibliothèque de tags (TLD)

Le fichier de description de la bibliothèque de tags (tag library descriptor file) est un fichier au format XML qui décrit une bibliothèque de tags. Les informations qu'il contient concernent la bibliothèque de tags elle-même et aussi chacun des tags qui la compose.

Ce fichier est utilisé par le conteneur Web lors de la compilation de la JSP pour remplacer le tag par du code Java.

Ce fichier doit toujours avoir pour extension .tld. Il doit être placé dans le répertoire web-inf du fichier war ou dans un de ses sous-répertoires. Le plus pratique est de tous les regrouper dans un répertoire nommé par exemple tags ou tld.

Comme tout bon fichier XML, le fichier TLD commence par un prologue :

Exemple :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
  PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
  "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
```

La DTD précisée doit correspondre à la version de l'API JSP utilisée. L'exemple précédent concernait la version 1.1, l'exemple suivant concerne la version 1.2

Exemple :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
  PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
  "http://java.sun.com/dtds/web-jsptaglibrary_1_2.dtd">
```

Le tag racine du document XML est le tag <taglib>.

Ce tag peut contenir plusieurs tags qui définissent les caractéristiques générales de la bibliothèque. Les tags suivants sont définis dans les spécifications 1.2 :

Nom	Rôle
tlib-version	version de la bibliothèque
jsp-version	version des spécifications JSP utilisées
short-name	nom court de la bibliothèque (optionnel)
uri	URI qui identifie de façon unique la bibliothèque : cette URI n'a pas besoin d'exister réellement
display-name	nom de la bibliothèque
small-icon	(optionnel)
large-icon	(optionnel)
description	description de la bibliothèque
validator	(optionnel)
listener	(optionnel)
tag	il en faut autant que de tags qui composent la bibliothèque

Pour chaque tag personnalisé défini dans la bibliothèque, il faut un tag <tag>. Ce tag permet de définir les caractéristiques d'un tag de la bibliothèque.

Ce tag peut contenir les tags suivants :

Nom	Rôle
name	nom du tag : il doit être unique dans la bibliothèque
tag-class	nom entièrement qualifié de la classe qui contient le handler du tag
tei-class	nom qualifié d'une classe fille de la classe javax.servlet.jsp.tagext.TagExtraInfo (optionnel)
body-content	type du corps du tag. Les valeurs possibles sont : • JSP : le corps du tag contient des tags JSP qui doivent être interprétés • tagdependent : l'interprétation du contenu du corps est faite par le tag • empty : le corps doit obligatoirement être vide La valeur par défaut est JSP
display-name	nom court du tag
small-icon	nom relatif à la bibliothèque d'un fichier gif ou jpeg contenant une icône. (optionnel)
large-icon	nom relatif à la bibliothèque d'un fichier gif ou jpeg contenant une icône. (optionnel)
description	description du tag (optionnel)
variable	(optionnel)
attribute	il en faut autant que d'attributs possédés par le tag (optionnel)
example	un exemple de l'utilisation du tag (optionnel)

Pour chaque attribut du tag personnalisé, il faut utiliser un tag <attribute>. Ce tag décrit un attribut d'un tag et peut contenir les tags suivants :

Nom	Description
name	nom de l'attribut
required	booléen qui indique la présence obligatoire de l'attribut
rtexprvalue	

	booléen qui indique si la page doit évaluer l'expression lors de l'exécution. Il faut donc mettre la valeur true si la valeur de l'attribut est fournie avec un tag JSP d'expression <%= %>

Le tag <Variable> contient les tags suivants :

Nom	Rôle
name-given	
name-from-attribut	
variable-class	nom de la classe de la valeur de l'attribut. Par défaut java.lang.String
declare	par défaut : True
scope	visibilité de l'attribut. Les valeurs possibles sont : • AT_BEGIN • NESTED • AT_END Par défaut : NESTED (optionnel)
description	description de l'attribut (optionnel)

Chaque bibliothèque doit être définie avec un fichier de description au format xml possédant une extension .tld. Le contenu de ce fichier doit pouvoir être validé avec une DTD fournie par Sun.

Ce fichier est habituellement stocké dans le répertoire web-inf de l'application web ou un de ses sous-répertoires.

Exemple :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
  <!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
  "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>testtaglib</shortname>
  <uri>http://fr.jmdoudoux.dej.taglib</uri>
  <info>Bibliotheque de test des taglibs</info>

  <tag>
  <name>testtaglib1</name>
  <tagclass>fr.jmdoudoux.dej.taglib.TestTaglib1</tagclass>
  <info>Tag qui affiche bonjour</info>
  </tag>
</taglib>
```

80.7.8. L'utilisation d'une bibliothèque de tags

Pour utiliser une bibliothèque de tags, il y a des actions à réaliser au niveau du code source de la JSP et au niveau de conteneur d'applications web pour la déployer.

80.7.8.1. L'utilisation dans le code source d'une JSP

Chaque bibliothèque utilisée dans une JSP doit être déclarée avant son utilisation en utilisant la directive taglib. Le plus simple est d'effectuer ces déclarations tout au début du code de la JSP.

Cette directive possède deux attributs :

- uri : l'URI de la bibliothèque telle que définie dans le fichier de description
- prefix : un préfixe qui servira d'espace de noms pour les tags de la bibliothèque dans la JSP

Exemple :

```
<%@ taglib uri="/WEB-INF/tld/testtaglib.tld" prefix="maTagLib" %>
```

L'attribut uri permet de donner une identité au fichier de description de la bibliothèque de tags (TLD). La valeur fournie peut être :

- directe (par exemple le nom du fichier avec son chemin relatif)

Exemple :

```
<%@ taglib uri="/WEB-INF/tld/testtaglib.tld" prefix="maTagLib" %>
```

- ou indirecte (concordance avec un nom logique défini dans un tag taglib du descripteur de déploiement de l'application web)

Exemple :

```
<%@ taglib uri= "/maTaglib" prefix= "maTagbib" %>
```

Dans ce dernier cas, il faut ajouter pour chaque bibliothèque un tag <taglib> dans le fichier de description de déploiement de l'application/WEB-INF/web.xml

Exemple :

```
  <taglib>
    <taglib-uri>/maTagLibTest</taglib-uri>
    <taglib-location>/WEB-INF/tld/testtaglib.tld</taglib-location>
  </taglib>
```

L'appel d'un tag se fait en utilisant un tag dont le nom a la forme suivante : prefix:tag

Le préfix est celui défini dans la directive taglib.

Exemple : un tag sans corps

```
<maTagLib:testtaglib1/>
```

Exemple : un tag avec corps

```
<prefix:tag>
   ...
</prefix:tag>
```

Le corps peut contenir du code HTML, du code JSP ou d'autre tag personnalisé.

Le tag peut avoir des attributs si ceux-ci ont été définis. La syntaxe pour les utiliser respecte la norme XML.

Exemple : un tag avec un paramètre constant

```
<prefix:tag attribut="valeur"/>
```

La valeur de cet attribut peut être une donnée dynamiquement évaluée lors de l'exécution :

Exemple : un tag avec un paramètre

```
<prefix:tag attribut="<%= uneVariable%>"/>
```

80.7.8.2. Le déploiement d'une bibliothèque

Au moment de la compilation de la JSP en servlet, le conteneur transforme chaque tag en un appel à un objet du type de la classe associée au tag.

Il y a deux types d'éléments auxquels le conteneur d'applications web doit pouvoir accéder :

- le fichier de description de la bibliothèque
- les classes des handlers de tags

Les classes des handlers de tags peuvent être stockées à deux endroits dans le fichier war selon leur format :

- s'ils sont packagés sous forme de fichiers jar alors ils doivent être placés dans le répertoire /WEB-INF/lib
- s'ils ne sont pas packagés alors ils doivent être placés dans le répertoire /WEB-INF/classes

80.7.9. Le déploiement et les tests dans Tomcat

Tomcat étant l'implémentation de référence pour les technologies servlets et JSP, il est pratique d'effectuer des tests avec cet outil.

La version de Tomcat utilisée dans cette section est la 3.2.1.

Le déploiement se fait en deux étapes :

- la copie des fichiers
- l'enregistrement de la bibliothèque

Les classes compilées doivent être copiées dans le répertoire WEB-INF/classes de la webapp si elles ne sont pas packagées dans une archive jar, sinon le ou les fichiers .jar doivent être copiés dans le répertoire WEB-INF/lib.

Le fichier .tld doit être copié dans le répertoire WEB-INF ou dans un de ses sous-répertoires.

Il faut ensuite enregistrer la bibliothèque dans le fichier de configuration web.xml contenu dans le répertoire web-inf du répertoire de l'application web.

Il faut ajouter dans ce fichier, un tag <taglib> pour chaque bibliothèque utilisée par l'application web. Ce tag contient deux informations :

- l'URI de la bibliothèque contenue dans le tag taglib-uri. Cette URI doit être identique à celle définie dans le fichier de description de la bibliothèque
- la localisation du fichier de description

Exemple :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">
<web-app>
  <welcome-file-list id="ListePageDAccueil">
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

  <taglib>
    <taglib-uri>/maTagLibTest</taglib-uri>
    <taglib-location>/WEB-INF/tld/testtaglib.tld</taglib-location>
  </taglib>
</web-app>
```

Il ne reste plus qu'à lancer Tomcat si ce n'est pas encore fait et à saisir l'url de la page contenant l'appel au tag personnalisé.

80.7.10. Les bibliothèques de tags existantes

Il existe de nombreuses bibliothèques de tags libres ou commerciales disponibles sur le marché. Cette section va tenter de présenter quelques-unes des plus connues et des plus utilisées du monde libre. Cette liste n'est pas exhaustive.

80.7.10.1. Struts

Struts est un framework pour la réalisation d'applications web reposant sur le modèle MVC 2.

Pour la partie vue, Struts utilise les JSP et propose en plus plusieurs bibliothèques de tags pour faciliter le développement de cette partie présentation. Struts possède quatre grandes bibliothèques :

- formulaire HMTL
- modèles (templates)
- Javabeans (bean)
- traitements logiques (logic)

Le site web de Struts se trouve à l'url : https://struts.apache.org/index.html.

Ce framework est détaillée dans le chapitre «Struts».

80.7.10.2. JSP Standard Tag Library (JSTL)

JSP Standard Tag Library (JSTL) est une spécification issue du travail du JCP sous la JSR numéro 52. Le chapitre «JSTL (Java server page Standard Tag Library)» fournit plus de détails sur cette spécification.

80.7.10.3. Apache Taglibs (Jakarta Taglibs)

Apache Taglibs est un ensemble de taglibs : la plupart a été déclarée deprecated notamment à cause de la standardisation de la JSTL.

Il propose en particulier, la bibliothèque Apache Standard Tag Library qui est une implémentation des versions 1.0, 1.1 et 1.2 de la spécification JSTL.

Le site officiel est à l'url : https://tomcat.apache.org/taglibs/

81. JSTL (Java server page Standard Tag Library)

Chapitre 81

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

JSTL est l'acronyme de Java server page Standard Tag Library. C'est un ensemble de tags personnalisés développé sous la JSR 052 qui propose des fonctionnalités souvent rencontrées dans les JSP :

- Tag de structure (itération, conditionnement ...)
- Internationalisation
- Exécution de requêtes SQL
- Utilisation de documents XML

JSTL nécessite un conteneur d'applications web qui implémente l'API servlet 2.3 et l'API JSP 1.2. L'implémentation de référence (JSTL-RI) de cette spécification est développée par le projet Taglibs du groupe Apache sous le nom " Standard ".

Il est possible de télécharger cette implémentation de référence à l'URL : https://tomcat.apache.org/taglibs/

JSTL est aussi inclus dans le JWSDP (Java Web Services Developer Pack), ce qui facilite son installation et son utilisation. Les exemples de cette section ont été réalisés avec le JWSDP 1.001

JSTL possède quatre bibliothèques de tags :

Rôle	TLD	Uri
Fonctions de base	c.tld	http://java.sun.com/jstl/core
Traitements XML	x.tld	http://java.sun.com/jstl/xml
Internationalisation	fmt.tld	http://java.sun.com/jstl/fmt
Traitements SQL	sql.tld	http://java.sun.com/jstl/sql

JSTL propose un langage nommé EL (Expression Langage) qui permet de faire facilement référence à des objets Java accessibles dans les différents contextes de la JSP.

La bibliothèque de tags JSTL est livrée en deux versions :

- JSTL-RT : les expressions pour désigner des variables utilisant la syntaxe JSP classique
- JSTL-EL : les expressions pour désigner des variables utilisant le langage EL

Pour plus informations, il est possible de consulter les spécifications à l'url suivante :
https://jcp.org/aboutJava/communityprocess/final/jsr052/

Ce chapitre contient plusieurs sections :

- ♦ Un exemple simple
- ♦ Le langage EL (Expression Language)
- ♦ La bibliothèque Core

- La bibliothèque XML
- La bibliothèque I18n
- La bibliothèque Database

81.1. Un exemple simple

Pour commencer, voici un exemple et sa mise en oeuvre détaillée. L'application web d'exemple se nomme test. Il faut créer un répertoire test dans le répertoire webapps de tomcat.

Pour utiliser JSTL, il faut copier les fichiers jstl.jar et standard.jar dans le répertoire WEB-INF/lib de l'application web.

Il faut copier les fichiers .tld dans le répertoire WEB-INF ou un de ses sous-répertoires. Dans la suite de l'exemple, ces fichiers ont été placés dans le répertoire /WEB-INF/tld.

Il faut ensuite déclarer les bibliothèques à utiliser dans le fichier web.xml du répertoire WEB-INF comme pour toute bibliothèque de tags personnalisés.

Exemple : pour la bibliothèque Core

```
<taglib>
    <taglib-uri>http://java.sun.com/jstl/core</taglib-uri>
    <taglib-location>/WEB-INF/tld/c.tld</taglib-location>
</taglib>
```

L'arborescence des fichiers est la suivante :

Exemple :

```
webapps
      test
            WEB-INF
                  lib
                        jstl.jar
                        standard.jar
                  tld
                        c.tld
                  web.xml
            test.jsp
```

Pour pouvoir utiliser une bibliothèque personnalisée, il faut utiliser la directive taglib :

Exemple :

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
```

Voici les codes source des différents fichiers de l'application web :

Exemple : fichier test.jsp

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
  <head>
    <title>Exemple</title>
  </head>

  <body>
      <c:out value="Bonjour" /><br/>
  </body>
</html>
```

Exemple : le fichier WEB-INF/web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app23.dtd">

<web-app>
  <taglib>
    <taglib-uri>http://java.sun.com/jstl/core</taglib-uri>
    <taglib-location>/WEB-INF/tld/c.tld</taglib-location>
  </taglib>
</web-app>
```

Pour tester l'application, il suffit de lancer Tomcat et de saisir l'url localhost:8080/test/test.jsp dans un browser.

81.2. Le langage EL (Expression Language)

JSTL propose un langage particulier constitué d'expressions qui permettent d'utiliser et de faire référence à des objets Java accessibles dans les différents contextes de la page JSP. Le but est de fournir un moyen simple d'accéder aux données nécessaires à une JSP.

La syntaxe de base est ${xxx} où xxx est le nom d'une variable d'un objet Java défini dans un contexte particulier. La définition dans un contexte permet de définir la portée de la variable (page, requête, session ou application).

EL permet facilement de s'affranchir de la syntaxe de Java pour obtenir une variable.

Exemple : accéder à l'attribut nom d'un objet personne situé dans la session avec Java

```
<%= session.getAttribute("personne").getNom() %>
```

Exemple : accéder à l'attribut nom d'un objet personne situé dans la session avec EL

```
${sessionScope.personne.nom}
```

EL possède par défaut les variables suivantes :

Variable	Rôle
PageScope	variable contenue dans la portée de la page (PageContext)
RequestScope	variable contenue dans la portée de la requête (HttpServletRequest)
SessionScope	variable contenue dans la portée de la session (HttpSession)
ApplicationScope	variable contenue dans la portée de l'application (ServletContext)
Param	paramètre de la requête http
ParamValues	paramètres de la requête sous la forme d'une collection
Header	en-tête de la requête
HeaderValues	en-têtes de la requête sous la forme d'une collection
InitParam	paramètre d'initialisation
Cookie	cookie
PageContext	objet PageContext de la page

EL propose aussi différents opérateurs :

Operateur	Rôle	Exemple
.	Obtenir une propriété d'un objet	${param.nom}
[]	Obtenir une propriété par son nom ou son indice	${param[" nom "]} ${row[1]}
Empty	Teste si un objet est null ou vide si c'est une chaîne de caractère. Renvoie un booléen	${empty param.nom}
== eq	teste l'égalité de deux objets	
!= ne	teste l'inégalité de deux objets	
< lt	test strictement inférieur	
> gt	test strictement supérieur	
<= le	test inférieur ou égal	
>= ge	test supérieur ou égal	
+	Addition	
-	Soustraction	
*	Multiplication	
/ div	Division	
% mod	Modulo	
&& and		
|| or		
! not	Négation d'une valeur	

EL ne permet pas l'accès aux variables locales. Pour pouvoir accéder à de telles variables, il faut obligatoirement en créer une copie dans une des portées particulières : page, request, session ou application

Exemple :

```
<%
  int valeur = 101;
%>
  valeur = <c:out value="${valeur}" /><BR/>
```

Résultat :

```
valeur =
```

Exemple : avec la variable copiée dans le contexte de la page

```
<%
```

```
  int valeur = 101;
  pageContext.setAttribute("valeur", new Integer(valeur));
%>
  valeur = <c:out value="${valeur}" /><BR/>
```

Résultat :

```
valeur = 101
```

81.3. La bibliothèque Core

Elle propose les tags suivants répartis dans trois catégories :

Catégorie	Tag
Utilisation de EL	set out remove catch
Gestion du flux (condition et itération)	if choose forEach forTokens
Gestion des URL	import url redirect

Pour utiliser cette bibliothèque, il faut la déclarer dans le fichier web.xml du répertoire WEB-INF de l'application web.

Exemple :

```
<taglib>
    <taglib-uri>http://java.sun.com/jstl/core</taglib-uri>
    <taglib-location>/WEB-INF/tld/c.tld</taglib-location>
</taglib>
```

Pour chaque JSP qui utilise un ou plusieurs tags, la bibliothèque doit être déclarée avec une directive taglib

Exemple :

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
```

81.3.1. Le tag set

Le tag set permet de stocker une variable dans une portée particulière (page, requête, session ou application).

Il possède plusieurs attributs :

Attribut	Rôle
value	valeur à stocker
target	nom de la variable contenant un bean dont la propriété doit être modifiée
property	nom de la propriété à modifier
var	nom de la variable qui va stocker la valeur

scope	portée de la variable qui va stocker la valeur

Exemple :

```
    <c:set var="maVariable1" value="valeur1" scope="page" />
    <c:set var="maVariable2" value="valeur2" scope="request" />
    <c:set var="maVariable3" value="valeur3" scope="session" />
    <c:set var="maVariable4" value="valeur4" scope="application" />
```

La valeur peut être déterminée dynamiquement.

Exemple :

```
    <c:set var="maVariable" value="${param.id}" scope="page" />
```

L'attribut target avec l'attribut property permettent de modifier la valeur d'une propriété (précisée avec l'attribut property) d'un objet (précisé avec l'attribut target).

La valeur de la variable peut être précisée dans le corps du tag plutôt que d'utiliser l'attribut value.

Exemple :

```
<c:set var="maVariable" scope="page">Valeur de ma variable</c:set>
```

81.3.2. Le tag out

Le tag out permet d'envoyer dans le flux de sortie de la JSP le résultat de l'évaluation de l'expression fournie dans le paramètre " value ". Ce tag est équivalent au tag d'expression <%= ... %> de JSP.

Il possède plusieurs attributs :

Attribut	Rôle
value	valeur à afficher (obligatoire)
default	définir une valeur par défaut si la valeur est null
escapeXml	booléen qui précise si les caractères particuliers (< > & ...) doivent être convertis en leurs équivalents HTML (&lt; &gt; &amp ; ...)

Exemple :

```
    <c:out value='${pageScope.maVariable1}' />
    <c:out value='${requestScope.maVariable2}' />
    <c:out value='${sessionScope.maVariable 3}' />
    <c:out value='${applicationScope.maVariable 4}' />
```

Il n'est pas obligatoire de préciser la portée dans laquelle la variable est stockée : dans ce cas, la variable est recherchée prioritairement dans la page, la requête, la session et enfin l'application.

L'attribut default permet de définir une valeur par défaut si le résultat de l'évaluation de la valeur est null. Si la valeur est null et que l'attribut default n'est pas utilisé alors c'est une chaîne vide qui est envoyée dans le flux de sortie.

Exemple :

```
   <c:out value="${personne.nom}" default="Inconnu" />
```

Le tag out est particulièrement utile pour générer le code dans un formulaire en remplaçant avantageusement les scriplets.

Exemple :

```
<input type="text" name="nom" value="<c:out value="${param.nom}"/>" />
```

81.3.3. Le tag remove

Le tag remove permet de supprimer une variable d'une portée particulière.

Il possède plusieurs attributs :

Attribut	Rôle
var	nom de la variable à supprimer (obligatoire)
scope	portée de la variable

Exemple :

```
    <c:remove var="maVariable1" scope="page" />
    <c:remove var="maVariable2" scope="request" />
    <c:remove var="maVariable3" scope="session" />
    <c:remove var="maVariable4" scope="application" />
```

81.3.4. Le tag catch

Ce tag permet de capturer des exceptions qui sont levées lors de l'exécution du code inclus dans son corps.

Il possède un attribut :

Attribut	Rôle
var	nom d'une variable qui va contenir des informations sur l'anomalie

Si l'attribut var n'est pas utilisé, alors toutes les exceptions levées lors de l'exécution du corps du tag sont ignorées.

Exemple : code non protégé

```
<c:set var="valeur" value="abc" />
<fmt:parseNumber var="valeurInt" value="${valeur}"/>
```

Résultat : une exception est lévée

```
javax.servlet.ServletException: In <parseNumber>, value attribute can not be parsed: "abc"
        at org.apache.jasper.runtime.PageContextImpl.handlePageException(PageContextImpl.java:
471)
        at org.apache.jsp.test$jsp.jspService(test$jsp.java:1187)
        at org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:107)
```

L'utilisation du tag catch peut empêcher le plantage de l'application.

Exemple :

```
<c:set var="valeur" value="abc" />
<c:catch var="erreur">
   <fmt:parseNumber var="valeurInt" value="${valeur}"/>
</c:catch>
<c:if test="${not empty erreur}">
```

```
   la valeur n'est pas numerique
</c:if>
```

Résultat :

```
la valeur n'est pas numerique
```

L'objet désigné par l'attribut var du tag catch possède une propriété message qui contient le message d'erreur

Exemple :

```
<c:set var="valeur" value="abc" />
<c:catch var="erreur">
   <fmt:parseNumber var="valeurInt" value="${valeur}"/>
</c:catch>
<c:if test="${not empty erreur}">
 <c:out value="${erreur.message}"/>
</c:if>
```

Résultat :

```
In &lt;parseNumber&gt;, value attribute can not be parsed: "abc"
```

Le souci avec ce tag est qu'il n'est pas possible de savoir quelle exception a été levée.

81.3.5. Le tag if

Ce tag permet d'évaluer le contenu de son corps si la condition qui lui est fournie est vraie.

Il possède plusieurs attributs :

Attribut	Rôle
test	condition à évaluer
var	nom de la variable qui contiendra le résultat de l'évaluation
scope	portée de la variable qui contiendra le résultat

Exemple :

```
<c:if test="${empty personne.nom}" >Inconnu</c:if>
```

Le tag peut ne pas avoir de corps s'il est simplement utilisé pour stocker le résultat de l'évaluation de la condition dans une variable.

Exemple :

```
<c:if test="${empty personne.nom}" var="resultat" />
```

Le tag if est particulièrement utile pour générer le code dans un formulaire en remplaçant avantageusement les scriplets.

Exemple : selection de la bonne occurrence dont la valeur est fournie en paramètre de la requête

```
<FORM NAME="form1" METHOD="post" ACTION="">
  <SELECT NAME="select">
    <OPTION VALUE="choix1" <c:if test="${param.select == 'choix1'}" >selected</c:if> >
      choix 1</OPTION>
    <OPTION VALUE="choix2" <c:if test="${param.select == 'choix2'}" >selected</c:if> >
      choix 2</OPTION>
    <OPTION VALUE="choix3" <c:if test="${param.select == 'choix3'}" >selected</c:if> >
      choix 3</OPTION>
  </SELECT>
</FORM>
```

Pour tester le code, il faut fournir en paramètre dans l'url select=choix2

Exemple :

```
http://localhost:8080/test/test.jsp?select=choix2
```

81.3.6. Le tag choose

Ce tag permet de traiter différents cas mutuellement exclusifs dans un même tag. Le tag choose ne possède pas d'attribut. Il doit cependant posséder un ou plusieurs tags fils « when ».

Le tag when possède l'attribut test qui permet de préciser la condition à évaluer. Si la condition est vraie alors le corps du tag when est évalué et le résultat est envoyé dans le flux de sortie de la JSP

Le tag otherwise permet de définir un cas qui ne correspond à aucun des autres cas inclus dans le tag. Ce tag ne possède aucun attribut.

Exemple :

```
<c:choose>
  <c:when test="${personne.civilite == 'Mr'}">
    Bonjour Monsieur
  </c:when>
  <c:when test="${personne.civilite == 'Mme'}">
    Bonjour Madame
  </c:when>
  <c:when test="${personne.civilite == 'Mlle'}">
    Bonjour Mademoiselle
  </c:when>
  <c:otherwise>
    Bonjour
  </c:otherwise>
</c:choose>
```

81.3.7. Le tag forEach

Ce tag permet de parcourir les différents éléments d'une collection et ainsi d'exécuter de façon répétitive le contenu de son corps.

Il possède plusieurs attributs :

Attribut	Rôle
var	nom de la variable qui contient l'élément en cours de traitement
items	collection à traiter
varStatus	nom d'une variable qui va contenir des informations sur l'itération en cours de traitement
begin	numéro du premier élément à traiter (le premier possède le numéro 0)
end	numéro du dernier élément à traiter
step	pas des éléments à traiter (par défaut 1)

A chaque itération, la valeur de la variable dont le nom est précisé par la propriété var change pour contenir l'élément de la collection en cours de traitement.

Pour les attributs, la seule obligation est d'avoir défini soit l'attribut items, soit les attributs begin et end.

Le tag forEach peut aussi réaliser des itérations sur les nombres et non sur des éléments d'une collection. Dans ce cas, il ne faut pas utiliser l'attribut items mais uniquement utiliser les attributs begin et end pour fournir les bornes inférieures et supérieures de l'itération.

Exemple :

```
<c:forEach begin="1" end="4" var="i">
<c:out value="${i}"/><br>
</c:forEach>v
```

Résultat :

```
1
2
3
4
```

L'attribut step permet de préciser le pas de l'itération.

Exemple :

```
<c:forEach begin="1" end="12" var="i" step="3">
  <c:out value="${i}"/><br>
</c:forEach>
```

Exemple :

```
1
4
7
10
```

L'attribut varStatus permet de définir une variable qui va contenir des informations sur l'itération en cours d'exécution. Cette variable possède plusieurs propriétés :

Attribut	Rôle
index	indique le numéro de l'occurrence dans l'ensemble de la collection
count	indique le numéro de l'itération en cours (en commençant par 1)
first	booléen qui indique si c'est la première itération
last	booléen qui indique si c'est la dernière itération

Exemple :

```
<c:forEach begin="1" end="12" var="i" step="3" varStatus="vs">
  index = <c:out value="${vs.index}"/> :
  count = <c:out value="${vs.count}"/> :
  value = <c:out value="${i}"/>
  <c:if test="${vs.first}">
     : Premier element
  </c:if>
  <c:if test="${vs.last}">
     : Dernier element
  </c:if>
  <br>
</c:forEach>
```

Résultat :

```
  index = 1 : count = 1 : value = 1 : Premier element
  index = 4 : count = 2 : value = 4
  index = 7 : count = 3 : value = 7
  index = 10 : count = 4 : value = 10 : Dernier element
```

81.3.8. Le tag forTokens

Ce tag permet de découper une chaîne selon un ou plusieurs séparateurs donnés et ainsi d'exécuter de façon répétitive le contenu de son corps autant de fois qu'il y a d'occurrences trouvées.

Il possède plusieurs attributs :

Attribut	Rôle
var	variable qui contient l'occurrence en cours de traitement (obligatoire)
items	la chaîne de caractères à traiter (obligatoire)
delims	précise le ou les séparateurs
varStatus	nom d'une variable qui va contenir des informations sur l'itération en cours de traitement
begin	numéro du premier élément à traiter (le premier possède le numéro 0)
end	numéro du dernier élément à traiter
step	pas des éléments à traiter (par défaut 1)

L'attribut delims peut avoir comme valeur une chaîne de caractères ne contenant qu'un seul caractère (délimiteur unique) ou un ensemble de caractères (délimiteurs multiples).

Exemple :

```
<c:forTokens var="token" items="chaine 1;chaine 2;chaine 3" delims=";">
  <c:out value="${token}" /><br>
</c:forTokens>v
```

Exemple :

```
chaine 1
chaine 2
chaine 3
```

Dans le cas où il y a plusieurs délimiteurs, chacun peut servir de séparateur.

Exemple :

```
<c:forTokens var="token" items="chaine 1;chaine 2,chaine 3" delims=";,">
  <c:out value="${token}" /><br>
</c:forTokens>
```

Attention : Il n'y a pas d'occurrence vide. Dans le cas où deux séparateurs sont juxtaposés dans la chaîne à traiter, ceux-ci sont considérés comme un seul séparateur. Si la chaîne commence ou se termine par un séparateur, ceux-ci sont ignorés.

Exemple :

```
<c:forTokens var="token" items="chaine 1;;chaine 2;;;chaine 3" delims=";">
  <c:out value="${token}" /><br>
</c:forTokens>
```

Résultat :

```
chaine 1
chaine 2
chaine 3
```

Il est possible de ne traiter qu'un sous-ensemble des occurrences de la collection. JSTL attribut à chaque occurrence un numéro incrémenté à partir de 0. Les attributs begin et end permettent de préciser une plage d'occurrences à traiter.

Exemple :

```
<c:forTokens var="token" items="chaine 1;chaine 2;chaine 3" delims=";" begin="1" end="1" >
  <c:out value="${token}" /><br>
</c:forTokens>
```

Résultat :

```
chaine 2
```

Il est possible de n'utiliser que l'attribut begin ou l'attribut end. Si seul l'attribut begin est précisé alors les dernières occurrences seront traitées. Si seul l'attribut end est précisé alors seuls les premières occurrences seront traitées.

Les attributs varStatus et step ont le même rôle que ceux du tag forEach.

81.3.9. Le tag import

Ce tag permet d'accéder à une ressource grâce à son URL pour l'inclure ou l'utiliser dans les traitements de la JSP. La ressource accédée peut être dans une autre application.

Son grand intérêt par rapport au tag <jsp :include> est de ne pas être limité au contexte de l'application web.

Il possède plusieurs attributs :

Attribut	Rôle
url	URL de la ressource (obligatoire)
var	nom de la variable qui va stocker le contenu de la ressource sous la forme d'une chaîne de caractères
scope	portée de la variable qui va stocker le contenu de la ressource
context	contexte de l'application web qui contient la ressource (si la ressource n'est pas l'application web courante)
charEncoding	jeu de caractères utilisé par la ressource
varReader	nom de la variable qui va stocker le contenu de la ressource sous la forme d'un objet de type java.io.Reader

L'attribut url permet de préciser l'URL de la ressource. Cette URL peut être relative à l'application web ou absolue.

Exemple :

```
<c:import url="/message.txt" /><br>
```

Par défaut, le contenu de la ressource est inclus dans la JSP. Il est possible de stocker le contenu de la ressource dans une chaîne de caractères en utilisant l'attribut var. Cet attribut attend comme valeur le nom de la variable.

Exemple :

```
<c:import url="/message.txt" var="message" />
<c:out value="${message}" /><BR/>
```

81.3.10. Le tag redirect

Ce tag permet de faire une redirection vers une nouvelle URL.

Les paramètres peuvent être fournis grâce à un ou plusieurs tags fils param.

Exemple :

```
<c:redirect url="liste.jsp">
  <c:param name="id" value="123"/>
</c:redirect>
```

81.3.11. Le tag url

Ce tag permet de formater une URL. Il possède plusieurs attributs :

Attribut	Rôle
value	base de l'URL (obligatoire)
var	nom de la variable qui va stocker l'URL
scope	portée de la variable qui va stocker l'URL
context	

Le tag url peut avoir un ou plusieurs tags fils « param ». Le tag param permet de préciser un paramètre et sa valeur pour qu'ils soient ajoutés à l'URL générée.

Le tag param possède deux attributs :

Attribut	Rôle
name	nom du paramètre
value	valeur du paramètre

Exemple :

```
<a href="<c:url url="/index.jsp"/>"/>
```

81.4. La bibliothèque XML

Cette bibliothèque permet de manipuler des données en provenance d'un document XML.

Elle propose les tags suivants répartis dans trois catégories :

Catégorie	Tag
Fondamentale	parse set out
Gestion du flux (condition et itération)	if

	choose forEach
Transformation XSLT	transform

Les exemples de cette section utilisent un fichier xml nommé personnes.xml dont le contenu est le suivant :

Fichier utilisé dans les exemples :

```
<personnes>
  <personne id="1">
    <nom>nom1</nom>
    <prenom>prenom1</prenom>
  </personne>
  <personne id="2">
    <nom>nom2</nom>
    <prenom>prenom2</prenom>
  </personne>
  <personne id="3">
    <nom>nom3</nom>
    <prenom>prenom3</prenom>
  </personne>
</personnes>
```

L'attribut select des tags de cette bibliothèque utilise la norme Xpath pour sa valeur. JSTL propose une extension supplémentaire à Xpath pour préciser l'objet sur lequel l'expression doit être évaluée. Il suffit de préfixer le nom de la variable par un $

Exemple : recherche de la personne dont l'id est 2 dans un objet nommé listepersonnes qui contient l'arborescence du document xml.

```
$listepersonnes/personnes/personne[@id=2]
```

L'implémentation de JSTL fournie avec le JWSDP utilise Jaxen comme moteur d'interprétation XPath. Donc pour utiliser cette bibliothèque, il faut s'assurer que les fichiers saxpath.jar et jaxen-full.jar soient présents dans le répertoire lib du répertoire WEB-INF de l'application web.

Pour utiliser cette bibliothèque, il faut la déclarer dans le fichier web.xml du répertoire WEB-INF de l'application web.

Exemple :

```
  <taglib>
    <taglib-uri>http://java.sun.com/jstl/xml</taglib-uri>
    <taglib-location>/WEB-INF/tld/x.tld</taglib-location>
  </taglib>
```

Lorsqu'une JSP utilise un ou plusieurs tags de la bibliothèque, celle-ci doit être déclarée avec une directive taglib.

Exemple :

```
<%@ taglib uri="http://java.sun.com/jstl/xml" prefix="x" %>
```

81.4.1. Le tag parse

Le tag parse permet d'analyser un document et de stocker le résultat dans une variable qui pourra être exploitée par la JSP ou une autre JSP selon la portée sélectionnée pour le stockage.

Attribut	Rôle

xml	contenu du document à analyser
var	nom de la variable qui va contenir l'arbre DOM généré par l'analyse
scope	portée de la variable qui va contenir l'arbre DOM
varDom	variable de type Document pour le document XML analysé
scopeDom	portée de la variable varDom
filter	filtre à appliquer sur le document source
system	URI du document XML en cours d'analyse

Exemple :

```
<c:import url="/personnes.xml" var="personnes" />
<x:parse xml="${personnes}" var="listepersonnes"  />
```

Dans cet exemple, il suffit simplement que le fichier personnes.xml soit dans le dossier racine de l'application web.

81.4.2. Le tag set

Le tag set est équivalent au tag set de la bibliothèque Core. Il permet d'évaluer l'expression Xpath fournie dans l'attribut select et de placer le résultat de cette évaluation dans une variable. L'attribut var permet de préciser la variable qui va recevoir le résultat de l'évaluation sous la forme d'un noeud de l'arbre du document XML.

Il possède plusieurs attributs :

Attribut	Rôle
select	expression XPath à évaluer
var	nom de la variable qui va stocker le résultat de l'évaluation
scope	portée de la variable qui va stocker le résultat

Exemple :

```
<c:import url="/personnes.xml" var="personnes" />
<x:parse xml="${personnes}" var="listepersonnes"  />
<x:set var="unepersonne" select="$listepersonnes/personnes/personne[@id=2]" />
<h1>nom = <x:out select="$unepersonne/nom"/></h1>
```

81.4.3. Le tag out

Le tag out est équivalent au tag out de la bibliothèque Core. Il permet d'évaluer l'expression Xpath fournie dans l'attribut select et d'envoyer le résultat dans le flux de sortie. L'attribut select permet de préciser l'expression Xpath qui doit être évaluée.

Il possède plusieurs attributs :

Attribut	Rôle
select	expression XPath à évaluer
escapeXML	true par défaut, si le contenu comprend des tags HTML, XML ou autres ils seront affichés tels quels. A false c'est l'évaluation des tags qui sera affichée

Exemple : Afficher le nom de la personne dont l'id est 2

```
<c:import url="/personnes.xml" var="personnes" />
<x:parse xml="${personnes}" var="listepersonnes"  />
<x:set var="unepersonne" select="$listepersonnes/personnes/personne[@id=2]" />
<h1><x:out select="$unepersonne/nom"/></h1>
```

Pour stocker le résultat de l'évaluation d'une expression dans une variable, il faut utiliser une combinaison du tag x:out et c:set

Exemple :

```
<c:set var="personneId">
      <x:out select="$listepersonnes/personnes/personne[@id=2]" />
    </c:set>
```

81.4.4. Le tag if

Ce tag est équivalent au tag if de la bibliothèque Core sauf qu'il évalue une expression XPath

Il possède plusieurs attributs :

Attribut	Rôle
select	expression XPath à évaluer sous la forme d'un booléen
var	nom de la variable qui va stocker le résultat de l'évaluation
scope	portée de la variable qui va stocker le résultat de l'évaluation

81.4.5. Le tag choose

Ce tag est équivalent au tag choose de la bibliothèque Core sauf qu'il évalue des expressions XPath

81.4.6. Le tag forEach

Ce tag est équivalent au tag forEach de la bibliothèque Core. Il permet de parcourir les noeuds issus de l'évaluation d'une expression Xpath.

Il possède plusieurs attributs :

Attribut	Rôle
select	expression XPath à évaluer (obligatoire)
var	nom de la variable qui va contenir le noeud en cours de traitement

Exemple :

```
<c:import url="/personnes.xml" var="personnes" />
<x:parse xml="${personnes}" var="listepersonnes"  />
<x:forEach var="unepersonne" select="$listepersonnes/personnes/*">
    <c:set var="personneId">
      <x:out select="$unepersonne/@id"/>
    </c:set>
    <c:out value="${personneId}" /> - <x:out select="$unepersonne/nom"/> &nbsp;
       <x:out select="$unepersonne/prenom"/> <br>
</x:forEach>
```

81.4.7. Le tag transform

Ce tag permet d'appliquer une transformation XSLT à un document XML. L'attribut xsl permet de préciser la feuille de style XSL. L'attribut optionnel xml permet de préciser le document xml.

Il possède plusieurs attributs :

Attribut	Rôle
xslt	feuille se style XSLT (obligatoire)
xml	nom de la variable qui contient le document XML à traiter
var	nom de la variable qui va recevoir le résultat de la transformation
scope	portée de la variable qui va recevoir le résultat de la transformation
xmlSystemId	deprecated : utiliser l'attribut docSuystemId
docSystemId	system identifier (URI) pour parser le document XML
xsltSystemId	system identifier (URI) parser le document XSLT
result	chaîne de caractères qui contient le résultat de la transformation

Exemple :

```
<x:transform xml='${docXml}' xslt='${feuilleXslt}'/>
```

Le document xml à traiter peut être fourni dans le corps du tag :

Exemple :

```
<x:transform xslt='${feuilleXslt}'>
  <personnes>
    <personne id="1">
      <nom>nom1</nom>
      <prenom>prenom1</prenom>
    </personne>
    <personne id="2">
      <nom>nom2</nom>
      <prenom>prenom2</prenom>
    </personne>
    <personne id="3">
      <nom>nom3</nom>
      <prenom>prenom3</prenom>
    </personne>
  </personnes>
</x:transform>
```

Le tag transform peut avoir un ou plusieurs noeuds fils param pour fournir des paramètres à la feuille de style XSLT.

81.5. La bibliothèque I18n

Cette bibliothèque facilite l'internationalisation d'une page JSP.

Elle propose les tags suivants répartis dans trois catégories :

Catégorie	Tag
Définition de la langue	setLocale
Formatage de messages	

	bundle message setBundle
Formatage de dates et nombres	formatNumber parseNumber formatDate parseDate setTimeZone timeZone

Pour utiliser cette bibliothèque, il faut la déclarer dans le fichier web.xml du répertoire WEB-INF de l'application web.

Exemple :

```
<taglib>
    <taglib-uri>http://java.sun.com/jstl/fmt</taglib-uri>
    <taglib-location>/WEB-INF/tld/fmt.tld</taglib-location>
</taglib>
```

La bibliothèque doit être déclarée avec la directive taglib pour chaque JSP qui utilise un ou plusieurs tags.

Exemple :

```
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>
```

Le plus simple pour mettre en oeuvre la localisation des messages, c'est de définir un ensemble de fichiers qui est appelé bundle en anglais.

Il faut définir un fichier pour la langue par défaut et un fichier pour chaque langue particulière. Tous ces fichiers ont un préfixe commun appelé basename et doivent avoir comme extension .properties. Les fichiers pour les langues particulières utilisent le préfixe commun suivit d'un underscore puis du code langue et éventuellement d'un underscore suivi du code pays. Ces fichiers doivent être inclus dans le classpath : le plus simple est de les copier dans le répertoire WEB-INF/classes de l'application web.

Exemple :

```
message.properties
message_en.properties
```

Dans chaque fichier, les clés sont identiques, seule la valeur associée à la clé change.

Exemple : le fichier message.properties pour le français (langue par défaut)

```
msg=bonjour
```

Exemple : le fichier message_en.properties pour l'anglais

```
msg=Hello
```

Pour plus d'information, voir le chapitre «L'internationalisation». .

81.5.1. Le tag bundle

Ce tag permet de préciser un bundle à utiliser dans les traitements contenus dans son corps.

Il possède plusieurs attributs :

Attribut	Rôle
basename	nom de base de la ressource à utiliser (obligatoire)
prefix	valeur préfixant chaque nom de clé pour les sous-attributs <fmt:message>

Exemple :

```
<fmt:bundle basename="message" >
  <fmt:message key="msg"/>
</fmt:bundle>
```

81.5.2. Le tag setBundle

Ce tag permet de déclarer un bundle par défaut.

Il possède plusieurs attributs :

Attribut	Rôle
basename	nom de base de la ressource à utiliser (obligatoire)
var	nom de la variable qui va stocker le nouveau bundle
scope	portée de la variable qui va recevoir le nouveau bundle

Exemple :

```
mon message =
<fmt:setBundle basename="message" />
  <fmt:message key="msg"/>
```

81.5.3. Le tag message

Ce tag permet de localiser un message.

Il possède plusieurs attributs :

Attribut	Rôle
key	clé du message à utiliser
bundle	bundle à utiliser
var	nom de la variable qui va recevoir le résultat du formatage
scope	portée de la variable qui va recevoir le résultat du formatage

Pour fournir chaque valeur, il faut utiliser un ou plusieurs tags fils param.

Exemple :

```
mon message =
<fmt:setBundle basename="message" />
  <fmt:message key="msg"/>
```

Résultat :

```
mon message = bonjour
```

Si aucune valeur n'est trouvée pour la clé fournie alors le tag renvoie ???XXX ??? où XXX représente le nom de la clé.

Exemple :

```
mon message =
<fmt:setBundle basename="message" />
  <fmt:message key="test"/>
```

Résultat :

```
mon message = ???test???
```

81.5.4. Le tag setLocale

Ce tag permet de sélectionner une nouvelle Locale.

Exemple :

```
<fmt:setLocale value="en"/>
mon message =
<fmt:setBundle basename="message" />
  <fmt:message key="msg"/>
```

Résultat :

```
mon message = Hello
```

81.5.5. Le tag formatNumber

Ce tag permet de formater des nombres selon la locale. L'attribut value permet de préciser la valeur à formater. L'attribut type permet de préciser le type de formatage à réaliser.

Il possède plusieurs attributs :

Attribut	Rôle
value	valeur à formater
type	CURRENCY ou NUMBER ou PERCENT
pattern	format personnalisé
currencyCode	code de la monnaie à utiliser pour le type CURRENCY
currencySymbol	symbole de la monnaie à utiliser pour le type CURRENCY
groupingUsed	booléen pour préciser si les nombres doivent être groupés
maxIntegerDigits	nombre maximum de chiffres dans la partie entière
minIntegerDigits	nombre minimum de chiffres dans la partie entière
maxFractionDigits	nombre maximum de chiffres dans la partie décimale
minFractionDigits	nombre minimum de chiffres dans la partie décimale
var	nom de la variable qui va stocker le résultat
scope	portée de la variable qui va stocker le résultat

Exemple :

```
<c:set var="montant" value="12345.67" />
montant = <fmt:formatNumber value="${montant}" type="currency"/>
```

81.5.6. Le tag parseNumber

Ce tag permet de convertir une chaîne de caractères qui contient un nombre en une variable décimale.

Il possède plusieurs attributs :

Attribut	Rôle
value	valeur à traiter
type	CURRENCY ou NUMBER ou PERCENT
parseLocale	Locale à utiliser lors du traitement
integerOnly	booléen qui indique si le résultat doit être un entier (true) ou un flottant (false)
pattern	format personnalisé
var	nom de la variable qui va stocker le résultat
scope	portée de la variable qui va stocker le résultat

Exemple : convertir en entier un identifiant passé en paramètre de la requête

```
<fmt:parseNumber value="${param.id}" var="id"/>
```

81.5.7. Le tag formatDate

Ce tag permet de formater des dates selon la Locale.

Il possède plusieurs attributs :

Attribut	Rôle
value	valeur à formater
type	DATE ou TIME ou BOTH
dateStyle	FULL ou LONG ou MEDIUM ou SHORT ou DEFAULT
timeStyle	FULL ou LONG ou MEDIUM ou SHORT ou DEFAULT
pattern	format personnalisé
timeZone	timeZone utilisée pour le formatage
var	nom de la variable qui va stocker le résultat
scope	portée de la variable qui va stocker le résultat

L'attribut value permet de préciser la valeur à formater. L'attribut type permet de préciser le type de formatage à réaliser. L'attribut dateStyle permet de préciser le style du formatage.

Exemple :

```
<jsp:useBean id="now" class="java.util.Date" />
Nous sommes le <fmt:formatDate value="${now}" type="date" dateStyle="full"/>.
```

81.5.8. Le tag parseDate

Ce tag permet d'analyser une chaîne de caractères contenant une date pour créer un objet de type java.util.Date.

Il possède plusieurs attributs :

Attribut	Rôle
value	valeur à traiter
type	DATE ou TIME ou BOTH
dateStyle	FULL ou LONG ou MEDIUM ou SHORT ou DEFAULT
timeStyle	FULL ou LONG ou MEDIUM ou SHORT ou DEFAULT
pattern	format personnalisé
parseLocale	Locale utilisée pour le formatage
timeZone	timeZone utilisée pour le formatage
var	nom de la variable de type java.util.date qui va stocker le résultat
scope	portée de la variable qui va stocker le résultat

81.5.9. Le tag setTimeZone

Ce tag permet de stocker un fuseau horaire dans une variable.

Il possède plusieurs attributs :

Attribut	Rôle
value	fuseau horaire à stocker (obligatoire)
var	nom de la variable de stockage
scope	portée de la variable de stockage

81.5.10. Le tag timeZone

Ce tag permet de préciser un fuseau horaire particulier à utiliser dans son corps.

Il possède plusieurs attributs :

Attribut	Rôle
value	chaîne de caractères ou objet java.util.TimeZone qui précise le fuseau horaire à utiliser

81.6. La bibliothèque Database

Cette bibliothèque facilite l'accès aux bases de données. Son but n'est pas de remplacer les accès réalisés grâce à des beans ou des EJB mais de fournir une solution simple, hélas non robuste, pour accéder à des bases de données. Ceci est cependant particulièrement utile pour développer des pages de tests ou des prototypes.

Elle propose les tags suivants répartis dans deux catégories :

Catégorie	Tag
Définition de la source de données	setDataSource
Exécution de requêtes SQL	query transaction update

Pour utiliser cette bibliothèque, il faut la déclarer dans le fichier web.xml du répertoire WEB-INF de l'application web.

Exemple :

```
  <taglib>
    <taglib-uri>http://java.sun.com/jstl/sql</taglib-uri>
    <taglib-location>/WEB-INF/tld/sql.tld</taglib-location>
</taglib>
```

La bibliothèque doit être déclarée par la directive taglib pour chaque JSP qui utilise un ou plusieurs tags.

Exemple :

```
<%@ taglib uri="http://java.sun.com/jstl/sql" prefix="sql" %>
```

81.6.1. Le tag setDataSource

Ce tag permet de créer une connexion vers la base de données à partir des données fournies dans les différents attributs du tag.

Il possède plusieurs attributs :

Attribut	Rôle
driver	nom de la classe du pilote JDBC à utiliser
source	URL de la base de données à utiliser
user	nom de l'utilisateur à utiliser lors de la connexion
password	mot de passe de l'utilisateur à utiliser lors de la connexion
var	nom de la variable qui va stocker l'objet créé lors de la connexion
scope	portée de la variable qui va stocker l'objet créé
dataSource	nom JNDI de la datasource

Exemple : accéder à une base via ODBC dont le DNS est test

```
<sql:setDataSource driver="sun.jdbc.odbc.JdbcOdbcDriver" url="jdbc:odbc:test"
user="" password=""/>
```

81.6.2. Le tag query

Ce tag permet de réaliser des requêtes de sélection sur une source de données.

Il possède plusieurs attributs :

Attribut	Rôle
sql	requête SQL à exécuter
var	nom de la variable qui stocke les résultats de l'exécution de la requête
scope	portée de la variable qui stocke les résultats
startRow	numéro de l'occurrence de départ à traiter
maxRow	nombre maximum d'occurrences à stocker
dataSource	connexion particulière à la base de données à utiliser

L'attribut sql permet de préciser la requête à exécuter :

Exemple :

```
<sql:query var="reqPersonnes" sql="SELECT * FROM personnes" />
```

Le résultat de l'exécution de la requête est stocké dans un objet qui implémente l'interface `javax.servlet.jsp.jstl.sql.Result` dont le nom est donné grâce à l'attribut var.

L'interface Result possède cinq getters :

Méthode	Rôle
String[] getColumnNames()	renvoie un tableau de chaines de caractères qui contient le nom des colonnes
int getRowCount()	renvoie le nombre d'enregistrements trouvés lors de l'exécution de la requête
Map[] getRows()	renvoie le résultat de la requête sous forme d'un tableau de Map. Chaque élément du tableau représente une rangée.
Object[][] getRowsByIndex()	renvoie un tableau contenant les colonnes et leurs valeurs
boolean isLimitedByMaxRows()	renvoie un booléen qui indique si le résultat de la requête a été limité

Exemple : connaitre le nombre d'occurrence renvoyées par la requête

```
<p>Nombre d'enregistrements trouvés : <c:out value="${reqPersonnes.rowCount}" /></p>
```

La requête SQL peut être précisée avec l'attribut sql ou dans le corps du tag :

Exemple :

```
<sql:query var="reqPersonnes" >
  SELECT * FROM personnes
</sql:query>
```

Le tag forEach de la bibliothèque Core est particulièrement utile pour itérer sur chaque occurrence retournée par la requête SQL.

Exemple :

```
<TABLE border="1" CELLPadding="4" cellspacing="0">
  <TR>
    <td>id</td>
    <td>nom</td>
    <td>prenom</td>
  </TR>

  <c:forEach var="row" items="${reqPersonnes.rows}" >
  <TR>
    <td><c:out value="${row.id}" /></td>
    <td><c:out value="${row.nom}" /></td>
    <td><c:out value="${row.prenom}" /></td>
  </TR>
  </c:forEach>
</TABLE>
```

Il est possible de fournir des valeurs à la requête SQL en remplaçant cette valeur par le caractère ?. Pour fournir la ou les valeurs, il faut utiliser un ou plusieurs tags fils param. Le tag param possède un seul attribut :

Attribut	Rôle
value	valeur de l'occurrence correspondante dans la requête SQL

Pour les valeurs de type date, il faut utiliser le tag dateParam.

Le tag dateParam possède plusieurs attributs :

Attribut	Rôle
value	objet de type java.util.date qui contient la valeur de la date (obligatoire)
type	format de la date : TIMESTAMP ou DATE ou TIME

Exemple :

```
<c:set var="id" value="2" />

<sql:query var="reqPersonnes" >
  SELECT * FROM personnes where id = ?
            <sql:param value="${id}" />
</sql:query>
```

81.6.3. Le tag transaction

Ce tag permet d'encapsuler plusieurs requêtes SQL dans une transaction. Il possède plusieurs attributs :

Attribut	Rôle
dataSource	connexion particulière à la base de données à utiliser
isolation	READCOMMITTED ou READUNCOMMITTED ou REPEATABLEREAD ou SERIALIZABLE

81.6.4. Le tag update

Ce tag permet de réaliser une mise à jour grâce à une requête SQL sur la source de données. Il possède plusieurs attributs :

Attribut	Rôle
sql	requête SQL à exécuter
var	nom de la variable qui stocke le nombre d'occurrences impactées par l'exécution de la requête
scope	portée de la variable qui stocke le nombre d'occurrences impactées
dataSource	connexion particulière à la base de données à utiliser

Exemple :

```
<c:set var="id" value="2" />
<c:set var="nouveauNom" value="nom 2 modifié" />

<sql:update var="nbRec">
UPDATE personnes
SET nom = ?
WHERE id=?
<sql:param value="${nouveauNom}"/>
<sql:param value="${id}"/>
</sql:update>

<p>nb enregistrements modifiés = <c:out value="${nbRec}"/></p>
```

82. Struts

Chapitre 82

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

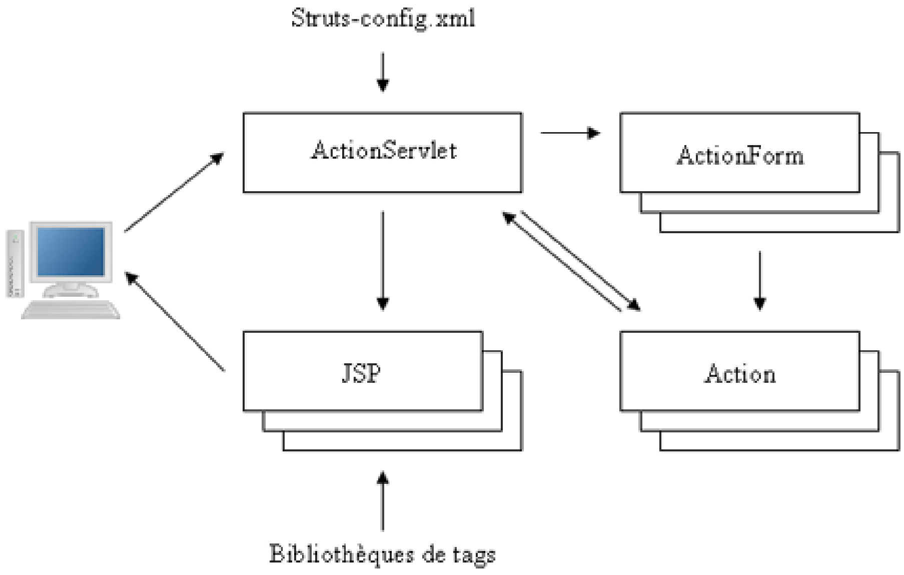
Framework legacy

Ce chapitre est conservé pour des raisons historiques

Struts est un framework pour applications web développé par le projet Jakarta de la fondation Apache. C'est le plus populaire des frameworks pour le développement d'applications web avec Java.

Il a été initialement développé par Craig Mc Clanahan qui l'a donné au projet Jakarta d'Apache en mai 2000. Depuis, Struts a connu un succès grandissant auprès de la communauté du libre et des développeurs à tel point qu'il sert de base à de nombreux autres framework open source et commerciaux et que la plupart des grands IDE propriétaires (Borland, IBM, BEA, ...) intègrent une partie dédiée à son utilisation.

Struts met en oeuvre le modèle MVC 2 basé sur une seule servlet faisant office de contrôleur et des JSP pour l'IHM. L'application de ce modèle permet une séparation en trois parties distinctes de l'interface, des traitements et des données de l'application.



Struts se concentre sur la vue et le contrôleur. L'implémentation du modèle est laissée libre aux développeurs : ils ont le choix d'utiliser des JavaBeans, un outil de mapping objet/relationnel, des EJB ou toute autre solution.

Pour le contrôleur, Struts propose une unique servlet par application qui lit la configuration de l'application dans un fichier au format XML. Cette servlet de type ActionServlet reçoit toutes les requêtes de l'utilisateur concernant l'application. En fonction du paramétrage, elle instancie un objet de type Action qui contient les traitements et renvoie une valeur particulière à la servlet. Celle-ci permet de déterminer la JSP qui affichera le résultat des traitements à l'utilisateur.

Les données issues de la requête sont encapsulées dans un objet de type ActionForm. Struts va utiliser l'introspection pour initialiser les champs de cet objet à partir des valeurs fournies dans la requête.

Struts utilise un fichier de configuration au format XML (struts-config.xml) pour connaître le détail des éléments qu'il va gérer dans l'application et comment ils vont interagir lors des traitements.

Pour la vue, Struts utilise par défaut des JSP avec un ensemble de plusieurs bibliothèques de tags personnalisés pour faciliter leur développement.

Struts propose aussi plusieurs services techniques : pool de connexions aux sources de données, internationalisation, ...

La dernière version ainsi que toutes les informations utiles peuvent être obtenues sur le site https://struts.apache.org/.

Il existe plusieurs versions de Struts : 1.0 (publiée en juin 2001), 1.1 et 1.2

Ce chapitre contient plusieurs sections :

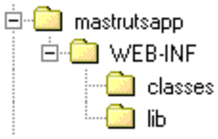
- ♦ L'installation et la mise en oeuvre
- ♦ Le développement des vues
- ♦ La configuration de Struts
- ♦ Les bibliothèques de tags personnalisés
- ♦ La validation de données

82.1. L'installation et la mise en oeuvre

Il faut télécharger la dernière version de Struts sur le site du projet Jakarta. La version utilisée dans cette section est la version 1.2.4.

Il suffit de décompresser le fichier jakarta-struts-1.2.4.zip dans un répertoire quelconque du système d'exploitation.

Il faut créer une structure de répertoires qui va accueillir l'application web, nommée par exemple mastrutsapp :



```
mastrutsapp
  WEB-INF
    classes
    lib
```

En utilisant Tomcat, une mise en oeuvre possible est de créer le répertoire de base de l'application dans le répertoire webapps.

Pour pouvoir utiliser Struts dans une application web, il faut copier les fichiers *.jar contenus du répertoire lib de Struts dans le répertoire WEB-INF/lib de l'application :

- commons-beanutils.jar
- commons-collection.jar
- commons-digester.jar
- commons-fileupload
- commons-logging.jar
- commons-validator.jar
- jakarta-oro.jar
- struts.jar

Il faut aussi copier les fichiers .tld (struts-bean.tld, struts-html.tld, struts-logic.tld, struts-nested.tld, struts-tiles.tld) dans le répertoire WEB-INF ou un de ses sous-répertoires.

Dans le répertoire WEB-INF, il faut créer deux fichiers :

- web.xml : le descripteur de déploiement de l'application
- struts-config.xml : le fichier de configuration de Struts

Le fichier web.xml minimal est le suivant :

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   version="2.4">
  <display-name>Mon application Struts de tests</display-name>

  <!-- Servlet controleur de Struts -->
  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <init-param>
      <param-name>debug</param-name>
      <param-value>2</param-value>
    </init-param>
    <init-param>
      <param-name>detail</param-name>
      <param-value>2</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
  </servlet>

  <!-- Mapping des url avec la servlet -->
  <servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>

  <!-- page d'accueil de l'application -->
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

  <jsp-config>
    <!-- Descripteur des bibliotheques personnalisees de Struts -->
    <taglib>
      <taglib-uri>/struts-bean</taglib-uri>
      <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
    </taglib>

    <taglib>
      <taglib-uri>/struts-html</taglib-uri>
      <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
    </taglib>

    <taglib>
      <taglib-uri>/struts-logic</taglib-uri>
      <taglib-location>/WEB-INF/struts-logic.tld</taglib-location>
    </taglib>

    <taglib>
      <taglib-uri>/struts-nested</taglib-uri>
      <taglib-location>/WEB-INF/struts-nested.tld</taglib-location>
    </taglib>

    <taglib>
      <taglib-uri>/struts-tiles</taglib-uri>
      <taglib-location>/WEB-INF/struts-tiles.tld</taglib-location>
    </taglib>
  </jsp-config>

</web-app>
```

Le mapping des URL de l'application prend généralement une des deux formes suivantes :

- préfixer chaque URL
- suffixer chaque URL avec une extension

Exemple de prefixe d'url :

```
  <servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>/do/*</url-pattern>
  </servlet-mapping>
```

Exemple de suffixe d'url :

```
  <servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>
```

Les exemples fournis sont simples : n'importe quel préfixe ou extension peut être utilisé avec sa forme respective.

Le fichier struts-config.xml minimal est le suivant :

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC
   "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
   "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">
<struts-config>

</struts-config>
```

Les fichiers web.xml et struts-config.xml seront complétés au fur et à mesure des sections suivantes.

Comme Struts met en oeuvre le modèle MVC, il est possible de développer séparément les différents composants de l'application.

82.1.1. Un exemple très simple

L'exemple de cette section va simplement demander le nom et le mot de passe de l'utilisateur et le saluer si ces deux données saisies ont une valeur précise.

Cet exemple est particulièrement simple et sera enrichi dans les autres sections de ce chapitre : son but est de proposer un exemple d'enchaînement de deux pages et de récupération des données d'un formulaire.

Le fichier struts-config.xml va contenir la définition des entités utilisées dans l'exemple : le Form Bean et l'Action.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config
PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 1.0//EN"
"http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">
<struts-config>

  <form-beans type="org.apache.struts.action.ActionFormBean">
    <form-bean name="loginForm" type="fr.jmdoudoux.dej.struts.data.LoginForm" />
  </form-beans>

  <action-mappings type="org.apache.struts.action.ActionMapping">
    <action path="/login" parameter="" input="/index.jsp" scope="request"
```

```
          name="loginForm" type="fr.jmdoudoux.dej.struts.controleur.LoginAction">
      <forward name="succes" path="/accueil.jsp" redirect="false" />
      <forward name="echec" path="/index.jsp" redirect="false" />
    </action>
  </action-mappings>

</struts-config>
```

Il faut écrire la page d'authentification.

Exemple : la page index.jsp

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<html:html locale="true">
  <head>
    <title>Authentification</title>
    <html:base/>
  </head>
  <body bgcolor="white">
    <html:form action="login" focus="nomUtilisateur">
      <table border="0" align="center">
        <tr>
          <td align="right">
            Utilisateur :
          </td>
          <td align="left">
            <html:text property="nomUtilisateur" size="20" maxlength="20"/>
          </td>
        </tr>
        <tr>
          <td align="right">
            Mot de Passe :
          </td>
          <td align="left">
            <html:password property="mdpUtilisateur" size="20" maxlength="20"
                          redisplay="false"/>
          </td>
        </tr>
        <tr>
          <td align="right">
            <html:submit property="submit" value="Submit"/>
          </td>
          <td align="left">
            <html:reset/>
          </td>
        </tr>
      </table>
    </html:form>
  </body>
</html:html>
```

Il faut aussi définir la page d'accueil qui sera affichée une fois l'utilisateur authentifié.

Exemple : la page accueil.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html:html locale="true">
  <head>
    <title>Accueil</title>
    <html:base/>
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
  </head>
  <body bgcolor="white">
     <h1> Bienvenue <bean:write name="loginForm" property="nomUtilisateur"/></h1>
  </body>
</html:html>
```

Il faut définir l'objet de type ActionForm qui va encapsuler les données saisies par l'utilisateur dans la page d'authentification.

Exemple : la classe LoginForm

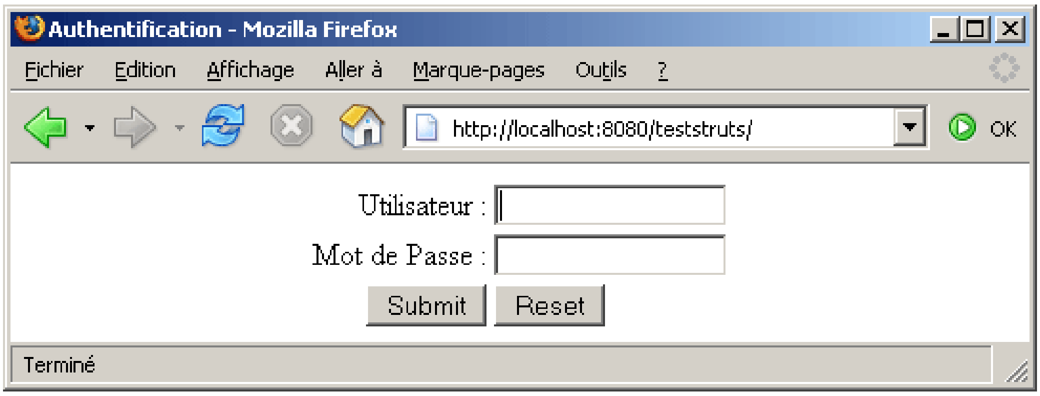
```
package fr.jmdoudoux.dej.struts.data;

import org.apache.struts.action.*;
import javax.servlet.http.HttpServletRequest;

public class LoginForm extends ActionForm {
  String nomUtilisateur;

  String mdpUtilisateur;

  public String getMdpUtilisateur() {
    return mdpUtilisateur;
  }

  public void setMdpUtilisateur(String mdpUtilisateur) {
    this.mdpUtilisateur = mdpUtilisateur;
  }

  public String getNomUtilisateur() {
    return nomUtilisateur;
  }

  public void setNomUtilisateur(String nomUtilisateur) {
    this.nomUtilisateur = nomUtilisateur;
  }

  public ActionErrors validate(ActionMapping mapping, HttpServletRequest request) {
    ActionErrors errors = new ActionErrors();
    return errors;
  }

  public void reset(ActionMapping mapping, HttpServletRequest request) {
    this.mdpUtilisateur = null;
    this.nomUtilisateur = null;
  }

}
```

Enfin, il faut définir un objet de type Action qui va encapsuler les traitements lors de la soumission du formulaire.

Exemple : la classe LoginAction

```
package fr.jmdoudoux.dej.struts.controleur;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

import fr.jmdoudoux.dej.struts.data.LoginForm;

public final class LoginAction extends Action {

  public ActionForward execute(ActionMapping mapping,
                               ActionForm form,
                               HttpServletRequest req,
                               HttpServletResponse res) throws Exception {
    String resultat = null;
    String nomUtilisateur = ((LoginForm) form).getNomUtilisateur();
    String mdpUtilisateur = ((LoginForm) form).getMdpUtilisateur();

    if (nomUtilisateur.equals("xyz") && mdpUtilisateur.equals("xyz")) {
      resultat = "succes";
    } else {
      resultat = "echec";
    }

    return mapping.findForward(resultat);
  }

}
```

Pour exécuter cet exemple, il faut le déployer dans un conteneur web (par exemple Tomcat)



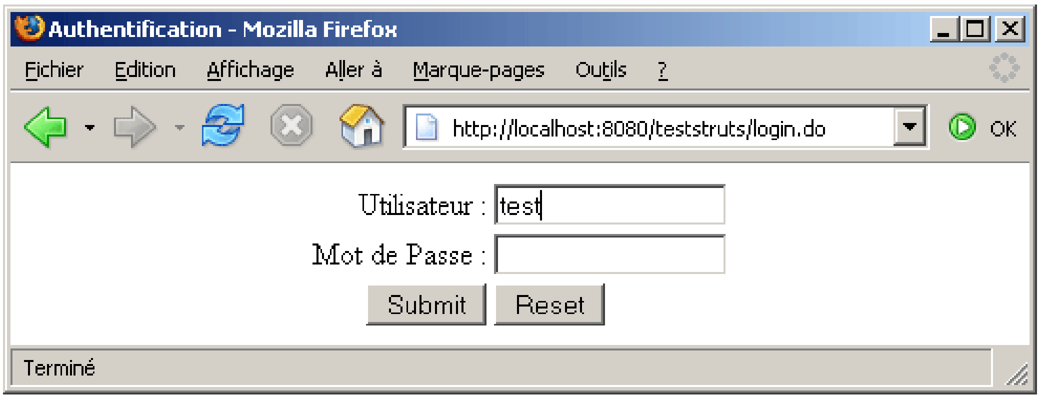
Authentification - Mozilla Firefox

http://localhost:8080/teststruts/

Utilisateur :

Mot de Passe :

Si le nom d'utilisateur et le mot de passe saisis ne valent pas « xyz » alors la page d'authentification est réaffichée.



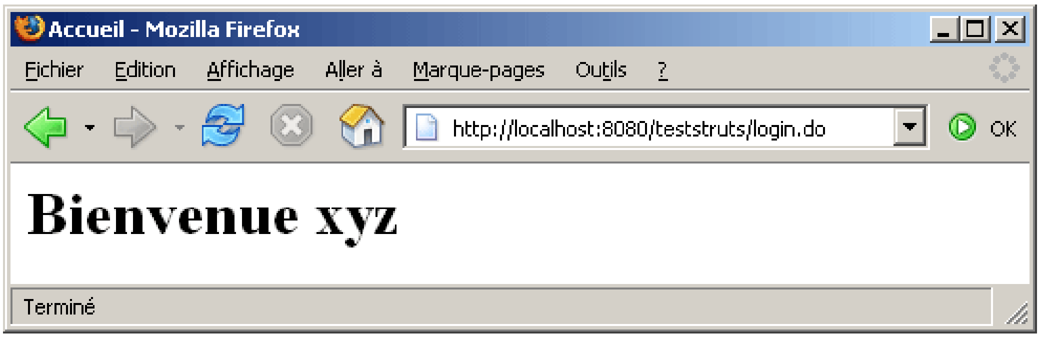
Authentification - Mozilla Firefox

http://localhost:8080/teststruts/login.do

Utilisateur : test

Mot de Passe :

Si le nom d'utilisateur et le mot de passe saisie valent « xyz » alors la page d'accueil s'affiche.

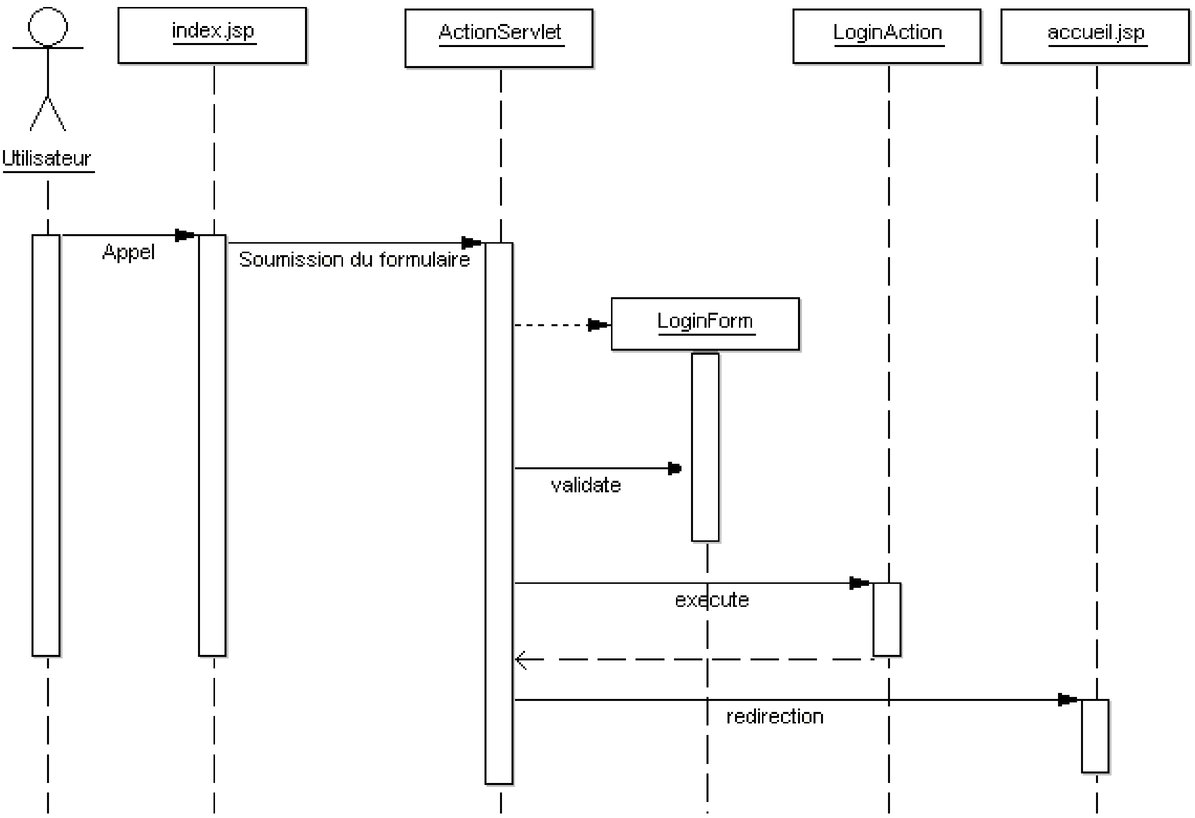


Accueil - Mozilla Firefox

http://localhost:8080/teststruts/login.do

Bienvenue xyz

Le diagramme de séquence ci-dessous résume les principales actions de cet exemple.



L'utilisateur appelle la page d'authentification index.jsp, saisit son nom d'utilisateur, son mot de passe et valide le formulaire.

L'ActionServlet intercepte la requête pour la traiter en effectuant les actions suivantes :

- Instancie un objet de type LoginForm et alimente ses données avec celles correspondantes dans la requête
- Appel de la méthode validate de la classe LoginForm pour valider les données saisies par l'utilisateur
- Détermination de l'Action à utiliser en fonction des informations contenues dans le fichier struts-config.xml. Dans l'exemple, c'est un objet de type LoginAction.
- Appel de la méthode execute() de la classe LoginAction qui contient les traitements à effectuer pour répondre à la requête. Elle renvoie un objet de type ActionForward
- En fonction de la valeur renvoyée par la méthode execute() et des informations du fichier de configuration, l'ActionServlet détermine la page à présenter à l'utilisateur
- La page déterminée est retournée au navigateur de l'utilisateur pour être affichée

82.2. Le développement des vues

Les vues représentent l'interface entre l'application et l'utilisateur. Avec le framework Struts, les vues d'une application web sont constituées par défaut de JSP et de pages HTML.

Pour faciliter leur développement, Struts propose un ensemble de nombreux tags personnalisés regroupés dans plusieurs bibliothèques possédant chacune un thème particulier :

- HTML : permet de faciliter le développement de pages Web en HTML
- Bean : permet de faciliter l'utilisation des Javabeans
- Logic : permet de faciliter la mise en oeuvre de la logique des traitements d'affichage
- Tiles : permet la gestion de modèles (templates)

Struts propose aussi au travers de ses tags de nombreuses fonctionnalités pour faciliter le développement : un formatage des données, une gestion des erreurs, ...

82.2.1. Les objets de type ActionForm

Un objet de type ActionForm est un objet respectant les spécifications des JavaBeans qui permet à Struts de mapper automatiquement les données saisies dans une page HTML avec les attributs correspondants dans l'objet. Il peut aussi réaliser une validation des données saisies par l'utilisateur.

Pour automatiser cette tâche, Struts utilise l'introspection pour rechercher un accesseur correspondant au nom du paramètre contenant la donnée dans la requête HTTP.

C'est la servlet faisant office de contrôleur qui instancie un objet de type ActionForm et alimente ses propriétés avec les valeurs contenues dans la requête émise à partir de la page.

Pour chaque page contenant des données à utiliser, il faut définir un objet qui hérite de la classe abstraite org.apache.struts.action.ActionForm. Par convention, le nom de cette classe est le nom de la page suivi de "Form".

Pour chaque donnée, il faut définir un attribut private ou protected qui contiendra la valeur, un getter et un setter public en respectant les normes de développement des Java beans.

Exemple :

```
package fr.jmdoudoux.dej.struts.data;

import org.apache.struts.action.*;
import javax.servlet.http.HttpServletRequest;

public class LoginForm extends ActionForm {
  String nomUtilisateur;

  String mdpUtilisateur;

  public String getMdpUtilisateur() {
    return mdpUtilisateur;
  }

  public void setMdpUtilisateur(String mdpUtilisateur) {
    this.mdpUtilisateur = mdpUtilisateur;
  }

  public String getNomUtilisateur() {
    return nomUtilisateur;
  }

  public void setNomUtilisateur(String nomUtilisateur) {
    this.nomUtilisateur = nomUtilisateur;
  }

...

}
```

La méthode reset() doit être redéfinie pour initialiser chaque attribut avec une valeur par défaut. Cette méthode est appelée par l'ActionServlet lorsqu'une instance de l'ActionForm est obtenue par la servlet et avant que cette dernière ne valorise les propriétés.

Exemple :

```
  public void reset(ActionMapping mapping, HttpServletRequest request) {
    this.mdpUtilisateur = null;
    this.nomUtilisateur = null;
  }
```

La signature de cette méthode est la suivante :
public void reset(ActionMapping mapping, HttpServletRequest request);

La méthode validate() peut être redéfinie pour permettre de réaliser des traitements de validation des données contenues dans l'ActionForm.

La signature de cette méthode est la suivante :
public ActionErrors validate(ActionMapping mapping, HttpServletRequest request);

Elle renvoie une instance de la classe ActionErrors qui encapsule les différentes erreurs détectées ou renvoie null si aucune erreur n'est rencontrée.

Exemple :

```
public ActionErrors validate(ActionMapping mapping, HttpServletRequest request) {
  ActionErrors errors = new ActionErrors();

  if ((nomUtilisateur == null) || (nomUtilisateur.length() == 0))
    errors.add("nomUtilisateur", new ActionError("erreur.nomutilisateur.obligatoire"));

  if ((mdpUtilisateur == null) || (mdpUtilisateur.length() == 0))
    errors.add("mdpUtilisateur", new ActionError("erreur.mdputilisateur.obligatoire"));

  return errors;
}
```

Comme les objets de type ActionForm sont des éléments de la vue du modèle MVC, les objets de type ActionForm ne doivent contenir aucun traitement métier. La méthode validate() ne doit contenir que des contrôles de surface (présence de données, taille des données, format des données, ...).

Il faut compiler cette classe et la placer dans le répertoire WEB-INF/classes suivi de l'arborescence correspondant au package de la classe.

Il faut aussi déclarer pour chaque ActionForm, un tag <form-bean> dans le fichier struts-config.xml. Ce tag possède plusieurs attributs :

Attribut	Rôle
Name	le nom sous lequel Struts va connaître l'objet
Type	le type complètement qualifié de la classe de type ActionForm

Exemple :

```
<form-beans type="org.apache.struts.action.ActionFormBean">
  <form-bean name="loginForm" type="fr.jmdoudoux.dej.struts.data.LoginForm" />
</form-beans>
```

Chaque objet de type ActionForm doit être défini avec un tag <form-beans> et <form-bean> dans le fichier de description struts-config.xml.

Pour demander l'exécution des traitements de validation des données, il est nécessaire d'utiliser l'attribut validate dans le fichier struts-config.xml.

Remarque : pour assurer un découplage entre la partie IHM et la partie métier, il n'est pas recommandé de passer à cette dernière une instance de type ActionForm. Il est préférable d'utiliser un objet dédié respectant le modèle de conception Data Transfert Object (DTO).

82.2.2. Les objets de type DynaActionForm

Le développement d'un objet de type ActionForm pour chaque page peut s'avérer fastidieux à écrire (même si des outils peuvent se charger de générer les getters et les setters nécessaires) et surtout à maintenir dans le cas d'une évolution. Ceci est d'autant plus vrai si cet objet n'est utilisé que pour obtenir les données du formulaire.

Struts propose les objets de type DynaActionForm qui permettent, après déclaration dans le fichier de configuration, d'obtenir dynamiquement les données sans avoir à développer explicitement un objet dédié.

Les DynaActionForm doivent donc obligatoirement être déclarés dans le fichier de configuration struts-config.xml comme les ActionForm.

Exemple :

```
  <form-beans>
    <form-bean name="saisirProduitActionForm"
               type="org.apache.struts.action.DynaActionForm">
      <form-property name="reference" type="java.lang.String"/>
      <form-property name="libelle" type="java.lang.String"/>
      <form-property name="prix" type="java.lang.String" initial="0"/>
    </form-bean>
  </form-beans>
```

Par défaut la méthode validate() de la classe DynaActionForm ne réalise aucun traitement. Pour pouvoir l'utiliser, il est nécessaire de créer une classe fille qui va hériter de DynaActionForm et dans laquelle la méthode validate() va être redéfinie. C'est cette classe fille qui devra alors être précisée dans l'attribut type du tag <form-bean>.

82.3. La configuration de Struts

L'essentiel de la configuration de Struts se fait dans le fichier de configuration struts-config.xml.

82.3.1. Le fichier struts-config.xml

Ce fichier au format XML contient le paramétrage nécessaire à l'exécution d'une application utilisant Struts.

Il doit se nommer struts-config.xml et il doit être dans le répertoire WEB-INF de l'application.

Le tag racine de ce document XML est le tag <struts-config>.

Ce fichier se compose de plusieurs parties :

- la déclaration des beans de formulaire (ActionForm) dans un tag <form-beans>
- la déclaration des redirections globales à toute l'application dans un tag <global-forwards>
- la déclaration des Action dans un tag <action-mappings>
- la déclaration des ressources dans un ou plusieurs tags <message-ressources>
- la déclaration des plugins dans un ou plusieurs tags <plug-in>

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config
PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 1.0//EN"
"http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">
<struts-config>

  <form-beans type="org.apache.struts.action.ActionFormBean">
    <form-bean name="loginForm" type="fr.jmdoudoux.dej.struts.data.LoginForm" />
  </form-beans>

  <action-mappings type="org.apache.struts.action.ActionMapping">
    <action path="/login" parameter="" input="/index.jsp" scope="request"
          name="loginForm" type="fr.jmdoudoux.dej.struts.controleur.LoginAction">
      <forward name="succes" path="/accueil.jsp" redirect="false" />
      <forward name="echec" path="/index.jsp" redirect="false" />
    </action>
  </action-mappings>

</struts-config>
```

Le tag <form-beans> permet de définir les objets de type ActionForm et DynaActionForm utilisés dans l'application.

Les DynaActionForm sont déclarés grâce à un tag <form-bean> fils du tag <form-beans>. Comme pour les ActionForm, le paramètre name permet de préciser le nom qui va faire référence au bean. L'attribut type doit avoir comme valeur org.apache.struts.action.DynaActionForm ou une classe pleinement qualifiée qui en hérite.

Chaque attribut du bean doit être déclaré dans un tag fils <form-property>. Ce tag possède plusieurs attributs :

- name : nom de la propriété
- type : type pleinement qualifié de la propriété suivi de [] pour un tableau
- size : taille si le type est un tableau
- initial : permet de préciser la valeur initiale de la propriété

Exemple :

```
  <form-beans>
    <form-bean name="saisirProduitActionForm"
               type="org.apache.struts.action.DynaActionForm">
      <form-property name="reference" type="java.lang.String"/>
      <form-property name="libelle" type="java.lang.String"/>
      <form-property name="prix" type="java.lang.String" initial="0"/>
    </form-bean>
  </form-beans>
```

Le tag <global-exception> permet de définir des handlers globaux à l'application pour traiter des exceptions.

Le tag <action-mappings> permet de définir l'ensemble des actions de l'application. Celles-ci sont unitairement définies grâce à un tag <action>.

Le tag Action permet d'associer une URL (/login.do dans l'exemple) avec un objet de type Action (LoginAction dans l'exemple). Ainsi, à chaque utilisation de cette URL, l'ActionServlet utilise la classe Action associée pour exécuter les traitements.

La propriété path permet d'indiquer l'URI d'appel de ce mapping : c'est cette valeur qui sera par exemple indiquée (suffixée ou préfixée selon le paramétrage du fichier web.xml) dans l'attribut action d'un formulaire ou href d'un lien.

La propriété type permet d'indiquer le nom pleinement qualifié de la classe Action qui sera utilisée par ce mapping.

La propriété name permet d'indiquer le nom d'un bean de type ActionForm associé à ce mapping. Cet objet encapsulera les données contenues dans la requête http.

La propriété scope permet de préciser la portée de l'objet ActionForm instancié par l'ActionServlet précisé par l'attribut name :

- request : la durée de vie des données ne concerne que la requête
- session : les données concernent un utilisateur
- application : les données sont communes à tous les utilisateurs de l'application

Il est préférable d'utiliser la portée la plus courte possible et d'éviter l'utilisation de la portée application.

L'attribut validate permet de préciser si les données de l'ActionForm doivent être validées en faisant appel à la méthode validate(). La valeur par défaut est true.

La propriété input permet de préciser l'URI de la page de saisie des données qui sera réaffichée en cas d'échec de la validation des données.

Le tag fils <forward> permet de préciser avec l'attribut path l'URI d'une page qui sera affichée lorsque l'Action renverra la valeur précisée dans l'attribut name. L'attribut redirect permet de préciser le type de redirection qui sera effectuée (redirect si la valeur est true sinon c'est un forward qui sera effectué). L'URI fournie doit être relative dans le cas d'un forward et relative ou absolue dans le cas d'un redirect.

Les informations contenues dans ce tag seront utilisées lors de l'instanciation d'objets de type ActionForward.

Le tag <global-forward> permet de définir des redirections communes à toute l'application. Ce tag utilise des tags fils de type <forward>. Les redirections définies localement sont prioritaires par rapport à celles définies de façon globales.

Le tag <message-ressources> permet de définir les ressources nécessaires à l'internationalisation de l'application.

Le tag <plug-in> permet de configurer des plugins de Struts tels que Tiles ou Validator.

Le tag <data-sources> permet de définir des sources de données. Chaque source de données est définie dans un tag <data-source>.

82.3.2. La classe ActionMapping

La classe ActionMapping encapsule les données définies dans un tag <Action> du fichier de configuration.

Chacune de ces ressources est définie dans le fichier de configuration struts-config.xml dans un tag <action> fils d'un tag <action-mappings>.

La méthode findForward() permet d'obtenir une redirection définie dans un tag <forward> de l'action ou dans un tag <global-forward>.

La classe ActionMappings encapsule une collection d'objets de type ActionMapping.

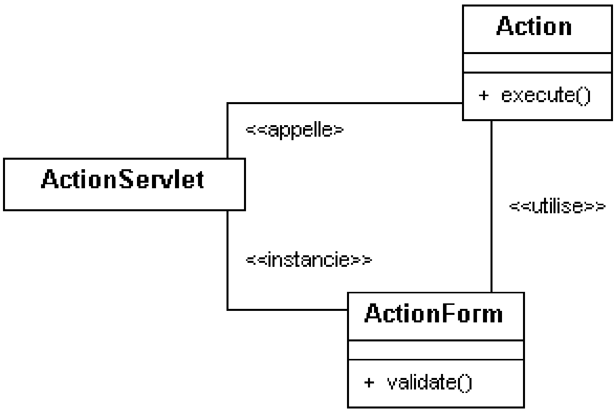
82.3.3. Le développement de la partie contrôleur

Basée sur le modèle MVC 2, la partie contrôleur de Struts se compose donc de deux éléments principaux :

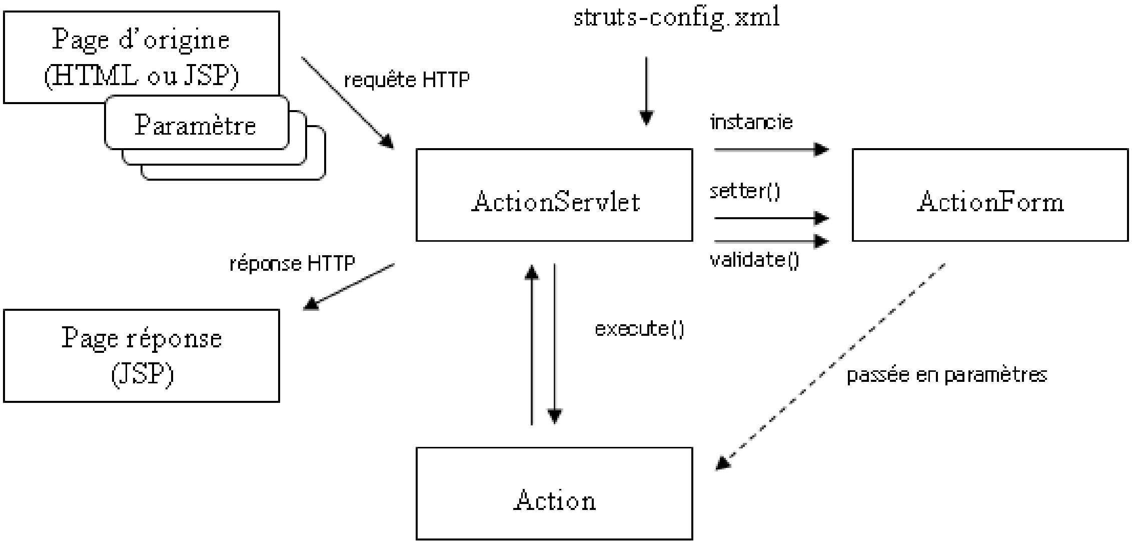
- une servlet de type org.apache.struts.action.ActionServlet
- plusieurs classes de type org.apache.struts.action.Action

La partie contrôleur est implémentée en utilisant une seule et unique servlet par application. Cette servlet doit hériter de la classe org.apache.struts.action.ActionServlet.

Cette servlet possède des traitements génériques qui utilisent les informations contenues dans le fichier struts-config.xml et dans des objets du type org.apache.struts.action.Action.



Une instance de la classe RequestProcessor est utilisée par l'ActionServlet en appelant sa méthode process() pour initialiser un objet de type ActionForm associé à l'action liée à la requête en cours de traitement.



L'ActionServlet vérifie la présence d'une instance du type de l'ActionForm dans la session : dans la négative, une nouvelle instance est créée et ajoutée à la session. La clé associée au bean dans la session est définie par l'attribut attribute du tag <Action>.

La requête est ensuite analysée : pour chaque attribut présent dans la requête, la servlet recherche dans l'ActionForm une propriété dont le nom correspond en utilisant l'introspection : si elle est trouvée, la servlet appelle son setter pour lui associer la valeur contenue dans la requête. La correspondance des noms doit être exacte en respectant la casse.

Si la validation est positionnée dans le fichier de configuration, la servlet appelle la méthode validate() de l'ActionForm. Si la validation réussie ou n'est pas demandée, l'ActionForm est passé en paramètre de la méthode execute() de l'instance d'Action.

82.3.4. La servlet de type ActionServlet

Le coeur d'une application Struts est composé d'une servlet de type org.apache.struts.action.ActionServlet.

Cette servlet reçoit les requêtes HTTP émises par le client et en fonction de celles-ci, elle appelle un objet du type Action qui lui est associé dans le fichier struts-config.xml. Le traitement d'une requête par une application Struts suit plusieurs étapes :

1. le navigateur client envoie une requête
2. réception de la requête par la servlet de type ActionServlet
3. en fonction de l'URI et du fichier de configuration struts-config.xml, la servlet instancie ou utilise l'objet de type ActionForm précisé. La servlet utilise l'introspection pour appeler les setters des propriétés dont les noms correspondent
4. la servlet instancie un objet de type Action associé à l'URI de la requête
5. la servlet appelle la méthode execute() de la classe Action. En retour de cet appel un objet de type ActionMapping permet d'indiquer à la servlet la page JSP qui sera affichée en réponse
6. la JSP génère la réponse HTML qui sera affichée sur le navigateur client

Pour respecter les spécifications J2EE, cette servlet doit être définie dans le fichier de déploiement web.xml de l'application web.

82.3.5. La classe Action

Un objet de type Action contient une partie spécifique de la logique métier de l'application : il est chargé de traiter ses données et de déterminer la page à afficher en fonction des traitements effectués.

Cet objet doit étendre la classe org.apache.struts.action.Action. Par convention, le nom de cette classe est le nom de la page suivi de "Action".

Il est important de développer ces classes de façon thread-safe : le contrôleur utilise une même instance pour traiter simultanément plusieurs requêtes. Il n'est donc pas recommandé d'utiliser des variables d'instances pour stocker des données sur une requête.

La méthode la plus importante de cette classe est la méthode execute(). C'est elle qui doit contenir les traitements qui seront exécutés. Depuis la version 1.1 de Struts, elle remplace la méthode perform() qui est deprecated mais toujours présente pour des raisons de compatibilité. La différence majeure entre les méthodes perform() et execute() est que cette dernière déclare la possibilité de lever une exception.

La méthode execute() attend plusieurs paramètres :

- Un objet de type ActionMapping
- Un objet de type ActionForm
- Un objet de type HttpServletRequest
- Un objet de type HtppServletResponse

Il existe une autre surcharge de la méthode execute() qui attend les mêmes paramètres sauf pour les deux derniers qui sont de types ServletRequest et ServletResponse.

Les traitements typiquement réalisés dans cette méthode sont les suivants :

- Effectuer un cast vers l'objet de type ActionForm à utiliser pour l'objet fourni en paramètre de la méthode : ceci permet un accès aux données spécifiques de l'objet de type ActionForm
- Réaliser les traitements requis sur ces données
- Déterminer la page de retour en fonction des traitements réalisés sous la forme d'un objet de type ActionForward.

Une bonne pratique de développement consiste à faire réaliser les traitements par des objets métiers dédiés indépendants de l'API Struts. Ces objets peuvent par exemple être des Javabeans ou des EJB.

Pour obtenir un objet de type ActionForward encapsulant la page réponse, il faut utiliser la méthode findForward() de l'objet de type ActionMapping passé en paramètre de la méthode execute(). La méthode findForward() attend en paramètre le nom de la page tel qu'il est défini dans le fichier struts-config.xml.

Cet objet est retourné au contrôleur qui assurera la redirection vers la page concernée.

Pour stocker les éventuelles erreurs rencontrées, il est nécessaire de créer une instance de la classe ActionErrors

Exemple :

```
ActionErrors erreurs = new ActionErrors();
```

Pour extraire les données issues de l'objet ActionForm, il est nécessaire d'effectuer un cast vers le type de l'instance fournie en paramètre :

Exemple :

```
    String nomUtilisateur = "";
    String mdpUtilisateur = "";

    if (form != null) {
      nomUtilisateur = ((LoginForm) form).getNomUtilisateur();
      mdpUtilisateur = ((LoginForm) form).getMdpUtilisateur();
    }
```

Pour extraire les données issues d'un objet de type DynaActionForm, il est nécessaire d'effectuer un cast vers le type DynaActionForm de l'instance fournie en paramètre.

Comme les objets de type DynaActionForm ne possèdent pas de getter et setter, pour obtenir la valeur d'une propriété d'un tel objet il est nécessaire d'utiliser la méthode get() en passant en paramètre le nom de la propriété et de caster la valeur retournée.

Exemple :

```
     DynaActionForm daf = (DynaActionForm)form;

     String reference = (String)daf.get("reference");
     String libelle = (String)daf.get("libelle");
     int prix = Integer.parseInt( (String)daf.get("prix")  );
```

Si une erreur est détectée dans les traitements, il faut instancier un objet de type ActionError et le fournir en paramètre avec le type de l'erreur à la méthode add() de l'instance de type ActionErrors.

Exemple :

```
    if (nomUtilisateur.equals("xyz") && mdpUtilisateur.equals("xyz")) {
      resultat = "succes";
    } else {
      erreurs.add(ActionErrors.GLOBAL_ERROR, new ActionError("erreur.login.invalide"));
      resultat = "echec";
    }
```

A la fin des traitements de la méthode execute(), si des erreurs ont été ajoutées il est nécessaire de faire appel à la méthode saveErrors() pour les enregistrer.

Exemple :

```
    if (!erreurs.isEmpty()) {
      saveErrors(req, erreurs);
    }
```

Pour permettre un affichage des erreurs, il faut faire renvoyer à la méthode une instance de la classe ActionForward() qui encapsule la page émettrice de la requête.

Exemple :

```
  return (new ActionForward(mapping.getInput()));
```

Sans erreur, le dernier traitement à réaliser est la création d'une instance de type ActionForward qui désignera la page à afficher en réponse à la requête.

Il y a deux façons d'obtenir cette instance :

- instancier directement un objet de type ActionForward
- utiliser la méthode findForward() de l'instance de type ActionMapping fournie en paramètre de la méthode execute()

Il existe plusieurs constructeurs pour la classe ActionForward dont les deux principaux sont :

- ActionForward(String path)
- ActionForward(String path, boolean redirect)

Le paramètre redirect est un booléen qui, avec la valeur true, fera procéder à une redirection vers la réponse (Response.sendRedirect()) et qui autrement provoquera le transfert vers la page réponse (RequestDispatcher.forward()).

L'utilisation de l'instance de type ActionMapping est sûrement la façon la plus pratique. Un appel à la méthode findForward() en précisant en paramètre le nom logique défini dans le fichier struts-config.xml permet d'obtenir un objet de type ActionForward pointant vers la page associée au nom logique.

Exemple :

```
    return mapping.findForward(resultat);
```

A partir de l'objet de type HttpRequest, il est possible d'accéder à la session en utilisant la méthode getSession().

Exemple :

```
HttpSession session = request.getSession();
session.setAttribute(" key ", user);
```

82.3.6. La classe DispatchAction

La classe DispatchAction permet d'associer plusieurs actions à un même formulaire. Cette situation est assez fréquente par exemple lorsqu'une page propose l'ajout, la modification et la suppression de données.

Elle va permettre en une seule action de réaliser une des opérations supportées par l'action. L'opération à réaliser selon l'action qui est sélectionnée par l'utilisateur doit être fournie dans la requête http sous la forme d'un champ caché de type Hidden ou en paramètre dans l'URL.

Exemple :

```
<%@ page contentType="text/html;charset=windows-1252"%>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean"%>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html"%>
<html:html locale="true">
  <html>
    <head>
      <meta http-equiv="Content-Type" content="text/html; charset=windows-1252"/>
      <title>untitled</title>
      <SCRIPT language="javascript" type="text/javascript">
      function setOperation(valeur){
        document.forms[0].operation.value=valeur;
      }
    </SCRIPT>
    </head>
    <body>
      <html:form action="operations.do" focusIndex="reference">
        <html:hidden property="operation" value="aucune"/>
        <table>
          <tr>
            <td>
              <bean:message key="app.saisirproduit.libelle.reference"/>:
            </td>
            <td>
              <html:text property="reference"/>
            </td>
          </tr>
          <tr>
            <td colspan="2" align="center">
              <html:submit onclick="setOperation('ajouter');">Ajouter</html:submit>
              <html:submit onclick="setOperation('modifier');">Modifier</html:submit>
              <html:submit onclick="setOperation('supprimer');">Supprimer</html:submit>
            </td>
          </tr>
        </table>
      </html:form>
    </body>
  </html>
</html:html>
```

L'implémentation de l'action doit hériter de la classe DispatchAction. Il est inutile de redéfinir la méthode execute() mais il faut définir autant de méthodes nommées avec les valeurs possibles des opérations.

L'introspection sera utilisée pour déterminer dynamiquement la méthode à appeler en fonction de l'opération reçue dans la requête.

Exemple :

```
package test.struts.controleur;

import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.actions.DispatchAction;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class OperationsAction extends DispatchAction {

  public ActionForward ajouter(
    ActionMapping       mapping,
    ActionForm          form,
    HttpServletRequest  request,
    HttpServletResponse response) throws IOException, ServletException {
    System.out.println("Appel de la methode ajouter()");
    return (mapping.findForward("succes"));
  }

  public ActionForward modifier(
    ActionMapping       mapping,
    ActionForm          form,
    HttpServletRequest  request,
    HttpServletResponse response) throws IOException, ServletException {
    System.out.println("Appel de la methode modifier()");
    return (mapping.findForward("succes"));
  }

  public ActionForward supprimer(
    ActionMapping       mapping,
    ActionForm          form,
    HttpServletRequest  request,
    HttpServletResponse response) throws IOException, ServletException {
    System.out.println("Appel de la methode supprimer()");
    return (mapping.findForward("succes"));
  }
}
```

Dans le fichier de configuration strut-config.xml, il faut déclarer l'action en précisant dans un attribut parameter le nom du paramètre de la requête qui contient l'opération à réaliser.

Exemple :

```
<struts-config>
...
  <form-beans>
...
    <form-bean name="operationsForm"
               type="org.apache.struts.action.DynaActionForm">
      <form-property name="operation" type="java.lang.String"/>
      <form-property name="reference" type="java.lang.String"/>
    </form-bean>
...
  </form-beans>
  <action-mappings>
...
    <action path="/operations" type="test.struts.controleur.OperationsAction"
            name="operationsForm" scope="request" validate="true" parameter="operation">
      <forward name="succes" path="/operations.jsp"/>
    </action>
...
  </action-mappings>
```

```
...
</struts-config>
```

Si la méthode à invoquer n'est pas définie dans la classe de type DispatchAction alors une exception est levée.

Exemple :

```
03-juil.-2006 13:14:43 org.apache.struts.actions.DispatchAction dispatchMethod
GRAVE: Action[/operations] does not contain method named supprimer
java.lang.NoSuchMethodException: test.struts.controleur.OperationsAction.supprimer(
org.apache.struts.action.ActionMapping, org.apache.struts.action.ActionForm,
javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse)
        at java.lang.Class.getMethod(Class.java)
```

Il est aussi possible d'utiliser plusieurs boutons avec pour valeur l'opération à réaliser. Ceci évite d'avoir à écrire du code JavaScript. Dans ce cas, chaque bouton doit avoir comme valeur de l'attribut property la valeur fournie à l'attribut parameter du tag <action>.

Exemple :

```
<%@ page contentType="text/html;charset=windows-1252"%>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean"%>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html"%>
<html:html locale="true">
  <html>
    <head>
      <meta http-equiv="Content-Type" content="text/html; charset=windows-1252"/>
      <title>untitled</title>
    </head>
    <body>
      <html:form action="operations.do" focusIndex="reference">
        <table>
          <tr>
            <td>
              <bean:message key="app.saisirproduit.libelle.reference"/>:
            </td>
            <td>
              <html:text property="reference"/>
            </td>
          </tr>
          <tr>
            <td colspan="2" align="center">
              <html:submit property="operation">ajouter</html:submit>
              <html:submit property="operation">modifier</html:submit>
              <html:submit property="operation">supprimer</html:submit>
            </td>
          </tr>
        </table>
      </html:form>
    </body>
  </html>
</html:html>
```

Attention cependant, la valeur du bouton est aussi son libellé : il est donc nécessaire de synchroniser le nom du bouton dans la vue et la méthode correspondante dans l'action. Ceci empêche l'internationalisation du libellé du bouton.

82.3.7. La classe LookupDispatchAction

Pour contourner le problème de l'internationalisation des opérations avec DispatchAction sans JavaScript, il est possible d'utiliser une action de type LookupDispatchAction.

Dans ce cas, le mapping ne se fait pas sur une valeur en dur mais sur la valeur d'une clé extraite des RessourcesBundles en fonction de la Locale courante.

La déclaration dans le fichier de configuration est similaire à celle nécessaire pour l'utilisation d'une action de type DispatchAction.

Exemple :

```
...
<struts-config>
  <form-beans>
..
    <form-bean name="operationsLookupForm"
               type="org.apache.struts.action.DynaActionForm">
      <form-property name="operation" type="java.lang.String"/>
      <form-property name="reference" type="java.lang.String"/>
    </form-bean>
...
  </form-beans>
  <action-mappings>
...
    <action path="/operationslookup" type="test.struts.controleur.OperationsLookupAction"
            name="operationsLookupForm" scope="request" validate="true" parameter="operation">
      <forward name="succes" path="/operationslookup.jsp"/>
    </action>
...
  </action-mappings>
...
</struts-config>
```

Dans la vue, le libellé des boutons de chaque action doit être défini dans les RessourcesBundles.

Exemple :

```
<%@ page contentType="text/html;charset=windows-1252"%>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean"%>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html"%>
<html:html locale="true">
  <html>
    <head>
      <meta http-equiv="Content-Type" content="text/html; charset=windows-1252"/>
      <title>Test LookupDispatchAction</title>
    </head>
    <body>
      <html:form action="operationslookup.do" focusIndex="reference">
        <table>
          <tr>
            <td>
              <bean:message key="app.saisirproduit.libelle.reference"/>:
            </td>
            <td>
              <html:text property="reference"/>
            </td>
          </tr>
          <tr>
            <td colspan="2" align="center">
              <html:submit property="operation">
                <bean:message key="operation.ajouter"/>
              </html:submit>
              <html:submit property="operation">
                <bean:message key="operation.modifier"/>
              </html:submit>
              <html:submit property="operation">
                <bean:message key="operation.supprimer"/>
              </html:submit>
            </td>
          </tr>
        </table>
      </html:form>
    </body>
  </html>
</html:html>
```

La valeur de chaque bouton doit être identique et précisée dans l'attribut property.

Il faut définir dans les ResourceBundles les libellés des boutons de chaque opération.

Exemple : ApplicationResources.properties

```
...
operation.ajouter   = Ajouter
operation.modifier  = Modifier
operation.supprimer = Supprimer
...
```

Exemple : ApplicationResources_en.properties

```
...
operation.ajouter   = Add
operation.modifier  = Modify
operation.supprimer = Delete
...
```

L'action doit hériter de la classe LookupDispatchAction. Il faut redéfinir la méthode getKeyMethodMap() pour qu'elle renvoie une collection de type Map dont chaque clé corresponde à la clé du ResourceBundle du bouton et chaque valeur à la méthode qui doit être invoquée.

La définition des méthodes de chaque opération est identique à celle utilisée avec une action de type DispatchAction.

Exemple :

```
package test.struts.controleur;

import java.util.HashMap;
import java.util.Map;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.actions.LookupDispatchAction;
import org.apache.struts.util.MessageResources;

public class OperationsLookupAction extends LookupDispatchAction {
  public static final String OPERATION_AJOUTER = "operation.ajouter";
  public static final String OPERATION_MODIFIER = "operation.modifier";
  public static final String OPERATION_SUPPRIMER = "operation.supprimer";


 public Map getKeyMethodMap() {
  Map map = new HashMap();
  map.put(OPERATION_AJOUTER, "ajouter");
  map.put(OPERATION_MODIFIER, "modifier");
  map.put(OPERATION_SUPPRIMER, "supprimer");
  return map;
}

  public ActionForward ajouter(
    ActionMapping       mapping,
    ActionForm          form,
    HttpServletRequest  request,
    HttpServletResponse response) throws IOException, ServletException {
    System.out.println("Appel de la methode ajouter()");
    return (mapping.findForward("succes"));
  }

  public ActionForward modifier(
    ActionMapping       mapping,
    ActionForm          form,
    HttpServletRequest  request,
    HttpServletResponse response) throws IOException, ServletException {
    System.out.println("Appel de la methode modifier()");
    return (mapping.findForward("succes"));
  }

  public ActionForward supprimer(
    ActionMapping       mapping,
    ActionForm          form,
    HttpServletRequest  request,
    HttpServletResponse response) throws IOException, ServletException {
    System.out.println("Appel de la methode supprimer()");
    return (mapping.findForward("succes"));
  }
}
```

Grâce à la méthode getKeyMethodMap(), la valeur de chaque opération est déterminée dynamiquement en fonction de la Locale.

82.3.8. La classe ForwardAction

Cette action permet uniquement une redirection vers une page sans qu'aucun traitement ne soit exécuté.

L'intérêt est de centraliser ces redirections dans le fichier de configuration plutôt que de les laisser en dur dans la ou les pages qui en ont besoin.

Il suffit de définir une action dans le fichier struts-config.xml en utilisant les attributs :

- path : l'URI de l'action
- type : org.apache.struts.actions.ForwardAction
- parameter : la page vers laquelle l'utilisateur va être redirigé

Exemple :

```
    <action path="/redirection"
            type="org.apache.struts.actions.ForwardAction"
            parameter="/test.jsp">
    </action>
```

Pour utiliser cette action, il suffit de faire un lien vers le path de l'action.

Exemple :

```
<html:link action="redirection.do">Page de test</html:link>
```

82.4. Les bibliothèques de tags personnalisés

L'utilisation des bibliothèques de tags de Struts nécessite leur définition dans le fichier de déploiement web.xml et leur déclaration dans chaque page qui les utilise.

Exemple :

```
  <%@ taglib uri="/WEB-INF/struts-html.tld"  prefix="html" %>
  <%@ taglib uri="/WEB-INF/struts-bean.tld"  prefix="bean" %>
  <%@ taglib uri="/WEB-INF/struts-logic.tld"  prefix="logic" %>
```

Les vues sont aussi composées selon le modèle MVC d'objets de type ActionForm ou DynaActionForm qui encapsulent les données d'une page. Ils permettent l'échange de données entre la vue et les objets métiers par le contrôleur.

82.4.1. La bibliothèque de tags HTML

Cette bibliothèque permet de faciliter le développement de page Web en HTML.

Pour utiliser cette bibliothèque, il faut, comme pour toute bibliothèque de tags personnalisés, réaliser plusieurs opérations :

1. copier le fichier struts-html.tld dans le répertoire WEB-INF de la webapp
2. configurer le fichier WEB-INF/web.xml pour déclarer la bibliothèque de tag

```
<taglib>
  <taglib-uri>struts-html.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
</taglib>
```

3. ajouter dans chaque page JSP qui va utiliser cette bibliothèque un tag de directive taglib précisant l'utilisation de la bibliothèque

```
<%@ taglib uri="struts-html.tld" prefix="html" %>
```

La plupart de ces tags encapsulent des tags HTML notamment pour les formulaires mais ils assurent aussi des traitements particuliers à Struts.

Exemple :

Un lien vers une URL absolue avec HTML doit intégrer le nom de la webapp :
<a href="/testwebapp/index.jsp">

La balise Struts correspondante sera indépendante de la webapp : elle tient compte automatiquement du contexte de l'application
<html:link page="/index.jsp">

Il est cependant préférable d'utiliser un mapping défini dans le fichier struts-config.xml plutôt que d'utiliser un lien vers la page JSP correspondante. Ceci va permettre l'exécution de l'Action correspondante.

Exemple :

```
<html:link page="/index.do">Accueil</html:link>
```

Tag	**Description**
base	Encapsule un tag HTML <base>
button	Encapsule un tag HTML <input type="button">
cancel	Encapsule un tag HTML <input type="submit"> avec la valeur Cancel
checkbox	Encapsule un tag HTML <input type="checkbox">
errors	Affiche les messages d'erreurs stockés dans la session
file	Encapsule un tag HTML <input type="file">
form	Encapsule un tag HTML <form>
frame	Encapsule un tag HTML <frame>
hidden	Encapsule un tag HTML <input type="hidden">
html	Encapsule un tag HTML <html>

image	Encapsule une action affichée sous la forme d'une image
img	Encapsule un tag HTML <img>
javascript	Assure la génération du code JavaScript requis par le plug-in Validator
link	Encapsule un tag HTML <A>
messages	Affiche les messages stockés dans la session
multibox	Assure le rendu de plusieurs checkbox
option	encapsule un tag HTML <option>
options	Assure le rendu de plusieurs options
optionsCollection	Assure le rendu de plusieurs options
password	Encapsule un tag HTML <input type="password">
radio	Encapsule un tag HTML <input type="radio">
reset	Encapsule un tag HTML <input type="reset">
rewrite	Le rendu d'une URI
select	Encapsule un tag HTML <select>
submit	Encapsule un tag HTML <input type="submit">
text	Encapsule un tag HTML <input type="text">
textarea	Encapsule un tag HTML <input type="textarea">
xhtml	Le rendu des tags HTML est au format XHTML

Les tags les plus utilisés seront détaillés dans les sections suivantes.

82.4.1.1. Le tag <html:html>

Ce tag génère un tag HTML <html>. Il possède plusieurs attributs dont les principaux sont :

Attribut	Rôle
lang	génère un attribut lang en accord avec celui stocké dans la session ou la requête, ou encore, selon la Locale par défaut
locale	utilise la valeur true pour forcer le stockage dans la session de la Locale correspondant à la langue de la requête Ce tag est deprecated depuis la version 1.2 car il crée automatiquement une session : utiliser l'attribut lang à la place
xhtml	utilise la valeur true pour assurer un rendu au format xhtml des tags

Ce tag doit être inclus dans un tag <html:form>.

82.4.1.2. Le tag <html:form>

Ce tag génère un tag HTML <form>.

Il possède de nombreux attributs correspondant aux attributs du tag html <form> dont les principaux sont :

Attribut	Rôle

action	URL à laquelle le formulaire sera soumis
enctype	type d'encodage du formulaire lors de la soumission
focus	nom de l'élément qui aura le focus au premier affichage de la page
method	méthode de soumission du formulaire
name	nom associé à la classe ActionForm
scope	portée de la classe ActionForm
target	cible d'affichage de la réponse
type	type de la classe ActionForm

82.4.1.3. Le tag <html:button>

Ce tag génère un tag HTML <input> de type button.

Il possède de nombreux attributs dont les principaux sont :

Attribut	Rôle
alt	correspond à l'attribut alt du tag HTML
altKey	clé du ResourceBundle dont la valeur sera affectée à l'attribut alt du tag HTML
bundle	nom du bean qui encapsule le ResourceBundle (utilisé lorsque plusieurs ResourceBundles sont définis)
disabled	true pour rendre le bouton non opérationnel
property	nom du paramètre dans la requête http lors de la soumission du formulaire : correspond à l'attribut name du tag HTML
title	correspond à l'attribut title du tag HTML
titleKey	clé du ResourceBundle dont la valeur sera affectée à l'attribut title du tag HTML
value	libellé du bouton : correspond à l'attribut value du tag HTML

Ce tag doit être inclus dans un tag <html:form>

Exemple :

```
<html:button property="valider" value="Valider" title="Valider les données" />
```

Résultat

```
<input type="button" name="valider" value="Valider" title="Valider les données">
```

82.4.1.4. Le tag <html:cancel>

Ce tag génère un tag HTML <input> de type button avec une valeur spécifique pour permettre d'identifier ce bouton comme étant celui de type "Cancel".

Il possède des attributs similaires au tag <html:button>.

Il n'est pas recommandé d'utiliser l'attribut property : il faut laisser la valeur par défaut de Struts pour lui permettre d'identifier ce bouton. La valeur par défaut de l'attribut property permet à Struts de déterminer la valeur de retour de la méthode Action.isCancelled.

Exemple :

```
<html:cancel />
```

Résultat

```
<input type="submit" name="org.apache.struts.taglib.html.CANCEL" value="Cancel" onclick="bCancel=true;">
```

82.4.1.5. Le tag <html:submit>

Ce tag génère un tag HTML <input type="submit"> permettant la validation d'un formulaire.

Il possède des attributs similaires au tag <html:button>.

Ce tag doit être inclus dans un tag <html:form>

Exemple :

```
<html:submit />
```

Résultat

```
<input type="submit" value="Submit">
```

82.4.1.6. Le tag <html:radio>

Ce tag génère un tag HTML <input type="radio"> permettant d'afficher un bouton radio.

Exemple :

```
<html:radio property="sexe" value="femme" />Femme<br>
<html:radio property="sexe" value="homme" />Homme<br>
```

Résultat :

```
<input type="radio" name="sexe" value="femme">Femme<br>
<input type="radio" name="sexe" value="homme">Homme<br>
```

82.4.1.7. Le tag <html:checkbox>

Ce tag génère un tag HTML <input type="checkbox"> permettant d'afficher un bouton de type case à cocher.

Exemple :

```
<html:checkbox property="caseACocher"> Une case à cocher</html:checkbox>
```

Résultat :

```
<input type="checkbox" name="caseACocher" value="on">Une case à cocher
```

82.4.2. La bibliothèque de tags Bean

Cette bibliothèque fournit des tags pour faciliter la gestion et l'utilisation des javabeans.

Pour utiliser cette bibliothèque, il faut, comme pour toute bibliothèque de tags personnalisés, réaliser plusieurs opérations :

1. copier le fichier struts-bean.tld dans le répertoire WEB-INF de la webapp
2. configurer le fichier WEB-INF/web.xml pour déclarer la bibliothèque de tag

```
<taglib>
  <taglib-uri>struts-bean.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
</taglib>
```

3. ajouter dans chaque page JSP qui va utiliser cette bibliothèque un tag de directive taglib précisant l'utilisation de la bibliothèque

```
<%@ taglib uri="struts-bean.tld" prefix="bean" %>
```

82.4.2.1. Le tag <bean:cookie>

Le tag <bean:cookie> permet d'obtenir la ou les valeurs d'un cookie.

Il possède plusieurs attributs :

Attribut	Rôle
id	identifiant du cookie
name	nom du cookie
multiple	précise si toutes les valeurs ou seulement la première valeur du cookie sont retournées
value	valeur du cookie; si celui-ci n'existe pas alors il est créé

82.4.2.2. Le tag <bean:define>

Le tag <bean:define> permet de définir une variable.

Il possède plusieurs attributs :

Attribut	Rôle
id	nom de la variable qui va être créée
name	nom du bean qui va fournir la valeur
property	propriété du bean qui va fournir la valeur
scope	portée du bean
toScope	portée de la variable créée
type	type de la variable créée
value	valeur à utiliser l'attribut name n'est pas utilisé

Exemple :

```
<jsp:useBean id="utilisateur" scope="page" class=" fr.jmdoudoux.dej.struts.data.Utilisateur"/>
<bean:define id="nomUtilisateur" name="utilisateur" property="nom"/>
Bienvenue <%= nomUtilisateur %>
```

Cet exemple permet de définir un bean de type Utilisateur qui est stocké dans la portée page. Une variable nomUtilisateur est définie et initialisée avec la valeur de la propriété nom du bean de type Utilisateur.

82.4.2.3. Le tag <bean:header>

Le tag <bean:header> est similaire au tag <bean:cookie> mais il permet de manipuler des données contenues dans l'en-tête de la requête HTTP.

82.4.2.4. Le tag <bean:include>

Le tag <bean:include> permet d'évaluer et d'inclure le rendu d'une autre page. Son mode de fonctionnement est similaire au tag JSP <jsp:include> excepté que le rendu de la page n'est pas inclus directement dans la page mais dans une variable.

Il possède plusieurs attributs :

Attribut	Rôle
id	nom de la variable créée qui va contenir le résultat. Cette variable sera stockée dans la portée page.
forward	nom d'une redirection globale définie dans le fichier de configuration
href	URL de la page
page	URI relative au contexte de l'application de la page

Exemple :

```
<bean:include id="barreNavigation"  page="/navigation.jsp"/>
<bean:write name="barreNavigation"  filter="false" />
```

Ce tag est utile notamment pour obtenir un document XML qu'il sera alors possible de manipuler.

82.4.2.5. Le tag <bean:message>

Le tag <bean:message> permet d'obtenir la valeur d'un libellé contenu dans un ResourceBundle.

Il possède plusieurs attributs :

Attribut	Rôle
arg0	valeur du premier paramètre de remplacement
arg1	valeur du second paramètre de remplacement
arg2	valeur du troisième paramètre de remplacement
arg3	valeur du quatrième paramètre de remplacement
arg4	valeur du cinquième paramètre de remplacement
bundle	nom du bean qui encapsule le ResourceBundle (utilisé lorsque plusieurs ResourceBundle sont définis)
key	clé du libellé à obtenir
locale	nom du bean qui stocke la Locale dans la session
name	nom du bean qui encapsule les données
property	propriété du bean précisé par l'attribut name contenant la valeur du libellé
scope	portée du bean précisé par l'attribut name

82.4.2.6. Le tag <bean:page>

Le tag <bean:page> permet d'obtenir une variable implicite définie par l'API JSP contenue dans la portée page.

Il possède plusieurs attributs :

Attribut	Rôle
id	nom de la variable à créer
property	variable implicite à extraire. Les valeurs possibles sont : application, config, request, response ou session

82.4.2.7. Le tag <bean:param>

Le tag <bean:param> est similaire au tag <bean:cookie> mais il permet de manipuler des données contenues dans les paramètres de la requête HTTP.

82.4.2.8. Le tag <bean:resource>

Le tag <bean:resource> permet d'obtenir la valeur d'une ressource sous la forme d'un objet de type java.io.InputStream ou String.

Il possède plusieurs attributs :

Attribut	Rôle
id	nom de la variable à créer
name	URI de la ressource relative à l'application à utiliser
input	permet d'obtenir la ressource sous la forme d'un objet de type java.io.InputStream. Sinon c'est un objet de type String qui est retourné

82.4.2.9. Le tag <bean:size>

Le tag <bean:size> permet d'obtenir le nombre d'éléments d'une collection ou d'un tableau. Ce tag crée une variable de type java.lang.Integer.

Il possède plusieurs attributs :

Attribut	Rôle
id	nom de la variable à créer
collection	expression renvoyant la collection à traiter
name	nom du bean qui encapsule la collection
property	propriété du bean qui encapsule la collection
scope	portée du bean

Exemple :

```
<bean:size id="count" name="elements" />
```

82.4.2.10. Le tag <bean:struts>

Le tag <bean:struts> permet de copier un objet Struts (FormBean, Mapping, Forward) dans une variable. Il possède plusieurs attributs :

Attribut	Rôle
id	nom de la variable à créer (attribut obligatoire)
formBean	nom du bean de type ActionForm
forward	nom de l'objet global de type ActionForward
mapping	nom de l'objet de type ActionMapping

82.4.2.11. Le tag <bean:write>

Le tag <bean:write> permet d'envoyer dans le JspWrite courant la valeur d'un bean ou d'une propriété d'un bean.

Il possède plusieurs attributs :

Attribut	Rôle
bundle	nom du bean qui encapsule le ResourceBundle (utilisé lorsque plusieurs ResourceBundle sont définis)
filter	la valeur true permet de remplacer les caractères spécifiques d'HTML par leur entité correspondante
format	format de conversion en chaîne de caractères
formatKey	clé du ResourceBundle qui précise le format de conversion en chaîne de caractères
ignore	la valeur true permet d'ignorer l'inexistence du bean dans la portée. La valeur false lève une exception si le bean n'est pas trouvé dans la portée. Par défaut, false
locale	nom du bean qui stocke la Locale dans la session
name	nom du bean qui encapsule les données (attribut obligatoire)
property	propriété du bean
scope	portée du bean

Exemple :

```
<jsp:useBean id="utilisateur" scope="page" class=" fr.jmdoudoux.dej.struts.data.Utilisateur"/>
<bean:write name="utilisateur" property="nom"/>
```

L'attribut format du tag permet de formater les données restituées par le bean.

Exemple :

```
<p><bean:write  name="monbean" property="date" format="dd/MM/yyyy  HH:mm"/></p>
```

L'attribut formatKey du tag permet de formater les données restituées par le bean à partir d'une clé des ResourceBundle : ceci permet d'internationaliser le formatage.

Exemple :

```
<p><bean:write  name="monbean" property="date"  formatKey="date.format"/></p>
```

Dans le fichier ApplicationResources.properties

date.format=dd/MM/yyyy HH:mm

Dans le fichier ApplicationResources_en.properties

date.format=MM/dd/yyyy HH:mm

Il est important que le format précisé soit compatible avec la Locale courante sinon une exception est levée

Exemple :

```
javax.servlet.jsp.JspException: Wrong format string: '#.##0,00'
          at org.apache.struts.taglib.bean.WriteTag.formatValue(WriteTag.java:376)
          at org.apache.struts.taglib.bean.WriteTag.doStartTag(WriteTag.java:292)
```

82.4.3. La bibliothèque de tags Logic

Cette bibliothèque fournie des tags pour faciliter l'utilisation de logiques de traitements pour l'affichage des pages.

Pour utiliser cette bibliothèque, il faut, comme pour toute bibliothèque de tags personnalisés, réaliser plusieurs opérations :

1. copier le fichier struts-logic.tld dans le répertoire WEB-INF de la webapp
2. configurer le fichier WEB-INF/web.xml pour déclarer la bibliothèque de tags

```
<taglib>
  <taglib-uri>struts-logic.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-logic.tld</taglib-location>
</taglib>
```

3. ajouter dans chaque page JSP qui va utiliser cette bibliothèque un tag de directive taglib précisant l'utilisation de la bibliothèque

```
<%@ taglib uri="struts-logic.tld" prefix="logic" %>
```

La plupart de ces tags encapsulent des tags de conditionnement des traitements ou d'exécution d'opérations sur le flot des traitements.

L'utilisation de ces tags évite l'utilisation de code Java dans les JSP.

Exemple :

```
  <jsp:useBean id="elements"  scope="request" class="java.util.List" />
    ...
  <%
    for (int i = 0; i  < elements.size(); i++)
    {
      MonElement monElement =  (MonElement)elements.get(i);
  %>
  <%=monElement.getLibelle()%>
  <%
    }
  %>
```

Tout le code Java peut être remplacé par l'utilisation de tag de la bibliothèque struts-logic.

Exemple :

```
<jsp:useBean  id="elements" scope="request" class="java.util.List" />
<logic:iterate id="monElement" name="elements" type="fr.jmdoudoux.dej.struts.data.MonElement">
  <bean:write name="monElement" property="libelle"/>
</logic:iterate>
```

Cette bibliothèque définit une quinzaine de tags.

Dans différents exemples de cette section, le bean suivant sera utilisé :

Exemple :

```
package test.struts.data;

import java.util.Date;

public  class MonBean {
  private String libelle;
  private Integer valeur;
  private Date date;

  public MonBean() {
    libelle="libelle de test";
    valeur = new Integer(123456);
    date = new Date();
  }

  public void setLibelle(String libelle) {
    this.libelle = libelle;
  }

  public String getLibelle() {
    return libelle;
  }

  public void setDate(Date date) {
    this.date = date;
  }

  public Date getDate() {
    return date;
  }

  public void  setValeur(Integer valeur) {
    this.valeur =  valeur;
  }

  public Integer  getValeur() {
    return valeur;
  }
}
```

L'intérêt de cette bibliothèque a largement diminué depuis le développement de la JSTL qui intègre en standard des fonctionnalités équivalentes. Il est d'ailleurs fortement recommandé d'utiliser dès que possible les tags de la JSTL à la place des tags de Struts.

82.4.3.1. Les tags <logic:empty> et <logic:notEmpty>

Le tag <logic:empty> permet de tester si une variable est null ou vide. Le tag <logic:notEmpty> permet de faire le test opposé.

Ils possèdent plusieurs attributs :

Attribut	Rôle
name	nom de la variable à tester si l'attribut property n'est pas précisé sinon c'est le nom de l'entité à tester (attribut obligatoire)
property	Nom de la propriété de la variable à tester
scope	Portée contenant la variable

82.4.3.2. Les tags <logic:equal> et <logic:notEqual>

Le tag <logic:equal> permet de tester l'égalité entre une variable et une valeur. Le tag <logic:notEqual> permet de faire le test opposé.

Ils possèdent plusieurs attributs :

Attribut	Rôle
Value	contient la valeur : celle-ci peut être une constante ou être déterminée dynamiquement par exemple avec le tag JSP <%= ... %> (attribut obligatoire)
cookie	nom du cookie dont la valeur doit être testée
header	nom de l'attribut de l'en-tête http dont la valeur doit être testée
name	nom de la variable dont la valeur doit être testée
property	nom de la propriété de la variable à tester
parameter	nom du paramètre http dont la valeur doit être testée
scope	portée contenant la variable

Exemple :

```
    <% int valeurReference = 123456; %>
  ...
    <logic:equal name="monbean"
    property="valeur"
    value="<%= valeurReference %>">
    <p>Les valeurs sont égales</p>
    </logic:equal>
```

82.4.3.3. Les tags <logic:lessEqual>, <logic:lessThan>, <logic:greaterEqual>, et <logic:greaterThan>

Ils sont similaires au tag <logic:equal> mais permettent respectivement de tester les conditions inférieur ou égal, strictement inférieur, supérieur ou égal et strictement supérieur.

82.4.3.4. Les tags <logic:match> et <logic:notMatch>

Le tag <logic:match> permet de tester si une valeur est contenue dans une variable. Le tag <logic:notMatch> permet de faire le test opposé.

Ils possèdent plusieurs attributs :

Attribut	Rôle
value	contient la valeur : celle-ci peut être une constante ou déterminée dynamiquement par exemple avec le tag JSP <%= ... %> (attribut obligatoire)
cookie	nom du cookie dont la valeur doit être testée
header	nom de l'attribut de l'en-tête http dont la valeur doit être testée
name	nom de la variable dont la valeur doit être testée
property	nom de la propriété de la variable à tester
parameter	nom du paramètre http dont la valeur doit être testée
scope	portée contenant la variable
location	permet de préciser la localisation de la valeur à rechercher dans la variable. Les valeurs possibles sont start et end pour une recherche respectivement en début et en fin. Sans préciser cet attribut, la recherche se fait dans toute la variable

82.4.3.5. Les tags <logic:present> et <logic:notPresent>

Le tag <logic:present> permet de tester l'existence d'une entité dans une portée donnée. Le tag <logic:notPresent> permet de faire le test opposé.

Ils possèdent plusieurs attributs :

Attribut	Rôle
cookie	nom du cookie dont la valeur doit être testée
header	nom de l'attribut de l'en-tête http dont la valeur doit être testée
name	nom de la variable dont la valeur doit être testée
property	nom de la propriété de la variable à tester
parameter	nom du paramètre http dont la valeur doit être testée
scope	portée contenant la variable

82.4.3.6. Le tag <logic:forward>

Le tag <logic:forward> permet de transférer le traitement de la requête vers une page définie dans les redirections globales de l'application.

Il ne possède qu'un seul attribut name qui permet de préciser le nom de la redirection globale définie dans le fichier de configuration struts-config.xml

Exemple : dans une JSP

```
<logic:forward name=">strong<login>/strong<" />
```

Exemple : dans le fichier de configuration

```
  <global-forwards>
    <forward name=">strong<login>/strong<" path="/login.jsp"/>
  </global-forwards>
```

82.4.3.7. Le tag <logic:redirect>

Le tag <logic:redirect> permet de rediriger l'affichage vers une autre page en utilisant la méthode HttpServletResponse.sendRedirect().

Il possède plusieurs attributs :

Attribut	Rôle
forward	nom de la redirection globale définie dans le fichier de configuration struts-config.xml à utiliser
href	URL de la ressource à utiliser
page	URL de la ressource relative au contexte de l'application (doit obligatoirement commencer par /)
name	collection de type Map qui contient les paramètres à passer à la ressource
paramId	nom de l'unique paramètre passé à la ressource
paramName	nom d'une variable dont la valeur sera utilisée comme valeur du paramètre
paramProperty	propriété de la variable paramName dont la valeur sera utilisée comme valeur du paramètre

L'avantage de ce tag est de permettre de modifier les paramètres fournis à la ressource.

82.4.3.8. Le tag <logic:iterate>

Ce tag permet de réaliser une itération sur une collection d'objets. Le corps du tag sera évalué pour chaque occurrence de l'itération.

Il possède plusieurs attributs :

Attribut	Rôle
id	nom de la variable qui va contenir l'occurrence courante de l'itération (attribut obligatoire)
name	nom de la variable qui contient la collection à parcourir
property	nom de la propriété de la variable name qui contient la collection à parcourir
scope	portée de la variable qui contient la collection
type	type pleinement qualifié des occurrences de la collection
indexId	nom de la variable qui va contenir l'index de l'occurrence courante
length	nombre maximum d'occurrences à traiter. Par défaut toute la collection est parcourue
offset	index de la première occurrence de l'itération. Par défaut c'est la première occurrence de la collection

82.5. La validation de données

La méthode validate() de la classe ActionForm permet de réaliser une validation des données fournies dans la requête.

Elle est appelée par l'ActionServlet lorsque l'attribut validate est positionné à true dans le tag <action>.

Exemple :

```
<action  path="/validerproduit"
         type="test.struts.controleur.ValiderProduitAction"
        name="saisirProduitForm"
        validate="true">
  <forward name="succes"  path="/listeproduit.jsp"/>
  <forward name="echec"  path="/saisirproduit.jsp"/>
</action>
```

Pour définir ses propres validations, il faut redéfinir la méthode validate() pour y coder les règles de validation. Si une erreur est détectée lors de l'exécution de ces règles, il faut instancier un objet de type ActionError et l'ajouter à l'objet ActionErrors retourné par la méthode validate(). Cet ajout se fait en utilisant la méthode add().

82.5.1. La classe ActionError

Cette classe encapsule une erreur survenue lors de la validation des données. C'est dans la méthode validate() de la classe ActionForm que les traitements doivent créer des instances de cette classe.

Le constructeur de cette classe attend en paramètre une chaîne de caractères qui précise le nom d'une clé du message d'erreur correspondant au message de l'erreur défini dans le fichier ressource bundle de l'application.

La méthode validate() de la classe ActionForm possède deux surcharges :

public ActionErrors validate(ActionMapping mapping, javax.servlet.http.HttpServletRequest request)

public ActionErrors validate(ActionMapping mapping, javax.servlet.ServletRequest request)

La première version est essentiellement mise en oeuvre car elle est utilisée pour les applications web.

Elle renvoie un objet de type ActionErrors qui va contenir les éventuelles erreurs détectées lors de la validation. Si la collection est vide ou nulle cela précise que la validation a réussie. Ceci permet à l'ActionServlet de savoir si elle va pouvoir appeler la méthode execute() de l'Action.

Par défaut, la méthode validate() de la classe ActionForm renvoie systématiquement null. Il est donc nécessaire de sous-classer la classe ActionForm et de redéfinir la méthode validate().

Exemple :

```
  public ActionErrors validate(ActionMapping  mapping, HttpServletRequest request) {
    ActionErrors errors = new ActionErrors();
    if ((nomUtilisateur == null) ||  (nomUtilisateur.length() == 0))
      errors.add("nomUtilisateur",  new ActionError("erreur.nomutilisateur.obligatoire"));
    if ((mdpUtilisateur == null) ||  (mdpUtilisateur.length() == 0))
      errors.add("mdpUtilisateur",  new ActionError("erreur.mdputilisateur.obligatoire"));
    return errors;
  }
```

Ce mécanisme peut aussi être mis en oeuvre dans la méthode execute() de la classe Action.

82.5.2. La classe ActionErrors

Cette classe encapsule une collection de type HashMap d'objets ActionError générés lors d'une validation.

C'est la méthode validate() de la classe ActionForm qui renvoie une instance de cette classe. Les traitements qu'elle contient se chargent de créer une instance de cette classe et d'utiliser la méthode add() pour ajouter des instances de la classe ActionError pour chaque erreur rencontrée.

Il est aussi possible de définir des erreurs dans la méthode : il faut créer un objet de type ActionErrors, utiliser sa méthode add() pour chaque erreur à ajouter et appeler la méthode saveErrors de la classe Action pour sauvegarder les erreurs.

Exemple :

```
    public ActionForward execute(
      ActionMapping       mapping,
      ActionForm          form,
      HttpServletRequest  request,
      HttpServletResponse response) throws  Exception {
      DynaActionForm daf       = (DynaActionForm) form;
      ActionForward  resultat  =  mapping.findForward("succes");
      String         reference = (String)  daf.get("reference");
      String         libelle   = (String) daf.get("libelle");
      int            prix      = Integer.parseInt((String)  daf.get("prix"));

      System.out.println("reference=" +  reference);
      System.out.println("libelle=" +  libelle);
      System.out.println("prix=" +  prix);

      if ((reference == null) ||  (reference.equals(""))) {
        ActionErrors errors = new ActionErrors();
        errors.add("reference", new  ActionError("app.saisirproduit.erreur.reference"));
        saveErrors(request, errors);

        resultat =  mapping.findForward("echec");
      }
      return resultat;
    }
```

Remarque : dans cet exemple, la validation des données est effectuée dans la méthode execute. Il est préférable d'effectuer cette tâche grâce à une des fonctionnalités proposées par Struts (validation par l'ActionForm ou le plug-in

Validator).

82.5.3. L'affichage des messages d'erreur

Le tag <html:errors> permet d'afficher les erreurs contenues dans l'instance courante de la classe ActionErrors.

Exemple :

```
<html:errors/>
```

Le plus simple est d'utiliser ce tag en début du corps de la page. Il se charge d'afficher toutes les erreurs (les erreurs globales et celles dédiées à un élément du formulaire) pour permettre leur gestion de façon globale.

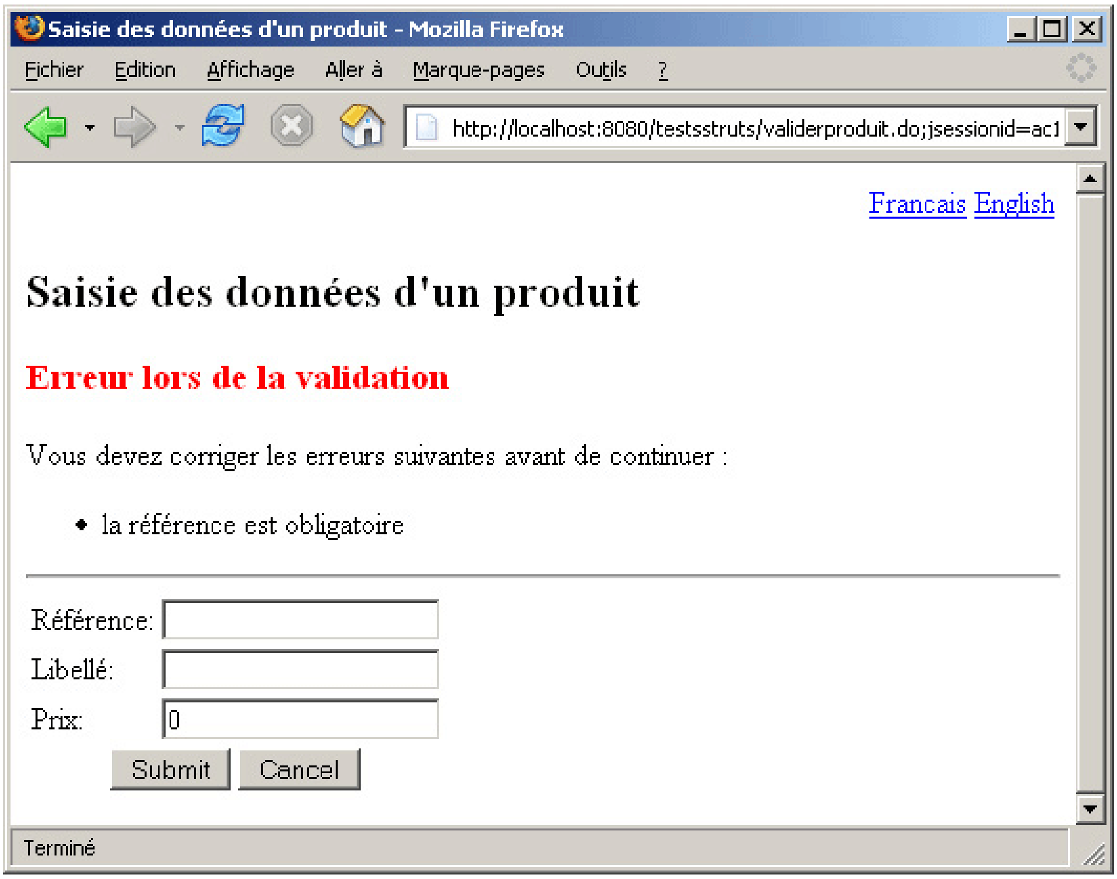
Ce tag recherche dans les ResourceBundle les deux clés errors.header et errors.footer dont les valeurs seront affichées avant les messages. A partir de la version 1.1 de Struts, les clés errors.prefix et error.suffix sont recherchées dans les ResourceBundles et ajoutées respectivement avant et après chaque message.

Exemple :

```
errors.prefix=<li>
errors.suffix=</li>
errors.header=<h3><font  color\="red">Erreur lors de la  validation</font></h3>
Vous devez corriger les erreurs suivantes avant de continuer \:<ul>
errors.footer=</ul><hr>
```

L'utilisation de tags HTML dans les ResourceBundle peut paraître choquante mais c'est la solution utilisée par Struts.

Exemple :


Saisie des données d'un produit - Mozilla Firefox
Fichier
Edition
Affichage
Aller à
Marque-pages
Outils
?
http://localhost:8080/testsstruts/validerproduit.do;jsessionid=ac1
Francais English

Saisie des données d'un produit

Erreur lors de la validation

Vous devez corriger les erreurs suivantes avant de continuer :

- la référence est obligatoire

Référence:
Libellé:
Prix: 0

Avec Struts 1.1, il est aussi possible d'utiliser le tag <html:errors> pour afficher des messages d'erreurs liés à un composant du formulaire. Dans ce cas, l'approche est légèrement différente.

L'exemple ci-dessous va afficher un message personnalisé pour un composant et un message d'erreur général.

Exemple : ApplicationResources.properties

```
...
app.saisirproduit.erreur.reference=la référence saisie est erronée
app.saisirproduit.erreur.libelle=le libelle saisi est erroné
app.saisirproduit.erreur.globale=une ou plusieurs erreurs sont survenues

errors.prefix=
errors.suffix=
errors.header=
errors.footer=
...
```

Comme les clés préfixées par errors sont utilisées pour chaque affichage d'erreur, leur contenu est laissé vide.

L'action instancie des objets de type ActionError si une erreur est détectée sur les données et l'associe au composant correspondant. Lors de l'ajout d'une erreur, il faut préciser l'identifiant du composant correspondant à son attribut property dans le tag de la page.

Si au moins une erreur est détectée sur une donnée alors une erreur globale est ajoutée à la liste des erreurs. Pour cela, il faut utiliser la constante ActionErrors.GLOBAL_ERROR lors de l'ajout de l'erreur dans la collection ActionErrors.

Exemple Struts 1.1 :

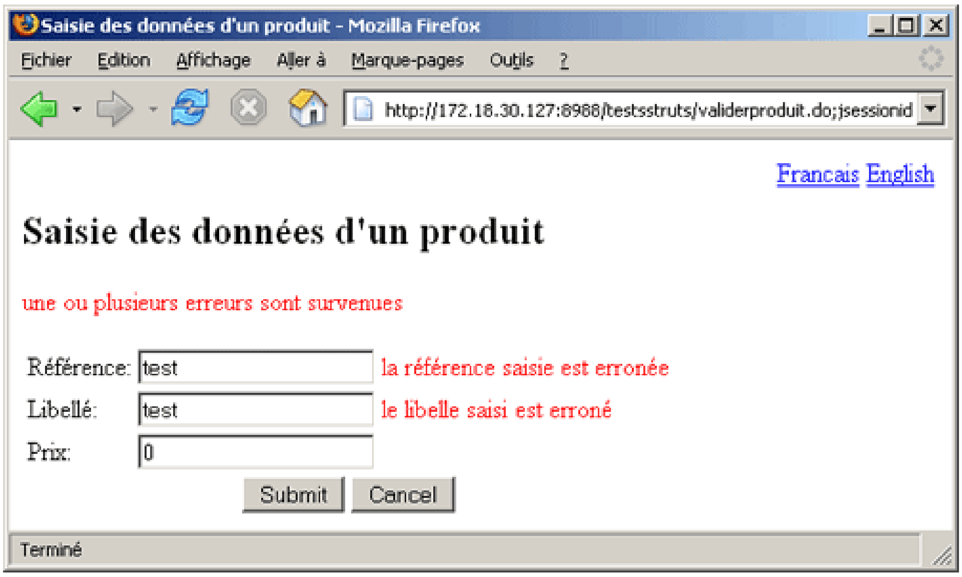
```
  ...
  public  ActionForward execute(
     ActionMapping       mapping,
     ActionForm          form,
     HttpServletRequest  request,
     HttpServletResponse response) throws Exception {
    DynaActionForm daf       = (DynaActionForm) form;
    ActionForward  resultat  = mapping.findForward("succes");
    ActionErrors   errors    = new ActionErrors();
    String         reference = (String) daf.get("reference");
    String         libelle   = (String) daf.get("libelle");
    int            prix      = Integer.parseInt((String)  daf.get("prix"));

    if (reference.equals("test")) {
      errors.add("reference", new  ActionError("app.saisirproduit.erreur.reference"));
    }
    if (libelle.equals("test")) {
      errors.add("libelle", new  ActionError("app.saisirproduit.erreur.libelle"));
    }

    if (!errors.isEmpty())
    {
      errors.add(ActionErrors.GLOBAL_ERROR,
        new  ActionError("app.saisirproduit.erreur.globale"));
      saveErrors(request, errors);
      resultat = mapping.findForward("echec");
    }

    return resultat;
  }
  ...
```

Il ne reste plus qu'à assurer l'affichage des messages d'erreurs dans la page. Pour le message associé à un composant il faut utiliser l'attribut property du tag <html:errors> en précisant comme valeur le nom du composant dont les messages doivent être affichés.

Pour afficher les messages d'erreurs globaux, il faut préciser dans l'attribut property la valeur de la constante ActionErrors.GLOBAL_ERROR.

Exemple Struts 1.1 :

```
<%@ taglib  uri="/WEB-INF/struts-bean.tld" prefix="bean"%>
<%@ taglib  uri="/WEB-INF/struts-html.tld" prefix="html"%>
<%@ taglib  uri="/WEB-INF/struts-logic.tld" prefix="logic"%>
<%@ page  contentType="text/html;charset=windows-1252"%>
<%@ page  import  ="org.apache.struts.action.*"    %>
<html:html locale="true">
  <head>
    <meta http-equiv="Content-Type" content="text/html;  charset=windows-1252"/>
    <title>
      <bean:message key="app.saisirproduit.titre"/>
    </title>
  </head>
  <body>
    <table width="100%">
      <tr>
        <td align="right">
          <html:link  href="changerlangue.do?langue=fr">Francais</html:link>
          <html:link  href="changerlangue.do?langue=en">English</html:link>
        </td>
      </tr>
    </table>
    <h2>
      <bean:message key="app.saisirproduit.titre"/>
    </h2>
    <html:form action="validerproduit.do"  focusIndex="reference">
      <logic:present name="<%=Action.ERROR_KEY%>">
      <P style="color:red;"><html:errors  property="<%=ActionErrors.GLOBAL_ERROR%>"/></P>
      </logic:present>
       <table>
        <tr>
          <td>
            <bean:message  key="app.saisirproduit.libelle.reference"/>:
         </td>
          <td>
            <html:text property="reference"/>
          </td>
          <td style="color:red;"><html:errors  property="reference"/></td>
        </tr>
        <tr>
          <td>
            <bean:message key="app.saisirproduit.libelle.libelle"/>:
          </td>
          <td>
            <html:text property="libelle"/>
          </td>
          <td style="color:red;"><html:errors  property="libelle"/></td>
        </tr>
        <tr>
          <td>
            <bean:message key="app.saisirproduit.libelle.prix"/>:
          </td>
          <td>
            <html:text property="prix"/>
          </td>
          <td></td>
        </tr>
        <tr>
          <td colspan="3" align="center">
            <html:submit/>
            <html:cancel/>
          </td>
        </tr>
      </table>
    </html:form>
  </body>
</html:html>
```



Saisie des données d'un produit - Mozilla Firefox
Fichier Edition Affichage Aller à Marque-pages Outils ?
http://172.16.30.127:8988/testsstruts/validerproduit.do;jsessionid
Francais English
Submit
Cancel
Terminé

Saisie des données d'un produit

une ou plusieurs erreurs sont survenues

Référence: test la référence saisie est erronée

Libellé: test le libelle saisi est erroné

Prix: 0

82.5.4. Les classes ActionMessage et ActionMessages

La classe ActionMessage, apparue avec Struts 1.1, fonctionne de la même façon que la classe ActionError mais elle encapsule des messages d'information qui ne sont pas des erreurs.

Ce type de message est pratique notamment pour afficher des messages de confirmation ou d'information aux utilisateurs.

La classe ActionMessages encapsule une collection d'ActionMessage.

Exemple :

```
      ActionMessages  actionMessages = new ActionMessages();
      actionMessages.add(ActionMessages.GLOBAL_MESSAGE,
        new  ActionMessage("liste.incomplete"));
      saveMessages(request,actionMessages);
```

La méthode add() permet d'ajouter des messages dans la collection.

La méthode clear() permet de supprimer tous les messages de la collections.

La méthode isEmpty() permet de savoir si la collection est vide et la méthode size() permet de connaître le nombre de messages stockés dans la collection.

82.5.5. L'affichage des messages

Le tag <html:messages> permet d'afficher les messages contenus dans l'instance courante de la classe ActionMessages.

Exemple :

```
     <logic:messagesPresent message="true">
      <html:messages  id="message" message="true">
        <bean:write  name="message"/>
       </html:messages>
     </logic:messagesPresent>
```

83. JSF (Java Server Faces)

Chapitre 83

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

83.1. La présentation de JSF

Les technologies permettant de développer des applications web avec Java ne cessent d'évoluer :

1. Servlets
2. JSP
3. MVC Model 1 : servlets + JSP
4. MVC Model 2 : un seule servlet + JSP
5. Java Server Faces

Java Server Faces (JSF) est une technologie dont le but est de proposer un framework qui facilite et standardise le développement d'applications web avec Java. Son développement a tenu compte des différentes expériences acquises lors de l'utilisation des technologies standard pour le développement d'applications web (servlet, JSP, JSTL) et de différents frameworks (Struts, ...).

Le grand intérêt de JSF est de proposer un framework qui puisse être mis en oeuvre par des outils pour permettre un développement de type RAD pour les applications web et ainsi faciliter le développement des applications de ce type. Ce type de développement était déjà courant pour des applications standalone ou clients/serveurs lourds avec des outils tels que Delphi de Borland, Visual Basic de Microsoft ou Swing avec Java.

Ce concept n'est pourtant pas nouveau dans les applications web puisqu'il est déjà mis en oeuvre par WebObject d'Apple et plus récemment par ASP.Net de Microsoft mais cette mise en oeuvre à grande échelle fût relativement tardive. L'adoption du RAD pour le développement web trouve notamment sa justification dans le coût élevé de développement de l'IHM à la « main » et souvent par copier/coller d'un mélange de plusieurs technologies (HTML, JavaScript, ...), rendant fastidieux et peu fiable le développement de ces applications.

Plusieurs outils commerciaux intègrent déjà l'utilisation de JSF notamment Studio Creator de Sun, WSAD d'IBM, JBuilder de Borland, JDevelopper d'Oracle, ...

Même si JSF peut être utilisé par codage à la main, l'utilisation d'un outil est fortement recommandée pour pouvoir mettre en oeuvre rapidement toute la puissance de JSF.

Ainsi de par sa complexité et sa puissance, JSF s'adapte parfaitement au développement d'applications web complexes en facilitant leur écriture.

Les pages officielles de cette technologie sont à l'URL :
https://www.oracle.com/java/technologies/javaserverfaces.html.

La version 1.0 de Java Server Faces, développée sous la JSR-127, a été validée en mars 2004.

JSF est une technologie utilisée côté serveur dont le but est de faciliter le développement de l'interface utilisateur en séparant clairement la partie « interface » de la partie « métier » d'autant que la partie interface n'est souvent pas la plus compliquée mais la plus fastidieuse à réaliser.

Cette séparation avait déjà été initiée avec la technologie JSP et particulièrement les bibliothèques de tags personnalisés. Mais JSF va encore plus loin en reposant sur le modèle MVC et en proposant de mettre en oeuvre :

- l'assemblage de composants serveurs qui génèrent le code de leur rendu avec la possibilité d'associer certains composants à une source de données encapsulée dans un bean
- l'utilisation d'un modèle de développement standardisé reposant sur l'utilisation d'événements et de listeners
- la conversion et la validation des données avant leur utilisation dans les traitements
- la gestion de l'état des composants de l'interface graphique
- la possibilité d'étendre les différents modèles et de créer ses propres composants
- la configuration de la navigation entre les pages
- le support de l'internationalisation
- le support pour l'utilisation par des outils graphiques du framework afin de faciliter sa mise en oeuvre

JSF se compose :

- d'une spécification qui définit le mode de fonctionnement du framework et une API : l'ensemble des classes de l'API est contenu dans les packages javax.faces.
- d'une implémentation de référence
- de bibliothèques de tags personnalisés fournies par l'implémentation pour utiliser les composants dans les JSP, gérer les événements, valider les données saisies, ...

Le rendu des composants ne se limite pas à une seule technologie même si l'implémentation de référence ne propose qu'un rendu des composants en HTML.

Le traitement d'une requête gérée par une application utilisant JSF suit un cycle de vie particulier constitué de plusieurs étapes :

- création de l'arbre de composants
- extraction des données des différents composants de la page
- conversion et validation des données
- extraction des données validées et mise à jour du modèle de données (javabean)
- traitements des événements liés à la page
- génération du rendu de la réponse

Ces différentes étapes sont transparentes lors d'une utilisation standard de JSF.

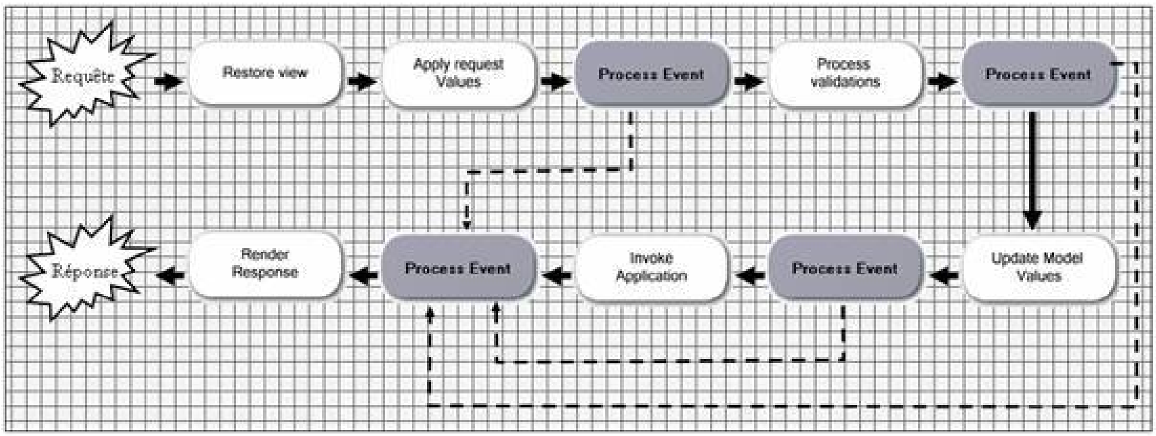
Ce chapitre contient plusieurs sections :

♦ La présentation de JSF
♦ Le cycle de vie d'une requête
♦ Les implémentations
♦ Le contenu d'une application
♦ La configuration de l'application
♦ Les beans
♦ Les composants pour les interfaces graphiques
♦ La bibliothèque de tags Core
♦ La bibliothèque de tags Html
♦ La gestion et le stockage des données
♦ La conversion des données
♦ La validation des données
♦ La sauvegarde et la restauration de l'état
♦ Le système de navigation
♦ La gestion des événements
♦ Le déploiement d'une application
♦ Un exemple d'application simple
♦ L'internationalisation
♦ Les points faibles de JSF

83.2. Le cycle de vie d'une requête

JSF utilise la notion de vue (view) qui est composée d'une arborescence ordonnée de composants inclus dans la page.

Les requêtes sont prises en charge et gérées par le contrôleur d'une application JSF (en général une servlet). Celle-ci va assurer la mise en oeuvre d'un cycle de vie des traitements en vue d'envoyer une réponse au client.



JSF propose pour chaque page un cycle de vie pour traiter la requête HTTP et générer la réponse. Ce cycle de vie est composé de plusieurs étapes :

1. Restore view ou Reconstruct Component Tree : cette première phase permet au serveur de recréer l'arborescence des composants qui composent la page. Cette arborescence est stockée dans un objet de type FacesContext et sera utilisée tout au long du traitement de la requête.
2. Apply Request Value : dans cette étape, les valeurs des données sont extraites de la requête HTTP pour chaque composant et sont stockées dans leur composant respectif dans le FaceContext. Durant cette phase des opérations de conversions sont réalisées pour permettre de transformer les valeurs stockées sous forme de chaînes de caractères dans la requête http en un type utilisé pour le stockage des données.
3. Perform validations : une fois les données extraites et converties, il est possible de procéder à leur validation en appliquant les validators enregistrés auprès de chaque composant. Les éventuelles erreurs de conversions sont stockées dans le FaceContext. Dans ce cas, l'étape suivante est directement « Render Response » pour permettre de réafficher la page avec les valeurs saisies et afficher les erreurs
4. Synchronize Model ou update model values : cette étape permet de stocker dans les composants du FaceContext leurs valeurs locales validées respectives. Les éventuelles erreurs de conversions sont stockées dans le FaceContext. Dans ce cas, l'étape suivante est directement « Render Response » pour permettre de réafficher la page avec les valeurs saisies et d' afficher les erreurs
5. Invoke Application Logic : dans cette étape, le ou les événements émis dans la page sont traités. Cette phase doit permettre de déterminer la page résultat qui sera renvoyée dans la réponse en utilisant les règles de navigation définies dans l'application. L'arborescence des composants de cette page est créée.
6. Render Response : cette étape se charge de créer le rendu de la page de la réponse.

83.3. Les implémentations

Java Server Faces est une spécification : il est donc nécessaire d'obtenir une implémentation de la part d'un tiers.

Plusieurs implémentations commerciales ou libres sont disponibles, notamment l'implémentation de référence de Sun et MyFaces qui est devenu un projet du groupe Apache.

83.3.1. L'implémentation de référence

Comme pour toute JSR validée, Sun propose une implémentation de référence des spécifications de la JSR, qui est la plus complète possible.

Plusieurs versions de l'implémentation de référence de Sun sont proposées :

Version	Date de diffusion
1.0	Mars 2004
1.1	Mai 2004
1.1_01	Septembre 2004

La solution la plus simple pour utiliser l'implémentation de référence est d'installer le JWSDK 1.3 qui est fourni en standard avec l'implémentation de référence de JSF. La version de JSF fournie avec le JWSDK 1.3 est la 1.0.

Pour utiliser la version 1.1, il faut supprimer le répertoire jsf dans le répertoire d'installation de JWSDK, télécharger l'implémentation de référence, décompresser son contenu dans le répertoire d'installation de JWSDK et renommer le répertoire jsf-1_1_01 en jsf.

Il est aussi possible de télécharger l'implémentation de référence sur le site de Sun et de l'installer « manuellement » dans un conteneur web tel que Tomcat. Cette procédure sera détaillée dans une des sections suivantes.

Pour cela, il faut télécharger le fichier jsf-1_1_01.zip et le décompresser dans un répertoire du système. L'archive contient les bibliothèques de l'implémentation, la documentation des API et des exemples.

Les exemples de ce chapitre vont utiliser cette version 1.1 de l'implémentation de référence des JSF.

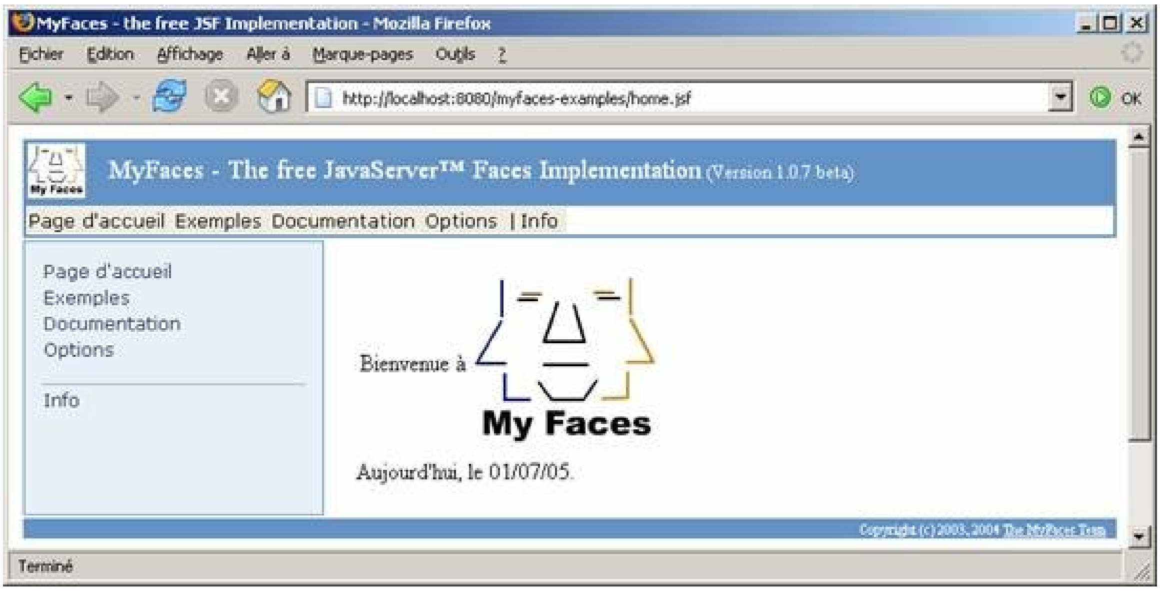
83.3.2. MyFaces

MyFaces est une implémentation libre des Java Server Faces qui est devenue un projet du groupe Apache.

Elle propose plusieurs composants spécifiques en plus de ceux imposés par les spécifications JSF.

Le site de MyFaces est à l'URL : https://myfaces.apache.org/

Il faut télécharger le fichier et le décompresser dans un répertoire du système. Il suffit alors de copier le fichier myfaces-examples.war dans le répertoire webapps de Tomcat, de relancer le serveur puis saisir l'URL http://localhost:8080/myfaces-examples


MyFaces - the free JSF Implementation - Mozilla Firefox
Fichier Edition Affichage Aller à Marque-pages Outils ?
http://localhost:8080/myfaces-examples/home.jsf
OK
MyFaces - The free JavaServer™ Faces Implementation (Version 1.0.7 beta)
Page d'accueil Exemples Documentation Options | Info
Page d'accueil
Exemples
Documentation
Options
Info
Copyright (c) 2003, 2004 The MyFaces Team
Terminé

Bienvenue à My Faces

Aujourd'hui, le 01/07/05.

Pour utiliser MyFaces dans ses propres applications, il faut réaliser plusieurs opérations.

Il faut copier les fichiers *.jar du répertoire lib de MyFaces et myfaces-jsf-api.jar dans le répertoire WEB-INF/lib de la webapp.

Dans chaque page qui va utiliser les composants de MyFaces, il faut déclarer la bibliothèque de tags dédiés.

Exemple :

```
<%@ taglib uri="http://myfaces.sourceforge.net/tld/myfaces_ext_0_9.tld" prefix="x"%>
```

83.4. Le contenu d'une application

Les applications utilisant JSF sont des applications web qui doivent respecter les spécifications de J2EE.

En tant que telles, elles doivent avoir la structure définie par J2EE pour toutes les applications web :

/
/WEB-INF
/WEB-INF/web.xml
/WEB-INF/lib
/WEB-INF/classes

Le fichier web.xml doit contenir au minimum certaines informations notamment, la servlet faisant office de contrôleur, le mapping des URL pour cette servlet et des paramètres.

Exemple :

```
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <display-name>Test JSF</display-name>
  <description>Application de tests avec JSF</description>
  <context-param>
    <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
    <param-value>client</param-value>
  </context-param>
  <!-- Faces Servlet -->
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup> 1 </load-on-startup>
  </servlet>
  <!-- Faces Servlet Mapping -->
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.jsf</url-pattern>
  </servlet-mapping>
</web-app>
```

Chaque implémentation nécessite un certain nombre de bibliothèques tierces pour son bon fonctionnement.

Par exemple, pour l'implémentation de référence, les bibliothèques suivantes sont nécessaires :

jsf-api.jar
jsf-ri.jar
jstl.jar
standard.jar
common-beanutils.jar
commons-digester.jar
commons-collections.jar
commons-logging.jar

Remarque : avec l'implémentation de référence, il n'y a aucun fichier .tld à copier car ils sont intégrés dans le fichier jsf-impl.jar.

Les fichiers nécessaires dépendent de l'implémentation utilisée.

Ces bibliothèques peuvent être mises à disposition de l'application selon plusieurs modes :

- incorporées dans le package de l'application dans le répertoire /WEB-INF/lib
- incluses dans le répertoire des bibliothèques partagées par les applications web des conteneurs web s'ils proposent une telle fonctionnalité. Par exemple avec Tomcat, il est possible de copier ces bibliothèques dans le répertoire shared/lib.

L'avantage de la première solution est de faciliter la portabilité de l'application sur différents conteneur web mais elle duplique ces fichiers si plusieurs applications utilisent JSF.

Les avantages et inconvénients de la première solution sont exactement à l'opposé de ceux de la seconde solution. Le choix de l'une ou l'autre est donc à faire en fonction du contexte de déploiement.

83.5. La configuration de l'application

Toute application utilisant JSF doit posséder au moins deux fichiers de configuration qui vont contenir les informations nécessaires à sa bonne exécution.

Le premier fichier est le descripteur de toute application web J2EE : le fichier web.xml contenu dans le répertoire WEB-INF.

Le second fichier est un fichier de configuration au format XML, particulier au paramétrage de JSF et nommé faces-config.xml.

83.5.1. Le fichier web.xml

Le fichier web.xml doit contenir au minimum certaines informations notamment, la servlet servant de contrôleur, le mapping des URLs pour cette servlet et des paramètres pour configurer JSF.

Exemple :

```
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <display-name>Test JSF</display-name>
  <description>Application de tests avec JSF</description>
  <context-param>
    <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
    <param-value>client</param-value>
  </context-param>

  <!-- Servlet servant de controleur-->
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup> 1 </load-on-startup>
  </servlet>

  <!-Le mapping de la servlet -->
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.faces</url-pattern>
  </servlet-mapping>
</web-app>
```

Le tag <servlet> permet de définir une servlet et plus particulièrement dans ce cas de préciser la servlet qui sera utilisée comme contrôleur dans l'application. Le plus simple est d'utiliser la servlet fournie avec l'implémentation de référence javax.faces.webapp.FacesServlet. Le tag <load-on-startup> avec comme valeur 1 permet de demander le chargement de cette servlet au lancement de l'application.

Le tag <servlet-mapping> permet de préciser le mapping des URLs qui seront traitées par la servlet. Ce mapping peut prendre deux formes :

- mapping par rapport à une extension : exemple <url-pattern>*.faces</url-pattern>.
- mapping par rapport à un préfixe : exemple <url-pattern>/faces/*</url-pattern>.

Les URL utilisées pour des pages mettant en oeuvre JSF doivent obligatoirement passer par cette servlet. Ces URLs peuvent être de deux formes selon le mapping défini.

Exemple :

- http://localhost:8080/nom_webapp/index.faces
- http://localhost:8080/nom_webapp/faces/index.jsp

Dans les deux cas, c'est la servlet utilisée comme contrôleur qui va déterminer le nom de la page JSP à utiliser.

Le paramètre de contexte javax.faces.STATE_SAVING_METHOD permet de préciser le mode d'échange de l'état de l'arbre des composants de la page. Deux valeurs sont possibles :

- client :
- server :

Il est possible d'utiliser l'extension .jsf pour les fichiers JSP utilisant JSF à condition de correctement configurer le fichier web.xml dans ce sens. Pour cela deux choses sont à faire :

- il faut demander le mapping des URL terminant par .jsf par la servlet

  ```
  <servlet-mapping>
  <servlet-name>jsp</servlet-name>
  <url-pattern>*.jsf</url-pattern>
  </servlet-mapping>
  ```

- il faut préciser à la servlet le suffix par défaut à utiliser

  ```
  <context-param>
  <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
  <param-value>.jsf</param-value>
  </context-param>
  ```

Le démarrage d'une application directement avec une page par défaut utilisant JSF ne fonctionne pas correctement. Il est préférable d'utiliser une page HTML qui va effectuer une redirection vers la page d'accueil de l'application

Exemple :

```
<html>
  <head>
    <meta http-equiv="Refresh" content= "0; URL=index.faces"/>
    <title>Demarrage de l'application</title>
  </head>
  <body>
    <p>D&eacute;marrage de l'application ...</p>
  </body>
</html>
```

Il suffit alors de préciser dans le fichier web.xml que cette page est la page par défaut de l'application.

Exemple :

```
...
<welcome-file-list>
  <welcome-file>index.htm</welcome-file>
</welcome-file-list>
...
```

83.5.2. Le fichier faces-config.xml

Le plus simple est de placer ce fichier dans le répertoire WEB-INF de l'application Web.

Il est aussi possible de préciser son emplacement dans un paramètre de contexte nommé javax.faces.application.CONFIG_FILES dans le fichier web.xml. Il est possible par ce biais de découper le fichier de configuration en plusieurs morceaux. Ceci est particulièrement intéressant pour de grosses applications car un seul fichier de configuration peut dans ce cas devenir très gros. Il suffit de préciser chacun des fichiers séparés par une virgule dans le tag <param-value>.

Exemple :

```
...
<context-param>
  <param-name>javax.faces.application.CONFIG_FILES</param-name>
  <param-value>
/WEB-INF/ma-faces-config.xml, /WEB-INF/navigation-faces.xml, /WEB-INF/beans-faces.xml
  </param-value>
</context-param>
...
```

Ce fichier au format XML permet de définir et de fournir des valeurs d'initialisation pour des ressources nécessaires à l'application utilisant JSF.

Ce fichier doit impérativement respecter la DTD proposée par les spécifications de JSF :

http://java.sun.com/dtd/web-facesconfig_1_0.dtd

Le tag racine du document XML est le tag <face-config>. Ce tag peut avoir plusieurs tags fils :

Tag	Rôle
application	permet de préciser ou de remplacer des éléments de l'application
factory	permet de remplacer des fabriques par des fabriques personnalisées de certaines ressources (FacesContextFactory, LifeCycleFactory, RenderKitFactory, ...)
component	définit un composant graphique personnalisé
converter	définit un convertisseur pour encoder/décoder les valeurs des composants graphiques (conversion de String en Object et vice versa)
managed-bean	définit un objet utilisé par un composant qui est automatiquement créé, initialisé et stocké dans une portée précisée
navigation-rule	définit les règles qui permettent de déterminer l'enchaînement des traitements de l'application
referenced-bean	
render-kit	définit un kit pour le rendu des composants graphiques
lifecycle	
validator	définit un validateur personnalisé de données saisies dans un composant graphique

Ces tags fils peuvent être utilisés 0 ou plusieurs fois dans le tag <face-config>.

Le tag <application> permet de préciser des informations sur les entités utilisées par l'internationalisation et/ou de remplacer des éléments de l'application.

Les éléments à remplacer peuvent être : ActionListener, NavigationHandler, ViewHandler, PropertyResolver, VariableResolver. Ceci n'est utile que si la version fournie dans l'implémentation ne correspond pas aux besoins et doit être personnalisée par l'écriture d'une classe dédiée.

Le tag fils <message-bundle> permet de préciser le nom de base des fichiers de ressources utiles à l'internationalisation.

Le tag <locale-config> permet de préciser les locales qui sont supportées par l'application. Il faut utiliser autant de tags fils <supported-locale> que de locales supportées. Le tag fil <default-locale> permet de préciser la locale par défaut.

Exemple :

```
...
<application>
  <message-bundle>fr.jmdoudoux.dej.jsf.monapp.bundles.Messages</message-bundle>
  <locale-config>
    <default-locale>fr</default-locale>
    <supported-locale>en</supported-locale>
  </locale-config>
</application>
...
```

83.6. Les beans

Les beans sont largement utilisés dans une application JSF notamment pour permettre l'échange de données entre les différentes entités et le traitement des événements.

Les beans sont des classes qui respectent une spécification particulière notamment la présence :

- de getters et de setters qui respectent une convention de nommage particulière pour les attributs
- un constructeur par défaut sans arguments

83.6.1. Les beans managés (managed bean)

Les beans managés sont des javabeans dont le cycle de vie va être contrôlé par le framework JSF en fonction des besoins et du paramétrage fourni dans le fichier de configuration.

Dans le fichier de configuration, chacun de ces beans doit être déclaré avec un tag <managed-bean>. Ce tag possède trois tags fils obligatoires :

- <managed-bean-name> : le nom attribué au bean (celui qui sera utilisé lors de son utilisation)
- <managed-bean-class> : le type pleinement qualifié de la classe du bean
- <managed-bean-scope> : précise la portée dans laquelle le bean sera stocké et donc utilisable

La portée peut prendre les valeurs suivantes :

- request : cette portée est limitée entre l'émission de la requête et l'envoi de la réponse. Les données stockées dans cette portée sont utilisables lors d'un transfert vers une autre page (forward). Elles sont perdues lors d'une redirection (redirect).
- session : cette portée permet la circulation de données entre plusieurs échanges avec un même client
- application : cette portée permet l'accès à des données pour toutes les pages d'une même application quelque soit l'utilisateur

Exemple :

```
...
<managed-bean>
  <managed-bean-name>login</managed-bean-name>
  <managed-bean-class>fr.jmdoudoux.dej.jsf.LoginBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
...
```

Il est possible de fournir des valeurs par défaut aux propriétés en utilisant le tag <managed-property>. Ce tag possède deux tags fils :

- <property-name> : nom de la propriété du bean
- <value> : valeur à associer à la propriété

Exemple :

```
...
<managed-bean>
  <managed-bean-name>login</managed-bean-name>
  <managed-bean-class>fr.jmdoudoux.dej.jsf.LoginBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>nom</property-name>
    <value>test</value>
  </managed-property>
</managed-bean>
...
```

Lorsque le bean sera instancié, JSF appellera automatiquement les setters des propriétés identifiées dans des tags <managed-property> avec leurs valeurs <value> respectives.

Pour initialiser la propriété à null, il faut utiliser le tag <null-value>

Exemple :

```
...
<managed-property>
  <property-name>nom</property-name>
  <null-value>
</managed-property>
...
```

Ces informations seront utilisées par JSF pour automatiser la création ou la récupération d'un bean lorsque celui-ci sera utilisé dans l'application.

Le grand intérêt de ce mécanisme est de ne pas avoir à se soucier de l'instanciation du bean ou de sa recherche dans la portée puisque c'est le framework qui va s'en occuper de façon transparente.

83.6.2. Les expressions de liaison de données d'un bean

Il est toujours nécessaire dans la partie présentation d'obtenir la valeur d'une donnée d'un bean pour par exemple l'afficher.

JSF propose une syntaxe basée sur des expressions qui facilitent l'utilisation des valeurs d'un bean. Ces expressions doivent être délimitées par #{ et }.

Basiquement une expression est composée du nom du bean suivi du nom de la propriété désirée séparés par un point.

Exemple :

```
<h:inputText value="#{login.nom}"/>
```

Cet exemple affecte la valeur de l'attribut nom du bean login au composant de type saisie de texte. Dans ce cas précis, c'est aussi cet attribut de ce bean qui recevra la valeur saisie lorsque la page sera envoyée au serveur.

En fonction du contexte le résultat de l'évaluation peut conduire à l'utilisation du getter (par exemple pour afficher la valeur) ou du setter (pour affecter la valeur après un envoi de la page). C'est JSF qui le détermine en fonction du contexte.

La notation par point peut être remplacée par l'utilisation de crochets. Dans ce cas, le nom de la propriété doit être mis entre simples ou doubles quotes dans les crochets.

Exemple :

```
login.nom
login["nom"]
login['nom']
```

Ces trois expressions sont rigoureusement identiques. Cette syntaxe peut être plus pratique lors de la manipulation de collections mais elle est obligatoire lorsque la propriété contient un point.

Exemple :

```
msg["login.titre"]
```

L'utilisation des quotes simples ou doubles est équivalente car il faut les imbriquer par exemple lors de leur utilisation comme valeur de l'attribut d'un composant.

Exemple :

```
<h:inputText value='#{login["nom"]}'/>
<h:inputText value="#{login['nom']'}"/>
```

Attention, la syntaxe utilisée par JSF est proche mais différente de celle proposée par JSTL : JSF utilise le délimiteur #{ ... } et JSTL utilise le délimiteur ${ ... } .

JSF définit un ensemble de variables prédéfinies et utilisables dans les expressions de liaison de données :

Variable	Rôle
header	une collection de type Map encapsulant les éléments définis dans les paramètres de l'en-tête de la requête http (seule la première valeur est renvoyée)
header-values	une collection de type Map encapsulant les éléments définis dans les paramètres de l'en-tête de la requête http (toutes les valeurs sont renvoyées sous la forme d'un tableau)
param	une collection de type Map encapsulant les éléments définis dans les paramètres de la requête http (seule la première valeur est renvoyée)
param-values	une collection de type Map encapsulant les éléments définis dans les paramètres de la requête http (toutes les valeurs sont renvoyées sous la forme d'un tableau)
cookies	une collection de type Map encapsulant les éléments définis dans les cookies
initParam	une collection de type Map encapsulant les éléments définis dans les paramètres d'initialisation de l'application
requestScope	une collection de type Map encapsulant les éléments définis dans la portée request
sessionScope	une collection de type Map encapsulant les éléments définis dans la portée session
applicationScope	une collection de type Map encapsulant les éléments définis dans la portée application
facesContext	une instance de la classe FacesContext
View	une instance de la classe UIViewRoot qui encapsule la vue

Lorsque qu'une variable est utilisée dans une expression, JSF recherche dans la liste des variables prédéfinies, puis recherche une instance dans la portée request, puis dans la portée session et enfin dans la portée application. Si aucune instance n'est trouvée, alors JSF crée une nouvelle instance en tenant compte des informations du fichier de configuration. Cette instanciation est réalisée par un objet de type VariableResolver de l'application.

La syntaxe des expressions possède aussi quelques opérateurs :

Opérateurs	Rôle	Exemple
+ - * / % div mod	opérateurs arithmétiques	
< <= > >= == != lt le gt ge eq ne	opérateurs de comparaisons	
&& || ! and or not	opérateurs logiques	<h:inputText rendered="#{!monBean.affichable}" />
Empty	opérateur vide : un objet null, une chaîne vide, un tableau ou une collection sans élément	
? :	opérateur ternaire de test	

Il est possible de concaténer les résultats des évaluations de plusieurs expressions simplement en les plaçant les uns à la suite des autres.

Exemple :

```
<h:outputText value="#{messages.salutation}, #{utilisateur.nom}!"/>
```

Il est parfois nécessaire d'évaluer une expression dans le code des objets métiers pour obtenir sa valeur. Comme tous les composants sont stockés dans le FaceContext, il est possible d'accéder à cet objet pour obtenir les informations désirées. Il est d'abord nécessaire d'obtenir l'instance courante de l'objet FaceContext en utilisant la méthode statique getCurrentInstance().

Exemple :

```
FacesContext context = FacesContext.getCurrentInstance();
ValueBinding binding = context.getApplication().createValueBinding("#{login.nom}");
String nom = (String) binding.getValue(context);
```

83.6.3. Les Backing beans

Les beans de type backing bean sont spécialement utilisés avec JSF pour encapsuler tout ou partie des composants d'une page et ainsi faciliter leur accès notamment lors des traitements.

Ces beans sont particulièrement utiles durant des traitements réalisés lors de validations ou de gestion d'événements car ils permettent un accès aux composants dont ils possèdent une référence.

Exemple :

```
package fr.jmdoudoux.dej.jsf;

import javax.faces.component.UIInput;

public class LoginBean {

  private UIInput composantNom;
  private String nom;
  private String mdp;


  public UIInput getComposantNom() {
    return composantNom;
  }

  public void setComposantNom(UIInput input) {
    composantNom = input;
  }

  public String getNom() {
    return nom;
  }
...
}
```

Dans la vue, il est nécessaire de lier un composant avec son attribut correspondant dans le backing bean. L'attribut binding d'un composant permet de réaliser cette liaison.

Exemple :

```
<h:inputText value="#{login.nom}" binding="#{login.composantNom}" />
```

83.7. Les composants pour les interfaces graphiques

JSF propose un ensemble de composants serveurs pour faciliter le développement d'interfaces graphiques utilisateur.

Pour les composants, JSF propose :

- un ensemble de classes qui gèrent le comportement et l'état d'un composant
- un modèle pour assurer le rendu du composant pour un type d'application (par exemple HTML)
- un modèle de gestion des événements émis par le composant reposant sur le modèle des listeners
- la possibilité d'associer à un composant un composant de conversion de données ou de validation des données

Tous ces composants héritent de la classe abstraite UIComponentBase.

JSF propose 12 composants de base :

UICommand	Composant qui permet de réaliser une action émettant un événement
UIForm	Composant qui regroupe d'autres composants dont l'état sera renvoyé au serveur
UIGraphic	Composant qui représente une image
UIInput	Composant qui permet de saisir des données
UIOutput	Composant qui permet d'afficher des données
UIPanel	Composant qui regroupe d'autres composants à afficher sous la forme d'un tableau
UIParameter	
UISelectItem	Composant qui représente un élément sélectionné dans un ensemble
UISelectItems	Composant qui représente un ensemble d'éléments
UISelectBoolean	Composant qui permet de sélectionner parmi deux états
UISelectMany	Composant qui permet de sélectionner plusieurs éléments d'un ensemble
UISelectOne	Composant qui permet de sélectionner un seul élément d'un ensemble

Ces classes sont des javabeans qui définissent les fonctionnalités de base des composants permettant la saisie et la sélection de données.

Chacun de ces composants possède un type, un identifiant, une ou plusieurs valeurs locales et des attributs. Ils sont extensibles et il est même possible de créer ses propres composants.

Le comportement de ces composants repose sur le traitement d'événements respectant le modèle de gestion des événements de JSF.

Ces classes ne sont pas utilisées directement : elles sont utilisées par la bibliothèque de tags personnalisés qui se charge de les instancier et de leur associer le modèle de rendu adéquat.

Ces classes ne prennent pas en charge le rendu du composant. Par exemple, un objet de type UICommand peut être rendu en HTML sous la forme d'un lien hypertexte ou d'un bouton de formulaire.

83.7.1. Le modèle de rendu des composants

Pour chaque composant, il est possible de définir un ou plusieurs modèles qui se chargent du rendu de ce composant dans un contexte client particulier (par exemple HTML).

L'association entre un composant et son modèle de rendu est réalisée dans un RenderKit : il précise pour chaque composant le ou les modèles de rendu à utiliser. Par exemple, un objet de type UISelectOne peut être rendu sous la forme d'un ensemble de boutons radio, d'une liste ou d'une liste déroulante. Chacun de ces rendus est défini par un objet de type Renderer.

L'implémentation de référence propose un seul modèle de rendu pour les composants qui génèrent de l'HTML.

Ce modèle favorise la séparation entre l'état, le comportement d'un composant et sa représentation finale.

Le modèle de rendu permet de définir la représentation visuelle des composants. Chaque composant peut être rendu de plusieurs façons avec plusieurs modèles de rendu. Par exemple, un composant de type UICommand peut être rendu sous la forme d'un bouton ou d'un lien hypertexte. Le rendu peut être HTML mais il est possible d'utiliser un autre système de rendu comme XML ou WML.

Le modèle de rendu met un oeuvre un ou plusieurs kits de rendus.

83.7.2. L'utilisation de JSF dans une JSP

Pour une utilisation dans une JSP, l'implémentation de référence propose deux bibliothèques de tags personnalisés :

- core : cette bibliothèque contient des fonctionnalités de bases ne générant aucun rendu. L'utilisation de cette bibliothèque est obligatoire car elle contient notamment l'élément view
- html : cette bibliothèque se charge des composants avec un rendu en HTML

Pour utiliser ces deux bibliothèques, il est nécessaire d'utiliser une directive taglib pour chacune d'elles au début de la page JSP.

Exemple :

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
```

Le préfixe est libre mais par convention ce sont ceux fournis dans l'exemple qui sont utilisés.

Le tag <view> est obligatoire dans toutes pages utilisant JSF. Cet élément va contenir l'état de l'arborescence des composants de la page si l'application est configurée pour stocker l'état sur le client.

Le tag <form> génère un tag HTML form qui définit un formulaire.

Exemple :

```
<html>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view>
<head>
  <title>Application de tests avec JSF</title>
</head>
<body>
  <h:form>
  ...
  </h:form>
</body>
</f:view>
</html>
```

83.8. La bibliothèque de tags Core

Cette bibliothèque est composée de 18 tags.

Tag	Rôle
actionListener	ajouter un listener pour une action sur un composant
attribute	ajouter un attribut à un composant
convertDateTime	ajouter un convertisseur de type DateTime à un composant
convertNumber	ajouter un convertisseur de type numérique à un composant
facet	définit un élément particulier d'un composant
loadBundle	charger un fichier contenant les chaînes de caractères d'une locale dans une collection de type Map
param	ajouter un paramètre à un composant
selectitem	définir l'élément sélectionné dans un composant permettant de faire un choix
selectitems	définir les éléments sélectionnés dans un composant permettant de faire un choix
subview	définir une sous-vue
verbatim	ajouter un texte brut à la vue
view	définir une vue
validator	ajouter un validateur à un composant
validateDoubleRange	ajouter un validateur de type « plage de valeurs réelles » à un composant
validateLength	ajouter un validateur de type « taille de la valeur » à un composant
validateLongRange	ajouter un validateur de type « plage de valeurs entières » à un composant
valueChangeListener	ajouter un listener pour un changement de valeur sur un composant

La plupart de ces tags permettent d'ajouter des objets à un composant. Leur utilisation sera détaillée tout au long de ce chapitre.

83.8.1. Le tag <selectItem>

Ce tag représente un élément dans un composant qui peut en contenir plusieurs.

Les attributs de base sont les suivants :

Attribut	Rôle
itemValue	contient la valeur de l'élément
itemLabel	contient le libellé de l'élément
itemDescription	contient une description de l'élément (utilisé uniquement par les outils de développement)
itemDisabled	contient l'état de l'élément
binding	contient le nom d'une méthode qui renvoie un objet de type javax.faces.model.SelectItem
id	contient l'identifiant du composant
value	contient une expression qui désigne un objet de type javax.faces.model.SelectItem

Exemple :

```
<f:selectItem value="#{test.elementSelectionne}"/>
```

L'attribut value attend en paramètre une expression désignant une méthode qui renvoie un objet de type SelectItem. Cet objet encapsule l'objet de la liste qui sera sélectionné.

Exemple :

```
...
public SelectItem getElementSelectionne() {
  return new SelectItem("Element 1");
}
...
```

La classe SelectItem possède quatre constructeurs qui permettent de définir les différentes propriétés qui composent l'élément.

83.8.2. Le tag <selectItems>

Ce tag représente une collection d'éléments dans un composant qui peut en contenir plusieurs.

Ce tag est particulièrement utile car il évite d'utiliser autant de tags selectItem que d'éléments à définir.

Exemple :

```
...
<h:selectOneRadio>
  <f:selectItems value="#{test.listeElements}"/>
</h:selectOneRadio>
...
```

La collection d'objets de type SelectItem peut être soit une collection soit un tableau.

Exemple : avec un tableau d'objects de type SelectItem

```
package fr.jmdoudoux.dej.jsf;

import javax.faces.model.SelectItem;

public class TestBean {

  private SelectItem[] elements = {
    new SelectItem(new Integer(1), "Element 1"),
    new SelectItem(new Integer(2), "Element 2"),
    new SelectItem(new Integer(3), "Element 3"),
    new SelectItem(new Integer(4), "Element 4"),
  };

  public SelectItem[] getListeElements() {
    return elements;
  }

...
}
```

La collection peut être de type Map : dans ce cas le framework associe la clé de chaque occurrence à la propriété itemValue et la valeur à la propriété itemLabel

Exemple :

```
package fr.jmdoudoux.dej.jsf;

import java.util.HashMap;
import java.util.Map;

import javax.faces.model.SelectItem;

public class TestBean {

  private Map elements = null;

  public Map getListeElements() {
    if (elements == null) {
      elements = new HashMap();
      elements.put("Element 1", new Integer(1));
      elements.put("Element 2", new Integer(2));
      elements.put("Element 3", new Integer(3));
      elements.put("Element 4", new Integer(4));
    }
    return elements;
  }

  public SelectItem getElementSelectionne() {
    return new SelectItem("Element 1");
  }

  ...
}
```

83.8.3. Le tag <verbatim>

Ce tag permet d'insérer du texte dans la vue.

Son utilisation est obligatoire dans le corps des tags JSF pour insérer autre chose qu'un tag JSF. Par exemple, pour insérer un tag HTML dans le corps d'un tag JSF, il est obligatoire d'utiliser le tag <verbatim>.

Les tags suivants peuvent avoir un corps : commandLink, outputLink, panelGroup, panelGrid et dataTable.

Exemple :

```
<h:outputLink value="http://java.sun.com" title="Java">
  <f:verbatim>
    Site Java de Sun
  </f:verbatim>
</h:outputLink>
```

Il est possible d'utiliser le tag <outputText> à la place du tag <verbatim>.

83.8.4. Le tag <attribute>

Ce tag permet de fournir un attribut quelconque à un composant puisque chaque composant peut stocker des attributs arbitraires.

Ce tag possède deux attributs :

Attribut	Rôle
Name	nom de l'attribut
Value	valeur de l'attribut

Dans le code d'un composant, il est possible d'utiliser la méthode getAttributes() pour obtenir une collection de type Map des attributs du composant.

Ceci offre un mécanisme souple pour fournir des paramètres sans être obligé de créer un nouveau composant ou de modifier un composant existant en lui ajoutant un ou plusieurs attributs.

83.8.5. Le tag <facet>

Ce tag permet de définir des éléments particuliers d'un composant.

Il est par exemple utilisé pour définir les lignes d'en-tête et de pied de page des tableaux.

Ce tag possède plusieurs attributs :

Attribut	Rôle
Name	Permet de préciser le type de l'élément généré par le tag Les valeurs possibles sont header et footer

Exemple :

```
<h:column>
  <f:facet name="header">
    <h:outputText value="Nom" />
  </f:facet>
  <h:outputText value="#{personne.nom}"/>
</h:column>
```

83.9. La bibliothèque de tags Html

Cette bibliothèque est composée de 25 tags qui permettent la réalisation de l'interface graphique de l'application.

Tag	Rôle
form	le tag <form> HTML
commandButton	un bouton
commandLink	un lien qui agit comme un bouton
graphicImage	une image
inputHidden	une valeur non affichée
inputSecret	une zone de saisie de texte monoligne dont la valeur n'est pas lisible
inputText	une zone de saisie de texte monoligne

inputTextarea	une zone de saisie de texte multiligne
outputLink	un lien
outputFormat	du texte affiché avec des valeurs fournies en paramètre
outputText	du texte affiché
panelGrid	un tableau
panelGroup	un panneau permettant de regrouper plusieurs composants
selectBooleanCheckbox	une case à cocher
selectManyCheckbox	un ensemble de cases à cocher
selectManyListbox	une liste déroulante où plusieurs éléments sont sélectionnables
selectManyMenu	un menu où plusieurs éléments sont sélectionnables
selectOneListbox	une liste déroulante où un seul élément est sélectionnable
selectOneMenu	un menu où un seul élément est sélectionnable
selectOneRadio	un ensemble de boutons radio
dataTable	une grille proposant des fonctionnalités avancées
column	une colonne d'une grille
message	le message d'erreur lié à un composant
messages	les messages d'erreur liés à tous les composants

83.9.1. Les attributs communs

Ces tags possèdent des attributs communs pouvant être regroupés en trois catégories :

- les attributs de base
- les attributs liés à HTML
- les attributs liés à JavaScript

Chaque tag utilise ou non chacun de ces attributs.

Les attributs de base sont les suivants :

Attribut	Rôle
id	contient l'identifiant du composant
binding	permet l'association avec un backing bean
rendered	contient un booléen qui indique si le composant doit être affiché
styleClass	contient le nom d'une classe CSS à appliquer au composant
value	contient la valeur du composant
valueChangeListener	permet l'association à une méthode qui va traiter les changements de valeurs
converter	contient une classe de conversion des données de chaîne de caractères en objet et vice versa
validator	contient une classe de validation des données
required	contient un booléen qui indique si une valeur doit obligatoirement être saisie

L'attribut id est très important car il permet d'avoir accès :

- au tag dans le code de la vue par d'autres tags
  <h:inputText id="nom" required="true"/>

<h:message for="nom"/>

- au tag dans le code JavaScript de la vue
- dans le code Java des objets métiers.
  UIComponent component = event.getComponent().findComponent("nomComposant");

L'attribut binding permet d'associer le composant avec un champ d'une classe de type bean. Un tel bean est nommé backing bean dans une application JSF.

Exemple :

```
...
<h:inputText value="#{login.nom}" id="nom" required="true" binding="#{login.inputTextNom}"/>
..

...
import javax.faces.component.UIComponent;

public class LoginBean {
  private String nom;

  private UIComponent inputTextNom;

  public UIComponent getInputTextNom() {
    return inputTextNom;
  }

  public void setInputTextNom(UIComponent inputTextNom) {
    this.inputTextNom = inputTextNom;
  }
...
```

L'attribut value permet de préciser la valeur d'un tag. Cette valeur peut être fournie sous deux formes :

- en dur dans le code :
  <h:outputText value="Bonjour"/>
- en utilisant une expression de liaison de données :
  <h:inputText value="#{login.nom}"/>

L'attribut converter permet de préciser une classe qui va convertir la valeur d'un objet en chaîne de caractères et vice versa. L'utilisation de cet attribut est détaillée dans une des sections suivantes.

L'attribut validator permet de préciser une classe qui va réaliser des contrôles de validation sur la valeur saisie. L'utilisation de cet attribut est détaillée dans une des sections suivantes.

L'attribut styleClass permet de préciser le nom d'un style défini dans une feuille de style CSS qui sera appliqué au composant.

Exemple : le fichier monstyle.css

```
.titre {
color:red;
}
```

Dans la vue, il faut inclure la feuille de style dans la partie en-tête de la page HTML.

Exemple :

```
...
<link href="monstyle.css" rel="stylesheet" type="text/css"/>
...
```

```
<h:outputText value="#{msg.login_titre}" styleClass="titre"/>
...
```

L'attribut renderer permet de préciser si le composant sera affiché ou non dans la vue. La valeur de l'attribut peut être obtenue dynamiquement par l'utilisation du langage d'expression.

Exemple :

```
<h:panelGrid rendered='#{listepersonnes.nbOccurrences gt 0}'/>
```

Les principaux attributs liés à HTML sont les suivants :

Attribut	Rôle
accesskey	contient le raccourci clavier pour donner le focus au composant
alt	contient le texte alternatif pour les composants non textuels
border	contient la taille de la bordure en pixel
disabled	permet de désactiver le composant
maxlength	contient le nombre maximum de caractères saisis
readonly	permet de rendre une zone de saisie en lecture seule
rows	contient le nombre de lignes visibles pour une zone de saisie multiligne
shape	contient la définition d'une région
size	contient la taille de la zone de saisie
style	contient le style CSS à utiliser
target	contient le nom de la frame cible pour l'affichage de la page
title	contient le titre du composant généralement transformé en une bulle d'aide
width	contient la largeur du composant

Le rôle de la plupart de ces tags est identique à leurs homologues définis dans HTML 4.0.

L'attribut style permet de définir un style CSS qui sera appliqué au composant. Cet attribut contient directement la définition du style à la différence de l'attribut styleClass qui contient le nom d'une classe CSS définie dans une feuille de style. Il est préférable d'utiliser l'attribut styleClass plutôt que l'attribut style afin de faciliter la maintenance de la charte graphique.

Exemple :

```
<h:outputText value="#{login.nom}" style="color:red;"/>
```

Les attributs liés à JavaScript sont :

Attribut	Rôle
onblur	perte du focus
onchange	changement de la valeur
onclick	clic du bouton de la souris sur le composant
ondblclick	double-clic du bouton de la souris sur le composant
onfocus	réception du focus

onkeydown	une touche est enfoncée
onkeypress	appui sur une touche
onkeyup	une touche est relachée
onmousedown	le bouton de la souris est enfoncé
onmousemove	déplacement du curseur de la souris sur le composant
onmouseout	déplacement du cuseur de la souris hors du composant
onmouseover	passage de la souris au-dessus du composant
onmouseup	le bouton de la souris est relaché
onreset	réinitialisation du formulaire
onselect	sélection du texte dans une zone de saisie
onsubmit	soumission du formulaire

83.9.2. Le tag <form>

Ce tag représente un formulaire HTML.

Il possède les attributs suivants :

Attribut	Rôle
binding, id, rendered, styleClass	attributs communs de base
accept, acceptcharset, dir, enctype, lang, style, target, title	attributs communs liés à HTML
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onreset, onsubmit	attributs communs liés aux événements JavaScript

Il est préférable de définir explicitement l'attribut id pour permettre son exploitation notamment dans le code JavaScript, sinon un id est généré automatiquement.

Ceci est d'autant plus important que les id des composants intégrés dans le formulaire sont préfixés par l'id du formulaire suivi du caractère deux-points. Il faut tenir compte de ce point lors de l'utilisation de code JavaScript faisant référence à un composant.

83.9.3. Les tags <inputText>, <inputTextarea>, <inputSecret>

Ces trois tags permettent de générer des composants pour la saisie de données.

Les attributs de ces tags sont les suivants :

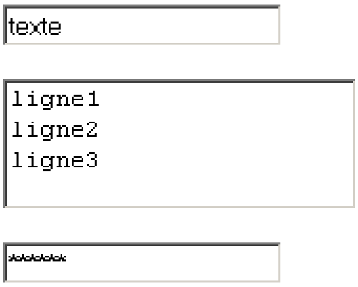
Attribut	Rôle
cols	définir le nombre de colonnes (pour le composant inputTextarea uniquement)
immediate	permettre de demander d'ignorer les étapes de validation des données
redisplay	permettre de réafficher le contenu lors du réaffichage de la page (pour le composant inputSecret uniquement)

required	rendre obligatoire la saisie d'une valeur
rows	définir le nombre de lignes affichées (pour le composant inputTextarea uniquement)
valueChangeListener	préciser une classe de type listener lors du changement de la valeur
binding, converter, id, rendered, required, styleClass, value, validator	attributs communs de base
accesskey, alt, dir, disabled,lang, maxlength, readonly, size, style, tabindex, title	attributs communs liés à HTML
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onselect	attributs communs liés aux événements JavaScript

Exemple :

```
<html>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view>
<head>
  <title>Saisie des données</title>
</head>
<body>
  <h:form>
    <h3>Saisie des données</h3>
    <p><h:inputText size="20" /></p>
    <p><h:inputTextarea rows="3" cols="20" /></p>
    <p><h:inputSecret size="20" /></p>
  </h:form>
</body>
</f:view>
</html>
```

Résultat



```
texte

ligne1
ligne2
ligne3

******
```

83.9.4. Le tag <ouputText> et <outputFormat>

Ces deux tags permettent d'insérer dans la vue une valeur sous la forme d'une chaîne de caractères. Par défaut, ils ne génèrent pas de tags HTML mais insèrent simplement la valeur dans la vue sauf si un style CSS est précisé avec l'attribut style ou styleClass. Dans ce cas, la valeur est contenue dans un tag HTML <span>.

Les attributs de ces deux tags sont :

Attribut	Rôle
escape	booléen qui précise si certains caractères de la valeur seront encodés ou non. La valeur par défaut est false.

binding, converter, id, rendered, styleClass, value	attributs communs de base
style, title	attributs communs liés à HTML

L'attribut escape est particulièrement utile pour encoder certains caractères spéciaux avec leur code HTML correspondant.

Exemple :

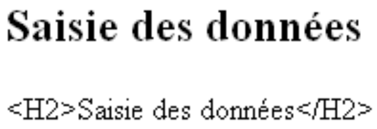
```
<h:outputText escape="true" value="Nombre d'occurrences > 200"/>
```

Le tag outputText peut être utilisé pour générer du code HTML en valorisant l'attribut escape à false.

Exemple :

```
<p><h:outputText escape="false" value="<H2>Saisie des données</H2>"/></p>

<p><h:outputText escape="true" value="<H2>Saisie des données</H2>"/></p>
```

Résultat :



Saisie des données

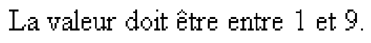
<H2>Saisie des données</H2>

Le tag outputFormat permet de formater une chaîne de caractères avec des valeurs fournies en paramètres.

Exemple :

```
<p>
  <h:outputFormat value="La valeur doit être entre {0} et {1}.">
    <f:param value="1"/>
    <f:param value="9"/>
  </h:outputFormat>
</p>
```

Résultat :



La valeur doit être entre 1 et 9.

Ce composant utilise la classe java.text.MessageFormat pour formater le message. L'attribut value doit donc contenir une chaîne de caractères utilisable par cette classe.

Le tag <param> permet de fournir la valeur de chacun des paramètres.

83.9.5. Le tag <graphicImage>

Ce composant représente une image : il génère un tag HTML <img>.

Les attributs de ce tag sont les suivants :

Attribut	Rôle

binding, id, rendered, styleClass, value	Attributs communs de base
alt, dir, height, ismap, lang, longdesc, style, title, url, usemap, width	Attributs communs liés à HTML
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup	Attributs communs liés aux événements JavaScript

Les attributs value et URL peuvent préciser l'URL de l'image.

Exemple :

```
<p><h:graphicImage value="/images/erreur.jpg" /></p>
<p><h:graphicImage url="/images/warning.jpg" /></p>
```

Résultat :

83.9.6. Le tag <inputHidden>

Ce composant représente un champ caché dans un formulaire.

Les attributs sont les suivants :

Attribut	Rôle
binding, converter, id, immediate, required, validator, value, valueChangeListener	attributs communs de base

Exemple :

```
<h:inputHidden value="#{login.nom}" />
```

Résultat dans le code HTML:

```
...
         <input type="hidden" name="_id0:_id12" value="test" />
...
```

83.9.7. Le tag <commandButton> et <commandLink>

Ces composants représentent respectivement un bouton de formulaire et un lien qui déclenche une action. L'action demandée sera traitée par le framework JSF.

Les attributs sont les suivants :

Attribut	Rôle
action	peut être une chaîne de caractères ou une méthode qui renvoie une chaîne de caractères qui sera traitée par le navigation handler.
actionListener	précise une méthode possédant une signature void nomMethode(ActionEvent) qui sera

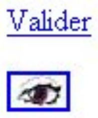
	exécutée lors d'un clic
image	URL tenant compte du contexte de l'application pour l'image qui sera utilisée à la place du bouton (uniquement pour le tag commandButton)
type	type de bouton généré : button, submit, reset (uniquement pour le tag commandButton)
value	le texte affiché par le bouton ou le lien
accesskey, alt, binding, id, lang, rendered, styleClass	attributs communs de base
coords (uniquement pour le tag commandLink), dir, disabled, hreflang (uniquement pour le tag commandLink), lang, readonly, rel (uniquement pour le tag commandLink), rev (uniquement pour le tag commandLink), shape (uniquement pour le tag commandLink), style, tabindex, target (uniquement pour le tag commandLink), title	attributs communs liés à HTML
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect	attributs communs liés aux événements JavaScript

Il est possible d'insérer dans le corps du tag <commandLink> d'autres composants qui feront partie intégrante du lien comme par exemple du texte ou une image.

Exemple :

```
<p>
  <h:commandLink>
    <h:outputText value="Valider"/>
  </h:commandLink>
</p>
<p>
  <h:commandLink>
    <h:graphicImage value="/images/oeil.jpg"/>
  </h:commandLink>
</p>
```

Résultat :


Valider

Il est aussi possible de fournir un ou plusieurs paramètres qui seront envoyés dans la requête en utilisant le tag <param> dans le corps du tag

Exemple :

```
<h:commandLink>
  <h:outputText value="Selectionner"/>
  <f:param name="id" value="1"/>
</h:commandLink>
```

Résultat dans le page HTML générée :

```
<a href="#" onclick="document.forms['_id0']['_id0:_idcl'].value='_id0:_id15';
          document.forms['_id0'].submit(); return false;">
        mg src="/test_JSF/images/oeil.jpg" alt="" /></a>
```

Le tag <commandLink> génère du code JavaScript dans la vue pour soumettre le formulaire lors d'un clic.

83.9.8. Le tag <ouputLink>

Ce composant représente un lien direct vers une ressource dont la demande ne sera pas traitée par le framework JSF.

Les attributs sont les suivants :

Attribut	Rôle
accesskey, binding, converter, id, lang, rendered, styleClass, value	attributs communs de base
charset, coords, dir, hreflang, lang, rel, rev, shape, style, tabindex, target, title, type	attributs communs liés à HTML
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup	attributs communs liés aux événements JavaScript

L'attribut value doit contenir l'URL qui sera utilisée dans l'attribut href du lien HTML. Si le premier caractère est un # (dièse) alors le lien pointe vers une ancre définie dans la même page.

Il est possible d'insérer dans le corps du tag ouputLink d'autres composants qui feront partie intégrante du lien.

Exemple :

```
<p>
  <h:outputLink value="http://java.sun.com">
    <h:graphicImage value="/images/java.jpg"/>
  </h:outputLink>
</p>
```

Résultat :

Le code HTML généré dans la page est le suivant :

<a href="http://java.sun.com"><img src="/test_JSF/images/java.jpg" alt="" /></a>

Attention, pour mettre du texte dans le corps du tag, il est nécessaire d'utiliser un tag verbatim ou outputText.

Exemple :

```
<p>
  <h:outputLink value="http://java.sun.com" title="Java">
    <f:verbatim>
       Site Java de Sun
    </f:verbatim>
  </h:outputLink>
</p>
```

83.9.9. Les tags <selectBooleanCheckbox> et <selectManyCheckbox>

Ces composants représentent respectivement une case à cocher et un ensemble de cases à cocher.

Les attributs sont les suivants :

Attribut	Rôle
disabledClass	classe CSS pour les éléments non sélectionnés (pour le tag selectManyCheckbox uniquement)
enabledClass	classe CSS pour les éléments sélectionnés (pour le tag selectManyCheckbox uniquement)
layout	préciser la disposition des éléments (pour le tag selectManyCheckbox uniquement)
binding, converter, id, immediate, styleClass, required, rendered, validator, value, valueChangeListener	attributs communs de base
accesskey, border, dir, disabled, lang, readonly, style, tabindex, title	attributs communs liés à HTML (border pour le tag selectManyCheckbox uniquement)
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect	attributs communs liés aux événements JavaScript

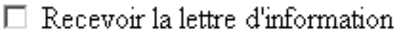
L'attribut layout permet de préciser la disposition des cases à cocher : lineDirection pour une disposition horizontale (c'est la valeur par défaut) et pageDirection pour une disposition verticale.

Le tag <selectBooleanCheckbox> dont la valeur peut être associée à une propriété booléenne d'un bean représente une case à cocher simple.

Exemple :

```
<h:selectBooleanCheckbox value="#{saisieOptions.recevoirLettre}">
</h:selectBooleanCheckbox> Recevoir la lettre d'information
```

Résultat :



Pour gérer l'état du composant, il faut utiliser l'attribut value en lui fournissant la valeur d'une propriété booléen d'un backing bean.

Exemple :

```
public class SaisieOptions {

  private boolean recevoirLettre;

  public void setRecevoirLettre(boolean valeur) {
    recevoirLettre = valeur;
  }

  public boolean getRecevoirLettre() {
    return recevoirLettre;
  }
...
```

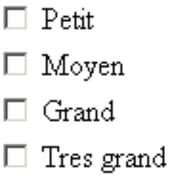
Le tag <selectManyCheckbox> représente un ensemble de cases à cocher. Dans cet ensemble, il est possible d'en sélectionner une ou plusieurs.

Chaque case à cocher est définie par un tag selectItem dans le corps du tag selectManyCheckbox.

Exemple :

```
<h:selectManyCheckbox layout="pageDirection">
  <f:selectItem itemValue="petit" itemLabel="Petit" />
  <f:selectItem itemValue="moyen" itemLabel="Moyen" />
  <f:selectItem itemValue="grand" itemLabel="Grand" />
  <f:selectItem itemValue="tresgrand" itemLabel="Tres grand" />
</h:selectManyCheckbox>
```

Résultat :



□ Petit

□ Moyen

□ Grand

□ Tres grand

Le rendu du composant est un tableau HTML dont chaque cellule contient une case à cocher encapsulée dans un tag HTML <label> :

Exemple :

```
<table>
 <tr>
  <td>
   <label><input name="_id0:_id1" value="petit" type="checkbox"> Petit</input></label></td>
 </tr>
 <tr>
  <td>
   <label><input name="_id0:_id1" value="moyen" type="checkbox"> Moyen</input></label></td>
 </tr>
 <tr>
  <td>
   <label><input name="_id0:_id1" value="grand" type="checkbox"> Grand</input></label></td>
 </tr>
 <tr>
  <td>
    <label><input name="_id0:_id1" value="tresgrand" type="checkbox"> Tres grand</input>
    </label></td>
 </tr>
</table>
```

83.9.10. Le tag <selectOneRadio>

Ce composant représente un ensemble de boutons radio dont un seul peut être sélectionné.

Les attributs sont les suivants :

Attribut	Rôle
disabledClass	classe CSS pour les éléments non sélectionnés
enabledClass	classe CSS pour les éléments sélectionnés
layout	préciser la disposition des éléments
binding, converter, id, immediate, styleClass, required, rendered, validator, value, valueChangeListener	Attributs communs de base

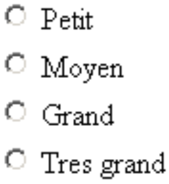
accesskey, border, dir, disabled, lang, readonly, style, tabindex, title	Attributs communs liés à HTML
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect	Attributs communs liés aux événements JavaScript

Les éléments peuvent être précisés un par un avec le tag <selectItem>.

Exemple :

```
<h:selectOneRadio layout="pageDirection">
  <f:selectItem itemValue="petit" itemLabel="Petit" />
  <f:selectItem itemValue="moyen" itemLabel="Moyen" />
  <f:selectItem itemValue="grand" itemLabel="Grand" />
  <f:selectItem itemValue="tresgrand" itemLabel="Tres grand" />
</h:selectOneRadio>
```

Résultat :



○ Petit
○ Moyen
○ Grand
○ Tres grand

Le rendu du composant est un tableau HTML dont chaque cellule contient un bouton radio encapsulé dans un tag HTML <label> :

Exemple :

```
<table>
  <tr>
    <td>
      <label><input type="radio" name="_id0:_id1" value="petit"> Petit</input></label></td>
  </tr>
  <tr>
    <td>
      <label><input type="radio" name="_id0:_id1" value="moyen"> Moyen</input></label></td>
  </tr>
  <tr>
    <td>
      <label><input type="radio" name="_id0:_id1" value="grand"> Grand</input></label></td>
  </tr>
  <tr>
    <td>
      <label><input type="radio" name="_id0:_id1" value="tresgrand"> Tres grand</input>
          </label></td>
  </tr>
</table>
```

Les éléments peuvent être précisés sous la forme d'un tableau de type SelecItem avec le tag <selectItems>.

Exemple :

```
<h:selectOneRadio value="#{saisieOptions.taille}" layout="pageDirection" id="taille">
<f:selectItems value="#{saisieOptions.tailleItems}"/>
</h:selectOneRadio>
```

Dans ce cas, le bean doit contenir au moins deux méthodes : getTaille() pour renvoyer la valeur de l'élément sélectionné et getTailleItems() qui renvoie un tableau d'objets de type SelectItems contenant les éléments.

Exemple :

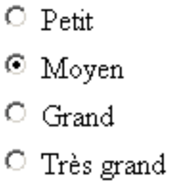
```
package fr.jmdoudoux.dej.jsf;

import javax.faces.model.*;

public class SaisieOptions {

  private Integer taille = null;

  private SelectItem[] tailleItems = {
    new SelectItem(new Integer(1), "Petit"),
    new SelectItem(new Integer(2), "Moyen"),
    new SelectItem(new Integer(3), "Grand"),
    new SelectItem(new Integer(4), "Très grand") };

  public SaisieOptions() {
    taille = new Integer(2);
  }

  public Integer getTaille() {
    return taille;
  }

  public void setTaille(Integer newValue) {
    taille = newValue;
  }

  public SelectItem[] getTailleItems() {
    return tailleItems;
  }
}
```

Le bean doit être déclaré dans le fichier faces-config.xml

Exemple :

```
<managed-bean>
  <managed-bean-name>saisieOptions</managed-bean-name>
  <managed-bean-class>fr.jmdoudoux.dej.jsf.SaisieOptions</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Résultat :



○ Petit
● Moyen
○ Grand
○ Très grand

83.9.11. Le tag <selectOneListbox>

Ce composant représente une liste d'éléments dont un seul peut être sélectionné.

Les attributs sont les suivants :

Attribut	Rôle
binding, converter, id, immediate, styleClass, required, rendered, validator, value, valueChangeListener	attributs communs de base

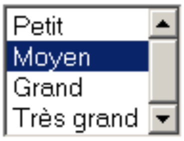
accesskey, dir, disabled, lang, readonly, style, size, tabindex, title	attributs communs liés à HTML
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect	attributs communs liés aux événements JavaScript

L'attribut size permet de préciser le nombre d'éléments de la liste affichée.

Exemple :

```
<h:selectOneListbox value="#{saisieOptions.taille}">
  <f:selectItems value="#{saisieOptions.tailleItems}"/>
</h:selectOneListbox>
```

Résultat :



83.9.12. Le tag <selectManyListbox>

Ce composant représente une liste d'éléments dont plusieurs peuvent être sélectionnés.

Les attributs sont les suivants :

Attribut	Rôle
binding, converter, id, immediate, styleClass, required, rendered, validator, value, valueChangeListener	attributs communs de base
accesskey, dir, disabled, lang, readonly, style, size, tabindex, title	attributs communs liés à HTML
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect	attributs communs liés aux événements JavaScript

Exemple :

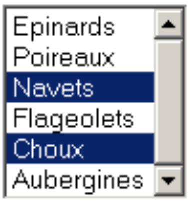
```
<h:selectManyListbox value="#{saisieOptions.legumes}">
  <f:selectItems value="#{saisieOptions.legumesItems}"/>
</h:selectManyListbox>
```

La liste des éléments sélectionnés doit pouvoir contenir zéro ou plusieurs valeurs sous la forme d'un tableau ou d'une liste.

Exemple :

```
package fr.jmdoudoux.dej.jsf;

import javax.faces.model.*;

public class SaisieOptions {

  private String[] legumes = {
    "navets", "choux" };
  private SelectItem[] legumesItems = {
    new SelectItem("epinards", "Epinards"),
    new SelectItem("poireaux", "Poireaux"),
    new SelectItem("navets", "Navets"),
    new SelectItem("flageolets", "Flageolets"),
    new SelectItem("choux", "Choux"),
    new SelectItem("aubergines", "Aubergines") };

  public SaisieOptions() {
  }

  public String[] getLegumes() {
    return legumes;
  }

  public SelectItem[] getLegumesItems() {
    return legumesItems;
  }
}
```

Résultat :



Il est possible d'utiliser un objet de type List à la place des tableaux.

Exemple :

```
package fr.jmdoudoux.dej.jsf;

import javax.faces.model.*;
import java.util.*;

public class SaisieOptions {

  private List legumes = null;

  private List legumesItems = null;

  public List getLegumesItems() {
    if (legumesItems == null) {
      legumesItems = new ArrayList();
      legumesItems.add(new SelectItem("epinards", "Epinards"));
      legumesItems.add(new SelectItem("poireaux", "Poireaux"));
      legumesItems.add(new SelectItem("navets", "Navets"));
      legumesItems.add(new SelectItem("flageolets", "Flageolets"));
      legumesItems.add(new SelectItem("choux", "Choux"));
      legumesItems.add(new SelectItem("aubergines", "Aubergines"));
    }
    return legumesItems;
  }

  public List getLegumes() {
    return legumes;
  }

  public void setLegumes(List newValue) {
    legumes = newValue;
  }

  public SaisieOptions() {
    legumes = new ArrayList();
    legumes.add("navets");
    legumes.add("choux");
  }
}
```

83.9.13. Le tag <selectOneMenu>

Ce composant représente une liste déroulante dont un seul élément peut être sélectionné.

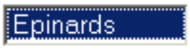
Les attributs sont les suivants :

Attribut	Rôle
binding, converter, id, immediate, styleClass, required, rendered, validator, value, valueChangeListener	attributs communs de base
accesskey, dir, disabled, lang, readonly, style, tabindex, title	attributs communs liés à HTML
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect	attributs communs liés aux événements JavaScript

Exemple :

```
<h:selectOneMenu value="#{saisieOptions.taille}">
<f:selectItems value="#{saisieOptions.tailleItems}"/>
</h:selectOneMenu>
```

Résultat :



Exemple : le code HTML généré

```
<select name="_id0:_id6" size="1"> <option value="1">Petit</option>
  <option value="2" selected="selected">Moyen</option>
  <option value="3">Grand</option>
  <option value="4">Tr&egrave;s grand</option>
</select>
```

83.9.14. Le tag <selectManyMenu>

Ce composant représente une liste d'éléments dont le rendu HTML est un tag select avec une seule option visible.

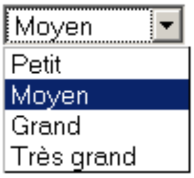
Les attributs sont les suivants :

Attribut	Rôle
binding, converter, id, immediate, styleClass, required, rendered, validator, value, valueChangeListener	Attributs communs de base
accesskey, dir, disabled, lang, readonly, style, tabindex, title	Attributs communs liés à HTML
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect	Attributs communs liés aux événements JavaScript

Exemple :

```
<h:selectManyMenu value="#{saisieOptions.legumes}">
<f:selectItems value="#{saisieOptions.legumesItems}"/>
</h:selectManyMenu>
```

Résultat :



83.9.15. Les tags <message> et <messages>

Des messages peuvent être émis lors de traitements. Ils sont stockés dans le contexte de l'application JSF pour être restitués dans la vue. Ils permettent notamment de fournir des messages d'erreurs aux utilisateurs.

JSF définit quatre types de messages :

- Information
- Warning
- Error
- Fatal

Chaque message possède un résumé et un descriptif.

Le tag <messages> permet d'afficher tous les messages stockés dans le contexte de l'application JSF.

Le tag message permet d'afficher un seul message, le dernier ajouté, pour un composant donné.

Les attributs sont les suivants :

Attributs	Rôle
errorClass	nom d'une classe CSS pour un message de type error
errorStyle	style CSS pour un message de type error
fatalClass	nom d'une classe CSS pour un message de type fatal
fatalStyle	style CSS pour un message de type fatal
globalOnly	booléen qui permet de n'afficher que les messages qui ne sont pas associés à un composant. Par défaut, False (uniquement pour le tag messages)
infoClass	nom d'une classe CSS pour un message de type Information
infoStyle	style CSS pour un message de type Information
Layout	format de la liste de messages : list ou table (uniquement pour le composant messages)
showDetail	booléen qui précise si la description des messages est affichée ou non. Par défaut, false pour le tag message et true pour le tag messages
showSummary	booléen qui précise si le résumé des messages est affiché ou non. Par défaut, true pour le tag message et false pour le tag messages
Tooltip	booléen qui précise si la description est affichée sous la forme d'une bulle d'aide
warnClass	nom d'une classe CSS pour un message de type Warning
warnStyle	le style CSS pour un message de type Warning
For	l'identifiant du composant pour lequel le message doit être affiché
binding, id, rendered, styleClass	attributs communs de base
style, title	attributs communs liés à HTML

83.9.16. Le tag <panelGroup>

Ce composant permet de regrouper plusieurs composants.

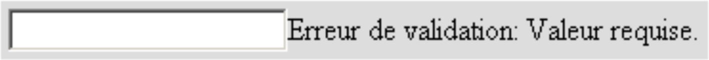
Les attributs sont les suivants :

Attribut	Rôle
binding, id, rendered, styleClass	attributs communs de base
style	style CSS

Exemple :

```
<td bgcolor='#DDDDDD'>
  <h:panelGroup>
    <h:inputText value="#{login.nom}" id="nom" required="true"/>
    <h:message for="nom"/>
  </h:panelGroup>
</td>
```

Résultat :



Erreur de validation: Valeur requise.

83.9.17. Le tag <panelGrid>

Ce composant représente un tableau HTML.

Les attributs sont les suivants :

Attribut	Rôle
bgcolor	couleur de fond du tableau
border	taille de la bordure du tableau
cellpadding	espacement intérieur de chaque cellule
cellspacing	espacement extérieur de chaque cellule
columnClasses	nom de classes CSS pour les colonnes. Il est possible de préciser plusieurs noms de classes qui seront utilisées sur chaque colonne
columns	nombre de colonnes du tableau
footerClass	nom de la classe CSS pour le pied du tableau
frame	précise les règles pour le contour du tableau. Les valeurs possibles sont : none, above, below, hsides, vsides, lhs, rhs, box, border
headerClass	nom de la classe CSS pour l'en-tête du tableau
rowClasses	nom de classes CSS pour les lignes. Il est possible de préciser deux noms de classes séparés par une virgule qui seront utilisés alternativement sur chaque ligne
rules	précise les règles de dessin des lignes entre les cellules. Les valeurs possibles sont : groups, rows, columns, all
summary	résumé du tableau

binding, id, rendered, styleClass, value	attributs communs de base
dir, lang, style, title, width	attributs communs liés à HTML
onclick, ondblclick, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup	attributs communs liées aux événements JavaScript

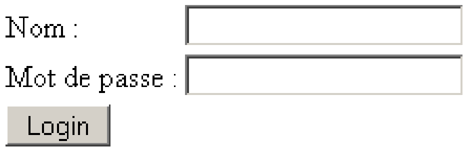
Par défaut les composants sont insérés les uns à la suite des autres dans les cellules en partant de la gauche vers la droite et en passant à la ligne suivante si nécessaire.

Il est possible de ne mettre qu'un seul composant par cellule. Ainsi pour placer plusieurs composants dans une cellule, il faut les regrouper dans un tag panelGroup.

Exemple :

```
<h:panelGrid columns="2">
  <h:outputText value="Nom : " />
  <h:panelGroup>
    <h:inputText value="#{login.nom}" id="nom" required="true"/>
    <h:message for="nom"/>
  </h:panelGroup>
  <h:outputText value="Mot de passe :" />
  <h:inputSecret value="#{login.mdp}"/>
  <h:commandButton value="Login" action="login"/>
</h:panelGrid>
```

Résultat :



Le code HTML généré est le suivant :

Exemple : le code HTML généré

```
...
<table>

  <tbody>
    <tr>
      <td>Nom : </td>
      <td><input id="_id0:nom" type="text" name="_id0:nom" /></td>
    </tr>
    <tr>
      <td>Mot de passe :</td>
      <td><input type="password" name="_id0:_id6" value="" /></td>
    </tr>
    <tr>
      <td><input type="submit" name="_id0:_id7" value="Login" /></td>
    </tr>
  </tbody>
</table>
...
```

83.9.18. Le tag <dataTable>

Ce composant représente un tableau HTML dans lequel des données vont pouvoir être automatiquement présentées. Ce composant est sûrement le plus riche en fonctionnalité et donc le plus complexe des composants fournis en standard.

Les attributs sont les suivants :

Attribut	Rôle
bgcolor	couleur de fond du tableau
border	taille de la bordure du tableau
cellpadding	espacement intérieur de chaque cellule
cellspacing	espacement extérieur de chaque cellule
columnClasses	nom de classes CSS pour les colonnes. Il est possible de préciser plusieurs noms de classes qui seront utilisées sur chaque colonne
first	index de la première occurrence des données qui sera affichée dans le tableau
footerClass	nom de la classe CSS pour le pied du tableau
frame	précise les règles pour le contour du tableau. Les valeurs possibles sont : none, above, below, hsides, vsides, lhs, rhs, box, border
headerClass	nom de la classe CSS pour l'en-tête du tableau
rowClasses	nom de classes CSS pour les lignes. Il est possible de préciser deux noms de classes qui seront utilisées alternativement sur chaque ligne
rules	précise les règles de dessin des lignes entre les cellules. Les valeurs possibles sont : groups, rows, columns, all
summary	résumé du tableau
var	nom de la variable qui va contenir l'occurrence en cours de traitement lors du parcours des données
binding, id, rendered, styleClass, value	attributs communs de base
dir, lang, style, title, width	attributs communs liés à HTML
onclick, ondblclick, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup	attributs communs liés aux événements JavaScript

Le tag <dataTable> parcourt les données et pour chaque occurrence, il crée une ligne dans le tableau.

L'attribut value représente une expression qui précise les données à utiliser. Ces données peuvent être sous la forme :

- d'un tableau
- d'un objet de type java.util.List
- d'un objet de type java.sql.ResultSet
- d'un objet de type javax.servlet.jsp.jstl.sql.Result
- d'un objet de type javax.faces.model.DataModel

Pour chaque élément encapsulé dans les données, le tag dataTable crée une nouvelle ligne.

Quelque soit le type qui encapsule les données, le composant dataTable va les mapper dans un objet de type DataModel. C'est cet objet que le composant va utiliser comme source de données. JSF définit 5 classes qui héritent de la classe DataModel : ArrayDataModel, ListDataModel, ResultDataModel, ResultSetDataModel et ScalarDataModel.

La méthode getWrappedObject() permet d'obtenir la source de données fournie en paramètre de l'attribut value.

L'attribut item permet de préciser le nom d'une variable qui va contenir les données d'une occurrence.

Chaque colonne est définie grâce à un tag <column>.

Exemple :

```
<h:dataTable value="#{listePersonnes.personneItems}" var="personne" cellspacing="4">
  <h:column>
    <f:facet name="header">
      <h:outputText value="Nom" />
    </f:facet>
    <h:outputText value="#{personne.nom}"/>
  </h:column>

  <h:column>
    <f:facet name="header">
      <h:outputText value="Prenom" />
    </f:facet>
    <h:outputText value="#{personne.prenom}" />
  </h:column>

  <h:column>
    <f:facet name="header">
      <h:outputText value="Date de naissance" />
    </f:facet>
    <h:outputText value="#{personne.datenaiss}"/>
  </h:column>

  <h:column>
    <f:facet name="header">
      <h:outputText value="Poids" />
    </f:facet>
    <h:outputText value="#{personne.poids}"/>
  </h:column>

  <h:column>
    <f:facet name="header">
      <h:outputText value="Taille" />
    </f:facet>
    <h:outputText value="#{personne.taille}"/>
  </h:column>

</h:dataTable>
```

L'en-tête et le pied du tableau sont précisés avec un tag <facet> pour chacun dans chaque tag <column>.

Dans l'exemple précédent l'instance listePersonnes est une classe dont le code est le suivant :

Exemple :

```
package fr.jmdoudoux.dej.jsf;

import java.util.ArrayList;
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.List;

public class PersonnesBean {

  private List PersonneItems = null;

  public List getPersonneItems() {
    if (PersonneItems == null) {
      PersonneItems = new ArrayList();
      PersonneItems.add(new Personne("Nom1", "Prenom1",
            new GregorianCalendar(1967, Calendar.OCTOBER, 22).getTime(),10,1.10f));
      PersonneItems.add(new Personne("Nom2", "Prenom2",
            new GregorianCalendar(1972, Calendar.MARCH, 10).getTime(),20,1.20f));
      PersonneItems.add(new Personne("Nom3", "Prenom3",
            new GregorianCalendar(1944, Calendar.NOVEMBER, 4).getTime(),30,1.30f));
      PersonneItems.add(new Personne("Nom4", "Prenom4",
            new GregorianCalendar(1958, Calendar.JULY, 19).getTime(),40,1.40f));
      PersonneItems.add(new Personne("Nom5", "Prenom5",
            new GregorianCalendar(1934, Calendar.JANUARY, 6).getTime(),50,1.50f));
      PersonneItems.add(new Personne("Nom6", "Prenom6",
            new GregorianCalendar(1989, Calendar.DECEMBER, 12).getTime(),60,1.60f));
    }
    return PersonneItems;
  }
}
```

La méthode getPersonneItems() renvoie une collection d'objets de type Personne.

La classe Personne encapsule simplement les données d'une personne.

Exemple :

```
package fr.jmdoudoux.dej.jsf;

import java.util.Date;

public class Personne {
  private String nom;
  private String prenom;
  private Date datenaiss;
  private int poids;
  private float taille;
  private boolean supprime;

  public Personne(String nom, String prenom, Date datenaiss, int poids,
    float taille) {
    super();
    this.nom = nom;
    this.prenom = prenom;
    this.datenaiss = datenaiss;
    this.poids = poids;
    this.taille = taille;
    this.supprime = false;
  }

  public boolean isSupprime() {
    return supprime;
  }

  public void setSupprime(boolean supprimer) {
    supprime = supprimer;
  }

  public Date getDatenaiss() {
    return datenaiss;
  }

  public void setDatenaiss(Date datenaiss) {
    this.datenaiss = datenaiss;
  }

  public String getNom() {
    return nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

  public int getPoids() {
    return poids;
  }

  public void setPoids(int poids) {
    this.poids = poids;
  }

  public String getPrenom() {
    return prenom;
  }
  public void setPrenom(String prenom) {
    this.prenom = prenom;
  }

  public float getTaille() {
    return taille;
  }

  public void setTaille(float taille) {
    this.taille = taille;
  }
}
```

Il est très facile de préciser un style particulier pour des lignes paires et impaires.

Il suffit de définir les deux styles désirés.

Exemple dans la partie en-tête de la JSP :

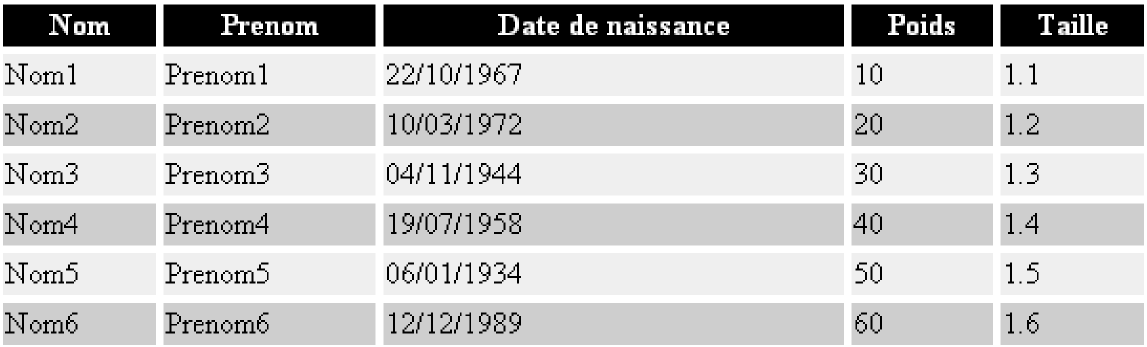
```
<STYLE type="text/css">
<!--
.titre {
background-color:#000000;
color:#FFFFFF;
}
.paire {
background-color:#EFEFEF;
}
.impaire {
background-color:#CECECE;
}
-->
</STYLE>
```

Il suffit d'utiliser les attributs headerClass, footerClass, rowClasses ou columnClasses. Avec ces deux derniers attributs, il est possible de préciser plusieurs styles séparés par une virgule pour définir l'apparence de chacune des lignes de façon répétitive.

Exemple :

```
<h:dataTable value="#{listePersonnes.personneItems}" var="personne"
  cellspacing="4" width="60%" rowClasses="paire,impaire" headerClass="titre">
```

Résultat :



Nom	Prenom	Date de naissance	Poids	Taille
Nom1	Prenom1	22/10/1967	10	1.1
Nom2	Prenom2	10/03/1972	20	1.2
Nom3	Prenom3	04/11/1944	30	1.3
Nom4	Prenom4	19/07/1958	40	1.4
Nom5	Prenom5	06/01/1934	50	1.5
Nom6	Prenom6	12/12/1989	60	1.6

Les éléments du tableau peuvent par exemple être sélectionnés grâce à une case à cocher pour permettre de réaliser des traitements sur les éléments marqués.

Il suffit de rajouter dans l'exemple précédent une colonne contenant une case à cocher et, sous le tableau, un bouton qui va réaliser les traitements sur les éléments cochés.

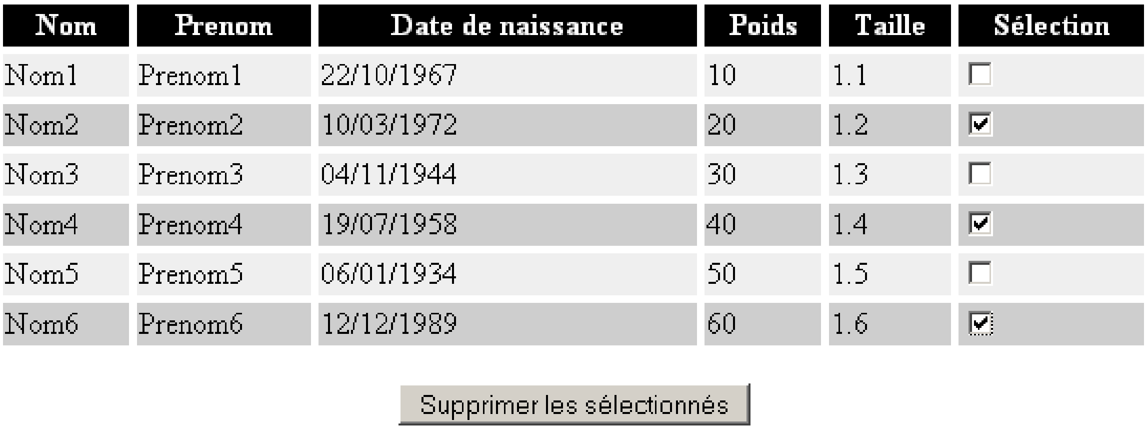
Exemple dans la JSP :

```
...
<h:form>
  <h1>Test</H1>
  <div align="center">
    <h:dataTable value="#{listePersonnes.personneItems}" var="personne"
          cellspacing="4" width="60%" rowClasses="paire,impaire" headerClass="titre">
...
    <h:column>
      <f:facet name="header">
        <h:outputText value="Sélection"/>
      </f:facet>
      <h:selectBooleanCheckbox value="#{personne.supprime}" />
    </h:column>

    </h:dataTable>

    <p>
      <h:commandButton value="Supprimer les sélectionnés"
        action="#{listePersonnes.supprimer}"/>
    </p>

  </div>
</h:form>
...
```

Il reste alors à ajouter les traitements dans la méthode supprimer() de la classe PersonnesBean qui sera appelée lors d'un clic sur le bouton « Supprimer les sélectionnés ».

Exemple :

```
public class PersonnesBean {

...

  public String supprimer() {
    Iterator iterator = personneItems.iterator();
    Personne pers=null;
    while (iterator.hasNext()) {
      pers = (Personne) iterator.next();
      System.out.println("nom="+pers.getNom()+" "+pers.isSupprime());
      // ajouter les traitements utiles
    }
    return null;
  }
}
```



Nom	Prenom	Date de naissance	Poids	Taille	Sélection
Nom1	Prenom1	22/10/1967	10	1.1	□
Nom2	Prenom2	10/03/1972	20	1.2	☑
Nom3	Prenom3	04/11/1944	30	1.3	□
Nom4	Prenom4	19/07/1958	40	1.4	☑
Nom5	Prenom5	06/01/1934	50	1.5	□
Nom6	Prenom6	12/12/1989	60	1.6	☑

Un clic sur le bouton « Supprimer les sélectionnés » affiche dans la console, la liste des éléments avec l'état de la case à cocher.

Exemple :

```
nom=Nom1 false
nom=Nom2 true
nom=Nom3 false
nom=Nom4 true
nom=Nom5 false
nom=Nom6 true
```

83.10. La gestion et le stockage des données

Les données sont stockées dans un ou plusieurs JavaBeans qui encapsulent les différentes données des composants.

Ces données possèdent deux représentations :

- une contenue en interne par le modèle
- une pour leur présentation dans l'interface graphique (pour la saisie ou l'affichage)

Chaque objet de type Renderer possède une représentation par défaut des données. La transformation d'une représentation en une autre est assurée par des objets de type Converter. JSF fournit en standard plusieurs objets de type Converter mais il est aussi possible de développer ses propres objets.

83.11. La conversion des données

JSF propose en standard un mécanisme de conversion des données. Celui-ci repose sur un ensemble de classes dont certaines sont fournies en standard pour des conversions de base. Il est possible de définir ses propres classes de conversion pour répondre à des besoins spécifiques.

Ces conversions sont nécessaires car toutes les données transmises et affichées le sont sous la forme de chaînes de caractères. Cependant, leur exploitation dans les traitements nécessite souvent qu'elles soient stockées dans un autre format pour être exploitées : un exemple flagrant est une donnée de type date.

Toutes les données saisies par l'utilisateur sont envoyées dans la requête http sous la forme de chaînes de caractères. Chacune de ces valeurs est désignée par « request value » dans les spécifications de JSF.

Ces valeurs sont stockées dans leurs composants respectifs dans des champs désignés par « submitted value » dans les spécifications.

Ces valeurs sont éventuellement converties implicitement ou explicitement et sont stockées dans leurs composants respectifs dans des champs désignés par « local value ». Ensuite, ces données sont éventuellement validées.

L'intérêt d'un tel procédé est de s'assurer que les données seront valides avant de pouvoir les utiliser dans les traitements. Si la conversion ou la validation échoue, les traitements du cyle de vie de la page sont arrêtés et la page est réaffichéeen montrant les messages d'erreurs. Sinon la phase de mise à jour des données (« Update model values ») du modèle est exécutée.

Les spécifications JSF imposent l'implémentation des convertisseurs suivants : javax.faces.DateTime, javax.faces.Number, javax.faces.Boolean, javax.faces.Byte, javax.faces.Character, javax.faces.Double, javax.faces.Float, javax.faces.Integer, javax.faces.Long, javax.faces.Short, javax.faces.BigDecimal et javax.faces.BigInteger.

JSF effectue une conversion implicite des données lorsque celles-ci correspondent à un type primitif, à BigDecimal ou BigInteger en utilisant les convertisseurs appropriés.

Deux convertisseurs sont proposés en standard pour mettre en oeuvre des conversions qui ne correspondent pas à des types primitifs :

- le tag convertNumber : utilise le convertisseur javax.faces.Number
- le tag convertDateTime : utilise le convertisseur javax.faces.DateTime

83.11.1. Le tag <convertNumber>

Ce tag permet d'ajouter à un composant un convertisseur de valeur numérique.

Ce tag possède les attributs suivants :

Attributs	Rôle
type	type de valeur. Les valeurs possibles sont number (par défaut), currency et percent
pattern	motif de formatage qui sera utilisé par une instance de java.text.DecimalFormat
maxFractionDigits	nombre maximum de chiffres composant la partie décimale
minFractionDigits	nombre minimum de chiffres composant la partie décimale
maxIntegerDigits	nombre maximum de chiffres composant la partie entière
minIntegerDigits	nombre minimum de chiffres composant la partie entière
integerOnly	booléen qui précise si uniquement la partie entière est prise en compte (false par défaut)
groupingUsed	booléen qui précise si le séparateur de groupe d'unité est utilisé (true par défaut)
locale	objet de type java.util.Locale permettant de définir la Locale à utiliser pour les conversions
currencyCode	code de la monnaie utilisée pour la conversion
currencySymbol	symbole de la monnaie utilisée pour la conversion

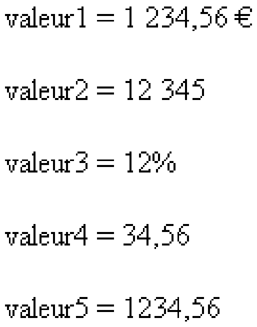
Exemple :

```
<p>valeur1 = <h:outputText value="#{convert.prix}">
<f:convertNumber type="currency"/>
</h:outputText>
</p>

<p>valeur2 = <h:outputText value="#{convert.poids}">
<f:convertNumber type="number"/>
</h:outputText>
</p>

<p>valeur3 = <h:outputText value="#{convert.ratio}">
<f:convertNumber type="percent"/>
</h:outputText>
</p>

<p>valeur4 = <h:outputText value="#{convert.prix}">
<f:convertNumber integerOnly="true" maxIntegerDigits="2"/>
</h:outputText>
</p>

<p>valeur5 = <h:outputText value="#{convert.prix}">
<f:convertNumber pattern="#.##"/>
</h:outputText>
</p>
```

Le code du bean utilisé dans cet exemple est le suivant :

Exemple :

```
package fr.jmdoudoux.dej.jsf;

public class Convert {
  private int poids;
  private float prix;
  private float ratio;
  public Convert() {
    super();
    this.poids = 12345;
    this.prix = 1234.56f ;
    this.ratio = 0.12f ;
  }


  public int getPoids() {
    return poids;
  }

  public void setPoids(int poids) {
    this.poids = poids;
  }

  public float getRatio() {
    return ratio;
  }

  public void setRatio(float ratio) {
    this.ratio = ratio;
  }

  public float getPrix() {
    return prix;
  }

  public void setPrix(float prix) {
    this.prix = prix;
  }
}
```



```
valeur1 = 1 234,56 €

valeur2 = 12 345

valeur3 = 12%

valeur4 = 34,56

valeur5 = 1234,56
```

83.11.2. Le tag <convertDateTime>

Ce tag permet d'ajouter à un composant un convertisseur de valeurs temporelles.

Ce tag possède les attributs suivants :

Attributs	Rôle
Type	type de valeur. Les valeurs possibles sont date (par défaut), time et both
dateStyle	style prédéfini de la date. Les valeurs possibles sont short, medium, long, full ou default
timeStyle	style prédéfini de l'heure. Les valeurs possibles sont short, medium, long, full ou default
Pattern	motif de formatage qui sera utilisé par une instance de java.text.SimpleDateFormat
Locale	objet de type java.util.Locale permettant de définir la locale à utiliser pour les conversions
timeZone	objet de type java.util.TimeZone utilisé lors des conversions

Exemple :

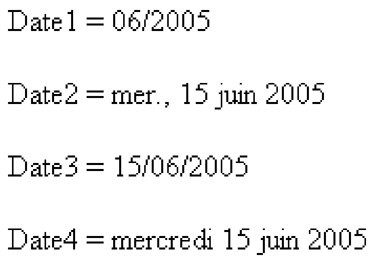
```
<p>Date1 = <h:outputText value="#{convertDate.dateNaiss}">
<f:convertDateTime pattern="MM/yyyy"/>
</h:outputText>
</p>

<p>Date2 = <h:outputText value="#{convertDate.dateNaiss}">
<f:convertDateTime pattern="EEE, dd MMM yyyy"/>
</h:outputText>
</p>

<p>Date3 = <h:outputText value="#{convertDate.dateNaiss}">
<f:convertDateTime pattern="dd/MM/yyyy"/>
</h:outputText>
</p>

<p>Date4 = <h:outputText value="#{convertDate.dateNaiss}">
<f:convertDateTime dateStyle="full"/>
</h:outputText>
</p>
```

Le code du bean utilisé comme source dans cet exemple est le suivant :

Exemple :

```
package fr.jmdoudoux.dej.jsf;

import java.util.Date;

public class ConvertDate {
  private Date dateNaiss;

  public ConvertDate() {
    super();
    this.dateNaiss = new Date();
  }

  public Date getDateNaiss() {
    return dateNaiss;
  }

  public void setDateNaiss(Date dateNaiss) {
    this.dateNaiss = dateNaiss;
  }
}
```

Résultat :



```
Date1 = 06/2005

Date2 = mer., 15 juin 2005

Date3 = 15/06/2005

Date4 = mercredi 15 juin 2005
```

83.11.3. L'affichage des erreurs de conversions

Les messages d'erreurs issus de ces conversions peuvent être affichés en utilisant les tag <message> ou <messages>.

Par défaut, ils contiennent une description : « Conversion error occured ».

Pour modifier ce message par défaut ou l'internationaliser, il faut définir une clé javax.faces.component.UIInput.CONVERSION dans le fichier properties de définition des chaînes de caractères.

Exemple :

```
javax.faces.component.UIInput.CONVERSION=La valeur saisie n'est pas correctement formatée.
```

83.11.4. L'écriture de convertisseurs personnalisés

JSF fournit en standard des convertisseurs pour les types primitifs et quelques objets de base. Il peut être nécessaire de développer son propre convertisseur pour des besoins spécifiques.

Pour écrire son propre convertisseur, il faut définir une classe qui implémente l'interface Converter. Cette interface définit deux méthodes :

- Object getAsObject(FacesContext context, UIComponent component, String newValue) : cette méthode permet de convertir une chaîne de caractères en objet
- String getAsString(FacesContext context, UIComponent component, Object value) : cette méthode permet de convertir un objet en chaîne de caractères

La méthode getAsObject() doit lever une exception de type ConverterException si une erreur de conversion est détectée dans les traitements.

La suite de ce chapitre sera développée dans une version future de ce document

83.12. La validation des données

JSF propose en standard un mécanisme de validation des données. Celui-ci repose sur un ensemble de classes qui permettent de faire des vérifications standard. Il est possible de définir ses propres classes de validation pour répondre à des besoins spécifiques.

La validation peut se faire de deux façons : au niveau de certains composants ou avec des classes spécialement développées pour des besoins spécifiques. Ces classes sont attachables à un composant et sont réutilisables. Ces validations sont effectuées côté serveur.

Les validators sont enregistrés sur des composants. Ce sont des classes qui utilisent des données pour effectuer des opérations de validation de la valeur des données : contrôle de présence, de type de données, de plage de valeurs, de format, ...

83.12.1. Les classes de validation standard

Toutes ces classes implémentent l'interface javax.faces.validator.Validator. JSF propose en standard plusieurs classes pour la validation :

- deux classes de validation sur une plage de données : LongRangeValidator et DoubleRangeValidator
- une classe de validation de la taille d'une chaîne de caractères : LengthValidator

Pour faciliter l'utilisation de ces classes, la bibliothèque de tags personnalisés Core propose des tags dédiés à la mise en oeuvre de ces classes :

- validateDoubleRange : utilise la classe DoubleRangeValidator
- validateLongRange : utilise la classe LongRangeValidator
- validateLength : utilise la classe LengthValidator

Ces trois tags possèdent deux attributs nommés minimum et maximum qui permettent de préciser respectivement la valeur de début et de fin selon le Validator utilisé. L'un, l'autre ou les deux attributs peuvent être utilisés.

L'ajout d'une validation sur un contrôle peut se faire de plusieurs manières :

- ajout d'une ou plusieurs validations directement dans la JSP
- ajout par programmation d'une validation en utilisant la méthode addValidator().
- certaines implémentations de composants peuvent contenir des validations implicites.

Pour ajouter une validation à un composant dans la JSP , il suffit d'insérer le tag de validation dans le corps du tag du composant.

Exemple :

```
<h:inputText id="nombre" converter="#{Integer}" required="true"
  value="#{saisieDonnees.nombre}">
  <f:validate_longrange minimum="1" maximum="9" />
</h:inputText>
```

Certaines implémentations de composants peuvent contenir des validations implicites en fonction du contexte. C'est par exemple le cas du composant <inputText> qui, lorsque son attribut required est à true, effectue un contrôle de présence de données saisies.

Exemple :

```
<h:inputText id="nombre" converter="#{Integer}" required="true"
  value="#{saisieDonnees.nombre}"/>
```

Toutes les validations sont faites côté serveur dans la version courante de JSF.

Les messages d'erreurs issus de ces conversions peuvent être affichés en utilisant les tags <message> ou <messages>.

Ils contiennent une description par défaut selon le validator utilisé commençant par « Validation error : ».

Pour modifier ce message par défaut ou l'internationaliser, il faut définir une clé dédiée dans le fichier properties de définition des chaînes de caractères. Les clés définies sont les suivantes :

- javax.faces.component.UIInput.REQUIRED
- javax.faces.validator.NOT_IN_RANGE
- javax.faces.validator.DoubleRangeValidator.MAXIMUM
- javax.faces.validator.DoubleRangeValidator.TYPE
- javax.faces.validator.DoubleRangeValidator.MINIMUM
- javax.faces.validator.LongRangeValidator.MAXIMUM
- javax.faces.validator.LongRangeValidator.MINIMUM
- javax.faces.validator.LongRangeValidator.TYPE
- javax.faces.validator.LengthValidator.MAXIMUM
- javax.faces.validator.LengthValidator.MINIMUM

83.12.2. Contourner la validation

Dans certains cas, il est nécessaire d'empêcher la validation. Par exemple, dans une page de saisie d'informations disposant d'un bouton « Valider » et « Annuler ». La validation doit être opérée lors d'un clic sur le bouton « Valider » mais ne doit pas l'être lors d'un clic sur le bouton « Annuler ».

Pour chaque composant dont l'action doit être exécutée sans validation, il faut mettre l'attribut immediate du composant à true.

Exemple :

```
<h:commandButton value="Annuler" action="annuler" immediate="true"/>
```

83.12.3. L'écriture de classes de validation personnalisées

JSF fournit en standard des classes de validation de base. Il peut être nécessaire de développer ses propres classes de validation pour des besoins spécifiques.

Pour écrire sa propre classe de validation, il faut définir une classe qui implémente l'interface javax.faces.validator.Validator. Cette interface définit une seule méthode :

- public void validate(FacesContext context, UIComponent component, Object toValidate) : cette méthode permet de réaliser les traitements de validation

Elle attend en paramètre :

- un objet de type FacesContext qui permet d'accéder au contexte de l'application jsf
- un objet de type UICOmponent qui contient une référence sur le composant dont la donnée est à valider
- un objet de type Object qui encapsule la valeur de la données à valider.

La méthode validate() doit lever une exception de type ValidatorException si une erreur dans les traitements de validation est détectée.

Exemple :

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.component.UIInput;
import javax.faces.context.FacesContext;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;

import com.sun.faces.util.MessageFactory;

public class NumeroDeSerieValidator implements Validator {

  public static final String CLE_MESSAGE_VALIDATION_IMPOSSIBLE =
    "message.validation.impossible";

  public void validate(FacesContext contexte, UIComponent composant,
    Object objet) throws ValidatorException {
    String valeur = null;
    boolean estValide = false;

    if ((contexte == null) || (composant == null)) {
      throw new NullPointerException();
    }
    if (!(composant instanceof UIInput)) {
      return;
    }

    valeur = objet.toString();

    Pattern p = Pattern.compile("[0-9][0-9]-[0-9][0-9][0-9]",Pattern.MULTILINE);
    Matcher m = p.matcher(valeur);
    estValide = m.matches();

    if (!estValide) {
      FacesMessage errMsg = MessageFactory.getMessage(contexte,
      CLE_MESSAGE_VALIDATION_IMPOSSIBLE);
      throw new ValidatorException(errMsg);
    }

  }
}
```

Dans l'exemple précédent, la valeur à valider doit respecter une expression régulière de la forme deux chiffres, un tiret et trois chiffres.

Si la validation échoue alors il sera nécessaire d'informer l'utilisateur de la raison de l'échec grâce à un message stocké dans le resourceBundle de l'application.

Exemple :

```
message.validation.impossible=Le format du numéro de série est erroné
```

La valeur du message dans le resourceBundle peut être obtenue en utilisant la méthode getMessage() de la classe MessageFactory. Cette méthode attend en paramètres le contexte JSF de l'application et la clé du ressourceBundle à extraire. Elle renvoie un objet de type FacesMessages. Il suffit de fournir cet objet à la nouvelle instance de la classe ValidatorException.

Pour pouvoir utiliser une classe de validation, il faut la déclarer dans le fichier de configuration.

Exemple :

```
<validator>
  <validator-id>fr.jmdoudoux.dej.jsf.NumeroDeSerie</validator-id>
  <validator-class>fr.jmdoudoux.dej.jsf.NumeroDeSerieValidator</validator-class>
</validator>
```

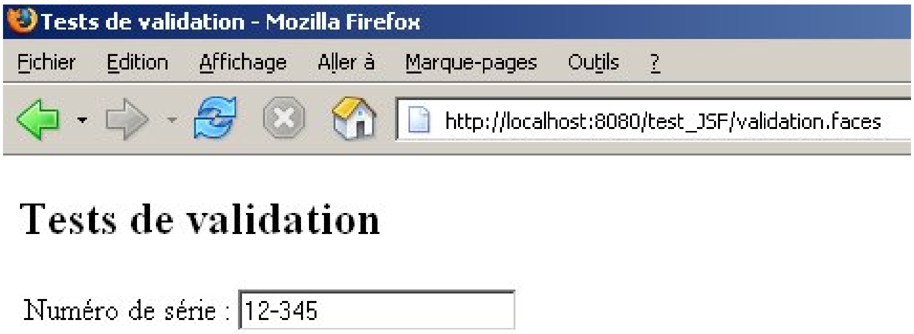
Le tag <validator-id> permet de définir un identifiant pour la classe de validation. Le tag <validator-class> permet de préciser la classe pleinement qualifiée.

Pour utiliser la classe de validation dans une page, il faut utiliser le tag <validator> en fournissant à l'attribut validatorId la valeur donnée au tag <validator-id> dans le fichier de configuration :

Exemple :

```
<h:panelGrid columns="2">
  <h:outputText value="Numéro de série : " />
  <h:panelGroup>
    <h:inputText value="#{validation.numeroSerie}" id="numeroSerie" required="true">
      <f:validator validatorId="fr.jmdoudoux.dej.jsf.NumeroDeSerie"/>
    </h:inputText>
    <h:message for="numeroSerie"/>
  </h:panelGroup>
</h:panelGrid>
```

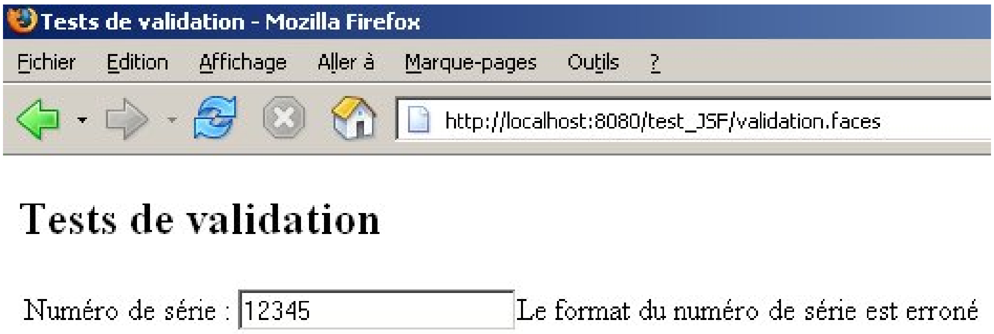
La saisie d'un numéro répondant à l'expression régulière et l'appui sur la touche entrée n'affiche aucun message d'erreur :


Tests de validation - Mozilla Firefox
Fichier
Edition
Affichage
Aller à
Marque-pages
Outils
?
http://localhost:8080/test_JSF/validation.faces

Tests de validation

Numéro de série : 12-345

La saisie d'un numéro ne répondant pas à l'expression régulière affiche le message d'erreur :


Tests de validation - Mozilla Firefox
Fichier Edition Affichage Aller à Marque-pages Outils ?
http://localhost:8080/test_JSF/validation.faces

Tests de validation

Numéro de série : 12345 Le format du numéro de série est erroné

83.12.4. La validation à l'aide de bean

Il est possible de définir une méthode dans un bean qui va offrir les services de validation. Cette méthode doit avoir une signature similaire à celle de la méthode validate() de l'interface Validator.

Exemple :

```
package fr.jmdoudoux.dej.jsf;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.component.UIInput;
import javax.faces.context.FacesContext;
import javax.faces.validator.ValidatorException;

import com.sun.faces.util.MessageFactory;

public class Validation {
  public static final String CLE_MESSAGE_VALIDATION_IMPOSSIBLE =
    "message.validation.impossible";

  private String numeroSerie;

  public String getNumeroSerie() {
    return numeroSerie;
  }

  public void setNumeroSerie(String numeroSerie) {
    this.numeroSerie = numeroSerie;
  }

  public void valider(FacesContext contexte, UIComponent composant, Object objet) {
    String valeur = null;
    boolean estValide = false;

    if ((contexte == null) || (composant == null)) {
      throw new NullPointerException();
    }
    if (!(composant instanceof UIInput)) {
      return;
    }

    valeur = objet.toString();

    Pattern p = Pattern.compile("[0-9][0-9]-[0-9][0-9][0-9]",Pattern.MULTILINE);
    Matcher m = p.matcher(valeur);
    estValide = m.matches();

    if (!estValide) {
      FacesMessage errMsg = MessageFactory.getMessage(contexte,
      CLE_MESSAGE_VALIDATION_IMPOSSIBLE);
      throw new ValidatorException(errMsg);
    }
  }
}
```

Pour utiliser cette méthode, il faut utiliser l'attribut validator et lui fournir en paramètre une expression qui désigne la méthode d'une instance du bean.

Exemple :

```
<h:panelGrid columns="2">
  <h:outputText value="Numéro de série : " />
  <h:panelGroup>
    <h:inputText value="#{validation.numeroSerie}" id="numeroSerie"
      required="true" validator="#{validation.valider}" />
    <h:message for="numeroSerie"/>
  </h:panelGroup>
</h:panelGrid>
```

Cette approche est particulièrement utile pour des besoins spécifiques à une application car sa mise en oeuvre est difficilement portable d'une application à une autre.

83.12.5. La validation entre plusieurs composants

De base, le modèle de validation des données proposé par JSF repose sur une validation unitaire de chaque composant. Il est cependant fréquent d'avoir besoin de faire une validation en fonction des données d'un ou plusieurs autres composants.

Pour réaliser ce genre de tâche, il faut créer un backing bean qui aura accès à chacun des composants nécessaires aux traitements et définir dans ce bean une méthode qui va réaliser les traitements de validation.

Exemple :

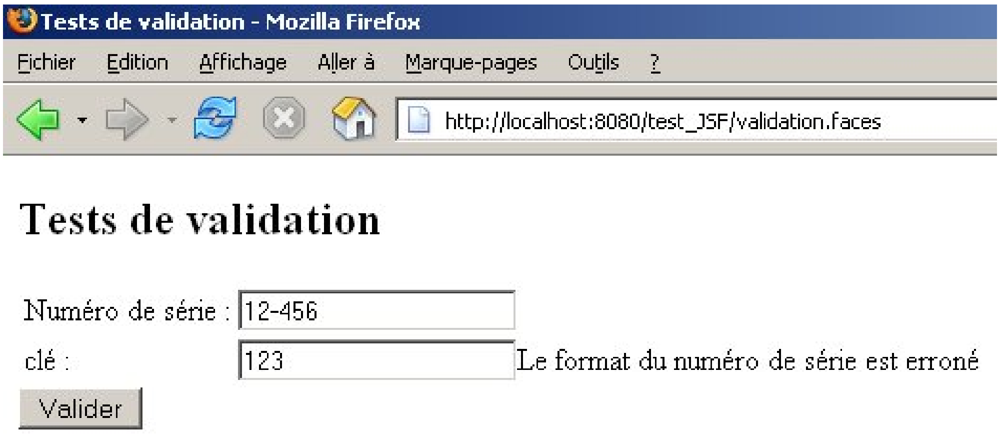
```
package fr.jmdoudoux.dej.jsf;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.component.UIInput;
import javax.faces.context.FacesContext;
import javax.faces.validator.ValidatorException;

import com.sun.faces.util.MessageFactory;

public class Validation {
  public static final String CLE_MESSAGE_VALIDATION_IMPOSSIBLE =
    "message.validation.impossible";

  private String numeroSerie;
  private String cle;
  private UIInput cleInput;
  private UIInput numeroSerieInput;

  public String getNumeroSerie() {
    return numeroSerie;
  }

  public void setNumeroSerie(String numeroSerie) {
    this.numeroSerie = numeroSerie;
  }
  public void valider(FacesContext contexte, UIComponent composant, Object objet) {
    String valeur = null;
    boolean estValide = false;

    if ((contexte == null) || (composant == null)) {
      throw new NullPointerException();
    }
    if (!(composant instanceof UIInput)) {
      return;
    }

    valeur = objet.toString();

    Pattern p = Pattern.compile("[0-9][0-9]-[0-9][0-9][0-9]",Pattern.MULTILINE);
    Matcher m = p.matcher(valeur);
    estValide = m.matches();

    if (!estValide) {
      FacesMessage errMsg = MessageFactory.getMessage(contexte,
      CLE_MESSAGE_VALIDATION_IMPOSSIBLE);
      throw new ValidatorException(errMsg);
    }
  }

  public void validerCle(FacesContext contexte, UIComponent composant, Object objet) {
    System.out.println("validerCle");

    String valeurNumero = numeroSerieInput.getLocalValue().toString();
    String valeurCle = cleInput.getLocalValue().toString();
    boolean estValide = false;
    if (contexte == null) {
      throw new NullPointerException();
    }

    Pattern p = Pattern.compile("[0-9][0-9]-[0-9][0-9][0-9]",Pattern.MULTILINE);
    Matcher m = p.matcher(valeurNumero);
    estValide = m.matches() && valeurCle.equals("789");

    System.out.println("estValide="+estValide);
    if (!estValide) {
      FacesMessage errMsg = MessageFactory.getMessage(contexte,
      CLE_MESSAGE_VALIDATION_IMPOSSIBLE);
      throw new ValidatorException(errMsg);
    }
  }

  public String getCle() {
    return cle;
  }

  public void setCle(String cle) {
    this.cle = cle;
  }

  public UIInput getCleInput() {
    return cleInput;
  }

  public void setCleInput(UIInput cleInput) {
    this.cleInput = cleInput;
  }

  public UIInput getNumeroSerieInput() {
    return numeroSerieInput;
  }

  public void setNumeroSerieInput(UIInput numeroSerieInput) {
    this.numeroSerieInput = numeroSerieInput;
  }
}
```

Il suffit alors d'ajouter un champ caché dans la vue sur lequel la classe de validation sera appliquée.

Exemple :

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ page language="java" %>
```

```
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<f:view>
<head>
  <title>Tests de validation</title>
</head>
<body bgcolor="#FFFFFF">
  <h:form>
  <h2>Tests de validation</h2>

  <h:panelGrid columns="2">
    <h:outputText value="Numéro de série : " />
    <h:panelGroup>
      <h:inputText value="#{validation.numeroSerie}" id="numeroSerie"
        required="true" binding="#{validation.numeroSerieInput}" />
      <h:message for="numeroSerie"/>
    </h:panelGroup>
    <h:outputText value="clé : " />
    <h:panelGroup>
      <h:inputText value="#{validation.cle}" id="cle" binding="#{validation.cleInput}"
        required="true" />
      <h:message for="validationCle"/>
    </h:panelGroup>
  </h:panelGrid>

  <h:inputHidden id="validationCle" validator="#{validation.validerCle}" value="nul"/>

  <h:commandButton value="Valider" action="submit"/>

  </h:form>
</body>
</f:view>
</html>
```


Tests de validation - Mozilla Firefox
Fichier
Edition
Affichage
Aller à
Marque-pages
Outils
?
http://localhost:8080/test_JSF/validation.faces

Tests de validation

Numéro de série : 12-456

clé : 123 Le format du numéro de série est erroné

Valider

83.12.6. L'écriture de tags pour un convertisseur ou un validateur de données

L'écriture d'un tag personnalisé facilite l'utilisation d'un convertisseur ou d'un validateur et permet de lui fournir des paramètres.

Il faut définir une classe nommée handler qui va contenir les traitements du tag. Cette classe doit hériter d'une sous-classe dédiée selon le type d'élément que va représenter le tag :

- ConverterTag : si le tag concerne un convertisseur
- ValidatorTag : si le tag concerne un validateur
- UIComponentTag et UIComponentBodyTag : si le tag concerne un composant

Le handler est un bean dont les propriétés doivent correspondre à chaque attribut défini dans le tag.

Pour pouvoir utiliser un tag personnalisé, il faut définir un fichier .tld.

Ce fichier au format XML défini dans les spécifications des JSP permet de fournir des informations sur la bibliothèque de tags personnalisés notamment la version des spécifications utilisées et des informations sur chaque tag.

Enfin, il est nécessaire de déclarer l'utilisation de la bibliothèque de tags personnalisés dans la JSP.

83.12.6.1. L'écriture d'un tag personnalisé pour un convertisseur

Il faut définir un handler pour le tag qui est un bean héritant de la classe ConverterTag.

Il est important dans le constructeur du handler de faire un appel à la méthode setConverterId() en lui passant un id défini dans le fichier de configuration de l'application JSF.

Il faut redéfinir la méthode release() dont les traitements vont permettre de réinitialiser les propriétés de la classe. Ceci est important lorsque, pour améliorer les performances, on souhaite placer ces objets dans un pool. La méthode release() est dans ce cas utilisée pour recycler les instances du pool non utilisées.

Il faut ensuite redéfinir la méthode createConverter() qui va permettre la création d'une instance du converter en utilisant les éventuels valeurs des attributs du tag.

La valeur fournie à un attribut d'un tag pour être soit un littéral soit une expression dont le contenu devra être évalué au moment de son utilisation.

La suite de ce chapitre sera développée dans une version future de ce document

83.12.6.2. L'écriture d'un tag personnalisé pour un validateur

L'écriture d'un tag personnalisé pour un validateur suit les mêmes règles que pour un convertisseur. La grande différence est que la classe handler doit hériter de la classe ValidatorTag. La méthode à appeler dans le constructeur est la méthode setValidatorId() et la méthode à redéfinir pour créer une instance du validateur est la méthode createValidator().

La suite de ce chapitre sera développée dans une version future de ce document

83.13. La sauvegarde et la restauration de l'état

JSF sauvegarde l'état de chaque élément présent dans la vue : les composants, les convertisseurs, les validateurs, ... pourvu que ceux-ci mettent en oeuvre un mécanisme adéquat.

Ces états sont stockés dans un champ de type hidden dans la vue pour permettre leur échange entre deux requêtes si l'application le prévoit dans le fichier de configuration.

Ce mécanisme peut prendre deux formes selon que :

- la classe qui encapsule l'élément implémente l'interface Serializable
- la classe qui encapsule l'élément implémente l'interface StateHolder

Dans le premier cas, c'est le mécanisme standard de la sérialisation qui sera utilisé. Il nécessite donc très peu voire aucun code particulier si les champs de la classe sont tous d'un type qui est sérialisable.

L'implémentation de l'interface StateHolder nécessite la définition des deux méthodes définies dans l'interface (saveState() et restoreState()) et la présence d'un constructeur par défaut. Cette approche peut être intéressante pour obtenir un contrôle très fin de la sauvegarde et de la restauration de l'état.

La méthode saveState(FacesContext) renvoie un objet sérialisable qui va contenir les données de l'état à sauvegarder. La méthode restoreState(FacesContext, Object) effectue l'opération inverse.

Il est aussi nécessaire de définir une propriété nommée transient de type booléen qui précise si l'état doit être sauvegardé ou non.

Si l'élément n'implémente pas l'interface Serializable ou StateHolder alors son état n'est pas sauvegardé entre deux échanges de la vue.

83.14. Le système de navigation

Une application de type web se compose d'un ensemble de pages dans lequel l'utilisateur navigue en fonction de ses actions.

Un système de navigation standard peut être facilement mis en oeuvre avec JSF grâce à un paramétrage au format XML dans le fichier de configuration de l'application.

Le système de navigation assure la gestion de l'enchaînement des pages en utilisant des actions. Les règles de navigation sont des chaînes de caractères qui sont associées à une page d'origine et qui permettent de déterminer la page de résultat. Toutes ces règles sont contenues dans le fichier de configuration face-config.xml.

La déclaration de ce système de navigation ressemble à celle utilisée dans le framework Struts.

Le système de navigation peut être statique ou dynamique. Dans ce dernier cas, des traitements particuliers doivent être mis en place pour déterminer la cible de la navigation.

Exemple :

```
...
  <navigation-rule>
    <from-view-id>/login.jsp</from-view-id>
    <navigation-case>
      <from-outcome>login</from-outcome>
      <to-view-id>/accueil.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>
...
```

La tag <navigation-rule> permet de préciser des règles de navigation.

La tag <from-view-id> permet de préciser la page concernée. Ce tag n'est pas obligatoire : sans sa présence, il est possible de définir une règle de navigation applicable à toutes les pages JSF de l'application.

Exemple :

```
<navigation-rule>
  <navigation-case>
    <from-outcome>logout</from-outcome>
    <to-view-id>/logout.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Il est aussi possible de désigner un ensemble de pages dans le tag <from-view-id> en utilisant le caractère * dans la valeur du tag. Ce caractère * ne peut être utilisé qu'une seule fois dans la valeur du tag et il doit être en dernière position.

Exemple :

```
<from-view-id>/admin/*</from-view-id>
```

Le tag <navigation-case> permet de définir les différents cas.

La valeur du tag <from-outcome> doit correspondre au nom d'une action.

Le tag <to-view-id> permet de préciser la page qui sera affichée. L'URL fournie comme valeur doit commencer par un slash et doit préciser une page possédant une extension brute (ne surtout pas mettre une URL utilisée par la servlet faisant office de contrôleur).

Le tag <redirect/> inséré juste après le tag <to-view-id> permet au navigateur de l'utilisateur d'effectuer la redirection vers la page indiquée.

La gestion de la navigation est assurée par une instance de la classe NavigationHandler, gérée au niveau de l'application. Ce gestionnaire utilise la valeur d'un attribut action d'un composant pour déterminer la page suivante et faire la redirection vers la page adéquate en fonction des informations fournies dans le fichier de configuration.

La valeur de l'attribut action peut être statique : dans ce cas la valeur est en dur dans le code de la vue

Exemple :

```
<h:commandButton action="login"/>
```

La valeur de l'attribut action peut être dynamique : dans ce cas la valeur est déterminée par l'appel d'une méthode d'un bean

Exemple :

```
<h:commandButton action="#{login.verifierMotDePasse}"/>
```

Dans ce cas, la méthode appelée ne doit pas avoir de paramètre et doit retourner une chaîne de caractères définie dans le fichier de configuration.

Lors des traitements par le NavigationHandler, si aucune action ne trouve de correspondance dans le fichier de configuration pour la page alors la page est simplement réaffichée.

83.15. La gestion des événements

Le modèle de gestion des événements de JSF est similaire à celui utilisé dans les JavaBeans : il repose sur les Listener et les Event pour traiter les événements générés dans les composants graphiques suite aux actions de l'utilisateur.

Un objet de type Event encapsule le composant à l'origine de l'événement et des données relatives à cet événement.

Pour être notifié d'un événement particulier, il est nécessaire d'enregistrer un objet qui implémente l'interface Listener auprès du composant concerné.

Lors de certaines actions de l'utilisateur, un événement est émis.

L'implémentation JSF propose deux types d'événements :

- Value changed : ces événements sont émis lors du changement de la valeur d'un composant de type UIInput, UISelectOne, UISelectMany, et UISelectBoolean
- Action : ces événements sont émis lors d'un clic sur un hyperlien ou un bouton qui sont des composants de type UICommand

JSF propose de transposer le modèle de gestion des événements des interfaces graphiques des applications standalone aux applications de type web utilisant JSF.

La gestion des événements repose donc sur deux types d'objets

- Event : classe qui encapsule l'événement lui-même
- Listener : classe qui va encapsuler les traitements à réaliser pour un type d'événement

Comme pour les interfaces graphiques des applications standalone, la classe de type Listener doit s'enregistrer auprès du composant concerné. Lorsque celui-ci émet un événement suite à une action de l'utilisateur, il appelle le Listener enregistré en lui fournissant en paramètre un objet de type Event.

Exemple :

```
<h:selectOneMenu ... valueChangeListener="#{choixLangue.langueChangement}">
  ...
</h:selectOneMenu>
```

JSF supporte trois types d'événements :

- les changements de valeurs : concernent les composants qui permettent la saisie ou la sélection d'une valeur lorsque cette valeur change
- les actions : concernent un clic sur un bouton (commandButton) ou un lien (commandLink)
- les événements liés au cycle de vie : ils sont émis par le framework JSF durant le cyle de vie des traitements

Les traitements des listeners peuvent affecter la suite du cycle de vie de plusieurs manières :

- par défaut, laisser les traitements se poursuivre
- demander l'exécution immédiate de la dernière étape en utilisant la méthode FacesContext.renderResponse()
- arrêter les traitements du cyle de vie en utilisant la méthode FacesContext.responseComplete()

83.15.1. Les événements liés à des changements de valeur

Il y a deux façons de préciser un listener de type valueChangeListener sur un composant :

- utiliser l'attribut valueChangeListener
- utiliser le tag valueChangeListener

L'attribut valueChangeListener permet de préciser, par une expression, la méthode exécutée durant les traitements du cyle de vie de la requête. Pour que ces traitements puissent être déclenchés, il faut soumettre la page.

Exemple :

```
<h:selectOneMenu value="#{choixLangue.langue}" onchange="submit()"
  valueChangeListener="#{choixLangue.langueChangement}">
  <f:selectItems value="#{choixLangue.langues}"/>
</h:selectOneMenu>
```

La méthode ne renvoie aucune valeur et attend en paramètre un objet de type ValueChangeEvent.

Exemple :

```
package fr.jmdoudoux.dej.jsf;

import java.util.Locale;
import javax.faces.context.FacesContext;
import javax.faces.event.ValueChangeEvent;
import javax.faces.model.SelectItem;

public class ChoixLangue {
  private static final String LANGUE_FR = "Français";
  private static final String LANGUE_EN = "Anglais";
  private String langue = LANGUE_FR;

  private SelectItem[] langueItems = {
    new SelectItem(LANGUE_FR, "Français"),
    new SelectItem(LANGUE_EN, "Anglais") };

  public SelectItem[] getLangues() {
    return langueItems;
  }

  public String getLangue() {
    return langue;
  }

  public void setLangue(String langue) {
    this.langue = langue;
  }

  public void langueChangement(ValueChangeEvent event) {
    FacesContext context = FacesContext.getCurrentInstance();
    System.out.println("Changement de la langue : "+event.getNewValue());
    if (LANGUE_FR.equals((String) event.getNewValue()))
      context.getViewRoot().setLocale(Locale.FRENCH);
    else
      context.getViewRoot().setLocale(Locale.ENGLISH);
    }
  }
}
```

La classe ValueChangeEvent possède plusieurs méthodes utiles :

Méthode	Rôle
UIComponent getComponent()	renvoie le composant qui a généré l'événement
Object getNewValue()	renvoie la nouvelle valeur (convertie et validée)
Object getOldValue()	renvoie la valeur précédente

Le tag valueChangeListener permet aussi de préciser un listener. Son attribut type permet de préciser une classe implémentant l'interface ValueChangeListener.

Exemple :

```
<h:selectOneMenu value="#{choixLangue.langue}" onchange="submit()">
  <f:valueChangeListener type="fr.jmdoudoux.dej.jsf.ChoixLangueListener"/>
  <f:selectItems value="#{choixLangue.langues}"/>
</h:selectOneMenu>
```

Une telle classe doit définir une méthode processValueChange() qui va contenir les traitements exécutés en réponse à l'événement.

Exemple :

```
package fr.jmdoudoux.dej.jsf;

import java.util.Locale;

import javax.faces.context.FacesContext;
import javax.faces.event.AbortProcessingException;
import javax.faces.event.ValueChangeEvent;
import javax.faces.event.ValueChangeListener;

public class ChoixLangueListener implements ValueChangeListener {
  private static final String LANGUE_FR = "Français";

  private static final String LANGUE_EN = "Anglais";

  public void processValueChange(ValueChangeEvent event)
    throws AbortProcessingException {
    FacesContext context = FacesContext.getCurrentInstance();
    System.out.println("Changement de la langue : " + event.getNewValue());
    if (LANGUE_FR.equals((String) event.getNewValue()))
      context.getViewRoot().setLocale(Locale.FRENCH);
    else
      context.getViewRoot().setLocale(Locale.ENGLISH);
    }
  }
}
```

83.15.2. Les événements liés à des actions

Les actions sont des clics sur des boutons ou des liens. Le clic sur un composant de type commandLink ou commandButton déclenche automatiquement la soumission de la page.

Il y a deux façons de préciser un listener de type actionListener sur un composant :

- utiliser l'attribut actionListener
- utiliser le tag actionListener

L'attribut actionListener permet de préciser, par une expression, la méthode exécutée durant les traitements du cyle de vie de la requête.

Exemple :

```
<table align="center" width="50%">
  <tr>
    <td width="50%"><h:commandButton image="images/bouton_valider.gif"
      actionListener="#{saisieDonnees.traiterAction}"
      id="Valider" />
    </td>
    <td><h:commandButton image="images/bouton_annuler.gif"
      actionListener="#{saisieDonnees.traiterAction}"
      id="Annuler"/>
    </td>
  </tr>
</table>
```

Cette méthode attend en paramètre un objet de type ActionEvent.

Exemple :

```
package fr.jmdoudoux.dej.jsf;

import javax.faces.context.FacesContext;
import javax.faces.event.ActionEvent;

public class SaisieDonnees {

  public void traiterAction(ActionEvent e) {

    FacesContext context = FacesContext.getCurrentInstance();

    String clientId = e.getComponent().getClientId(context);
    System.out.println("traiterAction : clientId=" + clientId);

  }
}
```

Le tag valueChangeListener permet aussi de préciser un listener. Son attribut type permet de préciser une classe implémentant l'interface ValueChangeListener.

Exemple :

```
<table align="center" width="50%">
  <tr>
    <td width="50%"><h:commandButton image="images/bouton_valider.gif" id="Valider" >
        <f:actionListener type="fr.jmdoudoux.dej.jsf.SaisieDonneesListener"/>
      </h:commandButton>
      </td>
    <td><h:commandButton image="images/bouton_annuler.gif" id="Annuler">
        <f:actionListener type="fr.jmdoudoux.dej.jsf.SaisieDonneesListener"/>
      </h:commandButton>
    </td>
  </tr>
</table>
```

Une telle classe doit définir la méthode processAction() définie dans l'interface.

Exemple :

```
package fr.jmdoudoux.dej.jsf;

import javax.faces.context.FacesContext;
import javax.faces.event.AbortProcessingException;
import javax.faces.event.ActionEvent;
import javax.faces.event.ActionListener;

public class SaisieDonneesListener implements ActionListener {

  public void processAction(ActionEvent e) throws AbortProcessingException {
    FacesContext context = FacesContext.getCurrentInstance();

    String clientId = e.getComponent().getClientId(context);
    System.out.println("processAction : clientId=" + clientId);

  }
}
```

83.15.3. L'attribut immediate

L'attribut immediate permet de demander les traitements immédiats des listeners.

Par exemple, sur une page un composant possède un attribut required et un second possède un listener. Les traitements du second doivent pouvoir être réalisés sans que le premier composant n'affiche un message d'erreur lié à sa validation.

Le cycle de traitement de la requête est modifié lorsque l'attribut immediate est positionné dans un composant. Dans ce cas, les données du composant sont converties et validées si nécessaire puis les traitements du listener sont exécutés à la place de l'étape « Process validations » (juste après l'étape Apply Request Value).

Exemple :

```
<h:selectOneMenu value="#{choixLangue.langue}" onchange="submit()" immediate="true">
  <f:valueChangeListener type="fr.jmdoudoux.dej.jsf.ChoixLangueListener"/>
  <f:selectItems value="#{choixLangue.langues}"/>
</h:selectOneMenu>
```

Par défaut, ceci modifie l'ordre d'exécution des traitements du cycle de vie mais n'empêche pour les traitements prévus de s'exécuter. Pour les inhiber, il est nécessaire de demande au framework JSF d'interrompre les traitements du cycle de vie en utilisant la méthode renderResponse() du FaceCcontext.

Exemple :

```
  public void langueChangement(ValueChangeEvent event) {
    FacesContext context = FacesContext.getCurrentInstance();
    System.out.println("Changement de la langue : " + event.getNewValue());
    if (LANGUE_FR.equals((String) event.getNewValue())) {
      context.getViewRoot().setLocale(Locale.FRENCH);
    } else {
      context.getViewRoot().setLocale(Locale.ENGLISH);
    }
    context.renderResponse();
  }
```

Le mode de fonctionnement est le même avec les actionListener hormis le fait que l'appel à la méthode renderResponse() est inutile puisqu'il est automatiquement fait par le framework.

83.15.4. Les événements liés au cycle de vie

Le framework émet des événements avant et après chaque étape du cycle de vie des requêtes. Ils sont traités par des phaseListeners.

L'enregistrement d'un phaseListener se fait dans le fichier de configuration dans un tag fils <phase-listener> fils du tag <lifecycle> qui doit contenir le nom pleinement qualifié d'une classe.

Exemple :

```
<faces-config>
...

  <lifecycle>
    <phase-listener>fr.jmdoudoux.dej.jsf.PhasesEcouteur</phase-listener>
  </lifecycle>

</faces-config>
```

La classe précisée doit implémenter l'interface javax.faces.event.PhaseListener qui définit trois méthodes :

- getPhaseId() : renvoie un objet de type PhaseId qui permet de préciser à quelle phase ce listener correspond
- beforePhase() : traitements à exécuter avant l'exécution de la phase
- afterPhase() : traitements à exécuter après l'exécution de la phase

La classe PhaseId définit des constantes permettant d'identifier chacune des phases : PhaseId.RESTORE_VIEW, PhaseId.APPLY_REQUEST_VALUES, PhaseId.PROCESS_VALIDATIONS, PhaseId.UPDATE_MODEL_VALUES, PhaseId.INVOKE_APPLICATION et PhaseId.RENDER_RESPONSE

Elle définit aussi la constante PhaseId.ANY_PHASE qui permet de demander l'application du listener à toutes les phases. Cela peut être très utile lors du débogage.

Exemple :

```
package fr.jmdoudoux.dej.jsf;

import javax.faces.event.PhaseEvent;
import javax.faces.event.PhaseId;
import javax.faces.event.PhaseListener;

public class PhasesEcouteur implements PhaseListener {

  public void afterPhase(PhaseEvent pe) {
    System.out.println("Apres " + pe.getPhaseId());
  }

  public void beforePhase(PhaseEvent pe) {
    System.out.println("Avant " + pe.getPhaseId());
  }
  public PhaseId getPhaseId() {
    return PhaseId.ANY_PHASE;
  }
}
```

Lors de l'appel de la première page de l'application, les informations suivantes sont affichées dans la sortie standard :

```
Avant RESTORE_VIEW 1
Apres RESTORE_VIEW 1
Avant RENDER_RESPONSE 6
Apres RENDER_RESPONSE 6
```

Lors d'une soumission de cette page avec une erreur de validation des données, les informations suivantes sont affichées dans la sortie standard :

```
Avant RESTORE_VIEW 1
Apres RESTORE_VIEW 1
Avant APPLY_REQUEST_VALUES 2
Apres APPLY_REQUEST_VALUES 2
Avant PROCESS_VALIDATIONS 3
Apres PROCESS_VALIDATIONS 3
Avant RENDER_RESPONSE 6
Apres RENDER_RESPONSE 6
```

Lors d'une soumission de cette page sans erreur de validation des données, les informations suivantes sont affichées dans la sortie standard :

```
Avant RESTORE_VIEW 1
Apres RESTORE_VIEW 1
Avant APPLY_REQUEST_VALUES 2
Apres APPLY_REQUEST_VALUES 2
Avant PROCESS_VALIDATIONS 3
Apres PROCESS_VALIDATIONS 3
Avant UPDATE_MODEL_VALUES 4
Apres UPDATE_MODEL_VALUES 4
Avant INVOKE_APPLICATION 5
Apres INVOKE_APPLICATION 5
Avant RENDER_RESPONSE 6
Apres RENDER_RESPONSE 6
```

83.16. Le déploiement d'une application

Une application utilisant JSF s'exécute dans un serveur d'applications contenant un conteneur web implémentant les spécifications servlet 1.3 et JSP 1.2 minimum. Une telle application doit être packagée dans un fichier .war.

La compilation des différentes classes de l'application nécessite l'ajout dans le classpath de la bibliothèque servlet.

Elle nécessite aussi l'ajout dans le classpath de la bibliothèque jsf-api.jar de la ou des bibliothèques requises par l'implémentation JSF utilisée.

Ces bibliothèques doivent aussi être disponibles pour le conteneur web qui va exécuter l'application. Le plus simple est de mettre ces fichiers dans le répertoire WEB-INF/lib.

83.17. Un exemple d'application simple

Cette section va développer une petite application constituée de deux pages. La première va demander le nom de l'utilisateur et la seconde afficher un message de bienvenue.

Il faut créer un répertoire, par exemple nommé Test_JSF et créer à l'intérieur la structure de l'application qui correspond à la structure de toute application Web selon les spécifications J2EE, notamment le répertoire WEB-INF avec ses sous-répertoires lib et classes.

Il faut ensuite copier les fichiers nécessaires à une utilisation de JSF dans l'application web.

Il suffit de copier les fichiers *.jar du répertoire lib de l'implémentation de référence vers le répertoire WEB-INF/lib du projet.

Il faut créer un fichier à la racine du projet et le nommer index.htm

Exemple :

```
<html>
  <head>
    <meta http-equiv="Refresh" content= "0; URL=login.faces"/>
    <title>Demarrage de l'application</title>
  </head>
  <body>
    <p>D&eacute;marrage de l'application ...</p>
  </body>
</html>
```

Il faut créer un fichier à la racine du projet et le nommer login.jsp :

Exemple :

```
<html>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view>
<head>
  <title>Application de tests avec JSF</title>
</head>
<body>
  <h:form>
    <h3>Identification</h3>
    <table>
      <tr>
        <td>Nom : </td>
        <td><h:inputText value="#{login.nom}"/></td>
      </tr>
      <tr>
        <td>Mot de passe :</td>
        <td><h:inputSecret value="#{login.mdp}"/></td>
      </tr>
      <tr>
        <td colspan="2"><h:commandButton value="Login" action="login"/></td>
      </tr>
    </table>
  </h:form>
</body>
</f:view>
</html>
```

Il faut créer une nouvelle classe nommée fr.jmdoudoux.dej.jsf.LoginBean et la compiler dans le répertoire WEB-INF/classes.

Exemple :

```
package fr.jmdoudoux.dej.jsf;

public class LoginBean {

  private String nom;
  private String mdp;

  public String getMdp() {
    return mdp;
  }

  public String getNom() {
    return nom;
  }

  public void setMdp(String string) {
    mdp = string;
  }

  public void setNom(String string) {
    nom = string;
  }
}
```

Il faut créer un fichier à la racine du projet et le nommer accueil.jsp : cette page contiendra la page d'accueil de l'application.

Il faut créer un fichier dans le répertoire /WEB-INF et le nommer faces-config.xml :

Exemple :

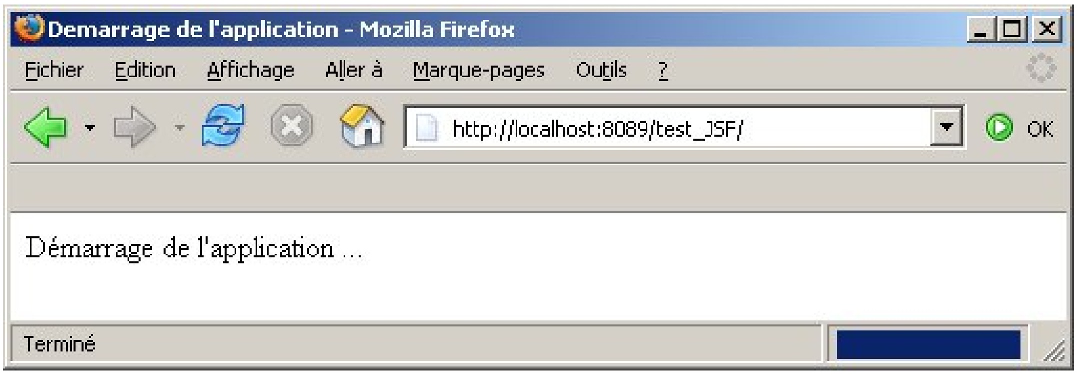
```
<?xml version="1.0"?>

<!DOCTYPE faces-config PUBLIC
"-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
"http://java.sun.com/dtd/web-facesconfig_1_0.dtd">

<faces-config>
  <navigation-rule>
    <from-view-id>/login.jsp</from-view-id>
    <navigation-case>
      <from-outcome>login</from-outcome>
      <to-view-id>/accueil.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>

  <managed-bean>
    <managed-bean-name>login</managed-bean-name>
    <managed-bean-class>fr.jmdoudoux.dej.jsf.LoginBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>

</faces-config>
```

Il faut créer un fichier dans le répertoire /WEB-INF et le nommer web.xml :

Exemple :

```
<?xml version="1.0"?>

<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.faces</url-pattern>
  </servlet-mapping>

  <welcome-file-list>
    <welcome-file>index.htm</welcome-file>
  </welcome-file-list>

</web-app>
```

Il suffit alors de démarrer Tomcat, puis d'ouvrir un navigateur et taper l'URL http://localhost:8089/test_JSF/ (en remplaçant le port 8089 par celui défini dans Tomcat).

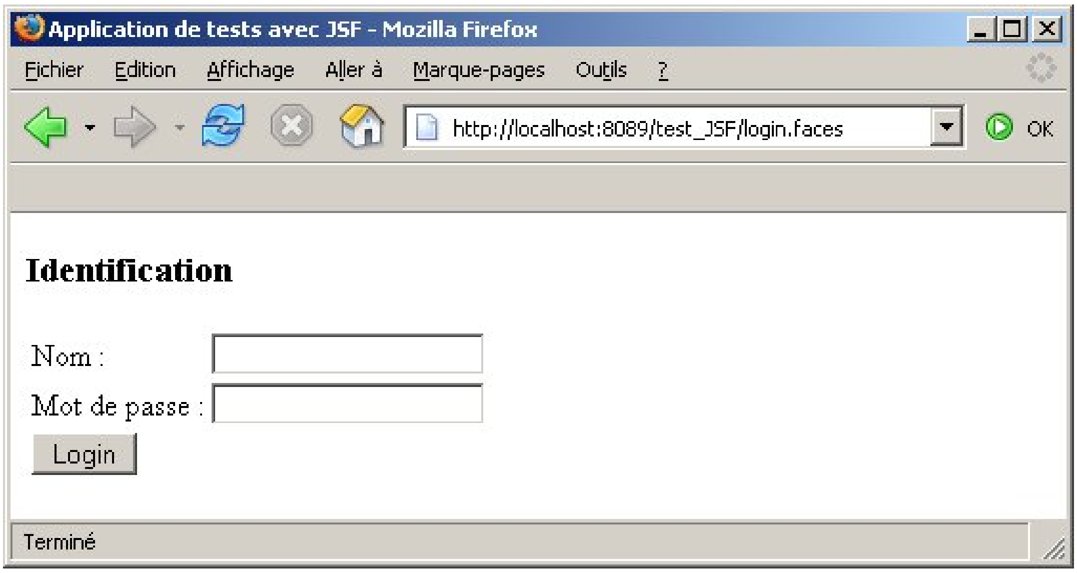


http://localhost:8089/test_JSF/

Démarrage de l'application ...

Terminé

Une fois l'application démarrée, la page de login s'affiche



http://localhost:8089/test_JSF/login.faces

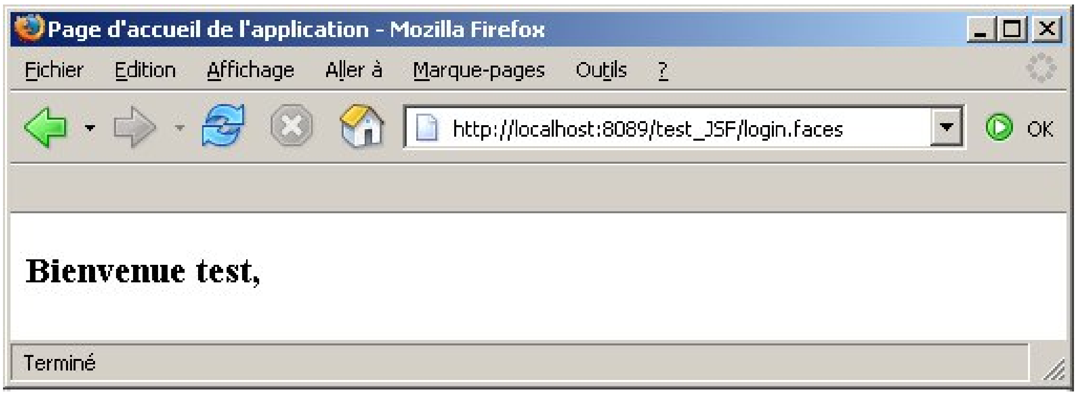
Identification

Nom :

Mot de passe :

Terminé

Il faut saisir un nom par exemple test et cliquer sur le bouton « Login ».



http://localhost:8089/test_JSF/login.faces

Bienvenue test,

Terminé

Cette exemple ne met en aucune façon en valeur la puissance de JSF mais permet simplement de mettre en place les éléments minimum pour une application l'utilisant.

83.18. L'internationalisation

JSF propose des fonctionnalités qui facilitent l'internationalisation d'une application.

Il faut définir un fichier au format properties qui va contenir la définition des chaînes de caractères. Un tel fichier possède les caractéristiques suivantes :

- le fichier doit avoir l'extension .properties
- il doit être dans le classpath de l'application
- il est composé d'une paire clé=valeur par ligne. La clé permet d'identifier de façon unique la chaîne de caractères

Exemple : le fichier msg.properties

```
login_titre=Application de tests avec JSF
login_identification=Identification
login_nom=Nom
login_mdp=Mot de passe
login_Login=Valider
```

Ce fichier correspond à la langue par défaut. Il est possible de définir d'autres fichiers pour d'autres langues. Ces fichiers doivent avoir le même nom suivi d'un underscore et du code langue défini par le standard ISO 639 avec toujours l'extension .properties.

Exemple :

```
msg.properties
msg_en.properties
msg_de.properties
```

Il faut bien sûr remplacer les valeurs de chaque chaîne par leurs traductions correspondantes.

Exemple :

```
login_titre=Tests of JSF
login_identification=Login
login_nom=Name
login_mdp=Password
login_Login=Login
```

Les langues disponibles doivent être précisées dans le fichier de configuration.

Exemple :

```
<faces-config>
...
  <application>
    <locale-config>
      <default-locale>fr</default-locale>
      <supported-locale>en</supported-locale>
    </locale-config>
  </application>
...
</faces-config>
```

Pour utiliser l'internationalisation dans les vues, il faut utiliser le tag <f:loadBundle> pour charger le fichier .properties nécessaire. Deux attributs de ce tags sont requis :

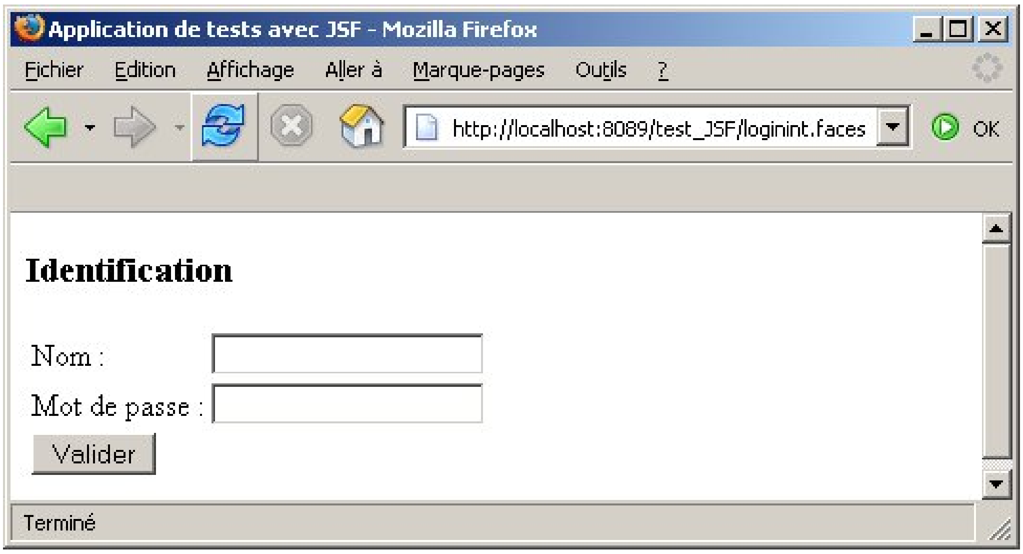
- basename : précise la localisation et le nom de base des fichiers .properties. La notation de la localisation est similaire à celle utilisée pour les packages
- var : précise le nom de la variable qui va contenir les chaînes de caractères

Il ne reste plus qu'à utiliser la variable définie en utilisant la notation avec un point pour la clé de la chaîne dont on souhaite utiliser la valeur.

Exemple :

```
<html>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view>
<f:loadBundle basename="fr.jmdoudoux.dej.jsf.msg" var="msg"/>

<head>
  <title><h:outputText value="#{msg.login_titre}"/></title>
</head>
<body>
  <h:form>
    <h3><h:outputText value="#{msg.login_identification}"/></h3>
    <table>
      <tr>
        <td><h:outputText value="#{msg.login_nom}"/> : </td>
        <td><h:inputText value="#{login.nom}"/></td>
      </tr>
      <tr>
        <td><h:outputText value="#{msg.login_mdp}"/> :</td>
        <td><h:inputSecret value="#{login.mdp}"/></td>
      </tr>
      <tr>
        <td colspan="2"><h:commandButton value="#{msg.login_Login}" action="login"/></td>
      </tr>
    </table>
  </h:form>
</body>
</f:view>
</html>
```

La langue à utiliser est déterminée automatiquement par JSF en fonction des informations contenues dans la propriété Accept-Language de l'en-tête de la requête et du fichier de configuration.


Application de tests avec JSF - Mozilla Firefox
Fichier
Edition
Affichage
Aller à
Marque-pages
Outils
?
http://localhost:8089/test_JSF/loginint.faces
OK

Identification

Nom :

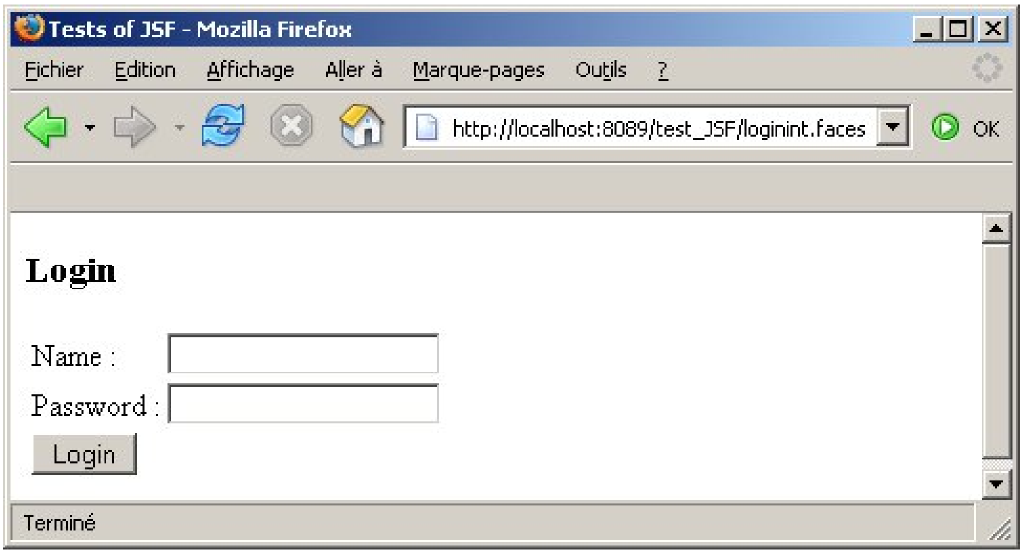
Mot de passe :

Valider

La langue peut aussi être forcée dans l'objet de type view en précisant le code langue dans l'attribut locale.

Exemple :

```
...
          <f:view locale="en">
...
```


Tests of JSF - Mozilla Firefox
Fichier Edition Affichage Aller à Marque-pages Outils ?
http://localhost:8089/test_JSF/loginint.faces
OK
Login
Name :
Password :
Login
Terminé

Elle peut aussi être déterminée dans le code des traitements. L'exemple suivant va permettre à l'utilisateur de sélectionner la langue utilisée entre Français et Anglais grâce à deux petites icônes cliquables.

Exemple :

```
<html>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view>
<f:loadBundle basename="fr.jmdoudoux.dej.jsf.Messages" var="msg"/>
<head>
<title>Application de tests avec JSF</title>
</head>
<body>
  <h:form>

  <table>
    <tr>
      <td>
        <h:commandLink action="#{langueApp.activerFR}" immediate="true">
          <h:graphicImage value="images/francais.jpg" style="border: 0px"/>
        </h:commandLink>
      </td>
      <td>
        <h:commandLink action="#{langueApp.activerEN}" immediate="true">
          <h:graphicImage value="images/anglais.jpg" style="border: 0px"/>
        </h:commandLink>
      </td>
      <td width="100%">&nbsp;</td>
    </tr>
  </table>

  <h3><h:outputText value="#{msg.login_titre}" /></h3>
  <p>&nbsp;</p>
  <h:panelGrid columns="2">
    <h:outputText value="#{msg.login_nom}" />
    <h:panelGroup>
      <h:inputText value="#{login.nom}" id="nom" required="true"
        binding="#{login.inputTextNom}"/>
      <h:message for="nom"/>
    </h:panelGroup>
    <h:outputText value="#{msg.login_mdp}" />
    <h:inputSecret value="#{login.mdp}"/>
    <h:commandButton value="#{msg.login_valider}" action="login"/>
  </h:panelGrid>

  </h:form>
  </body>
</f:view>
</html>
```

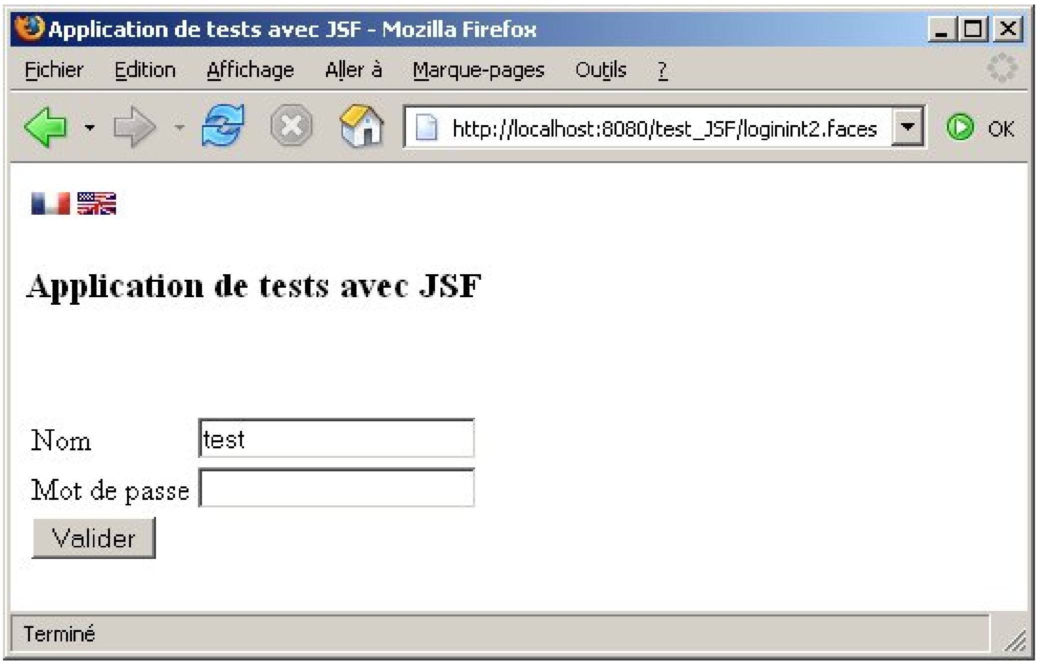
Ce code n'a rien de particulier si ce n'est l'utilisation de l'attribut immediate sur les liens sur le choix de la langue pour empêcher la validation des données lors d'un changement de la langue d'affichage.

Ce sont les deux méthodes du bean qui se chargent de modifier la Locale par défaut du contexte de l'application.

Exemple :

```
package fr.jmdoudoux.dej.jsf;

import java.util.Locale;

import javax.faces.context.FacesContext;

public class LangueApp {

  public String activerFR() {
    FacesContext context = FacesContext.getCurrentInstance();
    context.getViewRoot().setLocale(Locale.FRENCH);
    return null;
  }

  public String activerEN() {
    FacesContext context = FacesContext.getCurrentInstance();
    context.getViewRoot().setLocale(Locale.ENGLISH);
    return null;
  }
}
```

Lors de l'exécution, la page s'affiche en français par défaut.

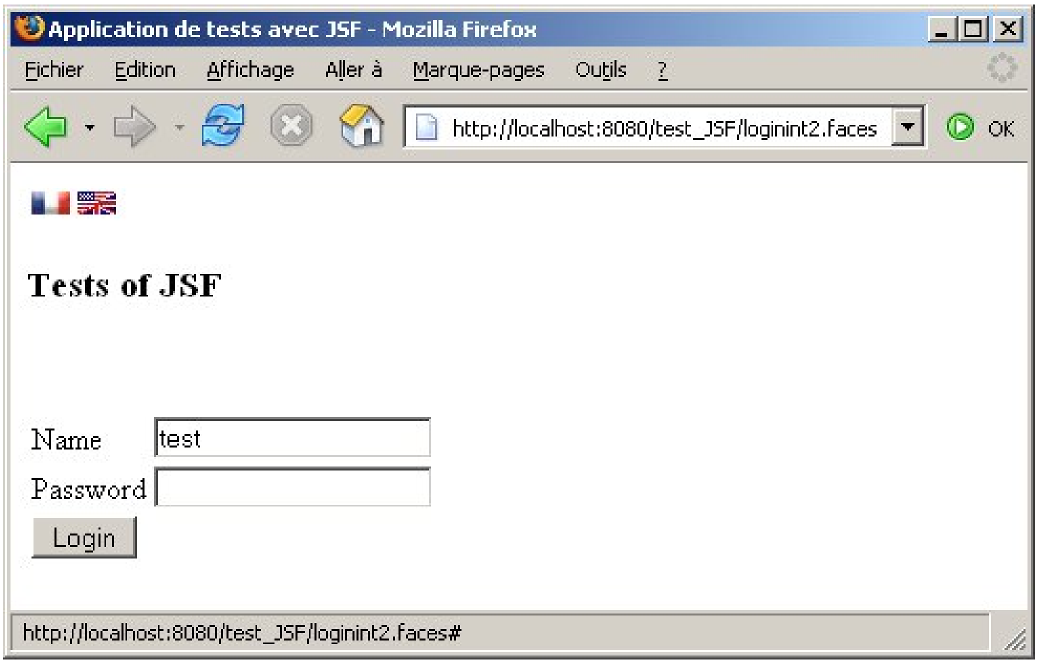

Application de tests avec JSF - Mozilla Firefox
Fichier
Edition
Affichage
Aller à
Marque-pages
Outils
?
http://localhost:8080/test_JSF/logint2.faces
OK

Application de tests avec JSF

Nom test

Mot de passe

Lors d'un clic sur la petite icône indiquant la langue anglaise, la page est réaffichée en anglais.


Application de tests avec JSF - Mozilla Firefox
Fichier Edition Affichage Aller à Marque-pages Outils ?
http://localhost:8080/test_JSF/loginint2.faces
OK

Tests of JSF

Name test

Password

Login

http://localhost:8080/test_JSF/loginint2.faces#

83.19. Les points faibles de JSF

Malgré ses nombreux points forts, JSF présente aussi quelques points faibles :

- Maturité de la technologie

  JSF est une technologie récente qui nécessite l'écriture de beaucoup de code. Bien que prévu pour être utilisé avec des utilitaires facilitant la rédaction de la majeure partie de ce code, à l'heure actuelle, seuls quelques outils supportent JSF.

- Manque de composants évolués en standard

  L'implémentation standard ne propose que des composants simples dont la plupart ont une correspondance directe en HTML. Hormis le composant dataTable aucun composant évolué n'est proposé en standard dans la version 1.0. Il est donc nécessaire de développer ses propres composants ou d'acquérir les composants nécessaires auprès de tiers.

- Consommation en ressources d'une application JSF

  L'exécution d'une application JSF est assez gourmande en ressource notamment mémoire à cause du mode de fonctionnement du cycle de traitement d'une page. Ce cycle de vie inclut la création en mémoire d'une arborescence des composants utilisés lors des différentes étapes des traitements.

- Le rendu des composants uniquement en HTML en standard

  Dans l'implémentation de référence le rendu des composants est uniquement possible en HTML alors que JSF intègre un système de rendu (Renderer) découplé des traitements des composants. Pour un rendu différent de HTML, il est nécessaire de développer ses propres Renderers ou d'acquérir un système de rendu auprès de tiers.

La suite de ce chapitre sera développée dans une version future de ce document

84. D'autres frameworks pour les applications web

Chapitre 84

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

En plus des solutions officielles de la plate-forme Java EE, servlets/JSP et JSF, l'écosystème Java dispose de nombreuses solutions pour le développements d'applications web. Ce chapitre propose une liste non exhaustive de ces solutions.

Ce chapitre contient plusieurs sections :

- Les frameworks pour les applications web
- Les moteurs de templates

84.1. Les frameworks pour les applications web

La communauté open source est très prolifique et propose des frameworks pour le développements d'applications web.

84.1.1. Tapestry

Tapestry est un framework orienté composants développé par la fondation Apache.

Le site officiel de ce projet est à l'url : tapestry.apache.org/.

84.1.2. Spring MVC

Spring MVC est l'implémentation d'un framework reposant sur MVC pour le développement d'applications web.

Le site officiel des projets Spring est à l'url : spring.io.

84.1.3. Play Framework

Play Framework est un framework créé par Guillaume Bort qui permet d'avoir une grande productivité.

Il se distingue des autres frameworks grâce à une approche et des fonctionnalités singulières particulièrement intéressantes :

- Ne repose pas sur la technologie servlet
- Pas de déploiement : compilation incrémentale gràce au compilateur d'Eclipse
- RestFull et stateless
- utilise un système de template reposant sur Groovy
- extensible par plugins

Play remet en cause certaines manières courantes de faire pour augmenter la productivité :

- utilisation de son propre modèle reposant sur JPA où les POJO héritent de la classe Model avec des champs public
- Les contrôleurs sont invoqués grâce à une table de routage : ils permettent différents rendus
- la vue est définie avec des templates

Plusieurs versions de Play ont été diffusées :

- version 1.0 en octobre 2009
- version 1.1 en octobre 2011
- version 2.0 en novembre 2012
- version 2.1 en février 2013
- version 2.2 en septembre 2013
- version 2.3 en mai 2014
- version 2.4 en mai 2015
- version 2.5 en mars 2016
- version 2.6 en juin 2017
- version 2.7 en février 2019
- version 2.8 en décembre 2019

Le site officiel de ce projet est à l'url : https://www.playframework.com.

84.1.4. Wicket

Wicket est un framework orienté composants. La partie présentation utilise des pages XHTML où les composants sont référencés par des identifiants.

Le site officiel de ce projet est à l'url : wicket.apache.org/.

84.1.5. ZK

Le site officiel de ce projet est à l'url : www.zkoss.org.

84.2. Les moteurs de templates

Il existe plusieurs moteurs de templates open source développés et utilisables en Java. Ils permettent d'associer dynamiquement un modèle statique avec des données pour générer un fichier qui peut être un document, une page web, ...

84.2.1. WebMacro

Webmacro est un moteur de template open source.

Le site officiel de Webmacro est à l'url : https://sourceforge.net/projects/webmacro/

84.2.2. FreeMarker

FreeMarker est un moteur de template open source développé en Java. Il permet facilement de générer des documents textuels (HTML, RTF, XML, code source, ...).

Le site officiel de FreeMarker est à l'url : https://freemarker.sourceforge.net/

84.2.3. Velocity

Velocity est un moteur de template open source développé en Java par la fondation Apache.

velocity.apache.org

84.2.4. StringTemplate

StringTemplate est un moteur de templates écrit en Java qui permet de générer des documents de type texte.

www.stringtemplate.org

Partie 12 : Le développement d'applications RIA / RDA

Partie 12 :

Développement

d'applications

RIA / RDA

Cette partie est consacrée au développement d'applications de type RIA (Rich Internet Application) et RDA (Rich Desktop Application). Ce type d'applications était déjà réalisable respectivement avec les technologies applets et Java Web Start. D'autres technologies standard ou open source sont apparues pour fournir de nouveaux moyens de les développer.

Cette partie contient plusieurs chapitres :

- Les applications riches de type RIA et RDA : présente les caractéristiques des applications riches et les principales solutions qui permettent de les développer.
- Les applets : plonge au coeur des premières applications qui ont rendu Java célèbre
- Java Web Start (JWS) : est une technologie qui permet le déploiement d'applications clientes riches à travers le réseau via un navigateur
- AJAX : présente ce concept qui permet de rendre les applications web plus conviviales et plus dynamiques. Le framework open source DWR est aussi détaillé.
- GWT (Google Web Toolkit) : GWT est un framework pour le développement d'applications de type RIA

85. Les applications riches de type RIA et RDA

Chapitre 85

Niveau : 🟩🟩⬜⬜⬜⬜ Elémentaire

Les applications de types client / serveur offrent une bonne ergonomie pour les utilisateurs mais possèdent de nombreux inconvénients notamment au niveau de la maintenance et surtout du déploiement.

Pour pallier ces inconvénients, les applications web se sont répandues. Elles reposent sur des traitements métier côté serveur et une IHM sur un client léger utilisant un simple navigateur web. Malheureusement, ce type d'application ne satisfait pas les utilisateurs notamment parce qu' elles représentent un régression au niveau de l'ergonomie et des interactions.

Les applications riches tentent de réconcilier les avantages des applications C/S et web en conservant le meilleur des deux types d'applications : facilité de déploiement, ergonomie et expérience utilisateur enrichie.

Le développement d'applications web avec Java met généralement en oeuvre un framework reposant sur le modèle MVC tel que Struts ou Spring MVC qui génère sur le serveur des pages HTML retournées au navigateur de l'utilisateur.

Généralement ces frameworks imposent de transmettre une requête http vers le serveur qui regénère toute la page pour tenir compte des modifications ou redirige vers une autre page. Ceci implique des limitations dans les possibilités offertes par les applications en terme d'expérience utilisateur.

Ces limitations sont influencées par les capacités des navigateurs :

- Non support complet ni homogène des standards (HTML, CSS, ...)
- Incompatibilité de JavaScript entre les différents navigateurs
- Certains composants graphiques nécessitent parfois d'être réécrits (onglets, pagination de données, wizard, treeview, ...)
- La sauvegarde de l'état d'une application repose généralement sur les cookies
- ...

Les applications de type RIA proposent une solution pour fournir aux applications exécutées dans un navigateur une expérience utilisateur proche de celle des applications standalone en proposant des fonctionnalités étendues notamment :

- Des composants graphiques évolués sont proposés (barre de menu, onglets, treeview, grille de données, ...)
- Support du drag and drop
- Support multi navigateur avec le même code
- Une meilleure réactivité grâce à un rafraichissement partiel de la page. Les requêtes http adressées au serveur ne concernent que les données à modifier dans la page. Le format utilisé peut varier selon les solutions utilisées : XML, JSON, ...
- Maintien de l'état de l'application côté client
- Un enrichissement des fonctionnalités graphiques notamment grâce à des effets visuels et une intégration forte du multimédia
- ...

Les applications riches peuvent être regroupées dans deux grandes catégories :

- RIA : Rich Internet Applications
- RDA : Rich Desktop Applications

Ce chapitre contient plusieurs sections :

- Les applications de type RIA
- Les applications de type RDA
- Les contraintes
- Les solutions RIA
- Les solutions RDA

85.1. Les applications de type RIA

Les applications de type RIA utilisent un navigateur pour la partie IHM de l'application. Pour permettre d'améliorer l'expérience utilisateur des applications, elles utilisent des technologies existantes depuis longtemps mais partiellement ou pas du tout exploitées. C'est notamment le cas de la technologie AJAX (Asynchronous JavaScript And Xml).

Il y a plusieurs solutions pour mettre en oeuvre Ajax :

- Tout développer manuellement en utilisant JavaScript et DHTML
- Utiliser des bibliothèques de composants telles que Prototype, Script.aculo.us, Dojo, Yahoo ! UI, Rico, Rialto, Ext, jQuery, ...
- Utiliser des frameworks tels que DWR

Les applications RIA peuvent utiliser uniquement les possibilités du navigateur ou avoir besoin d'un plug-in qui fournit un environnement d'exécution.

Les RIA ont cependant un certain nombre d'inconvénients :

- La multitude des solutions proposées et leur immaturité
- Les utilisateurs doivent adapter leur mode de navigation
- L'accessibilité est rarement assurée d'autant que ces solutions sont très riches
- Le référencement est parfois difficile
- ...

Les solutions RIA proposent généralement un environnement d'exécution, des bibliothèques et/ou des API, et des outils qui permettent d'être plus efficace et plus riche que le simple ajout d'Ajax dans une application de façon manuelle.

85.2. Les applications de type RDA

Les applications de type RDA reposent sur les technologies des applications de type web mais elles s'exécutent sur le bureau donc sans navigateur web. Elles permettent d'avoir les mêmes fonctionnalités qu'une application de type RIA mais exécutées en dehors du navigateur.

Elles nécessitent un environnement d'exécution installé sur le poste client, généralement sous la forme d'une machine virtuelle avec un ensemble d'API.

Elles offrent de meilleures interactivités notamment avec le système sous-jacent (drag & drop, accès au système de fichiers, ...). Pour des raisons de sécurité, les applications de type RDA peuvent avoir un accès au système sous-jacent sous réserve d'être signées. Cela offre un meilleure interactivité avec le système pour, par exemple, permettre une utilisation en mode déconnecté de l'application.

De plus, ces applications peuvent généralement être téléchargées sur internet et se mettre à jour via le réseau.

85.3. Les contraintes

Le développement d'applications de type RIA doit tenir compte de certaines contraintes inhérentes à ce type d'applications.

Les développeurs doivent utiliser les solutions RIA dans la limite de ce qu'elles peuvent proposer : toutes les applications ne peuvent pas être de type RIA. Par exemple, les applications de type RIA ne sont généralement pas adaptées pour des applications manipulant de grandes quantités de données.

Les développeurs d'applications web traditionnelles doivent tenir compte du mode de mise en oeuvre des applications RIA : la conception doit tenir compte du fait que l'application ne fonctionne pas sur un mode de rafraichissement à chaque requête/réponse. Ainsi, une application RIA est responsable du rafraichissement de ses données.

Le développement d'une application de type RIA nécessite la mise en oeuvre d'une architecture, notamment, côté serveur pour permettre de fournir à l'application les données et les traitements métiers nécessaires. Généralement, les solutions RIA ne concernent que la partie présentation et ne proposent aucune fonctionnalité dédiée pour la partie backend.

Lors de l'évaluation d'une solution, il est nécessaire d'évaluer ses capacités d'intégration avec la partie backend pour permettre les échanges de données et l'invocation de traitements métiers.

Les applications nécessitent plus l'intervention de graphistes pour définir l'IHM de l'application.

85.4. Les solutions RIA

Le besoin grandissant du marché concernant les applications riches se reflète dans l'activité des grands acteurs du marché comme Adobe, Sun, Microsoft, Google, ...

Ainsi, de nombreuses solutions sont proposées pour permettre le développement et la mise en oeuvre des applications riches. La plupart de ces solutions sont récentes et sont encore en cours de développement. Ces solutions ne sont donc pas toutes fiables mais elles évoluent très rapidement pour permettre de répondre à la demande importante du marché.

Parmi ces solutions, en plus des solutions reposant sur Java, il y a notamment Adobe Flex/Air et Microsoft Silverlight.

85.4.1. Les solutions RIA reposant sur Java

Dans le monde Java, Sun/Oracle propose Java FX. La fondation Eclipse propose Eclipse RCP (Rich Client Plateform) pour le développement d'applications de type RDA. Wazaabi repose sur RCP et XUL.

De nombreux frameworks open source facilitent aussi le développement de nouvelles applications de type RIA notamment :

- GWT (Google Web Toolkit)
- ZK
- Echo
- Ice Faces, Rich Faces
- Wicket
- TIBCO General Interface
- ...

85.4.1.1. Java FX

Java FX est un ensemble de technologies proposé pour le développement d'applications de type RIA.

Java FX a été présenté pour la première fois au JavaOne 2007 et a été la technologie mise en avant lors du JavaOne 2008.

Elle est arrivée tardivement, notamment vis-à-vis de Flex, et est de plus restée une bonne année sans réel outils : un interpréteur était disponible mais aucun compilateur ni IDE.

Depuis 2008, Java FX s'est enrichi d'outils ; un SDK et des fonctionnalités multimédia avancées grâce à l'intégration de codecs audio et vidéo.

Le grand intérêt de Java FX est son intégration avec Java. Par défaut, il faut coder l'application en utilisant Java FX Script qui est un langage de scripting déclaratif.

L'avantage de Java FX est qu'il ne nécessite qu'une machine virtuelle Java (JVM) de la plate-forme SE ou ME pour s'exécuter : cela apporte à Java FX un avantage certain car une JVM est installée sur une très large majorité d'appareils de différents types : notamment sur les ordinateurs de bureaux et portablee, encore plus sur les appareils téléphoniques mobiles et dans tous les lecteurs de disques Blu-Ray.

Une caractéristique de Java FX est d'être intégralement open source, ce qui n'est pas entièrement le cas de ses principaux concurrents directs.

Plusieurs sites relatifs à Java FX peuvent être consultés :

- https://www.oracle.com/java/technologies/javase/javafx-overview.html
- openjfx.io/

Oracle annonce la version 2.0 de JavaFX à JavaOne 2011.

En novembre 2011, JavaFX est développé en open source grâce à un sous-projet d'OpenJDK nommé OpenJFX.

85.4.1.2. Google GWT

Google propose GWT (Google Web Toolkit) pour le développement d'applications de type RIA.

L'application est écrite en Java avec un sous-ensemble de l'API standard et une API dédiée proposée par Google. L'ensemble du code est compilé pour générer du code JavaScript optimisé pour chaque navigateur.

Hormis une page hôte et l'utilisation des feuilles de style CSS, le développeur n'a besoin d'aucune connaissance sur les technologies web car elles sont encapsulées dans l'API. Le code Java écrit avec GWT ressemble plus à celui utilisé pour des applications AWT qu'au code d'une application de type web.

Les composants graphiques proposés par GWT sont relativement basiques mais des bibliothèques tierces permettent de fournir des composants évolués notamment grâce à la facilité d'encapsuler du code JavaScript dans GWT.

Le site officiel est à l'url https://www.gwtproject.org/

85.4.1.3. ZK

Le framework ZK est un framework open source pour le développement d'applications de type RIA mettant en oeuvre Ajax.

Pour le développement de l'interface graphique, ZK propose XUML (ZK User Interface Markup Langage) qui permet une description de l'interface en XML grâce à des composants XUL et XHTML.

ZK propose une gestion des événements et une intégration avec d'autres frameworks Java

Plusieurs langages sont supportés pour coder les traitements dont le principal est Java.

Le site officiel est à l'url https://www.zkoss.org/

85.4.1.4. Echo

Echo est un framework open source pour le développement orienté objet avec gestion des événements d'applications web riches.

Le développement de la partie IHM ressemble au développement d'applications graphiques de type client lourd :

composants orientés objets, gestion des événements, ...

Selon la version du framework Echo utilisée, une application peut prendre deux formes :

- Entièrement orientée serveur (Echo 2 et 3)
- Avoir une partie en JavaScript côté client (Echo 3 uniquement).

85.4.1.5. Apache Wicket

Wicket est un projet de la fondation Apache qui propose un framework orienté composants pour le développement d'applications web riches.

Le framework propose une séparation entre la partie présentation en XHTML et la partie traitement écrite en Java via des composants.

La page est encapsulée dans un objet et représentée dans une page XHTML dans lequel on ajoute des composants graphiques. La liaison se fait par un id.

Le site officiel est à l'url https://wicket.apache.org/

85.4.1.6. Les composants JSF

Plusieurs composants JSF proposent une implémentation d'Ajax dans leurs composants notamment :

- Myfaces Tobago :
- Myfaces Trinidad :
- IceFaces :
- Jboss RichFaces :
- ...

85.4.1.7. Tibco General Interface

General Interface est un framework open source pour le développement d'applications web riches. General Interface est diffusée en open source sous la licence BSD et sous une forme commerciale avec un support.

General Interface propose un IDE qui facilite le développement de la partie graphique d'une application en proposant d'utiliser le cliquer/glisser des composants.

Les échanges entre le client et le serveur se font grâce à des services web : ceci permet de rendre le framework GI plus indépendant de la solution backend utilisée.

L'application peut être exécutée dans Internet Explorer et Firefox sous Windows, Linux et Mac.

Le site officiel est à l'url http://www.generalinterface.org/

85.4.1.8. Eclipse RAP

Eclipse RAP (Rich Ajax Platform) repose sur l'API Eclipse RCP et génère une application html utilisant Ajax.

85.4.2. Les autres solutions RIA

Plusieurs fournisseurs proposent des solutions pour le développement d'applications de type RIA. Généralement ces solutions se concentrent sur la partie IHM et s'interfacent plus ou moins facilement avec un backend écrit en Java notamment au travers de services web par exemple.

85.4.2.1. Adobe/Apache Flex

Adobe Flex est un outil de développement pour créer des applications compilées sous la forme de fichiers swf exécutés dans le Flash player.

Adobe Flex repose sur MXML qui permet de créer l'interface graphique de manière déclarative en MXML et ActionScript pour être compilés en une application Flash. Le navigateur doit avoir le plug-in Flash pour pouvoir exécuter l'application.

Le site officiel d'Adobe Flex est à l'url : labs.adobe.com/technologies/flex/

Flex se concentre sur la partie IHM et permet une intégration facilitée avec un backend développé en Java dont notamment une solution open source : Blaze DS.

En novembre 2011, Adobe donne la technolgoqie Flex à la foundation Apache pour permettre son développement en open source.

85.4.2.2. Microsoft Silverlight

Silverlight (initialement connu sous le nom WPF/e) est la solution proposée par Microsoft pour le développement d'applications de type RIA.

Microsoft Silverlight repose sur XAML qui permet de décrire l'interface graphique en XML. Le plug-in Silverlight est requis pour l'exécution d'une application.

La version 1.0 utilise le langage JavaScript.

La version 2.0 de Silverlight permet de développer des applications avec les langages de la plate-forme .Net. Le plug-in de Silverlight 2.0 incorpore une machine virtuelle de type CLR mais seul un sous-ensemble de l'API de la plate-forme .Net est utilisable.

Silverlight propose la technologie DeepZone qui permet de faire des zooms sur une image.

Le site officiel de Microsoft Silverlight est à l'url : www.microsoft.com/silverlight

Le site Web officiel du framework Microsoft .NET : msdn.microsoft.com/fr-fr/netframework

85.4.2.3. Google Gears

Google Gears est une API et un plug-in qui permettent d'utiliser une base de données SQLite pour stocker des données en local et ainsi permettre à des applications Ajax de fonctionner en mode déconnecté. Un exemple de mise en oeuvre de cette API est proposé par Google Reader.

En décembre 2009, Google abandonne le développement de Gears au profit d'HTML 5 qui doit contenir une API similaire.

85.4.3. Une comparaison entre GWT et Flex

Le but est de fournir les principaux avantages et inconvénients de GWT et Flex.

	Avantages	Inconvénients
GWT	Nécessite uniquement un navigateur (pas de plug-in) Pas de nouveau langage à apprendre pour un développeur Java	Peu de composants graphiques évolués qui nécessitent généralement l'utilisation d'une bibliothèque tierce Pas de structure standard (type MVC) pour les applications
Flex	Richesse et cohérence des composants graphiques Rendu identique sur les navigateurs supportés	Nouveaux langages (MXML et ActionScript) à apprendre pour les développeurs Java, en plus de Java nécessaires pour développer la partie serveur Nécessite le plug-in Flash (il est cependant très largement déployé) Les outils pour être productif sont payants : Adobe propose un plug-in Eclipse (Flex Builder)

GWT et Flex possèdent des avantages et inconvénients communs :

- Leur communauté est importante et très productive
- Pour des sites web, le référencement est délicat avec les deux solutions

85.5. Les solutions RDA

Les solutions pour développer des applications de type RDA existent déjà sous plusieurs formes :

- Java avec Java Web Start
- Java avec des socles applicatifs : Eclipse RCP ou Netbeans RCP
- Adobe AIR

85.5.1. Adobe AIR

Adobe AIR (Adobe Integrated Runtime) propose un environnement d'exécution pour les applications Flex et/ou Html/Javascript.

Le site officiel d'Adobe AIR est à l'url : www.adobe.com/fr/products/air.html

AIR est utilisable sur plusieurs plates-formes desktop (Windows et Mac OS), mobiles (iOS, Android, BlackBerry PlayBook) et certaines télévisions.

Adobe AIR doit être installé sur le poste de l'utilisateur pour pouvoir lui permettre d'exécuter l'application AIR.

L'environnement d'exécution AIR est composé d'un navigateur sur une base WebKit

85.5.2. Eclipse RCP

Eclipse RCP (Rich Client Platform) est la base sur laquelle Eclipse repose. Ce socle utilise Java et SWT.

La suite de cette section sera développée dans une version future de ce document

85.5.3. Netbeans RCP

NetBeans RCP est la base sur laquelle NetBeans repose. Ce socle utilise Java et Swing.

La suite de cette section sera développée dans une version future de ce document

86. Les applets

Chapitre 86

Niveau : 🟩🟩🟨⬜⬜⬜ Intermédiaire

Technologie legacy
Ce chapitre est conservé pour des raisons historiques

Une applet est un programme Java qui s'exécute dans un logiciel de navigation supportant Java ou dans l'appletviewer du JDK.

Il est recommandé de tester les applets avec l'appletviewer car les navigateurs peuvent prendre l'applet contenu dans leurs caches plutôt que la dernière version compilée.

Les applets ont historiquement permis de réaliser des animations dans les pages web avant de voir arriver d'autres solutions comme Flash.

Attention : pour améliorer leur sécurité, les navigateurs modernes prévoient à terme de retirer le support des plugins de type NPAPI. Ceci conduit à l'arrêt du support de technologies reposant sur des plugins pour exécuter du code comme Flash, Silverlight et bien évidemment Java. Fin 2015, les principaux navigateurs ont annoncé l'arrêt ou une date de l'arrêt du support par les navigateurs. Les applications reposant sur des applets doivent envisager de migrer vers d'autres technologies supportées par les navigateurs.

Ainsi à partir de Java 9, l'API Applet est deprecated. Son éviction interviendra dans une version ultérieure de Java.

Ce chapitre contient plusieurs sections :

- L'intégration d'applets dans une page HTML
- Les méthodes des applets
- Les interfaces utiles pour les applets
- La transmission de paramètres à une applet
- Les applets et le multimédia
- Une applet pouvant s'exécuter comme une application
- Les droits des applets

86.1. L'intégration d'applets dans une page HTML

Dans une page HTML, il faut utiliser le tag APPLET avec la syntaxe suivante :

<APPLET CODE=« Exemple.class » WIDTH=200 HEIGHT=300 > </APPLET>

Le nom de l'applet est indiqué entre guillemets à la suite du paramètre CODE.

Les paramètres WIDTH et HEIGHT fixent la taille de la fenêtre de l'applet dans la page HTML. L'unité est le pixel. Il est préférable de ne pas dépasser 640 * 480 (VGA standard).

Le tag APPLET peut comporter les attributs facultatifs suivants :

Tag	Rôle
CODEBASE	permet de spécifier le chemin relatif au dossier de la page contenant l'applet. Ce paramètre suit le paramètre CODE. Exemple : CODE=nomApplet.class CODEBASE=/nomDossier
HSPACE et VSPACE	permettent de fixer la distance en pixels entre l'applet et le texte
ALT	affiche le texte spécifié par le paramètre lorsque le navigateur ne supporte pas Java ou que son support est désactivé.

Le tag PARAM permet de passer des paramètres à l'applet. Il doit être inclus entre les tags APPLET et /APPLET.

<PARAM nomParametre value=« valeurParametre »> </APPLET>

La valeur est toujours passée sous forme de chaîne de caractères donc entourée de guillemets.

Exemple : <APPLET code=« Exemple.class » width=200 height=300>

Le texte contenu entre <APPLET> et </APPLET> est affiché si le navigateur ne supporte pas java.

86.2. Les méthodes des applets

Le mécanisme d'initialisation d'une applet se fait en deux temps :

1. la machine virtuelle Java instancie l'objet Applet en utilisant le constructeur par défaut
2. la machine virtuelle Java envoie le message init() à l'objet Applet

Une classe dérivée de la classe java.applet.Applet hérite de méthodes qu'il faut redéfinir en fonction des besoins et doit être déclarée public pour fonctionner.

En général, il n'est pas nécessaire de faire un appel explicite aux méthodes init(), start(), stop() et destroy() : le navigateur se charge d'appeler ces méthodes en fonction de l'état de la page HTML contenant l'applet.

86.2.1. La méthode init()

Cette méthode permet l'initialisation de l'applet : elle n'est exécutée qu'une seule et unique fois après le chargement de l'applet.

86.2.2. La méthode start()

Cette méthode est appelée automatiquement après le chargement et l'initialisation (par la méthode init()) lors du premier affichage de l'applet.

86.2.3. La méthode stop()

Le navigateur appelle automatiquement la méthode lorsque l'on quitte la page HTML. Elle interrompt les traitements de tous les processus en cours.

86.2.4. La méthode destroy()

Elle est appelée après l'arrêt de l'applet ou lors de l'arrêt de la machine virtuelle. Elle libère les ressources et détruit les threads restants

86.2.5. La méthode update()

Elle est appelée à chaque rafraîchissement de l'écran ou appel de la méthode repaint(). Elle efface l'écran et appelle la méthode paint(). Ces actions provoquent souvent des scintillements. Il est préférable de redéfinir cette méthode pour qu'elle n'efface plus l'écran :

Exemple :

```
public void update(Graphics g) { paint (g);}
```

86.2.6. La méthode paint()

Cette méthode permet d'afficher le contenu de l'applet à l'écran. Ce rafraîchissement peut être provoqué par le navigateur ou par le système d'exploitation si l'ordre des fenêtres ou leur taille a été modifié ou si une fenêtre recouvre l'applet.

Exemple :

```
public void paint(Graphics g)
```

La méthode repaint() force l'utilisation de la méthode paint().

Il existe des méthodes dédiées à la gestion de la couleur de fond et de la couleur de premier plan

La méthode setBackground(Color), héritée de Component, permet de définir la couleur de fond d'une applet. Elle attend en paramètre un objet de la classe Color.

La méthode setForeground(Color) fixe la couleur d'affichage par défaut. Elle s'applique au texte et aux graphiques.

Les couleurs peuvent être spécifiées de trois manières différentes :

utiliser les noms standards prédéfinis	Color.nomDeLaCouleur Les noms prédéfinis de la classe Color sont : black, blue, cyan, darkGray, gray, green, lightGray, magenta, orange, pink, red, white, yellow

utiliser 3 nombres de type entier représentant le RGB	(Red,Green,Blue : rouge,vert, bleu)
	Exemple :
	`Color macouleur = new Color(150,200,250);`
	`setBackground (macouleur);`
	`// ou setBackground(150,200,250);`

utiliser 3 nombres de type float utilisant le système HSB	(Hue, Saturation, Brightness : teinte, saturation, luminance). Ce système est moins répandu que le RGB mais il permet notamment de modifier la luminance sans modifier les autres caractéristiques Exemple : `setBackground(0.0,0.5,1.0);` dans ce cas 0.0,0.0,0.0 représente le noir et 1.0,1.0,1.0 représente le blanc.

86.2.7. Les méthodes size() et getSize()

L'origine des coordonnées en Java est le coin supérieur gauche. Elles s'expriment en pixels avec le type int.

La détermination des dimensions d'une applet se fait de la façon suivante :

Exemple (code Java 1.0) :

```
Dimension dim = size();
int applargeur = dim.width;
int apphauteur = dim.height;
```

Avec le JDK 1.1, il faut utiliser getSize() à la place de size().

Exemple (code Java 1.1) :

```
public void paint(Graphics g) {
   super.paint(g);
   Dimension dim = getSize();
   int applargeur = dim.width;
   int apphauteur = dim.height;
   g.drawString("width = "+applargeur,10,15);
   g.drawString("height = "+apphauteur,10,30);
}
```

86.2.8. Les méthodes getCodeBase() et getDocumentBase()

Ces méthodes renvoient respectivement l'emplacement de l'applet sous forme d'adresse Web ou de dossier et l'emplacement de la page HTML qui contient l'applet.

Exemple :

```
public void paint(Graphics g) {
   super.paint(g);
   g.drawString("CodeBase = "+getCodeBase(),10,15);
   g.drawString("DocumentBase = "+getDocumentBase(),10,30);
}
```

86.2.9. La méthode showStatus()

Affiche un message dans la barre de statut de l'applet

Exemple :

```
public void paint(Graphics g) {
   super.paint(g);
   showStatus("message à afficher dans la barre d'état");
}
```

86.2.10. La méthode getAppletInfo()

Permet de fournir des informations concernant l'auteur, la version et le copyright de l'applet

Exemple :

```
static final String appletInfo = " test applet : auteur, 1999 \n\nCommentaires";

public String getAppletInfo() {
   return appletInfo;
}
```

Pour voir les informations, il faut utiliser l'option info du menu Applet de l'appletviewer.

86.2.11. La méthode getParameterInfo()

Cette méthode permet de fournir des informations sur les paramètres reconnus par l'applet

Le format du tableau est le suivant :

{ {nom du paramètre, valeurs possibles, description} , ... }

Exemple :

```
static final String[][] parameterInfo =
{ {"texte1", "texte1", " commentaires du texte 1" } ,
  {"texte2", "texte2", " commentaires du texte 2" } };

public String[][] getParameterInfo() {
   return parameterInfo;
}
```

Pour voir les informations, il faut utiliser l'option info du menu Applet de l'appletviewer.

86.2.12. La méthode getGraphics()

Elle retourne la zone graphique d'une applet : utile pour dessiner dans l'applet avec des méthodes qui ne reçoivent pas le contexte graphique en paramètres (ex : mouseDown() ou mouseDrag()).

86.2.13. La méthode getAppletContext()

Cette méthode permet l'accès à des fonctionnalités du navigateur.

86.2.14. La méthode setStub()

Cette méthode permet d'attacher l'applet au navigateur.

86.3. Les interfaces utiles pour les applets

86.3.1. L'interface Runnable

Cette interface fournit le comportement nécessaire à une applet pour devenir un thread.

Les méthodes start() et stop() de l'applet peuvent démarrer et arrêter un thread pour limiter l'usage des ressources machines lorsque la page contenant l'applet est inactive.

86.3.2. L'interface ActionListener

Cette interface permet à l'applet de répondre aux actions de l'utilisateur avec la souris

La méthode actionPerformed() définit les traitements associés aux événements.

Exemple (code Java 1.1) :

```
public void actionPerformed(ActionEvent evt) { ... }
```

Pour plus d'information, voir le chapitre «L'interception des actions de l'utilisateur».

86.3.3. L'interface MouseListener pour répondre à un clic de souris

Exemple (code Java 1.1) :

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class AppletMouse extends Applet implements MouseListener {
   int nbClick = 0;

   public void init() {
      super.init();
      addMouseListener(this);
   }

   public void mouseClicked(MouseEvent e) {
      nbClick++;
      repaint();
   }

   public void mouseEntered(MouseEvent e) {
   }

   public void mouseExited(MouseEvent e) {
   }

   public void mousePressed(MouseEvent e) {
   }

   public void mouseReleased(MouseEvent e) {
   }

   public void paint(Graphics g) {
      super.paint(g);
      g.drawString("Nombre de clics : " + nbClick, 10, 10);
   }

}
```

Pour plus d'information, voir le chapitre sur «L'interception des actions de l'utilisateur».

86.4. La transmission de paramètres à une applet

La méthode getParameter() retourne les paramètres écrits dans la page HTML. Elle retourne une chaîne de caractères de type String.

Exemple :

```
Sting parametre;
parametre = getParameter(" nom-parametre ");
```

Si le paramètre n'est pas renseigné dans la page HTML alors getParameter() retourne null

Pour utiliser les valeurs des paramètres, il sera souvent nécessaire de faire une conversion de la chaîne de caractères dans le type voulu en utilisant les Wrappers

Exemple :

```
String taille;
int hauteur;
taille = getParameter(" hauteur ");
Integer temp = new Integer(taille)
hauteur = temp.intValue();
```

Exemple :

```
int vitesse;
String paramvitesse = getParameter(" VITESSE ");
if (paramvitesse != null) vitesse = Integer.parseInt(paramVitesse);
// parseInt ne fonctionne pas avec une chaîne vide
```

Attention : l'appel à la méthode getParameter() dans le constructeur pas défaut lève une exception de type NullPointerException.

Exemple :

```
public MonApplet() {
   String taille;
   taille = getParameter(" message ");
}
```

86.5. Les applets et le multimédia

86.5.1. L'insertion d'images

Java supporte deux standards :

- le format GIF de Compuserve qui est beaucoup utilisé sur internet car il génère des fichiers de petites tailles contenant des images d'au plus 256 couleurs.
- et le format JPEG. qui convient mieux aux grandes images et à celles de plus de 256 couleurs car le taux de compression avec perte de qualité peut être précisé.

Pour la manipulation des images, le package nécessaire est java.awt.image.

La méthode getImage() possède deux signatures : getImage(URL url) et getImage (URL url, String name).

On procède en deux étapes : le chargement puis l'affichage. Si les paramètres fournis à getImage ne désignent pas une image, aucune exception n'est levée.

La méthode getImage() ne charge pas de données sur le poste client. Celles-ci seront chargées quand l'image sera dessinée pour la première fois.

Exemple :

```
public void paint(Graphics g) {
   super.paint(g);
   Image image=null;
   image=getImage(getDocumentBase( ), "monimage.gif"); //chargement de l'image
   g.drawImage(image, 40, 70, this);
}
```

Le sixième paramètre de la méthode drawImage() est un objet qui implémente l'interface ImageObserver. ImageObserver est une interface déclarée dans le package java.awt.image qui sert à donner des informations sur le fichier image. Souvent, on indique représentant l'applet elle-même this à la place de cet argument. La classe ImageObserver détecte le chargement et la fin de l'affichage d'une image. La classe Applet ce charge automatiquement de faire ces actions d'où le fait de mettre this.

Pour obtenir les dimensions de l'image à afficher on peut utiliser les méthodes getWidth() et getHeight() qui retourne un nombre entier en pixels.

Exemple :

```
int largeur = 0;
int hauteur = 0;
largeur = image.getWidth(this);
hauteur = image.getHeight(this);
```

86.5.2. L'utilisation des capacités audio

Seul le format d'extension .AU de Sun est supporté par Java. Pour utiliser un autre format, il faut le convertir.

La méthode play() permet de jouer un son.

Exemple :

```
import java.net.URL;

   ...
   try {
      play(new URL(getDocumentBase(), " monson.au "));
   } catch (java.net.MalformedURLException e) {}
```

La méthode getDocumentBase() détermine et renvoie l'URL de l'applet.

Ce mode d'exécution n'est valable que si le son n'est à reproduire qu'une seule fois, sinon il faut utiliser l'interface AudioClip.

Avec trois méthodes, l'interface AudioClip facilite l'utilisation des sons :

- public abstract void play() // jouer une seule fois le fichier
- public abstract void loop() // relancer le son jusqu'à interruption par la méthode stop ou la fin de l'applet
- public abstract void stop() // fin de la reproduction du clip audio

Exemple :

```
import java.applet.*;
import java.awt.*;
import java.net.*;
```

```
public class AppletMusic extends Applet {
   protected AudioClip aC = null;

   public void init() {
      super.init();
      try {
      AppletContext ac = getAppletContext();
      if (ac != null)
         aC = ac.getAudioClip(new URL(getDocumentBase(), "spacemusic.au"));
      else
         System.out.println(" fichier son introuvable ");
     } catch (MalformedURLException e) {}
      aC.loop();
   }
}
```

Pour utiliser plusieurs sons dans une applet, il suffit de déclarer plusieurs variables AudioClip.

L'objet retourné par la méthode getAudioClip() est un objet qui implémente l'interface AudioClip définit dans la machine virtuelle car il est très dépendant du système de la plate-forme d'exécution.

86.5.3. L'animation d'un logo

Exemple :

```
import java.applet.*;
import java.awt.*;

public class AppletAnimation extends Applet implements Runnable {
   Thread thread;
   protected Image tabImage[];
   protected int index;

   public void init() {
      super.init();
      // chargement du tableau d'images
      index = 0;
      tabImage = new Image[2];
      for (int i = 0; i < tabImage.length; i++) {
         String fichier = new String("monimage" + (i + 1) + ".gif ");
         tabImage[i] = getImage(getDocumentBase(), fichier);
      }
   }

   public void paint(Graphics g) {
      super.paint(g);
      // affichage de l'image
      g.drawImage(tabImage[index], 10, 10, this);
   }

   public void run() {
      // traitements exécutés par le thread
      while (true) {
         repaint();
         index++;
         if (index >= tabImage.length)
            index = 0;
         try {
            thread.sleep(500);
      } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }

   public void start() {
      // démarrage du thread
      if (thread == null) {
         thread = new Thread(this);
         thread.start();
      }
   }

   public void stop() {
      // arrêt du thread
      if (thread != null) {
         thread.stop();
         thread = null;
      }
   }

   public void update(Graphics g) {
      // la redéfinition de la méthode permet d'éviter les scintillements
      paint(g);
   }

}
```

La surcharge de la méthode paint() permet d'éviter le scintillement dû à l'effacement de l'écran et à son rafraîchissement. Dans ce cas, seul le rafraîchissement est effectué.

86.6. Une applet pouvant s'exécuter comme une application

Il faut rajouter une classe main à l'applet, définir sa fenêtre d'affichage, appeler les méthodes init() et start() et afficher la fenêtre.

Exemple (code Java 1.1) :

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class AppletApplication extends Applet implements WindowListener {

   public static void main(java.lang.String[] args) {
      AppletApplication applet = new AppletApplication();
      Frame frame = new Frame("Applet");
      frame.addWindowListener(applet);
      frame.add("Center", applet);
      frame.setSize(350, 250);
      frame.show();
      applet.init();
      applet.start();
   }

   public void paint(Graphics g) {
      super.paint(g);
      g.drawString("Bonjour", 10, 10);
   }

   public void windowActivated(WindowEvent e) { }

   public void windowClosed(WindowEvent e) { }

   public void windowClosing(WindowEvent e) {
      System.exit(0);
   }

   public void windowDeactivated(WindowEvent e) { }

   public void windowDeiconified(WindowEvent e) { }

   public void windowIconified(WindowEvent e) { }

   public void windowOpened(WindowEvent e) { }
}
```

86.7. Les droits des applets

Une applet est une application Java hébergée sur une machine distante (un serveur Web) et qui s'exécute, après chargement, sur la machine cliente équipée d'un navigateur. Ce navigateur contrôle les accès de l'applet aux ressources locales et ne les autorisent pas systématiquement : chaque navigateur définit sa propre règle.

Le modèle classique de sécurité pour l'exécution des applets, recommandé par Sun, distingue deux types d'applets : les applets non dignes de confiance (untrusted) qui n'ont pas accès aux ressources locales et externes, les applets dignes de confiance (trusted) qui ont l'accès. Dans ce modèle, une applet est par défaut untrusted.

La signature d'une applet permet de désigner son auteur et de garantir que le code chargé par le client est bien celui demandé au serveur. Cependant, une applet signée n'est pas forcément digne de confiance.

La suite de ce chapitre sera développée dans une version future de ce document

87. Java Web Start (JWS)

Chapitre 87

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Technologie legacy
Ce chapitre est conservé pour des raisons historiques

Développée avec la plate-forme Java 2, Java Web Start est une technologie qui permet le déploiement d'applications autonomes à travers le réseau. Elle permet l'installation d'une application grâce à un simple clic dans un navigateur. JWS a été inclus dans le J2RE 1.4. Pour les versions antérieures du J2RE, il est nécessaire de télécharger JWS et de l'installer sur le poste client.

JWS est le résultat des travaux de la JSR-56. La documentation de cette technologie est à l'url : https://docs.oracle.com/javase/8/docs/technotes/guides/javaws/.

JWS permet la mise à jour automatique de l'application si une nouvelle version est disponible sur le serveur et assure une mise en cache locale des applications pour accélérer leur réutilisation ultérieure.

La sécurité des applications exécutées est assurée par l'utilisation du bac à sable (sandbox) comme pour une applet, dès lors, pour certaines opérations il est nécessaire de signer l'application.

JWS utilise et implémente une API et un protocole nommé Java Network Lauching Protocol (JNLP).

Le grand avantage de Java Web Start est qu'il est inutile de modifier une application pour qu'elle puisse être déployée avec cette technologie (à condition que les fichiers contenant des ressources soient accédés en utilisant la méthode getResource() du classloader).

L'application doit être packagée dans un fichier jar qui sera associé sur le serveur à un fichier particulier de lancement.

L'utilisation d'une application grâce à JWS implique la réalisation de plusieurs étapes :

- Packager l'application dans un fichier jar en le signant si nécessaire
- Créer le fichier de lancement .jnlp
- Copier les deux fichiers sur le serveur web

Attention : à partir de Java 9, Java Web Start et le protocole JNLP sont deprecated. Oracle l'a retiré de son JDK à partir de Java 11.

Ce chapitre contient plusieurs sections :

- ♦ La création du package de l'application
- ♦ La signature d'un fichier jar
- ♦ Le fichier JNLP
- ♦ La configuration du serveur web
- ♦ Le fichier HTML
- ♦ Le test de l'application

- L'utilisation du gestionnaire d'applications
- L'API de Java Web Start

87.1. La création du package de l'application

L'application doit être packagée sous la forme d'un fichier .jar.

Il est possible de fournir une petite icône pour représenter l'application : celle-ci doit avoir une taille de 64 x 64 pixels au format Gif ou JPEG.

87.2. La signature d'un fichier jar

L'exemple de cette section crée un certificat et signe l'application avec ce dernier.

Exemple :

```
C:\java\>keytool -genkey -keystore mes_cles -alias cle_de_test
Tapez le mot de passe du Keystore :  test
Mot de passe de Keystore trop court, il doit compter au moins 6 caractères
Tapez le mot de passe du Keystore :  erreur keytool : java.lang.NullPointerExcep
tion
C:\java\>keytool -genkey -keystore mes_cles -alias cle_de_test
Tapez le mot de passe du Keystore :  mptest
Quels sont vos prénom et nom ?
  [Unknown] :  jean michel
Quel est le nom de votre unité organisationnelle ?
  [Unknown] :  test
Quelle est le nom de votre organisation ?
  [Unknown] :  test
Quel est le nom de votre ville de résidence ?
  [Unknown] :  Metz
Quel est le nom de votre état ou province ?
  [Unknown] :  France
Quel est le code de pays à deux lettres pour cette unité ?
  [Unknown] :  fr
Est-ce CN=jean michel, OU=test, O=test, L=Metz, ST=France, C=fr ?
  [non] :  oui
Spécifiez le mot de passe de la clé pour <cle_de_test>
        (appuyez sur Entrée s'il s'agit du mot de passe du Keystore) :
C:\java\>
C:\java\>keytool -selfcert -alias cle_de_test -keystore mes_cles
Tapez le mot de passe du Keystore :  mptest
C:\java\>keytool -list -keystore mes_cles
Tapez le mot de passe du Keystore :  mptest
Type Keystore : jks
Fournisseur Keystore : SUN
Votre Keystore contient 1 entrÚe(s)
cle_de_test, 12 nov. 2003, keyEntry,
Empreinte du certificat (MD5) : 9E:5A:61:CC:D8:88:02:59:1D:3B:41:C9:CA:26:1D:BD

C:\java>jarsigner -keystore mes_cles -signedjar MonJarSigne.jar MonApp.jar cle_d
e_test
Enter Passphrase for keystore:

Warning:
The signer certificate will expire within six months.

C:\java>dir
 Le volume dans le lecteur C s'appelle SW_Preload
 Le numéro de série du volume est 043F-2ED6

 Répertoire de C:\java

30/08/2009  11:44    <REP>          .
30/08/2009  11:44    <REP>          ..
30/08/2009  11:38             1 259 mes_cles
```

```
15/08/2009  23:58            15 793 MonApp.jar
30/08/2009  11:44            17 247 MonJarSigne.jar
               3 fichier(s)           34 299 octets
               0 Rép(s)  70 055 645 184 octets libres
```

87.3. Le fichier JNLP

Ce fichier au format XML permet de décrire l'application.

La racine de ce document XML est composée du tag <jnlp>. Son attribut codebase permet de préciser l'URL où sont stockés les fichiers indiqués par l'attribut href.

Le tag <information> permet de fournir des précisions qui seront utilisées par le gestionnaire d'applications sur le poste client. Ce tag possède plusieurs noeuds enfants :

Nom du tag	Rôle
title	Le nom de l'application
vendor	Nom de l'auteur de l'application
homepage	Préciser une page HTML qui contient des informations sur l'application grâce à son attribut href
description	Une description de l'application. Il est possible de préciser plusieurs types de descriptions grâce à l'attribut kind. Les valeurs possibles sont : one-line, short et tooltip. Pour utiliser plusieurs descriptions, il faut utiliser plusieurs tags Description avec l'attribut kind adéquat
offline-allowed	Ce tag précise que l'application peut être exécutée dans un mode déconnecté. L'avantage de ne pas préciser ce tag est de s'assurer que la dernière version de l'application est toujours utilisée mais cela nécessite obligatoirement une connexion pour toute exécution.
icon	Permet de préciser une URL vers une image de 64 x 64 pixels au format gif ou JPEG grâce à l'attribut href

Le tag <security> permet de préciser des informations concernant la sécurité.

Nom du tag	Rôle
all-permissions	Indique que l'application a besoin de tous les droits pour s'exécuter. L'application doit alors être obligatoirement signée. Si ce tag n'est pas précisé alors l'application s'exécute dans le bac à sable : elle possède donc les mêmes restrictions qu'une applet au niveau de la sécurité

Le tag <resources> permet de préciser des informations sur les ressources utilisées par l'application. L'attribut os permet de préciser des paramètres pour un système d'exploitation particulier.

Nom du tag	Rôle
j2se	Précise les JRE qui peuvent être utilisés par l'application. Les valeurs utilisables par l'attribut version sont 1.2, 1.3 et 1.4. Il est possible de préciser un numéro de version particulier ou d'utiliser le caractère * pour préciser n'importe quel numéro de release. L'ordre des différentes valeurs fournies est important.
jar	Précise un fichier .jar qui est utilisé par l'application
nativelib	Précise une bibliothèque utilisée par l'application qui contient du code natif
property	Précise une propriété système qui sera utilisable par l'application. L'attribut name précise le nom de la propriété et l'attribut value précise sa valeur

Le tag <application-desc> précise, grâce à son attribut main-class, la classe qui contient la méthode main().

Nom du tag	Rôle
argument	Préciser des arguments à l'application tels qu'ils pourraient être fournis sur une ligne de commandes

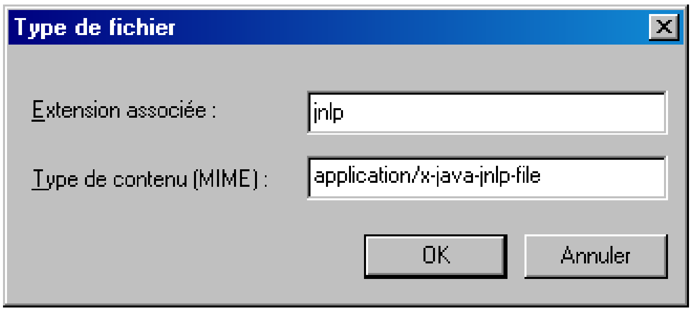
Exemple :

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.0+" codebase="http://localhost/" href="MonApplication.jnlp">
  <information>
    <title>Mon Application</title>
    <vendor>Jean Michel</vendor>
    <homepage href="http://localhost/"/>
    <description>Mon application</description>
    <description kind="short">une application de test</description>
    <offline-allowed/>
  </information>
  <security>
  </security>
  <resources>
    <j2se version="1.4"/>
    <jar href="MonApplication.jar"/>
  </resources>
  <application-desc main-class="fr.jmdoudoux.dej.jnlp.MonApplication" />
</jnlp>
```

87.4. La configuration du serveur web

Le serveur qui va fournir les fichiers doit être configuré pour qu'il associe le type MIME « application/x-java-jnlp-file » avec l'extension .jnlp

Par exemple sous IIS 5, il faut utiliser l'option propriété du menu contextuel du site. Dans l'onglet « En-Tête http », cliquez sur le bouton « Types de fichiers ». Dans la boîte de dialogue « Type de fichiers », cliquez sur le bouton « Nouveau type » si l'association n'est pas présente dans la liste. Une boîte de dialogue permet de saisir l'extension et le type MIME



Extension associée : jnlp

Type de contenu (MIME) : application/x-java-jnlp-file

Le type MIME permet au navigateur de connaître l'application qui devra être utilisée lors de la réception des données du serveur web.

87.5. Le fichier HTML

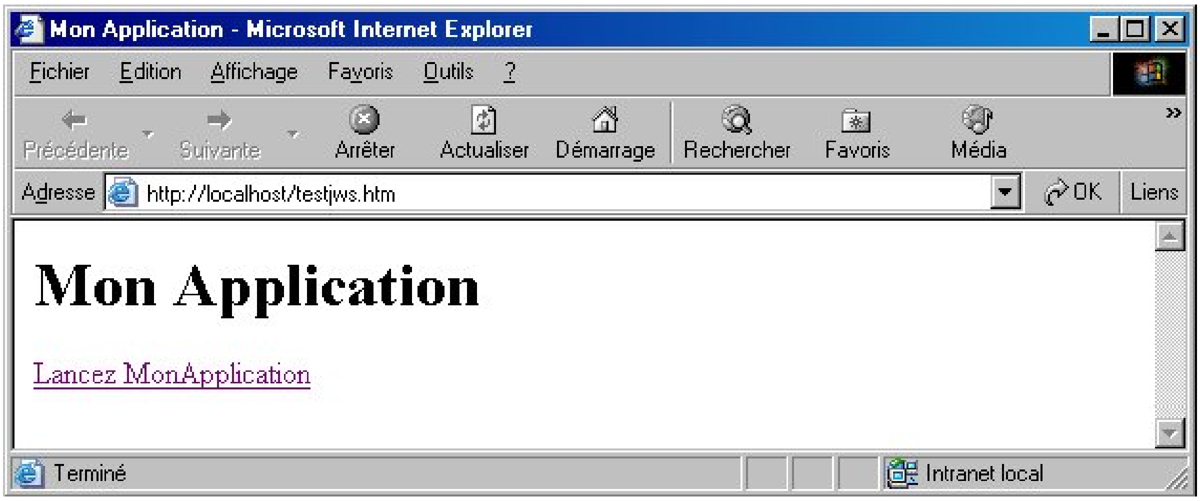
Hormis le code mininum requis par la norme HTML, la seule chose indispensable est un lien dont l'URL pointe vers le fichier .jnlp sur le serveur web.

Exemple :

```
<html>
  <head>
    <title>Mon Application</title>
  </head>
  <body>
    <H1>Mon Application</H1>
    <a href="http://localhost/Monapplication.jnlp">Lancez MonApplication</a>
  </body>
</html>
```

87.6. Le test de l'application

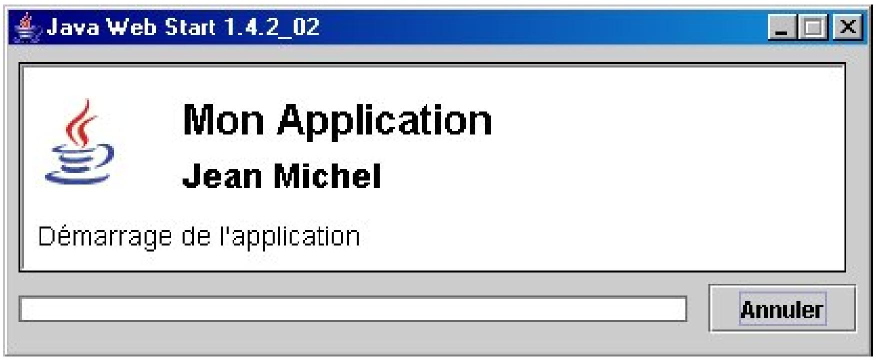
Il faut ouvrir un navigateur et saisir l'URL de la page contenant le lien vers le fichier jnlp


Mon Application - Microsoft Internet Explorer
Fichier Edition Affichage Favoris Outils ?
Précédente Suivante Arrêter Actualiser Démarrage Rechercher Favoris Média
Adresse http://localhost/testjws.htm
OK Liens
Terminé
Intranet local

Mon Application

Lancez MonApplication

Java Web Start se lance

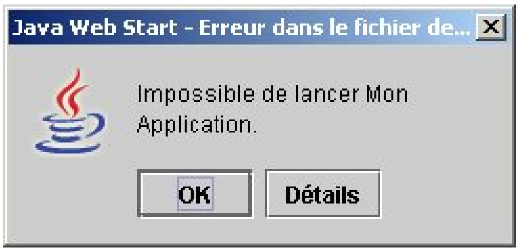


Mon Application

Jean Michel

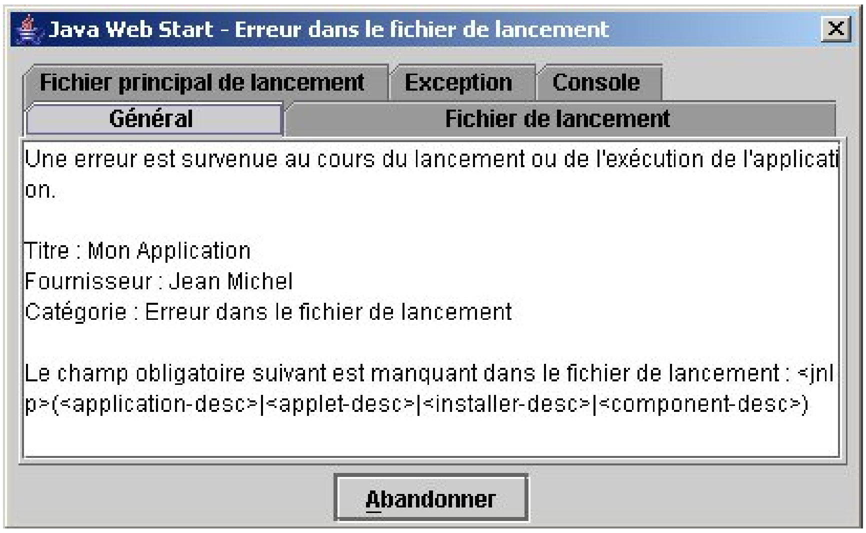
Démarrage de l'application

Si le fichier jnlp contient une erreur alors un message d'erreur est affiché.



Impossible de lancer Mon Application.

Cliquez sur « Détails » pour obtenir des informations sur l'erreur.

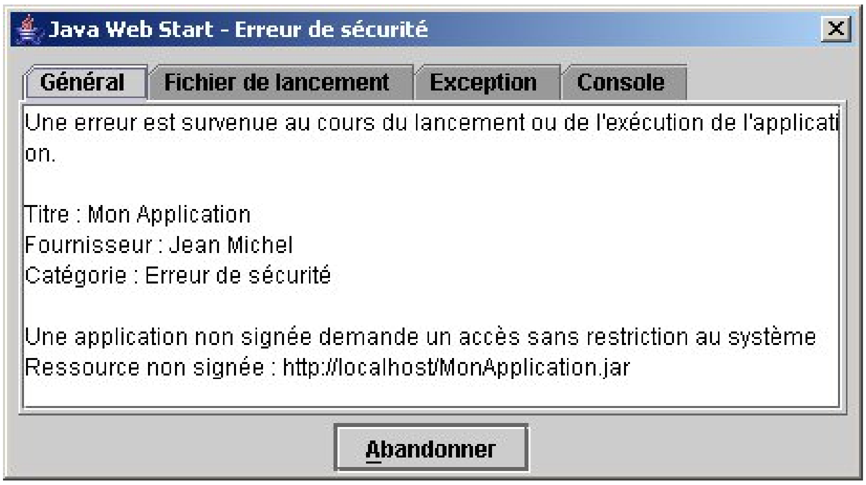

Java Web Start - Erreur dans le fichier de lancement
Fichier principal de lancement
Exception
Console
Général
Fichier de lancement
Abandonner

Une erreur est survenue au cours du lancement ou de l'exécution de l'application.

Titre : Mon Application
Fournisseur : Jean Michel
Catégorie : Erreur dans le fichier de lancement

Le champ obligatoire suivant est manquant dans le fichier de lancement : <jnlp>(<application-desc>|<applet-desc>|<installer-desc>|<component-desc>)

Si l'application nécessite un accès au système et que le fichier jar n'est pas signé alors un message d'erreur est affiché :

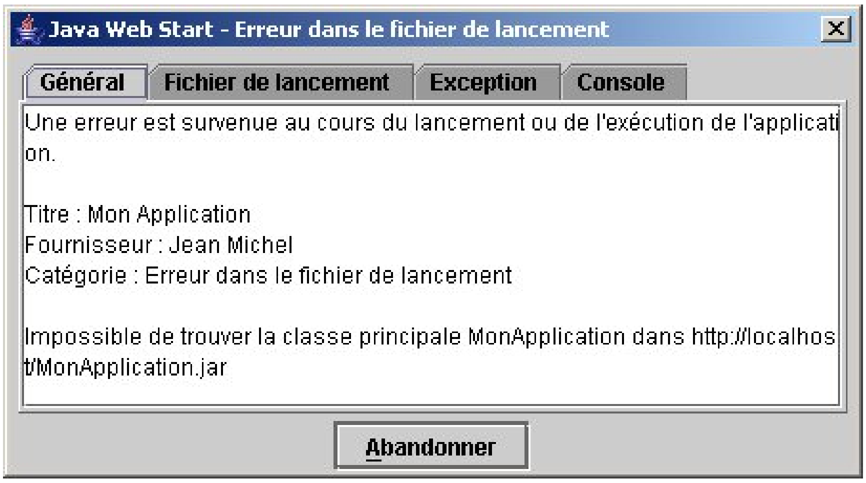


Une erreur est survenue au cours du lancement ou de l'exécution de l'application.

Titre : Mon Application
Fournisseur : Jean Michel
Catégorie : Erreur de sécurité

Une application non signée demande un accès sans restriction au système
Ressource non signée : http://localhost/MonApplication.jar

Si la classe précisée n'est pas trouvée dans le fichier jar indiqué alors un message d'erreur est affiché



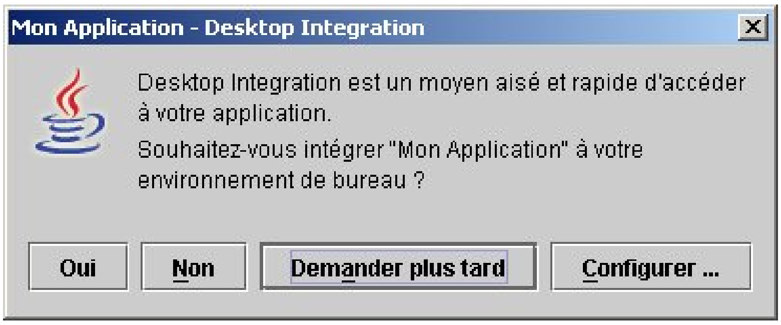
Une erreur est survenue au cours du lancement ou de l'exécution de l'application.

Titre : Mon Application
Fournisseur : Jean Michel
Catégorie : Erreur dans le fichier de lancement

Impossible de trouver la classe principale MonApplication dans http://localhost/MonApplication.jar

Dans cet exemple, pour résoudre le problème il faut indiquer le nom pleinement qualifié de la classe.

Au premier démarrage réussi d'une application, JWS demande si l'on souhaite créer un raccourci sur le bureau.



Mon Application - Desktop Integration

Desktop Integration est un moyen aisé et rapide d'accéder à votre application.

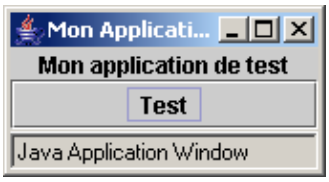
Souhaitez-vous intégrer "Mon Application" à votre environnement de bureau ?

En cliquant sur le bouton «Oui», JWS crée ce raccourci sur le bureau.

Exemple de raccourci :

```
"C:\Program Files\Java\j2re1.4.2_02\javaws\javaws.exe"
"@C:\Documents and Settings\administrateur\Application Data\
Sun\Java\Deployment\javaws\cache\indirect\indirect31560.ind"
```

L'application se lance



Mon Applicati...

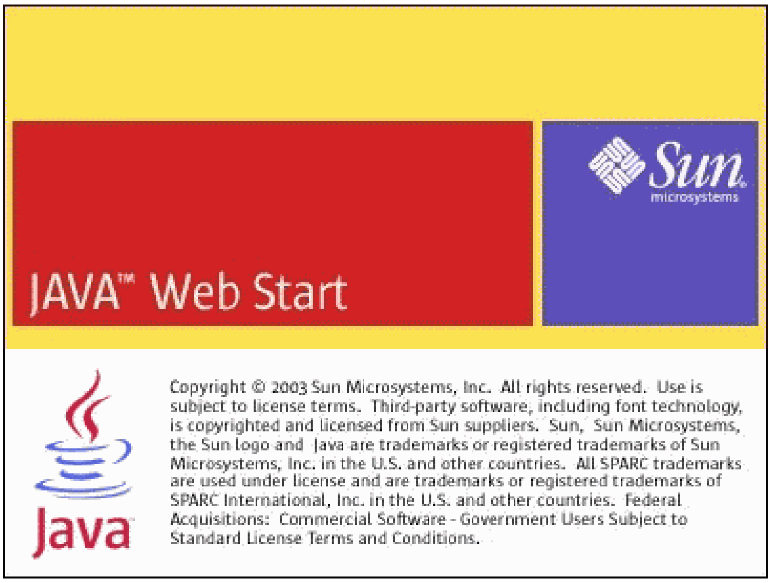
Mon application de test

Java Application Window

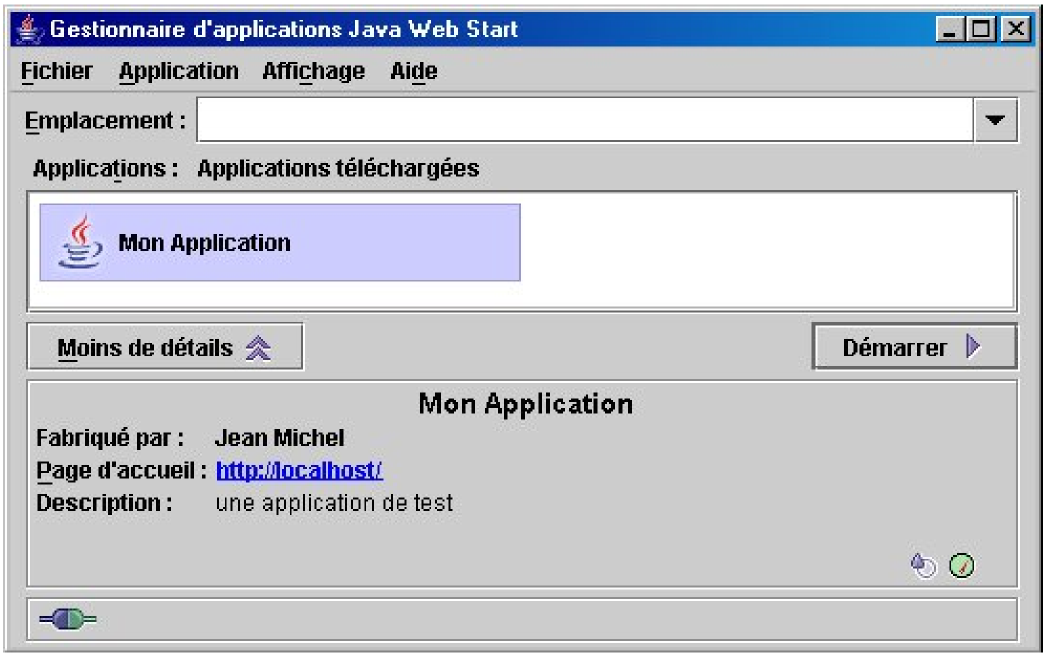
Comme pour les applets, par mesure de sécurité, un petit libellé en bas des fenêtres est affiché indiquant que la fenêtre est issue de l'exécution d'une application Java.

87.7. L'utilisation du gestionnaire d'applications

Pour lancer le gestionnaire d'applications, il suffit de double cliquer sur l'icône de « Java Web Start » sur le bureau.



Copyright © 2003 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms. Third-party software, including font technology, is copyrighted and licensed from Sun suppliers. Sun, Sun Microsystems, the Sun logo and Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Federal Acquisitions: Commercial Software - Government Users Subject to Standard License Terms and Conditions.



Emplacement :

Applications : Applications téléchargées

Mon Application

Mon Application

Fabriqué par : Jean Michel

Page d'accueil : http://localhost/

Description : une application de test

Le gestionnaire d'applications permet de gérer les applications en local : il permet de lancer les applications déjà téléchargées sur le poste et de les mettre à jour.

Plusieurs petites icônes peuvent apparaître selon le contexte

: une mise à jour de l'application est téléchargeable sur le serveur

: l'application peut être exécutée sans connexion au réseau

: l'application est mise en cache en local

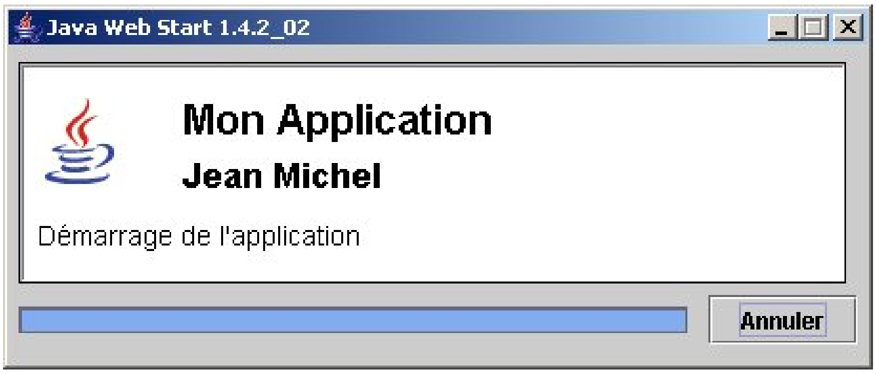
: l'application n'est pas signée

87.7.1. Le lancement d'une application

Pour lancer l'application, il suffit de sélectionner l'application concernée et de cliquer sur le bouton « Démarrer ».

Mon Application

Jean Michel



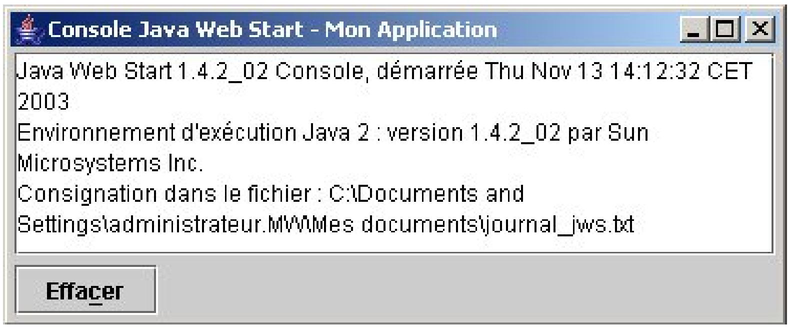
Mon Application

Jean Michel

Démarrage de l'application

87.7.2. L'affichage de la console

Dans les préférences, sur l'onglet « Avancé », cocher la case « Afficher la console Java »



```
Java Web Start 1.4.2_02 Console, démarrée Thu Nov 13 14:12:32 CET
2003
Environnement d'exécution Java 2 : version 1.4.2_02 par Sun
Microsystems Inc.
Consignation dans le fichier : C:\Documents and
Settings\administrateur.MW\Mes documents\journal_jws.txt
```

87.7.3. Consigner les traces d'exécution dans un fichier de log

Il permet aussi de configurer JWS. Par exemple, en cas de problème, il est possible de demander de consigner une trace d'exécution dans un fichier journal. Celui-ci est particulièrement utile lors du débogage.

Il est possible d'enregistrer les actions dans un fichier de log. Pour cela, il faut cocher la case « Consigner les sorties » et cliquer sur le bouton « Choisir le nom du fichier journal » pour sélectionner ou saisir le nom du fichier.

Exemple :

```
Java Web Start 1.4.2_02 Console, démarrée Thu Nov 13 13:54:36 CET 2003
Environnement d'exécution Java 2 : version 1.4.2_02 par Sun Microsystems Inc.
Consignation dans le fichier : C:\Documents and Settings\admin\Mes documents\journal_jws.txt
Java Web Start 1.4.2_02 Console, démarrée Thu Nov 13 13:54:41 CET 2003
Environnement d'exécution Java 2 : version 1.4.2_02 par Sun Microsystems Inc.
Consignation dans le fichier : C:\Documents and Settings\admin\Mes documents\journal_jws.txt
Java Web Start 1.4.2_02 Console, démarrée Thu Nov 13 13:55:14 CET 2003
Environnement d'exécution Java 2 : version 1.4.2_02 par Sun Microsystems Inc.
Consignation dans le fichier : C:\Documents and Settings\admin\Mes documents\journal_jws.txt
java.lang.NullPointerException
            at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
            at java.lang.ClassLoader.loadClass(Unknown Source)
            at java.lang.ClassLoader.loadClass(Unknown Source)
            at com.sun.javaws.Launcher.continueLaunch(Unknown Source)
            at com.sun.javaws.Launcher.handleApplicationDesc(Unknown Source)
            at com.sun.javaws.Launcher.handleLaunchFile(Unknown Source)
            at com.sun.javaws.Launcher.run(Unknown Source)
            at java.lang.Thread.run(Unknown Source)
```

87.8. L'API de Java Web Start

La suite de ce chapitre sera développée dans une version future de ce document

88. AJAX

Chapitre 88

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Le terme AJAX est l'acronyme de "Asynchronous JavaScript and XML", utilisé pour la première fois par Jesse James Garrett dans son article "AJAX: A New Approach to Web Applications". Ce terme s'est depuis popularisé.

Cependant AJAX est un acronyme qui reflète relativement mal ce qu'est AJAX en réalité : AJAX est un concept qui n'est pas lié particulièrement à un langage de développement et à un format d'échange de données. Pour faciliter la portabilité, la mise en oeuvre courante d'AJAX fait appel aux technologies Javascript et XML.

AJAX n'est pas une technologie mais plutôt une architecture technique qui permet d'interroger un serveur de manière asynchrone pour obtenir des informations permettant de mettre à jour dynamiquement la page HTML en manipulant son arbre DOM via DHTML.

AJAX est donc un modèle technique dont la mise en oeuvre intègre généralement plusieurs technologies :

- Une page web faisant usage d'AJAX (HTML/CSS/JavaScript)
- Une communication asynchrone avec un serveur pour obtenir des données
- Une manipulation de l'arbre DOM de la page pour permettre sa mise à jour (DHTML)
- Une utilisation d'un langage de script (JavaScript généralement) pour réaliser les différentes actions côté client

Historiquement, les développeurs d'applications web concentrent leurs efforts sur la partie métier et la persistance des données. La partie IHM est souvent délaissée essentiellement en invoquant les limitations de la technologie HTML. La maturité des technologies misent en oeuvre par le DHTML permettent maintenant de développer des applications plus riches et plus dynamiques.

Le but principal d'AJAX est d'éviter le rechargement complet d'une page pour n'en mettre qu'une partie à jour. Ceci permet donc d'améliorer l'interactivité et le dynamisme de l'application web qui le met en oeuvre.

Le fait de pouvoir opérer des actions asynchrones côté serveur et des mises à jour partielles d'une page web permet d'offrir de nombreuses possibilités de fonctionnalités :

- Rafraîchissement de données : par exemple rafraîchir le contenu d'une liste lors d'une pagination
- Auto-complétion d'une zone de saisie
- Validation de données en temps réel
- Modifier les données dans une table sans utiliser une page dédiée pour faire la mise à jour. Lors du clic sur un bouton modifier, il est possible de transformer les zones d'affichage en zones de saisie, d'utiliser AJAX pour envoyer une requête de mise à jour à la validation coté serveur et de réafficher les données modifiées à la place des zones de saisies
- ...

Les utilisations d'AJAX sont donc nombreuses mais cette liste n'est pas exhaustive : cependant elle permet déjà de comprendre qu'AJAX peut rendre les applications web plus dynamiques et interactives.

Les technologies requises pour mettre en oeuvre AJAX sont disponibles depuis plusieurs années (les concepts proposés par AJAX ne sont pas récents puisque Microsoft proposait déjà une solution équivalente dans Internet Explorer 5).

La mise à disposition de ces concepts dans la plupart des navigateurs récents permet à AJAX de connaître un énorme engouement essentiellement justifié par les fonctionnalités proposées et par des mises en oeuvres à succès des sites tels que Google Gmail, Google Suggest ou Google GMaps. Cet engouement va jusqu'à qualifier de façon générale l'utilisation d'AJAX et quelques autres concepts sous le terme Web 2.0.

Le succès d'AJAX est assuré par le fait qu'il apporte aux utilisateurs d'applications web des fonctionnalités manquantes déjà bien connues dans les applications de type standalone. La mise à jour dynamique de la page apporte aux utilisateurs une convivialité et une rapidité dans les applications web.

L'accroissement de l'utilisation d'AJAX permet de voir apparaître des frameworks qui facilitent sa mise en oeuvre et son intégration dans les applications. Un de ces framework, le framework DWR, est présenté dans ce chapitre.

Le Java BluePrints de Sun recense les meilleures pratiques d'utilisation d'AJAX avec J2EE : chaque référence propose une description, une solution de conception et un exemple de code fonctionnel mettant en oeuvre la solution. Ces références sont actuellement l'auto-complétion, une barre de progression et la validation des données d'un formulaire.

AJAX est aussi en cours d'intégration dans les Java Server Faces.

Ce chapitre contient plusieurs sections :

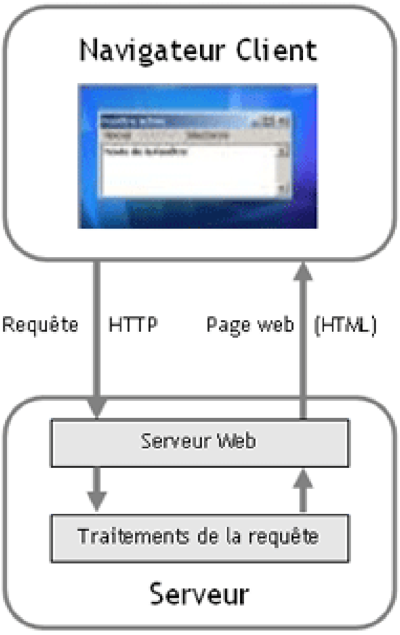
- ♦ La présentation d'AJAX
- ♦ Le détail du mode de fonctionnement
- ♦ Un exemple simple
- ♦ Des frameworks pour mettre en oeuvre AJAX

88.1. La présentation d'AJAX

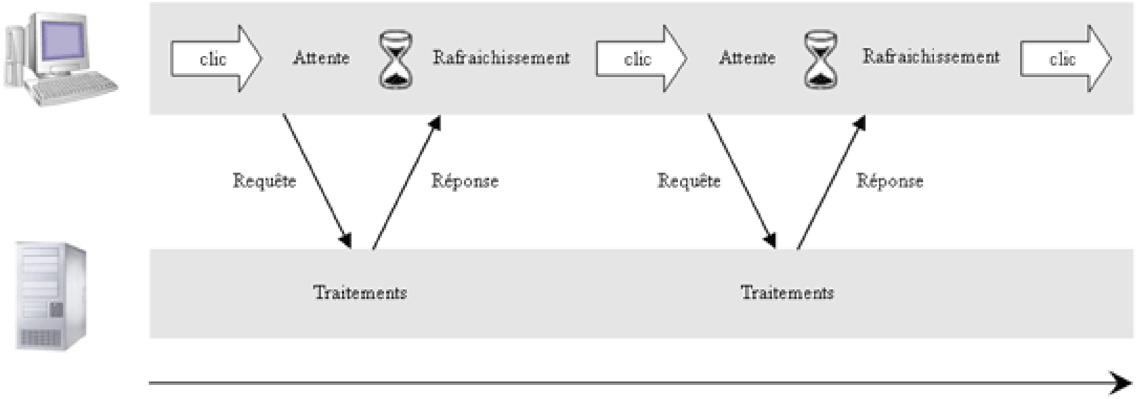
Traditionnellement les pages HTML ont besoin d'être entièrement rafraîchies dès lors qu'une simple portion de la page doit être rafraîchie. Ce mode de fonctionnement possède plusieurs inconvénients :

- limite les temps de réponse de l'application,
- augmente la consommation de bande passante et de ressources côté serveur
- perte du contexte lié au protocole http (utilisation de mécanismes tels que les cookies ou la session pour conserver un état)

Dans une application web, les échanges entre le client et le serveur sont opérés de manière synchrone. Chaque appel nécessitant un traitement côté serveur impose un rafraîchissement complet de la page. Le mode de fonctionnement d'une application web classique est donc le suivant :



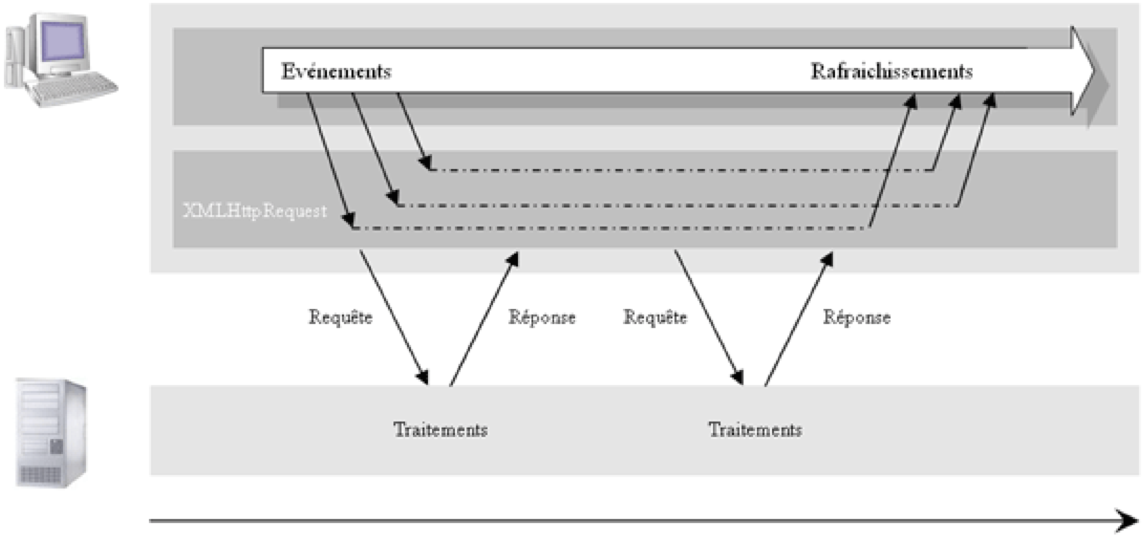
Avant AJAX, les applications web fonctionnaient sur un mode soumission/attente/rafraîchissement total de la page. Chaque appel au serveur retourne la totalité de la page qui est donc entièrement reconstruite et retournée au navigateur pour affichage. Durant cet échange, l'utilisateur est obligé d'attendre la réponse du serveur ce qui implique au mieux un clignotement lors du rafraîchissement de la page ou l'affichage d'une page blanche en fonction du temps de traitement de la requête par le serveur.



AJAX repose essentiellement sur un échange asynchrone entre le client et le serveur ce qui évite aux utilisateurs d'avoir un temps d'attente obligatoire entre leur action et la réponse correspondante tant qu'ils restent dans la même page. Ce mode de communication et le rafraîchissement partiel de la page en fonction des données reçues en réponse du serveur permettent d'avoir une meilleure réactivité aux actions de l'utilisateur.

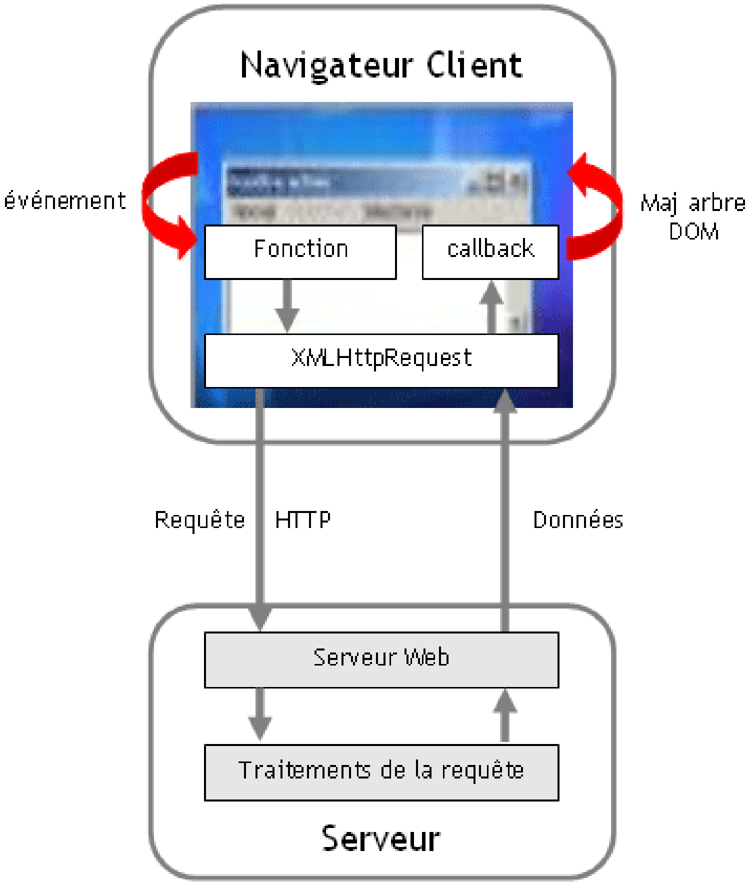
Avec AJAX :

- Le rafraîchissement partiel d'une page remplace le rafraîchissement systématique total de la page
- La communication asynchrone remplace la communication synchrone entre le client et le serveur. Ceci permet d'améliorer l'interactivité entre l'utilisateur et l'application



L'utilisation d'AJAX dans une application web permet des communications asynchrones et à l'utilisateur de rester dans la page courante. La mise à jour dynamique de la page en fonction de la réponse permet de rendre ses traitements transparents pour l'utilisateur et surtout de lui donner une impression de fluidité.

Le mode de fonctionnement d'une application web utilisant AJAX est le suivant :



Tous ces traitements sont déclenchés par un événement utilisateur sur un composant (clic, changement d'une valeur, perte du focus, ...) ou système (timer, ...) dans la page.

La partie centrale d'AJAX est un moteur capable de communiquer de façon asynchrone avec un serveur en utilisant le protocole http. Généralement les appels au serveur se font grâce à l'objet Javascript XMLHttpRequest. Cet objet n'est pas défini dans les spécifications courantes de JavaScript mais il est implémenté dans tous les navigateurs récents car il devient un standard de facto. Il est donc important de noter qu'AJAX ne fonctionnera pas sur des navigateurs anciens : ceci est à prendre en compte lors d'une volonté d'utilisation d'AJAX ou lors de sa mise en oeuvre.

L'objet JavaScript XMLHttpRequest occupe donc un rôle majeur dans AJAX puisqu'il assure les communications entre le client et le serveur. Généralement ces communications sont asynchrones pour permettre à l'utilisateur de poursuivre ses activités dans la page.

AJAX nécessite une architecture différente côté serveur : habituellement en réponse à une requête le serveur renvoie le contenu de toute la page. En réponse à une requête faite grâce à AJAX, le serveur doit renvoyer des informations qui seront utilisées côté client par du code JavaScript pour mettre à jour la page. Le format de ces informations est généralement XML mais ce n'est pas une obligation.

Le rafraîchissement partiel d'une page grâce à AJAX permet d'accroître la réactivité de l'IHM mais aussi de diminuer la bande passante et les ressources serveurs consommées lors d'un rafraîchissement complet de la page web.

La mise à jour partielle d'une page en modifiant directement son arbre DOM permet de conserver le contexte de l'état de la page. Les parties inchangées du DOM restent toujours connues et utilisables. Cependant un des effets pervers pour l'utilisateur est l'utilisation du bouton back du navigateur : l'utilisateur est habitué à obtenir l'état précédent de la page dans le cas d'un rafraîchissement à chaque action. Ce nouveau mode peut être déroutant pour l'utilisateur.

88.2. Le détail du mode de fonctionnement

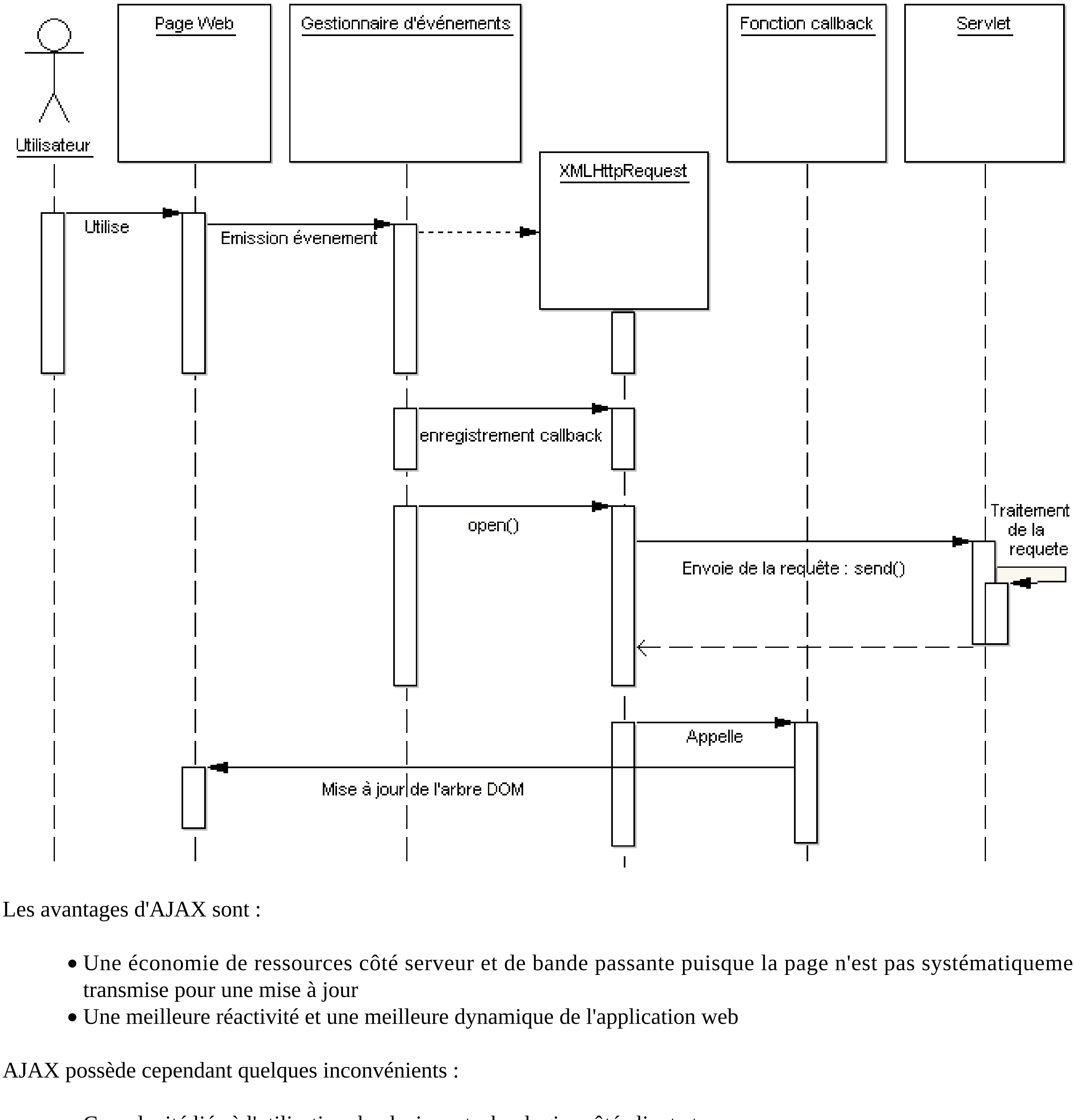
La condition pour utiliser AJAX dans une application web est que le support de JavaScript soit activé dans le navigateur et que celui-ci propose une implémentation de l'objet XMLHttpRequest.

L'objet XMLHttpRequest permet un échange synchrone ou asynchrone avec le serveur en utilisant le protocole HTTP. La requête http envoyée au serveur peut être de type GET ou POST.

Une communication asynchrone permet au navigateur de ne pas bloquer les actions de l'utilisateur en attendant la réponse du serveur. Ainsi, une fonction de type callback est enregistrée pour permettre son appel à la réception de la réponse http.

Côté serveur toute technologie permettant de répondre à une requête http peut être utilisée avec AJAX. J2EE et plus particulièrement les servlets se prêtent particulièrement bien à ces traitements. La requête http est traitée comme toutes les requêtes de ce type. En fonction des paramètres reçus de la requête, des traitements sont exécutés pour générer la réponse http.

A la réception de la réponse par le client, la fonction de type callback est appelée. Elle se charge d'extraire les données de la réponse et de réaliser les traitements de mise à jour de la page web en manipulant son arbre DOM.



Les avantages d'AJAX sont :

- Une économie de ressources côté serveur et de bande passante puisque la page n'est pas systématiquement transmise pour une mise à jour
- Une meilleure réactivité et une meilleure dynamique de l'application web

AJAX possède cependant quelques inconvénients :

- Complexité liée à l'utilisation de plusieurs technologies côté client et serveur
- Utilisation de JavaScript : elle implique la prise en compte des inconvénients de cette technologie : difficulté pour déboguer, différences d'implémentations selon le navigateur, code source visible, ...
- AJAX ne peut être utilisé qu'avec des navigateurs possédant une implémentation de l'objet XMLHttpRequest
- L'objet XMLHttpRequest n'est pas standardisé ce qui nécessite des traitements JavaScript dépendants du navigateur utilisé

- Le changement du mode de fonctionnement des applications web (par exemple : impossible de faire un favori vers une page dans un certain état, le bouton back ne permet plus de réafficher la page dans son état précédent la dernière action, ...)
- La mise en oeuvre de nombreuses fonctionnalités mettant en oeuvre AJAX peut faire rapidement augmenter le nombre de requêtes http à traiter par le serveur
- Le manque de frameworks et d'outils pour faciliter la mise en oeuvre

AJAX possède donc quelques inconvénients qui nécessitent une sérieuse réflexion pour une utilisation intensive dans une application. Un bon compromis est d'utiliser AJAX pour des fonctionnalités permettant une amélioration de l'interactivité entre l'application et l'utilisateur.

Actuellement, AJAX et en particulier l'objet XMLHttpRequest n'est pas un standard. De plus, reposant essentiellement sur JavaScript, son bon fonctionnement ne peut pas être assuré sur tous les navigateurs. Pour ceux avec qui cela peut l'être, le support de JavaScript doit être activé et il est quasiment impératif d'écrire du code dépendant du navigateur utilisé.

Il peut donc être nécessaire de prévoir, lors du développement de l'application, le bon fonctionnement de cette dernière sans utiliser AJAX. Cela permet notamment un fonctionnement correct sur les anciens navigateurs ou sur les navigateurs où le support de JavaScript est désactivé.

Le plus simple pour assurer cette tâche est de détecter au démarrage de l'application si l'objet XMLHttpRequest est utilisable dans le navigateur de l'utilisateur. Dans l'affirmative, l'application renvoie une version avec AJAX de la page sinon une version sans AJAX.

Comme la requête est asynchrone, il peut être important d'informer l'utilisation sur l'état des traitements en cours et surtout sur le succès ou l'échec de leur exécution. Avec un rafraîchissement traditionnel complet de la page c'est facile. En utilisant AJAX, il est nécessaire de faire usage de subtilités d'affichage ou d'effets visuels auxquels l'utilisateur n'est pas forcement habitué. Un exemple concret concerne un bouton de validation : il est utile de modifier le libellé du bouton pour informer l'utilisateur que les traitements sont en cours afin d'éviter qu'il clique plusieurs fois sur le bouton.

Il faut aussi garder à l'esprit que les échanges asynchrones ne garantissent pas que les réponses arrivent dans le même ordre que les requêtes correspondantes sont envoyées. Il est même tout à fait possible de ne jamais recevoir une réponse. Il faut donc être prudent si l'on enchaîne plusieurs requêtes.

88.3. Un exemple simple

Cet exemple va permettre de réaliser une validation côté serveur d'une donnée saisie en temps réel.

Une servlet permettra de réaliser cette validation. La validation proposée est volontairement simpliste et pourrait même être réalisée directement côté client avec du code JavaScript. Il faut cependant comprendre que les traitements de validation pourraient être beaucoup plus complexes avec par exemple une recherche dans une base de données, ce qui justifierait pleinement l'emploi d'une validation côté serveur.

Les actions suivantes sont exécutées dans cet exemple :

- Un événement déclencheur est émis (la saisie d'une donnée par l'utilisateur)
- Création et paramétrage d'un objet de type XMLHttpRequest
- Appel de la servlet par l'objet XMLHttpRequest
- La servlet exécute les traitements de validation et renvoie le résultat en réponse au format XML
- L'objet XMLHttpRequest appelle la fonction d'exploitation de la réponse
- La fonction met à jour l'arbre DOM de la page en fonction des données de la réponse

88.3.1. L'application de tests

La page de test est une JSP qui contient un champ de saisie.

Exemple :

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Test validation AJAX</title>
<script type="text/javascript">
<!--
var requete;

function valider() {
   var donnees = document.getElementById("donnees");
   var url = "valider?valeur=" + escape(donnees.value);
   if (window.XMLHttpRequest) {
       requete = new XMLHttpRequest();
   } else if (window.ActiveXObject) {
       requete = new ActiveXObject("Microsoft.XMLHTTP");
   }
   requete.open("GET", url, true);
   requete.onreadystatechange = majIHM;
   requete.send(null);
}

function majIHM() {
  var message = "";

  if (requete.readyState == 4) {
    if (requete.status == 200) {
      // exploitation des données de la réponse
      var messageTag = requete.responseXML.getElementsByTagName("message")[0];
      message = messageTag.childNodes[0].nodeValue;
      mdiv = document.getElementById("validationMessage");
      if (message == "invalide") {
         mdiv.innerHTML = "<img src='images/invalide.gif'>";
      } else {
         mdiv.innerHTML = "<img src='images/valide.gif'>";
      }
    }
  }
}

//-->
</script>
</head>
<body>
<table>
        <tr>
                <td>Valeur :</td>
                <td nowrap><input type="text" id="donnees" name="donnees" size="30"
                        onkeyup="valider();"></td>
                <td>
                <div id="validationMessage"></div>
                </td>
        </tr>
</table>
</body>
</html>
```

Le code JavaScript est détaillé dans les sections suivantes.

L'application contient aussi une servlet qui sera détaillée dans une des sections suivantes.

Le descripteur de déploiement de l'application contient la déclaration de la servlet.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
     "web-app_2_2.dtd">
<web-app>
```

```
        <display-name>Test de validation avec AJAX</display-name>
        <servlet>
                <servlet-name>ValiderServlet</servlet-name>
                <display-name>ValiderServlet</display-name>
                <description>Validation de données</description>
                <servlet-class>
                fr.jmdoudoux.dej.ajax.ValiderServlet</servlet-class>
        </servlet>
        <servlet-mapping>
                <servlet-name>ValiderServlet</servlet-name>
                <url-pattern>/valider</url-pattern>
        </servlet-mapping>
        <welcome-file-list>
                <welcome-file>index.jsp</welcome-file>
        </welcome-file-list>
</web-app>
```

88.3.2. La prise en compte de l'événement déclencheur

Un événement onkeyup est associé à la zone de saisie des données. Cet événement va appeler la fonction JavaScript valider().

Exemple :

```
<input type="text" id="donnees" name="donnees" size="30"
              onkeyup="valider();">
```

Ainsi la fonction sera appelée à chaque fois que l'utilisateur saisit un caractère.

88.3.3. La création d'un objet de type XMLHttpRequest pour appeler la servlet

La fonction JavaScript valider() va réaliser les traitements de la validation des données.

Exemple :

```
var requete;

function valider() {
   var donnees = document.getElementById("donnees");
   var url = "valider?valeur=" + escape(donnees.value);
   if (window.XMLHttpRequest) {
      requete = new XMLHttpRequest();
      requete.open("GET", url, true);
      requete.onreadystatechange = majIHM;
      requete.send(null);
   } else if (window.ActiveXObject) {
      requete = new ActiveXObject("Microsoft.XMLHTTP");
      if (requete) {
         requete.open("GET", url, true);
         requete.onreadystatechange = majIHM;
         requete.send();
      }
   } else {
      alert("Le navigateur ne supporte pas la technologie AJAX");
   }
}
```

Elle réalise les traitements suivants :

- récupère les données saisies
- détermine l'url d'appel de la servlet en passant en paramètre les données. Ces données sont encodées selon la norme http grâce à la fonction escape().

- instancie une requête de type XMLHttpRequest en fonction du navigateur utilisé
- associe à la requête l'url et la fonction à exécuter à la réponse
- exécute la requête

Comme dans de nombreux usages courants de JavaScript, des traitements dépendants du navigateur cible sont nécessaires à l'exécution. Dans le cas de l'instanciation de l'objet XMLHttpRequest, celui-ci est un ActiveX sous Internet Explorer et un objet natif sur les autres navigateurs qui le supportent.

La signature de la méthode open de l'objet XMLHttpRequest est XMLHttpRequest.open(String method, String URL, boolean asynchronous).

Le premier paramètre est le type de requête http réalisé par la requête (GET ou POST)

Le second paramètre est l'url utilisée par la requête.

Le troisième paramètre est un booléen qui précise si la requête doit être effectuée de façon asynchrone. Si la valeur passée est true alors une fonction de type callback doit être associée à l'événement onreadystatechange de la requête. La fonction précisée sera alors exécutée à la réception de la réponse.

La méthode send() permet d'exécuter la requête http en fonction des paramètres de l'objet XMLHttpRequest.

Pour une requête de type GET, il suffit de passer null comme paramètre de la méthode send().

Pour une requête de type POST, il faut préciser le Content-Type dans l'en-tête de la requête et fournir les paramètres de la fonction send().

Exemple :

```
requete.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
requete.send("valeur=" + escape(donnees.value));
```

L'objet XMLHttpRequest possède les méthodes suivantes :

Méthode	Rôle
abort()	Abandon de la requête
getAllResponseHeaders()	Renvoie une chaîne contenant les en-têtes http de la réponse
getResponseHeader(nom)	Renvoie la valeur de l'en-tête dont le nom est fourni en paramètre
setTimeouts(duree)	Précise la durée maximale pour l'obtention de la réponse
setRequestHeader(nom, valeur)	Précise la valeur de l'en-tête dont le nom est fourni en paramètre
open(méthode, url, [asynchrone[, utilisateur[, motdepasse]]]	Prépare une requête en précisant la méthode (Get ou Post), l'url, un booléen optionnel qui précise si l'appel doit être asynchrone et le user et/ou le mot de passe optionnel
send(data)	Envoi de la requête au serveur

L'objet XMLHttpRequest possède les propriétés suivantes :

Propriété	Rôle
onreadystatchange	Précise la fonction de type callback qui est appelée lorsque la valeur de la propriété readyState change
readyState	L'état de la requête : 0 = uninitialized 1 = loading 2 = loaded 3 = interactive

	4 = complete
responseText	Le contenu de la réponse au format texte
responseXML	Le contenu de la réponse au format XML
status	Le code retour http de la réponse
statusText	La description du code retour http de la réponse

Il peut être intéressant d'utiliser une fonction JavaScript qui va générer une chaîne de caractères contenant le nom et la valeur de chacun des éléments d'un formulaire.

Exemple :

```
function getFormAsString(nomFormulaire){

  resultat ="";
  formElements=document.forms[nomFormulaire].elements;

  for(var i=0; i<formElements.length; i++ ){
    if (i > 0) {
      resultat+="&";
    }
    resultat+=escape(formElements[i].name)+"="
      +escape(formElements[i].value);
 }

 return resultat;
}
```

Ceci facilite la génération d'une url qui aurait besoin de toutes les valeurs d'un formulaire.

88.3.4. L'exécution des traitements et le renvoi de la réponse par la servlet

La servlet associée à l'URI "/valider" est exécutée par le conteneur web en réponse à la requête.

Exemple :

```
package fr.jmdoudoux.dej.ajax;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Servlet ValiderServlet
 *
 */
 public class ValiderServlet extends javax.servlet.http.HttpServlet
                          implements javax.servlet.Servlet {

  /* (non-Java-doc)
         * @see javax.servlet.http.HttpServlet#HttpServlet()
         */
        public ValiderServlet() {
                super();
        }

        /* (non-Java-doc)
         * @see javax.servlet.http.HttpServlet#doGet(HttpServletRequest request,
         *                                           HttpServletResponse response)
         */
        protected void doGet(HttpServletRequest request, HttpServletResponse response)
                             throws ServletException, IOException {
```

```
    String resultat = "invalide";
    String valeur = request.getParameter("valeur");

    response.setContentType("text/xml");
    response.setHeader("Cache-Control", "no-cache");

    if ((valeur != null) && valeur.startsWith("X")) {
      resultat = "valide";
    }

    response.getWriter().write("<message>"+resultat+"</message>");
  }
}
```

La validation est assurée si la valeur fournie commence par un caractère "X".

La servlet renvoie simplement un texte indiquant l'état de la validation réalisée dans une balise message.

Il est important que le type Mime retourné dans la réponse soit de type "text/xml".

Il est préférable de supprimer la mise en cache de la réponse par le navigateur. Cette suppression est obligatoire si une même requête peut renvoyer une réponse différente lors de plusieurs appels.

88.3.5. L'exploitation de la réponse

L'objet XMLHttpRequest appelle la fonction de type callback majIHM() à chaque fois que la propriété readyState change de valeur.

La fonction majIHM() commence donc par vérifier la valeur de la propriété readyState. Si celle-ci vaut 4 alors l'exécution de la requête est complète.

Dans ce cas, il faut vérifier le code retour de la réponse http. La valeur 200 indique que la requête a été correctement traitée.

Exemple :

```
function majIHM() {
  var message = "";

  if (requete.readyState == 4) {
    if (requete.status == 200) {
      // exploitation des données de la réponse
      // ...
    } else {
       alert('Une erreur est survenue lors de la mise à jour de la page');
    }
  }
}
```

En utilisant la valeur de la réponse, la fonction modifie alors le contenu de la page en mettant à jour son arbre DOM. Cette valeur au format XML est obtenue en utilisant la fonction responseXML de l'instance de XMLHttpRequest. La valeur au format texte brut peut être obtenue en utilisant la fonction responseText.

Il est alors possible d'exploiter les données de la réponse.

Exemple :

```
function majIHM() {
  var message = "";

  if (requete.readyState == 4) {
    if (requete.status == 200) {
      // exploitation des données de la réponse
      var messageTag = requete.responseXML.getElementsByTagName("message")[0];
      message = messageTag.childNodes[0].nodeValue;
      mdiv = document.getElementById("validationMessage");
      if (message == "invalide") {
         mdiv.innerHTML = "<img src='images/invalide.gif'>";
      } else {
         mdiv.innerHTML = "<img src='images/valide.gif'>";
      }
    } else {
      alert('Une erreur est survenue lors de la mise à jour de la page.'+
                '\n\nCode retour = '+requete.statusText);
    }
  }
}
```

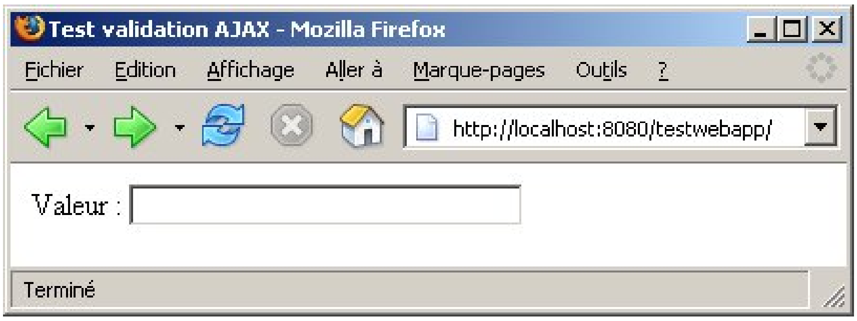
Il est aussi possible que la réponse contienne directement du code HTML à afficher. Il suffit simplement d'affecter le résultat de la réponse au format texte à la propriété innerHTML de l'élément de la page à rafraîchir.

Exemple :

```
function majIHM() {
  if (requete.readyState == 4) {
    if (requete.status == 200) {
      document.getElementById("validationMessage").innerHTML = requete.responseText;
    } else {
      alert('Une erreur est survenue lors de la mise à jour de la page.'+
                '\n\nCode retour = '+requete.statusText);
  }
}
```

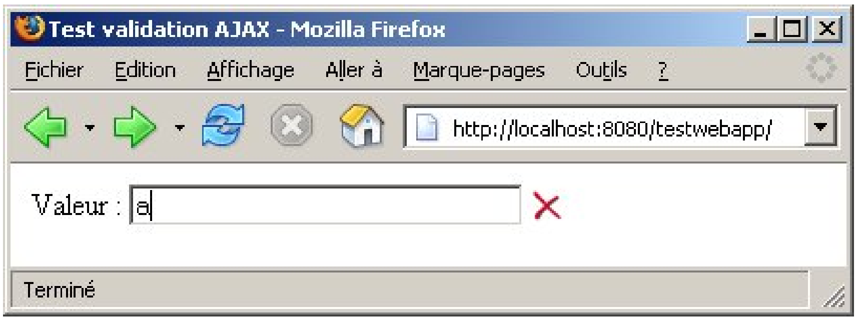
88.3.6. L'exécution de l'application

La page de test s'affiche au lancement de l'application


Test validation AJAX - Mozilla Firefox
Fichier
Edition
Affichage
Aller à
Marque-pages
Outils
?
http://localhost:8080/testwebapp/
Terminé

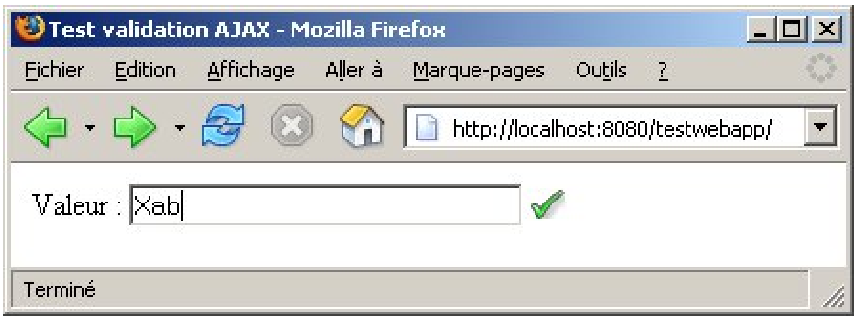
Valeur :

La saisie d'un caractère déclenche la validation


Test validation AJAX - Mozilla Firefox
Fichier
Edition
Affichage
Aller à
Marque-pages
Outils
?
http://localhost:8080/testwebapp/
Terminé

Valeur : a ✗

L'icône dépend du résultat de la validation.


Test validation AJAX - Mozilla Firefox
Fichier
Edition
Affichage
Aller à
Marque-pages
Outils
?
http://localhost:8080/testwebapp/
Terminé

Valeur : Xab

88.4. Des frameworks pour mettre en oeuvre AJAX

La mise en oeuvre directe de l'objet XmlHttpRequest est relativement lourde (nécessite l'écriture de nombreuses lignes de code), fastidieuse (pas facile à déboguer) et souvent répétitive. La mise en oeuvre de plusieurs technologies côté client et serveur peut engendrer de nombreuses difficultés notamment dans le code JavaScript (débogage difficile, gestion de la compatibilité du support par les navigateurs, ...).

Aussi de nombreux frameworks commencent à voir le jour pour faciliter le travail des développeurs. Cette section va détailler l'utilisation du framework DWR et proposer une liste non exhaustive d'autres frameworks.

88.4.1. Direct Web Remoting (DWR)

DWR (Direct Web Remoting) est une bibliothèque open source Java dont le but est de faciliter la mise en oeuvre d'AJAX dans les applications Java.

DWR se charge de générer le code JavaScript permettant l'appel à des objets Java de type bean qu'il suffit d'écrire. Sa devise est "Easy AJAX for Java".

DWR encapsule les interactions entre le code JavaScript côté client et les objets Java côté serveur : ceci rend transparent l'appel de ces objets côté client.

La mise en oeuvre de DWR côté serveur est facile :

- Ajouter le fichier dwr.jar au classpath de l'application
- Configurer une servlet dédiée aux traitements des requêtes dans le fichier web.xml
- Ecrire les beans qui seront utilisés dans les pages
- Définir ces beans dans un fichier de configuration de DWR

La mise en oeuvre côté client nécessite d'inclure des bibliothèques JavaScript générées dynamiquement par la servlet de DWR. Il est alors possible d'utiliser les fonctions JavaScript générées pour appeler les méthodes des beans configurés côté serveur.

DWR s'intègre facilement dans une application web puisqu'il repose sur une servlet. Elle s'intégrera plus particulièrement avec les applications mettant en oeuvre le framework Spring dont elle propose un support. DWR est aussi inclus dans le framework WebWork depuis sa version 2.2.

DWR fournit aussi une bibliothèque JavaScript proposant des fonctions de manipulations courantes en DHTML : modifier le contenu des conteneurs <DIV> ou <SPAN>, remplir une liste déroulante avec des valeurs, etc ...

DWR est une solution qui encapsule l'appel de méthodes de simples objets de type Javabean exécutés sur le serveur dans du code JavaScript généré dynamiquement. Le grand intérêt est de masquer toute la complexité de l'utilisation de l'objet XMLHttpRequest et de simplifier à l'extrême le code à développer côté serveur.

DWR se compose de deux parties :

- Du code JavaScript qui envoie des requêtes à la servlet et met à jour la page à partir des données de la réponse
- Une servlet qui traite les requêtes reçues et renvoie une réponse au navigateur

Côté serveur, une servlet est déployée dans l'application web. Cette servlet a deux rôles principaux :

1. Elle permet de générer dynamiquement des bibliothèques de code JavaScript. Deux de celles-ci sont à usage général. Une bibliothèque de code est générée pour chaque bean défini dans la configuration de DWR
2. Elle permet de traiter les requêtes émises par le code JavaScript générés pour appeler la méthode d'un bean

DWR génère dynamiquement le code JavaScript à partir des Javabeans configurés dans un fichier de paramètres en utilisant l'introspection. Ce code se charge d'encapsuler les appels aux méthodes du bean, ceci incluant la conversion du format des données de JavaScript vers Java et vice versa. Ce mécanisme est donc similaire à d'autres solutions de type RPC (remote procedure call).

Une fonction de type callback est précisée à DWR pour être exécutée par un bean à la réception de la réponse à la requête.

DWR facilite donc la mise en oeuvre d'AJAX avec Java côté serveur : il se charge de toute l'intégration de Javabeans pour permettre leur appel côté client de manière transparente.

Le site de DWR est à l'url : https://github.com/directwebremoting/dwr

La documentation de ce projet est particulièrement riche et de nombreux exemples sont fournis sur le site.

La version utilisée dans cette section est la version 1.1.1. Elle nécessite un JDK 1.3 et conteneur web supportant la version 2.2 de l'API servlet.

88.4.1.1. Un exemple de mise en oeuvre de DWR

Il faut télécharger le fichier dwr.jar sur le site officiel de DWR et l'ajouter dans le répertoire WEB-INF/Lib de l'application web qui va utiliser la bibliothèque.

Il faut ensuite déclarer dans le fichier de déploiement de l'application web.xml la servlet qui sera utilisée par DWR. Il faut déclarer la servlet et définir son mapping :

Exemple :

```
        <servlet>
                <servlet-name>dwr-invoker</servlet-name>
                <display-name>DWR Servlet</display-name>
                <servlet-class>uk.ltd.getahead.dwr.DWRServlet</servlet-class>

                <init-param>
                        <param-name>debug</param-name>
                        <param-value>true</param-value>
                </init-param>
        </servlet>

        <servlet-mapping>
                <servlet-name>dwr-invoker</servlet-name>
                <url-pattern>/dwr/*</url-pattern>
        </servlet-mapping>
```

Il faut créer un fichier de configuration pour DWR nommé dwr.xml dans le répertoire WEB-INF de l'application

Exemple :

```
<!DOCTYPE dwr PUBLIC
    "-//GetAhead Limited//DTD Direct Web Remoting 1.0//EN"
    "http://www.getahead.ltd.uk/dwr/dwr10.dtd">
<dwr>
  <allow>
    <create creator="new" javascript="JDate">
```

```
      <param name="class" value="java.util.Date"/>
    </create>
  </allow>
</dwr>
```

Ce fichier permet de déclarer à DWR la liste des beans qu'il devra encapsuler pour des appels en JavaScript. Dans l'exemple, c'est la classe java.util.Date fournie dans l'API standard qui est utilisée.

Le creator de type "new" instancie la classe en utilisant le constructeur sans argument. L'attribut javascript permet de préciser le nom de l'objet JavaScript qui sera utilisé côté client.

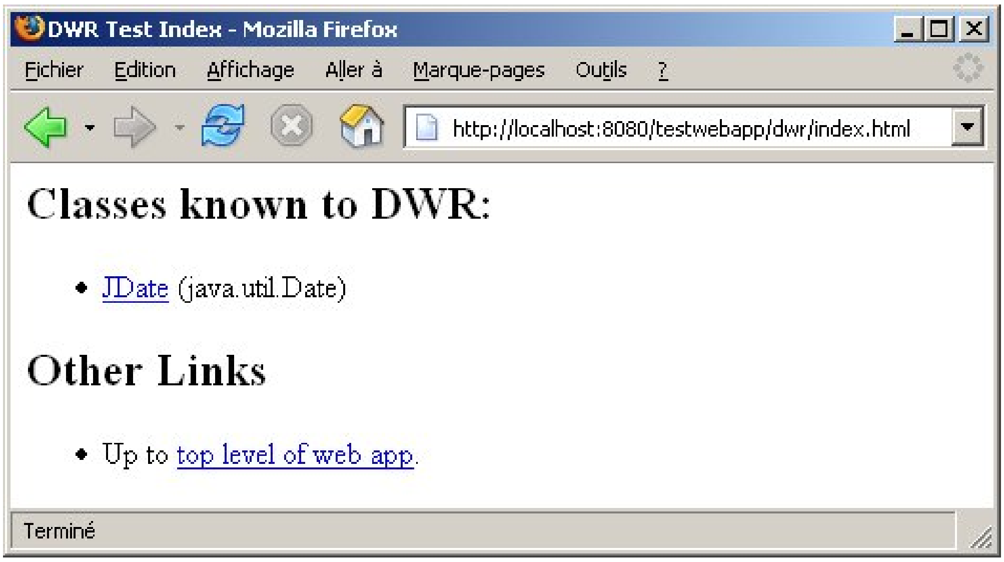
Le tag param avec l'attribut name ayant pour valeur class permet de préciser le nom pleinement qualifié du Bean à encapsuler.

DWR possède quelques restrictions :

- Il ne faut surtout pas utiliser comme noms de méthodes dans les beans exposés des mots réservés en JavaScript. Un exemple courant est le mot delete
- Il faut éviter l'utilisation de méthodes surchargées

Par défaut, DWR encapsule toutes les méthodes public de la classe définie. Il est donc nécessaire de limiter les méthodes utilisables par DWR à celles requises par les besoins de l'application soit dans la définition des membres de la classe soit dans le fichier de configuration de DWR.

Il suffit alors de lancer l'application et d'ouvrir un navigateur sur l'url de l'application en ajoutant /dwr


DWR Test Index - Mozilla Firefox
Fichier
Edition
Affichage
Aller à
Marque-pages
Outils
?
http://localhost:8080/testwebapp/dwr/index.html
Terminé

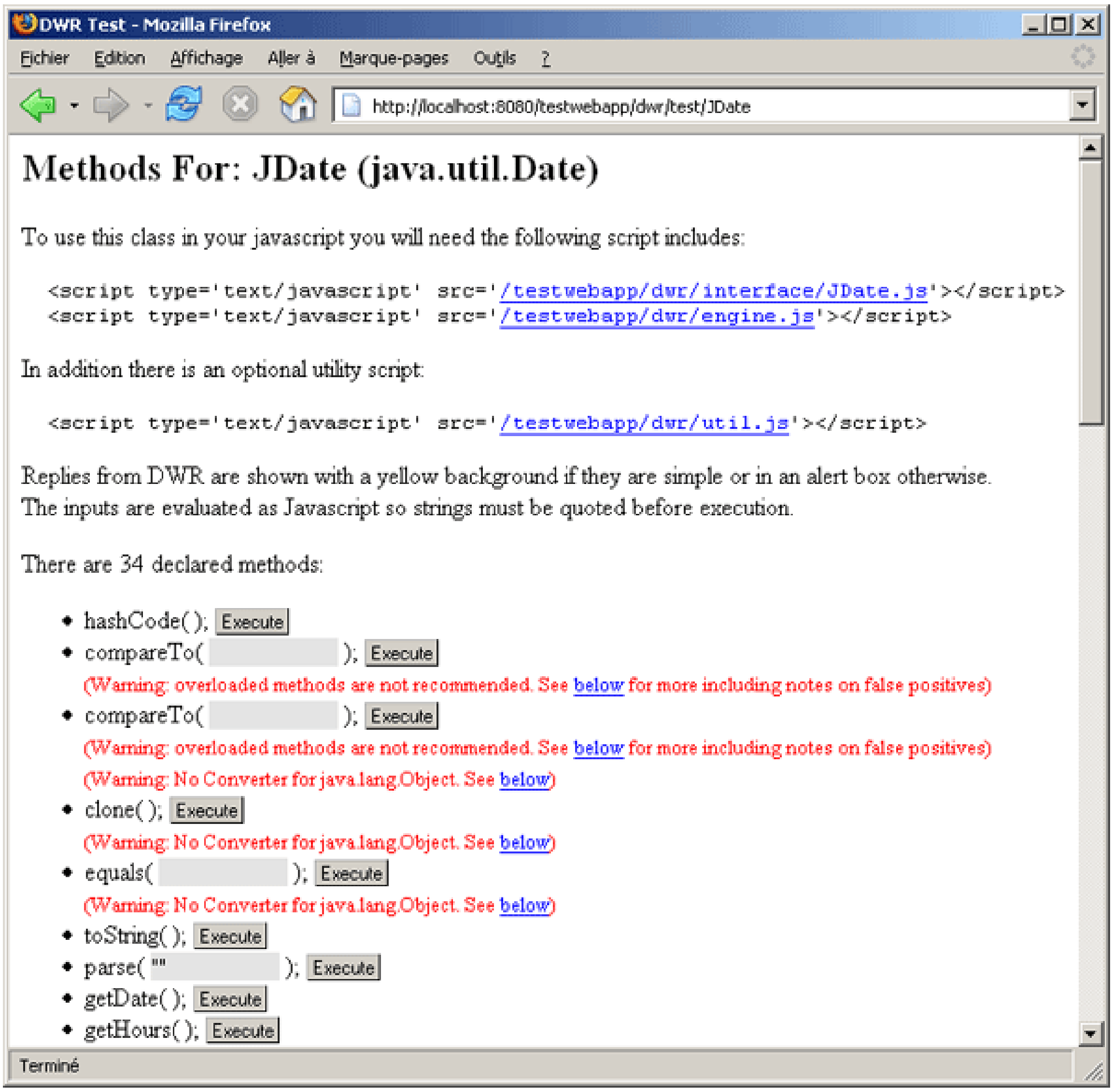
Classes known to DWR:

- JDate (java.util.Date)

Other Links

- Up to top level of web app.

Cette page liste tous les beans qui sont encapsulés par DWR. Il suffit de cliquer sur le lien d'un bean pour voir afficher une page de test de ce bean. Cette page génère dynamiquement une liste de toutes les méthodes pouvant être appelées en utilisant DWR.


DWR Test - Mozilla Firefox
Fichier Edition Affichage Aller à Marque-pages Outils ?
http://localhost:8080/testwebapp/dwr/test/JDate
Terminé

Methods For: JDate (java.util.Date)

To use this class in your javascript you will need the following script includes:

```
<script type='text/javascript' src='/testwebapp/dwr/interface/JDate.js'></script>
<script type='text/javascript' src='/testwebapp/dwr/engine.js'></script>
```

In addition there is an optional utility script:

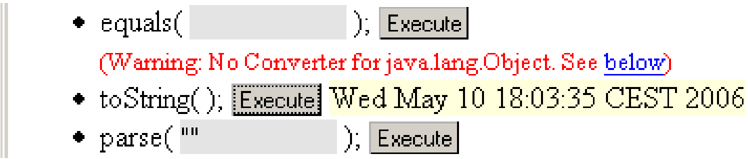
```
<script type='text/javascript' src='/testwebapp/dwr/util.js'></script>
```

Replies from DWR are shown with a yellow background if they are simple or in an alert box otherwise.
The inputs are evaluated as Javascript so strings must be quoted before execution.

There are 34 declared methods:

- hashCode(); Execute
- compareTo( ); Execute
  (Warning: overloaded methods are not recommended. See below for more including notes on false positives)
- compareTo( ); Execute
  (Warning: overloaded methods are not recommended. See below for more including notes on false positives)
  (Warning: No Converter for java.lang.Object. See below)
- clone(); Execute
  (Warning: No Converter for java.lang.Object. See below)
- equals( ); Execute
  (Warning: No Converter for java.lang.Object. See below)
- toString(); Execute
- parse(""); Execute
- getDate(); Execute
- getHours(); Execute

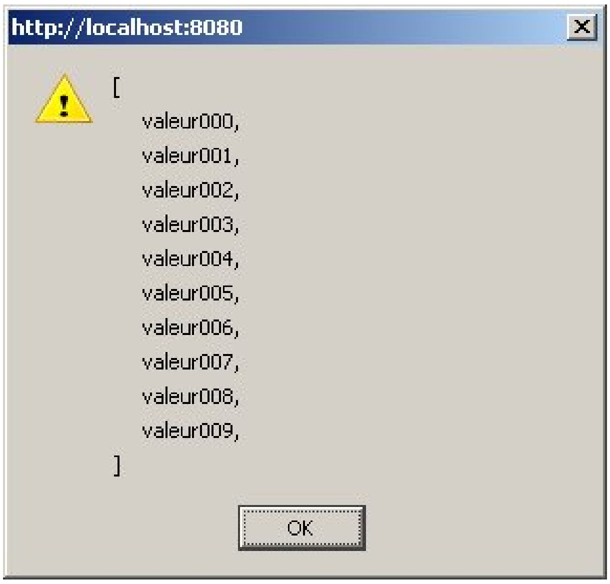
Pour exécuter dynamiquement une méthode sans paramètre, il suffit de simplement cliquer sur le bouton "Execute" de la méthode correspondante.



- equals( ); Execute
  (Warning: No Converter for java.lang.Object. See below)
- toString(); Execute Wed May 10 18:03:35 CEST 2006
- parse(""); Execute

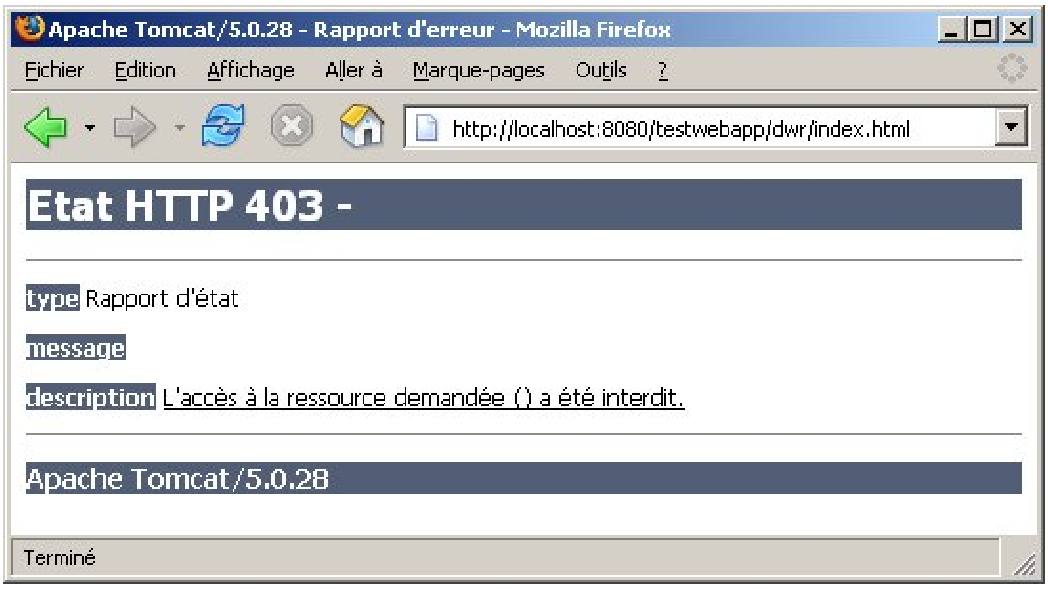
Pour exécuter dynamiquement une méthode avec paramètres, il suffit de saisir leurs valeurs dans leurs zones respectives et de cliquer sur le bouton "Execute".

Si la valeur retournée par la méthode n'est pas une valeur simple alors le résultat est affiché dans une boîte de dialogue.



```
[
    valeur000,
    valeur001,
    valeur002,
    valeur003,
    valeur004,
    valeur005,
    valeur006,
    valeur007,
    valeur008,
    valeur009,
]
```

Si le paramètre debug de la servlet DWR est à false, il n'est pas possible d'accéder à ses fonctionnalités de tests.


Apache Tomcat/5.0.28 - Rapport d'erreur - Mozilla Firefox
Fichier
Edition
Affichage
Aller à
Marque-pages
Outils
?
http://localhost:8080/testwebapp/dwr/index.html
Terminé

Etat HTTP 403 -

type Rapport d'état

message

description L'accès à la ressource demandée () a été interdit.

Apache Tomcat/5.0.28

Ce mode debug proposé par DWR est particulièrement utile lors de la phase de développement pour vérifier toutes les méthodes qui sont prises en compte par DWR et les tester. Pour des raisons de sécurité, il est fortement déconseillé de l'autoriser dans un contexte de production.

Pour permettre l'utilisation des scripts générés, il suffit de faire un copier/coller dans la partie en-tête de la page HTML des tags <SCRIPT> proposés dans la page de tests de DWR.

Exemple :

```
<script type='text/javascript' src='/testwebapp/dwr/interface/JDate.js'></script>
<script type='text/javascript' src='/testwebapp/dwr/engine.js'></script>
<script type='text/javascript' src='/testwebapp/dwr/util.js'></script>
```

Remarque : il est possible d'utiliser un chemin relatif plutôt qu'un chemin absolu pour ces ressources.

88.4.1.2. Le fichier DWR.xml

Le fichier dwr.xml permet de configurer DWR. Il est généralement placé dans le répertoire WEB-INF de l'application web exécutant DWR.

Le fichier dwr.xml a la structure suivante :

Exemple :

```
<!DOCTYPE dwr PUBLIC
    "-//GetAhead Limited//DTD Direct Web Remoting 1.0//EN"
    "http://www.getahead.ltd.uk/dwr/dwr10.dtd">

<dwr>

  <init>
    <creator id="..." class="..."/>
    <converter id="..." class="..."/>
  </init>

  <allow>
    <create creator="..." javascript="..."/>
    <convert converter="..." match="..."/>
  </allow>

  <signatures>
    ...
  </signatures>

</dwr>
```

Le tag optionnel <init> permet de déclarer ses propres créateurs et convertisseurs. Généralement, ce tag n'est pas utilisé car les créateurs et convertisseurs fournis en standard sont suffisants.

Le tag <allow> permet de définir les objets qui seront utilisés par DWR.

Le tag <create> permet de préciser la façon dont un objet va être instancié. Chaque classe qui pourra être appelée par DWR doit être déclarée avec un tel tag. Ce tag possède la structure suivante :

Exemple :

```
<allow>
  <create creator="..." javascript="..." scope="...">
    <param name="..." value="..."/>
    <auth method="..." role="..."/>
    <exclude method="..."/>
    <include method="..."/>
  </create>
  ...
</allow>
```

Les tags fils <param>, <auth>, <exclude>, <include> sont optionnels

La déclaration d'au moins un créateur est obligatoire. Il existe plusieurs types de créateurs spécifiés par l'attribut creator du tag fils <create> :

Type de créateur	Rôle
new	Instancie l'objet avec l'opérateur new
null	Ne crée aucune instance. Ceci est utile si la ou les méthodes utilisées sont statiques
scripted	Instancie l'objet en utilisant un script via BSF
spring	Le framework Spring est responsable de l'instanciation de l'objet
jsf	Utilise des objets de JSF
struts	Utilise des ActionForms de Struts
pageflow	Permet l'action au PageFlow de Beehive ou WebLogic

L'attribut javascript permet de donner le nom de l'objet Javascript. Il ne faut pas utiliser comme valeur un mot réservé de JavaScript.

L'attribut optionnel scope permet de préciser la portée du bean. Les valeurs possibles sont : application, session, request et page. Sa valeur par défaut est page.

Le tag <param> permet de fournir des paramètres au créateur. Par exemple, avec le creator new, il est nécessaire de fournir en paramètre le nom pleinement qualifié de la classe à instancier

Exemple :

```
<!DOCTYPE dwr PUBLIC
    "-//GetAhead Limited//DTD Direct Web Remoting 1.0//EN"
    "http://www.getahead.ltd.uk/dwr/dwr10.dtd">
<dwr>
  <allow>
    <create creator="new" javascript="JDate">
      <param name="class" value="java.util.Date"/>
    </create>
    <create creator="new" javascript="TestDWR">
      <param name="class" value="fr.jmdoudoux.dej.ajax.dwr.TestDWR"/>
    </create>
  </allow>
</dwr>
```

Avec le fichier de configuration dwr.xml, DWR propose un mécanisme qui permet de limiter les méthodes qui lui seront accessibles. Les tags <include> et <exclude> permettent respectivement d'autoriser ou d'exclure l'utilisation d'une liste de méthodes. Ces deux tags sont mutuellement exclusifs, en l'absence de l'un deux, toutes les méthodes sont utilisables.

Le tag <auth> permet de gérer la sécurité d'accès en utilisant les rôles J2EE de l'application : DWR propose donc la prise en compte des rôles J2EE définis dans le conteneur web pour restreindre l'accès à certaines classes.

Le tag <converter> permet de préciser la façon dont un objet utilisé en paramètre ou en type de retour va être converti. Un convertisseur assure la transformation des données entre le format des objets client (Javascript) et serveur (Java).

Chaque bean utilisé en tant que paramètre doit être déclaré dans un tel tag. Par défaut, l'utilisation du tag <converter> est inutile pour les primitives, les wrappers de ces primitives (Integer, Float, ...), les classes String et java.util.Date, les tableaux de ces types, les collections (List, Set, Map, ...) et certains objets de manipulation XML issus de DOM, JDOM et DOM4J.

Les convertisseurs Bean et Objet fournis en standard doivent être explicitement utilisés dans le fichier dwr.xml pour des raisons de sécurité.

Exemple :

```
<!DOCTYPE dwr PUBLIC
    "-//GetAhead Limited//DTD Direct Web Remoting 1.0//EN"
    "http://www.getahead.ltd.uk/dwr/dwr10.dtd">
<dwr>
  <allow>
    <create creator="new" javascript="TestDWR">
      <param name="class" value="fr.jmdoudoux.dej.ajax.dwr.TestDWR"/>
    </create>

    <convert converter="bean" match="fr.jmdoudoux.dej.ajax.dwr.Personne"/>
  </allow>
</dwr>
```

Il est possible d'utiliser le caractère joker *

Exemple :

```
    <convert converter="bean" match="fr.jmdoudoux.dej.ajax.dwr.*"/>
    <convert converter="bean" match="*"/>
```

Le convertisseur Bean permet de convertir un Bean en un tableau associatif JavaScript et vice versa en utilisant les mécanismes d'introspection.

Exemple :

```
public class Personne {
  public void setNom(String nom) { ... }
  public void setTaille(int taille) { ... }
  // ...
}
```

L'appel d'une méthode acceptant la classe Personne en paramètre peut se faire de la manière suivante dans la partie cliente :

Exemple :

```
var personne = { nom:"Test", taille:33 };
TestDWR.setPersonne(personne);
```

Il est possible de restreindre l'accès à certaines propriétés d'un bean dans son convertisseur.

Exemple :

```
<convert converter="bean" match="fr.jmdoudoux.dej.ajax.dwr.Personne"/>
  <param name="exclude" value="dateNaissance, taille"/>
</convert>
```

Exemple :

```
<convert converter="bean" match="fr.jmdoudoux.dej.ajax.dwr.Personne"/>
  <param name="include" value="nom, prenom"/>
</convert>
```

L'utilisation de ce dernier exemple est recommandée.

Le convertisseur Objet est similaire mais il utilise directement les membres plutôt que de passer par les getter/setter.

Il possède un paramètre force qui permet d'autoriser l'accès aux membres privés de l'objet par introspection.

Exemple :

```
<convert converter="object" match="fr.jmdoudoux.dej.ajax.dwr.Personne"/>
  <param name="force" value="true"/>
</convert>
```

88.4.1.3. Les scripts engine.js et util.js

Pour utiliser ces deux bibliothèques, il est nécessaire de les déclarer dans chaque page utilisant DWR.

Exemple :

```
<script type='text/javascript' src='/[WEB-APP]/dwr/engine.js'></script>
<script type='text/javascript' src='/[WEB-APP]/dwr/util.js'></script>
```

Le fichier engine.js est la partie principale côté JavaScript puisqu'il assure toute la gestion de la communication avec le serveur.

Certaines options de paramétrage peuvent être configurées en utilisant la fonction DWREngine.setX().

Il est possible de regrouper plusieurs communications en une seule en utilisant les fonctions DWREngine.beginBatch() et DWREngine.endBatch(). Lors de l'appel de cette dernière, les appels sont réalisés vers le serveur. Ce regroupement permet de réduire le nombre d'objets XMLHttpRequest créés et le nombre de requêtes envoyées au serveur.

Le fichier util.js propose des fonctions utilitaires pour faciliter la mise à jour dynamique de la page. Ces fonctions ne sont pas dépendantes d'autres éléments de DWR.

Fonction	Rôle
$(id)	Encapsuler un appel à la fonction document.getElementById() comme dans la bibliothèque Prototype
addOptions	Ajouter des éléments dans une liste ou un tag <ul> ou <ol>
removeAllOptions	Supprimer tous les éléments d'une liste ou d'un tag <ul> ou <ol>
addRows	Ajouter des lignes dans un tableau
removeAllRows	Supprimer toutes les lignes dans un tableau
getText	Renvoyer la valeur sélectionnée dans une liste
getValue	Renvoyer la valeur d'un élément HTML
getValues	Obtenir les valeurs de plusieurs éléments fournis sous la forme d'un ensemble de paires clé:valeur_vide dont la clé est l'id de l'élément à traiter
onReturn	Gérer l'appui sur la touche return avec un support multi-navigateur
selectRange(id, debut,fin)	Gérer une sélection dans une zone de texte avec un support multi-navigateur
setValue(id,value)	Mettre à jour la valeur d'un élément
setValues	Mettre à jour les valeurs de plusieurs éléments fournis sous la forme d'un ensemble de paires clé:valeur dont la clé est l'id de l'élément à modifier
toDescriptiveString(id, level)	Afficher des informations sur un objet avec un niveau de détail (0, 1ou 2)
useLoadingMessage	Mettre en place un message de chargement lors des échanges avec le serveur

88.4.1.4. Les scripts client générés

DWR assure un mapping entre les méthodes des objets Java et les fonctions JavaScript générées. Chaque objet Java est mappé sur un objet JavaScript dont le nom correspond à la valeur de l'attribut javascript du creator correspondant dans le fichier de configuration de DWR.

Le nom des méthodes est conservé comme nom de fonction dans le code JavaScript. Le premier paramètre de toutes les fonctions générées par DWR est la fonction de type callback qui sera exécutée à la réception de la réponse. Les éventuels autres paramètres correspondent à leurs équivalents dans le code Java.

DWR s'occupe de transformer un objet Java en paramètre ou en résultat en un équivalent dans le code JavaScript. Par exemple, une collection Java est transformée en un tableau d'objets JavaScript de façon transparente, l'utilisation des objets Java est donc nettement facilitée.

L'utilisation de la bibliothèque util.js peut être particulièrement pratique pour faciliter l'exploitation des données retournées et utilisées par les fonctions générées.

Des exemples d'utilisation sont fournis dans les sections d'exemples suivantes.

88.4.1.5. Un exemple pour obtenir le contenu d'une page

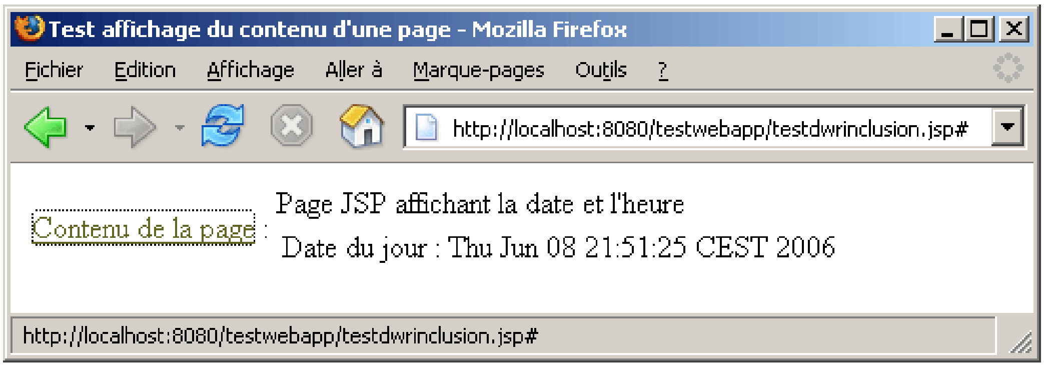
Il est possible qu'une méthode d'un bean renvoie le contenu d'une JSP en utilisant l'objet uk.ltd.getahead.dwr.ExecutionContext. Cet objet permet d'obtenir le contenu d'une url donnée.

Exemple : la JSP dont le contenu sera retourné

```
Page JSP affichant la date et l'heure
<table>
  <tr>
    <td>Date du jour :</td>
    <td nowrap><%=new java.util.Date()%></td>
  </tr>
</table>
```

Exemple :

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
  pageEncoding="ISO-8859-1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
  <title>Test affichage du contenu d'une page</title>
  <script type='text/javascript'
    src='/testwebapp/dwr/interface/TestDWR.js'></script>
  <script type='text/javascript' src='/testwebapp/dwr/engine.js'></script>
  <script type='text/javascript' src='/testwebapp/dwr/util.js'></script>

  <script type='text/javascript'>
  <!--

  function inclusion() {
    TestDWR.getContenuPage(afficherInclusion);
  }

  function afficherInclusion(data) {
    DWRUtil.setValue("inclusion", data);
  }

  function init() {
    DWRUtil.useLoadingMessage();
  }

  -->
  </script>
</head>
<body onload="init();">

<table>
  <tr>
    <td><a href="#" onclick="inclusion()">Contenu de la page</a> :</td>
    <td nowrap>
    <div id="inclusion"></div>
    </td>
  </tr>
</table>

</body>
</html>
```



Test affichage du contenu d'une page - Mozilla Firefox
Fichier
Edition
Affichage
Aller à
Marque-pages
Outils
?
http://localhost:8080/testwebapp/testdwrinclusion.jsp#

Contenu de la page : Page JSP affichant la date et l'heure

Date du jour : Thu Jun 08 21:51:25 CEST 2006

http://localhost:8080/testwebapp/testdwrinclusion.jsp#

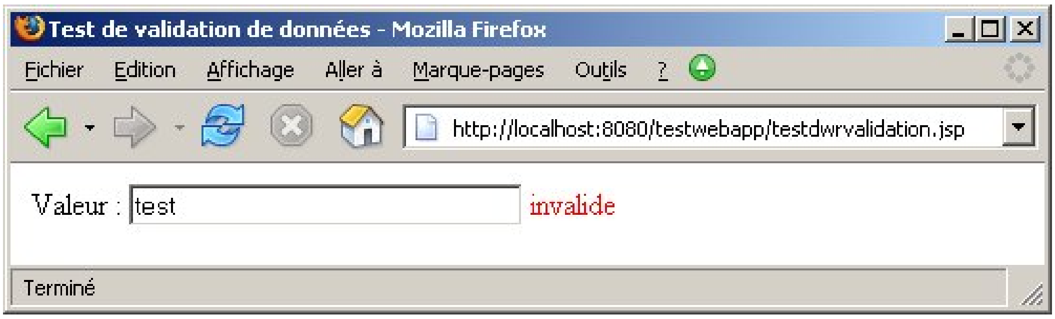
Lors d'un clic sur le lien, le contenu de la JSP est affiché dans le calque.

88.4.1.6. Un exemple pour valider des données

Dans cet exemple, à chaque saisie dans la zone de texte, le contenu est validé à la volée par un appel à une méthode d'un bean.

Exemple :

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
  <title>Test de validation de données</title>
  <script type='text/javascript' src='/testwebapp/dwr/interface/TestDWR.js'></script>
  <script type='text/javascript' src='/testwebapp/dwr/engine.js'></script>
  <script type='text/javascript' src='/testwebapp/dwr/util.js'></script>

  <script type='text/javascript'>
  <!--
  function valider() {
    TestDWR.validerValeur(afficherValidation, $("donnees").value);
  }

  function afficherValidation(data) {
    DWRUtil.setValue("validationMessage",data);
    if (data == "valide") {
      $("validationMessage").style.color='#00FF00';
    } else {
      $("validationMessage").style.color='#FF0000';
    }
  }

  function init() {
    DWRUtil.useLoadingMessage();
  }
  -->
  </script>
</head>
<body onload="init();">

<table>
        <tr>
                <td>Valeur :</td>
                <td nowrap><input type="text" id="donnees" name="donnees" size="30"
                        onkeyup="valider();"></td>
                <td>
                <div id="validationMessage"></div>
                </td>
        </tr>
</table>

</body>
</html>
```

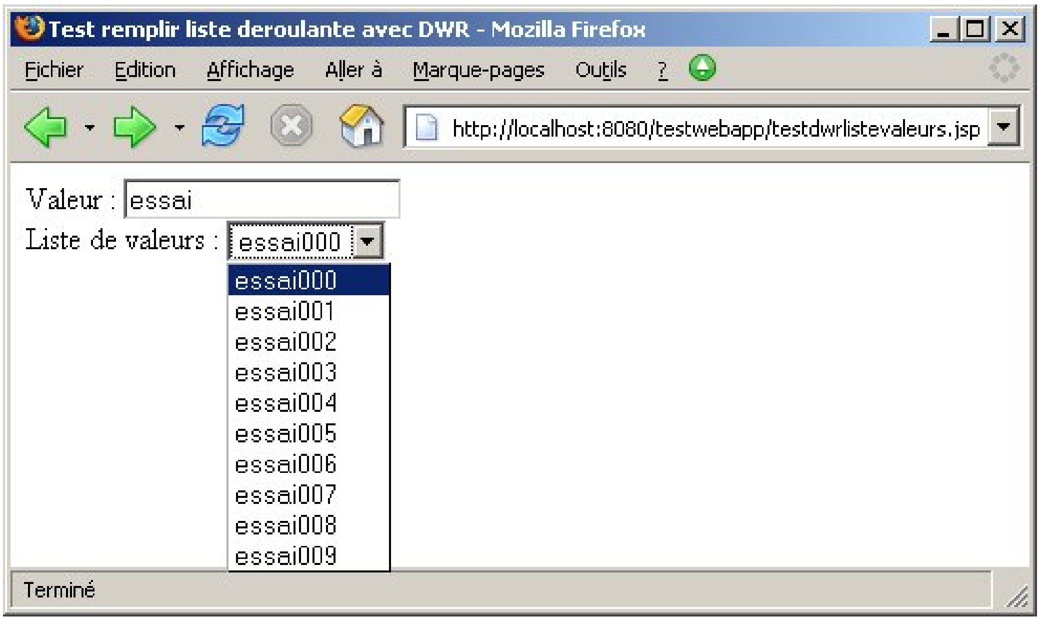

Test de validation de données - Mozilla Firefox
Fichier Edition Affichage Aller à Marque-pages Outils ?
http://localhost:8080/testwebapp/testdwrvalidation.jsp
Valeur : test
invalide
Terminé

Exemple :

```
  public String validerValeur(String valeur) {

    String resultat = "invalide";

    if ((valeur != null) && valeur.startsWith("X")) {
      resultat = "valide";
    }

    return resultat;
  }
```

88.4.1.7. Un exemple pour remplir dynamiquement une liste déroulante

Cet exemple va remplir dynamiquement le contenu d'une liste déroulante en fonction de la valeur d'une zone de saisie.


Test remplir liste deroulante avec DWR - Mozilla Firefox
Fichier Edition Affichage Aller à Marque-pages Outils ?
http://localhost:8080/testwebapp/testdwrlistevaleurs.jsp
Valeur : essai
Liste de valeurs : essai000
essai000
essai001
essai002
essai003
essai004
essai005
essai006
essai007
essai008
essai009
Terminé

Côté serveur la méthode getListeValeurs() du bean est appelée pour obtenir les valeurs de la liste déroulante. Elle attend en paramètre une chaîne de caractères et renvoie un tableau de chaînes de caractères.

Exemple :

```
package fr.jmdoudoux.dej.ajax.dwr;

public class TestDWR {

  public String[] getListeValeurs(String valeur) {
    String[] resultat = new String[10];

    for(int i = 0 ; i <10;i++ ) {
      resultat[i] = valeur+"00"+i;
    }
    return resultat;
  }

}
```

La page de l'application est composée d'une zone de saisie et d'une liste déroulante.

Exemple :

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Test remplir liste deroulante avec DWR</title>
<script type='text/javascript' src='/testwebapp/dwr/interface/TestDWR.js'></script>
<script type='text/javascript' src='/testwebapp/dwr/engine.js'></script>
<script type='text/javascript' src='/testwebapp/dwr/util.js'></script>

<script type='text/javascript'>
<!--
function rafraichirListeValeurs() {
  TestDWR.getListeValeurs(remplirListeValeurs, $("valeur").value);
}

function remplirListeValeurs(data) {
  DWRUtil.removeAllOptions("listevaleurs");
  DWRUtil.addOptions("listevaleurs", data);
  DWRUtil._selectListItem($("listevaleurs"),$("listevaleurs").options[0].value);
}

function init() {
  DWRUtil.useLoadingMessage();
  rafraichirListeValeurs();
}
-->
</script>
</head>
<body onload="init();">

<p>Valeur : <input type="text" id="valeur"
        onblur="rafraichirListeValeurs();" /><br />
Liste de valeurs : <select id="listevaleurs" style="vertical-align:top;"></select>
</p>

</body>
</html>
```

La fonction init() se charge d'initialiser le contenu de la liste déroulante au chargement de la page.

La fonction rafraichirListeValeurs() est appelée dès que la zone de saisie perd le focus. Elle utilise la fonction JavaScript TestDWR.getListeValeurs() générée par DWR pour appeler la méthode du même nom du bean. Les deux paramètres fournis à cette fonction permettent d'une part de préciser que c'est la fonction remplirListeValeurs() qui fait office de fonction de callback et d'autre part de fournir la valeur de la zone de saisie en paramètre de l'appel de la méthode getListeValeurs() du bean.

La fonction remplirListeValeurs() se charge de vider la liste déroulante, de remplir son contenu avec les données reçues en réponse du serveur (elles sont passées en paramètre de la fonction) et de sélectionner le premier élément de la liste. Pour ces trois actions, trois fonctions issues de la bibliothèque util.js de DWR sont utilisées.

La fonction addOptions() utilise les données passées en paramètre pour remplir la liste.

88.4.1.8. Un exemple pour afficher dynamiquement des informations

L'exemple de cette section va permettre d'afficher dynamiquement les données d'une personne sélectionnée. L'exemple est volontairement simpliste (la liste déroulante des personnes est en dur et les données de la personne sont calculées plutôt qu'extraites d'une base de données). Le but principal de cet exemple est de montrer la facilité d'utilisation des beans mappés par DWR dans le code JavaScript.

Le bean utilisé encapsule les données d'une personne

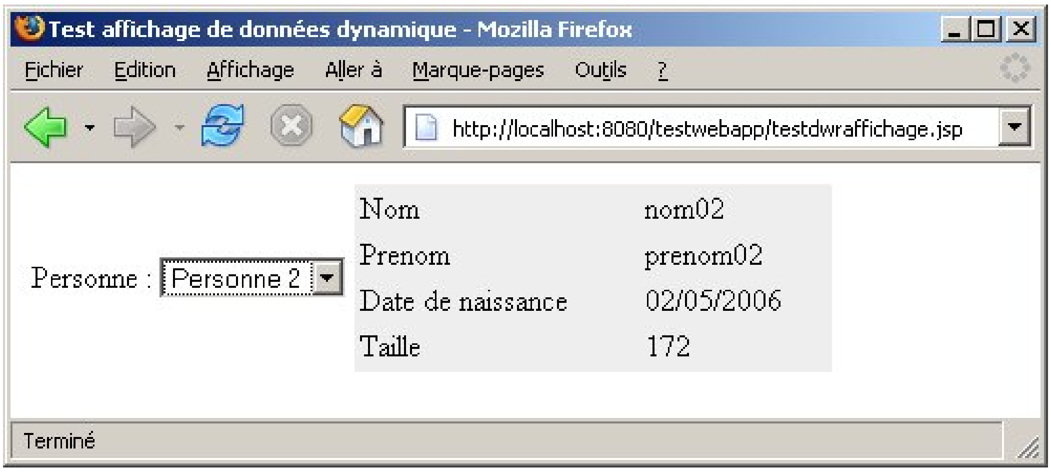
Exemple :

```
package fr.jmdoudoux.dej.ajax.dwr;

import java.util.Date;

public class Personne {
  private String nom;
  private String prenom;
  private String dateNaissance;
  private int taille;

  public Personne() {
    super();
  }

  public Personne(String nom, String prenom, String dateNaissance, int taille) {
    super();
    this.nom = nom;
    this.prenom = prenom;
    this.dateNaissance = dateNaissance;
    this.taille = taille;
  }

  public String getDateNaissance() {
    return dateNaissance;
  }

  public void setDateNaissance(String dateNaissance) {
    this.dateNaissance = dateNaissance;
  }

  public String getNom() {
    return nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

  public String getPrenom() {
    return prenom;
  }

  public void setPrenom(String prenom) {
    this.prenom = prenom;
  }

  public int getTaille() {
    return taille;
  }

  public void setTaille(int taille) {
    this.taille = taille;
  }

}
```

La page est composée d'une liste déroulante de personnes. Lorsqu'une personne est sélectionnée, les données de cette personne sont demandées au serveur et sont affichées.

Exemple :

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Test affichage de données dynamique</title>
<script type='text/javascript' src='/testwebapp/dwr/interface/TestDWR.js'></script>
<script type='text/javascript' src='/testwebapp/dwr/engine.js'></script>
<script type='text/javascript' src='/testwebapp/dwr/util.js'></script>

<script type='text/javascript'>
<!--
function rafraichir() {
  TestDWR.getPersonne(afficherPersonne, $("personnes").value);
}

function afficherPersonne(data) {
  DWRUtil.setValue("nomPersonne",data.nom);
  DWRUtil.setValue("prenomPersonne",data.prenom);
  DWRUtil.setValue("datenaissPersonne",data.dateNaissance);
  DWRUtil.setValue("taillePersonne",data.taille);
}

function init() {
  DWRUtil.useLoadingMessage();
}
-->
</script>
</head>
<body onload="init();">

<table>
        <tr>
                <td>Personne :</td>
                <td nowrap><select id="personnes" name="personnes"
                        onchange="rafraichir();">
                        <option value="1">Personne 1</option>
                        <option value="2">Personne 2</option>
                        <option value="3">Personne 3</option>
                        <option value="4">Personne 4</option>
                        </select>
                </td>
                <td>
                <div id="informationPersonne">
                <table bgcolor="#eeeeee" width="250">
                <tr><td>Nom</td><td><span id="nomPersonne"></span></td></tr>
                <tr><td>Prenom</td><td><span id="prenomPersonne"></span></td></tr>
                <tr><td>Date de naissance</td><td><span id="datenaissPersonne"></span></td></tr>
                <tr><td>Taille</td><td><span id="taillePersonne"></span></td></tr>
                </table>
                </div>
                </td>
        </tr>
</table>

</body>
</html>
```


Test affichage de données dynamique - Mozilla Firefox
Fichier Edition Affichage Aller à Marque-pages Outils ?
Terminé

http://localhost:8080/testwebapp/testdwraffichage.jsp

Personne : Personne 2

Nom	nom02
Prenom	prenom02
Date de naissance	02/05/2006
Taille	172

Exemple : le source de la méthode du bean qui recherche les données de la personne

```
public Personne getPersonne(String id) {
  int valeur = Integer.parseInt(id);
  if (valeur < 10) {
    id = "0"+id;
  }
  Personne resultat = new Personne("nom"+id,"prenom"+id,id+"/05/2006",170+valeur);
  return resultat;
}
```

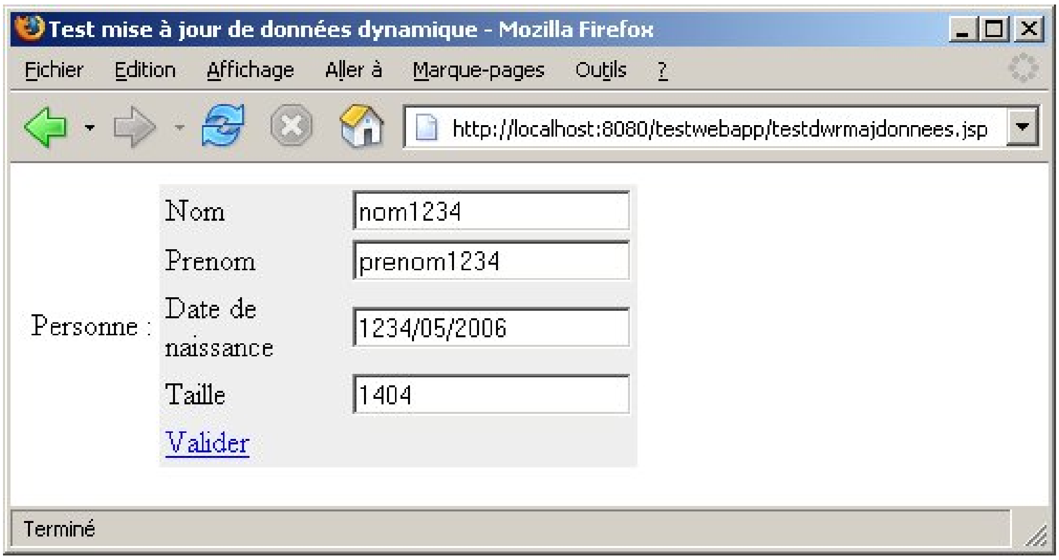
Dans le fichier de configuration dwr.xml, un convertisseur de type bean doit être déclaré pour le bean de type Personne

Exemple :

```
<allow>
  <create creator="new" javascript="TestDWR">
    <param name="class" value="fr.jmdoudoux.dej.ajax.dwr.TestDWR"/>
  </create>

  <convert converter="bean" match="fr.jmdoudoux.dej.ajax.dwr.Personne"/>
</allow>
```

88.4.1.9. Un exemple pour mettre à jour des données

Cet exemple va permettre de modifier les données d'une personne.

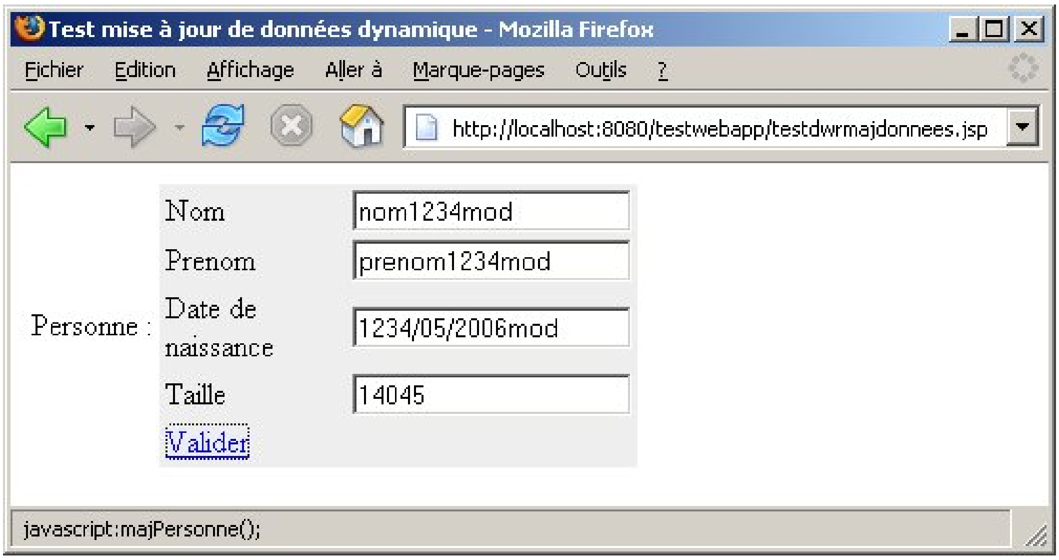

Test mise à jour de données dynamique - Mozilla Firefox
Fichier Edition Affichage Aller à Marque-pages Outils ?
Valider
Terminé

http://localhost:8080/testwebapp/testdwrmajdonnees.jsp

Personne :

Nom	nom1234
Prenom	prenom1234
Date de naissance	1234/05/2006
Taille	1404

Il suffit de modifier les données et de cliquer sur le bouton valider


Test mise à jour de données dynamique - Mozilla Firefox
Fichier Edition Affichage Aller à Marque-pages Outils ?
http://localhost:8080/testwebapp/testdwrmajdonnees.jsp

Personne : Nom nom1234mod
Prenom prenom1234mod
Date de naissance 1234/05/2006mod
Taille 14045
Valider

javascript:majPersonne();

Les données sont envoyées sur le serveur.

Exemple :

```
INFO: Exec[0]: TestDWR.setPersonne()
nom=nom1234mod
prenom=prenom1234mod
datenaiss=1234/05/2006mod
taille14045
```

Exemple : le source de la méthode du bean qui recherche les données de la personne

```
  public void setPersonne(Personne personne)
  {
    System.out.println("nom="+personne.getNom());
    System.out.println("prenom="+personne.getPrenom());
    System.out.println("datenaiss="+personne.getDateNaissance());
    System.out.println("taille"+personne.getTaille());
      // code pour rendre persistant l'objet fourni en paramètre
  }
```

Cette méthode affiche simplement les données reçues. Dans un contexte réel, elle assurerait les traitements pour rendre persistantes leurs modifications.

La page de l'application est la suivante.

Exemple :

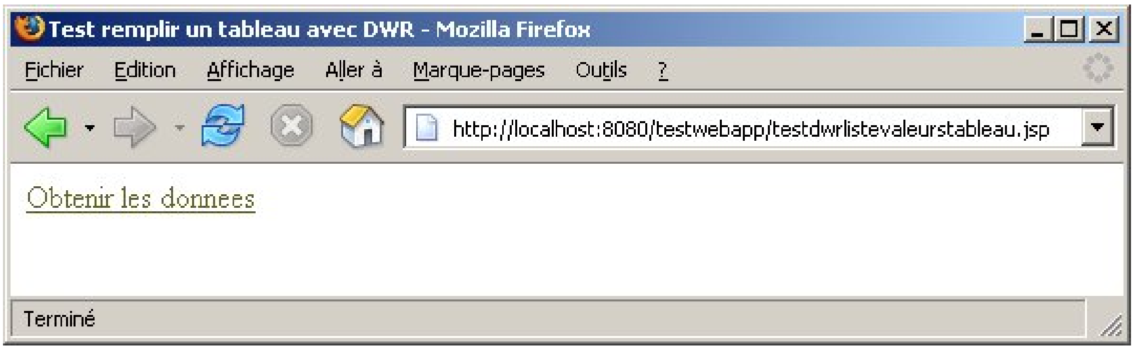
```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
  <title>Test mise à jour de données dynamique</title>
  <script type='text/javascript' src='/testwebapp/dwr/interface/TestDWR.js'></script>
  <script type='text/javascript' src='/testwebapp/dwr/engine.js'></script>
  <script type='text/javascript' src='/testwebapp/dwr/util.js'></script>

  <script type='text/javascript'>
  <!--
  var personne;

  function rafraichir() {
    TestDWR.getPersonne(afficherPersonne, "1234");
  }

  function afficherPersonne(data) {
    personne = data;
    DWRUtil.setValues(data);
  }

  function majPersonne()
  {
    DWRUtil.getValues(personne);
    TestDWR.setPersonne(personne);
  }

  function init() {
    DWRUtil.useLoadingMessage();
    rafraichir();
  }

  -->
  </script>
</head>
<body onload="init();">

<table>
  <tr>
    <td>Personne :</td>
    <td>
    <div id="informationPersonne">
    <table bgcolor="#eeeeee" width="250">
    <tr><td>Nom</td><td><input type="text" id="nom"></td></tr>
    <tr><td>Prenom</td><td><input type="text" id="prenom"></td></tr>
    <tr><td>Date de naissance</td><td><input type="text" id="dateNaissance"></td></tr>
    <tr><td>Taille</td><td><input type="text" id="taille"></td></tr>
    <tr><td colspan="2"><a href="javascript:majPersonne();">Valider</a></td></tr>
    </table>
    </div>
    </td>
        </tr>
</table>

</body>
</html>
```

Cet exemple utilise les fonctions getValues() et setValues() qui mappent automatiquement les propriétés d'un objet avec les objets de l'arbre DOM dont l'id correspond.

Remarque : il est important que l'objet personne qui encapsule les données de la personne soit correctement initialisé, ce qui est fait au chargement des données de la personne.

88.4.1.10. Un exemple pour remplir dynamiquement un tableau de données

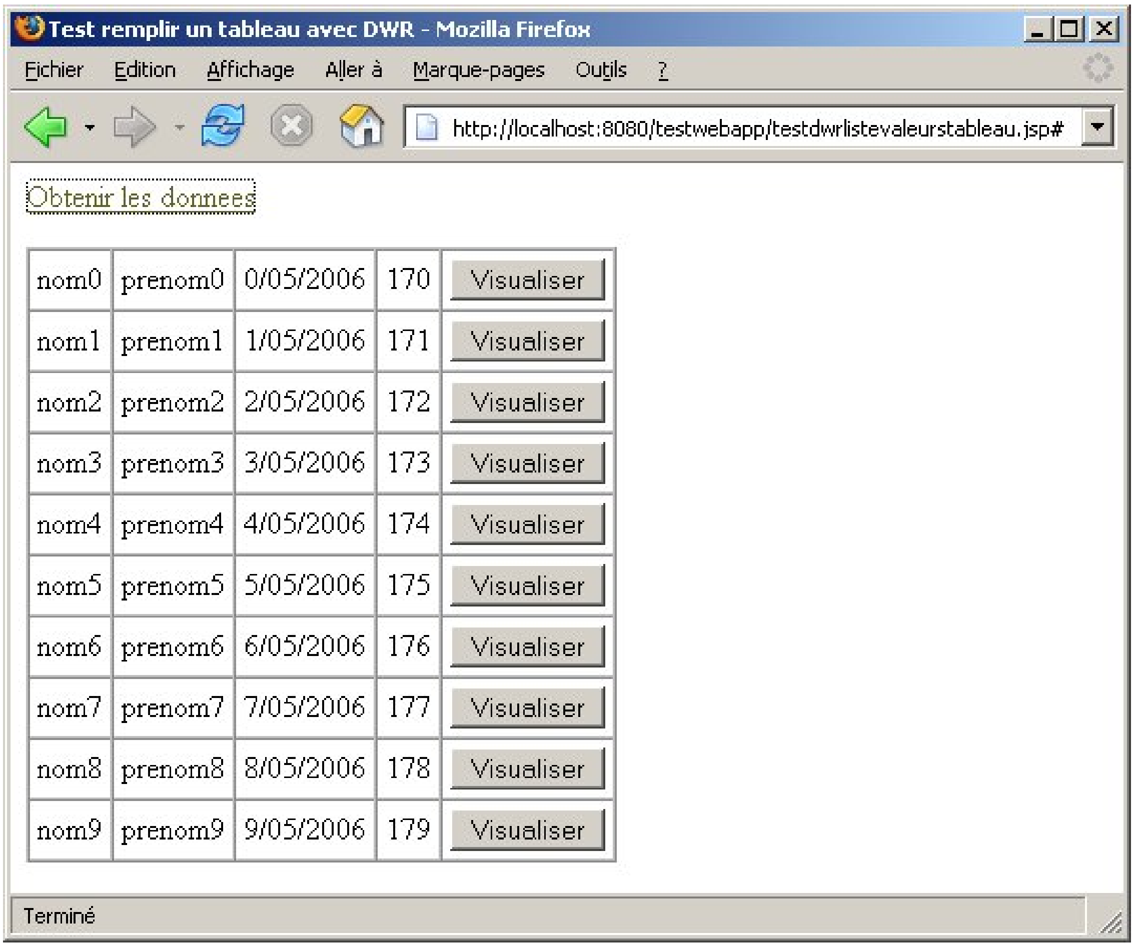
Cet exemple va remplir dynamiquement le contenu d'un tableau avec une collection d'objets.


Test remplir un tableau avec DWR - Mozilla Firefox
Fichier
Edition
Affichage
Aller à
Marque-pages
Outils
?
http://localhost:8080/testwebapp/testdwrlistevaleurstableau.jsp

Obtenir les donnees

Terminé

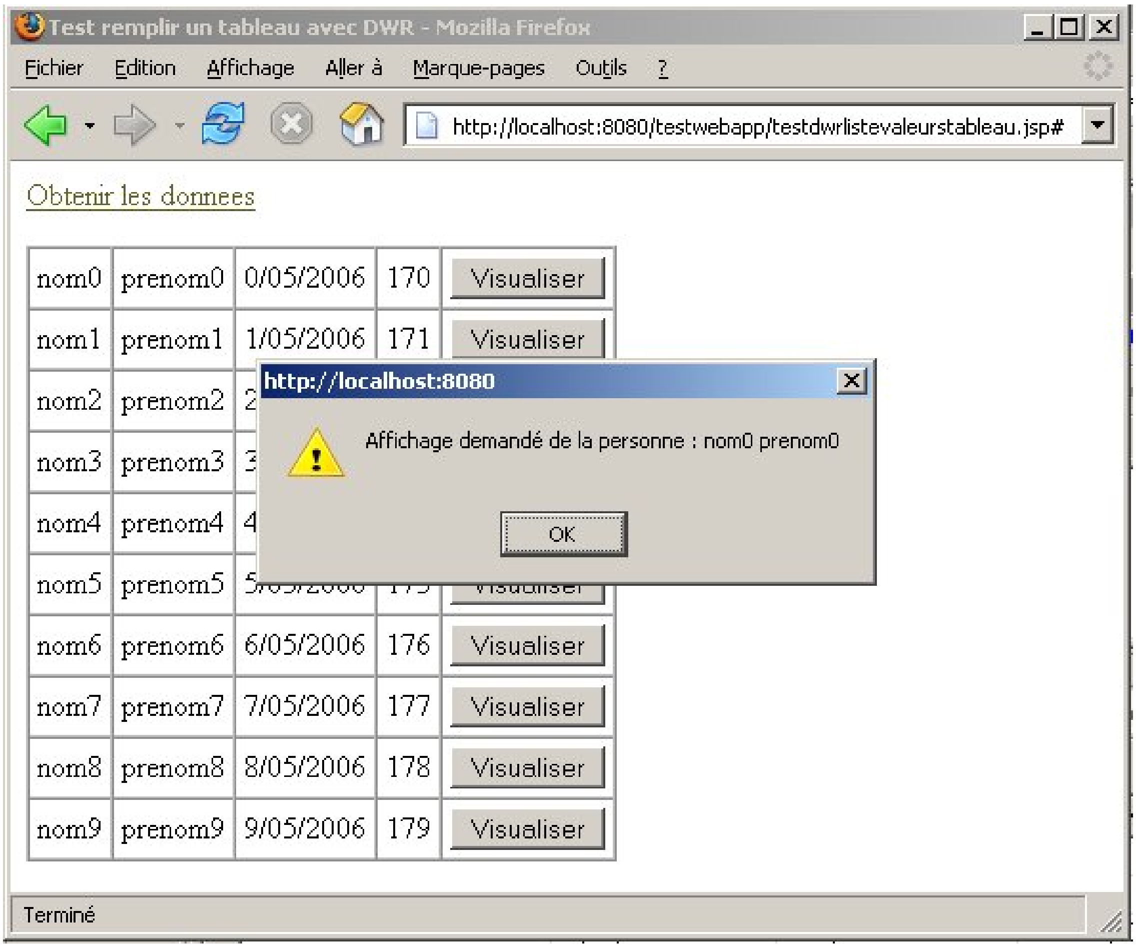
Un clic sur le lien permet d'afficher le tableau avec les données retournées par le serveur.


Test remplir un tableau avec DWR - Mozilla Firefox
Fichier Edition Affichage Aller à Marque-pages Outils ?
http://localhost:8080/testwebapp/testdwrlistevaleurstableau.jsp#
Terminé

Obtenir les donnees

nom0	prenom0	0/05/2006	170	Visualiser
nom1	prenom1	1/05/2006	171	Visualiser
nom2	prenom2	2/05/2006	172	Visualiser
nom3	prenom3	3/05/2006	173	Visualiser
nom4	prenom4	4/05/2006	174	Visualiser
nom5	prenom5	5/05/2006	175	Visualiser
nom6	prenom6	6/05/2006	176	Visualiser
nom7	prenom7	7/05/2006	177	Visualiser
nom8	prenom8	8/05/2006	178	Visualiser
nom9	prenom9	9/05/2006	179	Visualiser

Un clic sur le bouton "visualiser" affiche un message avec le nom et la personne concernée.


Test remplir un tableau avec DWR - Mozilla Firefox
Fichier Edition Affichage Aller à Marque-pages Outils ?
http://localhost:8080/testwebapp/testdwrlistevaleurstableau.jsp#
http://localhost:8080
OK
Terminé

Obtenir les donnees

nom0	prenom0	0/05/2006	170	Visualiser
nom1	prenom1	1/05/2006	171	Visualiser
nom2	prenom2	2		
nom3	prenom3	3		
nom4	prenom4	4		
nom5	prenom5			
nom6	prenom6	6/05/2006	176	Visualiser
nom7	prenom7	7/05/2006	177	Visualiser
nom8	prenom8	8/05/2006	178	Visualiser
nom9	prenom9	9/05/2006	179	Visualiser

Affichage demandé de la personne : nom0 prenom0

Côté serveur la méthode getPersonnes() du bean est appelée pour obtenir la liste des personnes sous la forme d'une collection d'objets de type Personne.

Exemple :

```
  public List getPersonnes() {
    List resultat = new ArrayList();
    Personne personne = null;

    for (int i = 0; i<10 ; i++) {
      personne = new Personne("nom"+i,"prenom"+i,i+"/05/2006",170+i);
      resultat.add(personne);
    }
    return resultat;
  }
```

La page de l'application est composée d'un calque contenant un tableau.

Exemple :

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
  <title>Test remplir un tableau avec DWR</title>
  <script type='text/javascript' src='/testwebapp/dwr/interface/TestDWR.js'></script>
  <script type='text/javascript' src='/testwebapp/dwr/engine.js'></script>
  <script type='text/javascript' src='/testwebapp/dwr/util.js'></script>

  <script type='text/javascript'>
  <!--
  function rafraichirListeValeurs() {
    TestDWR.getPersonnes(remplirListeValeurs);
  }

  function remplirListeValeurs(data) {

    DWRUtil.removeAllRows("tableau");

    if (data.length == 0) {
      alert("");
      $("donnees").style.visibility = "hidden";
    } else {
      DWRUtil.addRows("tableau",data,cellulesFonctions);
      $("donnees").style.visibility = "visible";
    }
  }

  // tableau des fonctions permettant d'assurer le rendu des différentes cellules du tableau
  var cellulesFonctions = [
    function(item) { return item.nom; },
    function(item) { return item.prenom; },
    function(item) { return item.dateNaissance; },
    function(item) { return item.taille; },
    function(item) {
      var btn = document.createElement("button");
      btn.innerHTML = "Visualiser";
      btn.itemId = item.nom+" "+item.prenom;
      btn.onclick = afficherPersonne;
      return btn;
    }
  ];


  function afficherPersonne() {
    alert("Affichage demandé de la personne : "+this.itemId);
  }

  function init() {
    DWRUtil.useLoadingMessage();
  }
  -->
  </script>
</head>
<body onload="init();">

<p><a href="#" onclick="rafraichirListeValeurs()">Obtenir les donnees</a></p>
<div id="donnees">
  <table id="tableau" border="1" cellpadding="4" cellspacing="0"></table>
</div>
</body>
</html>
```

Cet exemple met en oeuvre les fonctions de manipulation de tableaux de la bibliothèque util.js notamment la fonction DWRUtil.addRows("tableau",data,cellulesFonctions) qui permet d'ajouter un ensemble de lignes à un tableau HTML.

Elle attend en paramètre l'id du tableau à modifier, les données à utiliser et un tableau de fonctions qui vont définir le rendu de chaque cellule d'une ligne du tableau. Ces fonctions peuvent simplement retourner la valeur d'une propriété de l'objet courant ou renvoyer des objets plus complexes comme un bouton.

Remarque : pour les boutons générés, il serait préférable d'utiliser des id mieux adaptés comme un suffixe et un identifiant uniqueque de concaténer les noms et prénoms. Ici, le but est de rendre l'exemple le plus possible.

89. GWT (Google Web Toolkit)

Chapitre 89

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

GWT (Google Web Toolkit) est un framework open source de développement d'applications web mettant en oeuvre AJAX et développé par Bruce Johnson et Google.

Mi-2006, Google a diffusé GWT qui est un outil de développement d'applications de type RIA offrant une mise en oeuvre novatrice : le but est de faciliter le développement d'applications web mettant en oeuvre Ajax en faisant abstraction des incompatibilités des principaux navigateurs.

GWT propose de nombreuses fonctionnalités pour développer une application exécutable dans un navigateur et présetnant des comportements similaires à ceux d'une application desktop :

- création d'applications graphiques s'exécutant dans un navigateur
- pas besoin d'écrire du code Javascript sauf pour des besoins très spécifiques comme l'intégration d'une bibliothèque JavaScript existante
- utilisation de CSS pour personnaliser l'apparence
- mise en oeuvre d'Ajax sans manipuler l'arbre DOM de la page mais en utilisant des objets Java
- un ensemble riche de composants (widgets et panels)
- communication avec le serveur grâce à des appels asynchrones en échangeant des objets Java et en utilisant des exceptions pour signifier des problèmes
- internationalisation
- un système de gestion de l'historique sur le navigateur
- un parser XML
- détection des erreurs à la compilation
- ...

L'utilisation de GWT présente plusieurs avantages :

- pas de code JavaScript à écrire
- utilisation de Java comme langage de développement
- une meilleure productivité liée à l'utilisation du seul langage Java (un seul langage à utiliser, mieux connu que d'autres technologies notamment JavaScript, mise en oeuvre d'un débogueur, utilisation d'un IDE Java, ...)
- hormis les styles CSS et la page HTML qui encapsule l'application, il n'y a pas d'utilisation directe de technologies web
- le code généré par GWT supporte les principaux navigateurs
- la prise en main est facile même pour des débutants ce qui lui confert une bonne courbe d'apprentissage

Le site officiel de GWT est à l'url https://www.gwtproject.org/

Ce chapitre contient plusieurs sections :

- La présentation de GWT
- La création d'une application
- Les modes d'exécution
- Les éléments de GWT
- L'interface graphique des applications GWT
- La personnalisation de l'interface

- Les composants (widgets)
- Les panneaux (panels)
- La création d'éléments réutilisables
- Les événements
- JSNI
- La configuration et l'internationalisation
- L'appel de procédures distantes (Remote Procedure Call)
- La manipulation des documents XML
- La gestion de l'historique sur le navigateur
- Les tests unitaires
- Le déploiement d'une application
- Des composants tiers
- Les ressources relatives à GWT

89.1. La présentation de GWT

Le code de l'application est entièrement écrit en Java notamment la partie cliente qui devra s'exécuter dans un navigateur. Ce code Java n'est pas compilé en bytecode mais en JavaScript ce qui permet son exécution dans un navigateur.

Le coeur de GWT est donc composé du compilateur de code Java en JavaScript. L'avantage du code JavaScript produit est qu'il est capable de s'exécuter sur les principaux navigateurs sans adaptation particulière du source Java puisque le compilateur crée un fichier JavaScript optimisé pour chacun de ces navigateurs.

Le code à écrire pour la partie cliente est composé de plusieurs éléments :

- la syntaxe est celle de Java 1.4 (les fonctionnalités de Java 5 sont supportées à partir de la version 1.5 de GWT)
- un sous-ensemble des API de bases du JDK notamment des packages java.lang et java.util (particulièrement les classes qui pourront être compilées en JavaScript. La liste complète de ces classes est consultable à l'url https://www.gwtproject.org/doc/latest/RefJreEmulation.html)
- un ensemble de composants graphiques nommés widgets et de panels qui sont utilisés pour réaliser l'interface graphique

La partie graphique d'une application GWT est composée d'une petite partie en HTML, de CSS et surtout de classes Java dans lesquelles des composants sont utilisés avec des gestionnaires d'événements pour définir l'interface de l'application et les réponses aux actions des utilisateurs.

GWT propose un ensemble assez complet de composants graphiques nommés widgets fournis en standard : l'ensemble des widgets inclut des composants graphiques standards (boutons, zones de saisie de texte, listes déroulantes, ...) mais contient aussi des composants plus riches tels que des panneaux déroulants, des onglets, des arbres, des boîtes de dialogues, ... Il est aussi possible de créer ses propres composants ou d'intégrer des frameworks JavaScript (ext, Dojo, Rialto, Yahoo UIL, ...)

Le code Java pour développer en GWT est très ressemblant à celui à produire pour développer une application graphique utilisant AWT :

- instancier des composants
- ajouter ces composants dans la hiérarchie des composants de la page
- utiliser des gestionnaires d'événements pour répondre aux actions des utilisateurs

Exemple :

```
public class TestBonjour implements EntryPoint {
    public void onModuleLoad() {
        Button bouton = new Button("Saluer", new ClickListener() {
            public void onClick(Widget sender) {
                Window.alert("Bonjour");
            }
        });
        RootPanel.get().add(bouton);
    }
}
```

GWT propose aussi des outils pour assurer la communication avec la partie serveur en se reposant sur AJAX et offre aussi un support de JUnit.

Ce framework propose plusieurs originalités intéressantes :

- développement majoritairement en Java et un peu d'HTML : le code JavaScript est généré par le compilateur GWT
- développée en Java, une application GWT est plus facile à déboguer en utilisant un IDE Java
- GWT fournit de nombreux composants (widgets) et un système de gestion de leur positionnement (Layout)
- l'application GWT gère le support des principaux navigateurs
- ...

Le grand avantage de GWT est que l'application web utilisant Ajax et développée avec ce framework ne nécessite essentiellement que des connaissances en Java : quelques rudiments d'HTML et CSS sont nécessaires pour développer des applications GWT mais aucun code JavaScript n'est à écrire.

Une application GWT peut être écrite avec un des IDE Java : ceci rend l'application facilement débogable avec les fonctionnalités de l'IDE.

La version 1.4 de GWT repose sur Java 1.4.

La version 1.5 de GWT repose sur Java 1.5 et offre un support des fonctionnalités de Java 5 (énumérations, generics, ...)

Une application GWT est contenue dans un module. Un module est un ensemble de classes et un fichier de configuration. Un module possède un point d'entrée (entry point) qui correspond à la classe principale qui sera utilisée au lancement de l'application.

Côté serveur, il est possible d'utiliser toutes les technologies capables de traiter des requêtes http (Java, .Net, PHP, ...). Java est particulièrement bien adapté à cette tâche grâce à ses nombreuses API et frameworks disponibles.

Une application de gestion développée avec GWT est donc composée de classes Java :

- pour la partie IHM : les classes Java sont compilées en JavaScript pour permettre l'exécution de l'application dans un navigateur
- pour la partie serveur : les classes assurent les traitements métiers et la persistance de données.

Remarque : la partie serveur n'a pas d'obligation à être développée en Java. Elle peut être développée avec d'autres plates-formes mais GWT offre des facilités pour l'utilisation de Java

Lors du déploiement, l'application web sera générée à partir du code Java pour produire les codes HTML et JavaScript requis pour la partie cliente.

Une application GWT peut être exécutée dans deux modes :

- mode hôte (hosted mode) : il est utilisé lors de la phase de développement. Dans ce mode, l'application est exécutée sous la forme de bytecode dans une JVM. Ce mode permet donc la mise en oeuvre d'un débogueur pour faciliter la mise au point de l'application. Il utilise une version personnalisée d'un navigateur fourni par GWT en fonction de l'OS (Internet Exploreur sous Windows et Firefox sous Linux) et une machine virtuelle Java qui permet de transformer le code Java et de l'afficher dans le navigateur.
- mode web (web mode) : dans ce mode, le code Java est compilé pour générer les codes HTML et JavaScript de la partie client. L'application peut ainsi être exécutée dans les navigateurs supportés par GWT.

89.1.1. L'installation de GWT

Il faut télécharger GWT à l'url : https://www.gwtproject.org/download.html

La version utilisée dans ce chapitre est la 1.3.3 sous Windows. Le fichier téléchargé se nomme donc gwt-windows-1.3.3.zip. Il faut décompresser le contenu de ce fichier dans un répertoire du système en utilisant un outil gérant le format zip comme l'utilitaire jar fourni avec le JDK.

89.1.2. GWT version 1.6

La version 1.6 de GWT fut publiée en mai 2006. Cette version apporte de nombreuses évolutions notamment :

- une nouvelle structure pour les projets qui facilite le packaging de la partie serveur sous la forme d'une archive war
- un nouvel environnement d'exécution local qui utilise Jetty
- de nouveaux mécanismes de gestion des événements et le support d'évènements natifs
- de nouveaux composants (DatePicket, DateBox, LazyPanel)
- corrections de bugs
- ...

89.1.2.1. La nouvelle structure pour les projets

La structure des répertoires des projets GWT a été adaptée notamment pour respecter le standard des applications web et faciliter la création du packaging de déploiement sous la forme d'une archive war.

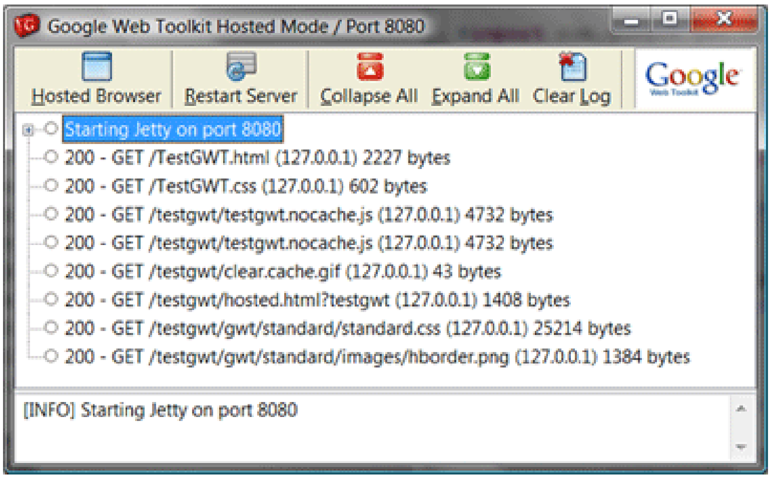
Le répertoire cible de génération des livrables se nomme d'ailleurs /war : ce répertoire contiendra le résultat des compilations mais aussi les ressources statiques nécessaires aux modules de l'application. Ainsi certaines adaptations sont nécessaires par rapport à l'organisation des projets des versions antérieures notamment :

- Le fichier de configuration web.xml de la partie serveur se trouve directement dans le sous-répertoire /war/WEB-INF.
- Toutes les bibliothèques requises par la partie serveur pour traiter les appels GWT-RPC doivent être ajoutées dans le sous-répertoire /war/WEB-INF/lib
- La page HTML qui contient l'application ainsi que les ressources requises doivent être mises dans le sous-répertoire war et non plus dans le sous-répertoire public comme c'était le cas dans les versions précédentes. Il est toujours possible d'inclure des ressources dans les sous répertoires des packages du module mais il faut que celles-ci soient spécifiques au module et manipulées dans le code. Pour accéder à une telle ressource, il est maintenant obligatoire d'utiliser la méthode GWT.getModuleBaseURL() pour obtenir le préfixe de l'url de la ressource.

Deux nouveaux outils sont proposés pour exploiter cette nouvelle structure de projet :

- HostedMode : permet l'exécution en mode hosted en remplacement de GWTShell
- Compiler : permet la compilation du code Java en JavaScript en remplacement de GWTCompiler

L'application GWTShell utilisait un serveur Tomcat embarqué. L'application HostedMode utilise un serveur Jetty embarqué.



```
Starting Jetty on port 8080
200 - GET /TestGWT.html (127.0.0.1) 2227 bytes
200 - GET /TestGWT.css (127.0.0.1) 602 bytes
200 - GET /testgwt/testgwt.nocache.js (127.0.0.1) 4732 bytes
200 - GET /testgwt/testgwt.nocache.js (127.0.0.1) 4732 bytes
200 - GET /testgwt/clear.cache.gif (127.0.0.1) 43 bytes
200 - GET /testgwt/hosted.html?testgwt (127.0.0.1) 1408 bytes
200 - GET /testgwt/gwt/standard/standard.css (127.0.0.1) 25214 bytes
200 - GET /testgwt/gwt/standard/images/hborder.png (127.0.0.1) 1384 bytes
```

```
[INFO] Starting Jetty on port 8080
```

La partie graphique cliente possède un nouveau bouton « Restart Server » qui permet de redémarrer le serveur Jetty : cela permet de prendre en compte des modifications dans la partie serveur sans avoir à arrêter et relancer l'application graphique cliente comme c'était le cas avec GWTShell.

89.1.2.2. Un nouveau système de gestion des événements

Le système de gestion des événements par listener est remplacé par un système de gestion par handler.

Les principales différences entre le deux systèmes sont :

- Les méthodes de type EventHandler ne possèdent qu'un seul paramètre de type GwtEvent. Par exemple, la classe ClickHandler possède la méthode onClick(ClickEvent)
- Chaque EventHandler ne possède qu'une seule méthode : ceci évite d'avoir à écrire des méthodes vides mais peut nécessiter de remplacer un listener par plusieurs handlers selon les besoins

La création de ses propres composants est facilitée car il n'est plus nécessaire de gérer manuellement les listeners. Tous les composants possèdent un objet de type HandlerManager dont le but est de gérer les handlers enregistrés auprès du composant.

La méthode addDomHandler() permet de gérer des événements natifs tels que ClickEvent par exemple : le handler est alors invoqué à l'émission de l'événement.

Exemple :

```
Button bouton = new Button("fermer");

bouton.addClickListener(new ClickListener() {
  @Override
  public void onClick(Widget sender) {
    // traitement du clic sur le bouton

  }
});
```

Exemple :

```
Button bouton = new Button("fermer");

bouton.addClickHandler(new ClickHandler() {
  @Override
  public void onClick(ClickEvent event) {
    // traitement du clic sur le bouton

  }
});
```

Certains handlers existants ont dû être adaptés :

Exemple :

```
final DisclosurePanel panel = new DisclosurePanel("Cliquez pour ouvrir");

panel.addEventHandler(new DisclosureHandler() {
  public void onClose(DisclosureEvent event) {
    panel.getHeaderTextAccessor().setText("Cliquez pour ouvrir");
  }

  public void onOpen(DisclosureEvent event) {
    panel.getHeaderTextAccessor().setText("Cliquez pour fermer");
  }
});

panel.add(new Image("images/logo_java.jpg"));
panel.setWidth("300px");
```

```
    RootPanel.get("app").add(panel);
```

Exemple :

```
    final DisclosurePanel panel = new DisclosurePanel("Cliquez pour ouvrir");

    panel.addOpenHandler(new OpenHandler<DisclosurePanel>() {
      @Override
      public void onOpen(OpenEvent<DisclosurePanel> event) {
        panel.getHeaderTextAccessor().setText("Cliquez pour fermer");
      }
    });

    panel.addCloseHandler(new CloseHandler<DisclosurePanel>() {
      @Override
      public void onClose(CloseEvent<DisclosurePanel> event) {
        panel.getHeaderTextAccessor().setText("Cliquez pour ouvrir");
      }
    });

    panel.add(new Image("images/logo_java.jpg"));
    panel.setWidth("300px");

    RootPanel.get("app").add(panel);
```

89.1.2.3. De nouveaux composants

Les composants DatePicker et DateBox permettent à l'utilisateur de sélectionner une date dans un calendrier.

Le panneau de type LazyPanel permet de retarder la création des objets utiles pour son rendu lorsqu'il sera affiché pour la première fois : ceci peut permettre d'améliorer le temps de démarrage de l'application qui n'est plus obligée d'instancier les composants de tous les éléments de l'application.

89.1.3. GWT version 1.7

GWT 1.7 est une mise à jour mineure qui apporte un meilleur support pour les dernières versions des navigateurs (Internet Explorer 8, Firefox 3.5 et Safari 4) et corrige quelques bugs majeurs. Elle a été publiée en juillet 2009.

Pour un projet en GWT 1.6, il suffit simplement de le recompiler avec la version 1.7, sans modifier le code, pour que l'application soit compatible avec les nouvelles versions des navigateurs supportés.

89.2. La création d'une application

GWT propose plusieurs scripts pour générer des projets GWT composés d'une structure de répertoires et de fichiers fournissant le minimum pour développer un projet.

Pour créer un nouveau projet, il faut créer un nouveau répertoire et utiliser l'application applicationCreator fournie avec GWT. Cet outil permet de créer une petite application d'exemple qui peut facilement servir de base pour le développement d'une application utilisant GWT.

La version Windows de GWT contient un script pour lancer l'application ApplicationCreator. Il suffit d'exécuter ce script avec en paramètre le nom pleinement qualifié de la classe principale de l'application. Le dernier package de cette classe doit se nommer obligatoirement client pour éviter une erreur lors de l'exécution de ApplicationCreator

Résultat :

```
D:\gwt-windows-1.3.3>ApplicationCreator fr.jmdoudoux.dejgwt.monapp
'fr.jmdoudoux.dejgwt.monapp': Please use 'client' as the final package, as in
'com.example.foo.client.MyApp'.
```

```
It isn't technically necessary, but this tool enforces the best practice.
Google Web Toolkit 1.3.3
ApplicationCreator [-eclipse projectName] [-out dir] [-overwrite] [-ignore] className

where
  -eclipse    Creates a debug launch config for the named eclipse project
  -out        The directory to write output files into (defaults to current)
  -overwrite  Overwrite any existing files
  -ignore     Ignore any existing files; do not overwrite
and
  className   The fully-qualified name of the application class to create
```

Par défaut, les fichiers générés le sont dans le répertoire principal. Pour préciser le répertoire à utiliser (celui-ci doit exister), il faut utiliser le paramètre -out.

Résultat :

```
D:\gwt-windows-1.3.3>mkdir MonApp

D:\gwt-windows-1.3.3>ApplicationCreator -out MonApp fr.jmdoudoux.dejgwt.client
.MonApp
Created directory MonApp\src
Created directory MonApp\src\com\jmdoudoux\testgwt
Created directory MonApp\src\com\jmdoudoux\testgwt\client
Created directory MonApp\src\com\jmdoudoux\testgwt\public
Created file MonApp\src\com\jmdoudoux\testgwt\MonApp.gwt.xml
Created file MonApp\src\com\jmdoudoux\testgwt\public\MonApp.html
Created file MonApp\src\com\jmdoudoux\testgwt\client\MonApp.java
Created file MonApp\MonApp-shell.cmd
Created file MonApp\MonApp-compile.cmd
```

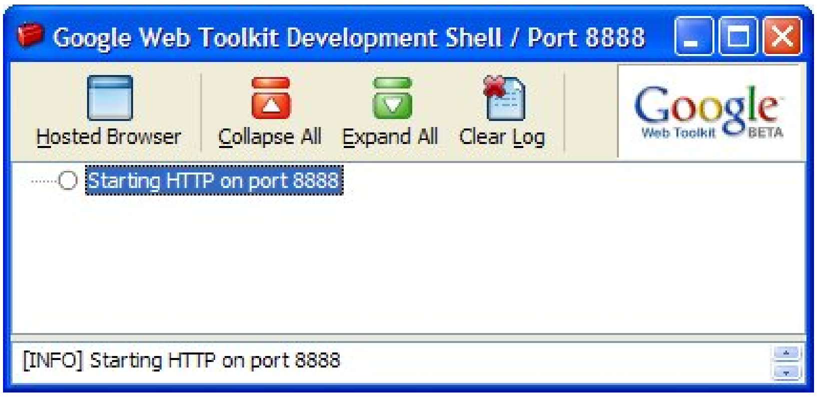
Plusieurs répertoires et fichiers sont créés :

- le répertoire src composé de plusieurs sous-répertoires contient les sources de l'application générée : le sous-répertoire public contient les pages html et le sous-répertoire client contient les sources Java
- le fichier MonApp.gwt.xml contient la configuration de l'application
- le fichier MonApp-shell.cmd permet d'exécuter l'application en mode hôte
- le fichier MonApp-compile.cmd permet de compiler et d'exécuter l'application en mode web

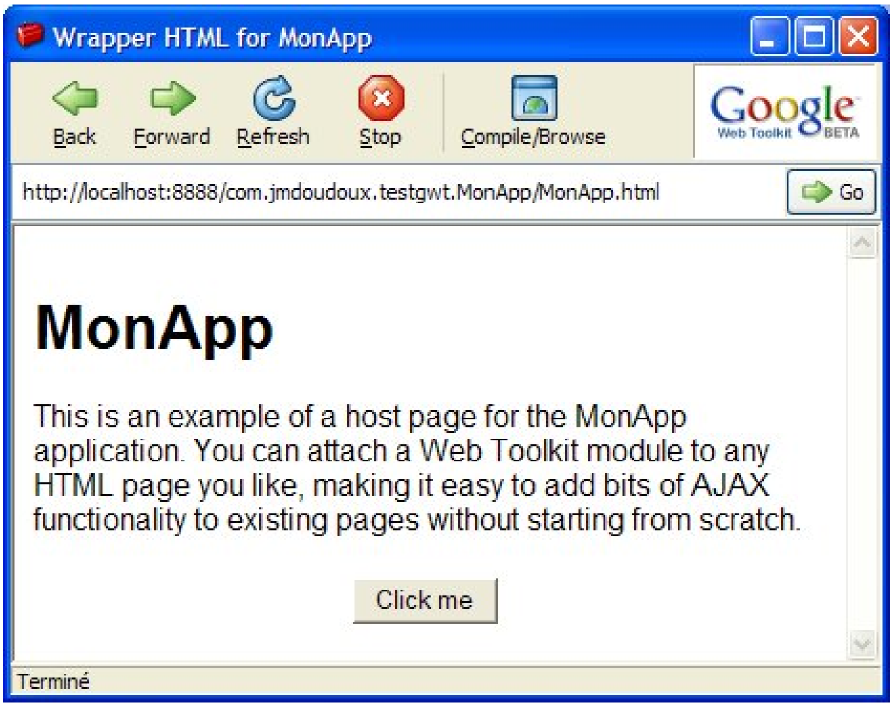
Pour exécuter l'application en mode hôte, il suffit donc de lancer le script MonApp-shell.cmd

Résultat :

```
D:\gwt-windows-1.3.3>cd MonApp

D:\gwt-windows-1.3.3\MonApp>dir
 Le volume dans le lecteur D s'appelle Java
 Le numéro de série du volume est D8C0-0514

 Répertoire de D:\gwt-windows-1.3.3\MonApp

30/07/2007  22:10    <REP>          .
30/07/2007  22:10    <REP>          ..
30/07/2007  22:10               186 MonApp-compile.cmd
30/07/2007  22:10               195 MonApp-shell.cmd
30/07/2007  22:10    <REP>          src
               2 fichier(s)             381 octets
               3 Rép(s)  16 037 978 112 octets libres

D:\gwt-windows-1.3.3\MonApp>MonApp-shell.cmd
```



```
Starting HTTP on port 8888
[INFO] Starting HTTP on port 8888
```



http://localhost:8888/com.jmdoudoux.testgwt.MonApp/MonApp.html

MonApp

This is an example of a host page for the MonApp application. You can attach a Web Toolkit module to any HTML page you like, making it easy to add bits of AJAX functionality to existing pages without starting from scratch.

Pour vérifier la bonne exécution de l'application, il suffit de cliquer sur le bouton "Click me" pour voir apparaître un message.

89.2.1. L'application générée

L'application générée se compose de plusieurs fichiers qui constituent sa base.

Les fichiers sources de l'application sont stockés dans un package qui contient toujours trois sous-répertoires :

Package	Rôle
client	Contient les classes qui composent la partie interface graphique de l'application. Seules les classes de ce package et de ses sous-packages seront compilées en Java
public	Contient les ressources statiques web : pages HTML, images, JavaScript, feuilles de style CSS, ...
server	Contient les classes qui seront exécutées côté serveur

L'application repose sur une page html nommée nom_du_projet.html dans le répertoire public.

Le code de l'application est contenu dans la classe nom_du_projet.java du répertoire client.

Une application GWT est contenue dans un module. La configuration d'un module est le fichier nom_du_projet.gwt.xml.

89.2.1.1. Le fichier MonApp.html

Le fichier MonApp.html du sous-répertoire public contient la structure de la page de l'application.

Le fichier html d'une application GWT est généralement très simple : la page html sert d'enveloppe pour recevoir les différents composants graphiques qui seront ajoutés grâce à du code Java.

Exemple :

```
<html>
<head>
<title>Wrapper HTML for MonApp</title>
<style>
  body,td,a,div,.p{font-family:arial,sans-serif}
  div,td{color:#000000}
  a:link,.w,.w a:link{color:#0000cc}
  a:visited{color:#551a8b}
  a:active{color:#ff0000}
</style>
<meta name='gwt:module' content='fr.jmdoudoux.dejgwt.MonApp'>

</head>
<body>
<script language="javascript" src="gwt.js"></script>
<iframe id="__gwt_historyFrame" style="width:0;height:0;border:0"></iframe>
<h1>MonApp</h1>
<p>This is an example of a host page for the MonApp application. You
can attach a Web Toolkit module to any HTML page you like, making it
easy to add bits of AJAX functionality to existing pages without
starting from scratch.</p>
<table align=center>
  <tr>
    <td id="slot1"></td>
    <td id="slot2"></td>
  </tr>
</table>
</body>
</html>
```

Les deux balises <td> possèdent des identifiants distincts : ils définissent des conteneurs dans lesquels les composants vont être ajoutés. Chaque identifiant sera utilisé dans le code Java pour obtenir une référence sur le conteneur.

Le script JavaScript gwt.js est utilisé pour lancer l'application notamment en exécutant la version de l'application dédiée au navigateur utilisé.

L'iframe est utilisé dans le mécanisme de gestion de l'historique de navigation.

89.2.1.2. Le fichier MonApp.gwt.xml

Ce fichier contient la définition et la configuration du module notamment :

- la classe qui fait office de point d'entrée dans l'application
- les dépendances
- les directives de compilation

C'est un fichier XML dont l'extension est .gwt.xml. Le tag racine est le tag <module>

Le tag <inherits> permet de préciser les fonctionnalités de base qui composeront le module.

Le tag <entry-point> permet de préciser la classe pleinement qualifiée qui est le point d'entrée de l'application : l'attribut class permet de préciser le nom de la classe principale de l'application.

Exemple :

```
<module>

  <!-- Inherit the core Web Toolkit stuff.                   -->
  <inherits name='com.google.gwt.user.User' />

  <!-- Specify the app entry point class.                    -->
  <entry-point class='fr.jmdoudoux.dejgwt.client.MonApp' />

</module>
```

89.2.1.3. Le fichier MonApp.java

La classe MonApp contient le code de l'application avec notamment :

- la définition des composants de l'interface graphique
- la définition des gestionnaires d'événements (listeners ou handlers) pour répondre aux actions de l'utilisateur
- les traitements en réponse aux événements

La méthode onModuleLoad() est le point d'entrée de l'application. Cette méthode contient la définition de l'IHM de l'application.

La mise en oeuvre des gestionnaires d'événements est similaire à celle d'autres framework permettant le développement d'interfaces graphiques tels que AWT, Swing ou SWT. Elle repose sur l'enregistrement de listeners généralement définis sous la forme de classes anonymes.

Exemple :

```
package fr.jmdoudoux.dejgwt.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.ClickListener;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.Widget;

/**
 * Entry point classes define <code>onModuleLoad()</code>.
 */
public class MonApp implements EntryPoint {

  /**
   * This is the entry point method.
   */
  public void onModuleLoad() {
    final Button button = new Button("Click me");
    final Label label = new Label();

    button.addClickListener(new ClickListener() {
      public void onClick(Widget sender) {
        if (label.getText().equals(""))
          label.setText("Hello World!");
        else
          label.setText("");
      }
    });

    // Assume that the host HTML has elements defined whose
    // IDs are "slot1", "slot2".  In a real app, you probably would not want
    // to hard-code IDs.  Instead, you could, for example, search for all
    // elements with a particular CSS class and replace them with widgets.
    //
    RootPanel.get("slot1").add(button);
    RootPanel.get("slot2").add(label);
  }
}
```

Important : les classes de la partie client ne peuvent pas faire référence aux classes de la partie serveur.

89.3. Les modes d'exécution

Comme évoqué en introduction, une application GWT peut être exécutée dans deux modes :

- Le mode hôte (hosted mode) : ce mode est utilisé pour le développement et la mise au point de l'application car il permet la mise en oeuvre d'un débogueur
- Le mode web (web mode) : ce mode est utilisé pour le déploiement et l'exploitation de l'application par les utilisateurs

89.3.1. Le mode hôte (hosted mode)

Dans ce mode, l'application est exécutée de façon hybride sous la forme de code Java exécuté dans un navigateur spécial. Ceci permet notamment l'utilisation du débogueur d'un IDE.

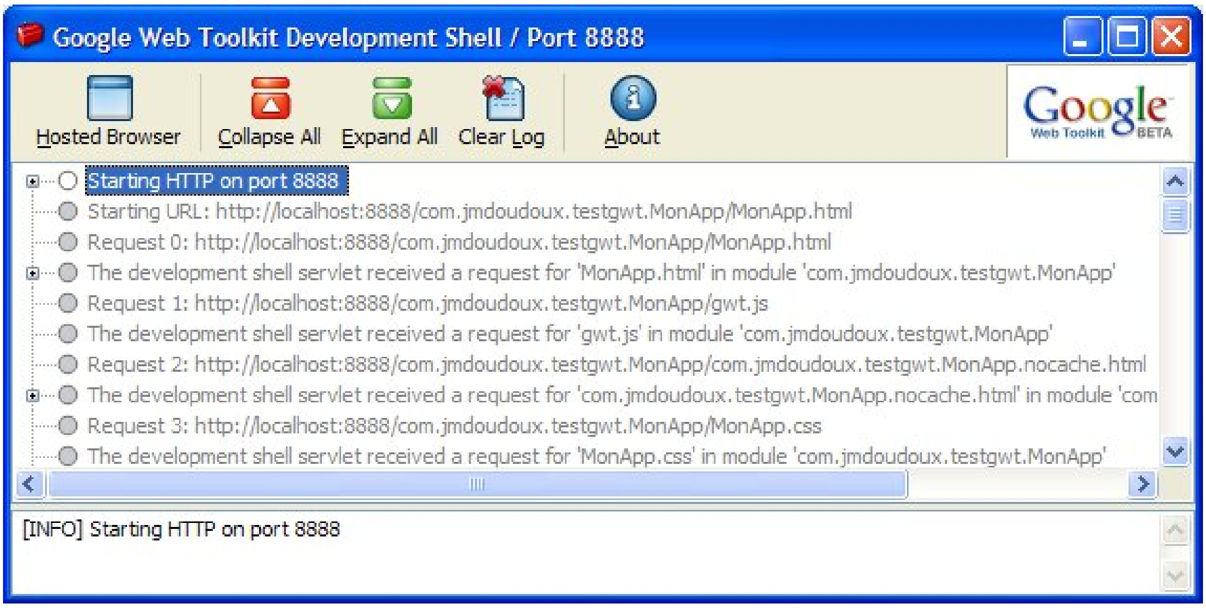
L'environnement d'exécution est composé d'une console, d'un conteneur web (Tomcat ou Jetty selon la version de GWT) et d'un navigateur dédié (Internet Exploreur sous Windows et Firefox sous Linux).

Remarque : le mode hôte n'est disponible que sous Windows et Linux.

Pour exécuter une application dans le mode hôte, il faut exécuter le script dont le nom se compose du nom de l'application et se termine par -shell. Plusieurs options peuvent être fournies à l'environnement d'exécution :

- -noserver
- -out
- -gen
- -logLevel level : permet de préciser niveau de trace. Level peut prendre les valeurs : ERROR, WARN, INFO, TRACE, DEBUG, SPAM, ALL

Exemple avec l'option -logLevel ALL



```
Starting HTTP on port 8888
Starting URL: http://localhost:8888/com.jmdoudoux.testgwt.MonApp/MonApp.html
Request 0: http://localhost:8888/com.jmdoudoux.testgwt.MonApp/MonApp.html
The development shell servlet received a request for 'MonApp.html' in module 'com.jmdoudoux.testgwt.MonApp'
Request 1: http://localhost:8888/com.jmdoudoux.testgwt.MonApp/gwt.js
The development shell servlet received a request for 'gwt.js' in module 'com.jmdoudoux.testgwt.MonApp'
Request 2: http://localhost:8888/com.jmdoudoux.testgwt.MonApp/com.jmdoudoux.testgwt.MonApp.nocache.html
The development shell servlet received a request for 'com.jmdoudoux.testgwt.MonApp.nocache.html' in module 'com
Request 3: http://localhost:8888/com.jmdoudoux.testgwt.MonApp/MonApp.css
The development shell servlet received a request for 'MonApp.css' in module 'com.jmdoudoux.testgwt.MonApp'
```

```
[INFO] Starting HTTP on port 8888
```

Le mode hôte, facilite grandement l'écriture et la mise au point de l'application en permettant :

- l'exécution de l'application
- la modification du code source, sa recompilation
- de relancer l'application simplement en cliquant sur le bouton Refresh

L'environnement d'exécution affiche deux fenêtres :

- la fenêtre "Google Web Toolkit Development Shell / Port 8888" : affiche les messages du serveur et permet d'interagir avec lui
- le navigateur qui affiche l'application

Lorsque l'application est exécutée en mode hôte :

- il n'y a pas besoin de compiler et déployer l'application à chaque modification
- pour tester une modification faite dans le code et compilée en bytecode, il suffit simplement de cliquer sur le bouton de rafraichissement du navigateur : ceci permet de tester rapidement des modifications
- il est possible d'utiliser le débogueur d'un IDE en positionnant des points d'arrêts

89.3.2. Le mode web (web mode)

Dans le mode web, la partie cliente de l'application doit être compilée en JavaScript. Un script de compilation est généré lors de la création de l'application. Le nom de ce script est composé du nom de l'application suivi de « -compile.cmd ».

Le compilateur possède plusieurs options :

- -logLevel
- -treeLogger
- -gen
- -out
- -style : style de lisibilité du code généré : OBFUSCATED (style par défaut), PRETTY ou DETAILED

Le compilateur génère plusieurs fichiers correspondant à chaque navigateur supporté par le compilateur et éventuellement un pour chaque langue mise en oeuvre pour internationaliser l'application. Ceci permet de réduire la taille du fichier JavaScript de l'application car elle ne contient que du code pour le navigateur et la Locale utilisés.

Résultat :

```
D:\gwt-windows-1.3.3\MonAppProjet>MonApp-compile.cmd
Output will be written into D:\gwt-windows-1.3.3\MonAppProjet\www\fr.jmdoudoux
testgwt.MonApp
Copying all files found on public path
Compilation succeeded
```

Le répertoire www est créé : il contient un sous-répertoire qui porte le nom du package principal de l'application. Ce répertoire contient les fichiers générés.

Résultat : le contenu du répertoire de D:\gwt-windows-1.3.3\MonAppProjet\www\fr.jmdoudoux.dejgwt.MonApp

```
D:\gwt-windows-1.3.3\MonAppProjet\www\fr.jmdoudoux.dejgwt.MonApp>dir
 Le volume dans le lecteur D s'appelle Java
 Le numéro de série du volume est D8C0-0514
13/08/2007  23:03    <REP>          .
13/08/2007  23:03    <REP>          ..
13/08/2007  23:03            38 805 0A3A82524C61CDBB6FEA7286E3F85391.cache.html
13/08/2007  23:03               801 0A3A82524C61CDBB6FEA7286E3F85391.cache.xml
13/08/2007  23:03            38 865 4D7E1609F2BC0A4229FFC55F98976B68.cache.html
13/08/2007  23:03               798 4D7E1609F2BC0A4229FFC55F98976B68.cache.xml
13/08/2007  23:03            38 628 921FF51D706A4D67E10268B5DD22D7D7.cache.html
13/08/2007  23:03               801 921FF51D706A4D67E10268B5DD22D7D7.cache.xml
13/08/2007  23:03            38 422 942F11931D582C6DF0166C3EA388BE17.cache.html
13/08/2007  23:03               798 942F11931D582C6DF0166C3EA388BE17.cache.xml
13/08/2007  23:03             2 900 fr.jmdoudoux.dejgwt.MonApp.nocache.html
13/08/2007  23:03            17 336 gwt.js
13/08/2007  23:03               444 history.html
13/08/2007  23:03               501 MonApp.css
13/08/2007  23:03               512 MonApp.html
13/08/2007  23:03                82 tree_closed.gif
13/08/2007  23:03                78 tree_open.gif
```

```
13/08/2007  23:03                61 tree_white.gif
              16 fichier(s)          179 832 octets
```

Les fichiers .cache.html contiennent le code JavaScript pour chaque navigateur. Le nom du fichier est encrypté. Chacun de ces fichiers possède un fichier avec le même nom et l'extension .cache.xml.

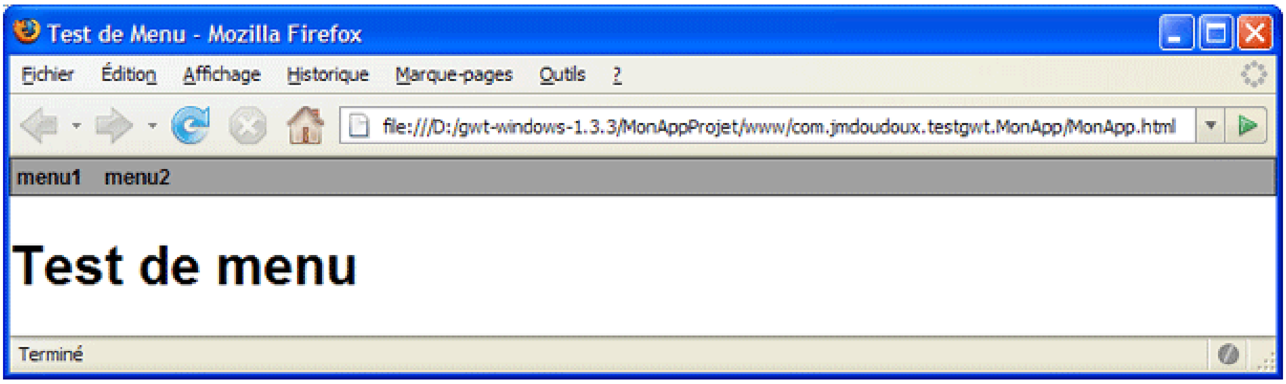
Ces fichiers donnent des informations sur le contenu des fichiers

Exemple : le fichier4D7E1609F2BC0A4229FFC55F98976B68.cache.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<cache-entry>
        <rebind-decision in="com.google.gwt.user.client.ui.impl.TextBoxImpl"
          out="com.google.gwt.user.client.ui.impl.TextBoxImpl"/>
        <rebind-decision in="com.google.gwt.user.client.impl.DOMImpl"
          out="com.google.gwt.user.client.impl.DOMImplMozilla"/>
        <rebind-decision in="com.google.gwt.user.client.impl.HistoryImpl"
          out="com.google.gwt.user.client.impl.HistoryImplStandard"/>
        <rebind-decision in="com.google.gwt.user.client.ui.impl.FormPanelImpl"
          out="com.google.gwt.user.client.ui.impl.FormPanelImpl"/>
        <rebind-decision in="fr.jmdoudoux.dejgwt.client.MonApp"
          out="fr.jmdoudoux.dejgwt.client.MonApp"/>
        <rebind-decision in="com.google.gwt.user.client.ui.impl.PopupImpl"
          out="com.google.gwt.user.client.ui.impl.PopupImpl"/>
</cache-entry>
```

Ainsi, le fichier 4D7E1609F2BC0A4229FFC55F98976B68 correspond au code pour le navigateur Mozilla.

L'ouverture du fichier MonApp.html dans un navigateur lance l'application


Test de Menu - Mozilla Firefox
Fichier
Édition
Affichage
Historique
Marque-pages
Outils
?
file:///D:/gwt-windows-1.3.3/MonAppProjet/www/com.jmdoudoux.testgwt.MonApp/MonApp.html
menu1
menu2
Test de menu
Terminé

Pour diffuser l'application qui ne contient pas de partie serveur, il suffit de copier les fichiers générés, hormis les fichiers .xml, dans un serveur web.

Pour une application qui contient une partie serveur, il faut packager l'application dans un war. Cette archive doit contenir :

- la partie cliente : les fichiers HTML et JavaScript générés ainsi que les ressources statiques (CSS, images, ...)
- la partie serveur : les fichiers .class, le fichier web.xml, les bibliothèques requises (gwt-servlet.jar, ...)

89.4. Les éléments de GWT

GWT se compose de plusieurs éléments :

- le compilateur qui compile du code Java en code JavaScript
- JSNI qui permet l'utilisation de code JavaScript dans le code Java
- JRE Emulation Library qui est un sous-ensemble des classes de base de Java

- une API qui fournit de nombreuses fonctionnalités : composants graphiques pour IHM, appels RPC vers un serveur, gestion de l'historique de navigation, parseur de documents XML, tests unitaires avec JUnit, ...

89.4.1. Le compilateur

Le compilateur GWT de code Java en JavaScript est encapsulé dans la classe com.google.gwt.dev.GWTCompiler.

Le compilateur traite l'entry point pour chacune de ses dépendances. Le compilateur utilise les fichiers sources mais n'utilise pas les fichiers .class.

Le compilateur des versions antérieures à la version 1.5 de GWT ne supporte que du code source respectant la syntaxe de Java 1.4 : les fonctionnalités de Java 5 ne sont donc pas supportées avant la version 1.5 de GWT. Cette restriction n'est valable que pour le code de la partie cliente qui sera transformé en JavaScript. La partie serveur n'est pas concernée par cette restriction puisqu'elle est compilée en bytecode pour être exécutée dans la JVM du serveur.

Le compilateur génère un fichier JavaScript par navigateur et par langage si l'internationalisation est utilisée dans l'application. Le fichier pour le navigateur concerné sera chargé au lancement de l'application. L'intérêt majeur est de limiter le code JavaScript au navigateur utilisé : ceci évite d'avoir à gérer de nombreuses opérations de tests sur le navigateur comme cela est fréquent dans le code JavaScript.

Cette fonctionnalité est aussi mise en oeuvre pour chaque langue utilisée pour internationaliser l'application. Le code JavaScript ne contient que les libellés pour la langue du fichier.

Le compilateur peut mettre en oeuvre des techniques d'obfuscation du code JavaScript généré afin de le protéger et surtout de réduire sa taille.

Au final, le code contenu dans chaque fichier généré est le plus réduit possible.

89.4.2. JRE Emulation Library

Pour utiliser certaines classes de la bibliothèque de base de Java, GWT propose le JRE Emulation Library qui contient les classes fréquemment utilisées dans les applications. Ces classes sont un sous-ensemble de la bibliothèque correspondant à celles qui peuvent être transformées en JavaScript par le compilateur.

Les classes du package java.lang incluses dans le JRE Emulation Library sont :

Boolean	Byte	Character
Class	Double	Float
Integer	Long	Math
Number	Object	Short
String	StringBuffer	System
Throwable	Error	Exception

Il existe aussi des différences entre leur utilisation dans une JVM et dans le JRE Emulation Library. Ces différences sont essentiellement imposées par la conversion du code en JavaScript.

Certaines fonctionnalités de ces classes sont ainsi différentes de leurs homologues de la bibliothèque Java, par exemple :

- System.out et System.err sont utilisables dans le mode hosted mais n'ont aucun effet dans le mode web
- les expressions régulières utilisées dans certaines méthodes la classe String (replaceAll(), replaceFirst()) sont différentes
- ...

Attention : JavaScript ne propose pas de support pour les entiers sur 64 bits représentés par une variable de type long en Java. Le compilateur transforme les types long en double. Le fonctionnement de l'application peut donc être différent dans le mode host et web.

La méthode getStackTrace() de la classe Throwable n'est pas utilisable dans le mode web.

Les assertions (mot clé assert) sont ignorées par le compilateur.

JavaScript n'est pas multithread : tout ce qui concerne le multithreading dans le langage Java est donc inutilisable et ignoré par le compilateur.

L'API reflexion permettant une utilisation dynamique des objets n'est pas utilisable. L'API GWT propose uniquement la méthode GWT.getTypeName() : elle renvoie une chaîne de caractères qui correspond au type de l'objet fourni en paramètre.

JavaScript ne propose pas le support pour la finalisation des objets lors de leur traitement par le garbage collector.

JavaScript ne propose pas le support d'une précision constante dans les calculs en virgule flottante. Il n'est donc pas recommandé d'effectuer de tels calculs dans la partie cliente. Des calculs en virgule flottante peuvent être réalisés mais leur précision n'est pas garantie.

La sérialisation proposée par Java n'est pas supportée par GWT qui a son propre mécanisme pour les appels de type RPC vers le serveur.

Pour des raisons de performance, il n'est pas recommandé d'utiliser des objets de type Long, Float ou Double comme clé pour des objets de type Map.

Les classes du package java.lang incluses dans le JRE Emulation Library sont :

AbstractCollection	AbstractList	AbstractMap
AbstractSet	ArrayList	Arrays
Collections	Date	HashMap
Stack	Vector	Collection
Comparator	EventListener	Iterator
List	Map	RandomAccess
Set		

89.4.3. Les modules

La classe qui est le point d'entrée d'un module doit implémenter l'interface EntryPoint. Cette interface définit la méthode void onLoadModule().

Un module contient un fichier descripteur au format XML. Son nom est composé du nom du module suivi de .gwt.xml

Il permet de préciser :

- Les autres modules utilisés
- Le nom de la classe qui sert de point d'entrée
- Les chemins des fichiers sources qui doivent être compilés en JavaScript
- Les chemins pour trouver les ressources publiques (CSS, images, javasript, ...)

Ce fichier est stocké dans le répertoire contenant la classe qui sert de point d'entrée au module.

Le tag <module> est le tag racine.

Le tag <inherits> permet de préciser un autre module qui sera utilisé. L'attribut name permet de préciser le nom du module.

Le tag <source> permet de préciser le répertoire qui contient des sources à compiler grâce à l'attribut path.

Le tag <stylesheet> permet de préciser une feuille de style CSS. L'attribut src permet de préciser le nom du fichier CSS.

Le tag <servlet> permet de définir une servlet qui sera utilisée pour les communications de type RPC avec le serveur en mode hosted. L'attribut path permet de préciser l'uri de la servlet. L'attribut class permet de préciser le nom pleinement qualifié de la classe qui encapsule la servlet.

89.4.4. Les limitations

Le code de l'application est compilée en JavaScript : seules les fonctionnalités compilables en JavaScript peuvent être utilisées. Ainsi par exemple, il n'est pas possible d'utiliser le type primitif long puisque JavaScript ne supporte pas le 64 bits. Cependant le code se compile parfaitement puisque chaque variable de type long est convertie en type double ce qui peut provoquer des effets de bord.

JavaScript n'est pas multithread : il faut en tenir compte lors du développement de l'application.

89.5. L'interface graphique des applications GWT

Pour la partie graphique de l'application, GWT propose un ensemble de composants de deux types :

- widgets : ce sont des contrôles utilisateurs soit de base (bouton, zone de texte, case à cocher, bouton radio, ...) soit plus riches en fonctionnalités (barre de menu, onglet, treeview, ...)
- panels : ces composants se chargent d'assurer la disposition des composants qui leurs sont rattachés à l'image des layouts manager de Swing. Certains panels proposent aussi de l'interactivité avec l'utilisateur.

En plus de ces composants proposés en standard par GWT, il est possible de développer ses propres composants.

Il existe plusieurs projets open source qui développent d'autres composants (calendrier, grille, ...) ou des composants qui encapsulent des bibliothèques JavaScript existantes (Scriptaculous, Google Search et Map, ...).

Les composants possèdent une double représentation :

- en Java, lors de l'écriture du code et de l'exécution de l'application dans le mode hôte
- dans l'arbre DOM de la page une fois le code compilé en JavaScript et exécuté dans le mode web

L'organisation des composants n'est pas assurée par des layouts mais par des panneaux qui rendent mieux en HTML. Par exemple :

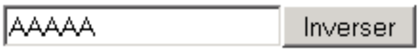
- HorizontalPanel : les composants sont mis les uns à coté des autres de gauche à droite
- FlowPanel : arrange les composants qu'il contient les uns à côté des autres en allant du haut à gauche vers le bas à droite
- AbsolutePanel : permet de préciser les coordonnées des composants
- ...

GWT propose un ensemble complet de composants graphiques (widgets) et panneaux (panels).

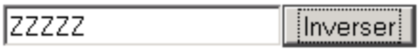
L'état de l'interface graphique est maintenu sur le client dans une application GWT.

Exemple :

```
package fr.jmdoudoux.dej.gwt.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.ClickListener;
import com.google.gwt.user.client.ui.FlowPanel;
import com.google.gwt.user.client.ui.Panel;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.TextBox;
import com.google.gwt.user.client.ui.Widget;

public class MainEntryPoint implements EntryPoint {

  private boolean etat = true;

  public MainEntryPoint() {
  }

  public void onModuleLoad() {
    final TextBox text = new TextBox();
    text.setText("AAAAA");
    final Button button = new Button();
    button.setText("Inverser");
    button.addClickListener(new ClickListener() {

      public void onClick(Widget sender) {
        etat = !etat;
        if (etat) {
          text.setText("AAAAA");
        } else {
          text.setText("ZZZZZ");
        }
      }
    });
    Panel main = new FlowPanel();
    RootPanel.get().add(main);
    main.add(text);
    main.add(button);
  }
}
```



Un clic sur le bouton "Inverser" inverse les lettres affichées



89.6. La personnalisation de l'interface

Comme une application GWT est compilée pour générer une application utilisant le DHTML et JavaScript, la personnalisation de l'interface de l'application repose sur les feuilles de style CSS.

Le rendu des composants d'une application GWT peut donc être assuré par des styles CSS.

La plupart des composants ayant un rendu graphique possèdent une classe de style CSS, par défaut, composée de gwt- suivi du nom du composant (exemple : gwt-Button, gwt-CheckBox, ...).

Il est possible d'utiliser une feuille de style CSS définie dans un fichier stocké dans le sous-répertoire public de l'application.

Exemple : le fichier monstyle.css

Résultat :

```
root {
    display: block;
}

.message
{
    color: blue;
    display: block;
    width: 450px;
    padding: 2px 4px;
    margin-top: 3px;
    text-decoration: none;
    text-align: center;
    font-family: Verdana,Arial,Helvetica,sans-serif;
    font-size: 10px;
    border: 1px solid;
    border-color: black;
    ext-decoration: none;
}

.erreur
{
    color: white;
    display: block;
    width: 450px;
    padding: 2px 4px;
    margin-top: 3px;
    text-decoration: none;
    text-align: center;
    font-family: Verdana,Arial,Helvetica,sans-serif;
    font-size: 10px;
    font-weight: bold;
    border: 1px solid;
    border-color: black;
    ext-decoration: none;
    background-color: red;
}
```

Pour que la feuille de style CSS soit prise en compte par l'application, il faut la déclarer dans le fichier de configuration du module. Cette déclaration se fait à l'aide du tag <stylesheet>. Son attribut src permet de préciser le nom du fichier CSS.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<module>
  <inherits name="com.google.gwt.user.User"/>
  <entry-point class="fr.jmdoudoux.dej.gwt.client.MainEntryPoint"/>
  <stylesheet src="monstyle.css"/>
</module>
```

Chaque composant possède plusieurs méthodes héritées de la classe UIObject et relatives aux styles CSS :

- addStyleName() : permet d'ajouter un style à la liste des styles du composant
- setStylePrimaryName() :
- setStyleName() : permet de forcer le sélecteur de classe du style utilisé en supprimant tous les styles appliqués

Exemple :

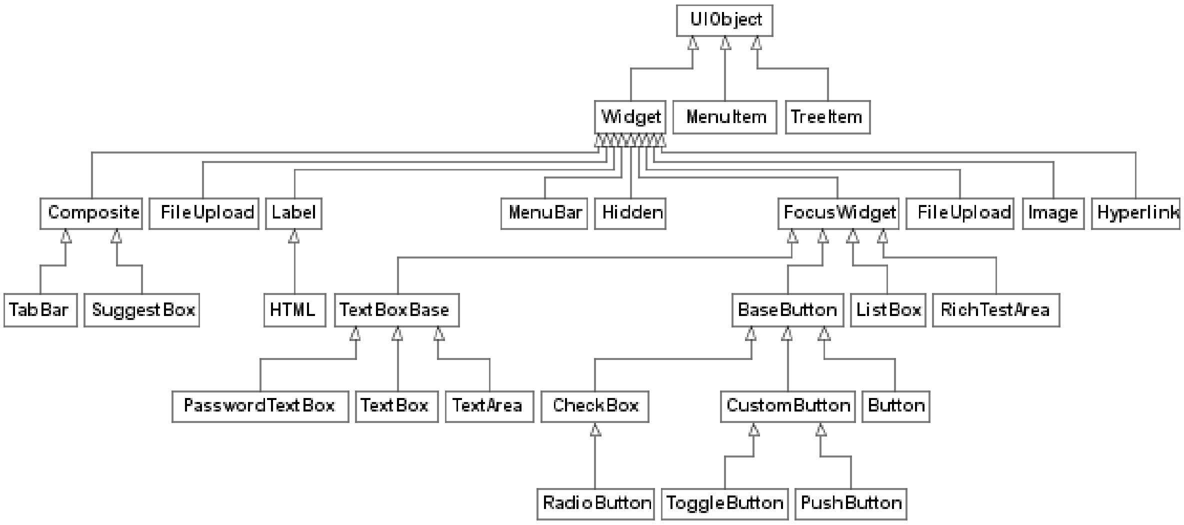
```
...
            public void onSuccess(Object result) {
                lblMessage.setStyleName("message");
                lblMessage.setText((String) result);
            }

            public void onFailure(Throwable caught) {
                lblMessage.setStyleName("erreur");
                lblMessage.setText("Echec de la communication");
            }
...
```

89.7. Les composants (widgets)

GWT propose un ensemble complet de composants graphiques de base pour le développement de l'interface graphique d'une application.

Tous les composants héritent de la classe com.google.gwt.user.client.ui.UIObject.

La classe UIObject est la super classe des classes Widget, MenuItem et TreeItem. La classe Widget est la super classe de la quasi-totalité des composants graphiques de GWT.



Composant	Description	Version de GWT	Classe CSS par défaut
Button	Bouton	1.0	gwt-Button
ButtonBase	Classe mère des boutons		
CheckBox	Case à cocher	1.0	gwt-CheckBox
Composite	Classe qui permet de créer un nouveau composant par assemblage		
DateBox	Zone de saisie de texte qui ouvre un DatePicker	1.6	
DatePicker	Permet de sélectionner une date	1.6	
FileUpload	Elément HTML de type <input type="file">	1.1	
FocusWidget	Classe mère des composants pouvant avoir le focus		
Hidden	Champ de type HIDDEN dans un formulaire HTTP	1.2	
HTML	Contient du code HTML	1.0	gwt-HTML
Hyperlink	Hyperlien	1.1	gwt-Hyperlink
Image	Image	1.1	gwt-Image
Label	Zone de texte		gwt-Label
ListBox	Liste ou liste déroulante		gwt-ListBox
MenuBar	Barre de menus	1.0	gwt-MenuBar
MenuItem	Elément d'une barre de menus	1.0	gwt-MenuItem
PasswordTextBox	Zone de saisie de texte masqué		gwt-PasswordTextBox
RadioButton	Bouton radio		gwt-RadioButton

RichTextArea	Zone de saisie de texte riche	1.4	
SuggestBox	Zone de saisie de texte qui suggère des valeurs selon la saisie	1.4	
TabBar	Une barre d'onglets	1.0	gwt-TabBar gwt-TabBarFirst gwt-TabBarRest gwt-TabBarItem gwt-TabBarItem-selected
TextArea	Zone de saisie de texte multiligne	1.0	gwt-TextArea
TextBox	Zone de saisie de texte monoligne	1.0	gwt-TextBox
TextBoxBase	Classe mère des zones de saisie de texte		
ToogleButton			
Tree	Treeview	1.0	gwt-Tree
TreeItem	Elément d'un composant treeview	1.0	gwt-TreeItem, gwt-TreeItem-selected
UIObject	Classe mère des éléments graphiques		
Widget	Classe mère des composants		

89.7.1. Les composants pour afficher des éléments

GWT propose plusieurs composants graphiques pour afficher du texte ou des images.

89.7.1.1. Le composant Image

La classe Image encapsule une image qui sera affichée. Dans l'arbre Dom, ce composant est un tag <IMG> d'HTML.

Exemple :

```
    Image image = new Image("images/logo_java.jpg");

    RootPanel.get("app").add(image);
```

Il est possible de gérer plusieurs événements émis par ce composant.

La classe Image possède plusieurs listeners utilisables jusqu'à GWT 1.5 : ClickListener, LoadHandler, MouseListener et MouseWheelListener .

Exemple :

```
    Image image = new Image("images/logo_java.jpg");
    final int largeur = image.getWidth();
    final int hauteur = image.getHeight();
    image.setSize(""+(largeur/2), ""+(hauteur/2));
    image.addClickListener(new ClickListener() {
      @Override
      public void onClick(Widget sender) {
        Image img = (Image) sender;
        if (img.getWidth() == largeur) {
          img.setSize(""+(largeur/2), ""+(hauteur/2));
        } else {
          img.setSize(""+largeur, ""+hauteur);
        }
      }
    } );

    RootPanel.get("app").add(image);
```

A partir de GWT 1.6, le composant Image propose plusieurs handlers : DomHandler, ClickHandler, ErrorHandler, LoadHandler, MouseDownHandler, MouseMoveHandler, MouseOutHandler, MouseOverHandler, MouseUpHandler et MouseWheelHandler.

Exemple :

```
  Image image = new Image("images/logo_java.jpg");
  final int largeur = image.getWidth();
  final int hauteur = image.getHeight();
  image.setSize(""+(largeur/2), ""+(hauteur/2));
  image.addClickHandler(new ClickHandler() {
    @Override
    public void onClick(ClickEvent event) {
      Image img = (Image) event.getSource();
      if (img.getWidth() == largeur) {
        img.setSize(""+(largeur/2), ""+(hauteur/2));
      } else {
        img.setSize(""+largeur, ""+hauteur);
      }
    }
  });

  RootPanel.get("app").add(image);
```

89.7.1.2. Le composant Label

La classe Label encapsule un simple texte qui est affiché.

Exemple :

```
  Label label = new Label("un libelle");

  RootPanel.get("app").add(label);
```

Il est possible de gérer plusieurs événements émis par ce composant.

La classe Label possède trois listeners utilisables jusqu'à GWT 1.5 : ClickListener, MouseListener et MouseWheelListener .

Exemple :

```
  Label label = new Label("un libelle");
  ClickListener listener = new ClickListener() {
    @Override
    public void onClick(Widget sender) {
      Window.alert("clic sur le libelle");
    }
  };

  label.addClickListener(listener);

  RootPanel.get("app").add(label);
```

A partir de GWT 1.6, le composant Label propose plusieurs handlers : ClickHandler, MouseDownHandler, MouseMoveHandler, MouseOutHandler, MouseOverHandler, MouseUpHandler et MouseWheelHandler.

Exemple :

```
    final Label label = new Label("un libelle");
    MouseOverHandler mouseOverHandler = new MouseOverHandler() {
      @Override
      public void onMouseOver(MouseOverEvent event) {
        label.setStyleName("label-over");
      }
    };

    MouseOutHandler mouseOutHandler = new MouseOutHandler() {
      @Override
      public void onMouseOut(MouseOutEvent event) {
        label.removeStyleName("label-over");
      }
    };

    label.addMouseOverHandler(mouseOverHandler);
    label.addMouseOutHandler(mouseOutHandler);

    RootPanel.get("app").add(label);
```

89.7.1.3. Le composant DialogBox

La classe DialogBox encapsule une fenêtre de type boîte de dialogue.

Exemple :

```
    private static class MessageInfoBox extends DialogBox {

    private MessageInfoBox(String message) {
      setText("Information");
      final DockPanel panel = new DockPanel();
      panel.setVerticalAlignment(HasAlignment.ALIGN_MIDDLE);
      panel.setHorizontalAlignment(HasAlignment.ALIGN_LEFT);
      panel.setStyleName("alignement-gauche");
      panel.add(new Label(message), DockPanel.CENTER);
      panel.add(new Image("imgages/information.jpg"), DockPanel.WEST);

      SimplePanel panelBouton = new SimplePanel();
      panelBouton.setStyleName("alignement-droite");
      Button boutonOk = new Button("OK");
      boutonOk.setWidth("120px");
      boutonOk.addClickHandler(new ClickHandler() {
        public void onClick(ClickEvent event) {
          MessageInfoBox.this.hide();
        }
      });
      panelBouton.add(boutonOk);
      panel.add(panelBouton, DockPanel.SOUTH);
      setWidget(panel);
    }

    public static void afficher(String message) {
      new MessageInfoBox(message).center();
    }
  }

  public void onModuleLoad() {
    ClickHandler handler = new ClickHandler() {
      @Override
      public void onClick(ClickEvent event) {
        MessageInfoBox.afficher("Ceci est une boite de dialogue d'information");
        }
    };
    Button bouton = new Button("Afficher");
    bouton.addClickHandler(handler);
    RootPanel.get("app").add(bouton);
  }
```

L'instance du panneau n'a pas besoin d'être rattachée au RootPanel ou à tout autre composant.

Les méthodes center() ou show() permettent d'afficher le panneau et la méthode hide() permet de le masquer.

Contrairement à un PopupPanel, il est possible de préciser la taille d'un DialogBox sans avoir besoin d'ajouter de composants mais en utilisant les méthodes setWitdh() et setHeight().

Avant GWT version 1.6, la gestion des événements se faisait avec des listeners. Un seul listener est défini pour ce composant : PopupListener

A partir de GWT version 1.6, la gestion des événements se fait avec des handlers. Plusieurs handlers sont utilisables avec ce composant : DomHandler et CloseHandler.

89.7.2. Les composants cliquables

89.7.2.1. La classe Button

La classe Button encapsule un bouton : elle hérite de la classe ButtonBase.

Le style CSS associé est .gwt-Button

Avant GWT version 1.6, la gestion des événements se faisait avec des listeners. Plusieurs listeners sont utilisables avec ce composant : ClickListener, FocusListener et KeyboardListener.

Exemple :

```
    ClickHandler handler = new ClickHandler() {
      @Override
      public void onClick(ClickEvent event) {
        Window.alert("Bonjour");
      }
    };
    Button bouton= new Button("Afficher");
    bouton.addClickHandler(handler);
    RootPanel.get("app").add(bouton);
```

A partir de GWT version 1.6, la gestion des événements se fait avec des handlers. Plusieurs handlers sont utilisables avec ce composant : BlurHandler, ClickHandler, DomHandler, FocusHandler, KeyDownHandler, KeyPressHandler, KeyUpHandler, MouseDownHandler, MouseMoveHandler, MouseOutHandler, MouseOverHandler, MouseUpHandler et MouseWheelHandler.

Exemple :

```
    ClickListener listener = new ClickListener() {
         @Override
         public void onClick(Widget sender) {
                Window.alert("Bonjour");
        }
    };
    Button bouton = new Button("Afficher", listener);
    RootPanel.get("app").add(bouton);
```

89.7.2.2. La classe PushButton

La suite de cette section sera développée dans une version future de ce document

89.7.2.3. La classe ToggleButton

La suite de cette section sera développée dans une version future de ce document

89.7.2.4. La classe CheckBox

La classe CheckBox encapsule un bouton de type case à cocher : elle hérite de la classe ButtonBase.

Le style CSS associé est .gwt-CheckBox et .gwt-CheckBox-disabled

Avant GWT version 1.6, la gestion des événements se faisait avec des listeners. Plusieurs listeners sont utilisables avec ce composant : ClickListener, FocusListener, KeyboardListener, MouseListener et MouseWheelListener.

Exemple :

```
ClickListener listener = new ClickListener() {
   @Override
   public void onClick(Widget sender) {
    CheckBox cb = (CheckBox) sender;
    if (cb.isChecked()) {
      Window.alert("Bonjour");
    }
  }
};

CheckBox bouton = new CheckBox("Afficher");
bouton.setChecked(false);
bouton.addClickListener(listener);
RootPanel.get("app").add(bouton);
```

A partir de GWT version 1.6, la gestion des événements se fait avec des handlers.Plusieurs handlers sont utilisables avec ce composant : BlurHandler, ClickHandler, DomHandler, FocusHandler, KeyDownHandler, KeyPressHandler, KeyUpHandler, MouseDownHandler, MouseMoveHandler, MouseOutHandler, MouseOverHandler, MouseUpHandler, et MouseWheelHandler

Exemple :

```
ClickHandler handler = new ClickHandler() {
     @Override
     public void onClick(ClickEvent event) {
    CheckBox cb = (CheckBox) event.getSource();
    if (cb.getValue()) {
      Window.alert("Bonjour");
    }
  }
};

CheckBox bouton = new CheckBox("Afficher");
bouton.setValue(false);
bouton.addClickHandler(handler);
RootPanel.get("app").add(bouton);
```

89.7.2.5. La classe RadioButton

La classe RadioButton encapsule un bouton radio : un seul bouton peut être sélectionné parmi ceux d'un même groupe.

Elle hérite de la classe CheckButton.

Le style CSS associé est .gwt-RadioButton

Pour déterminer ou modifier l'état du bouton, il faut utiliser la propriété value (la propriété checked est deprecated).

Exemple :

```
RadioButton rb1 = new RadioButton("valeurs", "Valeur 1");
RadioButton rb2 = new RadioButton("valeurs", "Valeur 2");
RadioButton rb3 = new RadioButton("valeurs", "Valeur 3");

rb2.setValue(true);
VerticalPanel panel = new VerticalPanel();
panel.add(rb1);
panel.add(rb2);
panel.add(rb3);

RootPanel.get("app").add(panel);
```

Avant GWT version 1.6, la gestion des événements se faisait avec des listeners. Plusieurs listeners sont utilisables avec ce composant : ClickListener, FocusListener, KeyboardListener, MouseListener et MouseWheelListener.

Exemple :

```
RadioButton radioButton1 = new RadioButton("valeurs", "Valeur 1");
RadioButton radioButton2 = new RadioButton("valeurs", "Valeur 2");
RadioButton radioButton3 = new RadioButton("valeurs", "Valeur 3");
ClickListener listener = new ClickListener() {
  @Override
  public void onClick(Widget sender) {
     CheckBox checkBox = (CheckBox) sender;
     Window.alert("bouton selectionne = "+checkBox.getText());
  }
};

radioButton1.addClickListener(listener);
radioButton2.addClickListener(listener);
radioButton3.addClickListener(listener);

radioButton2.setValue(true);
VerticalPanel panel = new VerticalPanel();
panel.add(radioButton1);
panel.add(radioButton2);
panel.add(radioButton3);

RootPanel.get("app").add(panel);
```

A partir de GWT version 1.6, la gestion des événements se fait avec des handlers. Plusieurs handlers sont utilisables avec ce composant : BlurHandler, ClickHandler, DomHandler, FocusHandler, KeyDownHandler, KeyPressHandler, KeyUpHandler, MouseDownHandler, MouseMoveHandler, MouseOutHandler, MouseOverHandler, MouseUpHandler et MouseWheelHandler

Exemple :

```
ClickHandler handler = new ClickHandler() {
  @Override
  public void onClick(ClickEvent event) {
    CheckBox cb = (CheckBox) event.getSource();
    if (cb.getValue()) {
      Window.alert("Bonjour");
    }
  }
};
CheckBox bouton = new CheckBox("Afficher");
bouton.setValue(false);
bouton.addClickHandler(handler);
RootPanel.get("app").add(bouton);
```

89.7.2.6. Le composant HyperLink

La suite de cette section sera développée dans une version future de ce document

89.7.3. Les composants de saisie de texte

89.7.3.1. Le composant TextBoxBase

La classe abstraite TextBoxBase encapsule les fonctionnalités de base d'une zone de saisie de texte.

Elle possède plusieurs méthodes pour agir sur le composant dont les principales sont :

Méthodes	Rôle
void cancelKey()	Supprimer un événement clavier reçu par le composant
int getCursorPos()	Obtenir la position courante du curseur dans le texte saisi
String getSelectedText()	Obtenir le texte sélectionné
int getSelectionLength()	Obtenir le nombre de caractères sélectionnés
String getText()	Obtenir le texte
String getValue()	Obtenir la valeur
boolean isReadOnly()	Déterminer si le composant est en lecture seule
void selectAll()	Sélectionner tout le texte
void setCursorPos(int pos)	Positionner le curseur dans le texte
void setReadOnly(boolean readOnly)	Définir si le composant est en lecture seule
void setSelectionRange(int pos, int length)	Définir la portion de texte sélectionnée
void setText(String text)	Initialiser le texte
void setTextAlignment(TextBoxBase.TextAlignConstant align)	Préciser l'alignement du texte
void setValue(String value)	Mettre à jour la valeur du composant sans émettre d'événements
void setValue(String value, boolean fireEvents)	Mettre à jour la valeur du composant

89.7.3.2. Le composant PasswordTextBox

La classe PasswordTextBox encapsule une zone de saisie de texte dont le contenu affiché est masqué. Elle hérite de la classe TextBox.

Exemple :

```
    PasswordTextBox textBox = new PasswordTextBox();
    textBox.setWidth("200px");
    textBox.setMaxLength(10);
    RootPanel.get("app").add(textBox);
    textBox.setFocus(true);
```

Avant GWT version 1.6, la gestion des événements se faisait avec des listeners. Tous les listeners utilisables avec ce composant sont hérités de sa classe mère TextBox.

A partir de GWT version 1.6, la gestion des événements se fait avec des handlers. Tous les handlers utilisables avec ce composant sont hérités de se classe mère TextBox.

89.7.3.3. Le composant TextArea

La classe TextArea encapsule une zone de saisie de texte multiligne. Elle hérite de la classe TextBoxBase.

La méthode setVisibleLines() permet de préciser le nombre de lignes qui seront visibles.

Exemple :

```
    TextArea textArea = new TextArea();
    textArea.setWidth("200px");
    textArea.setVisibleLines(5);
    RootPanel.get("app").add(textArea);
```

Avant GWT version 1.6, la gestion des événements se faisait avec des listeners. Plusieurs listeners sont utilisables avec ce composant : ChangeListener, ClickListener, FocusListener, KeyboardListener, MouseListener et MouseWheelListener

A partir de GWT version 1.6, la gestion des événements se fait avec des handlers. Plusieurs handlers sont utilisables avec ce composant : DomHandler, ChangeHandler, ValueChangeHandler, BlurHandler, ClickHandler, FocusHandler, KeyDownHandler, KeyPressHandler, KeyUpHandler, MouseDownHandler, MouseMoveHandler, MouseOutHandler, MouseOverHandler, MouseUpHandler et MouseWheelHandler.

89.7.3.4. Le composant TextBox

La classe TextBox encapsule une zone de saisie de texte. Elle hérite de la classe TextBoxBase.

Elle possède plusieurs méthodes pour agir sur le composant dont les principales sont :

Méthodes	Rôle
int getMaxLength()	Obtenir le nombre maximum de caractères saisissables
void setMaxLength(int length)	Limiter le nombre de caractères saisissables

Exemple :

```
    TextBox textBox = new TextBox();
    textBox.setWidth("200px");
    textBox.setText("mon texte");
    textBox.setMaxLength(50);

    RootPanel.get("app").add(textBox);
    textBox.setFocus(true);
    textBox.setCursorPos(4);
```

Avant GWT version 1.6, la gestion des événements se faisait avec des listeners. Plusieurs listeners sont utilisables avec ce composant : ChangeListener, ClickListener, FocusListener, KeyboardListener, MouseListener et

MouseWheelListener

Exemple :

```
    TextBox textBox = new TextBox();
    textBox.setWidth("200px");
    textBox.setMaxLength(10);

    textBox.addKeyboardListener(new KeyboardListener() {
      @Override
      public void onKeyDown(Widget sender, char keyCode, int modifiers) {
      }

      @Override
      public void onKeyPress(Widget sender, char keyCode, int modifiers) {
        if (!Character.isDigit(keyCode)) {
          ((TextBox) sender).cancelKey();
        }
      }

      @Override
      public void onKeyUp(Widget sender, char keyCode, int modifiers) {
      }

    });

    RootPanel.get("app").add(textBox);
    textBox.setFocus(true);
```

A partir de GWT version 1.6, la gestion des événements se fait avec des handlers. Plusieurs handlers sont utilisables avec ce composant : DomHandler, ChangeHandler, ValueChangeHandler, BlurHandler, ClickHandler, FocusHandler, KeyDownHandler, KeyPressHandler, KeyUpHandler, MouseDownHandler, MouseMoveHandler, MouseOutHandler, MouseOverHandler, MouseUpHandler et MouseWheelHandler.

Exemple :

```
    TextBox textBox = new TextBox();
    textBox.setWidth("200px");
    textBox.setMaxLength(10);

    textBox.addKeyPressHandler(new KeyPressHandler() {
      @Override
      public void onKeyPress(KeyPressEvent event) {
        if (!Character.isDigit(event.getCharCode())) {
          ((TextBox) event.getSource()).cancelKey();
        }
      }
    });

    RootPanel.get("app").add(textBox);
    textBox.setFocus(true) ;
```

89.7.3.5. Le composant RichTextArea

La classe RichTextArea encapsule un composant qui permet à un utilisateur de saisir du texte avec un contenu formaté en HTML.

Le composant RichTextArea est un composant évolué et tout à la fois basique : il permet le formatage de textes riches par l'utilisateur et propose des objets pour formater le contenu texte mais il ne propose pas d'ensemble de boutons pour faciliter la mise en oeuvre de ces fonctionnalités de formatage.

Selon le navigateur utilisé par l'utilisateur, les fonctionnalités utilisables avec ce composant varient. Ces fonctionnalités sont réparties en trois catégories : Aucune (none), Basique (basic) et Etendue (extended).

Chaque catégorie est encapsulée dans un objet de formatage :

- Basique : encapsulée dans la classe RichTextArea.BasicFormatter
- Etendue : encapsulée dans la classe RichTextArea.ExtendedFormatter

Pour déterminer si la catégorie est supportée par le navigateur, il faut vérifier qu'il est possible d'obtenir une instance de l'objet correspondant et utiliser respectivement les méthodes getBasicFormatter() et getExtendedFormatter().

La classe RichTextArea.BasicFormatter propose des méthodes pour des fonctionnalités de formatage basiques, notamment :

Méthode	Rôle
String getBackColor()	Obtenir la couleur de fond
String getForeColor()	Obtenir la couleur
boolean isBold()	Indiquer si la zone est en gras
boolean isItalic()	Indiquer si la zone est en italique
boolean isUnderlined()	Indiquer si la zone est soulignée
void selectAll()	Sélectionner tout le texte
void setBackColor(String color)	Modifier la couleur de fond
void setFontName(String name)	Modifier la police de caractères
void setFontSize(RichTextArea.FontSize fontSize)	Modifier la taille de la police de caractères
void setForeColor(String color)	Modifier la couleur
void setJustification(RichTextArea.Justification justification)	Modifier l'alignement
void toggleBold()	Basculer la zone en gras
void toggleItalic()	Basculer la zone en italique
void toggleUnderline()	Basculer la zone en souligné

Toutes ces méthodes agissent sur la sélection courante ou, à défaut, sur le mot sur lequel le curseur est positionné.

La classe RichTextArea.ExtendedFormatter propose des méthodes pour des fonctionnalités de formatage avancées, notamment :

Méthode	Rôle
void createLink(String url)	Créer un hyperlien sur la zone vers l'url fournie en paramètre
void insertHorizontalRule()	Insérer une barre de séparation
void insertImage(String url)	Insérer une image
void insertOrderedList()	Débuter une liste ordonnée
void insertUnorderedList()	Débuter une liste avec puces
boolean isStrikethrough()	Indique si la zone est barrée
void leftIndent()	Indenter la zone
void removeFormat()	Supprimer tout le formatage de la zone.
void removeLink()	Supprimer l'hyperlien de la zone
void toggleStrikethrough()	Basculer la zone en barré

Exemple :

```
    VerticalPanel panel = new VerticalPanel();
    final RichTextArea richTextBox = new RichTextArea();
    final TextArea textArea = new TextArea();

    final ExtendedFormatter ef = richTextBox.getExtendedFormatter();
    final BasicFormatter bf = richTextBox.getBasicFormatter();
    richTextBox.setWidth("400px");
    richTextBox.setHTML("<h1>Titre</H1><p>Voici le <b>contenu</b> du paragraphe.</p>");

    textArea.setWidth("400px");
    textArea.setVisibleLines(5);

    HorizontalPanel boutons = new HorizontalPanel();
    panel.add(boutons);
    if (bf != null) {
        // ajout des boutons de fomatage basique
        boutons.add(new Button("Gras", new ClickListener(){
            public void onClick(Widget sender)
            {
                bf.toggleBold();
            }
        }));

        // ajout des boutons de fomatage etendu
        if (ef != null) {
            boutons.add(new Button("Barre", new ClickListener() {
                public void onClick(Widget sender) {
                    ef.toggleStrikethrough();
                }
            }));
        }
    }

    boutons.add(new Button("Text", new ClickListener() {
        public void onClick(Widget sender) {
            textArea.setText(richTextBox.getText());
        }
    }));
    boutons.add(new Button("Html", new ClickListener() {
        public void onClick(Widget sender) {
            textArea.setText(richTextBox.getHTML());
        }
    }));

    panel.add(richTextBox);
    panel.add(textArea);

    RootPanel.get("app").add(panel);
    richTextBox.setFocus(true);
```

Avant GWT version 1.6, la gestion des événements se faisait avec des listeners. Plusieurs listeners sont utilisables avec ce composant : ClickListener, FocusListener, KeyboardListener et MouseWheelListener

A partir de GWT version 1.6, la gestion des événements se fait avec des handlers. Plusieurs handlers sont utilisables avec ce composant : DomHandler, BlurHandler, ClickHandler, FocusHandler, KeyDownHandler, KeyPressHandler, KeyUpHandler, MouseDownHandler, MouseMoveHandler, MouseOutHandler, MouseOverHandler, MouseUpHandler et MouseWheelHandler

89.7.4. Les composants de sélection de données

89.7.4.1. Le composant ListBox

La classe ListBox encapsule une liste ou une liste déroulante qui permet à l'utilisateur de choisir un ou plusieurs éléments.

La classe ListBox possède plusieurs constructeurs notamment :

Constructeur	Rôle
ListBox()	Constructeur par défaut
ListBox(boolean isMultipleSelect)	Instancier un composant qui autorisera la sélection multiple d'éléments

La classe ListBox possède plusieurs méthodes notamment :

Méthode	Rôle
void addItem(String item)	Ajouter un élément
void addItem(String item, String value)	Ajouter un élément en précisant sa valeur
void clear()	Supprimer tous les éléments
int getItemCount()	Obtenir le nombre d'éléments de la ListBox
String getItemText(int index)	Obtenir le texte de l'élément dont l'index est fourni en paramètre
int getSelectedIndex()	Obtenir l'index de l'élément sélectionné
String getValue(int index)	Obtenir la valeur de l'élément dont l'index est fourni en paramètre
int getVisibleItemCount()	Obtenir le nombre d'éléments visibles
void insertItem(String item, int index)	Insérer un élément à l'index fourni
void insertItem(String item, String value, int index)	Insérer un élément à l'index fourni en précisant la valeur
boolean isItemSelected(int index)	Déterminer si un élément est sélectionné
boolean isMultipleSelect()	Déterminer si la sélection multiple est possible
void removeItem(int index)	Supprimer l'élément dont l'index est fourni en paramètre
void setItemSelected(int index, boolean selected)	Sélectionner ou non l'élément dont l'index est fourni en paramètre
void setItemText(int index, String text)	Modifier le texte de l'élément dont l'index est fourni en paramètre
void setMultipleSelect(boolean multiple)	Activer ou non la sélection multiple. Deprecated : il faut utiliser le constructeur ListBox(boolean)
void setSelectedIndex(int index)	Modifier l'index correspondant à l'élément sélectionné
void setValue(int index, String value)	Modifier la valeur de l'élément dont l'index est fourni en paramètre
void setVisibleItemCount(int visibleItems)	Modifier le nombre d'éléments affichés

L'affichage du composant se fait par défaut sous la forme d'une liste déroulante.

Exemple :

```
ListBox listBox = new ListBox();

for (int i=1; i <10 ; i++) {
  listBox.addItem("element "+i);
}

RootPanel.get("app").add(listBox);
```

La méthode setVisibleItemCount() permet de préciser le nombre d'éléments de la liste qui seront affichés :

- Avec la valeur 1, le composant est affiché sous la forme d'une liste déroulante
- Avec une autre valeur, le composant est affiché sous la forme d'une liste dont le nombre d'éléments affichés correspond à la valeur fournie

Exemple :

```
    ListBox listBox = new ListBox();

    for (int i=1; i <10 ; i++) {
      listBox.addItem("element "+i);
    }
    listBox.setVisibleItemCount(5);

    RootPanel.get("app").add(listBox);
```

Il est possible de gérer plusieurs événements émis par ce composant.

La classe ListBox possède plusieurs listeners utilisables jusqu'à GWT 1.5 : ChangeListener, ClickListener, MouseListener et MouseWheelListener.

Exemple :

```
    ListBox listBox = new ListBox();

    for (int i=1; i <10 ; i++) {
      listBox.addItem("element "+i, ""+i);
    }
    listBox.addChangeListener(new ChangeListener(){
      @Override
      public void onChange(Widget sender) {
        ListBox list = (ListBox) sender;
        int index = list.getSelectedIndex();
        Window.alert("element selectionne, index="+index+", texte="+list.getItemText(index)+",
valeur="+list.getValue(index));
      }
    } );

    listBox.setVisibleItemCount(5);
    RootPanel.get("app").add(listBox);
```

A partir de GWT 1.6, le composant ListBox propose plusieurs handlers : ClickHandler, MouseDownHandler, MouseMoveHandler, MouseOutHandler, MouseOverHandler, MouseUpHandler et MouseWheelHandler.

Exemple :

```
    ListBox listBox = new ListBox();

    for (int i=1; i <10 ; i++) {
      listBox.addItem("element "+i, ""+i);
    }
    listBox.addChangeHandler(new ChangeHandler(){
      @Override
      public void onChange(ChangeEvent event) {
        ListBox list = (ListBox) event.getSource();
        int index = list.getSelectedIndex();
        Window.alert("element selectionne,
index="+index+", texte="+list.getItemText(index)+",
valeur="+list.getValue(index));
      }
    } );

    listBox.setVisibleItemCount(5);
    RootPanel.get("app").add(listBox);
```

89.7.4.2. Le composant SuggestBox

La classe SuggestBox encapsule une zone de texte qui permet de sélectionner des suggestions commençant par le texte saisi.

La classe abstraite SuggestOracle encapsule les suggestions liées à une requête.

L'implémentation par défaut est la classe MultiWordSuggestOracle qui recherche les suggestions parmi celles dont au moins un mot commence par le motif de recherche. La recherche n'est pas sensible à la casse et les suggestions retournées sont triées par ordre alphabétique avec le motif recherché mis en évidence.

La méthode add() permet d'ajouter une suggestion. La méthode addAll() permet d'ajouter plusieurs suggestions et la méthode clear() permet de supprimer toutes les suggestions.

Exemple :

```
    VerticalPanel panel = new VerticalPanel();
    MultiWordSuggestOracle oracle = new MultiWordSuggestOracle();
    oracle.add("Peche fruitee");
    oracle.add("Abricot");
    oracle.add("Banane");
    oracle.add("Fraise");
    oracle.add("Framboise");
    oracle.add("Pomme");
    oracle.add("Poire");

    SuggestBox suggestbox = new SuggestBox(oracle);
    suggestbox.setAnimationEnabled(true);
    panel.add(new Label("Fruit : "));
    panel.add(suggestbox);
    RootPanel.get("app").add(panel);

    suggestbox.setFocus(true);
```

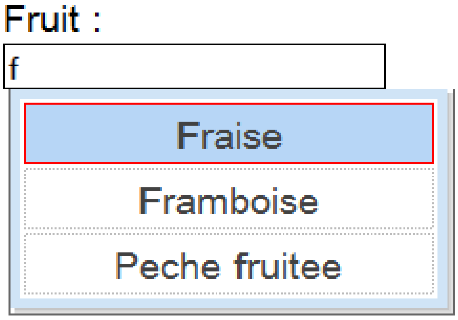
Les principaux styles CSS associés sont :

Style	Rôle
.gwt-SuggestBox	Apparence de la zone de saisie
.gwt-SuggestBoxPopup	Apparence de la popup affichant les suggestions
.gwt-SuggestBoxPopup .item	Apparence d'une suggestion
.gwt-SuggestBoxPopup .item-selected	Apparence de la suggestion sélectionnée

Résultat :

```
/* format de la zone de saisie */
.gwt-SuggestBox
{
border                 :    1px solid #000;
text-align             :    left;
width                  :    200px;
}

/* format de la popup affichant les suggestions */
.gwt-SuggestBoxPopup
{
cursor                 :    pointer;
border                 :    1px solid #666;
border-top             :    0;
background-color       :    #fff;
}

/* format d'une suggestion */
.gwt-SuggestBoxPopup .item
{
```

```
text-align             :    center;
border                 :    1px dotted #bbb;
width                  :    200px;
}

/* format de la suggestion selectionnee */
.gwt-SuggestBoxPopup .item-selected
{
border                       :    1px solid #f00;
}
```



Avant GWT version 1.6, la gestion des événements se faisait avec des listeners. Plusieurs listeners sont utilisables avec ce composant : ChangeListener, ClickListener, FocusListener et KeyboardListener.

A partir de GWT version 1.6, la gestion des événements se fait avec des handlers. Plusieurs handlers sont utilisables avec ce composant : KeyDownHandler, KeyPressHandler, KeyUpHandler, SelectionHandler, et ValueChangeHandler.

Il est aussi possible d'utiliser plusieurs handlers sur la zone de texte associée au composant dont l'instance est obtenue en invoquant la méthode getTextBox() : ChangeHandler et ClickHandler.

89.7.4.3. Le composant DateBox

La suite de cette section sera développée dans une version future de ce document

89.7.4.4. Le composant DatePicker

La suite de cette section sera développée dans une version future de ce document

89.7.5. Les composants HTML

La suite de cette section sera développée dans une version future de ce document

89.7.5.1. Le composant Frame

La suite de cette section sera développée dans une version future de ce document

89.7.5.2. Le composant HTML

Le composant HTML permet d'afficher un contenu HTML dans l'application. Il propose plusieurs constructeurs dont un qui permet de fournir le code HTML qui sera affiché.

Exemple :

```
    HTML widget = new HTML(
      "<div id='panneau' style='background-color: yellow; border:"+
      "1px solid black; width: 200px; text-align: center;'>"+
      "Mon Message d'avertissement</div>");
    RootPanel.get("app").add(widget);
```

La méthode setHTML() permet de modifier le contenu du code HTML affiché par le composant.

Il est possible de gérer plusieurs événements émis par ce composant.

La classe HTML possède plusieurs listeners utilisables jusqu'à GWT 1.5 : ClickListener, MouseListener et MouseWheelListener.

A partir de GWT 1.6, le composant HTML propose plusieurs handlers : DomHandler, ClickHandler, ErrorHandler, LoadHandler MouseDownHandler, MouseMoveHandler, MouseOutHandler, MouseOverHandler, MouseUpHandler et MouseWheelHandler.

89.7.5.3. FileUpload

La suite de cette section sera développée dans une version future de ce document

89.7.5.4. Hidden

La classe Hidden encapsule un champ invisile d'un formulaire HTML.

La classe Hidden possède plusieurs constructeurs notamment :

Constructeur	Rôle
Hidden()	Constructeur par défaut
Hidden(String name)	Constructeur qui attend le nom du champ en paramètre

Hidden(String name, String value)	Constructeur qui attend le nom et la valeur du champ en paramètres

Elle possède plusieurs méthodes qui permettent de manipuler les données de la classe, notamment :

Méthode	Rôle
String getDefaultValue()	Obtenir la valeur par défaut du champ
String getName()	Obtenir le nom du champ
String getValue()	Obtenir la valeur du champ
void setDefaultValue(String defaultValue)	Modifier la valeur par défaut du champ
void setName(name)	Modifier la nom du champ
void setValue(String value)	Modifier la valeur du champ

Ce composant n'ayant pas de rendu graphique, il ne possède aucun événement.

89.7.6. Le composant Tree

Le composant Tree encapsule l'affichage de données sous une forme arborescente. Chaque élément de l'arborescence qui possède au moins un élément fils peut être plié ou déplié pour faire apparaitre ou non les éléments de sa sous-arborescence.

La classe Tree propose plusieurs méthodes notamment :

Méthode	Rôle
void add(Widget widget)	Ajouter un élément affichant le composant fourni en paramètre à la racine de l'arborescence
TreeItem addItem(String itemText)	Ajouter un élément affichant un texte
void addItem(TreeItem item)	Ajouter un élément à la racine de l'arborescence
TreeItem addItem(Widget widget)	Ajouter un élément qui va afficher le composant fourni en paramètre
void clear()	Supprimer tous les éléments
void ensureSelectedItemVisible()	Rendre visible l'élément sélectionné, au besoin en dépliant ses parents
TreeItem getItem(int index)	Obtenir l'élément dont l'index est fourni
int getItemCount()	Obtenir le nombre d'éléments
TreeItem getSelectedItem()	Obtenir l'élément sélectionné
boolean isAnimationEnabled()	Déterminer si l'animation est activée lorsqu'un élément est replié ou déplié
java.util.Iterator<Widget> iterator()	Obtenir un iterator sur les composants
void removeItem(TreeItem item)	Supprimer un élément de la racine
void removeItems()	Supprimer tous les éléments de la racine
void setAnimationEnabled(boolean enable)	Activer ou non l'animation lorsqu'une branche est pliée ou dépliée
void setSelectedItem(TreeItem item)	Sélectionner un élément
java.util.Iterator<TreeItem> treeItemIterator()	Obtenir un itérateur sur les éléments de l'arborescence

La classe Tree est un conteneur pour des éléments de type TreeItem

Le classe TreeItem encapsule un élément d'un composant Tree. Elle propose plusieurs méthodes notamment :

Méthode	Rôle
TreeItem addItem(String itemText)	Ajouter un élément affichant le texte fourni en paramètre
void addItem(TreeItem item)	Ajouter un élément fils
TreeItem addItem(Widget widget)	Ajouter un composant comme élément fils
TreeItem getChild(int index)	Obtenir l'élément fils dont l'index est fourni en paramètre
int getChildCount()	Obtenir le nombre d'éléments fils
int getChildIndex(TreeItem child)	Obtenir l'index d'un élément fils
TreeItem getParentItem()	Obtenir l'élément père
boolean getState()	Déterminer si les éléments fils sont affichés ou non
String getText()	Obtenir le texte de l'élément
Tree getTree()	Obtenir le composant Tree encapsulant toute l'arborescence
Object getUserObject()	Obtenir un objet associé à l'élément
Widget getWidget()	Obtenir le composant associé à l'élément
boolean isSelected()	Déterminer si l'élément est sélectionné ou non
void remove()	Retirer l'élément de l'arborescence
void removeItem(TreeItem item)	Retirer l'élément fils
void removeItems()	Retirer tous les éléments fils
void setSelected(boolean selected)	Sélectionner ou non l'élément
void setState(boolean open)	Afficher ou non les éléments fils
void setText(String text)	Définir le texte de l'élément
void setUserObject(Object userObj)	Associer un objet à l'élément
void setWidget(Widget newWidget)	Définir le composant de l'élément

Les objets de type TreeItem ne peuvent être attachés qu'à un composant de type Tree.

Exemple :

```
    Tree tree = new Tree();
    tree.setAnimationEnabled(false);

    TreeItem element1 = new TreeItem("Element 1");
    element1.addItem("Element 1-1");
    element1.addItem("Element 1-2");
    element1.addItem("Element 1-3");
    tree.addItem(element1);
    TreeItem sousElement1_3 = new TreeItem("Element 1-3");
    sousElement1_3.addItem("Element 1-3-1");
    sousElement1_3.addItem("Element 1-3-2");
    TreeItem element2 = new TreeItem("Element 2");
    element2.addItem("Element 2-1");
    element2.addItem("Element 2-2");
    TreeItem element2_3 = element2.addItem("Element 2-3");
    tree.addItem(element2);
    element1.addItem(sousElement1_3);

    RootPanel.get("app").add(tree);
```

Il est possible d'afficher un composant dans un élément.

Exemple :

```
    Tree tree = new Tree();
    tree.setAnimationEnabled(false);

    TreeItem element1 = new TreeItem("Element 1");
    element1.addItem("Element 1-1");
    element1.addItem("Element 1-2");
    element1.addItem("Element 1-3");
    tree.addItem(element1);
    TreeItem sousElement1_3 = new TreeItem("Element 1-3");
    sousElement1_3.addItem(new CheckBox("Element 1-3-1"));
    sousElement1_3.addItem(new CheckBox("Element 1-3-2"));
    TextBox element1_3_3 = new TextBox();
    element1_3_3.setText("Element 1-3-3");
    sousElement1_3.addItem(element1_3_3);

    element1.addItem(sousElement1_3);

    RootPanel.get("app").add(tree);
```

Par défaut, le composant Tree ne possède pas de scrollbars : il s'agrandit au fur et à mesure des éléments dépliés. Pour limiter la taille du composant et afficher une scrollbar, il faut l'encapsuler dans un panneau de type ScrollPanel.

Exemple :

```
    Tree tree = new Tree();
    tree.setAnimationEnabled(false);

    TreeItem element1 = new TreeItem("Element 1");
    element1.addItem("Element 1-1");
    element1.addItem("Element 1-2");
    element1.addItem("Element 1-3");
    tree.addItem(element1);
    TreeItem sousElement1_3 = new TreeItem("Element 1-3");
    sousElement1_3.addItem("Element 1-3-1");
    sousElement1_3.addItem("Element 1-3-2");
    TreeItem element2 = new TreeItem("Element 2");
    element2.addItem("Element 2-1");
    element2.addItem("Element 2-2");
    TreeItem element2_3 = element2.addItem("Element 2-3");
    tree.addItem(element2);
    element1.addItem(sousElement1_3);
    ScrollPanel scrollPanel = new ScrollPanel(tree);
    scrollPanel.setWidth("250px");
    scrollPanel.setHeight("130px");

    RootPanel.get("app").add(scrollPanel);
    tree.setSelectedItem(element2_3);
    tree.ensureSelectedItemVisible();
```

Avant GWT version 1.6, la gestion des événements se faisait avec des listeners. Plusieurs listeners sont utilisables avec ce composant : FocusListener, KeyboardListener, MouseListener et TreeListener .

Exemple :

```
        Tree tree = new Tree();
    tree.setAnimationEnabled(false);

    TreeItem element1 = new TreeItem("Element 1");
    element1.addItem("Element 1-1");
    element1.addItem("Element 1-2");
    element1.addItem("Element 1-3");
    tree.addItem(element1);
    TreeItem sousElement1_3 = new TreeItem("Element 1-3");
    sousElement1_3.addItem(new CheckBox("Element 1-3-1"));
    sousElement1_3.addItem(new CheckBox("Element 1-3-2"));
    TextBox element1_3_3 = new TextBox();
    element1_3_3.setText("Element 1-3-3");
    sousElement1_3.addItem(element1_3_3);

    element1.addItem(sousElement1_3);
    tree.addTreeListener(new TreeListener() {
      @Override
      public void onTreeItemSelected(TreeItem item) {
        Window.alert("Selection : "+item.getText());
      }

      @Override
      public void onTreeItemStateChanged(TreeItem item) {
        if (item.getState()) {
          Window.alert("Affichage noeud fils : "+item.getText());
        } else {
          Window.alert("Masquage noeud fils : "+item.getText());
        }
      }
    });

    RootPanel.get("app").add(tree);
```

A partir de GWT version 1.6, la gestion des événements se fait avec des handlers. Plusieurs handlers sont utilisables avec ce composant : DomHandler, BlurHandler, CloseHandler, FocusHandler, KeyDownHandler, KeyPressHandler, KeyUpHandler, MouseDownHandler, MouseMoveHandler, MouseOutHandler, MouseOverHandler, MouseUpHandler, MouseWheelHandler, OpenHandler et SelectionHandler.

Exemple :

```
    Tree tree = new Tree();

    tree.setAnimationEnabled(false);
    TreeItem element1 = new TreeItem("Element 1");
    element1.addItem("Element 1-1");
    element1.addItem("Element 1-2");
    element1.addItem("Element 1-3");
    tree.addItem(element1);

    TreeItem sousElement1_3 = new TreeItem("Element 1-3");
    sousElement1_3.addItem(new CheckBox("Element 1-3-1"));
    sousElement1_3.addItem(new CheckBox("Element 1-3-2"));
    TextBox element1_3_3 = new TextBox();
    element1_3_3.setText("Element 1-3-3");
    sousElement1_3.addItem(element1_3_3);
    element1.addItem(sousElement1_3);

    tree.addSelectionHandler(new SelectionHandler<TreeItem>() {
      @Override
      public void onSelection(SelectionEvent<TreeItem> event) {
        Window.alert("Selection : " + event.getSelectedItem().getText());
      }
    });

    tree.addOpenHandler(new OpenHandler<TreeItem>() {
      @Override
      public void onOpen(OpenEvent<TreeItem> event) {
        Window.alert("Affichage noeud fils : "
            + event.getTarget().getText());
      }
    });

    tree.addCloseHandler(new CloseHandler<TreeItem>() {
      @Override
      public void onClose(CloseEvent<TreeItem> event) {
        Window.alert("Masquage noeud fils : "
            + event.getTarget().getText());
      }
    });

    RootPanel.get("app").add(tree);
```

89.7.7. Les menus

La classe MenuBar encapsule un menu et les éléments qui le composent.

Exemple :

```
package fr.jmdoudoux.dejgwt.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.user.client.Command;
import com.google.gwt.user.client.Window;
import com.google.gwt.user.client.ui.MenuBar;
import com.google.gwt.user.client.ui.RootPanel;

public class MonApp implements EntryPoint {

  public void onModuleLoad() {
    MenuBar menu = new MenuBar();
    MenuBar menu1 = new MenuBar(true);
    MenuBar menu2 = new MenuBar(true);

    menu2.addItem("menu2_1", new MonMenuCommand());
    menu2.addItem("menu2_2", new MonMenuCommand());
    menu2.addItem("menu2_3", new MonMenuCommand());

    menu1.addItem("menu1_1", new MonMenuCommand());
    menu1.addItem("menu1_2", new MonMenuCommand());

    menu.addItem("menu1", menu1);
    menu.addItem("menu2", menu2);
    menu.setAutoOpen(true);
    RootPanel.get("menu").add(menu);

    menu1.addStyleName("submenu");
    menu2.addStyleName("submenu");
  }

  public class MonMenuCommand implements Command {
    public void execute() {
      Window.alert("Element du menu clické");
    }
  }
}
```

Le rendu du menu est assuré par des styles CSS. Le plus simple est de les définir dans un fichier .css

Résultat :

```
body {
   margin: 0;
   padding: 0;
   background: #ffffff;
}

.gwt-MenuBar {
   background: #a0a0a0;
   border: 1px solid #3f3f3f;
   cursor: pointer;
}

.gwt-MenuBar .gwt-MenuItem {
   font-family: Arial, sans-serif;
   font-size: 12px;
   color: #ffffff;
   font-weight: bold;
   padding-right: 10px;

}
```

```
.gwt-MenuBar .gwt-MenuItem-selected {
   font-family: Arial, sans-serif;
   font-size: 12px;
   color: #ffffff;
   font-weight: bold;
   padding-right: 10px;
}
```

Ce fichier doit être déclaré dans le fichier de configuration de l'application grâce au tag <stylesheet>. L'attribut src permet de préciser le fichier.

Exemple :

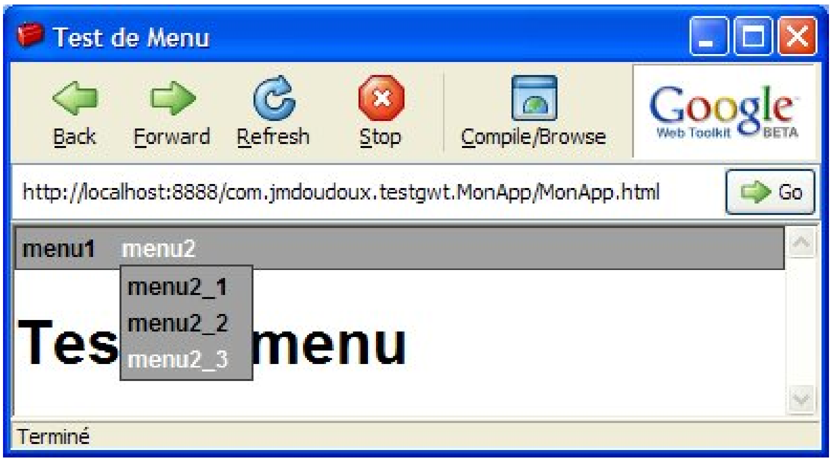
```
<module>

    <!-- Inherit the core Web Toolkit stuff.                  -->
    <inherits name='com.google.gwt.user.User' />

    <stylesheet src="MonApp.css" />

    <!-- Specify the app entry point class.                   -->
    <entry-point class='fr.jmdoudoux.dejgwt.client.MonApp' />

</module>
```

Enfin, un calque nommé menu est défini dans le fichier html de la page.

Exemple :

```
<html>
<head>
<title>Test de Menu</title>
<style>
  body,td,a,div,.p{font-family:arial,sans-serif}
  div,td{color:#000000}
  a:link,.w,.w a:link{color:#0000cc}
  a:visited{color:#551a8b}
  a:active{color:#ff0000}
</style>

<meta name='gwt:module' content='fr.jmdoudoux.dejgwt.MonApp'>
</head>
<body>
  <div id="menu"></div>
  <script language="javascript" src="gwt.js"></script>
  <iframe id="__gwt_historyFrame" style="width:0;height:0;border:0"></iframe>
  <h1>Test de menu</h1>
</body>
</html>
```

Résultat :



89.7.8. Le composant TabBar

La classe TabBar encapsule une barre d'onglets. Ce composant est essentiellement utilisé dans un panneau TabPanel

Exemple :

```
    VerticalPanel panel = new VerticalPanel();
    final TabBar tabBar = new TabBar();
    final Label label = new Label();

    tabBar.addTab("Onglet 1");
    tabBar.addTab("Onglet 2");
    tabBar.addTab("Onglet 3");
    tabBar.addTab("Onglet 4");
    tabBar.addTabListener(new TabListener() {
      @Override
      public boolean onBeforeTabSelected(SourcesTabEvents sender, int tabIndex) {
        return true;
      }

      @Override
      public void onTabSelected(SourcesTabEvents sender, int tabIndex) {
        int selectionne = tabBar.getSelectedTab();

        if (selectionne != 0) {
          tabBar.setTabEnabled(3, true);
        }

        if (tabIndex == 0) {
          tabBar.setTabEnabled(3, false);
        }

        label.setText("Selection de l'onglet : "+tabBar.getTabHTML(tabIndex));
      }
    });

    panel.add(tabBar);
    panel.add(label);
    tabBar.selectTab(0);

    RootPanel.get("app").add(panel);
```

Avant GWT version 1.6, la gestion des événements se faisait avec des listeners. Un seul listener est utilisable avec ce composant : TabListener.

Exemple :

```
    VerticalPanel panel = new VerticalPanel();
    final TabBar tabBar = new TabBar();
    final Label label = new Label();
```

```
tabBar.addTab("Onglet 1");
tabBar.addTab("Onglet 2");
tabBar.addTab("Onglet 3");
tabBar.addTab("Onglet 4");
tabBar.addTabListener(new TabListener() {
  @Override
  public boolean onBeforeTabSelected(
     SourcesTabEvents sender, int tabIndex)
  {
    return true;
  }

  @Override
  public void onTabSelected(SourcesTabEvents sender, int tabIndex)
  {
    int selectionne = tabBar.getSelectedTab();

    if (selectionne != 0) {
      tabBar.setTabEnabled(3, true);
    }

    if (tabIndex == 0) {
      tabBar.setTabEnabled(3, false);
    }

    label.setText("Selection de l'onglet : "+tabBar.getTabHTML(tabIndex));
  }
});
panel.add(tabBar);
panel.add(label);
tabBar.selectTab(0);

RootPanel.get("app").add(panel);
```

A partir de GWT version 1.6, la gestion des événements se fait avec des handlers. Plusieurs handlers sont utilisables avec ce composant : SelectionBeforeHandler et SelectionHandler.

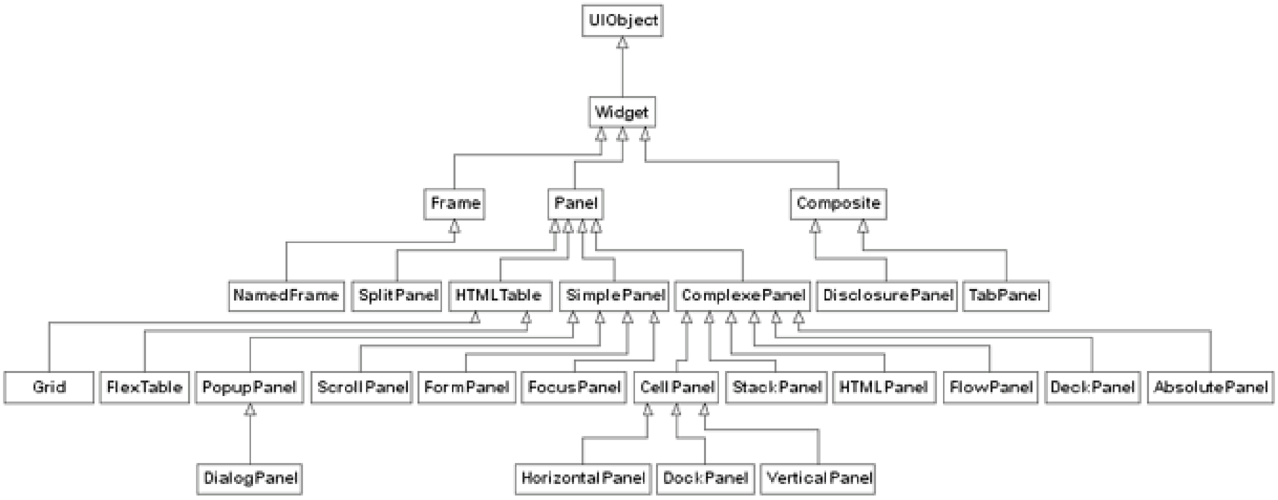
Exemple :

```
VerticalPanel panel = new VerticalPanel();
final TabBar tabBar = new TabBar();
final Label label = new Label();

tabBar.addTab("Onglet 1");
tabBar.addTab("Onglet 2");
tabBar.addTab("Onglet 3");
tabBar.addTab("Onglet 4");
tabBar.addSelectionHandler(new SelectionHandler<Integer>() {
  @Override
  public void onSelection(SelectionEvent<Integer> event) {
    int courant = tabBar.getSelectedTab();
    int selectionne = event.getSelectedItem();

    if (courant != 0) {
      tabBar.setTabEnabled(3, true);
    }

    if (selectionne == 0) {
      tabBar.setTabEnabled(3, false);
    }

    label.setText("Selection de l'onglet : "+tabBar.getTabHTML(selectionne));
  }
});

panel.add(tabBar);
panel.add(label);
tabBar.selectTab(0);

RootPanel.get("app").add(panel);
```

89.8. Les panneaux (panels)

Ils permettent d'organiser les composants affichés sur la page : selon leurs fonctionnalités, ils peuvent gérer le positionnement des composants ou leur visibilité.

Les panneaux permettent d'assurer la structure visuelle de l'application. GWT propose un ensemble complet de composants graphiques de types conteneurs pour organiser et assembler les composants graphiques.



Les composants de type Panel contiennent d'autres composants et permettent de les organiser. Ils sont donc plus que des conteneurs car ils sont aussi des gestionnaires de positionnement.

Ceci est essentiellement dû au fait que le rendu HTML de ces composants est généralement soit un élément table soit un élément div HTML.

Panneau	Rendu HTML	Version de GWT	Description
AbsolutePanel	DIV	1.0	Panneau qui permet le positionnement absolu (grâce à leur position) des composants
CaptionPanel		1.5	Panneau qui possède un titre
CellPanel	TABLE	1.0	Panneau abstrait pour une cellule d'un panneau qui en est composé
ComplexPanel		1.0	Classe abstraite de base pour les panneaux possédant plusieurs composants
DeckPanel	DIV		Panneau qui n'affiche qu'un seul composant à la fois parmi ceux qu'il contient
DisclosurePanel	TABLE	1.4	
DockPanel	TABLE	1.0	Panneau qui permet de positionner les composants dans 5 zones (N, S, E, W et centre)
FlexTable		1.0	
FlowPanel	DIV	1.0	Panneau qui est un simple DIV
FocusPanel	DIV	1.0	
FormPanel	DIV	1.1	Panneau qui contient un formulaire HTML
Frame	IFRAME	1.0	Panneau sous la forme d'un IFRAME
Grid		1.0	
HorizontalPanel	TABLE	1.0	Panneau avec alignement horizontal des composants

HorizontalSplitPanel	DIV	1.4	Panneau composé de deux cellules l'une à côté de l'autre, redimensionnable en hauteur
HTMLPanel	DIV	1.0	Panneau qui permet d'afficher un contenu HTML
HTMLTable			
LazyPanel	DIV	1.6	Panneau qui permet de différer la création de son rendu au moment de son affichage (GWT 1.6)
Panel		1.0	Classe abstraite de base pour les autres panels
PopupPanel			
RootPanel		1.0	
ScrollPanel	DIV	1.0	
SimplePanel	DIV	1.0	Classe abstraite de base pour les panneaux ne possédant qu'un seul composant
StackPanel	TABLE	1.0	
TabPanel	TABLE	1.0	Panneau sous la forme d'onglets
VerticalPanel	TABLE	1.0	Panneau avec alignement vertical des composants
VerticalSplitPanel	DIV	1.4	Panneau composé de deux cellules l'une au-dessus de l'autre, redimensionnable en hauteur

Comme pour les composants, ils possèdent une représentation en Java et en JavaScript. Le constructeur de la classe se charge de créer le ou les éléments nécessaires dans l'arbre DOM.

Chaque panneau peut contenir des composants ou d'autres panneaux. Le panneau principal est encapsulé dans la classe RootPanel.

Les autres panneaux servent de conteneur pour les composants graphiques.

La suite de cette section sera développée dans une version future de ce document

89.8.1. La classe Panel

La plupart des panneaux qui contiennent un ou plusieurs composants héritent de façon directe ou indirecte de la classe Panel.

Cette classe implémente l'interface HasWidgets. Cette interface définit plusieurs méthodes :

Méthode	Rôle
add(Widget)	Ajouter le composant fourni en paramètres
Clear()	Supprimer tous les composants contenus dans le panneau
iterator()	Obtenir un objet de type Iterator pour parcourir tous les composants inclus dans le panneau.
remove(Widget)	Supprimer le composant fourni en paramètre

Cette classe possède plusieurs classes filles directes : ComplexPanel, HorizontalSplitPanel, HTMLTable, SimplePanel, et VerticalSplitPanel.

89.8.2. La classe RootPanel

Cette classe encapsule la page affichée et l'associe au navigateur. C'est le seul panneau permettant cette association.

La classe RootPanel est la classe qui encapsule le panneau racine qui doit obligatoirement qui doit obligatoirement être au sommet de la hiérarchie des panneaux et composants.

Ce panneau encapsule une partie de la page html de l'application en permettant un accès à certains de ses éléments. Il n'est donc pas un conteneur au sens strict mais permet un accès aux éléments de l'arbre DOM de la page.

La méthode get() permet d'obtenir une référence sur l'élément du DOM dont l'id est fourni en paramètre de la méthode. Sans paramètre, cette méthode renvoie l'objet de type RootPanel qui encapsule la page.

Exemple :

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN">
<html>
  <head>
    <meta
http-equiv="content-type" content="text/html;
charset=ISO-8859-15">
    <link
type="text/css" rel="stylesheet" href="TestGWT.css">

<title>Application de test GWT</title>
    <script
type="text/javascript" language="javascript"
src="testgwt/testgwt.nocache.js"></script>
  </head>
  <body>

<h1>Application de test GWT</h1>
    <table
align="center">
      <tr>
        <td
id="menu"></td><td id="app"></td>
      </tr>
    </table>
  </body>
</html>
```

Il n'est pas possible d'instancier un objet de type RootPanel : il faut utiliser la méthode get() qui renvoie le panneau par défaut ou sa surcharge qui attend en paramètre l'id d'un élément de la page html de l'application.

Exemple :

```
    Label libelle = new Label("Bonjour");

    RootPanel.get("app").add(libelle);
```

Il est ainsi possible d'accéder à tous les éléments de la page html qui possèdent un id.

89.8.3. La classe SimplePanel

La classe SimplePanel est un panneau qui peut contenir un seul composant.

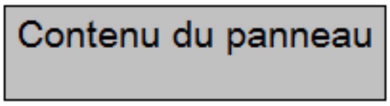
Ce panneau est implémenté sous la forme d'un simple tag DIV en HTML.

Exemple :

```
    SimplePanel panel = new SimplePanel();
    panel.setSize("200px", "50px");
    panel.addStyleName("monPanneau");
    Label label = new Label("Contenu du panneau");
    panel.add(label);
    RootPanel.get("app").add(panel);
```

Résultat :

```
.monPanneau {
border-color: black;
border-width : 1px;
border-style: solid;
background-color: silver;
text-align: center;
}
```



Contenu du panneau

89.8.4. La classe ComplexPanel

La classe abstraite ComplexPanel est la classe de base pour un panneau qui peut contenir plusieurs composants.

Elle propose plusieurs méthodes de base pour gérer les composants contenus dans le panneau notamment :

Méthode	Rôle
add(Widget, Element);	Ajouter un nouveau composant
getChildren()	Obtenir une collection des composants contenus dans le panneau
getWidget(int)	Obtenir le composant à partir de son index
getWidgetCount()	Obtenir le nombre de composants contenus dans le panneau
getWidgetIndex(Widget)	Obtenir l'index d'un composant
insert(Widget, Element, int)	Insérer un nouveau composant à l'index fourni
iterator()	Obtenir un iterator sur les composants du panneau
remove(int)	Supprimer le composant à l'index fourni
remove(Widget)	Supprimer le composant

La classe ComplexPanel possède plusieurs classes filles notamment : AbsolutePanel, CellPanel, DeckPanel, FlowPanel, HTMLPanel et StackPanel.

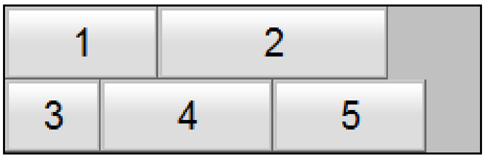
89.8.5. La classe FlowPanel

La classe FlowPanel est un panneau dans lequel les composants sont ajoutés les uns à la suite des autres avec un passage à la ligne dès que la place manque pour contenir le composant. Ce panneau arrange les composants qu'il contient les uns à côté des autres en allant du haut à gauche vers le bas à droite.

Le panneau FlowPanel est un simple élément HTML <DIV> avec le style display:inline dans l'arbre DOM .

Exemple :

```
    FlowPanel panel = new FlowPanel();
    panel.setWidth("250px");
    panel.addStyleName("monPanneau");

    Button bouton = new Button("1");
    bouton.setWidth("80px");
    panel.add(bouton);

    bouton = new Button("2");
    bouton.setWidth("120px");
    panel.add(bouton);

    bouton = new Button("3");
    bouton.setWidth("50px");
    panel.add(bouton);

    bouton = new Button("4");
    bouton.setWidth("90px");
    panel.add(bouton);

    bouton = new Button("5");
    bouton.setWidth("80px");
    panel.add(bouton);

    RootPanel.get("app").add(panel);
```



89.8.6. La classe DeckPanel

La classe DeckPanel est un panneau qui affiche un composant à la fois. Ce panneau est utilisé dans un TabPanel pour afficher le contenu de l'onglet sélectionné.

Un exemple typique d'utilisation est pour la mise en oeuvre d'un assistant.

Exemple :

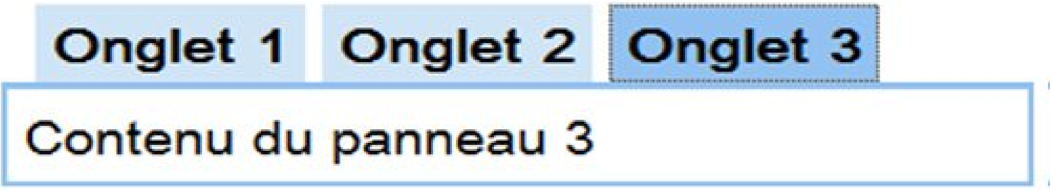
```
    final DeckPanel panel = new DeckPanel();
    panel.setSize("200px", "50px");
    panel.addStyleName("monPanneau");
    panel.add(new Label("Contenu cellule 1"));
    panel.add(new Label("Contenu cellule 2"));
    panel.add(new Label("Contenu cellule 3"));

    panel.showWidget(0);

    Timer timer = new Timer()
    {
      public void run()
      {
        int index = panel.getVisibleWidget();
        index++;
        if (index == panel.getWidgetCount()) {
          index = 0;
        }

        panel.showWidget(index);
      }
    };

    timer.scheduleRepeating(2000);

    RootPanel.get("app").add(panel);
```

89.8.7. La classe TabPanel

La classe TabPanel est un panneau composé d'onglets. Il est composé d'un ensemble de boutons, un pour chaque onglet et d'un composant DeckPanel qui affiche le contenu de l'onglet sélectionné.

Exemple :

```
    StringBuffer sb = new StringBuffer("<p>");
    TabPanel panel = new TabPanel();
    Panel contenu;
    contenu = new SimplePanel();
    contenu.add(new Label("Contenu du panneau 1"));
    panel.add(contenu, "Onglet 1");
    contenu = new SimplePanel();
    contenu.add(new Label("Contenu du panneau 2"));
    panel.add(contenu, "Onglet 2");
    contenu = new SimplePanel();
    contenu.add(new Label("Contenu du panneau 3"));
    panel.add(contenu, "Onglet 3");
    panel.selectTab(0);
    panel.setSize("500px", "250px");
    RootPanel.get("app").add(panel);
```



89.8.8. La classe FocusPanel

La suite de cette section sera développée dans une version future de ce document

89.8.9. La classe HTMLPanel

La classe HTMLPanel est un composant qui peut afficher du code HTML.

La classe HTMLPanel permet d'ajouter et de supprimer des éléments à partir de leur id.

Exemple :

```
    String html = "<table><tr><td nowrap><div id='libelle'>"
        + "</div></td><td><div id='saisie'>"
        + "</div></td></tr></table>";
    HTMLPanel panel = new HTMLPanel(html);

    panel.setSize("250px", "120px");
    panel.add(new Label("Libelle a saisir :"), "libelle");
    panel.add(new TextBox(), "saisie");

    RootPanel.get("app").add(panel);
```

89.8.10. La classe FormPanel

La suite de cette section sera développée dans une version future de ce document

89.8.11. La classe CellPanel

Cette classe est la classe abstraite pour une cellule d'un panneau qui en est composé. C'est la super classe de plusieurs panneaux : DockPannel, HorizontalPannel et VerticalPannel. Tous ces panneaux organisent les composants qu'ils contiennent dans des cellules.

89.8.12. La classe DockPanel

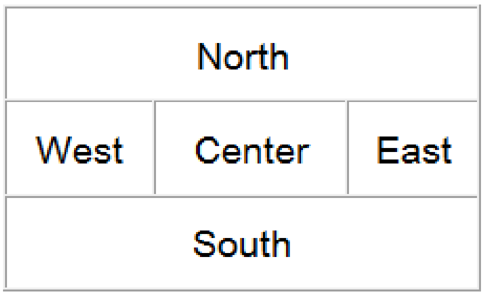
La classe DockPanel encapsule un panneau découpé en cinq parties positionnées relativement à la partie centrale :

- Une ligne qui contient la partie nord (NORTH)
- Une ligne qui contient les parties ouest (WEST), centre (CENTER) et est (EAST)
- Une ligne qui contient la partie sud (SOUTH)

Ce panneau utilise une table HTML.

Exemple :

```
    final DockPanel panel = new DockPanel();
    panel.setVerticalAlignment(HasAlignment.ALIGN_MIDDLE);
    panel.setHorizontalAlignment(HasAlignment.ALIGN_CENTER);
    panel.setWidth("250px");
    panel.setHeight("150px");
    panel.setBorderWidth(1);

    panel.add(new Label("North"), DockPanel.NORTH);
    panel.add(new Label("South"), DockPanel.SOUTH);
    panel.add(new Label("West"), DockPanel.WEST);
    panel.add(new Label("East"), DockPanel.EAST);
    panel.add(new Label("Center"), DockPanel.CENTER);
```



89.8.13. La classe HorizontalPanel

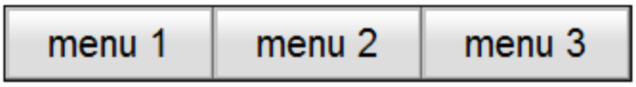
La classe HorizontalPanel encapsule un panneau qui peut contenir plusieurs composants alignés les uns à côté des autres.

Concrètement, c'est un tableau HTML où chaque composant est inséré dans une nouvelle cellule placée horizontalement à côté de la précédente.

Attention, même si sa taille est définie, le panneau n'est visible que s'il contient au moins un composant.

Exemple :

```
    HorizontalPanel panel = new HorizontalPanel();
    panel.addStyleName("monPanneau");

    Button menu1 = new Button("menu 1");
    Button menu2 = new Button("menu 2");
    Button menu3 = new Button("menu 3");

    panel.add(menu1);
    panel.add(menu2);
    panel.add(menu3);

    RootPanel.get("app").add(panel);
```



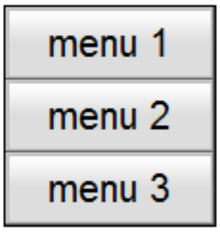
89.8.14. La classe VerticalPanel

La classe VerticalPanel encapsule un panneau qui peut contenir plusieurs composants alignés les uns au-dessus des autres. Ce panneau arrange les composants de façon verticale, les uns en dessous des autres comme dans une colonne.

Le panneau VerticalPanel est un élément HTML <TABLE> dans l'arbre DOM. A chaque appel de la méthode add(), une cellule est ajoutée dans une nouvelle ligne du tableau. Cette cellule contient le composant en paramètre de la méthode.

Exemple :

```
    VerticalPanel panel = new VerticalPanel();
    panel.addStyleName("monPanneau");

    Button menu1 = new Button("menu 1");
    Button menu2 = new Button("menu 2");
    Button menu3 = new Button("menu 3");

    panel.add(menu1);
    panel.add(menu2);
    panel.add(menu3);

    RootPanel.get("app").add(panel);
```



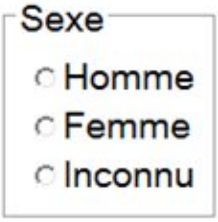
Attention, même si sa taille est définie, le panneau n'est visible que s'il contient au moins un composant..

89.8.15. La classe CaptionPanel

Ce panneau possède un titre : il permet de grouper des composants appartenant à un même ensemble fonctionnel.

Exemple :

```
    CaptionPanel panel = new CaptionPanel("Sexe");
    VerticalPanel vpBoutons = new VerticalPanel();
    vpBoutons.setStyleName("sexes");

    RadioButton rbHomme = new RadioButton("sexe","Homme");
    vpBoutons.add(rbHomme);
    RadioButton rbFemme = new RadioButton("sexe","Femme");
    vpBoutons.add(rbFemme);
    RadioButton rbInconnu = new RadioButton("sexe","Inconnu");
    vpBoutons.add(rbInconnu);
    panel.setContentWidget(vpBoutons);
```



89.8.16. La classe PopupPanel

La classe PopupPanel encapsule un panneau qui est capable de s'afficher au-dessus de tous les autres composants.

Il est possible que le panneau s'efface automatiquement (auto-hide) dès que l'utilisateur clique en dehors de celui-ci. Le panneau peut aussi être modal.

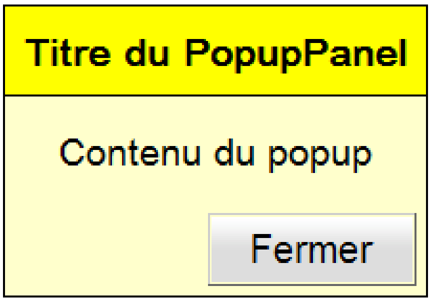
Ce panneau peut avoir de nombreuses utilités : afficher des données, demander une confirmation ou une petite quantité de données, verrouiller l'application, ...

Exemple :

```
  private static class TestPopupPanel extends PopupPanel {

    public TestPopupPanel(String message) {
      super(true, true);

      this.setStyleName("demo-popup");

      VerticalPanel contenuPopupPanel = new VerticalPanel();

      this.setAnimationEnabled(true);
      HTML titre = new HTML("Titre du PopupPanel");

      titre.setStyleName("demo-popup-header");
      HTML contenu = new HTML(message);

      contenu.setStyleName("demo-popup-message");

      // bouton pour fermer le popup

      ClickListener listener = new ClickListener() {
          public void onClick(Widget sender)
          {
            hide();
          }
      };
      Button boutonFermer = new Button("Fermer", listener);
      SimplePanel holder = new SimplePanel();
      holder.add(boutonFermer);

      holder.setStyleName("demo-popup-footer");
      contenuPopupPanel.add(titre);
      contenuPopupPanel.add(contenu);
      contenuPopupPanel.add(holder);

      this.setWidget(contenuPopupPanel);
    }

    public void onModuleLoad() {
      final TestPopupPanel popup = new TestPopupPanel("Contenu du popup");

      ClickListener listener = new ClickListener()
      {
        public void onClick(Widget sender)
        {
          popup.center();
        }
    };
    Button bouton = new Button("Afficher", listener);

    RootPanel.get("app").add(bouton);
  }
```

Résultat :

```
.demo-popup
{
background-color        :    #ffc;
border                  :    1px solid #000;
}

.demo-popup-header
{
background-color        :    #ff0;
font-weight             :    bold;
border-bottom           :    1px solid #000;
padding                 :    10px;
}

.demo-popup-message
{
padding                 :    15px;
text-align              :    center;
}

.demo-popup-footer
{
padding                 :    5px;
text-align              :    right;
width                   :    100%;
}
```



Cet exemple est purement éducatif car une partie de ses fonctionnalités est implémentée dans le composant DialogBox.

L'instance du panneau n'a pas besoin d'être rattachée au RootPanel ou à tout autre composant.

Les méthodes center() ou show() permettent d'afficher le panneau et la méthode hide() permet de le masquer.

La taille du panneau est déterminée en fonction de la taille du composant qu'il contient.

89.8.17. La classe DialogBox

La suite de cette section sera développée dans une version future de ce document

89.8.18. La classe DisclosurePanel

La classe DisclosurePanel est un panneau composé de deux parties : un en-tête toujours visible et une partie principale qu'il est possible de masquer ou d'afficher en cliquant sur l'en-tête.

Ce composant est pratique lorsqu'il y a beaucoup de données à afficher.

Exemple :

```
    final DisclosurePanel panel = new DisclosurePanel("Cliquez pour ouvrir");

    panel.addEventHandler(new DisclosureHandler() {
      public void onClose(DisclosureEvent event) {
        panel.getHeaderTextAccessor().setText("Cliquez pour ouvrir");
      }

      public void onOpen(DisclosureEvent event) {
        panel.getHeaderTextAccessor().setText("Cliquez pour fermer");
      }
    });

    panel.add(new Image("images/logo_java.jpg"));

    panel.setWidth("300px");
    RootPanel.get("app").add(panel);
```

Il est possible de personnaliser l'en-tête en dérivant d'un panneau par exemple HorizontalPanel et en fournissant une instance de cette classe à la méthode setHeader().

89.8.19. La classe AbsolutePanel

Ce panneau permet le positionnement absolu (grâce à des coordonnées) des composants dans le panneau. Le rendu de ce panneau en HTML est un div.

La taille du panneau n'est pas automatiquement agrandie lors de l'ajout de composant hors de sa surface d'affichage définie par sa taille.

Exemple :

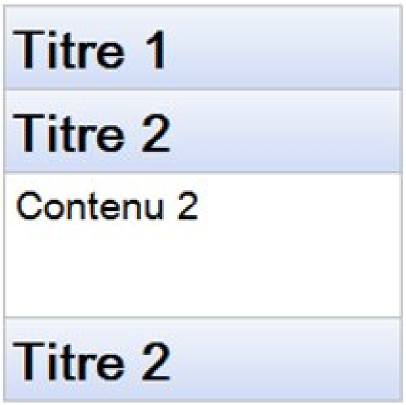
```
    AbsolutePanel panel = new AbsolutePanel();
    panel.setSize("250px", "100px");
    panel.setStyleName("monPanneau");
    Label label1 = new Label("Mon premier texte");
    panel.add(label1, 50, 30);
    Label label2 = new Label("Mon second texte");
    panel.add(label2, 65, 45);

    RootPanel.get("app").add(panel);
```

89.8.20. La classe StackPanel

La classe StackPanel encapsule un composant qui contient plusieurs sous-panneaux possédant un titre. Seul le contenu d'un des sous-panneaux est affiché.

Exemple :

```
    StackPanel panel = new StackPanel();
    Label label;
    label = new Label("Contenu 1");
    panel.add(label, "Titre 1", false);
    label = new Label("Contenu 2");
    panel.add(label, "Titre 2", false);
    label = new Label("Contenu 3");
    panel.add(label, "Titre 2", false);
    panel.setSize("200px", "200px");
    RootPanel.get("app").add(panel);
```

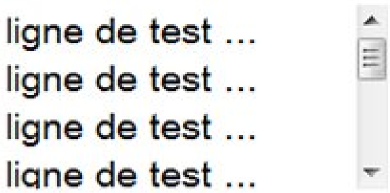


89.8.21. La classe ScrollPanel

La classe ScrollPanel est un panneau qui peut contenir un seul composant et qui possède une barre de défilement.

Exemple :

```
    StringBuffer sb = new StringBuffer("<p>");
    for (int i=0; i<10; i++) {
      sb.append("ligne de test ... <br>");
    }

    sb.append("</p>");
    ScrollPanel panel = new ScrollPanel(new HTML(sb.toString()));

    panel.setSize("200px", "100px");

    RootPanel.get("app").add(panel);
```


ligne de test ...
ligne de test ...
ligne de test ...
ligne de test ...

89.8.22. La classe FlexTable

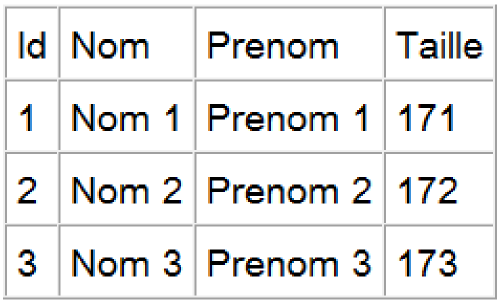
La classe FlexTable encapsule un panneau qui est une table dont le nombre de cellules peut varier pour chaque ligne. A sa création, une FlexTable n'a pas de taille explicite.

Ce panneau utilise une table HTML.

L'index de la première cellule vaut 0.

Exemple :

```
Personne[] personnes = new Personne[] {
    new Personne(1, "Nom 1", "Prenom 1", 171),
    new Personne(2, "Nom 2", "Prenom 2", 172),
    new Personne(3, "Nom 3", "Prenom 3", 173) };
FlexTable t = new FlexTable();

t.setTitle("Personnes");
t.setText(0, 0, "Id");
t.setText(0, 1, "Nom");
t.setText(0, 2, "Prenom");
t.setText(0, 3, "Taille");

t.setCellPadding(5);
t.setCellSpacing(0);

t.setBorderWidth(1);
for (int i = 0; i < 4; i++) {
  t.getColumnFormatter().addStyleName(i, "monPanneau");
}
for (int i = 0; i < personnes.length; i++) {
  Personne personne = personnes[i];
  t.setText(i + 1, 0, "" + personne.getId());
  t.setText(i + 1, 1, personne.getNom());
  t.setText(i + 1, 2, personne.getPrenom());
  t.setText(i + 1, 3, "" + personne.getTaille());
}

RootPanel.get("app").add(t);
```



Id	Nom	Prenom	Taille
1	Nom 1	Prenom 1	171
2	Nom 2	Prenom 2	172
3	Nom 3	Prenom 3	173

La méthode setColSpan() de la classe FlexCellFormater permet de fusionner deux cellules.

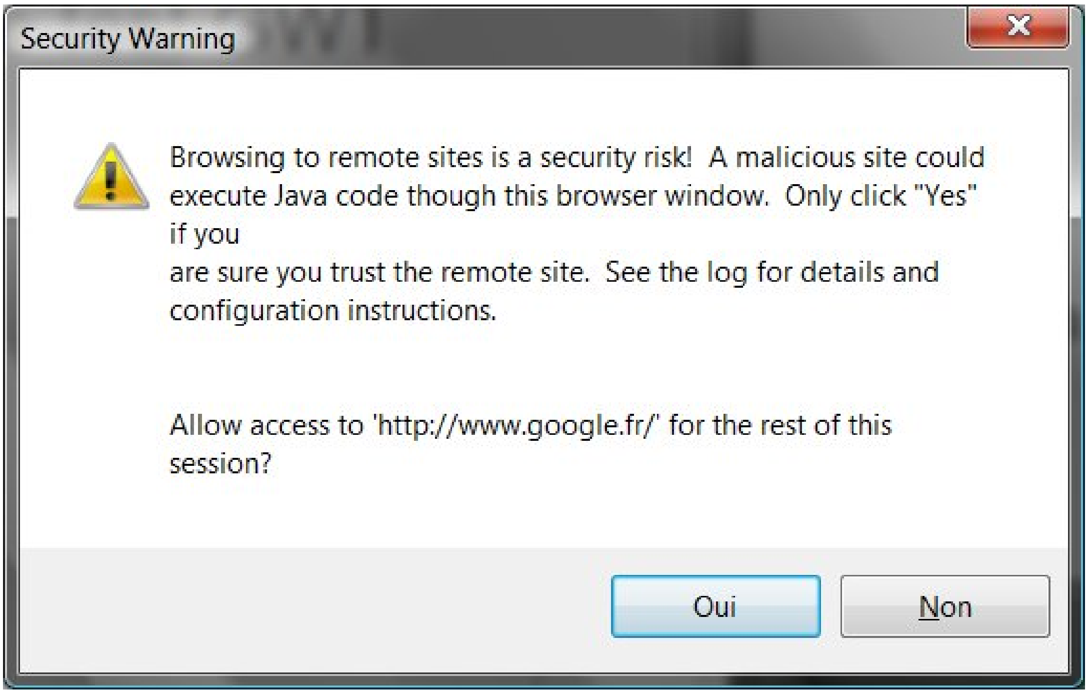
89.8.23. La classe Frame

Ce composant encapsule un IFrame HTML.

Exemple :

```
    Frame frame = new Frame("http://www.google.fr/");
    frame.setWidth("600px");
    frame.setHeight("350px");

    RootPanel.get("app").add(frame);
```

Les Iframes sont fréquemment utilisés pour effectuer des opérations depuis un site distant à l'insu de l'utilisateur. Lors de l'affichage de l'application utilisant un IFrame un message d'avertissement est affiché en demandant la confirmation de l'accès au site.



Security Warning

Browsing to remote sites is a security risk! A malicious site could execute Java code though this browser window. Only click "Yes" if you
are sure you trust the remote site. See the log for details and configuration instructions.

Allow access to 'http://www.google.fr/' for the rest of this session?

89.8.24. La classe Grid

Ce composant encapsule un tableau HTML : c'est donc une grille composée de cellules.

Il faut définir le nombre de cellules de la grille (nombre de colonnes et de lignes) avant de pouvoir insérer un composant dans une cellule.

L'ajout d'un composant dans une cellule se fait en utilisant la méthode setWidget().

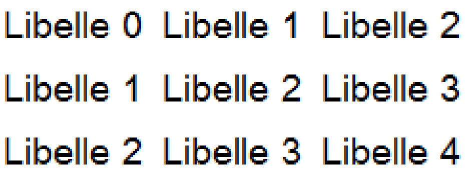
Exemple :

```
    Grid grille = new Grid(3, 3);
    grille.setSize("250px", "100px");

    for (int i = 0; i < 3 ; i++ ) {
      for (int j = 0; j < 3 ; j++ ) {
        grille.setWidget(i, j, new Label("Libelle "+(i+j)));
      }
    }

    RootPanel.get("app").add(grille);
```

La méthode setText() permet de facilement remplir une cellule de la grille avec du texte. L'exemple ci-dessous est identique au précédent.

Exemple :

```
    Grid grille = new Grid(3, 3);
    grille.setSize("250px", "100px");

    for (int i = 0; i < 3 ; i++ ) {
      for (int j = 0; j < 3 ; j++ ) {
        grille.setText(i, j, "Libelle "+(i+j));
      }
    }

    RootPanel.get("app").add(grille);
```



```
Libelle 0  Libelle 1  Libelle 2
Libelle 1  Libelle 2  Libelle 3
Libelle 2  Libelle 3  Libelle 4
```

Si la cellule à remplir est en dehors de la taille définie dans le constructeur alors une exception de type IndexOutOfBoundsException est levée.

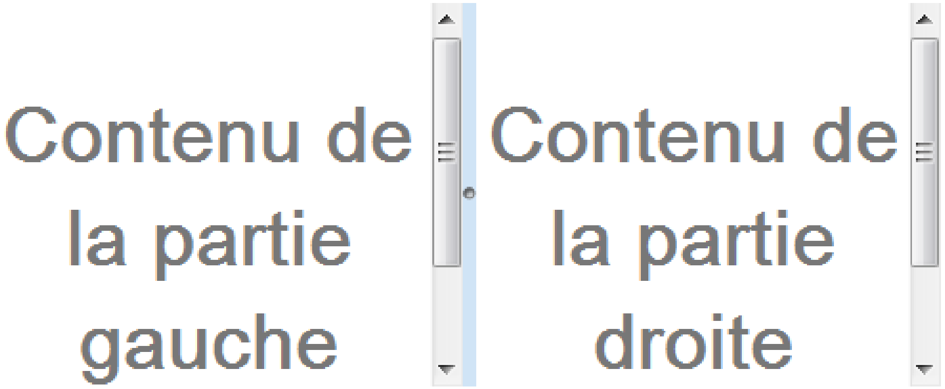
Il est possible de redimensionner le nombre de cellules de la grille grâce aux méthodes resize(), resizeColumns() et resizeRows().

89.8.25. La classe HorizontalSplitPanel

La classe HorizontalSplitPanel encapsule un panneau composé de deux cellules l'une à côté de l'autre. La taille des cellules est adaptable, l'une au détriment de l'autre. Si la taille d'une cellule est trop petite pour afficher l'ensemble de son contenu alors une barre de défilement est ajoutée.

Exemple :

```
    HorizontalSplitPanel panel = new HorizontalSplitPanel();
    panel.setSize("500px", "200px");
    panel.setLeftWidget(new HTML("<H1>Contenu de la partie gauche</H1>"));
    panel.setRightWidget(new HTML("<H1>Contenu de la partie droite</H1>"));

    RootPanel.get("app").add(panel);
```



Contenu de la partie gauche

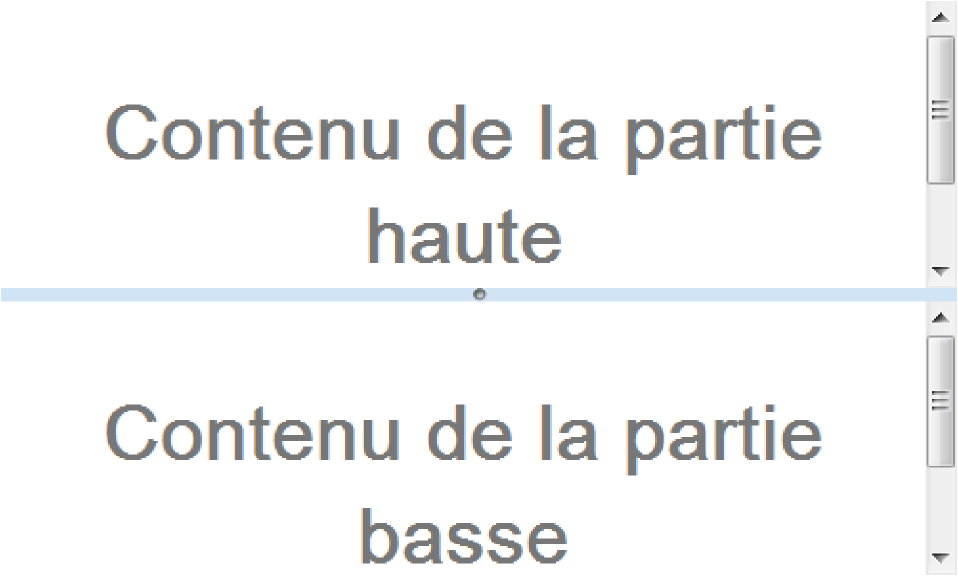
Contenu de la partie droite

89.8.26. La classe VerticalSplitPanel

La classe VerticalSplitPanel encapsule un panneau composé de deux cellules l'une au-dessus de l'autre. La taille des cellules est adaptable, l'une au détriment de l'autre.Si la taille d'une cellule est trop petite pour afficher l'ensemble de son contenu alors une barre de défilement est ajoutée.

Exemple :

```
   VerticalSplitPanel panel = new VerticalSplitPanel();
    panel.setSize("500px", "300px");
    panel.setTopWidget(new HTML("<H1>Contenu de la partie haute</H1>"));
    panel.setBottomWidget(new HTML("<H1>Contenu de la partie basse</H1>"));
    RootPanel.get("app").add(panel);
```



89.8.27. La classe HTMLTable

Cette classe abstraite est la classe mère des classes Grid et Flextable.

Les classes HTMLTable.CellFormatter, HTMLTable.ColumnFormatter et HTMLTable.RowFormatter permettent de formater respectivement le contenu d'une cellule, d'une colonne ou d'une ligne d'une table.

89.8.28. La classe LazyPanel

La classe LazyPanel est un composant qui est instancié au moment de son affichage.

Ceci peut permettre d'améliorer le temps de démarrage d'une application car les parties non affichées implémentées avec un LazyPanel ne sont plus instanciées au lancement de l'application mais uniquement au moment de leur affichage.

Pour utiliser un tel panneau, il faut hériter de la classe LazyPanel et redéfinir la méthode abstraite createWidget() en incluant le code de la création du rendu du panneau.

La méthode createWidget() sera invoquéee lorsque la méthode setVisible() du panneau sera utilisée.

Exemple :

```
  private static class TestLazyPanel extends LazyPanel {
    @Override
    protected Widget createWidget() {
      return new Label("Bonjour");
    }
  }

  public void onModuleLoad() {
    final Panel panel = new TestLazyPanel();

    panel.setVisible(false);
    PushButton bouton = new PushButton("Afficher");
    bouton.addClickHandler(new ClickHandler() {
      public void onClick(ClickEvent event) {
        panel.setVisible(true);
      }
    });

    RootPanel root = RootPanel.get("app");
    root.add(bouton);
    root.add(panel);
  }
```

89.9. La création d'éléments réutilisables

GWT permet de créer ses propres composants graphiques et permet aussi de créer des modules qui peuvent être utilisés par plusieurs projets.

89.9.1. La création de composants personnalisés

Le plus simple est de créer une classe qui hérite de la classe Composite mais il est aussi possible d'hériter d'un composant existant pour l'enrichir.

Le composant Composite permet de créer un nouveau composant autonome par composition. Il faut obligatoirement faire un appel à la méthode initWidget() à la fin du constructeur en lui passant en paramètre le panneau qui contient les éléments graphiques ou un composant.

La suite de cette section sera développée dans une version future de ce document

89.9.2. La création de modules réutilisables

Il est possible de développer un module qui va contenir des composants graphiques personnalisés, des classes dédiées ou des ressources, comme des images, afin de permettre sa réutilisation dans plusieurs applications GWT.

Ce module doit être packagé sous la forme d'un fichier .jar

La suite de cette section sera développée dans une version future de ce document

89.10. Les événements

Une application GWT est pilotée par des événements émis selon les actions de l'utilisateur sur les composants de l'application. La plupart des composants graphiques proposent l'émission d'événements en réaction aux actions de l'utilisateur.

La gestion des événements met en oeuvre des listeners d'une façon similaire à AWT ou Swing. Un listener est une interface qui doit être implémentée pour que les méthodes, appelées selon l'événement, effectuent les traitements à réaliser.

Des classes de types Adapter sont proposées afin de faciliter l'écriture de certains listeners : elles implémentent une interface de type Listener en définissant toutes les méthodes sans traitement. Il suffit de redéfinir la ou les méthodes requises en fonction des besoins.

Nom	Adapter	Méthodes
ChangeListener		void onChange(Widget sender)
ClickListener		void onClick(Widget sender)
EventListener		
FocusListener	FocusListenerAdapter	void onFocus(Widget sender) void onLostFocus(Widget sender)
KeyboardListener	KeyboardListenerAdapter	void onKeyDown(Widget sender, char keyCode, int modifiers) void onKeyPress(Widget sender, char keyCode, int modifiers) void onKeyUp(Widget sender, char keyCode, int modifiers)
LoadListener		void onError(Widget sender) void onLoad(Widget sender)
MouseListener	MouseListenerAdapter	void onMouseDown(Widget sender, int x, int y) void onMouseEnter(Widget sender) void onMouseLeave(Widget sender) void onMouseMove(Widget sender, int x, int y) void onMouseUp(Widget sender, int x, inty)
PopupListener		void onPopupClosed(PopupPanel sender, boolean autoClosed)
ScrollListener		void onScroll(Widget sender, int scrollLeft, int scrollTop)
TableListener		void onCellClicked(SourcedTableEvents sender, int row, int cell)
TabListener		void onBeforeTabSelected(SourcesTabEvents sender, int tabIndex) void onTabSelected(SourcesTabEvents sender, int tabIndex)
TreeListener		void onTreeItemSelected(TreeItem item) void onTreeItemStateChanged(TreeItem item)

Exemple :

```
Button b = new Button("Valider");
b.addClickListener(new ClickListener() {
  public void onClick(Widget sender) {
    // traitements lors du clic sur le bouton
  }
});
```

Exemple :

```
TextBox t = new TextBox();
t.addKeyboardListener(new KeyboardListenerAdapter() {
  public void onKeyPress(Widget sender, char keyCode, int modifiers) {
  // traitements lors de l'appui sur une touche
  }
});
```

89.11. JSNI

JSNI (JavaScript Native Interface) permet d'inclure du code JavaScript dans le code Java. Cette API a plusieurs utilités :

- appel à du code JavaScript non généré par GWT

- utiliser des bibliothèques de code JavaScript existantes

JSNI est utilisé par le compilateur pour fusionner le code JavaScript qu'il contient avec le code JavaScript généré à la compilation.

Le code JavaScript est inclus dans une méthode qualifiée avec le modificateur natif. Le code JavaScript lui-même est inclus entre les caractères /*-{ et }-*/;

Cette séquence de caractères à l'avantage d'être ignorée par le compilateur Java et exploitée par le compilateur de GWT.

Exemple :

```
public static native void alert(String msg) /*-{
    $wnd.alert(msg);
}-*/;
```

Il est possible de passer des paramètres qui seront utilisés par le code JavaScript.

Exemple :

```
public native int ajouter (int val1, int val2)
/*-{
var result = val1 + val2;
return result;
}-*/;
```

Il est possible de fournir en paramètre d'une méthode native des objets Java. Une syntaxe particulière permet d'utiliser ces objets dans le code JavaScript de la méthode : soit pour accéder à un champ ou pour invoquer une méthode.

Pour accéder à un champ d'un objet Java dans du code Javscript, il faut utiliser la syntaxe :

objet.@classe::champ

objet est la référence sur l'objet passé en paramètre
classe est le nom pleinement qualifié de la classe de l'objet
champ est le nom du champ à accéder

Pour utiliser une méthode d'un objet Java dans du code JavaScript, il faut utiliser la syntaxe :
objet.@classe::methode(signature)(parametres)

- objet est la référence sur l'objet passé en paramètre
- classe est le nom pleinement qualifié de la classe de l'objet
- méthode est le nom de la méthode à utiliser
- signature est la signature de la méthode
- paramètres est la liste des paramètres si nécessaire

Il est nécessaire de préciser la signature de la méthode car celle-ci peut être surchargée et cela permet ainsi de préciser la version qui doit être utilisée. La signature est précisée en suivant une convention spéciale pour chaque type utilisable.

Type dans la signature	Type Java
B	byte
S	short
I	int
J	long
F	float
D	double
C	char

Z	boolean
Lnom/pleinement/qualifie/classe;	nom.pleinement.qualifie.classe
[type	type[]

Certaines variables spécifiques sont définies dans JSNI.

variable	Rôle
$wnd	objet JavaScript Window
$doc	objet JavaScript Document

Exemple :

```
public class Alert {
  public static native void alert(String msg) /*-{
    $wnd.alert(msg);
  }-*/;
}

button1.addClickListener(new ClickListener() {
public void onClick(Widget sender) {
Alert.alert("clicked!" );
}
});
```

L'inconvénient de JNSI est que le code JavaScript n'est vérifiable qu'à l'exécution.

89.12. La configuration et l'internationalisation

GWT propose deux mécanismes pour internationaliser une application :

- étendre l'interface Messages pour inclure des fichiers de propriétés dans le code JavaScript généré à la compilation
- utiliser la classe Dictionary pour obtenir du texte contenant les traductions

Gwt propose deux mécanismes pour faciliter la mise en oeuvre de fonctionnalités de configuration de l'application :

- configuration statique : les données de configuration sont incluses à la compilation
- configuration dynamique : les données de configuration sont incluses à l'exécution de l'application

La configuration statique est mise en oeuvre grâce aux interfaces Constants ou Messages

Il faut étendre l'une ou l'autre de ces interfaces et définir des méthodes de type getter pour chaque propriété.

La configuration dynamique est mise en oeuvre grâce à la classe Dictionary.

89.12.1. La configuration

La suite de cette section sera développée dans une version future de ce document

89.12.2. L'internationalisation

L'internationalisation (I18N) permet de fournir le support de plusieurs langues pour une application. Même si son support n'est pas prévu, il peut être intéressant d'utiliser le mécanisme d'internationalisation pour centraliser les textes affichés par l'application. Ceci permet notamment de faciliter les vérifications orthographiques et grammaticales et la modification des textes sans altérer le code source.

Il faut définir un fichier de propriétés stocké dans le package client. Ce fichier contient sur chaque ligne une paire clé/valeur séparée par un caractère « = ».

Exemple : le fichier MonAppMessages.properties

```
menu1=Fichier
menu2=Editer
```

Il est possible de définir des paramètres dans les valeurs. Chacun de ces paramètres est numéroté à partir de 0. Un paramètre est défini en utilisant son numéro entouré par des accolades.

Exemple :

```
erreur=La valeur saisie doit être comprise entre {0} et {1}
```

Il faut définir un fichier de propriétés pour chaque langue proposée par l'application. Le nom de fichier doit être identique au fichier de propriétés initial suivi par un caractère underscore et le code langue. Les clés doivent être identiques et les valeurs doivent contenir leurs traductions.

Exemple : le fichier MonAppMessages_en.properties

```
menu1=File
menu2=Edit
```

Ensuite, créer une interface qui porte le nom du fichier de propriétés et qui hérite de l'interface com.google.gwt.i18n.client.Messages. Il faut définir une méthode qui renvoie une chaîne de caractères pour chaque clé définie dans le fichier de propriétés. Le nom de cette méthode doit correspondre exactement au nom de chaque clé.

Exemple :

```
package fr.jmdoudoux.dejgwt.client;

import com.google.gwt.i18n.client.Messages;

public interface MonAppMessages extends Messages {
  String menu1();
  String menu2();
}
```

Si des paramètres sont définis dans la valeur, il faut ajouter autant de paramètres à la méthode correspondante.

Il faut modifier le fichier de configuration de l'application en ajoutant un tag <inherits name="com.google.gwt.i18n.I18N"/> pour indiquer à GWT d'ajouter le support de l'internationalisation.

Il faut aussi ajouter un tag <extend-property> possédant un attribut name dont la valeur doit être égale à la Locale et un attribut values dont la valeur doit contenir le ou les codes langues supportés par l'application.

Exemple :

```
<module>
  <!-- Inherit the core Web Toolkit stuff. -->
  <inherits name='com.google.gwt.user.User' />
  <inherits name="com.google.gwt.i18n.I18N"/>
  <stylesheet src="MonApp.css" />
  <!-- Specify the app entry point class. -->
  <entry-point class='fr.jmdoudoux.dejgwt.client.MonApp' />

  <extend-property name="locale" values="en"/>
</module>
```

Dans le code de l'application, il faut utiliser la méthode GWT.Create() en fournissant en paramètre la classe correspondant à l'interface définie.

Exemple :

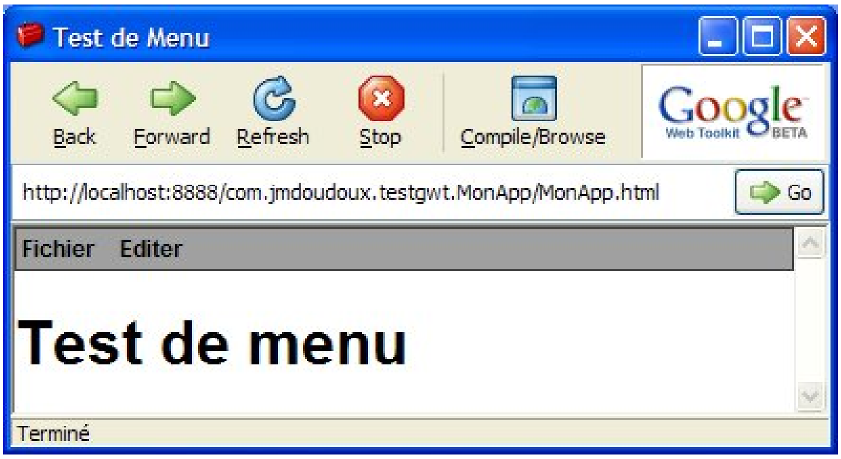
```
MonAppMessages messages = (MonAppMessages) GWT.create(MonAppMessages.class);
```

Il suffit d'utiliser l'objet instancié pour obtenir la valeur dont la clé correspond au nom de la méthode invoquée.

Exemple :

```
    // menu.addItem("menu1", menu1);
    menu.addItem(messages.menu1(), menu1);
    // menu.addItem("menu2", menu2);
    menu.addItem(messages.menu2(), menu2);<
```

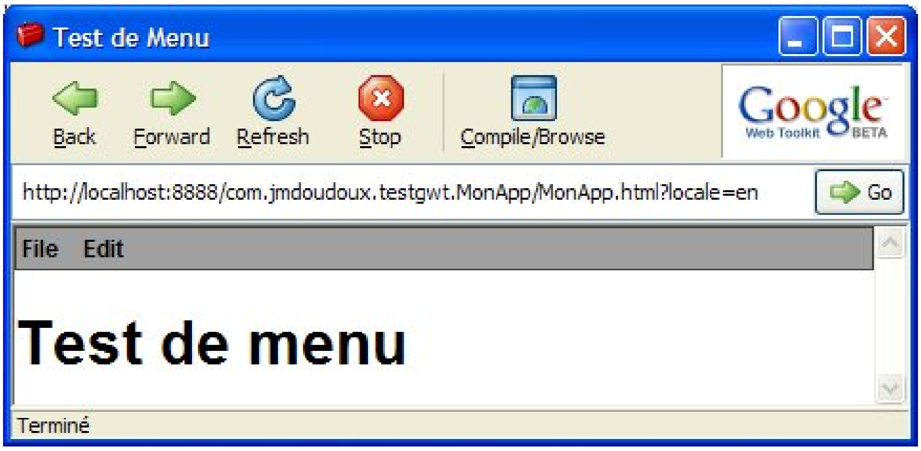
Au lancement de l'application, la langue par défaut est utilisée.



http://localhost:8888/com.jmdoudoux.testgwt.MonApp/MonApp.html

Test de menu

Pour afficher l'application dans une autre langue, il faut ajouter dans l'url le paramètre locale avec, comme valeur, le code langue désiré.



http://localhost:8888/com.jmdoudoux.testgwt.MonApp/MonApp.html?locale=en

Test de menu

89.13. L'appel de procédures distantes (Remote Procedure Call)

GWT propose plusieurs solutions pour permettre l'appel de traitements côté serveur :

- soumission de requêtes http
- utilisation du composant XmlHttpRequest
- utilisation du mode RPC de GWT

GWT propose des fonctionnalités pour permettre l'appel de procédures sur le serveur et ainsi mettre en oeuvre des fonctionnalités de type AJAX.

Le code côté serveur peut être réalisé avec n'importe quel langage proposant un support du traitement des requêtes HTTP. Ceci inclus Java EE notamment en utilisant des servlets. Une solution reposant sur Java côté serveur est cependant la plus facile à mettre en oeuvre.

En utilisant Java, GWT fournit deux classes qui encapsulent l'utilisation de l'objet JavaScript XMLHttpRequest :

- RequestBuilder : cette classe permet d'effectuer une requête sur le serveur et d'obtenir une réponse
- GWT-RPC : c'est un mécanisme dédié de GWT qui permet l'échange d'objets Java entre le client et le serveur en utilisant un format propre à GWT

Comme dans toute application de type Ajax, il est important d'indiquer à l'utilisateur que des traitements sont en cours : cela peut se faire par exemple à l'aide d'une zone de texte spéciale, d'une image animée, d'un changement de la forme du curseur, ...

89.13.1. GWT-RPC

GWT permet aux applications de communiquer avec le serveur au travers de son propre mécanisme d'appels de type RPC. Ce mécanisme assure la sérialisation des objets qui sont échangés entre la partie cliente en JavaScript et la partie serveur écrite en Java. Cette sérialisation n'est pas réalisée au travers d'un standard tel que XML, JSON, SOAP ou XML-RPC, mais elle met en oeuvre son propre format.

Les appels réalisés par l'application sont de type asynchrone.

Cette solution utilise, côté serveur, des servlets qui héritent de la classe RemoteServiceServlet.

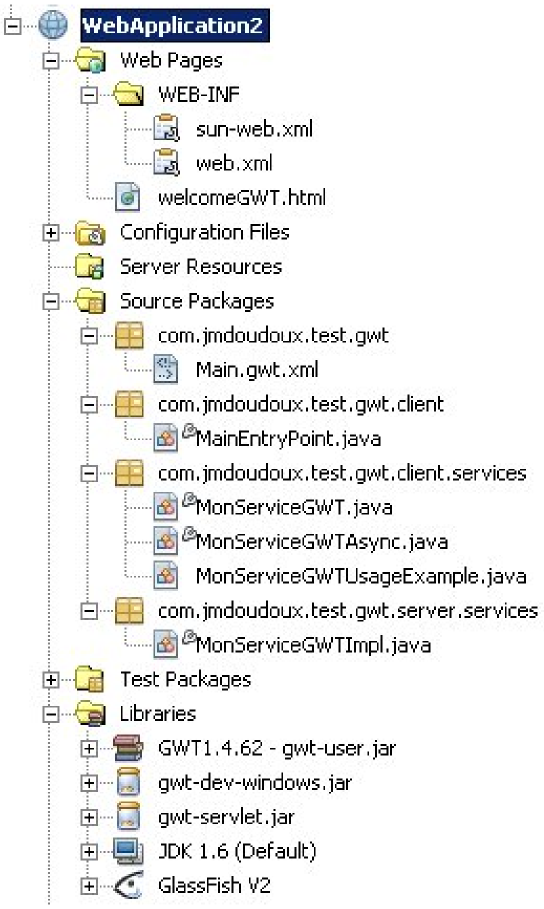
L'implémentation d'un service nécessite plusieurs étapes :

- Créer une interface qui héritent de com.google.gwt.user.client.rpc.RemoteService dans le package client de l'application
- Créer une interface pour l'appel asynchrone du service dans le package client de l'application
- Créer une servlet qui hérite de com.google.gwt.server.rpc.RemoteServiceServlet et qui implémente l'interface du service
- Déclarer la servlet dans le fichier web.xml de la webapp

89.13.1.1. Une mise oeuvre avec un exemple simple

Cette section va développer une petite application qui demande à l'utilisateur de saisir son prénom, invoque un service sur le serveur et affiche le message de salutation retourné par le service.

Exemple de projet dans Netbeans :



Dans l'exemple, les classes et interfaces sont regroupées dans deux sous-packages services au niveau de la partie client et de la partie serveur. Ceci n'est pas une obligation mais permet un meilleur découpage des sources.

GWT propose un mécanisme qui permet l'échange d'objets Java entre le client et le serveur. Pour mettre en oeuvre ce mécanisme il est nécessaire de définir trois entités :

Entités	localisation	Rôle
interface du service	Client et serveur	Décrit le service : la signature des méthodes
classe du service	Serveur	Implémentation du service
interface asynchrone du service	Client	Permet l'appel au service de façon asynchrone

L'interface du service est définie dans le sous-packages client/services. Elle hérite impérativement de l'interface com.google.gwt.user.client.rpc.RemoteService et va contenir les méthodes utilisables.

Exemple :

```
package fr.jmdoudoux.dej.gwt.client.services;

import com.google.gwt.user.client.rpc.RemoteService;

public interface MonServiceGWT extends RemoteService {
    public String saluer(String s);
}
```

L'interface pour l'appel asynchrone du service est définie dans le sous-package client/services de l'application. Par convention, elle possède le même nom que l'interface du service suffixé par Async.

Elle doit définir la méthode qui permettra l'invocation asynchrone de la méthode correspondante sur le serveur. Cette méthode doit avoir les caractéristiques suivantes :

- ne doit rien retourner,
- avoir les même paramètres que la méthode correspondante définie dans l'interface du service,
- avoir un paramètre supplémentaires de type com.google.gwt.user.client.rpc.AsyncCallback,
- ne déclarer aucune exception

Exemple :

```
package fr.jmdoudoux.dej.gwt.client.services;

import com.google.gwt.user.client.rpc.AsyncCallback;

public interface MonServiceGWTAsync {

    public void saluer(String s, AsyncCallback asyncCallback);
}
```

Pour la partie serveur, il faut définir une servlet dans le sous-package server/services qui hérite de com.google.gwt.server.rpc.RemoteServiceServlet et qui implémente l'interface du service.

Par convention, la classe de cette servlet possède le même nom que l'interface du service suffixé par Impl puisque c'est l'implémentation concrète du service

Exemple :

```
package fr.jmdoudoux.dej.gwt.server.services;

import com.google.gwt.user.server.rpc.RemoteServiceServlet;
import fr.jmdoudoux.dej.gwt.client.services.MonServiceGWT;

public class MonServiceGWTImpl extends RemoteServiceServlet implements
        MonServiceGWT {

    public String saluer(String s) {
        return "Bonjour " + s;
    }
}
```

Remarque : pour des raisons de simplicité, dans l'exemple ci-dessus, la servlet implémente les traitements du service. Il serait préférable de découpler la servlet qui hérite de RemoteServiceServlet et implémente l'interface du service. Pour cela, définir un objet métier de type POJO qui implémente l'interface du service : chaque méthode de l'interface de la servlet se charge d'invoquer la méthode correspondante de l'objet métier.

La servlet RemoteServiceServlet héritée pour l'implémentation du service propose quelques méthodes utiles.

La méthode getThreadLocalRequest() permet d'obtenir un objet de type HttpServletRequest qui encapsule la requête http.

La méthode getThreadLocalResponse() permet d'obtenir un objet de type HttpServletResponse qui encapsule la réponse http.

En résumé, voici une synthèse des entités à créer pour un service

Entités	Hérite de	Rôle
MonServiceGWT	RemoteService	Interface qui décrit le service. Utilisé côté client et serveur
MonServiceGWTAsync		Interface pour l'appel asynchrone. Son nom est composé par convention du nom de l'interface du service suffixé par Async. Contient toutes les méthodes de l'interface du service, sans valeur de retour, sans exception et avec les même paramètres plus un dernier paramètre de type AsyncCallback

Entités	Hérite de	Rôle
		Utilisé côté client uniquement
MonServiceGWTImpl	RemoteServiceServlet	Implémentation concrète de l'interface du service Utilisé côté serveur uniquement

L'utilisation de GWT-RPC passe par l'objet JavaScript XMLHttpRequest. Les données sont donc échangées entre le client et le serveur sous un mécanisme propre à GWT : les objets doivent donc être sérialisés côté client et désérialisés côté serveur. Côté serveur, c'est la classe RemoteServiceServlet qui automatise cette tâche.

Il faut déclarer la servlet dans le fichier web.xml de la webapp.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <servlet>
      <servlet-name>MonServiceGWT</servlet-name>
      <servlet-class>fr.jmdoudoux.dej.gwt.server.services.MonServiceGWTImpl</servlet-class>
    </servlet>
    <servlet-mapping>
      <servlet-name>MonServiceGWT</servlet-name>
      <url-pattern>/fr.jmdoudoux.dej.gwt.Main/services/monservicegwt</url-pattern>
    </servlet-mapping>
    <session-config>
      <session-timeout>30</session-timeout>
    </session-config>
    <welcome-file-list>
      <welcome-file>welcomeGWT.html</welcome-file>
    </welcome-file-list>
</web-app>
```

Pour utiliser le serveur Tomcat embarqué avec GWT en mode hosted, il faut déclarer la servlet dans le fichier de configuration du module pour que le client puisque invoquer le service.

Pour déclarer la servlet du service dans le fichier de configuration du module, il faut utiliser un tag servlet ayant deux attributs :

- path : chemin de mapping associé à la servlet
- class : nom pleinement qualifié de la classe de la servlet

Exemple :

```
<servlet path="/services/monservicegwt"
       class="fr.jmdoudoux.dej.gwt.server.services/MonServiceGWTImpl"/>
```

GWT va automatiquement référencer la servlet, dans le conteneur Tomcat, avec le chemin fourni pour permettre son invocation par le client lors de son exécution dans le mode hosted.

GWT ne propose que des échanges asynchrones avec le serveur puisqu'ils utilisent l'objet JavaScript XMLHttpRequest.

L'invocation d'un service RPC dans la partie cliente de l'application nécessite plusieurs étapes :

1. obtenir une instance de l'interface d'appel asynchrone du service en invoquant la méthode create() de la classe GWT
2. caster l'instance vers le type ServiceDefTarget
3. invoquer la méthode setServiceEntryPoint() en lui passant en paramètre l'url de la servlet qui implémente le service

4. créer une instance de la classe AsyncCallBack() qui implémente les traitements à réaliser en cas de succès et d'échec de l'appel du service
5. invoquer la méthode de l'interface appel asynchrone en lui passant en paramètre les paramètres d'appel du service et l'instance de type callback

Pour obtenir une instance de l'interface d'appel asynchrone du service, il faut invoquer la méthode create() de la classe GWT et caster le retour vers le type de l'interface asynchrone : cette instance sera le proxy qui permettra l'appel du service distant.

Pour préciser l'url d'appel du service, il faut caster l'instance de ce service en un objet de type ServiceDefTarget et invoquer sa méthode setServiceEntryPoint() en lui passant en paramètre l'url.

Le service doit être hébergé sur le même domaine et le même port du serveur que celui qui a fourni la page HTML au navigateur.

Le plus simple est de créer une méthode statique qui renvoie l'instance du type de l'interface asynchrone

Exemple :

```
    public static MonServiceGWTAsync getService() {
        MonServiceGWTAsync service = (MonServiceGWTAsync) GWT.create(MonServiceGWT.class);
        ServiceDefTarget endpoint = (ServiceDefTarget) service;
        String moduleRelativeURL = GWT.getModuleBaseURL() +
          "services/monservicegwt";
        endpoint.setServiceEntryPoint(moduleRelativeURL);
        return service;
    }
```

Il faut créer une instance d'un objet qui implémente l'interface com.google.gwt.client.rpc.asyncCallback. Le plus simple est de définir une classe anonyme interne. L'interface AsyncCallback définit deux méthodes

- void onFailure(Throwable e) : callback invoqué lors de l'échec de l'invocation du service
- void onSuccess(Object result) : callback invoqué lors de la réussite de l'invocation du service

Dans la méthode onSuccess(), il faut caster l'objet passé en paramètre qui contient le résultat de l'appel vers l'objet du type adéquat.

Exemple :

```
        // Instanciation d'un callback asynchrone pour traiter la réponse
        final AsyncCallback callback = new AsyncCallback() {

            public void onSuccess(Object result) {
                lblMessage.setText((String) result);
            }

            public void onFailure(Throwable caught) {
                lblMessage.setText("Echec de la communication : " + caught.getMessage());
            }
        };
```

Pour invoquer le service, il faut obtenir une instance du proxy et invoquer la méthode voulue en lui passant les paramètres à fournir au service et l'instance de l'interface AsynCallback qui prend en charge le retour de l'appel.

Exemple :

```
                        getService().saluer(text.getText(), callback);
```

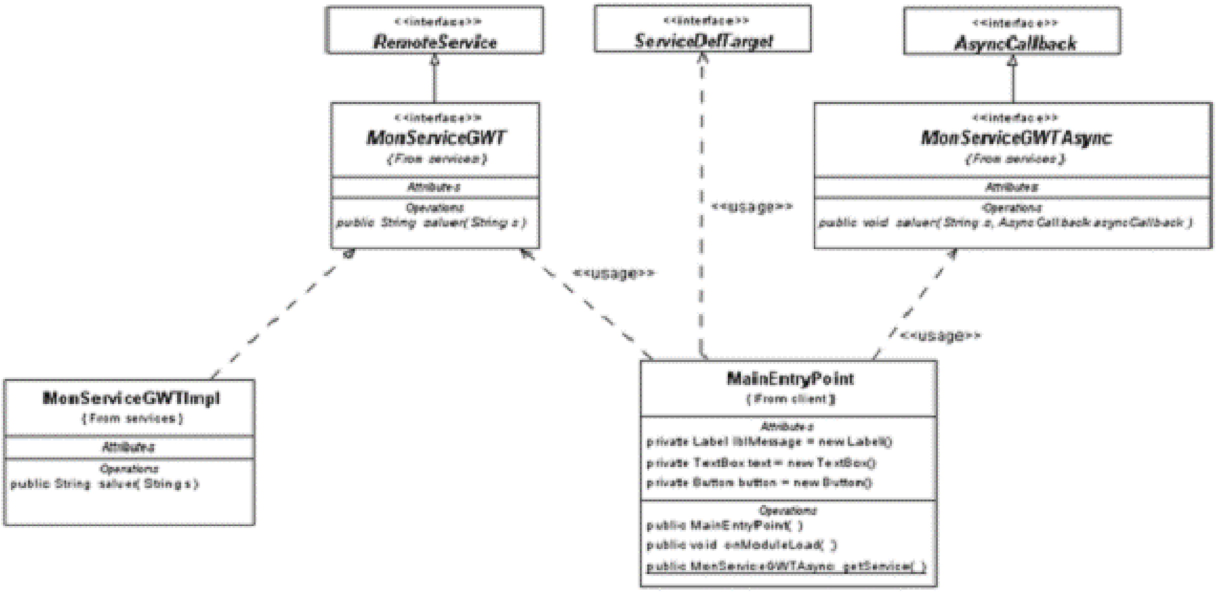
Il n'est pas possible de fournir null comme callbak même si aucun retour n'est attendu suite à l'appel au service.

L'exemple complet du code de l'application permet à l'utilisateur de saisir son prénom, d'invoquer le service et d'afficher le résultat de l'appel.

Exemple :

```
package fr.jmdoudoux.dej.gwt.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.core.client.GWT;
import com.google.gwt.user.client.rpc.AsyncCallback;
import com.google.gwt.user.client.rpc.ServiceDefTarget;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.ClickListener;
import com.google.gwt.user.client.ui.FlowPanel;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.Panel;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.TextBox;
import com.google.gwt.user.client.ui.Widget;
import fr.jmdoudoux.dej.gwt.client.services.MonServiceGWT;
import fr.jmdoudoux.dej.gwt.client.services.MonServiceGWTAsync;

public class MainEntryPoint implements EntryPoint {

    private Label lblMessage = new Label();
    private TextBox text = new TextBox();
    private Button button = new Button();

    public MainEntryPoint() {
    }

    public void onModuleLoad() {
        text.setText("");
        button.setText("Saluer");

        // Instanciation d'un callback asynchrone pour traiter la réponse
        final AsyncCallback callback = new AsyncCallback() {

            public void onSuccess(Object result) {
                lblMessage.setText((String) result);
            }

            public void onFailure(Throwable caught) {
                lblMessage.setText("Echec de la communication : " + caught.getMessage());
            }
        };

        button.addClickListener(new ClickListener() {

            public void onClick(Widget sender) {
                // invocation du service
                getService().saluer(text.getText(), callback);
            }
        });

        Panel main = new FlowPanel();
        RootPanel.get().add(main);
        main.add(text);
        main.add(button);
        main.add(lblMessage);
    }

    public static MonServiceGWTAsync getService() {
        MonServiceGWTAsync service = (MonServiceGWTAsync) GWT.create(MonServiceGWT.class);
        ServiceDefTarget endpoint = (ServiceDefTarget) service;
        String moduleRelativeURL = GWT.getModuleBaseURL() + "services/monservicegwt";
        endpoint.setServiceEntryPoint(moduleRelativeURL);
        return service;
    }
}
```

Le diagramme de classe ci-dessous décrit l'ensemble des classes et interfaces utilisées.



Il n'y a pas de relation au sens POO entre l'interface du service et l'interface d'appel asynchrone du service.

Pour un meilleur découpage du code source, il est possible de définir une classe dédiée qui implémente l'interface AsyncCallback

exemple : le code de l'application

Exemple :

```
package fr.jmdoudoux.dej.gwt.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.core.client.GWT;
import com.google.gwt.user.client.rpc.AsyncCallback;
import com.google.gwt.user.client.rpc.ServiceDefTarget;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.ClickListener;
import com.google.gwt.user.client.ui.FlowPanel;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.Panel;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.TextBox;
import com.google.gwt.user.client.ui.Widget;
import fr.jmdoudoux.dej.gwt.client.services.MonServiceGWT;
import fr.jmdoudoux.dej.gwt.client.services.MonServiceGWTAsync;

public class MainEntryPoint implements EntryPoint {

    private Label lblMessage = new Label();
    private TextBox text = new TextBox();
    private Button button = new Button();

    public MainEntryPoint() {
    }

    public void onModuleLoad() {
        text.setText("");
        button.setText("Saluer");

        button.addClickListener(new ClickListener() {

            public void onClick(Widget sender) {
                // invocation du service
                getService().saluer(text.getText(), new MonAsyncCallback(lblMessage));
            }
        });

        Panel main = new FlowPanel();
        RootPanel.get().add(main);
        main.add(text);
        main.add(button);
        main.add(lblMessage);
    }

    public static MonServiceGWTAsync getService() {
        MonServiceGWTAsync service = (MonServiceGWTAsync) GWT.create(MonServiceGWT.class);
        ServiceDefTarget endpoint = (ServiceDefTarget) service;
        String moduleRelativeURL = GWT.getModuleBaseURL() + "services/monservicegwt";
        endpoint.setServiceEntryPoint(moduleRelativeURL);
        return service;
    }
}
```

exemple : la classe MonAsyncCallback

Exemple :

```
package fr.jmdoudoux.dej.gwt.client;

import com.google.gwt.user.client.rpc.AsyncCallback;
import com.google.gwt.user.client.ui.Label;

public class MonAsyncCallback implements AsyncCallback {

    Label label;

    public MonAsyncCallback(Label label) {
        this.label = label;
    }

    public void onSuccess(Object result) {
        label.setStyleName("message");
        label.setText((String) result);
    }

    public void onFailure(Throwable caught) {
        label.setStyleName("erreur");
        label.setText("Echec de la communication");
    }
}
```

Il faut se souvenir dans les développements qu'une application JavaScript est monothread. Plusieurs callbacks ne peuvent donc pas être exécutés en simultané.

89.13.1.2. La transmission d'objets lors des appels aux services

Tout objet qui sera utilisé dans un échange de type GWT-RPC doit implémenter l'interface com.google.gwt.user.client.rpc.IsSerializable ou l'interface java.io.Serializable (Depuis GWT 1.4). L'usage de l'interface Serializable est recommandé car cela permet à l'objet de rester indépendant de GWT.

Les attributs déclarés transient ne seront pas sérialisés lors des échanges.

Il est nécessaire que l'objet possède un constructeur sans argument.

Exemple :

```
package fr.jmdoudoux.dej.gwt.client.vo;

import java.io.Serializable;

public class Personne implements Serializable {

    private String nom;
    private String prenom;
    private int taille;

    public Personne() {
    }

    public Personne(String nom, String prenom, int taille) {
        this.nom = nom;
        this.prenom = prenom;
        this.taille = taille;
    }

    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }

    public String getPrenom() {
        return prenom;
    }

    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }

    public int getTaille() {
        return taille;
    }

    public void setTaille(int taille) {
        this.taille = taille;
    }
}
```

Le service peut alors utiliser l'objet en paramètre d'entrée ou de sortie.

Exemple :

```
package fr.jmdoudoux.dej.gwt.client.services;

import com.google.gwt.user.client.rpc.RemoteService;
import fr.jmdoudoux.dej.gwt.client.vo.Personne;

public interface PersonneService extends RemoteService{
    public Personne obtenirParId(int id);

    public Personne[] obtenirToutes();
}
```

L'interface d'appel asynchrone du service ne fait pas référence au bean.

Exemple :

```
package fr.jmdoudoux.dej.gwt.client.services;

import com.google.gwt.user.client.rpc.AsyncCallback;

public interface PersonneServiceAsync {

    public abstract void obtenirParId(int id, AsyncCallback asyncCallback);

    public abstract void obtenirToutes(AsyncCallback asyncCallback);
}
```

L'implémentation du service peut utiliser toutes les API nécessaires à ses traitements, notamment celles relatives aux accès à une base de données pour extraire les informations requises. Dans l'exemple ci-dessous, les données sont simplement instanciées.

Exemple :

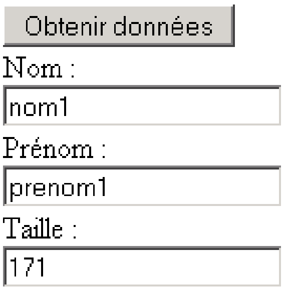
```
package fr.jmdoudoux.dej.gwt.server;

import com.google.gwt.user.server.rpc.RemoteServiceServlet;
import fr.jmdoudoux.dej.gwt.client.services.PersonneService;
import fr.jmdoudoux.dej.gwt.client.vo.Personne;

public class PersonneServiceImpl extends RemoteServiceServlet implements
        PersonneService {

    public Personne obtenirParId(int id) {
        return new Personne("nom"+id,"prenom"+id,170+id);
    }

    public Personne[] obtenirToutes() {
        Personne[] resultat = new Personne[5];
        for (int i = 1 ; i < 6 ; i++) {
          resultat[i] = new Personne("nom"+i,"prenom"+i,170+i);
        }
        return resultat;
    }
}
```

Dans l'application, il suffit de caster le résultat de l'invocation du service vers le type du bean pour obtenir une instance du bean contenant les données transmises par le serveur.

Exemple :

```
package fr.jmdoudoux.dej.gwt.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.core.client.GWT;
import com.google.gwt.user.client.rpc.AsyncCallback;
import com.google.gwt.user.client.rpc.ServiceDefTarget;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.ClickListener;
import com.google.gwt.user.client.ui.FlowPanel;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.Panel;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.TextBox;
import com.google.gwt.user.client.ui.Widget;
import fr.jmdoudoux.dej.gwt.client.services.PersonneService;
import fr.jmdoudoux.dej.gwt.client.services.PersonneServiceAsync;
import fr.jmdoudoux.dej.gwt.client.vo.Personne;

public class MainEntryPoint implements EntryPoint {

    private Label lblMessage = new Label();
    private Label lblNom = new Label("Nom : ");
    private Label lblPrenom = new Label("Prénom : ");
    private Label lblTaille = new Label("Taille : ");
    private TextBox textNom = new TextBox();
    private TextBox textPrenom = new TextBox();
    private TextBox textTaille = new TextBox();
    private Button button = new Button("Obtenir données");

    public MainEntryPoint() {
    }

    public void onModuleLoad() {

        // Instanciation d'un callback asynchrone pour traiter la réponse
        final AsyncCallback callback = new AsyncCallback() {

            public void onSuccess(Object result) {
                Personne personne = (Personne) result;
                textNom.setText(personne.getNom());
                textPrenom.setText(personne.getPrenom());
                textTaille.setText(""+personne.getTaille());
            }

            public void onFailure(Throwable caught) {
                lblMessage.setStyleName("erreur");
                lblMessage.setText("Echec de la communication");
            }
        };

        button.addClickListener(new ClickListener() {

            public void onClick(Widget sender) {
                // invocation du service
                getPersonneService().obtenirParId(1, callback);
                // getService().saluer(text.getText(), new MonAsyncCallback(lblMessage));
            }
        });

        Panel main = new FlowPanel();
        RootPanel.get().add(main);
        main.add(button);
        main.add(lblNom);
        main.add(textNom);
        main.add(lblPrenom);
        main.add(textPrenom);
        main.add(lblTaille);
        main.add(textTaille);
        main.add(lblMessage);
    }

    public static PersonneServiceAsync getPersonneService() {
        PersonneServiceAsync service = (PersonneServiceAsync)
          GWT.create(PersonneService.class);
        ServiceDefTarget endpoint = (ServiceDefTarget) service;
        String moduleRelativeURL = GWT.getModuleBaseURL() + "services/personneservice";
        endpoint.setServiceEntryPoint(moduleRelativeURL);
        return service;
    }
}
```

Le résultat de l'application après un appui sur le bouton et la réception de la réponse du serveur est le suivant :



Nom :
nom1
Prénom :
prenom1
Taille :
171

89.13.1.3. L'invocation périodique d'un service

Pour rafraichir périodiquement des données grâce à un appel serveur, il faut combiner l'utilisation d'un appel RPC et d'une instance de la classe Timer.

Dans l'exemple ci-dessous, la méthode obtenirValeur() d'un service renvoie un nombre aléatoire compris entre 0 et 1000.

Exemple :

```
package fr.jmdoudoux.dej.gwt.server.services;

import com.google.gwt.user.server.rpc.RemoteServiceServlet;
import fr.jmdoudoux.dej.gwt.client.services.MonServiceGWT;
```

```
public class MonServiceGWTImpl extends RemoteServiceServlet implements
        MonServiceGWT {

    public int obtenirValeur() {
        double valeur = Math.random() * 1000;
        return (int) Math.round(valeur);
    }
}
```

Dans l'IHM de l'application, un Timer est défini : son rôle est d'invoquer toutes les secondes la méthode du service et d'afficher le résultat.

Exemple :

```
package fr.jmdoudoux.dej.gwt.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.core.client.GWT;
import com.google.gwt.user.client.Timer;
import com.google.gwt.user.client.rpc.AsyncCallback;
import com.google.gwt.user.client.rpc.ServiceDefTarget;
import com.google.gwt.user.client.ui.FlowPanel;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.Panel;
import com.google.gwt.user.client.ui.RootPanel;
import fr.jmdoudoux.dej.gwt.client.services.MonServiceGWT;
import fr.jmdoudoux.dej.gwt.client.services.MonServiceGWTAsync;
import fr.jmdoudoux.dej.gwt.client.services.PersonneService;
import fr.jmdoudoux.dej.gwt.client.services.PersonneServiceAsync;

public class MainEntryPoint implements EntryPoint {

    private Label lblMessage = new Label();

    public MainEntryPoint() {
    }

    public void onModuleLoad() {

        Timer timer = new Timer() {

            AsyncCallback callback = new AsyncCallback() {

                public void onSuccess(Object result) {
                    lblMessage.setText(" valeur = "+ result);
                }

                public void onFailure(Throwable caught) {
                    lblMessage.setText("Echec de la communication");
                }
            };

            public void run() {
                getService().obtenirValeur(callback);
            }
        };
        timer.scheduleRepeating(1000);

        Panel main = new FlowPanel();
        RootPanel.get().add(main);
        main.add(lblMessage);
    }

    public static MonServiceGWTAsync getService() {
        MonServiceGWTAsync service = (MonServiceGWTAsync) GWT.create(MonServiceGWT.class);
        ServiceDefTarget endpoint = (ServiceDefTarget) service;
        String moduleRelativeURL = GWT.getModuleBaseURL() + "services/monservicegwt";
        endpoint.setServiceEntryPoint(moduleRelativeURL);
        return service;
    }
}
```

89.13.2. L'objet RequestBuilder

La suite de cette section sera développée dans une version future de ce document

89.13.3. JavaScript Object Notation (JSON)

La classe JSONObject encapsule un message au format JSON. Pour l'utiliser, il suffit de créer une instance de cette classe et d'utiliser la méthode put() pour ajouter une propriété en fournissant en paramètre son nom et sa valeur. La méthode toString() permet d'obtenir le message sous la forme d'une chaîne de caractères.

La suite de cette section sera développée dans une version future de ce document

89.14. La manipulation des documents XML

GWT propose un parseur XML reposant sur DOM pour permettre l'analyse ou la création de documents XML. GWT utilise le parseur du navigateur ce qui permet d'avoir de bonnes performances lors de son utilisation.

Pour utiliser les fonctionnalités de manipulation de documents XML de GWT, il faut ajouter dans la configuration du module un tag <inherits> ayant un attribut name avec la valeur « com.google.gxt.xml.XML ».

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<module>
  <inherits name="com.google.gwt.user.User"/>
  <inherits name ="com.google.gwt.xml.XML"/>
  <entry-point class="fr.jmdoudoux.dej.gwt.client.MainEntryPoint"/>
</module>
```

Pour la mise en oeuvre de l'API DOM, GWT propose plusieurs classes regroupées dans le package com.google.gwt.xml.client.

Exemple :

```
package fr.jmdoudoux.dej.gwt.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.user.client.Window;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.ClickListener;
import com.google.gwt.user.client.ui.FlowPanel;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.Panel;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.TextBox;
import com.google.gwt.user.client.ui.Widget;
import com.google.gwt.xml.client.Document;
import com.google.gwt.xml.client.Element;
import com.google.gwt.xml.client.Node;
import com.google.gwt.xml.client.NodeList;
import com.google.gwt.xml.client.XMLParser;

public class MainEntryPoint implements EntryPoint {
    private Label lblMessage = new Label();
    private Label lblNom = new Label("Nom : ");
    private Label lblPrenom = new Label("Prénom : ");
    private Label lblTaille = new Label("Taille : ");
    private TextBox textNom = new TextBox();
    private TextBox textPrenom = new TextBox();
    private TextBox textTaille = new TextBox();
    private Button button = new Button("Afficher données");

    public MainEntryPoint() {
    }

    public void onModuleLoad() {

        button.addClickListener(new ClickListener() {

            public void onClick(Widget sender) {
                Document doc = XMLParser.parse("<personne><nom>nom1</nom>"
                  +"<prenom>prenom1</prenom>"
                  +"<taille>170</taille></personne>");
                Element root = doc.getDocumentElement();
                NodeList children = root.getChildNodes();
                for (int i = 0 ; i < children.getLength(); i++) {
                    Node node = children.item(i);
                  Window.alert("node name="+node.getNodeName()
                    +" value="+node.getFirstChild().getNodeValue());
                }

                textNom.setText(children.item(0).getFirstChild().getNodeValue());
                textPrenom.setText(children.item(1).getFirstChild().getNodeValue());
                textTaille.setText(children.item(2).getFirstChild().getNodeValue());
            }
        });

        Panel main = new FlowPanel();
        RootPanel.get().add(main);
        main.add(button);
        main.add(lblNom);
        main.add(textNom);
        main.add(lblPrenom);
        main.add(textPrenom);
        main.add(lblTaille);
        main.add(textTaille);
        main.add(lblMessage);
    }
}
```

89.15. La gestion de l'historique sur le navigateur

Les applications utilisant AJAX modifient seulement les portions nécessaires d'une page sans la recharger entièrement : ce type d'applications est nommé SPI (Single Page Interface).

Il en résulte pour l'utilisateur une modification de ses habitudes avec le bouton Back du navigateur. Avec des applications web n'utilisant pas Ajax, l'utilisateur peut toujours revenir à la page précédente en utilisant ce bouton.

La suite de cette section sera développée dans une version future de ce document

89.16. Les tests unitaires

Un support des tests unitaires automatisés est proposé par GWT avec JUnit. La version de JUnit supportée est la 3.

Pour écrire un cas de test, il faut écrire une classe qui hérite de la classe GWTTestCase. Il faut définir la méthode getModuleName() et définir les tests en écrivant la ou les méthodes commençant par test.

GWT propose un script pour générer un fichier de tests unitaires et deux scripts pour exécuter ces tests.

Exemple :

```
D:\gwt-windows-1.3.3>junitCreator
-junit D:/api/junit3.8.1/junit.jar -module fr.jmdoudoux.dejgwt.MonApp
-out MonAppProjet
fr.jmdoudoux.dejgwt.client.MonAppTests
Created
directory MonAppProjet\test\com\jmdoudoux\testgwt\client
Created file
MonAppProjet\test\com\jmdoudoux\testgwt\client\MonAppTests.java
Created
file MonAppProjet\MonAppTests-hosted.cmd
Created
file MonAppProjet\MonAppTests-web.cmd
```

La syntaxe de junitcreator est la suivante :

```
JUnitCreator -junit pathToJUnitJar -module moduleName [-eclipse projectName]
[-out dir] [-overwrite] [-ignore] className
```

Le script junitcreator possède plusieurs paramètres :

- -junit : chemin complet du fichier junit.jar (obligatoire)
- -module : nom du module GWT (obligatoire)
- -eclipse : nom du projet Eclipse dans lequel sera créé un fichier de configuration pour lancer les tests
- -out : répertoire dans lequel les fichiers seront créés (par défaut le répertoire courant)
- -overwrite : remplacement des fichiers existants
- -ignore : ne pas remplacer les fichiers existants
- className : nom pleinement qualifié de la classe de tests générée

Le fichier fr.jmdoudoux.dejgwt.client.MonAppTests.java dans le répertoire test est créé pour servir de base aux tests.

junitCreator -junit D:/api/junit3.8.1/junit.jar -module fr.jmdoudoux.dejgwt -eclipse MonAppProjet -out MonAppProjet fr.jmdoudoux.dejgwt.client.MonAppTests

Exemple :

```
package fr.jmdoudoux.dejgwt.client;

import com.google.gwt.junit.client.GWTTestCase;

/**
 * GWT JUnit tests must extend GWTTestCase.
 */
public class MonAppTests extends GWTTestCase {
  /**
   * Must refer to a valid module that sources this class.
   */
  public String getModuleName() {
    return "fr.jmdoudoux.dejgwt.monApp";
  }
  /**
   * Add as many tests as you like.
   */
  public void testSimple() {
    assertTrue(true);
  }
}
```

Deux scripts sont créés pour exécuter les tests unitaires :

- MonAppTests-web.cmd
- MonAppTests-hosted.cmd

Pour mettre en oeuvre Junit dans un module GWT, il faut :

- ajouter le fichier junit.jar au classpath
- ajouter une entrée dans le fichier de configuration appname.gwt.xml
  <inherits name="com.google.gwt.junit.JUnit"/>
- créer une classe qui hérite de la classe com.google.gwt.junit.client.GWTTestCase
- réécrire la méthode getModuleName() pour quelle renvoie le nom pleinement qualifié du module à tester
- écrire les cas de tests sous la forme de méthodes

Dans une méthode de test, il est possible de :

- Tester et modifier l'état d'un composant
- Simuler des événements
- Appeler des traitements côté serveur
- Créer des composants

La suite de cette section sera développée dans une version future de ce document

89.17. Le déploiement d'une application

La suite de cette section sera développée dans une version future de ce document

89.18. Des composants tiers

Les composants fournis en standard avec GWT sont assez basiques. Pour pouvoir développer une IHM avec des composants plus riches, il est nécessaire d'utiliser une des bibliothèques tierces proposées notamment par la communauté open source.

Il existe deux formes de composants tiers :

- native : les composants sont écrits en GWT
- wrapper : les composants encapsulent du code JavaScript existant en utilisant JSNI

89.18.1. GWT-Dnd

GWT-Dnd est une bibliothèque qui propose un support pour le drag and drop dans les applications GWT.

89.18.2. MyGWT

MyGWT est une bibliothèque open source de composants pour GWT.

89.18.3. GWT-Ext

GWT-Ext est un wrapper de la bibliothèque JavaScript Ext 2.0. Ext qui est une bibliothèque de composants JavaScript très riche proposant des composants graphiques évolués (grilles avec tri, pagination et filtre, treeview, ...)

Le site officiel du projet est à l'url http://gwt-ext.com

Une démo est consultable à l'url http://gwt-ext.com/demo/ : elle permet de visualiser toute la richesse de la bibliothèque et propose pour chaque exemple de visualiser le code source correspondant.

Produit	Version utilisée
GWT	1.4.60
GWT-Ext	2.0.1
ext	2.1

89.18.3.1. Installation et configuration

Pour utiliser GWT-Ext, il faut :

- télécharger la bibliothèque GWT-Ext et décompresser le contenu de l'archive dans un répertoire du système.
- ajouter le fichier gwtext.jar au classpath du projet GWT.
- télécharger la bibliothèque JavaScript ext. La version 2.0.2 utilisée est diffusée sous licence LGPL.
- décompresser l'archive téléchargée dans un répertoire du système.
- créer un sous-répertoire js/ext dans le sous-répertoire public du projet GWT.

Ensuite, il faut copier les ressources suivantes dans le sous-répertoire ext créé précédemment :

- Les répertoires : adapter et resources
- Les fichiers : ext-all.js, ext-all-debug.js, ext-core.js et ext-core-debug.js

Enfin, il est nécessaire d'ajouter la bibliothèque dans la configuration du module en ajoutant :

- un tag inherits avec l'attribut name ayant pour valeur com.gwtext.GwtExt
- deux tags script avec l'attribut src ayant pour valeurs js/ext/adapter/ext/ext-base.js et js/ext/ext-all.js
- un tag stylesheet avec l'attribut src ayant pour valeur /ext/resources/css/ext-all.css

Exemple :

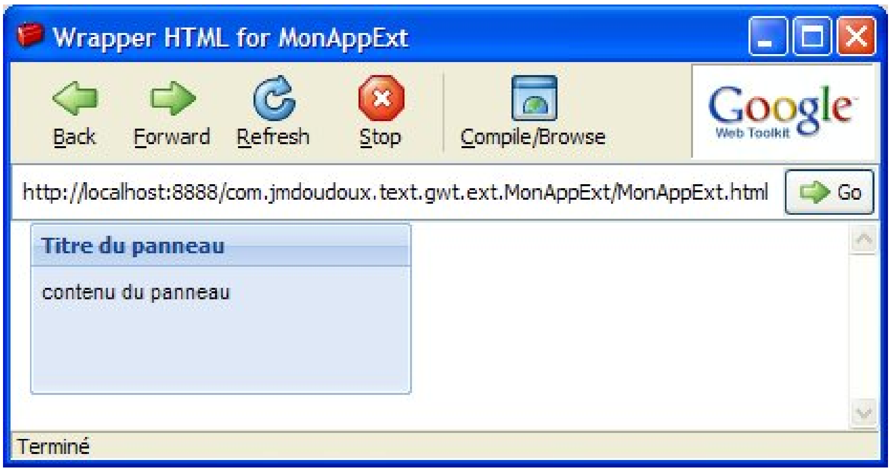
```
<module>

        <!-- Inherit the core Web Toolkit stuff.                  -->
        <inherits name='com.google.gwt.user.User'/>
      <inherits name='com.gwtext.GwtExt' />

        <!-- Specify the app entry point class.                   -->
        <entry-point class='fr.jmdoudouxtext.gwt.ext.client.MonAppExt'/>

        <stylesheet src="js/ext/resources/css/ext-all.css" />
        <script src="js/ext/adapter/ext/ext-base.js" />
        <script src="js/ext/ext-all.js" />
</module>
```

89.18.3.2. La classe Panel

La classe Panel encapsule un panneau qui possède un titre et un contenu.

Exemple :

```
package fr.jmdoudouxtext.gwt.ext.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.user.client.ui.RootPanel;
import com.gwtext.client.widgets.Panel;

public class MonAppExt implements EntryPoint {
  public void onModuleLoad() {
    Panel mainPanel = new Panel() {
      {
         setTitle("Titre du panneau");
         setHeight(90);
         setWidth(200);
         setFrame(true);
         setHtml("<p>Contenu du panneau</p>");
         setStyle("margin: 10px 10px 10px 10px;");
      }
   };

   RootPanel.get().add(mainPanel);
  }
}
```


Wrapper HTML for MonAppExt
Back
Forward
Refresh
Stop
Compile/Browse
Google Web Toolkit
http://localhost:8888/com.jmdoudoux.text.gwt.ext.MonAppExt/MonAppExt.html
Go
Titre du panneau
contenu du panneau
Terminé

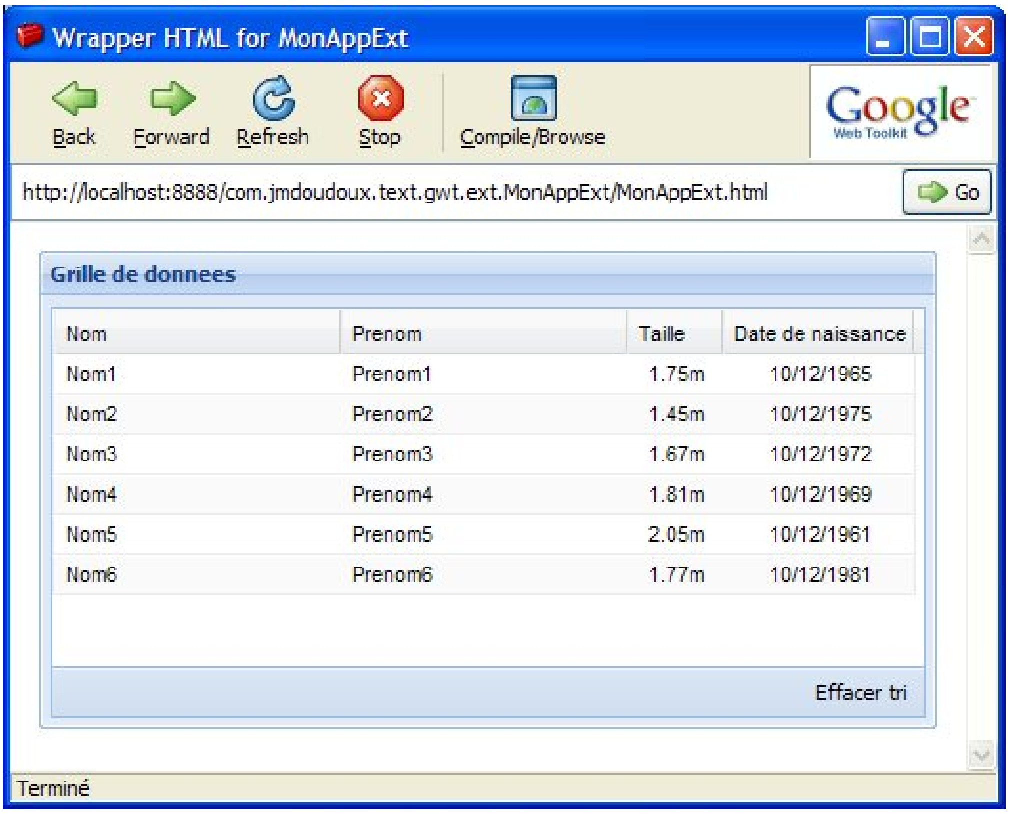
89.18.3.3. La classe GridPanel

La classe GridPanel encapsule un panneau avec un titre et une grille de données.

Exemple :

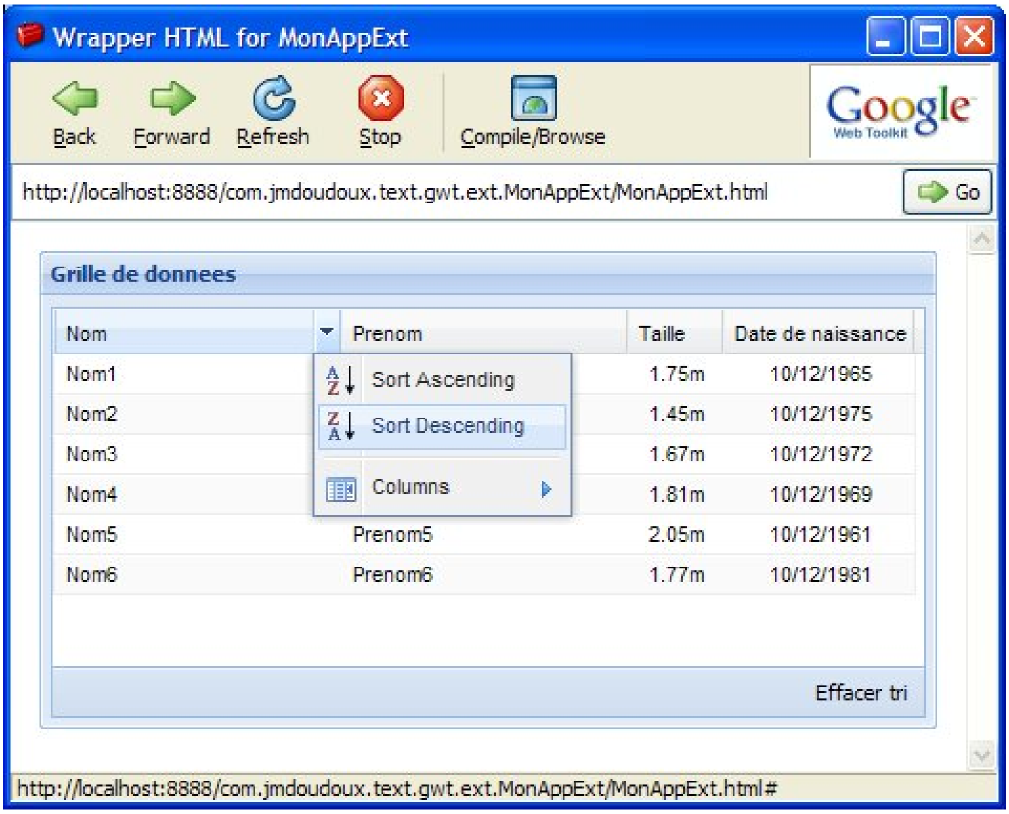
```
package fr.jmdoudouxtext.gwt.ext.client;

import java.util.Date;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.i18n.client.DateTimeFormat;
import com.google.gwt.user.client.ui.RootPanel;
import com.gwtext.client.core.EventObject;
import com.gwtext.client.data.ArrayReader;
import com.gwtext.client.data.DateFieldDef;
import com.gwtext.client.data.FieldDef;
import com.gwtext.client.data.FloatFieldDef;
import com.gwtext.client.data.MemoryProxy;
import com.gwtext.client.data.Record;
import com.gwtext.client.data.RecordDef;
import com.gwtext.client.data.Store;
import com.gwtext.client.data.StringFieldDef;
import com.gwtext.client.widgets.Button;
import com.gwtext.client.widgets.Panel;
import com.gwtext.client.widgets.Toolbar;
import com.gwtext.client.widgets.ToolbarButton;
import com.gwtext.client.widgets.event.ButtonListenerAdapter;
import com.gwtext.client.widgets.grid.CellMetadata;
import com.gwtext.client.widgets.grid.ColumnConfig;
import com.gwtext.client.widgets.grid.ColumnModel;
import com.gwtext.client.widgets.grid.GridPanel;
import com.gwtext.client.widgets.grid.Renderer;

public class MonAppExt implements EntryPoint {
```

```
private static final DateTimeFormat dateFormatter = DateTimeFormat.getFormat("M/d/y");

public void onModuleLoad() {
  final GridPanel grille = new GridPanel();

  Panel panneau = new Panel();
  panneau.setBorder(false);
  panneau.setPaddings(15);

  RecordDef recordDef = new RecordDef(
          new FieldDef[]{
                  new StringFieldDef("nom"),
                  new StringFieldDef("prenom"),
                  new FloatFieldDef("taille"),
                  new DateFieldDef("datenais", "d/m/Y")
          }
  );


  Object[][] donnees =  new Object[][]{
    new Object[]{"Nom1", "Prenom1", new Double(1.75), "13/10/1965"},
    new Object[]{"Nom2", "Prenom2", new Double(1.45), "13/10/1975"},
    new Object[]{"Nom3", "Prenom3", new Double(1.67), "13/10/1972"},
    new Object[]{"Nom4", "Prenom4", new Double(1.81), "13/10/1969"},
    new Object[]{"Nom5", "Prenom5", new Double(2.05), "13/10/1961"},
    new Object[]{"Nom6", "Prenom6", new Double(1.77), "13/10/1981"}
  };
  MemoryProxy proxy = new MemoryProxy(donnees);

  ArrayReader reader = new ArrayReader(recordDef);
  Store store = new Store(proxy, reader);
  store.load();
  grille.setStore(store);


  ColumnConfig[] colonnes = new ColumnConfig[]{
          new ColumnConfig("Nom", "nom",
            150, true, null, "nom"),
          new ColumnConfig("Prenom", "prenom",
            150, true, null, "prenom"),
          new ColumnConfig("Taille", "taille",
            50, true, new Renderer() {
              public String render(Object value, CellMetadata cellMetadata,
                Record record, int rowIndex, int colNum, Store store) {
                return "<div>" + value + "m</div>";
            }}),
          new ColumnConfig("Date de naissance", "datenais",
            100, true, new Renderer() {
              public String render(Object value, CellMetadata cellMetadata,
                Record record, int rowIndex, int colNum, Store store) {
                Date date = (Date)value;
                return "<div>" +
                  dateFormatter.format(date) + "</div>";
          }})
  };

  ColumnModel columnModel = new ColumnModel(colonnes);
  grille.setColumnModel(columnModel);

  grille.setFrame(true);
  grille.setStripeRows(true);

  grille.setHeight(250);
  grille.setWidth(470);
  grille.setTitle("Grille de donnees");

  Toolbar bottomToolbar = new Toolbar();
  bottomToolbar.addFill();
  bottomToolbar.addButton(new ToolbarButton("Effacer tri",
    new ButtonListenerAdapter() {
      public void onClick(Button button, EventObject e) {
          grille.clearSortState(true);
      }
    }));
    grille.setBottomToolbar(bottomToolbar);

    panneau.add(grille);

    RootPanel.get().add(panneau);
  }
}
```


Wrapper HTML for MonAppExt
Back
Forward
Refresh
Stop
Compile/Browse
Google Web Toolkit
http://localhost:8888/com.jmdoudoux.text.gwt.ext.MonAppExt/MonAppExt.html
Go
Grille de donnees
Effacer tri
Terminé

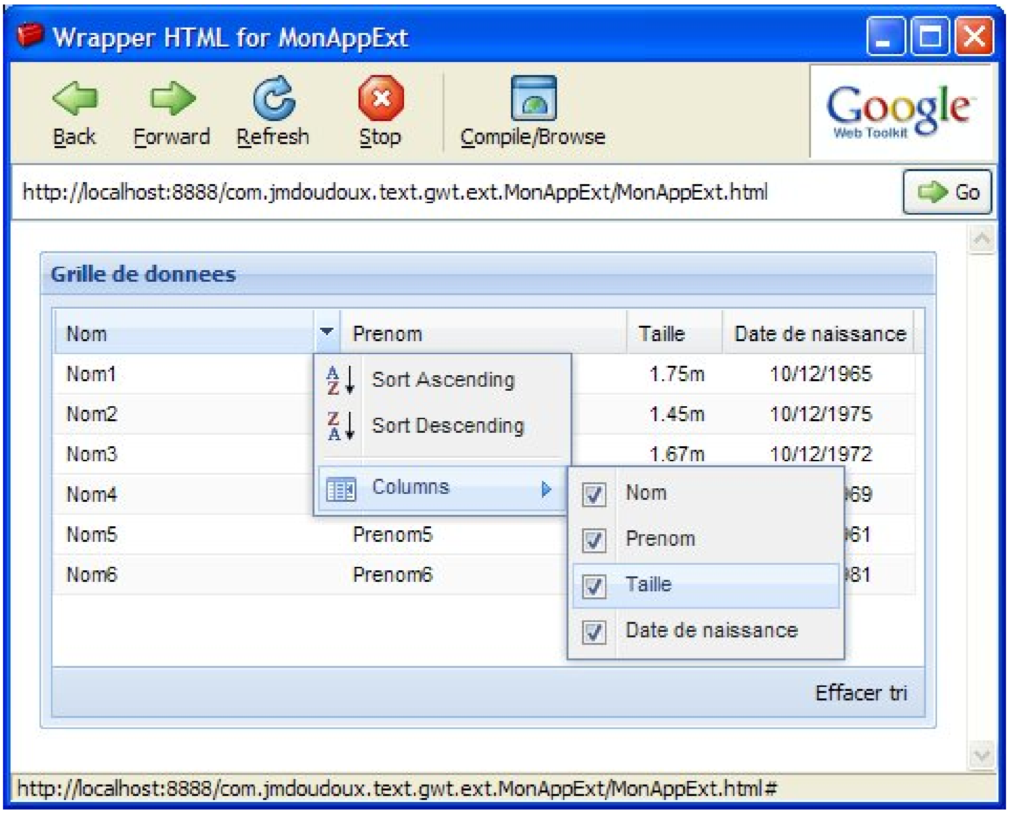
Nom	Prenom	Taille	Date de naissance
Nom1	Prenom1	1.75m	10/12/1965
Nom2	Prenom2	1.45m	10/12/1975
Nom3	Prenom3	1.67m	10/12/1972
Nom4	Prenom4	1.81m	10/12/1969
Nom5	Prenom5	2.05m	10/12/1961
Nom6	Prenom6	1.77m	10/12/1981

Le composant offre en standard des fonctionnalités avancées comme le tri des données d'une colonne.


Wrapper HTML for MonAppExt
Back
Forward
Refresh
Stop
Compile/Browse
Google Web Toolkit
http://localhost:8888/com.jmdoudoux.text.gwt.ext.MonAppExt/MonAppExt.html
Go
Grille de donnees
Sort Ascending
Sort Descending
Columns
Effacer tri
http://localhost:8888/com.jmdoudoux.text.gwt.ext.MonAppExt/MonAppExt.html#

Nom	Prenom	Taille	Date de naissance
Nom1		1.75m	10/12/1965
Nom2		1.45m	10/12/1975
Nom3		1.67m	10/12/1972
Nom4		1.81m	10/12/1969
Nom5	Prenom5	2.05m	10/12/1961
Nom6	Prenom6	1.77m	10/12/1981

Le composant permet aussi de sélectionner les colonnes qui seront affichées.



http://localhost:8888/com.jmdoudoux.text.gwt.ext.MonAppExt/MonAppExt.html

Grille de donnees

Nom	Prenom	Taille	Date de naissance
Nom1		1.75m	10/12/1965
Nom2		1.45m	10/12/1975
Nom3		1.67m	10/12/1972
Nom4			69
Nom5	Prenom5		61
Nom6	Prenom6		81

http://localhost:8888/com.jmdoudoux.text.gwt.ext.MonAppExt/MonAppExt.html#

89.19. Les ressources relatives à GWT

Le site officiel de GWT : https://www.gwtproject.org/

Le guide de démarrage : https://www.gwtproject.org/gettingstarted.html

Le guide du développeur : https://www.gwtproject.org/doc/latest/DevGuide.html

Partie 13 : Le développement d'applications avec Spring

Partie 13 :

Développement d'applications avec Spring

Cette partie couvre le framework Spring et quelques-unes des nombreuses extensions qui composent son portfolio. Spring est une solution open source complète pour le développement d'applications reposant sur un conteneur qui implémente le motif de conception inversion de contrôle et sur l'utilisation de l'AOP. Pour les principales fonctionnalités, Spring propose sa solution mais facilite aussi l'intégration de frameworks ou de solutions existantes.

Cette partie contient les chapitres suivants :

- Spring : ce chapitre est une présentation générale de Spring
- Spring Core : ce chapitre détaille la configuration et la mise en oeuvre du conteneur Spring qui gère le cycle de vie des beans
- La mise en oeuvre de l'AOP avec Spring : présente la mise en oeuvre de l'AOP avec Spring
- La gestion des transactions avec Spring : ce chapitre présente les différentes possibilités de gestion des transactions dans une application Spring
- Spring et JMS : ce chapitre couvre la mise en oeuvre de JMS dans Spring
- Spring et JMX : ce chapitre détaille la façon dont Spring facilite la mise en oeuvre de JMX

90. Spring

Chapitre 90

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Spring est un socle pour le développement d'applications, principalement d'entreprises mais pas obligatoirement. Il fournit de nombreuses fonctionnalités parfois redondantes ou qui peuvent être configurées ou utilisées de plusieurs manières : ceci laisse le choix au développeur d'utiliser la solution qui lui convient le mieux et/ou qui répond aux besoins.

Spring est ainsi un des frameworks les plus répandus dans le monde Java : sa popularité a grandie au profit de la complexité de Java EE notamment pour ses versions antérieures à la version 5 mais aussi grâce à la qualité et la richesse des fonctionnalités qu'il propose :

- son coeur reposant sur un conteneur de type IoC assure la gestion du cycle de vies des beans et l'injection des dépendances
- l'utilisation de l'AOP
- des projets pour faciliter l'intégration avec de nombreux projets open source ou API de Java EE

Spring était un framework applicatif à ses débuts mais maintenant c'est une véritable plate-forme composée du framework Spring, de projets qui couvrent de nombreux besoins et de middlewares.

Spring permet une grande flexibilité dans les fonctionnalités et les projets utilisés dans une application. Il est par exemple possible d'utiliser le conteneur Spring pour gérer de façon basique les beans sans utiliser l'AOP. Par contre, certains projets et certaines fonctionnalités ont des dépendances avec d'autres projets.

Spring est associé à la notion de conteneur léger (lightweight container) par opposition aux conteneurs lourds que sont les serveurs d'applications Java EE.

Le site officiel du framework Spring est à l'url https://spring.io

Ce chapitre contient plusieurs sections :

♦ Le but et les fonctionnalités proposées par Spring
♦ L'historique de Spring
♦ Spring Framework
♦ Les projets du portfolio Spring
♦ Les avantages et les inconvénients de Spring
♦ Spring et Java EE

90.1. Le but et les fonctionnalités proposées par Spring

Le but de Spring est de faciliter et de rendre productif le développement d'applications, particulièrement les applications d'entreprises.

Spring propose de nombreuses fonctionnalités de base pour le développement d'applications :

- un conteneur léger implémentant le design pattern IoC pour la gestion des objets et de leurs dépendances en offrant des fonctionnalités avancées concernant la configuration et l'injection automatique. Un de ses points forts est d'être non intrusif dans le code de l'application tout en permettant l'assemblage d'objets faiblement couplés.
- une gestion des transactions par déclaration offrant une abstraction du gestionnaire de transactions sous-jacent
- faciliter le développement des DAO de la couche de persistance en utilisant JDBC, JPA, JDO ou une solution open source comme Hibernate, iBatis, ... et une hiérarchie d'exceptions
- un support pour un usage interne à Spring (notamment dans les transactions) ou personnalisé de l'AOP qui peut être mis en oeuvre avec Spring AOP pour les objets gérés par le conteneur et/ou avec AspectJ
- faciliter la testabilité de l'application
- ...

Spring favorise l'intégration avec de nombreux autres frameworks notamment ceux de type ORM ou web.

Une application typique utilisant Spring est généralement structurée en trois couches :

- la couche présentation : interface homme machine
- la couche service : interface métier avec mise en oeuvre de certaines fonctionnalités (transactions, sécurité, ...)
- la couche accès aux données : recherche et persistance des objets du domaine

Spring est utilisé pour créer et injecter les objets requis de la couche précédente.

Les objets du domaine sont utilisés dans les échanges entre ces couches. Les objets du domaine ne sont pas créés ni gérés par Spring : ils sont instanciés directement en utilisant l'opérateur new. Il peut donc exister de nombreuses instances uniques des objets du domaine.

90.2. L'historique de Spring

Le framework Spring a été initialement développé par Rod Johnson et Juergen Holler.

Spring a connu plusieurs versions :

- Spring 1.0 : mars 2004
- Spring 1.1 : septembre 2004
- Spring 1.2 : mai 2005
- Spring 2.0 : octobre 2006
- Spring 2.5 : novembre 2007
- Spring 3.0 : décembre 2009
- Spring 3.1 : courant 2011

Spring 1.0 implémente les fonctionnalités de base du framework :

- le conteneur qui implémente le motif de conception IoC
- le développement orienté POJO
- l'AOP par déclaration
- le support de JDBC, ORM et frameworks Web
- la configuration XML basée sur une DTD

Spring 1.2

- support de JMX
- support JDO 2, Hibernate 3, TopLink
- support de JCA CCI, JDBC Rowset
- déclaration des transactions avec @Transactional

Spring 2.0 apporte de nombreuses nouveautés :

- le support et l'utilisation d'AspectJ
- la configuration XML basée sur un schéma XML
- des simplifications de la configuration notamment avec des namespaces dédiés (beans, tx, aop, lang, util, jee, p)

- les Message Driven POJO
- les annotations @Repository, @Configurable

Spring 2.5 apporte de nombreuses nouveautés pour faciliter sa configuration :

- l'ajout de nouveaux namespaces (context, jms) avec de nouveaux tags
- l'enrichissement des namespaces existants (jee, aop)
- l'ajout d'annotations concernant le cycle de vie des beans (@Service, @Component, @Controller), autowiring (@Autowired, @Qualifier, @Required), la gestion des transactions (@Transactional) et support des annotations standards de Java 5 (@PostConstruct, @PreDestroy, @Resource)
- les tests d'intégration reposant sur Junit 4 et des annotations (@ContextConfiguration, @TestExecutionListeners, @BeforeTransaction, @AfterTransaction)

Spring 3.0 apporte de nombreuses nouveautés pour sa configuration et les fonctionnalités proposées :

- des possibilités enrichies de configurer le context en utilisant des annotations : annotations issues du projet Spring JavaConfig qui sont ajoutées dans Spring Core (@Configuration, @Bean, @DependsOn, @Primary, @Lazy, @Import, @ImportResource et @Value)
- Spring Expression Langage (SpEL) : un langage d'expressions utilisable pour la définition des beans dans Spring Core et pour certaines fonctionnalités dans des projets du portfolio
- le support de REST
- Object to XML Mapping (OXM) : abstraction pour utiliser des solutions de mapping objet/XML initialement proposée par le projet Spring Web services et intégrée dans Spring Core
- requiert un Java SE 5.0 ou supérieur (refactoring des API pour une utilisation des generics, des varargs, de java.util.concurrent, ...)
- une nouvelle modularisation : la distribution de Spring en jar a été revue pour que chaque module ait son propre jar. L'archive spring.jar n'est plus proposée
- le support de moteurs de bases de données embarquées (Derby, HSQL, H2)
- le support de la validation (JSR 303), du data binding et de la conversion de type
- le support JSR 330
- le scheduling par configuration, annotations (@Async, @Scheduled) ou API
- l'ajout de nouveaux namespaces (task, jdbc, mvc)
- la compatibilité forte avec Spring 2.5
- le support de l'API Porlet 2.0 par Spring MVC

Spring 3.1 :

- support des conversations
- support des caches
- ajout de la notion de profile qui permet d'avoir des configurations du context différentes pour chaque environnement
- ajout de nouvelles annotations pour définir certaines fonctionnalités de namespaces dans la configuration
- support des servlets 3.0

90.3. Spring Framework

Spring Framework contient toutes les fonctionnalités de base pour développer des applications.

Le coeur de Spring Framework 3.0 est composé d'un ensemble d'une vingtaine de modules qui sont regroupés en plusieurs parties :

- Spring Core Container : regroupe les modules de base pour mettre en oeuvre le conteneur
- AOP and Instrumentation : permet de mettre en oeuvre l'AOP
- Data Acces/Integration : regroupe les modules d'accès aux données
- Web : regroupe les modules pour le développement d'applications web
- Test : propose des fonctionnalités pour les tests automatisés avec Spring

La partie Spring Core Container contient plusieurs modules :

- Spring Core et Spring Beans : contiennent les fonctionnalités de base notamment le conteneur et des utilitaires

- Spring Context : propose un support de la définition du context Spring (sa configuration) mais aussi des fonctionnalités de base comme le mail, l'internationalisation, JNDI, ...
- Spring Expression Langage (SpEL) : propose un langage d'expressions pour interroger et manipuler les objets gérés par le conteneur

La partie AOP and Instrumentation contient plusieurs parties :

- Spring AOP : propose un support de l'AOP
- AspectJ : propose une intégration d'AspectJ
- Instrumentation : propose une instrumentation des classes et plusieurs implémentations de classloaders utilisés par certains serveurs d'applications

La partie Data Acces/Integration contient plusieurs modules

- Spring JDBC : propose une abstraction de l'utilisation de JDBC avec notamment une hiérarchie d'exceptions dédiées
- Spring ORM : propose un support pour des outils de type ORM (JPA, JDO, Hibernate, iBatis)
- Spring Transaction : propose un support déclaratif et par programmation de la gestion des transactions
- Spring OXM : propose une abstraction pour le mapping objet/XML avec un support de JAXB, Castor, XMLBeans, JiBX et XStream
- Spring JMS : propose des fonctionnalités pour faciliter la mise en oeuvre de JMS avec Spring

La partie Web contient plusieurs modules :

- Spring Web : propose des fonctionnalités de base pour les développements web (initialisation du conteneur, gestion des contextes, support multipart, extraction des paramètres d'une requête http, ...)
- Spring Web-Servlet : framework pour le développement d'applications qui met en oeuvre le motif de conception MVC. Ceci permet entre autres de choisir la technologie utilisée pour la vue (JSP, Velocity, Tiles, iText, ...)
- Spring Web-Struts : propose un support de Struts
- Spring Web-Portlet : propose un support pour les portlets

La partie Test contient un seul module :

- Spring Test : propose un support pour les tests automatisés avec un support de JUnit et TestNG

Ces modules sont utilisés comme base pour le développement d'applications.

90.4. Les projets du portfolio Spring

Spring propose aussi un ensemble très complet de modules additionnels qui ne cesse de s'enrichir pour faciliter la mise en oeuvre de certaines fonctionnalités dans les applications.

Ainsi, Spring est un portfolio de nombreux projets qui couvrent un grand nombre de besoins. Tous ces projets reposent sur le coeur de Spring : Spring Core.

Voici une liste non exhaustive de ces projets :

- Spring Framework : contient les fonctionnalités de base de Spring
- Spring Web Flow : permet de gérer l'enchaînement des pages d'une application web
- Spring BlazeDS Integration : a pour but de simplifier le développement d'applications qui utilisent Spring, Adobe BlazeDS et Adobe Flex pour la partie IHM
- Spring Web Services : permet de développer des services web de type SOAP orientés document en utilisant la manière contract first
- Spring Roo : permet le développement rapide d'applications reposant sur Spring. Roo repose sur une grande partie de configuration et fait un usage intensif de l'AOP.
- Spring Security (Acegi Security) : permet de gérer l'authentification et les habilitations d'une application web (ressources web, invocation de méthodes de services grâce à l'AOP, d'instances du modèle).
- Spring Batch : permet le développement des applications de type batch qui peuvent utiliser des transactions et gérer de gros volumes de données

- Spring Integration : a pour but de fournir une implémentation des Enterprise Integration Patterns (EIP) utilisable comme une extension du modèle de programmation Spring
- Spring AMQP : permet de faciliter l'utilisation du protocole de messaging AMQP
- Spring Gemfire : facilite l'utilisation de la solution de cache distribué Gemfire dans les applications Spring
- SpringSource dm Server (Eclipse Virgo) : est un serveur d'applications Java modulaires. Ce projet a été confié à la fondation Eclipse et possède maintenant le nom de projet Virgo.
- Spring Dynamic Modules For OSGi(tm) Service Platforms (Eclipse Gemini) :
- Spring LDAP : a pour but de simplifier l'utilisation d'annuaires de type LDAP
- Spring IDE : IDE reposant sur Eclipse et un ensemble de plugins dédiés pour faciliter le développement d'applications avec Spring
- Spring Extensions : regroupe un ensemble de sous-projets incubateurs qui concernent des extensions à Spring
- Spring Rich Client : a pour but de simplifier le développement d'applications utilisant Swing
- Spring .NET : est un portage de Spring sur la plate-forme .Net
- Spring BeanDoc :
- Spring Social : a pour but de faciliter la connexion à certaines applications sociales comme Twitter ou FaceBook
- Spring Data : a pour but de faciliter l'utilisation de solutions de type No SQL. Il est composé de plusieurs sous-projets, un pour les différentes solutions supportées
- Spring Mobile : est une extension de Spring MVC pour le développement d'applications web pour appareils mobiles
- Spring Android : a pour but de faciliter le développement de certaines fonctionnalités d'applications Android natives (REST et Auth)

Plusieurs projets ne sont plus maintenus :

- Spring JavaConfig : est intégré dans Spring Core depuis la version 3.0
- Spring Modules : est remplacé par Spring Extensions

Tous les projets de Spring sont open source et sont, pour la plupart, diffusés sous licence Apache Version 2.0.

90.5. Les avantages et les inconvénients de Spring

Spring est un framework open source majoritairement développé par SpringSource mais il n'est pas standardisé par le JCP.

Il est très largement utilisé dans le monde Java, ce qui en fait un standard de facto et constitue une certaine garantie sur la pérennité du framework.

Spring propose une très bonne intégration avec des frameworks open source (Struts, Hibernate, ...) ou des standards de Java (Servlets, JMS, JDO, ...)

Toutes les fonctionnalités de Spring peuvent s'utiliser dans un serveur Java EE et pour la plupart dans un simple conteneur web ou une application standalone.

Les fonctionnalités offertes par Spring sont très nombreuses et les sujets couverts ne cessent d'augmenter au fur et mesure des nouvelles versions et des nouveaux projets ajoutés au portfolio.

La documentation de Spring est complète et régulièrement mise à jour lors de la diffusion de chaque nouvelle version.

La mise en oeuvre de Spring n'est pas toujours aisée car il existe généralement plusieurs solutions pour mettre en oeuvre une fonctionnalité : par exemple, généralement avec Spring 3.0, une fonctionnalité est utilisable par configuration XML, par annotations ou par API. Bien sûr cela permet de choisir mais cela impose un arbitrage selon ses besoins.

Il n'est pas rare que les livrables aient une taille importante du fait des nombreuses librairies requises par Spring et ses dépendances.

90.6. Spring et Java EE

Spring est né de l'idée de fournir une solution plus simple et plus légère que celle proposée par Java 2 EE. C'est pour cette raison que Spring a été initialement désigné comme un conteneur léger (lightweight container).

L'idée principale de Spring est de proposer un framework qui utilise de simples POJO pour développer des applications plutôt que d'utiliser des EJB complexes dans un conteneur.

Spring ne respecte pas les spécifications de Java EE mais il intègre et utilise de nombreuses API de Java EE (Servlet, JMS, ...). Spring propose aussi une intégration avec certains composants de Java EE notamment les EJB.

Java EE utilise une approche convention over configuration : par exemple, les EJB sont par défaut transactionnels. Spring utilise une approche ou la configuration doit être explicite.

Spring est de plus en plus controversé notamment à cause de son empattement et de sa complexité croissante. De plus, face à la simplification engagée par Java EE à partir de sa version 5 et à l'ajout de l'injection de dépendances dans Java EE 6, le choix entre Java EE et Spring n'est plus aussi facile.

91. Spring Core

Chapitre 91

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Le coeur de Spring est composé de Spring Core : un conteneur qui implémente le motif de conception IoC (Inversion of Control). Ce conteneur prend en charge la création, la gestion du cycle de vie et les dépendances des objets qu'il gère.

La définition de ces objets est faite dans la déclaration du contexte de Spring dans un fichier de configuration XML ou partiellement réalisée en utilisant des annotations.

Pour mettre en oeuvre certaines fonctionnalités, Spring a recourt à la programmation orientée aspect (AOP : Aspect Oriented Programming).

Ce chapitre contient plusieurs sections :

- Les fondements de Spring
- Le conteneur Spring
- Le fichier de configuration
- L'injection de dépendances
- Spring Expression Langage (SpEL)
- La configuration en utilisant les annotations
- Le scheduling

91.1. Les fondements de Spring

La mise en oeuvre de Spring repose sur le motif de conception IoC et sur la programmation orientée aspects (AOP) pour développer des applications reposant sur des beans qui sont de simples POJO.

91.1.1. L'inversion de contrôle

L'IoC est un principe abstrait qui définit un motif de conception dans lequel le flux de contrôle d'un système est inversé par rapport à un développement procédural.

L'injection de dépendances est un motif de conception qui propose un mécanisme pour fournir à un composant les dépendances dont il a besoin. C'est une forme particulière d'inversion de contrôle.

L'injection de dépendances permet à une application de déléguer la gestion du cycle de vie de ses dépendances et leurs injections à une autre entité. L'application ne crée pas directement les instances des objets dont elle a besoin : les dépendances d'un objet ne sont pas gérées par l'objet lui-même mais sont gérées et injectées par une entité externe à l'objet.

Dans le cas classique, l'objet invoque le constructeur de ses dépendances pour obtenir les instances requises en utilisant l'opérateur new. Cela induit un couplage fort entre l'objet et sa dépendance. Pour réduire ce couplage, il est possible par exemple de définir une interface et d'utiliser une fabrique pour créer une instance mais cela nécessite beaucoup de code.

Avec le motif de conception IoC, la gestion des objets est confiée à un objet dédié. Celui-ci se charge de créer les instances requises et de les fournir par injection. Cette injection peut concrètement se faire de plusieurs manières :

- passer la ou les instances en paramètre du constructeur
- fournir l'instance en invoquant le setter d'une propriété
- fournir la ou les instances en paramètre de la fabrique invoquée pour créer l'instance

Les classes et les interfaces de base du conteneur Spring sont contenues dans les packages org.springframework.beans et org.springframework.context.

Une configuration permet de définir les objets qui sont gérés par le conteneur, généralement sous la forme d'un fichier XML : les informations fournies permettent au conteneur d'instancier et d'initialiser l'objet et ses dépendances.

91.1.2. La programmation orientée aspects (AOP)

L'AOP est utilisée par Spring pour fournir des fonctionnalités transverses (par exemple la gestion des transactions) ou spécifiques (par exemple l'injection de dépendances dans un bean non géré par Spring) de manières déclaratives dans la configuration XML ou grâce à des annotations.

Spring permet aussi d'utiliser l'AOP pour des besoins propres de l'application.

Historiquement, Spring propose l'utilisation de proxys avec Spring AOP : les proxys sont générés dynamiquement grâce à la bibliothèque CGLib. Ils interceptent dynamiquement les appels des méthodes et invoquent le code des greffons des aspects.

Depuis sa version 2.5, il est préférable d'utiliser AspectJ car il propose plus de fonctionnalités. Spring AOP ne peut être utilisé que sur des beans qui sont gérés par Spring.

91.1.3. Les beans Spring

Le terme bean est utilisé par Spring comme il aurait pu utiliser les termes component ou object. Le conteneur Spring est capable de gérer des JavaBeans mais aussi la plupart des classes.

Un bean doit obligatoirement être défini avec une classe dont le nom pleinement qualifié est précisé et possédant au moins un identifiant unique dans le conteneur. Les autres identifiants sont considérés comme des alias.

Chaque bean possède un nom qui sera déterminé par le conteneur si aucun n'est explicitement fourni. Le nom devrait suivre, par convention, la convention de nommage standard des instances (débuter par une minuscule puis utiliser la convention camel case).

Un bean peut avoir dans sa définition des informations relatives à sa configuration ce qui permet de préciser comment le bean sera géré dans le conteneur (singleton ou prototype, mode d'instanciation, paramètres du constructeur, valeurs des propriétés, dépendances, mode d'autowiring, méthodes à invoquer lors de l'initialisation ou de la destruction, ...)

91.2. Le conteneur Spring

Le conteneur se charge de créer les instances, de les configurer et de gérer les objets requis par l'application. Comme ces objets interagissent, généralement un objet possède des dépendances qui vont aussi être gérées par le conteneur.

Le conteneur peut donc être vu comme une fabrique évoluée qui gère le cycle de vie des objets et la gestion de leurs dépendances.

L'interface org.springframework.beans.factory.BeanFactory définit les fonctionnalités de base du conteneur.

Lors de sa création, le conteneur va vérifier la configuration qui lui est fournie pour par exemple détecter les références à des objets non définis, les dépendances circulaires, ...

Le conteneur essaie selon la configuration de créer les instances le plus tardivement possible : cependant, les singletons sont par défaut créés au lancement du conteneur.

Spring est capable de gérer n'importe quel bean. Seules quelques contraintes doivent être respectées pour permettre à Spring de réaliser l'injection de dépendances s'il y en a.

Chaque bean géré par Spring possède un ou plusieurs identifiants uniques.

La ou les instances d'un bean sont créées par Spring selon la configuration soit sous la forme d'un singleton (instance unique) ou de prototype (une instance est créée à chaque demande au conteneur)

91.2.1. L'interface BeanFactory

L'interface BeanFactory décrit les fonctionnalités de base du conteneur.

Elle contient plusieurs méthodes :

Méthode	Rôle
boolean containsBean(String)	Indiquer si le conteneur est capable de fournir une instance du bean dont le nom est fourni en paramètre
Object getBean(String)	Obtenir une instance d'un bean géré par le conteneur
T getBean(String, Class<T>) T getBean(Class<T>)	Depuis Spring 3.0, obtenir une instance d'un bean géré par le conteneur. Le type du bean est défini grâce aux generics
Class< ?> getType(String)	Obtenir le type du bean dont le nom est fourni en paramètre
boolean isSingleton(String)	Indiquer si un bean dont le nom est fourni en paramètre est un singleton (true) ou un prototype (false)
String[] getAliases(String)	Obtenir les alias du bean dont le nom est fourni en paramètre
Map<String, T> getBeansOfType(Class<T>)	Depuis Spring 3.0, retourner une collection de type map qui contient les beans gérés par le conteneur dont le type est fourni en paramètre

Spring fournit plusieurs implémentations de cette interface. Une implémentation de BeanFactory peut être vue comme une fabrique capable de gérer un ensemble de beans et leurs dépendances.

Une implémentation de BeanFactory permet de stocker des informations sur les JavaBeans qu'elle va gérer. La BeanFactory fournit une instance et gère le cycle de vie du Bean tout en stockant en interne les informations de la définition d'un bean dans une instance de type BeanDefinition.

La classe BeanDefinition encapsule toutes les informations utiles au BeanFactory pour créer une instance. Ces informations concernent la classe elle-même mais aussi ses dépendances.

91.2.2. L'interface ApplicationContext

L'interface ApplicationContext hérite des interfaces BeanFactory et MessageSource.

Elle ajoute des fonctionnalités permettant notamment l'accès aux ressources et une gestion d'événements.

La gestion d'événements est assurée grâce à la classe ApplicationEvent et à l'interface EventListener.

La mise en oeuvre des événements se fait en utilisant le motif de conception observateur. Lorsqu'un événement est émis, il est propagé à chaque bean qui implémente l'interface ApplicationListener. L'implémentation des méthodes de cette interface doit se charger des traitements à exécuter pour l'événement.

Spring propose plusieurs événements en standard :

- ContextRefreshEvent : événement informant du rafraîchissement du conteneur
- ContextStartedEvent : événement informant du démarrage du conteneur
- ContextStoppedEvent : événement informant de l'arrêt du conteneur
- ContextClosedEvent : événement informant de la fin de vie du conteneur
- RequestHandledEvent : événement informant de la fin du traitement d'une requête HTTP. Cet événement n'est utilisé que pour les applications web qui utilisent la DispatcherServlet.

Il est possible de définir ses propres événements qui doivent implémenter l'interface ApplicationEvent.

La méthode publishEvent() de l'interface ApplicationContext permet de demander l'émission synchrone d'un événement.

La méthode registerShutdownHook() permet de demander au conteneur d'être informé d'un arrêt de la JVM pour lui permettre d'exécuter correctement les traitements liés à la destruction des beans afin de gérer correctement leur cycle de vie. Ceci est particulièrement utile dans des applications non web.

91.2.3. Obtenir une instance du conteneur

Spring propose plusieurs classes qui encapsulent le conteneur selon la façon dont on souhaite le configurer et les fonctionnalités requises.

Il existe deux interfaces principales pour le conteneur selon les fonctionnalités souhaitées :

- org.springframework.beans.factory.BeanFactory : définit les fonctionnalités de base de conteneur.
- org.springframework.context.ApplicationContext : propose quelques fonctionnalités supplémentaires comme une gestion d'événements et de ressources

Pour ces deux interfaces, Spring propose plusieurs solutions pour charger son fichier de configuration.

91.2.3.1. Obtenir une instance du conteneur de type BeanFactory

L'utilisation d'une implémentation de BeanFactory est pratique pour une application exécutée dans un environnement possédant des ressources limitées.

La classe XmlBeanFactory permet d'instancier le contexte et de charger sa configuration à partir du contenu d'un fichier de configuration XML.

Exemple :

```
BeanFactory factory = new XmlBeanFactory(new FileSystemResource("c:/beans.xml"));
```

Il est aussi possible d'utiliser la classe ClassPathResource pour rechercher le fichier de configuration dans le classpath de l'application.

Exemple :

```
XMLBeanFactory factory = new XMLBeanFactory(new ClassPathResource ("appContext.xml"));
```

91.2.3.2. Obtenir une instance du conteneur de type ApplicationContext

Spring propose trois implémentations de l'interface ApplicationContext :

- ClassPathXmlApplicationContext : charge la définition du contexte à partir d'un fichier XML contenu dans le classpath.
- FileSystemXmlApplicationContext : charge la définition du contexte à partir d'un fichier XML contenu dans le système de fichiers.
- XmlWebApplicationContext : charge la définition du contexte à partir d'un fichier XML contenu dans une application web.

Exemple :

```
ApplicationContext context = new ClassPathXmlApplicationContext(new String[] {"context.xml"});
```

Exemple :

```
ApplicationContext context = new FileSystemXmlApplicationContext("appContext.xml");
```

La création d'un ApplicationContext pour une application web peut se faire de deux façons selon la version des spécifications de l'API Servlet implémentée par le conteneur.

Pour un conteneur implémentant les spécifications Servlet 2.3 et inférieure, il faut utiliser la servlet ContextLoaderServlet.

Exemple :

```
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/servicesContext.xml /WEB-INF/daoContext.xml
      /WEB-INF/applicationContext.xml</param-value>
  </context-param>

  <servlet>
    <servlet-name>context</servlet-name>
    <servlet-class>org.springframework.web.context.ContextLoaderServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
```

La servlet ContextLoaderServlet, démarrée au lancement de la webapp, va charger le fichier de configuration. Par défaut, ce fichier doit se nommer WEB-INF/applicationContext.xml

Pour un conteneur implémentant les spécifications Servlet 2.4 et supérieures : il faut utiliser un listener de type ContextLoaderListener.

Exemple :

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/daoContext.xml /WEB-INF/servicesContext.xml
  /WEB-INF/applicationContext.xml</param-value>
</context-param>

<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

La classe org.springframework.web.context.ContextLoaderListener est utilisée pour charger le ou les fichiers de configuration de l'application web.

Le paramètre contextConfigLocation permet de préciser le ou les fichiers de configuration à utiliser. Plusieurs fichiers peuvent être précisés en utilisant un espace, une virgule ou un point virgule comme séparateur. Il est aussi possible d'utiliser des motifs par exemple /WEB-INF/*Context.xml pour désigner tous les fichiers finissant par Context.xml dans le répertoire WEB-INF ou /WEB-INF/**/*Context.xml pour désigner tous les fichiers finissant par Context.xml dans le répertoire WEB-INF et tous ses sous-répertoires.

Si le paramètre contextConfigLocation n'est pas défini, le listener ou la servlet utilisent par défaut le fichier /WEB-INF/applicationContext.xml.

91.2.4. Obtenir une instance d'un bean par le conteneur

La méthode getBean() de l'interface BeanFactory qui attend en paramètre l'identifiant de l'objet permet d'obtenir une instance de cet objet selon la configuration fournie au conteneur.

Exemple :

```
package fr.jmdoudoux.dej.spring;

import org.apache.log4j.Logger;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import fr.jmdoudoux.dej.spring.service.PersonneService;

public class MonApp {

  private static Logger LOGGER = Logger.getLogger(MonApp.class);

  public static void main(final String[] args) throws Exception {

    LOGGER.info("Debut de l'application");

    ClassPathXmlApplicationContext appContext = new ClassPathXmlApplicationContext(
        new String[] { "appContext.xml" });

    PersonneService personneService = (PersonneService) appContext
        .getBean("personneService");

    // ...

    LOGGER.info("Fin de l'application");
  }
}
```

Spring 3.0 utilise les generics de Java 5. Une surcharge de la méthode getBean() possède donc la signature :

<T> T getBean(Class<T> classType) throws BeansException;

Il n'est plus nécessaire de faire un cast lors de l'invocation de la méthode getBean()

MonBean monBean = context.getBean(MonBean.class) ;

Une autre surcharge permet de préciser l'identifiant du bean et sa classe.

<T> T getBean(String name, Class<T> classType) throws BeansException;

91.3. Le fichier de configuration

Ce fichier est un document au format XML dont le tag racine est le tag <beans> et qui respecte un schéma fourni par Spring.

Chaque objet géré par le conteneur est défini dans un tag fils <bean>.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
  <bean id="..." class="...">
    <!-- configuration et description des dependances -->
  </bean>
  <bean id="...' class="...">
    <!-- configuration et description des dependances -->
  </bean>
</beans>
```

Remarque : tous les objets utilisés dans une application ne doivent pas être gérés par le conteneur. Les objets gérés sont les principaux objets de chaque couche (contrôleurs, services, DAO, ...) ainsi que des composants techniques configurables (fabriques, connections à des ressources, ...).

Le nombre d'objets gérés pouvant être important selon la taille de l'application, il est possible de repartir la configuration dans plusieurs fichiers de configuration.

Il y a deux façons de préciser tous les fichiers de configuration au conteneur

1) Pour les préciser au constructeur qui va instancier le conteneur, utiliser le tag <import> dans le fichier de configuration principal pour chaque fichier à inclure

Exemple :

```
<beans>
  <import resource="services.xml"/>
  <import resource="resources/persistance.xml"/>
  <bean id="..." class="..."/>
</beans>
```

Le chemin du fichier précisé dans l'attribut resource peut être :

- relatif au chemin du fichier de configuration
- la localisation pleinement qualifiée du fichier

Le chemin complet du fichier peut être précisé en utilisant :

- le chemin absolu dans le système de fichiers (exemple : file:C:/java/monapp/config/services.xml)
- le chemin dans un élément du classpath (exemple : classpath:/config/services.xml)

2) Il est possible d'utiliser le contenu d'une variable d'environnement de la JVM dans le chemin pour ne pas le gérer en dur : le marqueur ${nom_de_la_variable} sera remplacé par la valeur de la variable dont le nom est nom_de_la_variable.

La séparation de la configuration dans plusieurs fichiers est particulièrement utile si la taille du fichier de configuration devient importante.

Pour utiliser plusieurs fichiers de configuration, il y a plusieurs solutions :

- utiliser le constructeur de l'ApplicationContext qui accepte les noms des fichiers de configuration
- utiliser le tag <import> dans le fichier de configuration pour importer un fichier de configuration dans un autre. Chaque fichier est précisé avec l'attribut resource.

91.3.1. La définition d'un objet dans le fichier de configuration

Chaque objet géré par le conteneur est défini dans un tag fils <bean>.

La définition d'un objet doit contenir au minimum un identifiant et le nom pleinement qualifié de sa classe. Il peut aussi contenir : une portée (scope), les arguments à fournir au constructeur, les valeurs de propriétés, des méthodes d'initialisation et de destruction, des paramètres sur le mode de gestion du cycle de vie de l'objet, ...

Chaque objet géré par le conteneur doit avoir au moins un identifiant unique qui permet d'y faire référence notamment dans les dépendances.

Cet identifiant peut être fourni grâce à l'attribut id. Si aucun identifiant n'est défini pour un objet, le conteneur va lui en assigner un par défaut. Par convention, l'identifiant d'un objet commence par une minuscule.

Exemple :

```
<bean id="monBean" class="fr.jmdoudoux.dej.spring.MonBean" />
```

L'attribut name permet de fournir plusieurs identifiants pour un objet, chacun étant séparé par une virgule ou un point virgule.

L'attribut class permet de préciser le nom pleinement qualifié de la classe qui sera utilisée pour créer une nouvelle instance de l'objet. Cet attribut est obligatoire si la configuration ne précise aucun autre moyen pour obtenir une instance.

Par défaut, le conteneur invoque un constructeur pour créer une nouvelle instance : le constructeur utilisé est celui dont la signature correspond aux paramètres fournis.

Il est possible de demander l'instanciation en invoquant une méthode statique de la classe qui agit comme une fabrique. Le nom de la méthode statique est précisé avec l'attribut factory-method. Cette méthode statique doit appartenir à la classe car le conteneur va l'invoquer pour obtenir une instance. Elle peut attendre des paramètres.

Exemple :

```
<bean id="monBean" class="fr.jmdoudoux.dej.spring.MonBean" factory-method="createInstance"/>
```

Il est aussi possible de demander au conteneur d'invoquer une fabrique pour créer une nouvelle instance. Dans ce cas, l'attribut class ne doit pas être renseigné. L'attribut factory-bean doit préciser l'identifiant de l'objet géré par le conteneur qui encapsule la fabrique et l'attribut factory-method doit préciser le nom de la méthode à invoquer.

Exemple :

```
<bean id="monBeanFactory" class="fr.jmdoudoux.dej.spring.MonBeanFactory"/>

<bean id="monBean" factory-bean="monBeanFactory" factory-method="creerInstance"/>
```

Il est possible de définir un alias sur le nom du bean en utilisant le tag <alias>

Exemple :

```
<alias name="idDuBean" alias="aliasDuBean"/>
```

91.3.2. La portée des beans (scope)

Un bean géré par le conteneur possède une portée (scope).

Pour un usage général, Spring propose deux portées :

- singleton : le conteneur ne peut avoir qu'une seule instance pour un identifiant de bean. Chaque fois qu'une instance du bean sera demandée, c'est cette unique instance qui sera renvoyée par le conteneur
- prototype : chaque fois qu'une instance du bean sera demandée, le conteneur va créer une nouvelle instance

Spring propose d'autres portées notamment celles dédiées aux applications web (request, session et global-session).

La portée est définie dans le fichier de configuration en utilisant l'attribut scope du tag <bean>. La valeur fournie est celle de la portée souhaitée (singleton ou prototype).

Remarque : avant la version 2.0 de Spring, il fallait utiliser l'attribut singleton qui attend une valeur booléenne.

Par défaut, les beans ont une portée singleton : la grande majorité des beans gérés dans un conteneur Spring sont généralement des singletons.

Le conteneur ne peut pas gérer la destruction d'un bean avec une portée prototype : c'est de la responsabilité de l'application qui seule peut savoir quand l'instance n'est plus utilisée.

91.3.3. Les callbacks liés au cycle de vie des beans

Il est possible de définir des callbacks liés à l'initialisation et la destruction d'un bean qui seront invoqués par le conteneur.

Remarque : l'utilisation de ces callbacks n'est pas fortement recommandée par elle lie les beans à Spring puisque ceux-ci doivent implémenter des interfaces de Spring.

L'interface InitializingBean définit la méthode afterPropertiesSet(). Cette méthode peut contenir des traitements qui seront exécutés par le conteneur après l'initialisation d'une nouvelle instance du bean.

Remarque : il est préférable de préciser le nom d'une méthode contenant ces traitements comme valeur de l'attribut init-method du tag <bean>. Le résultat sera le même mais le bean ne sera pas lié à Spring.

L'interface DisposeableBean définit la méthode destroy(). Cette méthode peut contenir des traitements qui seront exécutés par le conteneur avant la suppression d'un bean du conteneur.

Remarque : il est préférable de préciser le nom d'une méthode contenant ces traitements comme valeur de l'attribut destroy-method du tag <bean>. Le résultat sera le même mais le bean ne sera pas lié à Spring.

Dans le fichier de configuration, il est possible de définir des méthodes d'initialisation et de destruction qui si elles sont présentes dans un bean seront automatiquement invoquées au moment opportun par le conteneur.

L'attribut default-init-method du tag <beans> permet de définir une méthode d'initialisation par défaut. Sa valeur doit contenir le nom de la méthode.

L'attribut default-destroy-method du tag <beans> permet de définir une méthode de destruction par défaut. Sa valeur doit contenir le nom de la méthode.

A partir de Spring 2.5, il est aussi possible d'utiliser les annotations @PostConstruct et @PreDestroy pour annoter respectivement les méthodes d'initialisation et de destruction.

Il est possible de combiner ces différentes solutions. Dans ce cas, le conteneur utilise un ordre précis :

- pour l'initialisation : les méthodes annotées avec @PostConstruct, la méthode afterPropertiesSet() de l'interface InitializingBean, une méthode personnalisées grâce à l'attribut init-method
- pour la destruction : les méthodes annotées avec @PreDestroy, la méthode destroy() de l'interface DisposeableBean, une méthode personnalisées grâce à l'attribut destroy-method

91.3.4. L'initialisation tardive

Le comportement par défaut du conteneur est d'instancier les singletons lors de son initialisation. Il est possible de demander l'instanciation lors de la première utilisation en utilisant l'attribut lazy-init du tag <bean> avec la valeur true. Ceci n'empêchera pas le conteneur d'instancier le singleton si celui-ci est une dépendance d'un autre singleton qui est instancié au démarrage du conteneur.

Il est possible de préciser l'attribut lazy-init de chaque bean en utilisant l'attribut default-lazy-init du tag <beans>. Cet attribut est configuré au niveau du contexte et s'applique donc sur tous les beans gérés.

Exemple :

```
<beans default-lazy-init="true">
  <!-- ... -->
</beans>
```

91.3.5. L'utilisation de valeurs issues d'un fichier de propriétés

Certaines valeurs de propriétés dans le fichier de configuration peuvent être extraites d'un fichier de properties. La résolution se fait en utilisant un placeholder ayant la forme ${xxx} où xxx est la clé dans le fichier de properties.

Exemple :

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource"
p:driverClassName="${jdbc.driverClassName}"
p:url="${jdbc.url}"
p:username="${jdbc.username}"
p:password="${jdbc.password}"/>
```

La résolution est effectuée par une instance particulière de BeanFactoryPostProcessor nommée PropertyPlaceholderConfigurer.

Exemple :

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="location" value="classpath:jdbc.properties"/>
</bean>
```

Spring 2.5 propose l'espace de nommage context qui simplifie cette déclaration en utilisant le tag <property-placeholder>. Sa propriété location permet de préciser le chemin du fichier de properties.

Exemple :

```
<context:property-placeholder location="classpath:jdbc.properties"/>
```

L'externalisation de certaines valeurs est très pratique notamment pour celles qui sont différentes selon l'environnement d'exécution.

91.3.6. Les espaces de nommage

Le but de ces espaces de nommage est de simplifier la définition du contexte.

Spring 2.0 propose plusieurs espaces de nommage (namespaces) : aop, jee, lang, tx et util.

Spring 2.5 ajoute les espaces de nommage context et jms.

Spring définit aussi plusieurs autres espaces de nommage pour des usages plus particuliers : oxm, sws, ...

Il est aussi possible de définir ses propres espaces de nommage.

91.3.6.1. L'espace de nommage beans

L'espace de nommage beans est obligatoire et est utilisé comme espace de nommage par défaut dans le fichier de configuration.

L'uri du schéma correspondant est www.springframework.org/schema/beans

Le schéma est défini dans le fichier xsd www.springframework.org/schema/beans/spring-beans-x.x.xsd

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

<!-- ... -->

</beans>
```

Le tag <beans> est l'élément racine du fichier de configuration du context Spring.

Le tag <beans> peut avoir plusieurs tags fils : <alias>, <bean>, <description> et <import>.

Le tag <bean> permet de configurer un bean. Il possède plusieurs attributs :

Attribut	Rôle
abstract	Booléen qui précise si le bean est abstrait : la valeur true indique au conteneur de ne pas créer d'instance
autowire	Permet de préciser comment le bean sera injecté : byType, byName, constructor, autodetect, none (pas d'autowiring)
autowire-candidate	Booléen qui précise si l'instance du bean peut être utilisée lors de l'injection de dépendances
class	Le nom pleinement qualifié de la classe du bean
dependency-check	Préciser les dépendances qui seront valorisées par le conteneur : simple (pour les primitives), object (pour les objets), default, none, all
depends-on	Préciser un bean qui devra être initialisé avant que le bean soit instancié
destroy-method	Préciser une méthode qui sera invoquée lorsque le bean est déchargé du conteneur
factory-bean	Préciser un bean dont la méthode indiquée par l'attribut factory-method sera utilisée comme fabrique
factory-method	Préciser une méthode statique du bean indiqué par l'attribut factory-bean qui sera utilisée par le conteneur comme une fabrique
id	Identifiant du bean
init-method	Nom de la méthode d'initialisation qui sera invoquée une fois l'instance créée et les dépendances injectées
lazy-init	Booléen qui indique si l'instance sera initialisée tardivement
name	Nom du bean
parent	Préciser un bean dont la configuration sera héritée par le bean
scope	Permet de préciser la portée du bean : singleton par défaut, prototype, request, session

Le tag <bean> peut avoir plusieurs tags fils : <constructor-arg>, <description>,<lookup-method>,<meta>,<property> et <replaced-method>.

Le tag <constructor-arg> permet d'utiliser l'injection par constructeur : il permet de fournir une valeur ou une référence sur un bean géré par le conteneur

Le tag <lookup-method> permet d'utiliser l'injection par getter : le getter est remplacé par une autre implémentation qui retourne une instance particulière.

Le tag <property> permet d'utiliser l'injection par setter pour fournir une valeur à une propriété. Cette valeur peut être une référence sur un autre bean géré par le conteneur.

Le tag <replaced-method> permet de remplacer les traitements d'une méthode du bean par une autre implémentation.

Le tag <alias> permet de définir un alias pour un bean.

Le tag <import> permet d'importer une autre partie de la définition du contexte de Spring. Ce tag est particulièrement utile pour définir le contexte dans plusieurs fichiers, chacun contenant des définitions de beans par thème fonctionnel ou technique (services, transactions, accès aux données, ...)

Le tag <description> permet de fournir une description de la définition du contexte ou d'un bean.

91.3.6.2. L'espace de nommage P

Spring 2.0 a introduit un espace de nommage particulier nommé p dont le but est de réduire la quantité de code à produire dans le fichier de configuration XML du contexte Spring.

L'uri du schéma correspondant est www.springframework.org/schema/p

L'espace de nommage p de Spring est une alternative pour définir les propriétés des beans dans le fichier de configuration : au lieu d'utiliser un tag fils <property>, il est possible d'utiliser l'espace de nommage p pour définir les propriétés sous la forme d'attributs du tag <bean>.

Exemple :

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

    <bean name="monBean" class="fr.jmdoudoux.dej.spring.MonBean">
        <property name="maPropriete" value="maValeur"/>
    </bean>
</beans>
```

Exemple :

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

    <bean name="monBean"
        class="fr.jmdoudoux.dej.spring.MonBean"
        p:maPropriete="maValeur"/>
</beans>
```

L'espace de nommage p permet donc de réduire la quantité de code XML.

L'espace de nommage ne possède pas de définition dans un schéma dédié, ce qui permet d'utiliser n'importe quel nom de propriété comme attribut.

Exemple :

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource"
p:driverClassName="oracle.jdbc.driver.OracleDriver"
p:url="jdbc:oracle:thin:@monServeur:1521:maBase"
p:username="root"
```

```
p:password="mdp"/>
```

Il est aussi possible de fournir une référence sur un autre bean comme valeur en utilisant le suffixe -ref

Exemple :

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

    <bean name="monAutreBean"
        class="fr.jmdoudoux.dej.spring.MonAutreBean"/>

    <bean name="monBean"
        class="fr.jmdoudoux.dej.spring.MonBean"
        p:maPropriete="maValeur"
        p:maDependance-ref="monAutreBean"/>
</beans>
```

91.3.6.3. L'espace de nommage jee

Spring 2.0 a introduit un espace de nommage particulier nommé jee qui permet de faciliter la configuration en utilisant un environnement Java EE par exemple en obtenant un objet d'un annuaire JNDI ou une référence à un EJB.

L'uri du schéma correspondant est www.springframework.org/schema/jee

Le schéma est défini dans le fichier xsd www.springframework.org/schema/beans/spring-jee-x.x.xsd

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:jee="http://www.springframework.org/schema/jee"

xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/jee
http://www.springframework.org/schema/jee/spring-jee-2.0.xsd">
<!-- ... -->
</beans>
```

Le tag <jee:jndi-environment> définit des propriétés pour permettre un accès à l'annuaire JNDI.

Le tag <jee:jndi-lookup> permet d'obtenir une référence sur un objet configuré dans un annuaire en utilisant JNDI.

Exemple :

```
<jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/MaDataSource"/>
```

Ce tag facilite l'utilisation d'un objet de type JndiObjectFactoryBean.

Le tag <jee:local-slsb> permet d'obtenir une référence sur un EJB Session Stateless local et de créer un proxy pour accéder à cet EJB.

Exemple :

```
  <jee:local-slsb id="monEJB" jndi-name="monEJB"
    business-interface="fr.jmdoudoux.dej.ejb.MonEJB" />
```

Ce tag utilise un objet de type LocalStatelessSessionProxyFactoryBean.

Le tag <jee:remote-slsb> permet d'obtenir une référence sur un EJB Session Stateless distant et de créer un proxy pour accéder à cet EJB.

Ce tag utilise un objet de type RemoteStatelessSessionProxyFactoryBean.

91.3.6.4. L'espace de nommage lang

Spring 2.0 a introduit l'espace de nommage lang qui permet de configurer dans le contexte des objets définis dans un langage de scripting comme par exemple Groovy.

L'uri du schéma correspondant est www.springframework.org/schema/lang

Le schéma est défini dans le fichier xsd www.springframework.org/schema/beans/spring-lang-x.x.xsd

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:lang="http://www.springframework.org/schema/lang"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/lang
       http://www.springframework.org/schema/lang/spring-lang-3.0.xsd">

<!-- ... -->

</beans>
```

Plusieurs langages dynamiques sont supportés : Groovy, JRuby et BeanShell.

Exemple :

```
<lang:groovy id="monBean" script-source="classpath:MonBean.groovy">
  <lang:property name="message" value="Bienvenue"/>
</lang.groovy>
```

Le tag <defaults> permet de configurer certaines propriétés applicables pour les beans définis avec un langage dynamique.

Le tag <groovy> permet de configurer dans le contexte un bean défini en Groovy.

La propriété script-source permet de préciser le script qui contient la définition du bean.

La propriété refresh-check-delay permet de préciser une durée en millisecondes entre chaque vérification de changement dans le script avec son rechargement le cas échéant. Si cet attribut n'est pas valorisé alors aucune vérification n'est faite. Ce rafraîchissement permet de tenir compte d'une modification du script sans avoir à recharger la classe : c'est un des intérêts des langages dynamiques.

Exemple :

```
  <lang:groovy id="monAutreBean" script-source="classpath:MonAutreBean.groovy"
    refresh-check-delay="60000" />
  <bean id="monBean" class="fr.jmdoudoux.dej.spring.MonBean">
    <property name="dependance" ref="monAutreBean" />
  </bean>
```

Le tag fille <inline-script> permet de fournir un script Groovy dans son corps.

Le tag fille <property> permet de configurer une propriété du script Groovy

Le tag <jruby> permet de configurer dans le contexte un bean défini en JRuby. Son mode d'utilisation est similaire à celui du tag <groovy>.

L'attribut obligatoire <script-interfaces> doit contenir la ou les interfaces qui seront utilisées par Spring pour créer un proxy qui se chargera d'invoquer la classe JRuby.

Le tag <bsh> permet de configurer dans le contexte un bean défini en BeanShell.

91.3.6.5. L'espace de nommage context

Spring 2.5 a introduit l'espace de nommage context qui permet de faciliter la configuration des beans de Spring définis dans le contexte.

L'uri du schéma correspondant est www.springframework.org/schema/context

Le schéma est défini dans le fichier xsd www.springframework.org/schema/beans/spring-context-x.x.xsd

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/lang"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<!-- ... -->

</beans>
```

Le tag <property-placeholder> permet de demander l'activation du remplacement des marqueurs ${...} par leur valeur dans les fichiers de propriétés correspondantes.

Exemple :

```
<context:property-placeholder location="file:////etc/monApp.properties"/>
```

Ce tag permet de définir un objet de type PropertyPlaceHolderConfigurer. Pour une configuration précise de cet objet, il faut définir sa propre instance explicitement.

L'utilisation de ce tag permet facilement d'externaliser certaines valeurs de configuration.

Exemple :

```
<context:property-placeholder location="classpath:monApp.properties"/>
<bean id="monBean" class="fr.jmdoudoux.dej.spring.MonBean">
  <property name="maValeur" ref="${monApp.maValeur}"/>
</bean>
```

Spring 2.5 propose le tag <annotation-config> qui va activer la recherche de plusieurs annotations pour configurer les beans dans le contexte.

Le tag <annotation-config> permet de définir simplement des beans de type CommonAnnotationBeanPostProcessor, AutowiredAnnotationBeanPostProcessor, RequiredAnnotationBeanPostProcessor et PersistenceAnnotationBeanPostProcessor. Ceux-ci se chargent de rechercher et d'utiliser des annotations pour configurer les beans dans le contexte.

Exemple :

```
<context:annotation-config/>
```

Un bean de type CommonAnnotationBeanPostProcessor permet d'utiliser les annotations de la JSR 250 @Resource, @PostConstruct, @PreDestroy, @WebServiceRef et @EJB.

Un bean de type AutowiredAnnotationBeanPostProcessor permet d'utiliser l'annotation @Autowired

Un bean de type RequiredAnnotationBeanPostProcessor permet d'utiliser l'annotation @Required

Un bean de type PersistenceAnnotationBeanPostProcessor permet d'utiliser les annotations @PersistenceContext et @PersistenceUnit de JPA.

Le tag <component-scan> permet de préciser les packages qui devront être scannés à la recherche de classes annotées dans le but de configurer le contexte et d'activer cette recherche.

Exemple :

```
<context:component-scan base-package="fr.jmdoudoux.dej.spring.services"/>
```

Dans l'exemple ci-dessus, les classes du packages fr.jmdoudoux.dej.sprint.services seront scannées à la recherche de beans annotés avec @Component, @Service, @Repository ou @Controller pour permettre leur définition dans le contexte.

Il est possible d'utiliser des tags fils <include-filter> et/ou <exclude-filter> pour filtrer les classes à scanner dans le ou les packages proposés.

Le tag <load-time-weaver> permet d'activer le tissage au runtime. L'utilisation de ce tag permet aussi d'activer l'injection des dépendances dans les instances de classes non gérées par Spring et annotées avec @Configurable. Dans ce cas, la classe AnnotationBeanConfigurerAspect incluse dans la bibliothèque spring-aspects.jar doit être dans le classpath.

Exemple :

```
<context:load-time-weaver/>
```

Il est possible de préciser une implémentation particulière du type LoadTimeWeaver qui sera utilisée en la précisant comme valeur de l'attribut weaver-class. Si cet attribut n'est pas précisé l'instance de LoadTimeWeaver utilisée sera déterminée automatiquement.

L'attribut aspectj-weaving permet d'activer ou non le tissage au runtime en utilisant AspectJ : les valeurs possibles sont on (LTW activé), off (LTW désactivé), autodetect (valeur par défaut qui active le LTW si un fichier META-INF/aop.xml peut être trouvé par Spring).

Le tag <spring-configured> permet d'activer l'injection de dépendances dans des instances qui ne sont pas gérées par Spring : ceci concerne les classes qui sont annotées avec @Configurable.

Exemple :

```
<context:spring-configured/>
```

Ce tag permet de définir un objet de type AnnotationBeanConfigurerAspect.

Le tag <mbean-export> permet d'activer l'exposition des MBeans définis dans le context avec l'annotation @ManagedResource en utilisant une instance de type MBeanExporter.

Ce tag remplace la définition d'un objet de type AnnotationMBeanExporter.

Exemple :

```
<context:mbean-export/>
```

L'attribut server permet de préciser le nom du serveur de MBeans à utiliser.

L'attribut default-domain permet de préciser le domain par défaut qui sera utilisé.

Le tag <mbean-server> permet d'activer le serveur de MBeans de la plate-forme. Celui-ci est automatiquement détecté pour les JVM 1.5 et ultérieures, Weblogic à partir de la version 9 et Websphere à partir de la version 6.1.

Exemple :

```
<context:mbean-server/>
```

L'attribut id permet de préciser le nom du serveur de MBeans qui par défaut est "mbeanServer".

Le tag <property-override> permet d'activer le remplacement des valeurs des propriétés de beans.

Ce tag remplace la définition d'un bean de type PropertyOverrideConfigurer.

Exemple :

```
<context:property-override location="classpath:maconfig.properties"/>
<bean id="monBean" class="fr.jmdoudoux.dej.spring.MonBean">
  <property name="maValeur" ref="0"/>
</bean>
```

La clé du fichier de properties est l'id du bean suivi d'un point suivi du nom de la propriété.

Exemple :

```
monBean.maValeur=1234
```

Les valeurs contenues dans le fichier de properties sont toujours littérales : il n'est pas possible de faire référence à un autre bean.

Dans la définition du contexte, il n'est pas possible de voir que la valeur d'une propriété sera remplacée.

91.3.6.6. L'espace de nommage util

Spring 2.0 a introduit un espace de nommage particulier nommé util qui facilite certaines fonctionnalités de configuration comme la déclaration d'un bean dont la valeur est une constante ou une collection (list, map ou set).

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
       http://www.springframework.org/schema/util
       http://www.springframework.org/schema/util/spring-util-2.0.xsd">

<!-- ... -->

</beans>
```

Le tag <util:constant> permet de définir un bean à partir d'une constante.

Exemple :

```
<util:constant id="integerMaxValue" static-field="java.lang.Integer.MAX_VALUE"/>
```

L'attribut static-field permet de préciser la constante.

La propriété id permet de définir le nom du bean.

Exemple :

```
<bean class="fr.jmdoudoux.dej.spring.MonBean">
  <property name="maValeur" ref="integerMaxValue"/>
</bean>
```

Il est aussi possible d'imbriquer le tag <util:constant> dans la déclaration d'un bean

Exemple :

```
<bean class="fr.jmdoudoux.dej.spring.MonBean">
  <property name="maValeur">
    <util:constant static-field="java.lang.Integer.MAX_VALUE"/>
  </property>
</bean>
```

Le tag <util:property-path> permet de définir un bean à partir d'une propriété d'un autre bean

Exemple :

```
<util:property-path id="maValeur" path="monBean.maValeur"/>
```

L'utilisation de ce tag peut remplacer la définition d'un bean en utilisant la classe org.springframework.beans.factory.config.PropertyPathFactoryBean.

Le tag <util:properties> permet de définir un bean de type Properties à partir du contenu d'un fichier .properties.

Exemple :

```
<util:properties id="mesProperties" location="classpath:config.properties"/>
```

L'utilisation de ce tag peut remplacer la définition d'un bean en utilisant la classe org.springframework.beans.factory.config.PropertiesFactoryBean.

Exemple :

```
<bean id="mesProperties"
      class="org.springframework.beans.factory.config.PropertiesFactoryBean">
  <property name="location" value=" classpath:config.properties"/>
</bean>
```

Le tag <util:list> permet de définir un bean qui est une collection de type List.

Exemple :

```
<util:list id="prenoms">
  <value>Beatrice</value>
  <value>Michel</value>
  <value>Thomas</value>
  <value>Valerie</value>
</util:list>
```

L'attribut id permet de préciser le nom du bean.

L'attribut list-class permet de préciser le type de la collection de type List

Exemple :

```
<util:list id="prenoms" list-class="java.util.LinkedList">
  <value>Beatrice</value>
  <value>Michel</value>
  <value>Thomas</value>
  <value>Valerie</value>
</util:list>
```

Chaque élément de la collection est précisé avec un tag fils value dont le corps contient la valeur.

L'utilisation de ce tag peut remplacer la création d'un bean en utilisant la classe org.springframework.beans.factory.config.ListFactoryBean.

Exemple :

```
<bean id="prenoms" class="org.springframework.beans.factory.config.ListFactoryBean">
  <property name="sourceList">
    <list>
      <value>Beatrice</value>
      <value>Michel</value>
      <value>Thomas</value>
      <value>Valerie</value>
    </list>
  </property>
</bean>
```

Le tag <util:map> permet de définir un bean qui est une collection de type Map.

Exemple :

```
<util:map id="nombres">
  <entry key="1" value="Un"/>
  <entry key="2" value="Deux"/>
  <entry key="3" value="Trois"/>
  <entry key="4" value="Quatre"/>
</util:map>
```

L'attribut id permet de préciser le nom du bean.

L'attribut map-class permet de préciser le type de la collection de type Map

Exemple :

```
<util:map id="nombres" map-class="java.util.TreeMap">
  <entry key="1" value="Un"/>
  <entry key="2" value="Deux"/>
  <entry key="3" value="Trois"/>
  <entry key="4" value="Quatre"/>
</util:map>
```

L'utilisation de ce tag peut remplacer la création d'un bean en utilisant la classe org.springframework.beans.factory.config.MapFactoryBean.

Exemple :

```
<bean id="nombres" class="org.springframework.beans.factory.config.MapFactoryBean">
  <property name="sourceMap">
    <map>
      <entry key="1" value="Un"/>
      <entry key="2" value="Deux"/>
      <entry key="3" value="Trois"/>
      <entry key="4" value="Quatre"/>
    </map>
  </property>
</bean>
```

Le tag <util:set> permet de définir un bean qui est une collection de type Set.

Exemple :

```
<util:set id="prenoms">
  <value>Beatrice</value>
  <value>Michel</value>
  <value>Thomas</value>
  <value>Valerie</value>
</util:set>
```

L'attribut id permet de préciser le nom du bean.

L'attribut set-class permet de préciser le type de la collection de type Set.

Exemple :

```
<util:set id="prenoms" set-class="java.util.TreeSet">
  <value>Beatrice</value>
  <value>Michel</value>
  <value>Thomas</value>
  <value>Valerie</value>
</util:set>
```

L'utilisation de ce tag peut remplacer la création d'un bean en utilisant la classe org.springframework.beans.factory.config.SetFactoryBean.

91.4. L'injection de dépendances

L'injection de dépendances est une mise en oeuvre de l'IoC : les instances des dépendances vont être injectées automatiquement par le conteneur selon la configuration lors de l'instanciation d'un objet.

L'implémentation de l'injection de dépendances proposée par Spring peut se faire de deux façons :

- injection par le constructeur : les instances des dépendances sont fournies par le conteneur lorsque celui-ci invoque le constructeur de la classe pour créer une nouvelle instance
- injection par un setter : le conteneur invoque le constructeur puis invoque les setters de l'instance pour fournir chaque instance des dépendances

Il est aussi possible de fournir les instances des dépendances en paramètre de l'invocation d'une fabrique mais au final celle-ci va utiliser l'injection par le constructeur ou par les setters.

91.4.1. L'injection de dépendances par le constructeur

La classe à instancier doit proposer un constructeur qui attend en paramètre la ou les instances des dépendances requises.

Exemple :

```
public class PersonneService {

  private PersonneDao personneDao;

  public PersonneService(PersonneDao personneDao) {
    this.personneDao = personneDao;
  }

  // méthodes de la classe qui peuvent utiliser la dépendance
}
```

Dans le fichier de configuration, le tag <constructor-arg> permet de fournir un paramètre au constructeur.

L'attribut value permet de fournir une valeur au paramètre.

Exemple :

```
<bean id="monBean" class="fr.jmdoudoux.dej.spring.MonBean">
  <constructor-arg value="123"/>
  <constructor-arg value="456"/>
</bean>
```

Le constructeur invoqué par le conteneur est celui dont la signature comprend les types des arguments configurés. Si les types ne sont pas identifiables ou ne suffisent pas pour déterminer de façon certaine le constructeur à utiliser, le conteneur utilise l'ordre des paramètres définit dans la configuration.

L'attribut type permet d'indiquer au conteneur le type de la valeur et ainsi facilite le travail du conteneur pour déterminer le constructeur à utiliser.

Exemple :

```
<bean id="monBean" class="fr.jmdoudoux.dej.spring.MonBean">
  <constructor-arg type="int" value="123"/>
  <constructor-arg type="java.lang.String" value="456"/>
</bean>
```

Pour faciliter encore plus son travail, l'attribut index permet de préciser l'index du paramètre définit. Le premier paramètre possède l'index 0.

Exemple :

```
<bean id="monBean" class="fr.jmdoudoux.dej.spring.MonBean">
  <constructor-arg index="0" value="123"/>
  <constructor-arg index="1" value="456"/>
</bean>
```

91.4.2. L'injection de dépendances par un setter

Dans ce cas, la classe doit proposer une propriété avec un setter respectant les conventions JavaBean pour chacune de ses dépendances.

Le conteneur va créer une instance en invoquant le constructeur par défaut puis va invoquer le setter de chaque dépendance en lui passant en paramètre celle définie dans la configuration.

Exemple :

```
public class PersonneService {

  private PersonneDao personneDao;

  public void setPersonneDao(PersonneDao personneDao) {
    this.personneDao = personneDao;
  }

  // méthodes de la classe qui peuvent utiliser la dépendance
}
```

Dans le fichier de configuration, le tag <property> permet de fournir un paramètre au setter d'une propriété.

Exemple :

```
<bean id="monBean" class="fr.jmdoudoux.dej.spring.MonBean">
  <property name="prop1" value="123"/>
  <property name="prop2" value="456"/>
</bean>
```

L'attribut obligatoire name permet de préciser le nom de la propriété concernée.

L'attribut value permet de préciser une valeur statique.

L'attribut ref permet de préciser un bean géré par le conteneur comme valeur du paramètre : sa valeur doit contenir l'identifiant du bean concerné dans le contexte Spring.

91.4.3. Le passage des valeurs pour les paramètres

Les tags <constructor-arg> et <property> possèdent plusieurs attributs pour assigner une valeur. Ils possèdent aussi plusieurs tags fils qui permettent de réaliser cette opération.

La valeur peut aussi être fournie en utilisant un tag fils <value> ou <bean>.

Le tag fils <value> permet de fournir une valeur dans sa représentation sous la forme d'une chaîne de caractères.

Le tag <bean> peut aussi être utilisé pour laisser le constructeur créer une instance : dans ce cas, il n'est pas nécessaire de fournir un identifiant.

Les tags fils <idref> et <ref> permettent de préciser un bean géré par le conteneur comme valeur du paramètre : son attribut bean doit contenir l'identifiant du bean concerné.

Les tags fils <list>, <maps>, <props> et <set> permettent de fournir des collections comme valeur du paramètre.

Le tag <null> permet de préciser la valeur null au paramètre.

Le tag <value> vide permet de préciser une chaîne de caractères vide.

91.4.4. Le choix entre injection par constructeur ou par setter

Il n'y a pas de règle immuable dans la mesure où chacune des deux méthodes possède des avantages et des inconvénients.

L'avantage d'utiliser l'injection par constructeur est que l'instance obtenue est complètement initialisée suite à son instanciation.

Si le nombre de dépendances est important ou si certaines sont optionnelles, le choix d'utiliser l'injection par constructeur n'est peut être pas judicieux.

L'injection par constructeur peut aussi induire une dépendance circulaire : par exemple une classe A a une dépendance avec une instance de la classe B et la classe B a une dépendance avec une instance de la classe A. Dans ce cas, le conteneur lève une exception.

L'injection par setter peut permettre de modifier l'instance de la dépendance : cela n'est pas toujours souhaitable mais peut être utile dans certaines circonstances.

Le conteneur Spring permet aussi de mixer ces deux modes d'injection : une partie par constructeur et une autre par setter.

Le choix doit donc tenir compte du contexte d'utilisation.

91.4.5. L'autowiring

L'autowiring laisse le conteneur déterminer automatiquement l'objet géré par le conteneur qui sera injecté comme dépendance d'un autre objet.

L'utilisation de l'autowiring permet de simplifier grandement le fichier de configuration car il devient superflu de définir les valeurs des paramètres fournis au setter ou au constructeur pour injecter les dépendances.

L'attribut autowire du tag bean permet de demander l'activation de l'autowiring pour le bean.

L'autowiring peut fonctionner selon plusieurs stratégies :

- no : pas d'autowiring. Les dépendances doivent être explicitement décrites dans la configuration
- byName : permet d'injecter automatiquement une propriété selon son nom : le conteneur recherche dans les objets gérés un objet unique dont l'identifiant correspond au nom de la propriété. S'il est trouvé, le conteneur l'injecte
- byType : permet d'injecter automatiquement une propriété selon son type : le conteneur recherche dans les objets gérés un objet unique qui soit du type de la propriété. Si celui-ci est trouvé, il est injecté. Si plusieurs objets sont trouvés, alors une exception est levée car le conteneur ne peut pas déterminer quel objet utiliser. Si aucun objet n'est trouvé, alors la propriété reste null par défaut. L'utilisation de la valeur objects dans l'attribut dependency-check va lever une exception dans ce cas.
- constructor : similaire au mode byType mais en utilisant les paramètres du constructeur
- autodetect : utilise le mode constructor ou byType après analyse de la classe du bean

L'attribut default-autowire du tag <beans> permet de préciser le mode de fonctionnement par défaut de l'autowiring.

La définition explicite prend toujours le pas sur l'autowiring.

L'autowiring ne peut être utilisé que pour des objets : il n'est pas possible d'utiliser l'autowiring sur des propriétés de types primitif ou String ou des tableaux.

L'autowiring permet de simplifier la configuration des beans mais aussi de maintenir cette configuration à jour lors de l'ajout d'une nouvelle dépendance dans un bean qui utilise l'autowiring.

Avec l'autowiring, comme les dépendances ne sont plus explicitement définies dans la configuration, certains outils ne pourront plus les déterminer.

Pour que l'autowiring fonctionne correctement, le conteneur doit être en mesure de déterminer de façon non ambigüe l'instance qui sera injectée.

Il est possible de marquer des objets comme ne devant pas être pris en compte comme candidats à l'autowiring. Il suffit pour cela de donner la valeur false à l'attribut autowire-candidate.

Remarque : la mise en oeuvre de l'autowiring peut aussi se faire avec des annotations dédiées.

91.5. Spring Expression Langage (SpEL)

Spring 3.0 a introduit un langage d'expressions nommé Spring Expression Language (en abrégé Spring EL ou SpEL).

Spring EL est un langage d'expressions pour permettre l'interrogation et la manipulation d'objets au runtime : il permet de faciliter la configuration en utilisant un langage d'expressions.

Il existe déjà plusieurs langages d'expressions comme Unified EL ou JBoss EL mais Spring a décidé de développer son propre langage d'expressions notamment pour lui permettre d'être indépendant dans ses évolutions.

SpEL est un langage d'expressions, similaire à Unified EL utilisé dans les JSP, qui permet principalement d'accéder à des données.

Avec SpEL, la configuration de Spring n'est plus limitée à des valeurs fixes ou des références vers d'autres objets : il est possible de déterminer des valeurs dynamiquement.

SpEL fait partie de Spring Core : il est ainsi potentiellement utilisable dans tous les projets du portfolio Spring. SpEL est déjà utilisé par plusieurs projets du portfolio notamment Spring Security et Spring Batch.

La syntaxe utilisée par SpEL pour définir une expression est de la forme #{<expression>}.

SpEL supporte de nombreuses fonctionnalités : expressions littérales, assignations, opérateurs, expressions régulières, manipulations de collections, ...

SpEL propose aussi des fonctionnalités relatives aux classes : accès aux propriétés, invocation de méthodes, invocation de constructeurs, ...

SpEL peut être utilisé dans la configuration du contexte (dans le fichier de configuration XML ou avec certaines annotations) ou être évalué dynamiquement dans le code de l'application en utilisant l'API dédiée.

SpEL est contenu dans le package org.springframework.expression

Une exception de type SpelEvaluationException est levée si l'évaluation de l'expression échoue.

SpEL est utilisable :

- dans les fichiers de configuration Spring
- dans l'annotation @Value
- dans des JSP avec le tag <spring:eval>
- dans une application en utilisant une API dédiée

91.5.1. La mise en oeuvre de SpEL dans la définition du contexte

SpEL peut être utilisé dans la déclaration du contexte.

L'exemple ci-dessous utilise SpEL dans le fichier de configuration XML pour mettre en oeuvre différentes fonctionnalités.

Exemple :

```
  <bean id="monBean" class="fr.jmdoudoux.dej.spring.MonBean">
    <property name="nombreAleatoire" value="#{ T(java.lang.Math).random () * 100.0 }" />
    <property name="defaultLocale" value="#{systemProperties['user.region'] }" />
    <property name="monAutrePropriete" value="#{monAutreBean.propriete}" />
  </bean>

  <bean id="monAutreBean" class="fr.jmdoudoux.dej.spring.MonBean">
    <property name="propriete" value="AutreValeur" />
  </bean>
```

SpEL peut aussi être utilisé dans l'annotation @Value

L'exemple ci-dessous utilise SpEL pour obtenir une valeur d'une propriété système

Exemple :

```
  @Value("#{ systemProperties['user.name'] }")
  public void setUserName(final String userName) {
    this.userName = userName;
  }
```

Dans l'exemple ci-dessous, un fichier de propriétés est déclaré dans le contexte.

Exemple :

```
<util:properties id="appConfig" location="classpath:conf.properties"/>
```

Il est possible d'utiliser SpEL pour obtenir des valeurs du fichier de properties selon leurs clés.

Exemple :

```
@Component
public class MonBean {

  @Value("#{appConfig['db.Host']}")
  private String dbHost;

  private String dbUser;

  @Value("#{appConfig['db.User']}")
  public void setDbUser(final String dbUser) {
    this.dbUser = dbUser;
  }
}
```

SpEL peut aussi être utilisé pour créer une nouvelle instance.

Exemple :

```
public class MonBean {
  private Date dateDebut;

  @Value("#{new java.util.Date()}")
  public void setDateDebut(final Date dateDebut) {
    this.dateDebut = dateDebut;
  }
}
```

91.5.2. L'utilisation de l'API

Pour utiliser l'API, il faut créer une instance de type ExpressionParser qui est le parseur d'expressions.

Spring propose une implémentation sous la forme de la classe SpelExpressionParser qui est un parseur d'expressions SpEL.

La méthode parseExpression() renvoie une instance de type Expression qui encapsule l'expression fournie en paramètre.

Exemple :

```
package fr.jmdoudoux.dej.spring;

import org.springframework.expression.Expression;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;

public class ExpressionTest {

  public static void main(final String[] args) {

    ExpressionParser parser = new SpelExpressionParser();
    Expression expression = parser.parseExpression("'Hello world'");
    String message = (String) expression.getValue();
    System.out.println("Message = " + message);
  }
}
```

La chaîne de caractères fournie en paramètre de la méthode parseExpression() doit être une expression valide de SpEL.

La méthode getValue() de la classe Expression permet d'obtenir le résultat de l'évaluation de l'expression.

Exemple :

```
    ExpressionParser parser = new SpelExpressionParser();
    Expression exp = parser.parseExpression("'Hello world'.substring(0, 5)");
    String resultat = exp.getValue(String.class);
    System.out.println("Resultat = " + resultat);
```

La méthode getValue() possède une version surchargée qui prend en paramètre le type de la valeur de retour sous la forme d'une instance de Class<T>.

Exemple :

```
    ExpressionParser parser = new SpelExpressionParser();
    Expression expression = parser.parseExpression("' Hello'.charAt(5)");
    Character character = expression.getValue(Character.class);
    System.out.println(character);
```

Le type de la valeur de retour peut utiliser les generics.

Exemple :

```
    ExpressionParser parser = new SpelExpressionParser();
    Expression expression = parser.parseExpression("new java.util.ArrayList()");
    List liste = expression.getValue(List.class);
    System.out.println(liste);
```

Le parser SpEL est threadsafe : il est donc possible de partager son instance.

Exemple :

```
package fr.jmdoudoux.dej.spring;

import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
```

```
public class TestParser {

  public final static ExpressionParser EXPRESSION_PARSER = new SpelExpressionParser();
}
```

Le parser est utilisé pour évaluer des expressions au runtime. L'expression peut être évaluée dans un contexte particulier encapsulé dans un objet de type ParserContext.

Le contexte d'évaluation peut contenir un état ou un comportement qui sera pris en compte lors de l'évaluation de l'expression.

Il est par exemple possible d'enregistrer une variable dans le contexte pour que celle-ci puisse être utilisée dans l'expression. La valeur fournie dans le contexte est alors utilisée lors de l'évaluation de l'expression.

Exemple :

```
package fr.jmdoudoux.dej.spring;

import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;

public class TestParser {

  public final static ExpressionParser EXPRESSION_PARSER = new SpelExpressionParser();

  public static void main(final String[] args) {

    StandardEvaluationContext context = new StandardEvaluationContext();
    context.setVariable("maVariable", "test");
    String valeur = EXPRESSION_PARSER.parseExpression("#maVariable").getValue(
        context, String.class);
    System.out.println(valeur);
  }
}
```

91.5.3. Des exemples de mise en oeuvre

Dans l'exemple ci-dessous, un objet de type Double est déclarée en invoquant la méthode random() de la classe java.lang.Math. Cet objet est fourni en dépendance de la propriété maValeur d'un bean de type MonBean.

Exemple :

```
  <bean id="nombreAleatoire" class="java.lang.Math" factory-method="random" />
  <bean id="monBean" class="fr.jmdoudoux.dej.spring.MonBean"
    p:maValeur-ref="nombreAleatoire" />
```

Il est possible d'utiliser SpEL pour obtenir la valeur du bean.

Exemple :

```
  <bean id="nombreAleatoire" class="java.lang.Math" factory-method="random" />
  <bean id="monBean" class="fr.jmdoudoux.dej.spring.MonBean"
    p:maValeur="#{nombreAleatoire}" />
```

L'expression "#{nombreAleatoire}" sera évaluée pour faire référence au bean dont l'identifiant est nombreAleatoire.

L'exemple ci-dessous va utiliser SpEL dans la configuration des beans du contexte pour définir une valeur aléatoire et utiliser cette valeur en paramètre du constructeur d'un bean.

Exemple :

```
  <bean id=" nombreAleatoire" class="java.lang.Integer">
    <constructor-arg value="#{T(java.lang.Math).random() " />
  </bean>
  <bean id="valeur" class="java.lang.Integer">
    <constructor-arg value="#{nombreAleatoire.intValue()}" />
  </bean>
```

L'exemple ci-dessous va récupérer une variable d'environnement de la JVM pour utiliser différents fichiers de configuration selon cette variable.

Exemple :

```
  <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="location"
      value="#{systemProperties['environnement'] ?:'dev'}/conf.properties" />
  </bean>
```

Des fichiers de configuration par environnement sont définis, chacun dans son sous-répertoire.

Résultat :

```
src/main/resources/dev/conf.properties
src/main/resources/int/conf.properties
src/main/resources/prod/conf.properties
```

Au lancement de l'application, la variable d'environnement « environnement » est définie.

Exemple :

```
java -jar monapp.jar -denvironnement=prod
```

L'exemple ci-dessous va utiliser SpEL avec l'annotation @Value

Exemple :

```
package fr.jmdoudoux.dej.spring;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class MaClasse {
  private final String valeur;

  @Autowired
  public MaClasse(@Value("#{systemProperties['valeur']}") final String valeur)
  {
    this.valeur = valeur;
  }

  public String getValeur() {
    return this.valeur;
  }
}
```

SpEL peut manipuler des chaînes de caractères.

Exemple :

```
    ExpressionParser parser = new SpelExpressionParser();
    Expression exp =parser.parseExpression("'Test'");
    System.out.println((String) exp.getValue());
    System.out.println(exp.getValue(String.class));
    exp = parser.parseExpression("new String('Test').toUpperCase()");
    System.out.println((String) exp.getValue());
```

91.5.4. La syntaxe de SpEL

La syntaxe de SpEL est proche de celle d'Unified EL.

Les variables utilisables avec SpEL sont :

- le nom de chaque bean du contexte Spring
- systemProperties
- systemEnvironment
- request, session, contextParameters et contextAttributes dans un contexte web

Une exception de type SpelParseException est levée lors de l'évaluation de l'expression si celle-ci n'est pas syntaxiquement correcte.

91.5.4.1. Les types de base

Les types de base se composent :

- De chaînes de caractères
- De valeurs primitives

Les chaînes de caractères sont entourées par de simples quotes.

Exemple :

```
    Expression exp = EXPRESSION_PARSER.parseExpression("'Hello'");
    String valeur = exp.getValue(String.class);
    System.out.println(valeur);
```

Les valeurs primitives sont supportées comme dans tout autre langage d'expressions.

Exemple :

```
    System.out.println(EXPRESSION_PARSER.parseExpression("0").getValue(
        byte.class));
    System.out.println(EXPRESSION_PARSER.parseExpression("0").getValue(
        short.class));
    System.out.println(EXPRESSION_PARSER.parseExpression("0").getValue(
        int.class));
    System.out.println(EXPRESSION_PARSER.parseExpression("0L").getValue(
        long.class));
    System.out.println(EXPRESSION_PARSER.parseExpression("0.1F").getValue(
        float.class));
    System.out.println(EXPRESSION_PARSER.parseExpression("0.1D").getValue(
        double.class));
    System.out.println(EXPRESSION_PARSER.parseExpression("true").getValue(
        boolean.class));
    System.out.println(EXPRESSION_PARSER.parseExpression("'a'").getValue(
        char.class));
```

91.5.4.2. L'utilisation d'objets

SpEL permet un accès aux membres d'un bean.

L'exemple ci-dessous crée une nouvelle instance de MonBean et initialise sa propriété id.

Exemple :

```
ExpressionParser parser = new SpelExpressionParser();
MonBean monBean = parser.parseExpression(
    "new fr.jmdoudoux.dej.spring.MonBean()").getValue(MonBean.class);

StandardEvaluationContext context = new StandardEvaluationContext(monBean);
context.setVariable("monId", "1234");
parser.parseExpression("id=#monId").getValue(context);
```

La méthode setVariable() permet d'affecter une valeur à la variable monId dans le contexte.

L'expression "id=#monId" est évaluée en utilisant le contexte : la propriété id du bean est initialisée avec la valeur de la variable monId.

SpEL permet un accès aux propriétés d'un bean en utilisant la syntaxe avec un point.

Exemple :

```
Expression exp = EXPRESSION_PARSER.parseExpression("'Hello'.bytes");
byte[] bytes = exp.getValue(byte[].class);
System.out.println(bytes);
```

L'accès aux propriétés peut être chaîné.

Exemple :

```
Expression exp = EXPRESSION_PARSER.parseExpression("'Hello'.bytes.length");
int taille = exp.getValue(int.class);
System.out.println(taille);
```

91.5.4.3. Les opérateurs

SpEL offre un large éventail d'opérateurs. Il propose des opérateurs logiques, mathématiques et relationnels standards.

Exemple :

```
System.out.println(EXPRESSION_PARSER.parseExpression("true and true")
    .getValue(boolean.class));
System.out.println(EXPRESSION_PARSER.parseExpression("true or true")
    .getValue(boolean.class));
System.out.println(EXPRESSION_PARSER.parseExpression("!false").getValue(
    boolean.class));
System.out.println(EXPRESSION_PARSER.parseExpression("not false").getValue(
    boolean.class));
System.out.println(EXPRESSION_PARSER.parseExpression("true and not false")
    .getValue(boolean.class));
```

SpEL propose tous les opérateurs relationnels standard.

Exemple :

```
System.out.println(EXPRESSION_PARSER.parseExpression("2==2").getValue(
    boolean.class));
System.out.println(EXPRESSION_PARSER.parseExpression("2<3").getValue(
    boolean.class));
System.out.println(EXPRESSION_PARSER.parseExpression("3>2").getValue(
    boolean.class));
System.out.println(EXPRESSION_PARSER.parseExpression("0!=1").getValue(
    boolean.class));
```

SpEL propose tous les opérateurs mathématiques standard.

Exemple :

```
    System.out.println(EXPRESSION_PARSER.parseExpression("2+2").getValue(
        int.class));
    System.out.println(EXPRESSION_PARSER.parseExpression("2-2").getValue(
        int.class));
    System.out.println(EXPRESSION_PARSER.parseExpression("2/2").getValue(
        int.class));
    System.out.println(EXPRESSION_PARSER.parseExpression("2*2").getValue(
        int.class));
    System.out.println(EXPRESSION_PARSER.parseExpression("2^2").getValue(
        int.class));
    System.out.println(EXPRESSION_PARSER.parseExpression("1e10").getValue(
        double.class));
```

SpEL permet la concaténation de chaînes de caractères

Exemple :

```
    System.out.println(EXPRESSION_PARSER.parseExpression("'Hello'+' world'")
        .getValue(String.class));
```

SpEL propose un support de l'opérateur ternaire

Exemple :

```
    System.out.println(EXPRESSION_PARSER.parseExpression(
        "true ? 'vrai' : 'faux'").getValue(String.class));
```

SpEL propose un support de l'opérateur elvis qui est une forme simplifiée de l'opérateur ternaire.

Exemple :

```
    System.out.println(EXPRESSION_PARSER.parseExpression("null?:'sans valeur'")
        .getValue(String.class));
```

91.5.4.4. L'utilisation de types

SpEL peut accéder aux instances de java.lang.Class en utilisant l'opérateur T (T signifiant Type).

Il faut fournir à l'opérateur T le nom pleinement qualifié de la classe sauf pour les classes du package java.lang.

Résultat :

```
T(java.util.Math)
T(Integer)
```

Avec l'opérateur T, il est possible d'accéder aux membres static d'une classe.

Exemple :

```
<bean id="monBean3" class="fr.jmdoudoux.dej.spring.MonBean"
p:maValeur="#{T(java.util.Math).random()}"
p:monLibelle="#{T(String).format('Hello %s', 'world')}"
p:valeurMax="#{T(INTEGER).MAX_VALUE}"/>
```

L'opérateur instanceOf permet de tester si une variable est d'un certain type.

Exemple :

```
<bean id="monBean" class="fr.jmdoudoux.dej.spring.MonBean" p:maChaine="Test" />
<bean id="monAutreBean" class="fr.jmdoudoux.dej.spring.MonAutreBean"
  p:isString="#{monBean.maChaine instanceof T(String)}" />
```

L'opérateur instanceof permet de vérifier qu'un objet est d'un type donné : le type doit être précisé en utilisant l'opérateur T.

Exemple :

```
    boolean estUnEntier = EXPRESSION_PARSER.parseExpression(
        "0 instanceof T(Integer)").getValue(boolean.class);
    System.out.println(estUnEntier);
```

91.5.4.5. L'invocation d'un constructeur ou d'une méthode

L'opérateur new permet de créer une nouvelle instance d'une classe.

Exemple :

```
  <bean id="monBean" class="fr.jmdoudoux.dej.spring.MonBean">
    <property name="dateDebut" value="#{new java.util.Date()}" />
  </bean>
```

L'exemple ci-dessous utilise SpEL pour créer une nouvelle instance d'un bean en invoquant un de ses constructeurs avec paramètres.

Exemple :

```
    ExpressionParser parser = new SpelExpressionParser();
    MonBean monBean = parser.parseExpression(
        "new  fr.jmdoudoux.dej.spring.MonBean('test',1234,false)").getValue(
        MonBean.class);
```

Il est obligatoire d'utiliser le nom pleinement qualifié de la classe à instancier.

SpEL permet l'invocation d'une méthode dans une expression, ce qui est une fonctionnalité intéressante pour un langage d'expressions.

Exemple :

```
  <bean id="monBean" class="fr.jmdoudoux.dej.spring.MonBean"
  p:isFichierPresent="#{new java.io.File('/tmp/monFichier.tmp').exists()}" />
```

Dans cet exemple, la propriété isFichierPresent est initialisée avec le résultat de l'évaluation de la condition d'existence du fichier.

Exemple :

```
    Expression exp = EXPRESSION_PARSER
        .parseExpression("'Hello'.concat(' World')");
    String valeur = exp.getValue(String.class);
    System.out.println(valeur);
```

91.5.4.6. L'utilisation d'expressions régulières

SpEL propose un support des expressions régulières avec l'opérateur matches qui renvoie un booléen.

Exemple :

```
  <util:map id="regExpsExemples" value-type="java.lang.Boolean"
    key-type="java.lang.String">
    <entry key="cas1" value="#{'a demain' matches 'a.*'}" />
    <entry key="cas2" value="#{'a demain' matches 'b.*'}" />
  </util:map>
```

Exemple :

```
package fr.jmdoudoux.dej.spring;

import java.util.Map;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestParser {

  private static ClassPathXmlApplicationContext appContext;

  public static void main(final String[] args) {

    appContext = new ClassPathXmlApplicationContext("application-context.xml");

    Map<String, Boolean> map = (Map) appContext.getBean("regExpsExemples");
  }
}
```

L'opérateur matches permet d'appliquer une expression régulière : il renvoie un booléen qui précise si l'évaluation de l'expression correspond.

Exemple :

```
    boolean resultat = EXPRESSION_PARSER.parseExpression(
        "+10.123e-4 matches '[-+]?[0-9]*\\.?[0-9]+([eE][-+]?[0-9]+)?$'")
        .getValue(boolean.class);
```

SpEL est capable de parser une date.

Exemple :

```
    Date date = EXPRESSION_PARSER.parseExpression("'2011/03/31'").getValue(
        Date.class);
```

91.5.4.7. La manipulation de collections

SpEL propose quelques fonctionnalités particulières concernant les collections notamment :

- La sélection dans une collection
- La projection de collections

SpEL permet de faire une sélection dans une collection en appliquant un filtre pour créer une nouvelle collection qui est un sous-ensemble de la collection d'origine.

L'opérateur de sélection possède la syntaxe ?[selection-expression] à utiliser sur une collection.

Exemple :

```
  <util:list id="prenoms">
    <value>Alain</value>
    <value>Aurelie</value>
    <value>Bruno</value>
    <value>Charles</value>
    <value>Laure</value>
    <value>Maurice</value>
    <value>Michel</value>
    <value>Monique</value>
    <value>Sylvie</value>
    <value>Thierry</value>
    <value>Veronique</value>
  </util:list>

  <util:map id="prenomsCommencantParM" value-type="java.lang.Object"
    key-type="java.lang.String">
    <entry key="M" value="#{prenoms.?[startsWith('M')]}" />
  </util:map>
```

Dans l'exemple ci-dessus, SpEL est utilisé pour créer une collection des prénoms commençants par la lettre M.

Exemple :

```
  <util:map id="premierPrenomCommencantParM" value-type="java.lang.Object"
    key-type="java.lang.String">
    <entry key="M Premier" value="#{prenoms.^[startsWith('M')]}" />
    <entry key="M Dernier" value="#{prenoms.$[startsWith('M')]}" />
  </util:map>
```

Il est possible d'obtenir le premier élément correspondant à l'expression grâce à l'opérateur ^[selection-expression] ou le dernier élément grâce à l'opérateur $[selection-expression].

SpEL permet de faire une projection qui permet d'obtenir des données des éléments d'une collection pour créer une nouvelle collection.

L'opérateur de projection crée une nouvelle collection en évaluant une expression sur les éléments d'une collection source.

L'opérateur de projection possède la syntaxe ![expression] à utiliser sur une collection qui ne doit pas être de type Set.

SpEL permet la sélection dans une collection de type List.

Exemple :

```
    List<Integer> nombres = new ArrayList<Integer>(Arrays.asList(1, 2, 3, 4, 5,
        6, 7, 8, 9, 10));
    EvaluationContext context = new StandardEvaluationContext();
    context.setVariable("list", nombres);
    List<?> nombrePairs = EXPRESSION_PARSER.parseExpression(
        "#list.?[#this%2==0]").getValue(context, List.class);
```

La variable #this fait référence à l'élément courant dans la liste durant son parcours.

SpEL permet la sélection dans une collection de type Map

SpEL permet d'obtenir une valeur d'une collection de type Map avec une syntaxe similaire à celle d'un tableau dans laquelle la valeur de la clé est fournie entre les crochets.

Exemple :

```
    Map<String, String> map = new HashMap<String, String>();
    map.put("cle1", "valeur1");
    EvaluationContext context = new StandardEvaluationContext();
    context.setVariable("map", map);
    Expression exp = EXPRESSION_PARSER.parseExpression("#map['cle1']");
    String valeur = exp.getValue(context, String.class);
```

La projection de collections (Collection projection) permet d'extraire les éléments d'une collection en filtrant sur un motif commun aux différentes valeurs. Dans l'exemple ci-dessous, tous les nom* seront extraits.

Exemple :

```
    List<Personne> list = new ArrayList<Personne>(Arrays.asList(new Personne(
        "noml"), new Personne("nom2"), new Personne("nom3")));
    List<String> noms = (List<String>) EXPRESSION_PARSER.parseExpression(
        "#root.![nom]").getValue(list);
```

91.6. La configuration en utilisant les annotations

A partir de la version 2.5 de Spring et en utilisant une version 5 ou supérieure de Java, il est possible d'utiliser des annotations pour réaliser une partie de la configuration des beans.

Comme lors de toutes utilisations d'annotations à la place d'un fichier XML, la configuration est décentralisée mais aussi grandement simplifiée tout en réduisant la taille du fichier de configuration.

Le conteneur doit être informé de l'utilisation d'annotations pour réaliser une partie de la configuration. L'espace de nommage context propose le tag <annotation-config> qui permet de préciser l'utilisation des annotations dans la configuration.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">
  <context:annotation-config/>
</beans>
```

Spring 2.5 propose ses propres annotations et un support des annotations @Resource, @PostConstruct et @PreDestroy définies dans Java 5.

L'utilisation des annotations ne peut pas remplacer intégralement le fichier de configuration : il faut au moins y déclarer les PostBeanProssessor et certains composants techniques (DataSource, TransactionManager, ...) pour lesquels la déclaration n'est pas possible par annotations.

Il est aussi possible de mixer les déclarations en utilisant le fichier de configuration et les annotations. Cependant, pour faciliter la maintenance, il faut rester le plus cohérent possible, par exemple, en déclarant l'autowiring dans le fichier de configuration ou par annotations mais en éviter de panacher les deux.

Remarque : la configuration par le fichier XML est prioritaire sur la configuration faite avec les annotations. Ainsi, si un bean est déclaré comme étant un singleton dans le fichier de configuration et qu'il est annoté avec l'annotation @Scope("protype"), le bean sera géré comme un singleton par le conteneur.

91.6.1. L'annotation @Scope

Cette annotation introduite par Spring 2.5 permet de préciser le cycle de vie du bean. Elle s'applique sur une classe.

Les valeurs utilisables sont : singleton, prototype, session et request. Ces deux dernières ne sont utilisables que dans des applications web.

Comme pour la définition des beans dans le fichier de configuration, le scope par défaut est singleton pour un bean déclaré grâce aux annotations.

L'annotation @Scope permet de préciser une portée différente de singleton.

Exemple :

```
@Controller
@Scope("prototype")
public class MonController {
  // ...
}
```

L'annotation @Scope est prise en charge par le conteneur par un objet de type org.springframework.context.annotation.ScopeMetadataResolver.

91.6.2. L'injection de dépendances avec les annotations

Spring 2.5 propose la possibilité de réaliser une partie de la configuration relative à l'injection de dépendance grâce à des annotations.

Quatre annotations sont utiles dans ce but : @Autowired, @Configurable, @Qualifier et @Resource.

Il ne faut pas oublier de déclarer le namespace context et l'utilisation du schéma spring-context dans le fichier de configuration du contexte.

Le tag <annotation-config> va déclarer plusieurs BeanPostProcessor dont RequiredAnnotationBeanPostProcessor, AutowiredAnnotationBeanPostProcessor, CommonAnnotationBeanPostProcessor et PersistenceAnnotationBeanPostProcessor.

Il faut ensuite indiquer au conteneur les packages qu'il devra scanner pour rechercher les classes annotées. La balise <context:component-scan> n'est obligatoire que si les annotations concernant des stéréotypes sont utilisées.

Dans la configuration, il faut utiliser le tag <component-scan> de l'espace de nommage context. L'attribut base-package permet de préciser le nom du package : le contenu du package et de ses sous-packages seront scannés.

Exemple :

```
<context:component-scan base-package="fr.jmdoudoux.dej.spring.services"/>
```

Il est possible d'avoir un contrôle très fin sur les classes à scanner en utilisant des filtres.

Exemple :

```
<context:component-scan
      base-package="fr.jmdoudoux.dej.spring.services" use-default-filters="false">
  <context:exclude-filter type="annotation"
    expression="org.springframework.stereotype.Repository"/>
</context:component-scan>
```

L'annotation @Autowired permet de demander une injection automatique par type.

L'annotation @Qualifier permet de faciliter la détermination du bean à injecter grâce à l'annotation @Autowired dans le cas où plusieurs instances gérées par le conteneur correspondent au type souhaité.

L'annotation @Configurable permet de déclarer une classe dont les dépendances seront gérées par le conteneur alors que les instances ne le sont pas. Ces dépendances sont injectées lors de l'invocation du constructeur de la classe. Cette annotation requiert la mise en oeuvre d'AOP avec AspectJ.

L'annotation @Resource permet une injection automatique par nom : elle est définie dans la JSR 250.

91.6.2.1. L'annotation @Required

L'annotation @org.springframework.beans.factory.annotation.Required introduite par Spring 2.0 permet de valider une injection de dépendances quelle que soit la méthode qui a permis cette injection.

Lors de la création d'une instance du bean, le conteneur va s'assurer que la propriété est valorisée à la fin de l'initialisation du bean soit explicitement soit via l'autowiring.

Si à la fin de l'initialisation du bean, la propriété n'est pas valorisée alors une exception de type org.springframework.beans.factory.BeanInitializationException est levée par le conteneur.

L'utilisation de cette annotation peut éviter d'avoir une exception de type NullPointerException levée lors de l'exécution des traitements de la classe.

L'annotation @org.springframework.beans.factory.annotation.Required s'utilise sur le setter d'une propriété car elle ne peut s'appliquer que sur une méthode.

Exemple :

```
public class PersonneService {

  private PersonneDao personneDao;

  @Required
  public void setPersonneDao(PersonneDao personneDao) {
    this.personneDao = personneDao;
  }
}
```

L'annotation @Required est traitée dans le conteneur par RequiredAnnotationBeanPostProcessor.

91.6.2.2. L'annotation @Autowired

L'annotation @org.springframework.beans.factory.annotation.Autowired introduite par Spring 2.5 permet de faire de l'injection automatique de dépendances basée sur le type.

L'utilisation de l'espace de nommage context permet d'activer l'utilisation des annotations @Autowired en utilisant <context:annotation-config/>.

L'attribut required permet de préciser si l'injection d'une instance dans la propriété est obligatoire ou non. Par défaut, sa valeur est true.

Exemple :

```
@Autowired(required=false)
private MaClasse maClasse;
```

Il est possible de préciser une valeur par défaut si l'injection automatique de la dépendance ne s'est pas faite.

Exemple :

```
@Autowired(required=false)
private MaClasse maClasse = new MaClasseImpl;
```

L'annotation @Autowired s'utilise sur une propriété, un setter ou un constructeur.

Exemple d'injection sur un champ :

```
public class PersonneService {

  @Autowired
  private PersonneDao personneDao;

  public void setPersonneDao(PersonneDao personneDao) {
    this.personneDao = personneDao;
  }

  // methodes de la classe qui peuvent utiliser la dependance
}
```

L'annotation @Autowired peut être utilisée sur une méthode ou un constructeur qui attend un ou plusieurs paramètres.

Exemple d'injection sur un constructeur :

```
public class PersonneService {

  private PersonneDao personneDao;

  @Autowired
  public PersonneService(PersonneDao personneDao) {
    this.personneDao = personneDao;
  }
}
```

Exemple d'injection sur un setter :

```
public class PersonneService {

  private PersonneDao personneDao;

  @Autowired
  public void setPersonneService(PersonneDao personneDao) {
    this.personneDao = personneDao;
  }
}
```

Exemple d'injection sur une méthode

Exemple :

```
public class PersonneService {

  private PersonneDao personneDao;

  @Autowired
  public void initialiser(PersonneDao personneDao, boolean utiliserCache) {
    this.personneDao = personneDao;
  }
}
```

L'annotation @Autowired peut être utilisée sur plusieurs constructeurs mais un seul constructeur peut être annoté avec @Autowired ayant l'attribut required à true (qui est la valeur par défaut).

@Autowired peut aussi être utilisée sur des tableaux et des collections typées (Collection, List, Set, Map). La collection sera alors automatiquement remplie avec les occurrences du type gérées par le conteneur.

Il est possible d'obtenir toutes les instances d'un type géré par le conteneur en appliquant @Autowired sur une propriété qui est un tableau ou une collection typée.

Exemple :

```
public class MonService {

  @Autowired
  private BaseService[] services;

  public void setServices(BaseService[] services) {
    this.services = services;
  }
}
```

Exemple :

```
public class MonService {

  private Set<BaseService> services;

  @Autowired
  public void setServices(Set<BaseService> services) {
    this.services = services;
  }
}
```

Il est aussi possible d'utiliser une collection de type Map avec des clés de type String qui seront les idenfiants des beans gérés par le conteneur, les beans eux-mêmes étant les valeurs associées.

Exemple :

```
public class MonService {

  private Map<String, BaseService> services;

  @Autowired
  public void setServices(Map<String, BaseService> services) {
    this.services = services;
  }
}
```

Par défaut, le conteneur va lever une exception s'il n'arrive pas à trouver une instance qui corresponde au type à auto injecter. L'attribut required de l'annotation @Autowired permet de modifier ce comportement : sa valeur par défaut est true. L'utilisation de cet attribut est préférable à l'utilisation de l'annotation @Required.

L'annotation @Autowired permet aussi l'injection d'une instance de plusieurs types d'objets de Spring notamment BeanFactory et ApplicationContext.

Exemple :

```
public class MaClasse {

  @Autowired
  private ApplicationContext context;

}
```

L'annotation @Autowired est traitée dans le conteneur (BeanFactory ou ApplicationContext) par la classe AutowiredAnnotationBeanPostProcessor. Un AutowiredAnnotationBeanPostProcessor est automatiquement configuré en utilisant les annotations <context:component-scan> ou <context:annotation-config> dans le fichier de configuration.

Le conteneur va lever une exception de type BeanCreationException si :

- @Autowired ne peut trouver aucun bean correspondant au type de l'entité annotée sauf si l'attribut required est à false
- @Autowired trouve plusieurs beans correspondants au type et que l'élément annoté n'est pas un tableau ou une collection

L'attribut primary du tag <bean> permet de définir le bean qui sera utilisé prioritairement lors de l'injection de dépendance par type.

L'exemple ci-dessous est un exemple complet d'une petite application standalone qui demande un service au conteneur Spring et l'utilise pour afficher un message. Ce service possède une dépendance auto-injectée vers un DAO qui renvoie le message à afficher.

La fichier de configuration du contexte ne contient que deux tags.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:context="http://www.springframework.org/schema/context"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

        <context:annotation-config />
        <context:component-scan base-package="fr.jmdoudoux.dej.spring"/>

</beans>
```

Le DAO est défini par une interface.

Exemple :

```
package fr.jmdoudoux.dej.spring;

public interface MonDao {
  public abstract String getMessage();
}
```

Le DAO implémente son interface.

Exemple :

```
package fr.jmdoudoux.dej.spring;

import org.springframework.stereotype.Repository;

@Repository("monDao")
public class MonDaoImpl implements MonDao {

  @Override
  public String getMessage() {
    return "Bonjour";
  }
}
```

Le service est défini par une interface.

Exemple :

```
package fr.jmdoudoux.dej.spring;
import org.springframework.stereotype.Service;

public interface MonService {
  public void afficher();
}
```

Le service implémente son interface et possède une dépendance vers le DAO.

Exemple :

```
package fr.jmdoudoux.dej.spring;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service("monService")
public class MonServiceImpl implements MonService {

  @Autowired
  private MonDaoImpl monDao;

  @Override
  public void afficher() {
    System.out.println(monDao.getMessage());
  }
}
```

Une classe de test permet de demander une instance au conteneur Spring et d'invoquer sa méthode afficher().

Exemple :

```
import org.springframework.context.support.ClassPathXmlApplicationContext;
import fr.jmdoudoux.dej.spring.MonService;

public class Main {

  private static ClassPathXmlApplicationContext appContext;

  public static void main(final String[] args) {
    appContext = new ClassPathXmlApplicationContext("application-context.xml");

    try {
      MonService monService = appContext.getBean(MonService.class);
      monService.afficher();
    } catch (final Exception e) {
      e.printStackTrace();
      System.exit(1);
    }
  }
}
```

Résultat :

```
Bonjour
```

L'auto injection en utilisant les annotations a des avantages mais aussi des inconvénients.

L'injection est quelque chose qui doit être configuré mais l'utilisation des annotations nécessite une recompilation lors du changement ou de l'ajout d'une annotation. Généralement, ces changements imposent aussi des modifications dans le code, ce qui limite ce petit inconvénient.

91.6.2.3. L'annotation @Qualifier

L'annotation @Qualifier permet de qualifier le candidat à une injection automatique avec son nom. Ceci est particulièrement utile lorsque plusieurs instances sont du type à injecter.

Lors de la détermination d'une instance à injecter automatiquement par autowiring, se basant donc sur le type, il est possible que plusieurs instances gérées par le conteneur soit éligibles notamment lorsque le type à injecter est une interface ou une classe abstraite. Il faut alors aider le conteneur à choisir la bonne instance en utilisant l'annotation @Qualifier.

L'annotation org.springframework.beans.factory.annotation.Qualifier a été introduite par Spring 2.5. Elle peut s'appliquer sur un champ, une méthode ou un constructeur ou sur un paramètre d'un champ ou un constructeur.

Exemple sur un champ :

```
@Autowired
@Qualifier("personne1")
private Personne personne;
```

Exemple sur un setter :

```
@Autowired
public void setPersonne(@Qualifier("personne1") Personne personne) {
  this.personne = personne;
}
```

Exemple sur un paramètre d'une méthode :

```
@Autowired
public void initialiser(@Qualifier("personne1") Personne personne) {
  this.personne = personne;
}
```

Exemple sur un constructeur :

```
@Autowired
public PersonneService(@Qualifier("personne1") Personne personne) {
  this.personne = personne;
}
```

L'annotation permet de préciser une valeur qui sera utilisée par le conteneur comme qualificateur pour déterminer le bean de l'instance à utiliser lors de l'injection.

L'annotation @Qualifier s'utilise en conjonction avec l'annotation @Autowired.

L'utilisation la plus facile de @Qualifier est de lui fournir en paramètre le nom du bean concerné et de l'appliquer sur la propriété qui sera injectée : dans ce cas l'injection se fait par nom et non par type, ce qui revient à utiliser l'annotation @Resource lorsqu'elle est utilisée sur un champ ou une méthode. Cependant, @Autowired et @Qualifier peuvent être utilisés sur un constructeur ou une méthode ayant plusieurs paramètres.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

  <context:annotation-config />

  <bean id="personneService" class="fr.jmdoudoux.dej.spring.service.PersonneServiceImpl" />

  <bean id="personne1" class="fr.jmdoudoux.dej.spring.entite.Personne">
    <property name="nom"    value="nom1" />
    <property name="prenom" value="prenom1" />
  </bean>
  <bean id="personne2" class="fr.jmdoudoux.dej.spring.entite.Personne">
    <property name="nom"    value="nom2" />
    <property name="prenom" value="prenom2" />
  </bean>

</beans>
```

Dans l'exemple ci-dessus, deux beans de type Personne sont déclarés. Leurs identifiants sont utilisés comme valeurs de l'attribut de l'annotation @Qualifier.

Exemple :

```
package fr.jmdoudoux.dej.spring.service;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.TransactionDefinition;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.transaction.support.DefaultTransactionDefinition;

import fr.jmdoudoux.dej.spring.MonException;
import fr.jmdoudoux.dej.spring.entite.Personne;

public class PersonneServiceImpl implements PersonneService {

  @Autowired(required = true)
  @Qualifier("personne1")
  private Personne personne1;

  @Autowired(required = true)
  @Qualifier("personne2")
  private Personne personne2;

  @Override
  public void afficher() {
    System.out.println(personne1);
    System.out.println(personne2);
  }
}
```

Il est aussi possible d'utiliser l'annotation @Qualifer sur la classe pour lui associer un qualificateur qui pourra être utilisé à la place du nom du bean.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

 <context:component-scan base-package="fr.jmdoudoux.dej.spring"/>

  <bean id="personneService" class="fr.jmdoudoux.dej.spring.service.PersonneServiceImpl" />
</beans>
```

L'annotation @Qualifier est utilisée pour associer un qualificateur à la classe : c'est une sorte de marqueur qui va permettre de préciser lors d'une injection automatique par type que c'est une instance de ce type que l'on souhaite.

Exemple :

```
package fr.jmdoudoux.dej.spring.entite;

import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;

@Qualifier("pers1")
@Component("personne1")
public class Personne1 extends Personne {

}
```

Exemple :

```
package fr.jmdoudoux.dej.spring.entite;

import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;

@Qualifier("pers2")
@Component("personne2")
public class Personne2 extends Personne {

}
```

L'annotation @Qualifier est utilisée sur les propriétés à injecter pour préciser le qualificateur qui va permettre au conteneur de sélectionner l'instance à injecter.

Exemple :

```
package fr.jmdoudoux.dej.spring.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;

import fr.jmdoudoux.dej.spring.entite.Personne;

public class PersonneServiceImpl implements PersonneService {

  @Autowired(required = true)
  @Qualifier("pers1")
  private Personne personne1;

  @Autowired(required = true)
  @Qualifier("pers2")
  private Personne personne2;

  @Override
  public void afficher() {
    System.out.println(personne1);
    System.out.println(personne2);
  }
}
```

L'annotation @Autowired étant naturellement utilisée pour une injection par type, il est préférable de donner une valeur sémantique à un qualificateur.

Il est aussi possible de définir ses propres annotations qui seront des qualifiers : pour cela, il faut définir une annotation qui soit elle-même annotée avec @Qualifier.

Exemple :

```
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface MonQualifier {

  String value();

}
```

Une fois défini, il est possible d'utiliser le nouveau qualifieur.

Exemple :

```
@Autowired
@MonQualifier
private Personne personne;
```

Il est possible de définir des attributs pour un qualificateur personnalisé.

Exemple :

```
@Qualifier
public @interface MonQualifier {
  String indicateur();
  int niveau();
}
```

Exemple :

```
@Category(indicateur="employe", niveau="3")
public class Personne {
// ...
}
```

Il est aussi possible de définir un qualificateur dans le fichier de configuration donc sans utiliser d'annotation mais en déclarant un bean de type CustomAutowireConfigurer.

Exemple :

```
<bean class="org.springframework.beans.factory.annotation.CustomAutowireConfigurer">
  <property name="customQualifierTypes">
    <set>
      <value>fr.jmdoudoux.dej.spring.MonQualifier</value>
      <value>org.example.Offline</value>
    </set>
  </property>
</bean>
```

Pour qualifier un bean déclaré dans le fichier de configuration, il faut utiliser le tag <qualifier> fils du tag <bean>. Son attribut value permet de définir la valeur du qualificateur.

Exemple :

```
<bean id="personne" class="fr.jmdoudoux.dej.spring.entite.Personne">
  <qualifier type="MonQualifier" />
</bean>
```

91.6.2.4. L'annotation @Resource

L'annotation @javax.annotation.Resource permet de demander l'injection d'un bean par son nom.

Son support est proposé depuis la version 2.5 de Spring. La spécification de cette annotation est faite dans la JSR 250 (commons annotations).

L'annotation @Resource est fournie en standard avec Java SE 6 : pour l'utiliser avec Java 5, il faut ajouter le jar de l'implémentation de la JSR 250.

Elle s'utilise sur un champ ou une méthode.

Elle est prise en charge par le CommonAnnotationBeanPostProcessor.

Exemple :

```
<bean class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor"/>
```

Sans attribut, cette annotation agit comme l'annotation @Autowired en permettant l'injection de beans. La différence est que @Resource propose la résolution par nom alors que @Autowired propose la résolution par type.

Exemple :

```
public class PersonneDaoImpl implements PersonneDao {
  @Resource
  private DataSource dataSource;
}
```

Exemple :

```
private DataSource dataSource;

@Resource
public void setDataSource(DataSource dataSource) {
  this.dataSource = dtasource ;
}
```

L'annotation tente de déterminer le bean concerné en recherchant celui dont le nom correspond à l'attribut name de @Resource. Sinon, la recherche s'effectue sur le nom de la propriété s'il est fourni et, pour terminer, sur le type du bean puisque les recherches par noms ont échoué.
Pour demander l'injection d'un bean précis, il faut fournir son identifiant comme valeur de l'attribut name.

Exemple :

```
public class PersonneDaoImpl implements PersonneDao {
  @Resource(name="maDataSource")
  private DataSource dataSource;
}
```

Exemple :

```
private DataSource dataSource;

@Resource(name="maDataSource")
public void setDataSource(DataSource dataSource) {
  this.dataSource = dtasource ;
}
```

Il est possible de désactiver la recherche par type si la recherche par nom échoue en passant la valeur false à la propriété fallbackToDefaultTypeMatch de l'instance de type CommonAnnotationBeanPostProcessor.

Exemple :

```
<bean class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor">
  <property name="fallbackToDefaultTypeMatch" value="false"/>
</bean>
```

La propriété alwaysUseJndiLookup est un booléen possédant la valeur false par défaut qui permet de demander la résolution de la dépendance annotée avec @Resource dans un annuaire JNDI.

91.6.2.5. L'annotation @Configurable

Spring 2.0 permet d'utiliser l'AOP pour intercepter l'invocation d'un constructeur et réaliser l'injection des dépendances de la classe.

L'annotation @org.springframework.beans.factory.annotation.Configurable introduite par Spring 2.0 permet d'indiquer à Spring qu'il pourra injecter les dépendances d'un bean bien que son conteneur ne gère pas son cycle de vie.

L'annotation @Configurable permet de demander à Spring d'ajouter des aspects afin d'injecter les dépendances de la classe lors de l'invocation du constructeur. Ainsi, même si la classe n'est pas gérée par Spring, Spring sera en mesure d'injecter ses dépendances.

Typiquement cela concerne des objets qui ne sont pas gérés par le conteneur de Spring par exemple des objets du domaine ou une servlet.

Son exploitation se fait de façon transparente avec l'AOP : le bean est tissé avec des aspects qui vont assurer l'injection de dépendances notamment lors de l'invocation du constructeur.

Pour que l'annotation @Configurable soit prise en charge, il faut ajouter le tag <context:spring-configured/> dans la configuration du contexte Spring.

Exemple :

```
<context:spring-configured />
```

Spring tisse les aspects qui vont se charger de réaliser l'injection des dépendances lors de l'invocation du constructeur de la classe. Ce tissage peut se faire de plusieurs manières :

- en utilisant le compilateur d'AspectJ qui va enrichir le bytecode de la classe
- en utilisant le tissage au runtime d'AspectJ qui va utiliser un classloader dédié pour enrichir le bytecode de la classe au moment de son chargement dans la JVM

Pour utiliser le tissage au runtime, il faut définir un agent au lancement de la JVM en lui ajoutant l'option "-javaagent:aspectjweaver-1.5.0.jar". Il est aussi nécessaire de rendre le fichier META-INF/aop.xml accessible dans le classpath

Exemple :

```
<aspectj>
  <weaver options="-verbose">
    <include within="fr.jmdoudoux.dej.spring.*" />
    <exclude within="*..*CGLIB*" />
  </weaver>
</aspectj>
```

Les tags <include> et <exclude> permettent de limiter les classes qui seront concernées par le tissage. Cela améliore les performances de ce processus qui se limite aux classes concernées.

Dans l'exemple, les proxys générés par Hibernate sont exclus du tissage.

Chaque fois qu'une instance de la classe sera créée, Spring va assurer l'injection des dépendances qu'il gère. Ceci évite d'avoir à utiliser des paramètres de méthodes pour fournir les instances de ces dépendances.

Les dépendances qui seront à injecter peuvent être définies dans le fichier de configuration ou en utilisant l'annotation @Autowired.

Si les dépendances sont décrites dans le fichier de configuration, il faut fournir en attribut de l'annotation @Configurable l'id de la définition du bean à utiliser. Cette définition doit être faite dans le fichier de configuration du contexte Spring.

Exemple :

```
<bean id="monBean" lazy-init="true">
  <property name="maDependance" ref="MaDependanceImpl" />
</bean>
```

L'attribut de l'annotation @Configurable doit correspondre à la valeur de l'attribut id du tag <bean>.

L'exemple ci-dessous va définir un bean annoté avec @Configurable qui possède une dépendance vers une classe de type MonService qui sera injectée automatiquement par Spring grâce à l'annotation @Autowired.

Exemple :

```
package fr.jmdoudoux.dej.spring.beans;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Configurable;

import fr.jmdoudoux.dej.spring.service.MonService;

@Configurable(dependencyCheck = true)
public class MonBean {

  @Autowired
  private MonService monService;

  public boolean valider() {
    return monService.validerDonnees(this);
  }
}
```

Le service est décrit par une interface.

Exemple :

```
package fr.jmdoudoux.dej.spring.service;

import fr.jmdoudoux.dej.spring.beans.MonBean;

public interface MonService {
  boolean validerDonnees(MonBean bean);
}
```

Le service qui implémente l'interface définie est annoté avec @Service pour demander à Spring de gérer son cycle de vie.

Exemple :

```
package fr.jmdoudoux.dej.spring.service;

import org.springframework.stereotype.Service;
import fr.jmdoudoux.dej.spring.beans.MonBean;

@Service
public class MonServicelmpl implements MonService {

  public boolean validerDonnees(final MonBean bean) {
    return true;
  }
}
```

L'application crée une nouvelle instance de la classe MonBean et invoque sa méthode valider(). Spring va alors automatiquement injecter le service lors de l'invocation du constructeur.

Exemple :

```
package fr.jmdoudoux.dej.spring;

import org.springframework.context.support.ClassPathXmlApplicationContext;
import fr.jmdoudoux.dej.spring.beans.MonBean;

public class MonApp {

private static ClassPathXmlApplicationContext appContext;

  public static void main(final String[] args) {
    appContext = new
    ClassPathXmlApplicationContext("conf/spring/context.xml");

    try {
      final MonBean monBean = new MonBean();
      System.out.println(monBean.valider());
    }
    catch (final Exception e) {
      e. printStackTrace();
      System.exit(1);
    }

    System.exit(0);
  }
}
```

Le fichier context.xml ne contient aucune description de beans.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context-2.5.xsd
  http://www.springframework.org/schema/aop
  http://www.springframework.org/schema/aop/spring-aop-2.0.xsd"

  <context:annotation-config />
  <context:spring-configured />
  <context:component-scan base-package="fr.jmdoudoux.dej.spring"/>

</beans>
```

Pour compiler et exécuter cet exemple, il faut que les jar requis soient dans le classpath :

spring-core-3.0.1.RELEASE.jar, spring-context-3.0.1.RELEASE.jar, spring-beans-3.0.1.RELEASE.jar, commons-logging-1.1.1. jar, spring-asm-3.0.1.RELEASE.jar, spring-expression-3.0.1.RELEASE.jar, spring-aop-3. 0. 1.RELEASE.jar, spring-aspects-3.0.1.RELEASE.jar, aspectjrt-1.6.8. jar, spring-tx-3.0.1.RELEASE.jar

A l'exécution, il faut utiliser l'option -javaagent de la JVM avec comme valeur le chemin vers la bibliothèque aspectjweaver.

exemple : -javaagent:lib/aspectjweaver-1.6.1.jar

Il est important que le tissage des aspects Spring soit réalisé au runtime comme dans l'exemple ou lors de la compilation pour que l'injection des dépendances soit réalisée.

L'utilisation de ce mécanisme reposant sur l'AOP, il faut tenir compte des contraintes de cette technologie :

- avec un tissage à la compilation : le processus de build doit intégrer le tissage ce qui augmente son temps d'exécution
- avec un tissage au runtime : le temps de démarrage de l'application est augmenté

91.6.2.6. Exemple d'injection de dépendance avec @Configurable

L'exemple ci-dessous va définir une servlet qui utilise une classe ayant une dépendance vers un objet géré par le conteneur Spring.

Exemple :

```
public class MaServlet extends HttpServlet {

  private UserService userService = null;

  @Override
  protected void doGet(HttpServletRequest req,
    HttpServletResponse resp) throws ServletException, IOException {

    userService = new UserService();
    String idUser = req.getParameter("idUser");
    if (userService.isAdmin(idUser)) {
      resp.sendRedirect("admin/index.jsp");
    } else {
      resp.sendRedirect("accueil.jsp");
    }
  }
}
```

La servlet utilise une classe MonService. Cette classe possède une dépendance vers une instance de la classe UserDao. L'injection sera réalisée par Spring lors de l'invocation du constructeur grâce à l'ajout d'aspects demandé par l'annotation @Configurable.

Exemple :

```
package fr.jmdoudoux.dej.spring.services;
import fr.jmdoudoux.dej.spring.dao.UserDao;

@Configurable
public class UserServicelmpl implements UserService {
  private UserDao userDao;

  public boolean isAdmin(String idUser) {
    User user = userDao.getById(idUser);
    return user.isAdmin();
  }

  public UserDAO getUserDAO() {
    return userDAO;
  }

  public void setUserDAO(UserDAO userDAO) {
    this.userDAO = userDAO;
  }
}
```

Le fichier de configuration contient la définition des beans et la déclaration de l'utilisation des annotations.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context-2.5.xsd
  http://www.springframework.org/schema/aop
  http://www.springframework.org/schema/aop/spring-aop-2.0.xsd"

  <bean id="userDao" class="fr.jmdoudoux.dej.spring.dao.UserDaoImpl" />

  <bean class="fr.jmdoudoux.dej.spring.services.UserService">
    <property name="userDao" ref="userDao"/>
  </bean>

  <context:spring-configured/>
  <context:annotation-config/>
  <context:component-scan base-package="fr.jmdoudoux.dej.spring"/>

  <context:load-time-weaver/>

</beans>
```

Le tag <spring-configured> précise que les aspects seront tissés en fonction de la configuration de Spring.

Le tag <annotation-config> précise que Spring doit obtenir une partie de sa configuration à partir des annotations.

Le tag <load-time-weaver> permet de demander la mise en place d'un tisseur d'aspects qui va tisser les aspects au chargement des classes. L'utilisation de ce tag requière qu'un javaagent soit précisé au lancement de la JVM

Exemple .

-javaagent:/path/to/the/spring-agent. jar

Les bibliothèques de Spring 3.0.1 requises pour exécuter cet exemple sont : spring-core, spring-context, spring-beans, common-logging, spring-aop, spring-aspects, spring-expression, spring-tomcat-weaver, aspectjweaver, aspectjrt

91.6.2.7. Les annotations relatives à la gestion du cycle de vie

Spring 2.5 propose un support des annotations standard définies dans la JSR 250.

Pour activer le support de ces annotations, il faut définir un bean de type CommonAnnotationBeanPostProcessor ou à partir de Spring 2.5, il est possible d'utiliser le tag <annotation-config> de l'espace de nommage context.

91.6.2.8. L'annotation @PostConstruct

Cette annotation permet de marquer une méthode comme devant être exécutée à l'initialisation d'une nouvelle instance.

Exemple :

```
public class MonBean {
  @PostConstruct
  public void initialiser() {
    // ...
  }
  // ...
}
```

Elle est définie dans la JSR 250 et prise en charge par Spring en remplacement de l'utilisation de l'interface InitializingBean. L'utilisation de cette annotation est ainsi moins intrusive.

Elle peut aussi remplacer l'utilisation de la propriété init-method du tag <bean> dans le fichier de configuration.

91.6.2.9. L'annotation @PreDestroy

Cette annotation permet de marquer une méthode comme devant être exécutée à la destruction d'une instance.

Exemple :

```
public class MonBean {
  @PreDestroy
  public void detruire() {
    // ...
  }
  // ...
}
```

Elle est définie dans la JSR 250 et prise en charge par Spring en remplacement de l'utilisation de l'interface DisposableBean. L'utilisation de cette annotation est ainsi moins intrusive.

Elle peut aussi remplacer l'utilisation de la propriété destroy-method du tag <bean> dans le fichier de configuration.

91.6.3. Les annotations concernant les stéréotypes

Spring propose plusieurs annotations qui permettent de marquer des classes avec des stéréotypes particuliers.

91.6.3.1. Les stéréotypes Spring

Le fait de marquer des classes avec une annotation relative à un stéréotype permet au framework d'effectuer des actions de définition dans la configuration Spring de ces classes.

Les différents stéréotypes possèdent chacun une annotation :

- @Component : permet de préciser que le bean est un composant
- @Repository : permet de préciser que le bean est un repository (dao)
- @Service : permet de préciser que le bean est un service sans état
- @Controller : permet de préciser que le bean est un contrôleur Spring MVC

L'attribut par défaut name permet de préciser l'identifiant du bean.

Il est préférable d'utiliser une annotation plus spécifique telle que @Controller, @Service ou @Repository pour associer un stéréotype à une classe plutôt que d'utiliser @Component.

Tous les composants autodétectés sont implicitement nommés avec le nom de leur classe commençant par une minuscule.

Exemple :

```
package fr.jmdoudoux.dej.spring.web.controller;

@Controller
public class MonController { ... }
```

L'exemple ci-dessous est équivalent à l'exemple précédent.

Exemple :

```
<bean id="monController" class="fr.jmdoudoux.dej.spring.web.controller.MonController"/>
```

Il est possible de forcer explicitement le nom du bean en passant la valeur de ce nom en paramètre par défaut de l'annotation.

Exemple :

```
@Controller("maPage")
public class MonController { ... }
```

L'exemple ci-dessus est équivalent à

Exemple :

```
<bean id="maPage" class=" fr.jmdoudoux.dej.spring.web.controller.MonController"/>
```

Ainsi toute la partie relative à la déclaration de beans de type Service ou DAO peut être retirée du fichier de configuration et remplacée par l'utilisation des annotations. Le fichier de configuration est ainsi grandement allégé et la configuration est facilitée mais celle-ci n'est plus centralisée dans un ou plusieurs fichiers de configuration.

91.6.3.2. La recherche des composants

Spring 2.5 propose la classe ClassPathBeanDefinitionScanner qui se charge de rechercher les classes annotées avec des stéréotypes dans la liste de packages (et des sous-packages) fournie en paramètre.

L'utilisation de cette classe se fait dans le fichier de configuration en utilisant le tag <component-scan> du namespace context. Le tag <context:component-scan> du fichier de configuration demande au conteneur de rechercher des annotations particulières qui associent un stéréotype aux beans.

Exemple :

```
<context:component-scan base-package="fr.jmdoudoux.dej.spring"/>
```

Il est possible de préciser plusieurs packages comme valeur de l'attribut base-package en les séparant par une virgule.

Exemple :

```
<context:component-scan
      base-package="fr.jmdoudoux.dej.spring.services, fr.jmdoudoux.dej.spring.dao"/>
```

Par défaut, toutes les classes annotées avec @Component, @Service, @Repository et @Controller sont détectées lors de la recherche.

Il est possible d'avoir un contrôle plus fin sur les packages à scanner en utilisant un filtre grâce aux tags <include-filter> ou <exclude-filter>.

Exemple :

```
<context:component-scan base-package="fr.jmdoudoux.dej.spring">
  <context:include-filter type="annotation"
    expression="org.springframework.stereotype.Service"/>
  <context:include-filter type="assignable"
```

```
    expression="fr.jmdoudoux.dej.spring.services.BaseService"/>
  <context:include-filter type="aspectj"
    expression="fr.jmdoudoux.dej.spring..*Service"/>
  <context:include-filter type="custom"
    expression=" fr.jmdoudoux.dej.spring.MonTypeFilterImpl"/>
  <context:include-filter type="regex"
    expression=" fr.jmdoudoux.dej.spring.services.*Service"/>
</context:component-scan>
```

L'attribut type permet de préciser le format du filtre à appliquer : il peut prendre plusieurs valeurs :

Valeur de l'attribut type	Rôle
Annotation	Préciser un stéréotype
Aspectj	Utiliser une expression AspectJ
Assignable	Préciser une classe ou une interface qui peut être étendue ou implémentée
Custom	Une implémentation de l'interface org.springframework.core.type.TypeFilter
Regex	Utiliser une expression régulière

La valeur du filtre est précisée avec l'attribut expression : la valeur doit être compatible avec le format précisé par l'attribut type.

Il est aussi possible de désactiver les filtres par défaut en passant la valeur false à l'attribut use-default-filters du tag <component-scan>

Exemple :

```
<context:component-scan base-package="fr.jmdoudoux.dej.spring.web"
    use-default-filters="false">
  <context:include-filter type="assignable"
    expression="fr.jmdoudoux.dej.spring.web.BaseController"/>
</context:component-scan>
```

91.6.3.3. L'annotation @Component

Cette annotation introduite par Spring 2.5 permet de marquer une classe avec le stéréotype component.

Ce stéréotype désigne un composant générique géré par Spring qui sera autodétecté.

91.6.3.4. L'annotation @Repository

Cette annotation introduite par Spring 2.0 permet de marquer une classe avec le stéréotype repository. L'annotation @Repository est une spécialisation de l'annotation @Component.

Ce stéréotype désigne une classe qui est un DAO.

L'annotation @Repository permet au framework de traiter de façon générique les exceptions JDBC pour les transformer selon une hiérarchie de type DataAccessException. Ceci facilite la gestion des exceptions pour les services qui utilisent les DAO.

91.6.3.5. L'annotation @Service

Cette annotation introduite par Spring 2.5 permet de marquer une classe avec le stéréotype service. L'annotation @Service est une spécialisation de l'annotation @Component.

Ce stéréotype désigne une classe qui est un service métier.

91.6.3.6. L'annotation @Controller

Cette annotation introduite par Spring 2.5 permet de marquer une classe avec le stéréotype @Controller. L'annotation @Controller est une spécialisation de l'annotation @Component.

Ce stéréotype désigne un contrôleur de Spring MVC.

91.6.4. Le remplacement de la configuration par des annotations

Il faut annoter les Dao avec @Repository et les services avec @Service.

Il faut modifier le fichier de configuration pour ajouter la prise en charge des annotations et préciser les packages à scanner pour rechercher les annotations.

Il faut ajouter l'annotation @Autowired sur les attributs à injecter ou sur leurs setters publics. L'utilisation de setters facilite l'injection d'objets de type mock dans les tests unitaires.

Par défaut, l'injection automatique se fait en se basant sur le type. Pour que l'injection se base sur les noms, il faut fournir la valeur byName à l'attribut default-autowire dans la configuration.

Il faut supprimer la déclaration des beans annotés avec @Service et @Repository : attention les classes qui ne sont pas annotées avec ces deux annotations doivent toujours être définies dans la configuration (par exemple un DataSource, un TransactionManager ou une SessionFactory). Avant de les supprimer définitivement, il est préférable de réaliser les tests en mettant les définitions de ces beans en commentaires.

91.6.5. Le support de la JSR 330

Spring 3.0 offre un support de la JSR 330 (Dependency Injection for Java) qui propose l'utilisation de plusieurs annotations standard pour l'injection de dépendances. La JSR 330 (Dependency Injection for Java) définit un ensemble d'annotations qui permet de définir des dépendances, leur fournisseur et leur portée.

L'API de la JSR 330 est très simple puisqu'elle contient une interface et 5 annotations. Les spécifications de la JSR ont été gérées par Google et Spring. Il est donc normal que Spring implémente cette JSR.

Les annotations proposées par la JSR 330 peuvent être utilisées à la place de celles équivalentes proposées par Spring. L'intérêt d'utiliser les annotations JSR 330 est qu'elles rendent le code moins dépendant de la solution d'injection de dépendance utilisée (Spring, CDI, Guice, ...).

Pour utiliser les annotations de la JSR 330, des beans de type BeanPostProcessor doivent être définis dans le contexte : le plus simple est d'utiliser le tag <context :annotation-config> ou le tag <context:component-scan>

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">
  <context:component-scan base-package="fr.jmdoudoux.dej.spring" />
</beans>
```

Pour utiliser les annotations de la JSR 330 avec Spring, il faut aussi ajouter la bibliothèque de l'API dans le classpath par exemple celle fournie par Spring : com.springsource.javax.inject-1.0.0.jar.

91.6.5.1. L'annotation @Inject

L'annotation @Inject est similaire à l'annotation @Autowired.

L'annotation @Inject permet d'identifier une variable dont le type est une classe ou une interface et dont le contenu sera injecté par le conteneur.

Exemple :

```
package fr.jmdoudoux.dej.spring;

import javax.inject.Inject;

import org.springframework.stereotype.Service;

@Service("monService")
public class MonServiceImpl implements MonService {

  @Inject
  private MonDao monDao;

  @Override
  public void afficher() {
    System.out.println(monDao.getMessage());
  }
}
```

La principale différence entre @Autowired et @Inject est que cette dernière ne possède pas d'attribut required qui permet d'indiquer que l'injection est optionnelle : l'injection avec @Inject doit toujours se faire sur une instance.

91.6.5.2. L'annotation @Qualifier

L'annotation @javax.inject.Qualifier permet de définir une métadonnée sur un bean. Cette annotation permet notamment de définir d'autres annotations qui permettront de qualifier des beans.

Son but est de préciser au conteneur l'instance qui sera injectée automatiquement notamment lorsque plusieurs types peuvent être choisis par le conteneur.

Spring possède sa propre annotation @Qualifier et la JSR 330 possède une annotation @Qualifier : il ne faut donc pas confondre l'annotation @javax.inject.Qualifer et l'annotation @Qualifier de Spring. Bien que leurs noms soient identiques, leurs rôles sont différents : elles ne peuvent donc pas être interverties.

L'annotation @Qualifier de Spring est utilisée comme discriminant pour déterminer le type à utiliser lorsque plusieurs candidats sont possibles pour être injectés. Le discriminant est la valeur de l'attribut qui lui est passé en paramètre.

L'annotation @javax.inject.Qualifier peut être utilisée de différentes manières avec Spring.

La première possibilité est de définir sa propre annotation personnalisée qui sera utilisée pour qualifier une dépendance injectée automatiquement et définie dans le fichier de configuration. Cette annotation utilise une valeur fournie en paramètre pour préciser le qualificateur.

Exemple :

```
package fr.jmdoudoux.dej.spring;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import javax.inject.Qualifier;

@Target({ ElementType.FIELD, ElementType.PARAMETER, ElementType.TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface MonQualifier {
  String value();
}
```

L'annotation @MonQualifier est elle-même annotée avec @Qualifier. Elle peut être utilisée pour qualifier une dépendance.

Exemple :

```
package fr.jmdoudoux.dej.spring;

import javax.inject.Inject;

public class DepartementServiceImpl implements DepartementService {
  @Inject
  @MonQualifier("directeurGeneral")
  private Personne directeurGeneral;
  @Inject
  @MonQualifier("directeurAdjoint")
  private Personne directeurAdjoint;

  @Override
  public void afficher() {
    System.out.println("Directeur general : " + directeurGeneral.getNom());
    System.out.println("Directeur adjoint : " + directeurAdjoint.getNom());
  }

  // ...
}
```

Les identifiants des beans doivent correspondre aux valeurs fournies en paramètre de l'annotation.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">

  <context:annotation-config />

  <bean id="directeurGeneral" class="fr.jmdoudoux.dej.spring.Personne">
   <constructor-arg name="nom" value="NomDG" />
  </bean>
  <bean id="directeurAdjoint" class="fr.jmdoudoux.dej.spring.Personne">
   <constructor-arg name="nom" value="NomDA" />
  </bean>

  <bean id="departementService"
    class="fr.jmdoudoux.dej.spring.DepartementServiceImpl" />
</beans>
```

La seconde possibilité utilise aussi une annotation @Qualifier personnalisée pour qualifier la dépendance mais aussi les beans eux-mêmes. Cette solution repose entièrement sur les annotations : le fichier de configuration est alors simpliste.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">

  <context:component-scan base-package="fr.jmdoudoux.dej.spring" />

</beans>
```

L'annotation est utilisée pour qualifier un bean dans sa définition.

Exemple :

```
package fr.jmdoudoux.dej.spring;

import org.springframework.stereotype.Component;

@MonQualifier("directeurGeneral")
@Component
public class PersonneDirecteurGeneral extends Personne {

  public PersonneDirecteurGeneral() {
    super("NomDG");
  }

  public PersonneDirecteurGeneral(final String nom) {
    super(nom);
  }

  // ...
}
```

Exemple :

```
package fr.jmdoudoux.dej.spring;

import org.springframework.stereotype.Component;

@MonQualifier("directeurAdjoint")
@Component
public class PersonneDirecteurAdjoint extends Personne {

  public PersonneDirecteurAdjoint() {
    super("NomDA");
  }

  public PersonneDirecteurAdjoint(final String nom) {
    super(nom);
  }

  // ...
}
```

La troisième solution est de définir une annotation @Qualifier personnalisée pour chaque qualificateur.

Exemple :

```
package fr.jmdoudoux.dej.spring;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import javax.inject.Qualifier;

@Target({ ElementType.FIELD, ElementType.PARAMETER, ElementType.TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface DirecteurGeneral {
}
```

Exemple :

```
package fr.jmdoudoux.dej.spring;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import javax.inject.Qualifier;

@Target({ ElementType.FIELD, ElementType.PARAMETER, ElementType.TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface DirecteurAdjoint {
}
```

Ces deux annotations sont utilisées pour qualifier les beans dans leur définition.

Exemple :

```
package fr.jmdoudoux.dej.spring;

import org.springframework.stereotype.Component;

@DirecteurGeneral
@Component
public class PersonneDirecteurGeneral extends Personne {

  public PersonneDirecteurGeneral() {
    super("NomDG");
  }

  public PersonneDirecteurGeneral(final String nom) {
    super(nom);
  }

  // ...
}
```

Exemple :

```
package fr.jmdoudoux.dej.spring;

import org.springframework.stereotype.Component;

@DirecteurAdjoint
@Component
public class PersonneDirecteurAdjoint extends Personne {

  public PersonneDirecteurAdjoint() {
    super("NomDA");
  }

  public PersonneDirecteurAdjoint(final String nom) {
    super(nom);
  }

  // ...
}
```

Les annotations personnalisées sont aussi utilisées pour qualifier les dépendances.

Exemple :

```
package fr.jmdoudoux.dej.spring;

import javax.inject.Inject;

import org.springframework.stereotype.Service;

@Service("departementService")
public class DepartementServiceImpl implements DepartementService {
  @Inject
  @DirecteurGeneral
  private Personne directeurGeneral;
  @Inject
  @DirecteurAdjoint
  private Personne directeurAdjoint;

  @Override
  public void afficher() {
    System.out.println("Directeur general : " + directeurGeneral.getNom());
    System.out.println("Directeur adjoint : " + directeurAdjoint.getNom());
  }

  // ...
}
```

Cette troisième solution nécessite la création de plusieurs annotations mais elle offre un typage fort qui peut éviter des erreurs liées à l'utilisation de chaînes de caractères.

91.6.5.3. L'annotation @Named

L'annotation @javax.inject.Named est similaire à l'annotation @Qualifier de Spring.

L'annotation @javax.inject.Named est une version particulière de l'annotation @javax.inject.Qualifier : cette annotation peut être vue comme un qualificateur car @Named est annotée avec @Qualifier.

L'annotation @Named permet d'associer un nom à un bean. Si aucun nom n'est fourni en paramètre, le nom du bean sera le nom de la classe avec sa première lettre en minuscule.

Exemple :

```
package fr.jmdoudoux.dej.spring;

import javax.inject.Inject;
import javax.inject.Named;

import org.springframework.stereotype.Service;

@Service("monService")
public class MonServiceImpl implements MonService {

  @Inject
  @Named("monDaoSecondaire")
  private MonDao monDao;

  @Override
  public void afficher() {
    System.out.println(monDao.getMessage());
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.spring;
```

```
import org.springframework.stereotype.Repository;

@Repository("monDaoPrimaire")
public class MonDaoImpl implements MonDao {

  @Override
  public String getMessage() {
    return "Bonjour primaire";
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.spring;

import org.springframework.stereotype.Repository;

@Repository("monDaoSecondaire")
public class MonDaoImpl2 implements MonDao {

  @Override
  public String getMessage() {
    return "Bonjour secondaire";
  }
}
```

Résultat :

```
15 mai 2011 16:36:17 org.springframework.context.support.AbstractApplicationContext
 prepareRefresh
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@15eb0a9:
startup date [Sun May 15 16:36:17 CEST 2011]; root of context hierarchy
15 mai 2011 16:36:17 org.springframework.beans.factory.xml.XmlBeanDefinitionReader
 loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [application-context.xml]
15 mai 2011 16:36:17 org.springframework.context.annotation.
ClassPathScanningCandidateComponentProvider registerDefaultFilters
INFO: JSR-330 'javax.inject.Named' annotation found and supported for component scanning
15 mai 2011 16:36:17 org.springframework.beans.factory.annotation.
AutowiredAnnotationBeanPostProcessor <init>
INFO: JSR-330 'javax.inject.Inject' annotation found and supported for autowiring
15 mai 2011 16:36:17 org.springframework.beans.factory.support.DefaultListableBeanFactory
 preInstantiateSingletons
INFO: Pre-instantiating singletons in org.springframework.beans.factory.support.
DefaultListableBeanFactory@18f6235: defining beans
[org.springframework.context.annotation.internalConfigurationAnnotationProcessor,
org.springframework.context.annotation.internalAutowiredAnnotationProcessor,
org.springframework.context.annotation.internalRequiredAnnotationProcessor,
org.springframework.context.annotation.internalCommonAnnotationProcessor,
monDaoPrimaire,monDaoSecondaire,monService]; root of factory hierarchy
Bonjour secondaire
```

91.6.5.4. Le choix entre les annotations de Spring et celles de la JSR 330

Spring permet une utilisation de ses propres annotations et des annotations de la JSR 330 : le choix d'utiliser les unes ou les autres dépend des besoins et de l'approche souhaitée. Les annotations de la JSR 330 rendent le code moins dépendant de la solution IoC utilisée mais les annotations de Spring proposent quelques petites fonctionnalités supplémentaires.

Il existe aussi quelques subtilités mineures entre des annotations équivalentes dont il faut tenir compte notamment dans le cas d'une migration.

Enfin il est possible de mixer l'utilisation de ces annotations dans une même application même si cela peut rendre la configuration moins homogène.

91.6.5.5. Le remplacement des annotations de Spring par celles de la JSR 330

Il faut :

- remplacer les annotations @Service et @Component par @Named
- remplacer les annotations @Autowired par @Inject avec @Named au besoin

Exemple :

```
package fr.jmdoudoux.dej.spring;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Service;

@Service("monService")
public class MonServiceImpl implements MonService {

  @Autowired
  @Qualifier("monDaoSecondaire")
  private MonDao monDao;

  @Override
  public void afficher() {
    System.out.println(monDao.getMessage());
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.spring;

import javax.inject.Inject;
import javax.inject.Named;

@Named("monService")
public class MonServiceImpl implements MonService {

  @Inject
  @Named("monDaoSecondaire")
  private MonDao monDao;

  @Override
  public void afficher() {
    System.out.println(monDao.getMessage());
  }
}
```

91.6.6. La configuration grâce aux annotations

Avec Spring 3.0, il est possible de configurer le contexte en utilisant des annotations au lieu du fichier de configuration XML.

Spring 3.0 propose plusieurs annotations issues du projet JavaConfig pour définir certaines fonctionnalités de la configuration dans le code Java notamment @Configuration et @Bean.

Une classe annotée avec @Configuration permet de demander au conteneur d'utiliser cette classe pour instancier des beans. L'annotation @Bean s'utilise sur une méthode d'une classe annotée avec @Configuration qui crée une instance d'un bean.

L'utilisation de ces annotations requiert que le tag <context:component-scan> soit utilisé dans la configuration XML du contexte de Spring.

Exemple :

```
package fr.jmdoudoux.dej.spring;
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import fr.jmdoudoux.dej.spring.dao.MonDao;
import fr.jmdoudoux.dej.spring.dao.MonDaoImpl;
import fr.jmdoudoux.dej.spring.service.MonService;
import fr.jmdoudoux.dej.spring.service.MonServiceImpl;

@Configuration
public class AppConfiguration {
  @Bean
  public MonService monService() {
    return new MonServiceImpl();
  }

  @Bean
  public MonDao monDao() {
    return new MonDaoImpl();
  }
}
```

La classe AnnotationConfigApplicationContext doit être utilisée pour créer le contexte Spring à partir des classes de l'application utilisant les annotations @Configuration et @Bean.

La classe AnnotationConfigApplicationContext possède plusieurs constructeurs.

Il est possible de fournir en paramètre les instances de type Class des classes à scanner.

Exemple :

```
package fr.jmdoudoux.dej.spring;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class AnnotationConfigTest {
  public void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfiguration.class);
    MonServiceImpl monService = ctx.getBean(MonServiceImpl.class);
  }
}
```

La classe AnnotationConfigApplicationContext est aussi capable de scanner des répertoires pour trouver les classes annotées avec @Configuration en les passant en paramètres du constructeur.

Exemple :

```
package fr.jmdoudoux.dej.spring;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class AnnotationConfigTest {
  public void main(String[] args) {
    ApplicationContext ctx =
      new AnnotationConfigApplicationContext("fr.jmdoudoux.dej.spring");
    MonServiceImpl monService = ctx.getBean(MonServiceImpl.class);
  }
}
```

Les méthodes register(Class< ?>...) et scan(String...) permettent respectivement de préciser un ensemble de classes et de pckages qui seront scannées à la recherche des annotations @Configuration et @Bean pour enrichir la configuration du context.

Il est important d'invoquer la méthode refresh() de la classe AbstractApplicationContext pour prendre en compte les informations de la configuration trouvées.

91.6.6.1. L'annotation @Configuration

L'annotation @Configuration permet d'indiquer qu'une classe contient des méthodes qui seront utilisées par le conteneur Spring pour créer des instances : elle est donc à utiliser sur une classe dont le rôle est de créer des instances de beans.

Cette annotation est similaire au tag <beans> du fichier de configuration.

Exemple :

```
package fr.jmdoudoux.dej.spring;

@Configuration
public class ServiceFactory {
  @Bean
  public MonService monService() {
    return new MonServiceImpl();
  }
}
```

Cet exemple est équivalent à la définition de contexte ci-dessous

Exemple :

```
<beans>
  <bean id="monService" class="fr.jmdoudoux.dej.spring.MonServiceImpl"/>
</beans>
```

Pour utiliser l'annotation @Configuration, il faut obligatoirement que les bibliothèques asm et CGLIB soient dans le classpath.

La bibliothèque asm est utilisée pour modifier ou générer du bytecode au runtime : elle était livrée avec Spring jusqu'à la version 2.5. A partir de la version 3.0, la bibliothèque asm n'est plus fournie avec Spring : elle doit être téléchargée séparément à l'url https://asm.ow2.io/.

Si la bibliothèque asm n'est pas trouvée dans le classpath, une exception de type java.lang.ClassNotFoundException est levée pour le type org.objectweb.asm.Type

La bibliothèque cglib est aussi utilisée par Spring pour manipuler les classes au runtime. Comme asm, elle n'est aussi pas fournie dans la version 3.0 de Spring. Elle est téléchargeable à l'url http://cglib.sourceforge.net/.

Si la bibliothèque cglib n'est pas trouvée, alors une exception de type java.lang.IllegalStateException est levée avec le message « CGLIB is required to process @Configuration classes ».

Les classes annotées avec @Configuration doivent respecter certaines contraintes :

- ne doivent pas être final
- ne peuvent pas être déclarées dans une méthode d'une autre classe
- doivent avoir un constructeur par défaut sans paramètre
- ne peuvent pas utiliser l'annotation @Autowired sur les paramètres de leurs constructeurs

L'annotation @Configuration est elle-même annotée avec @Component : les classes annotées avec @Configuration sont scannées et peuvent donc utiliser l'annotation @Autowired sur des champs ou des méthodes.

91.6.6.2. L'annotation @Bean

Cette annotation est à utiliser sur une méthode qui se charge de créer une instance d'un bean.

L'annotation @Bean indique au conteneur que la méthode pour être utilisée pour créer une instance d'un bean. L'identifiant est fourni en tant qu'attribut name de l'annotation ou, à défaut, correspond au nom de la méthode avec la première lettre en minuscule. L'attribut name est un tableau de chaînes de caractères ce qui permet de préciser des alias.

Cette annotation est similaire au tag <bean> du fichier de configuration.

Elle possède plusieurs attributs :

Attribut	Rôle
String init-method	Nom d'une méthode du bean qui sera invoquée lors de l'initialisation du bean (optionnel)
String destroy-method	Nom d'une méthode du bean qui sera invoquée lorsque le bean sera retiré du conteneur (optionnel)
Autowire autowire	Permet de préciser si les dépendances doivent être injectées automatiquement et si oui dans quel mode. Les valeurs possibles sont Autowire.BY_NAME, Autowire.BY_TYPE et Autowire.NO (valeur par défaut Autowire.NO)
String[] name	Identifiant et alias du bean

L'annotation @Bean ne possède pas d'attribut qui permette de préciser les propriétés scope, lazy et primary. Pour cela, il faut utiliser les annotations @Scope, @Lazy et @Primary.

Par défaut, c'est le nom de la méthode annotée avec @Bean qui sera utilisé pour le nom du bean.

Exemple XML

Exemple :

```
<beans>
  <bean name="monBean" class="fr.jmdoudoux.dej.spring.MonBean">
</beans>
```

Exemple avec annotations

Exemple :

```
@Configuration
public class AppConfig {
  @Bean
  public MonBean monBean() { return new MonBean();}
}
```

L'annotation @Scope peut être utilisée pour préciser la portée des instances créées. L'annotation @Scope s'utilise dans ce cas sur des méthodes annotées avec @Bean.

L'annotation @Bean peut aussi être utilisée sur un bean annoté avec @Component. Dans ce cas, l'annotation @Bean est équivalente à l'élément factory-method du fichier de configuration XML.

91.6.6.3. L'annotation @DependsOn

Cette annotation permet de préciser des beans dont dépend le bean annoté avec @DependsOn : le conteneur garantit que les beans précisés en paramètre de cette annotation seront créés avant le bean annoté.

Le rôle de cette annotation est similaire à l'attribut depends-on du tag <bean> dans le fichier de configuration.

L'annotation @DependsOn peut être utilisée sur une classe annotée avec @Component ou sur une méthode annotée avec @Bean.

91.6.6.4. L'annotation @Primary

L'annotation @Primary permet de préciser que cette classe doit être privilégiée lors d'une injection de type autowire et que plusieurs classes peuvent être injectées. Si une seule de ces classes est annotée avec @Primary, c'est une de ces instances qui sera injectée.

Le rôle de cette annotation est similaire à l'attribut primary du tag <bean> dans le fichier de configuration.

L'annotation @Primary peut être utilisée sur une classe annotée avec @Component ou sur une méthode annotée avec @Bean.

91.6.6.5. L'annotation @Lazy

Cette annotation est à utiliser sur une classe et permet de demander l'initialisation tardive d'un bean de type singleton. Si cette annotation est utilisée avec la valeur true, l'instance du bean ne sera initialisée que lorsqu'un autre bean en aura besoin.

L'annotation @Lazy peut être utilisée sur une classe annotée avec @Component ou @Configuration ou sur une méthode annotée avec @Bean. Utilisée sur une classe avec @Configuration, elle permet de demander l'initialisation de tous les beans créés par les méthodes annotées avec @Bean.

91.6.6.6. L'annotation @Import

L'annotation @Import permet de demander l'importation de classes annotées avec @Configuration : les beans créés par les méthodes annotées @Bean des classes fournies en paramètre pourront être injectées au moyen de l'annotation @Autowired.

L'annotation @Import est équivalente au tag <import> du fichier de configuration XML.

91.6.6.7. L'annotation @ImportResource

L'annotation @ImportResource permet de demander l'importation de ressources qui contiennent la définition de beans.

L'annotation @ImportResource est équivalente au tag <import> du fichier de configuration XML.

91.6.6.8. L'annotation @Value

L'annotation @Value permet de préciser une valeur par défaut qui sera le résultat de l'évaluation de l'expression Spring EL fournie en paramètre.

L'annotation @Value peut être utilisée sur un champ ou un paramètre d'une méthode ou d'un constructeur.

91.7. Le scheduling

Spring 3.0 propose une mise en oeuvre simplifiée du scheduling notamment en proposant un espace de nommage dédié et des annotations.

91.7.1. La définition dans le fichier de configuration du context

Le namespace task peut être utilisé pour planifier l'exécution de méthodes de beans. L'espace de nommage doit être déclaré dans la configuration du contexte.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:task="http://www.springframework.org/schema/task"
  xsi:schemaLocation="http://www.springframework.org/schema/task
  http://www.springframework.org/schema/task/spring-task-3.0.xsd
  http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
</beans>
```

L'espace de nommage comprend le tag <scheduler> qui définit un objet de type ThreadPoolTaskExecutor. Le tag <scheduler> possède plusieurs attributs :

Attributs	Rôle
id	Identifiant du pool : sera utilisé comme préfixe pour le nom des threads du pool
pool-size	Permet de préciser le nombre de threads dans le pool. Par défaut, il n'y a qu'un seul thread

Exemple :

```
<task:scheduler id="monScheduler" pool-size="10" />
```

Le tag <scheduled-taks> permet d'associer des tâches au scheduler. Une tâche est définie par une méthode d'un bean géré par Spring dont l'exécution est planifiée.

Chaque tâche est définie avec un tag <scheduled> qui possède plusieurs attributs :

Attribut	Rôle
ref	Préciser l'identifiant du bean qui sera utilisé par la tâche
method	Préciser le nom de la méthode qui sera exécutée par la tâche
cron	Planifier l'exécution grâce à une expression de type cron
fixed-delay	Planifier l'exécution grâce à une durée exprimée en millisecondes qui précise le temps d'attente entre deux exécutions. Ce temps démarre à la fin de l'exécution courante : une même tâche ne peut pas être exécutée plusieurs fois à un même instant.
fixed-rate	Planifier l'exécution grâce à une durée exprimée en millisecondes qui précise le temps d'attente entre deux exécutions. Si le temps d'exécution de la tâche est plus long que le temps précisé en paramètre alors la tâche aura plusieurs exécutions en cours

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:p="http://www.springframework.org/schema/p"
  xmlns:task="http://www.springframework.org/schema/task"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd
    http://www.springframework.org/schema/task
    http://www.springframework.org/schema/task/spring-task-3.0.xsd">
  <task:scheduled-tasks scheduler="monScheduler">
    <task:scheduled ref="maTache" method="executer"
      fixed-delay="5000" />
  </task:scheduled-tasks>

  <task:scheduler id="monScheduler" pool-size="10" />

  <bean id="maTache" class="fr.jmdoudoux.dej.spring.task.MaTache"/>

</beans>
```

Le bean qui encapsule les traitements à exécuter doit respecter deux contraintes : la méthode exécutée ne doit rien retourner et ne doit pas avoir de paramètre.

Exemple :

```
package fr.jmdoudoux.dej.spring.task;

import java.util.Date;

public class MaTache {

  public void executer() {
    System.out.println("Excution de la tache MaTache " + new Date());
  }
}
```

91.7.2. La définition grâce aux annotations

Spring 3.0 propose de mettre en oeuvre le scheduling grâce à des annotations.

L'annotation @Scheduler permet de planifier l'exécution de la méthode annotée.

L'annotation @Async permet l'exécution de la méthode annotée de manière asynchrone.

L'utilisation de ces annotations n'est possible que si le tag <task:annotation-driven> est utilisé dans le fichier de configuration du contexte.

Il possède plusieurs attributs :

Attributs	Rôle
executor	Permet de fournir une instance de TaskExecutor qui sera utilisée lors de l'exécution des méthodes de manière asynchrone
Scheduler	Permet de fournir une instance de TaskScheduler qui sera utilisée pour planifier les tâches définies avec @Scheduler

Exemple :

```
<task:annotation-driven executor="monExecutor" scheduler="monScheduler" />
<task:executor id="monExecutor" pool-size="10" />
<task:scheduler id="monScheduler" pool-size="10" />
```

Le tag <executor> permet de définir un pool de TaskExecutor pour prendre en charge l'invocation asynchrone des méthodes annotées avec @Async.

Le tag <scheduler> permet de définir un pool de ThreadPoolTaskExecutor pour prendre en charge l'exécution des tâches définies avec l'annotation @Scheduled.

91.7.2.1. La définition grâce à l'annotation @Scheduled

La configuration peut se faire avec l'annotation @Scheduled sur une méthode qui ne doit rien retourner et ne posséder aucun paramètre.

L'attribut fixedDelay permet de définir la configuration avec une période fixe : il permet de définir le temps d'attente entre la fin de l'exécution et le lancement de la suivante.

Exemple :

```
package fr.jmdoudoux.dej.spring.task;

import java.util.Date;

import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Service;

@Service
public class MaTache {

  public MaTache() {
    System.out.println("instanciation de la classe MaTache");
  }

  @Scheduled(fixedDelay = 5000)
  public void executer() {
    System.out.println("Exécution de la tâche MaTache " + new Date());
  }
}
```

L'attribut fixedRate permet de définir la configuration avec un délai fixe : il permet de définir le temps d'attente entre deux exécutions : ce temps démarre dès l'exécution courante.

Exemple :

```
  @Scheduled(fixedRate = 60000)
  public void executer() {
    System.out.println("Exécution de la tâche MaTache " + new Date());
  }
```

L'attribut cron permet de définir la configuration avec une syntaxe proche de celle de la commande Unix cron

Exemple :

```
  @Scheduled(cron="0/10 * * * * ?")
  public void executer() {
    System.out.println("Exécution de la tâche MaTache " + new Date());
  }
```

Il faut s'assurer qu'il n'y aura qu'une seule instance de la classe dont les méthodes sont annotées avec @Scheduled.

Il est nécessaire que les bibliothèques requises pour la mise en oeuvre de l'AOP soient présentes dans le classpath : org.springframework.aop.-3.0.5.jar et com.springsource.org.aopalliance-1.0.0.jar.

Le fichier de configuration du contexte est très simple :

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:p="http://www.springframework.org/schema/p"
  xmlns:task="http://www.springframework.org/schema/task"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd
    http://www.springframework.org/schema/task
    http://www.springframework.org/schema/task/spring-task-3.0.xsd">

  <context:component-scan base-package="fr.jmdoudoux.dej.spring.task" />
  <task:annotation-driven />

</beans>
```

91.7.3. L'invocation de méthodes de manière asynchrone

L'annotation @Async permet de définir l'invocation d'une méthode de manière asynchrone : l'appel est délégué à un TaskExecutor.

La méthode annotée avec @Async peut avoir des paramètres et par défaut ne renvoie rien.

Exemple :

```
package fr.jmdoudoux.dej.spring.task;

import org.springframework.scheduling.annotation.Async;
import org.springframework.stereotype.Component;

@Component
public class MonTraitement {

  @Async()
  public void executer(final int numero) {
    String nomThread = Thread.currentThread().getName();
    System.out.println("   " + nomThread + " début du traitement " + numero);
    try {
      Thread.sleep(5000);
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
    }
    System.out.println("   " + nomThread + " fin du traitement " + numero);
  }
}
```

Une tâche avec une exécution périodique est planifiée pour lancer les traitements asynchrones.

Exemple :

```
package fr.jmdoudoux.dej.spring.task;

import java.util.Date;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Service;

@Service
public class MaTache {

  @Autowired
  private MonTraitement monTraitement;

  public MaTache() {
    System.out.println("instanciation de la classe MaTache");
  }

  @Scheduled(fixedDelay = 15000)
  public void executer() {
    System.out.println("Début exécution de la tâche MaTache " + new Date());
    for (int i = 1; i < 6; i++) {
      monTraitement.executer(i);
    }
    System.out.println("Fin exécution de la tâche MaTache " + new Date());
  }
}
```

Lors de l'exécution de la tâche, 5 threads sont lancés pour exécuter les traitements.

Résultat :

```
instanciation de la classe MaTache
Déebut exécution de la tâche MaTache Sun Jun 05 18:31:20 CEST 2011
Fin exécution de la tâche MaTache Sun Jun 05 18:31:20 CEST 2011
   SimpleAsyncTaskExecutor-2 début du traitement 2
   SimpleAsyncTaskExecutor-4 début du traitement 4
   SimpleAsyncTaskExecutor-1 début du traitement 1
   SimpleAsyncTaskExecutor-3 début du traitement 3
   SimpleAsyncTaskExecutor-5 début du traitement 5
   SimpleAsyncTaskExecutor-1 fin du traitement 1
   SimpleAsyncTaskExecutor-2 fin du traitement 2
   SimpleAsyncTaskExecutor-4 fin du traitement 4
   SimpleAsyncTaskExecutor-3 fin du traitement 3
   SimpleAsyncTaskExecutor-5 fin du traitement 5
Début exécution de la tâche MaTache Sun Jun 05 18:31:35 CEST 2011
Fin exécution de la tâche MaTache Sun Jun 05 18:31:35 CEST 2011
   SimpleAsyncTaskExecutor-6 début du traitement 1
   SimpleAsyncTaskExecutor-8 début du traitement 3
   SimpleAsyncTaskExecutor-10 début du traitement 5
   SimpleAsyncTaskExecutor-7 début du traitement 2
   SimpleAsyncTaskExecutor-9 début du traitement 4
```

Par défaut le nom des threads commence par « SimpleAsyncTaskExecutor ». Il est possible de modifier ce préfixe en précisant un identifiant au TaskExecutor.

Exemple :

```
  <task:annotation-driven executor="monExecutor" scheduler="monScheduler" />
  <task:executor id="monExecutor" pool-size="10" />
  <task:scheduler id="monScheduler" pool-size="10" />
```

Résultat :

```
instanciation de
la classe MaTache
Début exécution de la tâche MaTache Sun Jun 05 18:27:56 CEST 2011
Fin exécution de la tâche MaTache Sun Jun 05 18:27:56 CEST 2011
   monExecutor-2 début du traitement 2
   monExecutor-4 début du traitement 4
   monExecutor-1 début du traitement 1
   monExecutor-3 début du traitement 3
   monExecutor-5 début du traitement 5
   monExecutor-2 fin du traitement 2
   monExecutor-4 fin du traitement 4
   monExecutor-1 fin du traitement 1
   monExecutor-3 fin du traitement 3
   monExecutor-5 fin du traitement 5
Debut exécution de la tâche MaTache Sun Jun 05 18:28:11 CEST 2011
   monExecutor-6 debut du traitement 1
Fin exécution de la tâche MaTache Sun Jun 05 18:28:11 CEST 2011
   monExecutor-8 début du traitement 3
   monExecutor-10 début du traitement 5
   monExecutor-7 début du traitement 2
   monExecutor-9 début du traitement 4
```

La méthode invoquée de manière asynchrone peut avoir une valeur de retour qui doit être du type Future<T>.

Exemple :

```
package fr.jmdoudoux.dej.spring.task;

import java.util.Date;
import java.util.concurrent.Future;

import org.springframework.scheduling.annotation.Async;
import org.springframework.scheduling.annotation.AsyncResult;
import org.springframework.stereotype.Component;

@Component
public class MonTraitement {

  @Async
  public Future<Personne> rechercher(final long id) {
    System.out.println(new Date()
        + " MonTraitement debut invocation rechercher");
    Personne personne = new Personne("Nom" + id);
    try {
      Thread.sleep(10000);
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
    }
    System.out.println(new Date() + " MonTraitement fin invocation rechercher");
    return new AsyncResult<Personne>(personne);
  }
}
```

La valeur de retour doit être fournie en paramètre du constructeur d'une instance de type AsyncResult().

La méthode get() de l'interface Future permet d'obtenir la valeur de retour avec une attente éventuelle si le traitement asynchrone n'est pas encore terminé.

Exemple :

```
package fr.jmdoudoux.dej.spring.task;

import java.util.Date;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class MaTache {

  @Autowired
  private MonTraitement monTraitement;

  public MaTache() {
    System.out.println("instanciation de la classe MaTache");
  }

  public void rechercher() {
    System.out.println(new Date()
        + " MaTache Debut invocation methode rechercher()");
    Future<Personne> resultat = monTraitement.rechercher(1234L);

    try {
      System.out.println(new Date() + " MaTache debut traitement");
      Thread.sleep(5000);
      System.out.println(new Date() + " MaTache fin traitement");
    } catch (InterruptedException e) {
    }

    System.out.println(new Date() + " MaTache Obtention du resultat");
    try {
      System.out.println(new Date() + " MaTache Personne = " + resultat.get());
    } catch (InterruptedException e) {
    } catch (ExecutionException e) {
      e.printStackTrace();
    }
    System.out.println(new Date()
        + " MaTache Fin invocation methode rechercher()");
  }
}
```

Si une exception est levée durant les traitements, elle est encapsulée dans une exception de type ExecutionException qui sera levée par la méthode get().

Pour pouvoir utiliser l'annotation @Async, la bibliothèque CGLib doit être présente dans le classpath : pour cela il faut par exemple ajouter la bibliothèque com.springsource.net.sf.cglib-2.2.0.jar.

92. La mise en oeuvre de l'AOP avec Spring

Chapitre 92

Niveau : Confirmé

L'AOP permet de facilement mettre en place des fonctionnalités dans différents points d'une application. Ces fonctionnalités sont désignées sous le terme advice : elles sont exécutées lors d'événements nommés joinpoint (par exemple l'invocation d'une méthode ou d'un constructeur, ...).

Les endroits où les advices seront invoqués lorsque le joinpoint est réalisé sont définis grâce à des pointcuts.

Une opération de tissage est nécessaire pour permettre l'exécution des aspects au runtime : ce tissage peut être réalisé dynamiquement (grâce à un classloader ou la création de proxys) ou par compilation.

L'AOP est particulièrement intéressante pour mettre en oeuvre certaines fonctionnalités techniques transverses comme les transactions. C'est d'ailleurs grâce à l'AOP que les transactions sont gérées par Spring. La gestion des transactions devient alors déclarative et ne requiert plus de code supplémentaire utilisant une API dédiée.

L'AOP est un des mécanismes importants utilisés par Spring : il l'utilise lui-même pour mettre en oeuvre certaines fonctionnalités notamment les transactions, l'annotation @Configurable, ROO, ...

Ainsi, l'AOP peut être utilisée :

- Indirectement, lors de l'utilisation de ces fonctionnalités
- Directement, pour mettre en oeuvre ses propres Aspects : Spring facilite alors cette mise en oeuvre

Spring met en oeuvre l'AOP de deux façons :

- Spring AOP : solution de Spring reposant sur des proxys créés dynamiquement
- AspectJ : solution open source du projet Eclipse qui permet un tissage des aspects au runtime ou à la compilation par enrichissement du bytecode. Depuis la version 2.0, Spring propose un support AspectJ pour la mise en oeuvre de l'AOP en complément de sa propre solution reposant sur les proxys.

L'AOP peut être mise en oeuvre via Spring AOP ou AspectJ de plusieurs manières :

- Avec AspectJ avec un tissage au chargement (Load Time Weaving) ou à la compilation
- Avec ou sans les annotations AspectJ avec Spring AOP
- Avec un mixte de Spring AOP et AspectJ

La mise en oeuvre peut donc se faire par déclaration dans le fichier de configuration ou par des annotations selon la solution de tissage utilisée. Toutes les combinaisons de syntaxe de déclaration avec la méthode de tissage ne sont pas possibles :

	Syntaxe AspectJ	Annotation style AspectJ	XML dans la définition du context
Tissage par Spring	Non	Oui	Oui
Tissage par AspectJ	Oui	Oui	Non

Spring AOP ne permet qu'un tissage au runtime qui va créer des proxys dynamiquement lors du chargement du contexte, selon la configuration indiquée.

Spring AOP ne propose pas un support des fonctionnalités de programmation orientée aspect aussi poussé que celui proposé par AspectJ. La mise en oeuvre de Spring AOP ne peut se faire que sous certaines conditions :

- Seuls les points de jonction liés à l'exécution d'une méthode sont supportés
- Les aspects Spring AOP sont définis dans la configuration du contexte : ils ne peuvent donc s'appliquer que sur des objets gérés par le conteneur Spring car ils reposent sur des proxys exécutés dynamiquement.
- Les aspects ne peuvent être appliqués que sur des méthodes public et non static

Ce chapitre contient plusieurs sections :

- ♦ Spring AOP
- ♦ AspectJ
- ♦ Spring AOP et AspectJ
- ♦ L'utilisation des namespaces

92.1. Spring AOP

Spring AOP est un module du framework Spring qui permet une mise en oeuvre d'une partie des fonctionnalités de l'AOP. Il propose un tisseur d'aspects sous la forme de proxys qui sont créés dynamiquement au runtime.

Depuis la version 2.0, la définition d'un aspect avec Spring AOP peut se faire grâce à une déclaration dans le fichier de configuration du contexte ou grâce aux annotations d'AspectJ.

Durant l'injection des dépendances, le conteneur Spring va créer un proxy dynamique pour l'interface concernée et c'est ce proxy qui sera injecté. Ce proxy est en charge d'exécuter le code des greffons lors de l'invocation des méthodes concernées de l'interface. Un des avantages des aspects est qu'ils sont facilement activables/désactivables : les fonctionnalités qu'ils contiennent peuvent alors être activées ou non sans modifier les classes greffées.

Spring 2.0 facilite la configuration d'AOP en proposant un schéma et un espace de nommage associé dédiés.

Spring AOP utilise des proxys ce qui ne nécessite pas d'outils particulier comme c'est le cas avec AspectJ pour réaliser le tissage (classloader ou compilateur dédié). Spring permet une exécution des advices sur une instance précise alors qu'avec AspectJ l'advice sera exécuté pour toutes les instances puisque la définition est faite sur le type.

92.1.1. Les différents types d'advices

Le but de Spring AOP n'est pas de proposer un support complet des fonctionnalités de l'AOP mais de proposer la possibilité de mettre en oeuvre des fonctionnalités transverses qui s'intègrent dans le conteneur Spring. Ainsi, Spring AOP ne propose qu'un support des points de jonction de type exécution de méthodes. Pour la mise en oeuvre d'autres types de points de jonction, il faut utiliser une solution qui propose leur support comme AspectJ.

Spring AOP propose 5 types d'advices :

- before : le code de l'advice est exécuté avant l'exécution de la méthode. Il n'est pas possible d'inhiber l'invocation de la méthode sauf si une exception est levée dans l'advice
- after returning : le code de l'aspect est exécuté après l'exécution de la méthode qui renvoie une valeur de retour (aucune exception n'est levée)
- after throwing : le code de l'aspect est exécuté lorsqu'une exception est levée suite à l'invocation de la méthode
- after : le code de l'aspect est exécuté après l'exécution de la méthode, même si une exception est levée.
- around : le code de l'aspect permet de lancer l'exécution de la méthode et ainsi de réaliser des traitements avant, pour par exemple conditionner l'invocation de la méthode et des traitements après

Il est recommandé d'utiliser l'advice le plus adapté au besoin plutôt que de tout faire avec un advice de type around : cette bonne pratique permet de simplifier le code et d'éviter des erreurs potentielles.

Les paramètres des advices sont fortement typés.

92.1.2. Spring AOP sans les annotations AspectJ

Sans utiliser les annotations AspectJ, il est possible de mettre en oeuvre Spring AOP en utilisant le fichier de configuration du contexte pour déclarer et configurer les aspects.

L'exemple de cette section va développer un service sur lequel l'invocation des méthodes va être tracée grâce à un aspect. Cet aspect trace les invocations des méthodes des services en affichant les paramètres d'invocation et la valeur de retour.

Les fonctionnalités du service sont définies par une interface.

Exemple :

```
package fr.jmdoudoux.dej.spring.service;

import fr.jmdoudoux.dej.spring.entite.Personne;

public interface PersonneService {
  void afficher();

  void ajouter(Personne personne);
}
```

L'implémentation du service est volontairement basique.

Exemple :

```
package fr.jmdoudoux.dej.spring.service;

import fr.jmdoudoux.dej.spring.entite.Personne;

public class PersonneServiceImpl implements PersonneService {

  @Override
  public void afficher() {
    try {
      Thread.sleep(250);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }

  @Override
  public void ajouter(final Personne personne) {
    try {
      Thread.sleep(500);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

92.1.2.1. La définition de l'aspect

Les traitements de l'aspect vont simplement tracer l'invocation d'une méthode avec les paramètres utilisés, invoquer la méthode et tracer la fin de l'invocation.

Exemple :

```
package fr.jmdoudoux.dej.spring.aspect;

import org.apache.log4j.Logger;
import org.aspectj.lang.ProceedingJoinPoint;
```

```
public class TraceInvocation {

  private static Logger LOGGER = Logger.getLogger(TraceInvocation.class);
  private int order;

  public Object afficherTrace(final ProceedingJoinPoint joinpoint)
      throws Throwable {
    String nomMethode = joinpoint.getTarget().getClass().getSimpleName() + "."
        + joinpoint.getSignature().getName();

    final Object[] args = joinpoint.getArgs();
    final StringBuffer sb = new StringBuffer();
    sb.append(joinpoint.getSignature().toString());
    sb.append(" avec les parametres : (");

    for (int i = 0; i < args.length; i++) {
      sb.append(args[i]);
      if (i < args.length - 1) {
        sb.append(", ");
      }
    }
    sb.append(")");
    LOGGER.info("Debut methode : " + sb);
    try {
      Object obj = joinpoint.proceed();
    } finally {
      LOGGER.info("Fin methode : " + nomMethode + " retour=" + obj);
    }
    return obj;
  }
}
```

Le code de l'aspect à exécuter doit être contenu dans une méthode qui attend en paramètre un objet de type ProceedingJoinPoint. La classe contenant la méthode doit être instanciable par le contexte.

La classe ProceedingJoinPoint d'AspectJ est utilisée pour obtenir des informations sur le point de jonction et invoquer les traitements qui lui sont associés en utilisant la méthode proceed().

La déclaration et la configuration des aspects se font dans le fichier de configuration.

L'espace de nommage aop permet la déclaration de la configuration de l'AOP notamment en proposant les tags pour configurer les aspects, les points de coupe et les advices.

L'aspect fait référence à un bean géré par le conteneur.

La définition du point de coupe utilise la syntaxe d'AspectJ.

L'advice est une association entre le point de coupe et la méthode de l'aspect à exécuter. Cinq types d'advices sont utilisables : before, after returning, after throwing, after et around.

92.1.2.2. La déclaration de l'aspect

Spring 2.0 permet la déclaration des aspects dans la configuration de son contexte qui utilise la syntaxe d'AspectJ pour les définitions des pointcuts. Dans ce cas, l'aspect n'a pas besoin d'être annoté : c'est un simple bean qui doit être déclaré dans la configuration.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop"
xmlns:tx="http://www.springframework.org/schema/tx"
```

```
xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">
  <context:component-scan base-package="fr.jmdoudoux.dej.spring" />
  <aop:config>
    <aop:aspect id="traceInvocationAspect" ref="tracerInvocation">
      <aop:pointcut id="traceInvocationPointcut"
        expression="execution(* fr.jmdoudoux.dej.spring.service.*ServiceImpl.*(..))" />
      <aop:around pointcut-ref="traceInvocationPointcut" method="afficherTrace" />
    </aop:aspect>
  </aop:config>
  <bean id="tracerInvocation" class="fr.jmdoudoux.dej.spring.aspect.TraceInvocation"/>
  <bean id="personneService" class="fr.jmdoudoux.dej.spring.service.PersonneServiceImpl" />
</beans>
```

Il faut déclarer dans la configuration le bean qui contient le code de l'aspect à exécuter.

La configuration de l'AOP se fait avec un tag <aop:config>.

Chaque aspect est défini grâce à un tag <aop:aspect> : l'attribut ref permet de préciser l'identifiant du bean qui contient les traitements de l'aspect.

Le point de coupe est défini en utilisant un tag <aop:pointcut>. Son attribut expression permet de définir les méthodes concernées en utilisant une expression régulière.

La définition des points de jonction se fait en utilisant des expressions régulières pour définir les méthodes concernées. Plusieurs caractères particuliers peuvent être utilisés pour définir un filtre sur les classes et la signature des méthodes :

- Le caractère « * » indique n'importe quel caractère sauf le caractère * lui-même ou n'importe quel élément (modificateur, type de retour, classe, méthode, paramètre)
- Les caractères « .. » indiquent n'importe quelle signature ou sous-package
- Le caractère « + » indique n'importe quel sous-type

Exemple :

public * *(..) : toutes les méthodes public

* get*(..) : toutes les méthodes commençant par get

* fr.jmdoudoux.dej.spring.service.IMonService.*(..)) : toutes les méthodes de l'interface IMonService

* fr.jmdoudoux.dej.spring.service.*.*(..)) : toutes les méthodes du package fr.jmdoudoux.dej.spring.service

* fr.jmdoudoux.dej.spring.service..*.*(..)) : toutes les méthodes du package fr.jmdoudoux.dej.spring.service et de ses sous-packages

92.1.2.3. Le namespace aop

Pour pouvoir être utilisé, le namespace aop doit être déclaré dans le tag racine du fichier de définition du contexte.

Exemple :

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd">
```

```
</beans>
```

Le schéma AOP propose plusieurs tags pour permettre la définition des aspects dans le fichier de configuration du contexte :

- <aop:config> : configure Spring AOP
- <aop:advisor> : définit un advisor
- <aop:pointcut> : définit un point de coupe avec une expression régulière
- <aop:aspectj-autoproxy> : permet d'activer le support des annotations AspectJ pour la création des proxys
- <aop:scoped-proxy> : crée un proxy pour un bean
- <aop:spring-configured> : permet d'activer le support de l'annotation @Configurable

Le tag <aop:config> permet dans le fichier de définition du contexte de configurer Spring AOP. Il peut notamment contenir la définition des points de coupe, des advisors et des aspects. Il est possible d'utiliser plusieurs tags <aop:config> dans un même fichier de configuration. L'ordre de déclaration des points de coupe, des advisors et des aspects doit être respecté à l'intérieur d'un tag <aop:config>.

Il est possible de définir un ou plusieurs points de coupe, chacun étant identifié par un nom unique en utilisant le tag <aop:pointcut>. Le nom est fourni en utilisant l'attribut id. L'attribut expression permet de définir une expression régulière qui va définir le point de coupe. La syntaxe de cette expression est identique à celle utilisée avec les annotations AspectJ. Le tag <aop:pointcut> peut être utilisé comme tags fils du tag <aop:config> ou <aop:aspect>.

La définition d'un advice se fait en utilisant un tag dédié pour chaque advice supporté par Spring AOP (before, after returning, after throwing, after, et around). Ces tags sont à utiliser en tant que tag fils du tag <aop:aspect>.

Le tag <aop:before> permet de définir un advice de type before : cet advice permet d'exécuter une méthode de la classe qui encapsule les traitements de l'aspect juste avant l'exécution des méthodes qui correspondent au point de coupe.

Exemple :

```
  <aop:config>
    <aop:aspect id="traceInvocationAspect" ref="tracerInvocation">
      <aop:pointcut id="traceInvocationPointcut"
        expression="execution(* fr.jmdoudoux.dej.spring.service.*ServiceImpl.*(..))" />
      <aop:before pointcut-ref="traceInvocationPointcut"
        method="afficherDebutTrace" />
    </aop:aspect>
  </aop:config>

  <bean id="tracerInvocation" class="fr.jmdoudoux.dej.spring.aspect.TraceInvocation" />
```

L'attribut pointcut permet de fournir une expression régulière qui précise le point de coupe.

L'attribut pointcut-ref permet de fournir l'identifiant du point de coupe préalablement défini.

L'attribut method permet de préciser le nom de la méthode de la classe encapsulant les traitements de l'aspect qui sera exécutée. Un paramètre de type org.aspectj.lang.JoinPoint dans la signature de la méthode permet d'obtenir des informations sur le point de jonction.

Exemple :

```
package fr.jmdoudoux.dej.spring.aspect;

import org.apache.log4j.Logger;
import org.aspectj.lang.JoinPoint;

public class TraceInvocation {

  private static Logger LOGGER = Logger.getLogger(TraceInvocation.class);

  public void afficherDebutTrace(final JoinPoint joinpoint) throws Throwable {
    final Object[] args = joinpoint.getArgs();
    final StringBuffer sb = new StringBuffer();
    sb.append(joinpoint.getSignature().toString());
    sb.append(" avec les parametres : (");

    for (int i = 0; i < args.length; i++) {
      sb.append(args[i]);
      if (i < args.length - 1) {
        sb.append(", ");
      }
    }
    sb.append(")");

    LOGGER.info("Debut methode : " + sb);
  }
}
```

Le tag <aop:after-returning> définit un advice de type after returning : cet advice permet d'exécuter une méthode de la classe encapsulant les traitements de l'aspect après l'exécution sans qu'une exception soit levée des méthodes qui correspondent au point de coupe.

Exemple :

```
  <aop:config>
    <aop:aspect id="traceInvocationAspect" ref="tracerInvocation">
      <aop:pointcut id="traceInvocationPointcut"
        expression="execution(* fr.jmdoudoux.dej.spring.service.*ServiceImpl.*(..))" />
      <aop:after-returning pointcut-ref="traceInvocationPointcut"
        method="afficherFinNormaleTrace" returning="result" />
    </aop:aspect>
  </aop:config>

  <bean id="tracerInvocation" class="fr.jmdoudoux.dej.spring.aspect.TraceInvocation" />
```

L'attribut pointcut-ref permet de fournir l'identifiant du point de coupe préalablement défini.

L'attribut method permet de préciser le nom de la méthode de la classe encapsulant les traitements de l'aspect qui sera exécutée.

L'attribut returning permet de préciser le nom du paramètre de la méthode qui va contenir la valeur de retour de l'exécution de la méthode. Dans ce cas, la méthode de l'aspect doit avoir un paramètre dont le type est identique à celui des valeurs de retour des méthodes du point de coupe. Le nom de ce paramètre doit correspondre à celui fourni dans l'attribut returning. La méthode de l'aspect peut aussi avoir un paramètre optionnel de type org.aspectj.lang.JoinPoint.StaticPart qui permet d'obtenir des informations sur le point de jonction.

Exemple :

```
package fr.jmdoudoux.dej.spring.aspect;

import org.apache.log4j.Logger;
import org.aspectj.lang.JoinPoint.StaticPart;

public class TraceInvocation {

  private static Logger LOGGER = Logger.getLogger(TraceInvocation.class);

  public void afficherFinNormaleTrace(final StaticPart staticPart, final Object result)
      throws Throwable {
    String nomMethode = staticPart.getSignature().toLongString();
    LOGGER.info("Fin methode :  " + nomMethode + " retour=" + result);
  }
}
```

Le tag <aop:after-throwing> permet de définir un advice de type after throwing : cet advice permet d'invoquer une méthode de la classe qui encapsule les traitements de l'aspect après l'exécution ayant levé une exception des méthodes qui correspondent au point de coupe.

Exemple :

```
  <aop:config>
    <aop:aspect id="traceInvocationAspect" ref="tracerInvocation">
      <aop:pointcut id="traceInvocationPointcut"
        expression="execution(* fr.jmdoudoux.dej.spring.service.*ServiceImpl.*(..))" />
      <aop:after-throwing pointcut-ref="traceInvocationPointcut"
        method="afficherExceptionTrace" throwing="exception" />
    </aop:aspect>
  </aop:config>

  <bean id="tracerInvocation" class="fr.jmdoudoux.dej.spring.aspect.TraceInvocation" />
```

L'attribut pointcut-ref permet de fournir l'identifiant du point de coupe préalablement défini.

L'attribut method permet de préciser le nom de la méthode de la classe encapsulant les traitements de l'aspect qui sera exécutée.

L'attribut throwing permet de préciser le nom du paramètre de la méthode qui va contenir l'exception levée durant l'exécution. Dans ce cas, la méthode de l'aspect doit avoir un paramètre du type Exception à traiter pour les méthodes du point de coupe dont le nom correspond à celui fourni dans l'attribut throwing. La méthode de l'aspect peut aussi avoir un paramètre optionnel de type org.aspectj.lang.JoinPoint.StaticPart qui permet d'obtenir des informations sur le point de jonction.

Exemple :

```
package fr.jmdoudoux.dej.spring.aspect;

import org.apache.log4j.Logger;
import org.aspectj.lang.JoinPoint.StaticPart;

public class TraceInvocation {

  private static Logger LOGGER = Logger.getLogger(TraceInvocation.class);

  public void afficherExceptionTrace(final StaticPart staticPart,
      final Exception exception) throws Throwable {
    String nomMethode = staticPart.getSignature().toLongString();
    LOGGER.error("Exception durant la methode :  " + nomMethode, exception);
  }
}
```

Le tag <aop:after> permet de définir un advice de type after : cet advice permet d'invoquer une méthode de la classe qui encapsule les traitements de l'aspect après l'exécution des méthodes qui correspondent au point de coupe qu'une exception soit levée ou non durant leur exécution.

Exemple :

```
  <aop:config>
    <aop:aspect id="traceInvocationAspect" ref="tracerInvocation">
      <aop:pointcut id="traceInvocationPointcut"
        expression="execution(* fr.jmdoudoux.dej.spring.service.*ServiceImpl.*(..))" />
      <aop:after pointcut-ref="traceInvocationPointcut"
        method="afficherFinTrace" />
    </aop:aspect>
  </aop:config>

  <bean id="tracerInvocation" class="fr.jmdoudoux.dej.spring.aspect.TraceInvocation" />
```

L'attribut pointcut-ref permet de fournir l'identifiant du point de coupe préalablement défini.

L'attribut method permet de préciser le nom de la méthode de la classe encapsulant les traitements de l'aspect qui sera exécutée. Un paramètre de type org.aspectj.lang.JoinPoint dans la signature de la méthode permet d'obtenir des informations sur le point de jonction.

Exemple :

```
package fr.jmdoudoux.dej.spring.aspect;

import org.apache.log4j.Logger;
import org.aspectj.lang.JoinPoint.StaticPart;

public class TraceInvocation {

  private static Logger LOGGER = Logger.getLogger(TraceInvocation.class);

  public void afficherFinTrace(final JoinPoint joinpoint) throws Throwable {
    final Object[] args = joinpoint.getArgs();
    final StringBuffer sb = new StringBuffer();
    sb.append(joinpoint.getSignature().toString());
    sb.append(" avec les parametres : (");

    for (int i = 0; i < args.length; i++) {
      sb.append(args[i]);
      if (i < args.length - 1) {
        sb.append(", ");
      }
    }
    sb.append(")");

    LOGGER.info("Fin methode : " + sb);
  }
}
```

Le tag <aop:around> permet de définir un advice de type around : cet advice permet d'invoquer une méthode de la classe qui encapsule les traitements de l'aspect. Cette méthode va permettre de contrôler l'invocation des méthodes qui correspondent au point de coupe qu'une exception soit levée ou non durant leur exécution. Elle permet donc d'exécuter des traitements avant l'invocation, peut conditionner cette invocation et exécuter des traitements suite à cette invocation.

Exemple :

```
  <aop:config>
    <aop:aspect id="traceInvocationAspect" ref="tracerInvocation">
      <aop:pointcut id="traceInvocationPointcut"
        expression="execution(* fr.jmdoudoux.dej.spring.service.*ServiceImpl.*(..))" />
      <aop:around pointcut-ref="traceInvocationPointcut"
        method="afficherTrace" />
    </aop:aspect>
  </aop:config>

  <bean id="tracerInvocation" class="fr.jmdoudoux.dej.spring.aspect.TraceInvocation" />
```

L'attribut pointcut-ref permet de fournir l'identifiant du point de coupe préalablement défini.

L'attribut method permet de préciser le nom de la méthode de la classe encapsulant les traitements de l'aspect qui sera exécutée. Un paramètre de type org.aspectj.lang.ProceedingJoinPoint dans la signature de la méthode permet d'obtenir des informations sur le point de jonction et de demander l'invocation de la méthode liée au point de jonction en utilisant la méthode proceed().

Exemple :

```
package fr.jmdoudoux.dej.spring.aspect;

import org.apache.log4j.Logger;
import org.aspectj.lang.ProceedingJoinPoint;

public class TraceInvocation {

  private static Logger LOGGER = Logger.getLogger(TraceInvocation.class);

  public Object afficherTrace(final ProceedingJoinPoint joinpoint)
      throws Throwable {
    String nomMethode = joinpoint.getTarget().getClass().getSimpleName() + "."
```

```
        + joinpoint.getSignature().getName();

    final Object[] args = joinpoint.getArgs();
    final StringBuffer sb = new StringBuffer();
    sb.append(joinpoint.getSignature().toString());
    sb.append(" avec les parametres : (");

    for (int i = 0; i < args.length; i++) {
      sb.append(args[i]);
      if (i < args.length - 1) {
        sb.append(", ");
      }
    }
    sb.append(")");

    LOGGER.info("Debut methode : " + sb);

    try {
      Object obj = joinpoint.proceed();
    } finally {
    LOGGER.info("Fin methode :  " + nomMethode + " retour=" + obj);
    }
    return obj;
  }
}
```

92.1.2.4. Une autre implémentation de l'aspect

Il est aussi possible d'implémenter l'aspect en utilisant deux points de coupe de type before et after-returning.

Le code de l'aspect doit alors avoir deux méthodes, une pour chaque point de coupe avec leurs signatures respectives.

Exemple :

```
package fr.jmdoudoux.dej.spring.aspect;

import org.apache.log4j.Logger;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.JoinPoint.StaticPart;
import org.aspectj.lang.ProceedingJoinPoint;
import org.springframework.core.Ordered;

public class TraceInvocation implements Ordered {

  private static Logger LOGGER = Logger.getLogger(TraceInvocation.class);
  private int order;

  public void afficherDebutTrace(final JoinPoint joinpoint) throws Throwable {
    final Object[] args = joinpoint.getArgs();
    final StringBuffer sb = new StringBuffer();
    sb.append(joinpoint.getSignature().toString());
    sb.append(" avec les parametres : (");

    for (int i = 0; i < args.length; i++) {
      sb.append(args[i]);
      if (i < args.length - 1) {
        sb.append(", ");
      }
    }
    sb.append(")");

    LOGGER.info("Debut methode : " + sb);
  }

  public void afficherFinTrace(final StaticPart staticPart, final Object result)
      throws Throwable {
    String nomMethode = staticPart.getSignature().toLongString();
    LOGGER.info("Fin methode :  " + nomMethode + " retour=" + result);
  }

  @Override
  public int getOrder() {
    return this.order;
  }

  public void setOrder(final int order) {
    this.order = order;
  }
}
```

La déclaration de l'aspect dans le fichier de configuration utilise les deux points de coupe.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

  <context:component-scan base-package="fr.jmdoudoux.dej.spring" />

  <aop:config>
    <aop:aspect id="monitorerPerfAspect" ref="monitorerPerf">
      <aop:pointcut id="methodeService"
        expression="execution(* fr.jmdoudoux.dej.spring.service.*ServiceImpl.*(..))" />
      <aop:around method="executer" pointcut-ref="methodeService" />
    </aop:aspect>
    <aop:aspect id="traceInvocationAspect" ref="tracerInvocation">
      <aop:pointcut id="traceInvocationPointcut"
        expression="execution(* fr.jmdoudoux.dej.spring.service.*ServiceImpl.*(..))" />
      <aop:before pointcut-ref="traceInvocationPointcut"
        method="afficherDebutTrace" />
      <aop:after-returning pointcut-ref="traceInvocationPointcut"
        method="afficherFinTrace" returning="result" />
    </aop:aspect>
  </aop:config>

  <bean id="monitorerPerf" class="fr.jmdoudoux.dej.spring.aspect.MonitorePerf">
    <property name="order" value="1" />
  </bean>

  <bean id="tracerInvocation" class="fr.jmdoudoux.dej.spring.aspect.TraceInvocation">
    <property name="order" value="2" />
  </bean>

  <bean id="personneService" class="fr.jmdoudoux.dej.spring.service.
  PersonneServiceImpl" />

</beans>
```

L'application de test demande une instance du service à Spring et invoque ses méthodes ajouter() et afficher().

Exemple :

```
package fr.jmdoudoux.dej.spring;

import org.apache.log4j.Logger;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import fr.jmdoudoux.dej.spring.entite.Personne;
import fr.jmdoudoux.dej.spring.service.PersonneService;
```

```
public class MonApp {

  private static Logger LOGGER = Logger.getLogger(MonApp.class);

  public static void main(final String[] args) throws Exception {

    LOGGER.info("Debut de l'application");

    ClassPathXmlApplicationContext appContext = new ClassPathXmlApplicationContext(
        new String[] { "appContext.xml" });

    PersonneService personneService = (PersonneService) appContext
        .getBean("personneService");

    LOGGER.info("Debut invocation du service");
    try {
      personneService.ajouter(new Personne());
    } catch (Exception e) {
      LOGGER.error("exception " + e.getClass().getName() + " interceptee");
    }

    personneService.afficher();

    LOGGER.info("Fin invocation du service");

    LOGGER.info("Fin de l'application");
  }
}
```

Les bibliothèques requises sont : spring-aop 3.0.5, spring-asm 3.0.5, spring-aspect 3.0.5, spring-beans 3.0.5, spring-core 3.0.5, spring-context 3.0.5, spring-expression 3.0.5, aspectjrt 1.6.8, aspectjweaver 1.6.8, aopalliance 1.0, commons-logging 1.1.1, log4j 1.2.16

La bibliothèque aspectjrt est requise car certaines classes sont utilisées lors de la mise en oeuvre de Spring AOP notamment dans le code de l'aspect.

Résultat :

```
2011-07-03 19:07:31,671  INFO [fr.jmdoudoux.dej.spring.MonApp] Debut de l'application
2011-07-03 19:07:32,718  INFO [fr.jmdoudoux.dej.spring.MonApp] Debut invocation du service
2011-07-03 19:07:32,718  INFO [fr.jmdoudoux.dej.spring.aspect.TraceInvocation] Debut methode
 : void fr.jmdoudoux.dej.spring.service.PersonneService.ajouter(Personne) avec les
parametres : (fr.jmdoudoux.dej.spring.entite.Personne@26d607)
2011-07-03 19:07:33,218  INFO [fr.jmdoudoux.dej.spring.aspect.TraceInvocation] Fin methode :
public abstract void fr.jmdoudoux.dej.spring.service.PersonneService.ajouter(
fr.jmdoudoux.dej.spring.entite.Personne)
retour=null
2011-07-03 19:07:33,218  INFO [fr.jmdoudoux.dej.spring.aspect.TraceInvocation] Debut methode
 : void fr.jmdoudoux.dej.spring.service.PersonneService.afficher() avec les parametres : ()
2011-07-03 19:07:33,468  INFO [fr.jmdoudoux.dej.spring.aspect.TraceInvocation] Fin methode :
public abstract void fr.jmdoudoux.dej.spring.service.PersonneService.afficher() retour=null
2011-07-03 19:07:33,468  INFO [fr.jmdoudoux.dej.spring.MonApp] Fin invocation du service
2011-07-03 19:07:33,468  INFO [fr.jmdoudoux.dej.spring.MonApp] Fin de l'application
```

92.1.2.5. La gestion de l'ordre des aspects

Il est possible de définir plusieurs aspects sur un même point de coupe. Dans ce cas, il peut être nécessaire de définir l'ordre d'exécution des aspects.

L'exemple de cette section va définir un aspect pour mesurer le temps d'exécution d'une méthode qui sera invoquée au même endroit que l'aspect qui trace les invocations.

Les aspects doivent alors implémenter l'interface Ordered qui ne définit qu'une seule méthode getOrder() renvoyant un entier.

Le plus simple est de définir un setter sur un champ order, ce qui va permettre de configurer la valeur du numéro d'ordre dans la configuration du contexte.

Exemple :

```
package fr.jmdoudoux.dej.spring.aspect;

import org.apache.log4j.Logger;
import org.aspectj.lang.ProceedingJoinPoint;
import org.springframework.core.Ordered;
import org.springframework.util.StopWatch;

public class MonitorePerf implements Ordered {
  private static Logger LOGGER = Logger.getLogger(MonitorePerf.class);
  private int order;

  public Object executer(final ProceedingJoinPoint joinpoint) throws Throwable {
    Object returnValue;
    StopWatch clock = new StopWatch(getClass().getName());
    try {
      clock.start(joinpoint.toString());
      returnValue = joinpoint.proceed();
    } finally {
      clock.stop();
      LOGGER.info("temps d'execution : " + clock.prettyPrint());
    }
    return returnValue;
  }

  @Override
  public int getOrder() {
    return this.order;
  }

  public void setOrder(final int order) {
    this.order = order;
  }
}
```

Dans le fichier de configuration du contexte, le second aspect est défini et l'ordre est précisé pour les deux aspects en utilisant leur propriété order.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

  <context:component-scan base-package="fr.jmdoudoux.dej.spring" />

  <aop:config>
    <aop:aspect id="monitorerPerfAspect" ref="monitorerPerf">
      <aop:pointcut id="methodeService"
        expression="execution(* fr.jmdoudoux.dej.spring.service.*ServiceImpl.*(..))" />
      <aop:around method="executer" pointcut-ref="methodeService" />
    </aop:aspect>
    <aop:aspect id="traceInvocationAspect" ref="tracerInvocation">
      <aop:pointcut id="traceInvocationPointcut"
        expression="execution(* fr.jmdoudoux.dej.spring.service.*ServiceImpl.*(..))" />
      <aop:around pointcut-ref="traceInvocationPointcut" method="afficherTrace" />
    </aop:aspect>
  </aop:config>

  <bean id="monitorerPerf" class="fr.jmdoudoux.dej.spring.aspect.MonitorePerf">
    <property name="order" value="1" />
  </bean>

  <bean id="tracerInvocation" class="fr.jmdoudoux.dej.spring.aspect.TraceInvocation">
    <property name="order" value="2" />
  </bean>

  <bean id="personneService" class="fr.jmdoudoux.dej.spring.service.PersonneServiceImpl" />

</beans>
```

Lors de l'exécution de l'exemple, les aspects sont invoqués dans l'ordre précisé.

Résultat :

```
2011-06-26 17:48:40,890  INFO [fr.jmdoudoux.dej.spring.MonApp] Debut de
 l'application
2011-06-26 17:48:41,921  INFO [fr.jmdoudoux.dej.spring.MonApp] Debut
invocation du service
2011-06-26 17:48:41,921  INFO [fr.jmdoudoux.dej.spring.aspect.TraceInvocation]
Debut methode : void fr.jmdoudoux.dej.spring.service.PersonneService.ajouter(
Personne) avec les parametres : (fr.jmdoudoux.dej.spring.entite.Personne@419d05)
2011-06-26 17:48:42,421  INFO [fr.jmdoudoux.dej.spring.aspect.TraceInvocation]
Fin methode :  PersonneServiceImpl.ajouter retour=null
2011-06-26 17:48:42,421  INFO [fr.jmdoudoux.dej.spring.aspect.MonitorePerf]
temps d'execution : StopWatch 'fr.jmdoudoux.dej.spring.aspect.MonitorePerf':
running time (millis) = 500
-----------------------------------------
ms     %     Task name
-----------------------------------------
00500  100 %  execution(void fr.jmdoudoux.dej.spring.service.PersonneService.
ajouter(Personne))

2011-06-26 17:48:42,421  INFO [fr.jmdoudoux.dej.spring.aspect.TraceInvocation]
Debut methode : void fr.jmdoudoux.dej.spring.service.PersonneService.afficher(
) avec les parametres : ()
2011-06-26 17:48:42,671  INFO [fr.jmdoudoux.dej.spring.aspect.TraceInvocation]
Fin methode :  PersonneServiceImpl.afficher retour=null
2011-06-26 17:48:42,671  INFO [fr.jmdoudoux.dej.spring.aspect.MonitorePerf]
temps d'execution : StopWatch 'fr.jmdoudoux.dej.spring.aspect.MonitorePerf':
running time (millis) = 250
-----------------------------------------
ms     %     Task name
-----------------------------------------
00250  100 %  execution(void fr.jmdoudoux.dej.spring.service.PersonneService.
afficher())

2011-06-26 17:48:42,671  INFO [fr.jmdoudoux.dej.spring.MonApp] Fin invocation
du service
2011-06-26 17:48:42,671  INFO [fr.jmdoudoux.dej.spring.MonApp] Fin de
l'application
```

Il est possible de simplifier encore plus le fichier de configuration en utilisant les annotations pour définir les beans et les aspects puisque les aspects sont aussi des beans.

92.1.3. Spring AOP avec les annotations AspectJ

La mise en oeuvre de Spring AOP peut se faire en utilisant les annotations d'AspectJ pour réaliser sa définition. Bien que ce soit les annotations d'AspectJ qui sont utilisées, le tissage ne va pas être réalisé avec AspectJ mais avec Spring AOP.

L'utilisation des annotations d'AspectJ requiert un Java SE 5 ou ultérieur.

La classe qui contient les traitements de l'aspect utilise les annotations d'AspectJ pour définir l'aspect, le point de coupe et les points de jonction. La configuration est dans ce cas simplifiée.

La classe de l'aspect doit être annotée avec @Aspect.

Exemple :

```
package fr.jmdoudoux.dej.spring.aspect;

import org.apache.log4j.Logger;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class TraceInvocation {

  private static Logger LOGGER = Logger.getLogger(TraceInvocation.class);

  @Around("traceInvocationPointcut()")
  public Object afficherTrace(final ProceedingJoinPoint joinpoint)
      throws Throwable {
    String nomMethode = joinpoint.getTarget().getClass().getSimpleName() + "."
        + joinpoint.getSignature().getName();

    final Object[] args = joinpoint.getArgs();
    final StringBuffer sb = new StringBuffer();
    sb.append(joinpoint.getSignature().toString());
    sb.append(" avec les parametres : (");

    for (int i = 0; i < args.length; i++) {
      sb.append(args[i]);
      if (i < args.length - 1) {
        sb.append(", ");
      }
    }
    sb.append(")");

    LOGGER.info("Debut methode : " + sb);

    try {
      Object obj = joinpoint.proceed();
    } finally {
      LOGGER.info("Fin methode :  " + nomMethode + " retour=" + obj);
    }
    return obj;
  }

  @Pointcut("execution(* fr.jmdoudoux.dej.spring.service.*ServiceImpl.*(..))")
  public void traceInvocationPointcut() {
  }

}
```

Le code de l'aspect à exécuter doit être contenu dans une méthode dont la signature est particulière et dépend de l'annotation utilisée pour préciser son point de jonction. Dans l'exemple ci-dessus, elle attend en paramètre un objet de type ProceedingJoinPoint puisque l'annotation utilisée est @Around.

L'annotation @Pointcut permet de définir des points de coupe. Elle s'utilise sur une méthode sans traitement d'une classe ou d'une interface annotée avec @Aspect. Cette classe ou interface peut avoir plusieurs méthodes annotées avec @Pointcut.

Exemple :

```
package fr.jmdoudoux.dej.spring.aspect;

@Aspect
public interface ITraceInvocation {
  @Pointcut("execution(* fr.jmdoudoux.dej.spring.service.*ServiceImpl.*(..))")
  void traceInvocationPointcut();
}
```

Comme la définition de l'aspect est faite avec des annotations, le fichier de configuration est grandement simplifié.

Pour utiliser des aspects définis avec les annotations d'AspectJ par Spring AOP, il faut utiliser le tag <aop:aspectj-autoproxy> dans le fichier de configuration. La classe qui encapsule l'aspect doit aussi être définie dans la configuration du contexte.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:aop="http://www.springframework.org/schema/aop"
 xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">
  <aop:aspectj-autoproxy/>
  <bean id="tracerInvocation" class="fr.jmdoudoux.dej.spring.aspect.TraceInvocation">
  </bean>
  <bean id="personneService" class="fr.jmdoudoux.dej.spring.service.PersonneServiceImpl" />
</beans>
```

Remarque : bien que les annotations d'AspectJ soient utilisées, le tissage n'est pas réalisé par AspectJ mais par Spring AOP en créant dynamiquement des proxys.

Les bibliothèques requises sont : spring-aop 3.0.5, spring-asm 3.0.5, spring-aspect 3.0.5, spring-beans 3.0.5, spring-core 3.0.5, spring-context 3.0.5, spring-expression 3.0.5, aspectjrt 1.6.8, aspectjweaver 1.6.8, aopalliance 1.0, commons-logging 1.1.1, log4j 1.2.16

Il est possible de simplifier encore plus le fichier de définition du contexte en déclarant les beans du service et des aspects grâce aux annotations. Pour cela, il faut permettre au conteneur Spring de détecter automatiquement ces beans, même les aspects, en les annotant avec @Component. Il est très important que le bean de l'aspect soit déclaré dans le contexte pour permettre à Spring AOP de créer le proxy requis : si l'aspect n'est pas annoté avec @Component, l'aspect ne sera tout simplement pas exécuté.

Exemple :

```
package fr.jmdoudoux.dej.spring.aspect;

import org.apache.log4j.Logger;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.JoinPoint.StaticPart;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
@Component
public class TraceInvocation {
  private static Logger LOGGER = Logger.getLogger(TraceInvocation.class);
  public void afficherDebutTrace(final JoinPoint joinpoint) throws Throwable {
    final Object[] args = joinpoint.getArgs();
    final StringBuffer sb = new StringBuffer();

    sb.append(joinpoint.getSignature().toString());
    sb.append(" avec les parametres : (");
    for (int i = 0; i < args.length; i++) {
      sb.append(args[i]);
      if (i < args.length - 1) {
        sb.append(", ");
      }
    }
    sb.append(")");

    LOGGER.info("Debut methode : " + sb);
  }

  public void afficherFinTrace(final StaticPart staticPart, final Object result)
      throws  Throwable {
    String nomMethode = staticPart.getSignature().toLongString();

    LOGGER.info("Fin methode : " + nomMethode + " retour=" + result);
  }

  @Around("traceInvocationPointcut()")
  public Object afficherTrace(final ProceedingJoinPoint joinpoint)
      throws Throwable {
    String nomMethode = joinpoint.getTarget().getClass().getSimpleName() + "."
        +  joinpoint.getSignature().getName();
    final Object[] args = joinpoint.getArgs();
    final StringBuffer sb = new StringBuffer();

    sb.append(joinpoint.getSignature().toString());
    sb.append(" avec les parametres : (");
    for (int i = 0; i < args.length; i++) {
      sb.append(args[i]);
      if (i < args.length - 1) {
        sb.append(", ");
      }
    }

    sb.append(")");

    LOGGER.info("Debut methode : " + sb);
    try {
      Object obj = joinpoint.proceed();
    } finally {
      LOGGER.info("Fin methode :  " + nomMethode + " retour=" + obj);
    }
    return obj;
  }

  @Pointcut("execution(* fr.jmdoudoux.dej.spring.service.*ServiceImpl.*(..))")
  public void traceInvocationPointcut() {
  }
}
```

Le fichier de configuration est alors minimaliste.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop"
xmlns:tx="http://www.springframework.org/schema/tx"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

  <context:component-scan base-package="fr.jmdoudoux.dej.spring" />
  <aop:aspectj-autoproxy/>

</beans>
```

Le résultat de l'exécution est le même.

Il est possible d'implémenter l'aspect en utilisant deux points de coupe de types before et after-returning et leurs annotations respectives.

Le code de l'aspect doit alors avoir deux méthodes, une pour chaque point de coupe avec leurs signatures respectives.

Exemple :

```
package fr.jmdoudoux.dej.spring.aspect;

import org.apache.log4j.Logger;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.JoinPoint.StaticPart;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class TraceInvocation {
  private static Logger LOGGER = Logger.getLogger(TraceInvocation.class);

  @Before("traceInvocationPointcut()")
  public void afficherDebutTrace(final JoinPoint joinpoint) throws Throwable {
    final Object[] args = joinpoint.getArgs();
    final StringBuffer sb = new StringBuffer();
    sb.append(joinpoint.getSignature().toString());
    sb.append(" avec les parametres : (");
    for (int i = 0; i < args.length; i++) {
      sb.append(args[i]);
      if (i < args.length - 1) {
        sb.append(", ");
      }
    }

    sb.append(")");

    LOGGER.info("Debut methode : " + sb);
  }

  @AfterReturning(pointcut = "traceInvocationPointcut()", returning = "result")
  public void afficherFinTrace(final StaticPart staticPart, final Object result)
      throws  Throwable {
    String nomMethode = staticPart.getSignature().toLongString();

    LOGGER.info("Fin methode : " + nomMethode + " retour=" + result);
  }

  @Pointcut("execution(* fr.jmdoudoux.dej.spring.service.*ServiceImpl.*(..))")
  public void traceInvocationPointcut() {
  }
}
```

Attention : toutes les fonctionnalités d'AspectJ ne sont pas prises en charge par Spring AOP. Une exception est levée si Spring AOP rencontre une fonctionnalité non supportée.

Résultat :

```
Caused by:
java.lang.IllegalArgumentException: DeclarePrecendence not presently supported
in Spring AOP
```

92.1.3.1. La gestion de l'ordre des aspects

La gestion de l'ordre des aspects définis avec les annotations AspectJ se fait de la même façon que pour les aspects définis dans la configuration du contexte puisqu'au final dans les deux cas, c'est Spring AOP qui prend en charge les aspects. Il faut aussi utiliser l'interface Ordered qui possède une seule méthode getOrder(). Cette méthode getOrder() doit renvoyer le numéro d'ordre d'exécution de l'aspect.

Exemple :

```
package fr.jmdoudoux.dej.spring.aspect;

import org.apache.log4j.Logger;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.JoinPoint.StaticPart;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.core.Ordered;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class TraceInvocation implements Ordered {
  private static Logger LOGGER = Logger.getLogger(TraceInvocation.class);
  private int order;

  @Around("traceInvocationPointcut()")
  public Object afficherTrace(final ProceedingJoinPoint joinpoint)
      throws Throwable {
    String nomMethode = joinpoint.getTarget().getClass().getSimpleName() + "."
        + joinpoint.getSignature().getName();
    final Object[] args = joinpoint.getArgs();
    final StringBuffer sb = new StringBuffer();

    sb.append(joinpoint.getSignature().toString());
    sb.append(" avec les parametres : (");
    for (int i = 0; i < args.length; i++) {
      sb.append(args[i]);
      if (i < args.length - 1) {
        sb.append(", ");
      }
    }
    sb.append(")");
    LOGGER.info("Debut methode : " + sb);
    try {
      Object obj = joinpoint.proceed();
    }
    finally {
      LOGGER.info("Fin methode :  " + nomMethode + " retour=" + obj);
    }
    return obj;
  }

  @Override
  public int getOrder() {
    return order;
  }

  @Value("2")
  public void setOrder(final int order) {
    this.order = order;
  }
  @Pointcut("execution(* fr.jmdoudoux.dej.spring.service.*ServiceImpl.*(..))")
  public void traceInvocationPointcut() {
  }
}
```

Le second aspect est défini avec son propre numéro d'ordre d'invocation.

Exemple :

```
package fr.jmdoudoux.dej.spring.aspect;
```

```
import org.apache.log4j.Logger;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.core.Ordered;
import org.springframework.stereotype.Component;
import org.springframework.util.StopWatch;

@Component
@Aspect
public class MonitorePerf implements Ordered {
  private static Logger LOGGER = Logger.getLogger(MonitorePerf.class);
  private int order;

  @Around("monitorePerfPointcut()")
  public Object executer(final ProceedingJoinPoint joinpoint) throws Throwable {
    Object returnValue;
    StopWatch clock = new StopWatch(getClass().getName());
    try {
      clock.start(joinpoint.toString());
      returnValue = joinpoint.proceed();
    } finally {
      clock.stop();
      LOGGER.info("temps d'execution : " + clock.prettyPrint());
    }
    return returnValue;
  }

  @Override
  public int getOrder() {
    return order;
  }

  @Pointcut("execution(* fr.jmdoudoux.dej.spring.service.*ServiceImpl.*(..))")
  public void monitorePerfPointcut() {
  }

  @Value("1")
  public void setOrder(final int order) {
    this.order = order;
  }
}
```

Il faut être attentif au numéro d'ordre attribué à chaque aspect selon le type d'advice utilisé et le résultat souhaité. Dans l'exemple ci-dessus, le but est d'avoir dans les logs les traces d'exécution suivies des informations sur le temps d'exécution. C'est pourtant l'aspect de monitoring qui possède le numéro d'ordre d'exécution 1 puisque l'aspect utilise l'advice around.

Comme toute la configuration est faite avec des annotations, le fichier de définition du contexte est toujours aussi simple.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop"
xmlns:tx="http://www.springframework.org/schema/tx"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

  <context:component-scan base-package="fr.jmdoudoux.dej.spring" />
  <aop:aspectj-autoproxy/>
```

```
</beans>
```

Lors de l'exécution, l'ordre est respecté.

Résultat :

```
2011-07-10 16:49:25,546  INFO [fr.jmdoudoux.dej.spring.MonApp] Debut de l'application
2011-07-10 16:49:26,546  INFO [fr.jmdoudoux.dej.spring.MonApp] Debut invocation du service
2011-07-10 16:49:26,578  INFO [fr.jmdoudoux.dej.spring.aspect.TraceInvocation]
Debut methode : void fr.jmdoudoux.dej.spring.service.PersonneService.ajouter(Personne)
avec les parametres : (fr.jmdoudoux.dej.spring.entite.Personne@55a338)
2011-07-10 16:49:27,078  INFO
[fr.jmdoudoux.dej.spring.aspect.TraceInvocation]
Fin methode :  PersonneServiceImpl.ajouter retour=null
2011-07-10 16:49:27,078  INFO [fr.jmdoudoux.dej.spring.aspect.MonitorePerf]

temps d'execution :
StopWatch 'fr.jmdoudoux.dej.spring.aspect.MonitorePerf':
running time (millis) = 500
-----------------------------------------
ms     %
Task name
-----------------------------------------
00500  100 %
execution(void fr.jmdoudoux.dej.spring.service.PersonneService.
ajouter(Personne))
2011-07-10 16:49:27,078  INFO [fr.jmdoudoux.dej.spring.aspect.TraceInvocation]
Debut methode : void fr.jmdoudoux.dej.spring.service.PersonneService.afficher()
avec les parametres : ()
2011-07-10 16:49:27,328  INFO [fr.jmdoudoux.dej.spring.aspect.TraceInvocation]
Fin methode :  PersonneServiceImpl.afficher retour=null
2011-07-10 16:49:27,328  INFO
[fr.jmdoudoux.dej.spring.aspect.MonitorePerf]
temps d'execution :
StopWatch 'fr.jmdoudoux.dej.spring.aspect.MonitorePerf':
running time (millis) = 250
-----------------------------------------
ms     %
Task name
-----------------------------------------
00250  100 %
execution(void fr.jmdoudoux.dej.spring.service.PersonneService.afficher())
2011-07-10 16:49:27,328  INFO [fr.jmdoudoux.dej.spring.MonApp] Fin invocation du service
2011-07-10 16:49:27,328  INFO [fr.jmdoudoux.dej.spring.MonApp] Fin de l'application
```

Pour modifier cet ordre, il suffit de changer la valeur de la propriété order. L'ordre d'exécution des aspects est alors inversé.

Résultat :

```
2011-07-10 16:57:57,703  INFO [fr.jmdoudoux.dej.spring.MonApp] Debut de l'application
2011-07-10 16:57:58,718  INFO [fr.jmdoudoux.dej.spring.MonApp] Debut invocation du service
2011-07-10 16:57:58,750  INFO [fr.jmdoudoux.dej.spring.aspect.TraceInvocation]
Debut methode : void fr.jmdoudoux.dej.spring.service.PersonneService.ajouter( Personne)
avec les parametres : (fr.jmdoudoux.dej.spring.entite.Personne@ b1074a)
2011-07-10 16:57:59,250  INFO [fr.jmdoudoux.dej.spring.aspect.MonitorePerf]
temps d'execution :
StopWatch 'fr.jmdoudoux.dej.spring.aspect.MonitorePerf':
running time (millis) = 500
-----------------------------------------
ms     %
Task name
-----------------------------------------
00500  100 %
execution(void fr.jmdoudoux.dej.spring.service.PersonneService.
ajouter(Personne))
2011-07-10 16:57:59,250  INFO [fr.jmdoudoux.dej.spring.aspect.TraceInvocation]
Fin methode :  PersonneServiceImpl.ajouter retour=null
2011-07-10 16:57:59,250  INFO [fr.jmdoudoux.dej.spring.aspect.TraceInvocation]
```

```
Debut methode : void fr.jmdoudoux.dej.spring.service.PersonneService.afficher()
avec les parametres : ()
2011-07-10 16:57:59,500  INFO [fr.jmdoudoux.dej.spring.aspect.MonitorePerf]
temps d'execution :
StopWatch 'fr.jmdoudoux.dej.spring.aspect.MonitorePerf':
running time (millis) = 250
-----------------------------------------
ms     %
Task name
-----------------------------------------
00250  100 %
execution(void fr.jmdoudoux.dej.spring.service.PersonneService.afficher())
2011-07-10 16:57:59,500  INFO [fr.jmdoudoux.dej.spring.aspect.TraceInvocation]

Fin methode :  PersonneServiceImpl.afficher retour=null
2011-07-10 16:57:59,500  INFO [fr.jmdoudoux.dej.spring.MonApp] Fin invocation du service
2011-07-10 16:57:59,500  INFO [fr.jmdoudoux.dej.spring.MonApp] Fin de l'application
```

92.2. AspectJ

Spring permet aussi une utilisation d'AspectJ pour mettre en oeuvre l'AOP : AspectJ propose un support très complet des possibilités offertes par l'AOP.

AspectJ utilise sa propre syntaxe pour la création d'un aspect mais surtout les aspects peuvent être tissés au runtime avec un agent dédié (classloader qui enrichit le bytecode lors de son chargement) ou à la compilation avec le compilateur dédié d'AspectJ.

Exemple :

```
public aspect HelloAspectJ {

  pointcut methodeMain() : execution(* main(..));

  after() returning : methodMain() {
    System.out.println("Hello AspectJ!");
  }
}
```

AspectJ 5 permet la création d'un aspect sous la forme d'une simple classe annotée avec des annotations dédiées comme @Aspect. L'aspect ci-dessus peut ainsi être défini avec l'annotation @Aspect.

Exemple :

```
@Aspect
public class HelloAspectJ {

  @Pointcut("execution(* main(..))")
  public void methodeMain() {}

  @AfterReturning("methodeMain()")
  public void saluer() {
    System.out.println("Hello AspectJ!");
  }
}
```

92.2.1. AspectJ avec LTW (Load Time Weaving)

Avec cette solution, le tissage des aspects va être réalisé dynamiquement, aux chargements des classes concernées, par un agent d'AspectJ.

Le code de l'application, du service, du bean et de l'aspect sont les mêmes que dans l'exemple utilisant Spring AOP avec les annotations d'AspectJ. Les différences vont se faire au niveau de la configuration du contexte Spring, des bibliothèques requises et de l'utilisation de l'agent dans la JVM.

Le fichier de configuration du contexte n'a plus besoin de contenir des définitions relatives aux aspects.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">

  <context:annotation-config />
  <context:spring-configured />

  <context:component-scan base-package="fr.jmdoudoux.dej.spring" />

</beans>
```

Il faut définir un fichier META-INF/aop.xml accessible par le classpath qui va contenir les informations sur le tissage à réaliser par AspectJ.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<aspectj>
  <aspects>
    <aspect name="fr.jmdoudoux.dej.spring.aspect.MonitorePerf" />
    <aspect name="fr.jmdoudoux.dej.spring.aspect.TraceInvocation" />
  </aspects>

  <weaver options="-XnoInline -Xlint:ignore -verbose -showWeaveInfo">
    <include name="fr.jmdoudoux.dej.spring.service..*" />
  </weaver>
</aspectj>
```

Le tag racine de ce fichier de configuration est le tag <aspectj>.

Les aspects doivent être déclarés chacun dans un tag <aspect> fils du tag <aspects>. L'attribut name permet de préciser le nom pleinement qualifié de l'aspect.

Le tag weaver permet de configurer le tissage des aspects. L'attribut options permet de définir les options du tisseur.

Les options de tissage «-verbose » et « -showWeaveInfo » sont particulièrement utiles dans l'environnement de développement pour obtenir des informations sur les opérations réalisées par AspectJ (enregistrement des aspects, leur tissage, ...).

Le tag fils <include> permet de préciser sous la forme d'une expression régulière les classes qui sont concernées par le tissage.

Il faut lancer la JVM avec l'option -javaagent:chemin_vers_aspectjweaver.jar

exemple :

-javaagent:lib/aspectjweaver-1.6.1.jar

Le classpath de l'application contient les bibliothèques : org.springframework.asm-3.0.5.RELEASE.jar, org.springframework.aspects-3.0.5.RELEASE.jar, org.springframework.beans-3.0.5.RELEASE.jar, org.springframework.context-3.0.5.RELEASE.jar, org.springframework.core-3.0.5.RELEASE.jar, org.springframework.expression-3.0.5.RELEASE.jar, org.apache.commons.logging-1.1.1.jar, aspectrt.jar, log4j-1.2.16.jar

Résultat :

```
2011-08-09 18:57:48,968  INFO [fr.jmdoudoux.dej.spring.MonApp] Debut de
l'application
2011-08-09 18:57:49,265  INFO [org.springframework.context.support.
ClassPathXmlApplicationContext] Refreshing org.springframework.context.support.
ClassPathXmlApplicationContext@2d189c: startup date [Tue Aug 09 18:57:49 CEST
2011]; root of context hierarchy
2011-08-09 18:57:49,578  INFO [org.springframework.beans.factory.xml.
XmlBeanDefinitionReader] Loading XML bean definitions from class path resource [
appContext.xml]
2011-08-09 18:57:50,515  INFO [org.springframework.beans.factory.support.
DefaultListableBeanFactory] Pre-instantiating singletons in org.springframework.
beans.factory.support.DefaultListableBeanFactory@e8a0cd: defining beans [org.
springframework.context.annotation.internalConfigurationAnnotationProcessor,org.
springframework.context.annotation.internalAutowiredAnnotationProcessor,org.
springframework.context.annotation.internalRequiredAnnotationProcessor,org.
springframework.context.annotation.internalCommonAnnotationProcessor,org.
springframework.context.config.internalBeanConfigurerAspect,personne1,personne2,
personneService]; root of factory hierarchy
Invocation constructeur PersonneServiceImpl()
2011-08-09 18:57:50,656  INFO [fr.jmdoudoux.dej.spring.MonApp] Debut
invocation du service
2011-08-09 18:57:50,687  INFO [fr.jmdoudoux.dej.spring.aspect.TraceInvocation]
Debut methode : void fr.jmdoudoux.dej.spring.service.PersonneServiceImpl.
ajouter(Personne) avec les parametres : (fr.jmdoudoux.dej.spring.entite.
Personne@12f195)
2011-08-09 18:57:51,187  INFO [fr.jmdoudoux.dej.spring.aspect.TraceInvocation]
Fin methode :  public void fr.jmdoudoux.dej.spring.service.
PersonneServiceImpl.ajouter(fr.jmdoudoux.dej.spring.entite.Personne) retour=
null
2011-08-09 18:57:51,187  INFO [fr.jmdoudoux.dej.spring.aspect.MonitorePerf]
temps d'execution : StopWatch 'fr.jmdoudoux.dej.spring.aspect.MonitorePerf':
running time (millis) = 516
-----------------------------------------
ms     %     Task name
-----------------------------------------
00516  100 %  execution(void fr.jmdoudoux.dej.spring.service.
PersonneServiceImpl.ajouter(Personne))

2011-08-09 18:57:51,187  INFO [fr.jmdoudoux.dej.spring.aspect.TraceInvocation]
Debut methode : void fr.jmdoudoux.dej.spring.service.PersonneServiceImpl.
afficher() avec les parametres : ()
2011-08-09 18:57:51,437  INFO [fr.jmdoudoux.dej.spring.aspect.TraceInvocation]
Fin methode :  public void fr.jmdoudoux.dej.spring.service.
PersonneServiceImpl.afficher() retour=null
2011-08-09 18:57:51,437  INFO [fr.jmdoudoux.dej.spring.aspect.MonitorePerf]
temps d'execution : StopWatch 'fr.jmdoudoux.dej.spring.aspect.MonitorePerf':
running time (millis) = 250
-----------------------------------------
ms     %     Task name
-----------------------------------------
00250  100 %  execution(void fr.jmdoudoux.dej.spring.service.
PersonneServiceImpl.afficher())

2011-08-09 18:57:51,437  INFO [fr.jmdoudoux.dej.spring.MonApp] Fin invocation
du service
2011-08-09 18:57:51,437  INFO [fr.jmdoudoux.dej.spring.MonApp] Fin de
l'application
```

L'avantage de cette solution est qu'elle permet d'ajouter des aspects sur des beans qui ne sont pas gérés par Spring.

92.3. Spring AOP et AspectJ

Il est possible dans une même application d'utiliser des aspects tissés par Spring AOP et par AspectJ avec LTW.

Lorsque les aspects sont définis en utilisant les annotations d'AspectJ, il est nécessaire de préciser les aspects qui seront tissés par Spring AOP et ceux qui le seront par AspectJ.

Les aspects pris en charge par Spring AOP doivent être précisés en utilisant le tag <aop:include> dans le fichier de configuration du contexte de Spring.

Les aspects pris en charge par AspectJ doivent être précisés dans le fichier aop.xml pour un tissage dynamique.

Ainsi chaque tisseur prendra en charge les aspects qui le concernent.

L'exemple ci-dessous va mettre en oeuvre un aspect avec Spring AOP et un autre avec AspectJ.

Exemple :

```
@Component("traceInvocation")
@Aspect
public class TraceInvocation {
  private static Logger LOGGER = Logger.getLogger(TraceInvocation.class);

  @Around("traceInvocationPointcut()")
  public Object afficherTrace(final ProceedingJoinPoint joinpoint)
    throws Throwable {
    String nomMethode = joinpoint.getTarget().getClass().getSimpleName() + "."
        + joinpoint.getSignature().getName();
    final Object[] args = joinpoint.getArgs();
    final StringBuffer sb = new StringBuffer();

    sb.append(joinpoint.getSignature().toString());
    sb.append(" avec les parametres : (");
    for (int i = 0; i < args.length; i++) {
      sb.append(args[i]);
      if (i < args.length - 1) {
        sb.append(", ");
      }
    }

    sb.append(")");

    LOGGER.info("Debut methode : " + sb);
    Object obj = joinpoint.proceed();

    LOGGER.info("Fin methode : " + nomMethode + " retour=" + obj);
    return obj;
  }

  @Pointcut("execution(* fr.jmdoudoux.dej.spring.service.*ServiceImpl.*(..))")
  public void traceInvocationPointcut() {
  }
}
```

Cet aspect va être pris en charge par Spring AOP. Dans le fichier de déclaration du contexte, le bean qui encapsule l'aspect est fourni comme valeur de l'attribut name du tag <aop:include>. Ce tag est un tag fils du tag <aop:aspectj-autoproxy>

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:tx="http://www.springframework.org/schema/tx"
xmlns:aop="http://www.springframework.org/schema/aop"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

  <context:component-scan base-package="fr.jmdoudoux.dej.spring" />

  <aop:aspectj-autoproxy>
```

```
    <aop:include name="traceInvocation" />
  </aop:aspectj-autoproxy>

</beans>
```

Le second aspect est écrit de manière similaire en incluant en plus une gestion de son ordre d'exécution par rapport aux autres aspects.

Exemple :

```
package fr.jmdoudoux.dej.spring.aspect;

import org.apache.log4j.Logger;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.core.Ordered;
import org.springframework.stereotype.Component;
import org.springframework.util.StopWatch;

@Component
@Aspect
public class MonitorePerf implements Ordered {
  private static Logger LOGGER = Logger.getLogger(MonitorePerf.class);
  private int order;

  @Around("monitorePerfPointcut()")
  public Object executer(final ProceedingJoinPoint joinpoint) throws Throwable {
    Object returnValue;

    StopWatch clock = new StopWatch(getClass().getName());
    try {
      clock.start(joinpoint.toString());
      returnValue = joinpoint.proceed();
    } finally {
      clock.stop();
      LOGGER.info("temps d'execution : " + clock.prettyPrint());
    }
    return returnValue;
  }

  @Override
  public int getOrder() {
    return order;
  }

  @Pointcut("execution(* fr.jmdoudoux.dej.spring.service.*ServiceImpl.*(..))")
  public void monitorePerfPointcut() {
  }

  @Value("2")
  public void setOrder(final int order) {
    this.order = order;
  }
}
```

Cet aspect est pris en charge par AspectJ grâce à une configuration définie dans un fichier nommé aop-ajc.xml stocké dans le sous-répertoire META-INF.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<aspectj>
  <weaver options="-XnoInline -Xlint:ignore -verbose -showWeaveInfo">
    <include name="fr.jmdoudoux.dej.spring.service.*ServiceImpl"/>
  </weaver>
  <aspects>
    <aspect name="fr.jmdoudoux.dej.spring.aspect.MonitorePerf"/>
  </aspects>
</aspectj>
```

Ce fichier de configuration d'AspectJ permet de préciser :

- les options utilisées par le tisseur d'AspectJ en utilisant l'attribut options du tag <weaver>
- au tisseur sur quelles classes il doit réaliser son action en utilisant le tag <include> fils du tag <weaver> : dans l'exemple ci-dessus, c'est toutes les classes dont le nom se termine par ServiceImpl du package fr.jmdoudoux.dej.spring.service.
- quels aspects doivent être tissés en utilisant le tag <aspect> du tag <aspects> : dans l'exemple ci-dessus, c'est uniquement l'aspect encapsulé dans la classe MonitorePerf.

Les autres fichiers sont identiques à ceux des exemples précédents.

Pour permettre le tissage au runtime par AspectJ, il est nécessaire de préciser l'agent adéquat à la JVM en utilisant l'option -javaagent:chemin_vers_aspectjweaver.jar

Exemple :

-javaagent:lib/aspectjweaver-1.6.1.jar

Le classpath de l'application contient les bibliothèques : org.springframework.aop-3.0.5.RELEASE.jar, org.springframework.asm-3.0.5.RELEASE.jar, org.springframework.aspects-3.0.5.RELEASE.jar, org.springframework.beans-3.0.5.RELEASE.jar, org.springframework.context-3.0.5.RELEASE.jar, org.springframework.core-3.0.5.RELEASE.jar, org.springframework.expression-3.0.5.RELEASE.jar, org.springframework.instrument-3.0.5.RELEASE.jar, org.apache.commons.logging-1.1.1.jar, aspectrt.jar, log4j-1.2.16.jar, aopalliance-1.0.0.jar, com.springframework.org.aspectj.weaver.-.1.6.8.RELEASE.jar,

Résultat :

```
[AppClassLoader@fabe9] info AspectJ Weaver Version 1.6.8
built on Friday Jan 8, 2010 at 21:53:37 GMT
[AppClassLoader@fabe9] info register classloader sun.misc.Launcher$AppClassLoader@fabe9
[AppClassLoader@fabe9] info using configuration
file:/C:/java/api/spring-framework-3.0.5.RELEASE/dist/
org.springframework.aspects-3.0.5.RELEASE.jar!/META-INF/aop.xml
[AppClassLoader@fabe9] info using configuration
/C:/Documents%20and%20Settings/
jm/Documents/workspace-sts-2.5.1.RELEASE/TestSpringAOPetAspectJ/bin/META-INF/aop-ajc.xml
[AppClassLoader@fabe9] info register aspect org.springframework.beans.factory.
aspectj.AnnotationBeanConfigurerAspect
[AppClassLoader@fabe9] info register aspect org.springframework.scheduling.
aspectj.AnnotationAsyncExecutionAspect
[AppClassLoader@fabe9] info register aspect org.springframework.transaction.
aspectj.AnnotationTransactionAspect
[AppClassLoader@fabe9] info register aspect fr.jmdoudoux.dej.spring.aspect.
MonitorePerf
2011-07-04 19:25:13,140  INFO [fr.jmdoudoux.dej.spring.MonApp] Debut de l'application
2011-07-04 19:25:13,406  INFO [org.springframework.context.support.
ClassPathXmlApplicationContext] Refreshing  org.springframework.context.support.
ClassPathXmlApplicationContext@1860038: startup date [Tue Sep 06 19:25:13 CEST 2011];
root of context hierarchy
2011-07-04 19:25:13,687  INFO [org.springframework.beans.factory.xml.
 XmlBeanDefinitionReader] Loading XML bean
definitions from class path resource [appContext.xml]
[AppClassLoader@fabe9] weaveinfo Join point 'method-execution(void fr.
jmdoudoux.dej.spring.service.PersonneServiceImpl.afficher())'in Type 'fr.
jmdoudoux.dej.spring.service.PersonneServiceImpl'
(PersonneServiceImpl.java:17)
advised by around advice from 'fr.jmdoudoux.dej.spring.aspect.MonitorePerf'
 (MonitorePerf.java)
[AppClassLoader@fabe9] weaveinfo Join point 'method-execution(void fr.
jmdoudoux.dej.spring.service.PersonneServiceImpl.ajouter(fr.jmdoudoux.dej.
spring.entite.Personne))' in Type 'fr.jmdoudoux.dej.spring.service.
 PersonneServiceImpl'(PersonneServiceImpl.java:27) advised by around advice from
```

```
'fr.jmdoudoux.dej.spring.aspect.MonitorePerf' (MonitorePerf.java)
2011-07-04 19:25:14,750  INFO [org.springframework.beans.factory.support.
DefaultListableBeanFactory] Pre-instantiating singletons in org.springframework.
beans.factory.support.DefaultListableBeanFactory@79801c:
defining beans [
monitorePerf,traceInvocation,personne1,personne2,personneService,org.
springframework.context.annotation.internalConfigurationAnnotationProcessor,org.
springframework.context.annotation.internalAutowiredAnnotationProcessor,org.
springframework.context.annotation.internalRequiredAnnotationProcessor,org.
springframework.context.annotation.internalCommonAnnotationProcessor,org.
springframework.aop.config.internalAutoProxyCreator];
root of factory hierarchy
Invocation constructeur PersonneServiceImpl()
2011-07-04 19:25:15,015  INFO [fr.jmdoudoux.dej.spring.MonApp] Debut invocation du service
2011-07-04 19:25:15,046  INFO [fr.jmdoudoux.dej.spring.aspect.TraceInvocation]
Debut methode : void fr.jmdoudoux.dej.spring.service.PersonneService.ajouter(
Personne) avec les parametres : (fr.jmdoudoux.dej.spring.entite.Personne@6e96ff)
2011-07-04 19:25:15,546  INFO [fr.jmdoudoux.dej.spring.aspect.MonitorePerf]
temps d'execution : StopWatch 'fr.jmdoudoux.dej.spring.aspect.MonitorePerf':
running time (millis) = 500
-----------------------------------------
ms
%     Task name
-----------------------------------------
00500  100
%  execution(void fr.jmdoudoux.dej.spring.service.PersonneServiceImpl.ajouter(Personne))
2011-07-04 19:25:15,546  INFO [fr.jmdoudoux.dej.spring.aspect.TraceInvocation]
Fin methode :
PersonneServiceImpl.ajouter retour=null
2011-07-04 19:25:15,546  INFO [fr.jmdoudoux.dej.spring.aspect.TraceInvocation]
Debut methode : void fr.jmdoudoux.dej.spring.service.PersonneService.afficher(
) avec les parametres : ()
2011-07-04 19:25:15,796  INFO [fr.jmdoudoux.dej.spring.aspect.MonitorePerf]
temps d'execution : StopWatch 'fr.jmdoudoux.dej.spring.aspect.MonitorePerf':
running time (millis) = 250
-----------------------------------------
ms
%     Task name
-----------------------------------------
00250  100
%  execution(void fr.jmdoudoux.dej.spring.service.PersonneServiceImpl.afficher())
2011-07-04 19:25:15,796  INFO [fr.jmdoudoux.dej.spring.aspect.TraceInvocation]
Fin methode :
PersonneServiceImpl.afficher retour=null
2011-07-04 19:25:15,796  INFO [fr.jmdoudoux.dej.spring.MonApp] Fin invocation du service
2011-07-04 19:25:15,796  INFO [fr.jmdoudoux.dej.spring.MonApp] Fin de l'application
```

92.4. L'utilisation des namespaces

92.4.1. L'utilisation du tag <context:load-time-weaver>

Depuis la version 2.5 de Spring, il est possible d'utiliser le tag <context:load-time-weaver> dans le fichier de configuration pour demander le tissage en utilisant un agent fourni par Spring plutôt que l'agent d'AspectJ.

Attention : le tissage par l'agent Spring ne peut se faire que sur des objets qui sont définis dans le contexte et donc gérés dans le conteneur.

La bibliothèque spring-agent doit être ajoutée au classpath.

La JVM doit être lancée avec l'option -javaagent:chemin bibliothèque spring-agent.jar

Si le tag <context:load-time-weaver> est utilisé dans une application web déployée dans un conteneur web Tomcat, il faut préciser l'utilisation d'un classloader dédié dans le fichier META-INF/context.xml de la webapp

Exemple :

```
<?xml version='1.0' encoding='utf-8'?>
<Context reloadable="true">
  <Loader loaderClass=
    "org.springframework.instrument.classloading.tomcat.TomcatInstrumentableClassLoader"/>
</Context>
```

93. La gestion des transactions avec Spring

Chapitre 93

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Spring permet une gestion et une propagation des transactions. Depuis Spring 2.0, les transactions sont mises en oeuvre en utilisant l'AOP.

Généralement, c'est la couche service qui assure la gestion des transactions des traitements. La déclaration des méthodes qui doivent être transactionnelles peut se faire par déclaration dans la configuration de Spring ou par des annotations.

L'utilisation des transactions peut se faire par déclaration ou par programmation en utilisant une API dédiée. L'utilisation des transactions de manière déclarative est la façon la plus simple de les mettre en oeuvre car c'est celle qui limite les impacts dans le code de l'application.

La déclaration du comportement transactionnel se fait au niveau des méthodes de toutes les classes concernées. Cependant, Spring n'est pas en mesure de propager un contexte transactionnel dans des appels de méthodes distantes.

Depuis Spring 2.0, il n'est plus nécessaire de déclarer un bean de type TransactionProxyFactoryBean mais il faut utiliser les tags de l'espace de nommage tx.

La mise en oeuvre des transactions avec Spring se fait essentiellement de manière déclarative : la façon la plus simple d'utiliser une transaction avec Spring est d'ajouter la déclaration de l'espace de nommage tx, le tag <tx:annotation-driven/> dans le fichier de configuration et d'utiliser le tag @Transaction sur les classes et/ou les méthodes concernées.

Ce chapitre contient plusieurs sections :

- La gestion des transactions par Spring
- La propagation des transactions
- L'utilisation des transactions de manière déclarative
- La déclaration des transactions avec des annotations
- La gestion du rollback des transactions
- La mise en oeuvre d'aspects sur une méthode transactionnelle
- L'utilisation des transactions via l'API
- L'utilisation d'un gestionnaire de transactions reposant sur JTA

93.1. La gestion des transactions par Spring

La mise en oeuvre des transactions repose sur une abstraction qui permet de les mettre en oeuvre de façon similaire quelle que soit l'implémentation sous-jacente de la gestion des transactions (transactions globales avec JTA ou transactions locales avec JDBC, JPA, JDO, Hibernate, ...).

Un gestionnaire de transactions doit implémenter l'interface org.springframework.transaction.PlatformTransactionManager.

Spring propose plusieurs gestionnaires de transactions notamment :

- org.springframework.orm.hibernate3.HibernateTransactionManager : pour utiliser Hibernate
- org.springframework.transaction.jta.JtaTransactionManager : pour utiliser une implémentation de JTA fournie par un serveur d'applications
- org.springframework.jdbc.datasource.DataSourceTransactionManager : pour utiliser une datasource avec JDBC

L'utilisation des transactions est ainsi identique quelque soit la solution utilisée : seule la déclaration d'un gestionnaire de transactions est à faire et c'est lui qui va se charger de gérer les transactions de manière spécifique à la solution utilisée.

Une transaction gérée par Spring possède plusieurs caractéristiques :

- isolation : permet de préciser le niveau d'isolation de la transaction par rapport aux autres transactions.
- propagation : permet de préciser comment les traitements s'intègrent dans un contexte transactionnel
- timout : le temps maximum durant lequel la transaction peut s'exécuter. Au delà, la transaction est annulée (rollback).
- read only : permet de préciser si les données sont lues uniquement ou si elles peuvent être mises à jour ceci afin de permettre certaines optimisations

93.2. La propagation des transactions

Une transaction peut être logique ou physique.

Une transaction logique est gérée par Spring : une ou plusieurs transactions logiques permettent à Spring de déterminer le statut de la transaction physique.

La propagation PROPAGATION_REQUIRED crée une transaction logique pour chaque méthode dont le contexte transactionnel possède ce type de propagation. Durant la portée de cette transaction logique, celle-ci peut être validée ou annulée.

Comme les transactions logiques peuvent être imbriquées, pour indiquer à une transaction englobante qu'une transaction sous-jacente a été annulée, une exception de type UnexpectedRollbackException est levée.

La propagation PROPAGATION_REQUIRES_NEW crée une nouvelle transaction indépendante pour chaque méthode dont le contexte transactionnel possède ce type de propagation. Chaque contexte transactionnel dispose de sa propre transaction physique. Le rollback de la transaction n'a aucune incidence sur le rollback d'une transaction englobante.

La propagation PROPAGATION_NESTED utilise une seule transaction physique avec des savepoints. Il est donc possible de faire un rollback dans le contexte transactionnel jusqu'au précédent savepoint sans annuler l'intégralité de la transaction physique sous-jacente qui poursuit son exécution. Le gestionnaire de transaction doit permettre un support des savepoints ce qui pour le moment n'est possible qu'avec le DataSourceTransactionManager qui utilise les transactions JDBC.

93.3. L'utilisation des transactions de manière déclarative

Historiquement, l'utilisation des transactions avec Spring se faisait de manière déclarative dans le fichier de configuration XML. Cette déclaration pouvait devenir fastidieuse et source d'erreur car relativement compliquée et lourde en fonction de la taille de l'application. La possibilité de déclarer les transactions avec des annotations a grandement simplifié la tâche du développeur.

Le support des transactions par Spring de manière déclarative se fait à deux niveaux :

- une définition par des métadonnées en utilisant la configuration ou des annotations. Les métadonnées sont utilisées pour réaliser le tissage des aspects relatifs à la gestion des transactions (AspectJ).
- l'utilisation de proxys grâce à Spring AOP

93.3.1. La déclaration des transactions dans la configuration du contexte

La déclaration des transactions dans la configuration du contexte se fait dans le fichier XML en utilisant les espaces de nommages tx et aop.

Il faut définir un gestionnaire de transactions (TransactionManager) spécifique au mode de fonctionnement des accès aux données.

Il faut définir un advice lié au gestionnaire de transactions qui va permettre de déclarer le mode de propagation des transactions dans les méthodes désignées par des motifs.

Il faut enfin définir la configuration des aspects à tisser sur les méthodes qui doivent être transactionnelles. Celles-ci sont définies par des points de coupe grâce à des motifs auxquels sont associés un advice.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

  <bean id="datasource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">
    <property name="driverClassName" value="${jdbc.driverClassName}" />
    <property name="url" value="${jdbc.url}" />
    <property name="username" value="${jdbc.username}" />
    <property name="password" value="${jdbc.password}" />
  </bean>

  <bean id="txManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="datasource" />
  </bean>

  <tx:advice id="serviceTxAdvice" transaction-manager="txManager">
    <tx:attributes>
      <tx:method name="find*" propagation="REQUIRED"
        read-only="true" />
      <tx:method name="*" propagation="REQUIRED" />
    </tx:attributes>

  </tx:advice>

  <tx:advice id="daoTxAdvice" transaction-manager="txManager">
    <tx:attributes>
      <tx:method name="find*" propagation="REQUIRED" />
      <tx:method name="*" propagation="MANDATORY" />
    </tx:attributes>
  </tx:advice>

  <aop:config>
    <aop:pointcut id="serviceMethodes"
      expression="execution(*fr.jmdoudoux.dej.spring.service.*ServiceImpl.*(..))" />
    <aop:advisor advice-ref="serviceTxAdvice" pointcut-ref="serviceMethodes" />
  </aop:config>

  <aop:config>
    <aop:pointcut id="daoMethodes"
      expression="execution(*fr.jmdoudoux.dej.spring.dao.*DaoImpl*.*(..))" />
    <aop:advisor advice-ref="daoTxAdvice" pointcut-ref="daoMethodes" />
  </aop:config>

</beans>
```

Un advice est défini sous le nom txAdvice en lui associant le TransactionManager grâce à l'attribut transaction-manager : les méthodes dont le nom commence par get sont en lecture seule, les autres méthodes sont en lecture/écriture qui est le mode par défaut.

Le tag <tx:advice/> possède plusieurs attributs :

nom de l'attribut	Rôle	Valeur par défaut
Propagation	Préciser le mode de propagation de la transaction	REQUIRED
Isolation	Préciser le niveau d'isolation	DEFAULT
Transaction	Préciser si la transaction est en lecture seule ou lecture/écriture	read/write
Timeout	Préciser le timeout avant le rollback de la transaction	par défaut, c'est le timeout du gestionnaire de transactions sous-jacent utilisé, ou aucun si aucun timeout n'est supporté

Le tag <tx:method/> possède plusieurs attributs :

Nom de l'attribut	Rôle	Valeur par défaut
Name	nom de la ou des méthodes concernées en utilisant un motif dans lequel le caractère * peut être utilisé (exemple : get*) (obligatoire)	
Propagation	mode de propagation de la transaction	REQUIRED
Isolation	niveau d'isolation de la transaction	DEFAULT
Timeout	timeout de la transaction en secondes	-1
read-only	la transaction est en mode lecture seule	No
rollback-for	la ou les exceptions (séparées par un caractère ";") qui provoquent un rollback de la transaction	
no-rollback-for	la ou les exceptions (séparées par un caractère ";") qui ne provoquent pas un rollback de la transaction	

Remarque : par défaut, toutes les exceptions de type RuntimeException provoquent un rollback mais les exceptions de type checked ne provoquent pas de rollback.

Le tag <aop:config> permet la configuration du tissage des aspects relatifs aux transactions en définissant un point de coupe précisé sous la forme d'une expression régulière d'AspectJ fournie comme valeur de l'attribut expression.

Grâce au tissage, l'advice sera exécuté lors de l'invocation de chaque méthode définie par le point de coupe.

Généralement, toutes les méthodes des services doivent être transactionnelles. Pour cela, le point de coupe doit utiliser une expression qui désigne toutes les méthodes et tous les services.

Exemple :

```
<aop:config>
  <aop:pointcut id="serviceMethodes" expression="execution(*
fr.jmdoudoux.dej.spring.service.*.*(..))"/>
  <aop:advisor advice-ref="txAdvice" pointcut-ref="serviceMethodes"/>
</aop:config>
```

La gestion des transactions dans les services n'est pas toujours aussi générique et il peut être nécessaire de définir plusieurs pointcuts et advisors pour différentes configurations transactionnelles.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">
  <aop:config>
    <aop:pointcut id="defaultServiceOperation"
      expression="execution(* fr.jmdoudoux.dej.spring.service.*Service.*(..))" />
    <aop:pointcut id="noTxServiceOperation"
      expression="execution(* fr.jmdoudoux.dej.spring.service.*Cache.*(..))" />

    <aop:advisor pointcut-ref="defaultServiceOperation"
      advice-ref="defaultTxAdvice" />
    <aop:advisor pointcut-ref="noTxServiceOperation"
      advice-ref="noTxAdvice" />
  </aop:config>

  <tx:advice id="defaultTxAdvice">
    <tx:attributes>
      <tx:method name="get*" read-only="true" />
      <tx:method name="*" />
    </tx:attributes>
  </tx:advice>

  <tx:advice id="noTxAdvice">
    <tx:attributes>
      <tx:method name="*" propagation="NEVER" />
    </tx:attributes>
  </tx:advice>

  <bean id="personneService" class="fr.jmdoudoux.dej.spring.service.PersonneServiceImpl" />

  <bean id="personneCache" class="fr.jmdoudoux.dej.spring.service.cache.PersonneCache" />

</beans>
```

93.3.2. Un exemple de déclaration de transactions dans la configuration

Cette section fournit un exemple complet de déclaration d'un service dont les méthodes sont transactionnelles. La configuration des transactions se fait dans la configuration du contexte de l'application.

Le service est défini par une interface

Exemple :

```
package fr.jmdoudoux.dej.spring.service;

import java.util.List;

import fr.jmdoudoux.dej.spring.entite.Personne;

public interface PersonneService {
  void ajouter(Personne personne);

  Personne getParId(long id);

  List<Personne> getTous();
  void modifier(Personne personne);
}
```

Le service implémente l'interface.

Exemple :

```
package fr.jmdoudoux.dej.spring.service;

import java.util.List;
import fr.jmdoudoux.dej.spring.entite.Personne;

public class PersonneServiceImpl implements PersonneService {
  @Override
  public void ajouter(final Personne personne) {
    throw new UnsupportedOperationException();
  }

  @Override
  public Personne getParId(final long id) {
    throw new UnsupportedOperationException();
  }

  @Override
  public List<Personne> getTous() {
    throw new UnsupportedOperationException();
  }

  @Override
  public void modifier(final Personne personne) {
    throw new UnsupportedOperationException();
  }
}
```

Comme l'implémentation de toutes les méthodes du service lève une exception de type RuntimeException, les transactions provoqueront un rollback.

Les méthodes préfixées par get sont en lecture seule (read-only) alors que les autres méthodes sont utilisées pour des mises à jour (read-write).

Exemple :

```
<?xml version="1.0"
encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

      xmlns:aop="http://www.springframework.org/schema/aop"
      xmlns:tx="http://www.springframework.org/schema/tx"
      xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd

http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">
      <tx:advice id="txAdvice" transaction-manager="txManager">
            <tx:attributes>
                  <tx:method name="get*" read-only="true" />
                  <tx:methodname="*" />
            </tx:attributes>
      </tx:advice>
      <aop:config>
            <aop:pointcut id="personneServiceOperation"
      expression="execution(* fr.jmdoudoux.dej.spring.service.PersonneService.*(..))" />
            <aop:advisor advice-ref="txAdvice" pointcut-ref="personneServiceOperation" />
      </aop:config>
      <bean id="datasource" class="org.apache.commons.dbcp.BasicDataSource"
            destroy-method="close">
            <property name="driverClassName" value="org.apache.derby.jdbc.ClientDriver" />
            <property name="url" value="jdbc:derby://localhost/MaBaseDeTest" />
            <!--  property name="username" value="" / -->
            <!--   property name="password" value="" / -->
      </bean>
      <bean id="txManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
            <property name="dataSource" ref="datasource" />
      </bean>
      <bean id="personneService"
        class="fr.jmdoudoux.dej.spring.service.PersonneServiceImpl" />
</beans>
```

Dans l'exemple ci-dessous, le point de coupe concerne toutes les méthodes de la classe PersonneService en définissant un advisor qui le lie à l'advice.

Le PlatformTransactionManager est défini sous la forme d'un bean nommé txManager.

L'utilisation des transactions est alors transparente dans le code appelant.

Exemple :

```
package fr.jmdoudoux.dej.spring;
import org.apache.log4j.Logger;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import fr.jmdoudoux.dej.spring.entite.Personne;
import fr.jmdoudoux.dej.spring.service.PersonneService;

public class MonApp {
  private static Logger LOGGER = Logger.getLogger(MonApp.class);

  public static void main(final String[] args) throws Exception {
    LOGGER.info("Debut de l'application");

    ClassPathXmlApplicationContext appContext = new ClassPathXmlApplicationContext(
        new String[] {"appContext.xml" });

    PersonneService personneService = (PersonneService) appContext
      .getBean("personneService");

    LOGGER.info("Debut invocation du service");
    try {
      personneService.ajouter(new Personne());
    } catch (Exception e) {
      LOGGER.error("exception " + e.getClass().getName() + " interceptee");
    }

    LOGGER.info("Fin invocation du service");
    LOGGER.info("Fin de l'application");
  }
}
```

L'application de test charge le contexte, lui demande une instance du service et invoque sa méthode ajouter().

Le niveau de traces dans le fichier de configuration de Log4J est configuré sur debug pour permettre de voir le détail des actions réalisées par Spring pour gérer les transactions.

Exemple :

```
<?xml version="1.0" encoding= "UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration
xmlns:log4j="http://jakarta.apache.org/log4j/">
  <appender name="stdout" class="org.apache.log4j.ConsoleAppender">
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%d %p [%c] -%m%n"/>
    </layout>
  </appender>

  <root>
    <priority value="debug"/>
    <appender-ref ref="stdout"/>
  </root>
</log4j:configuration>
```

Le classpath de l'application contient les bibliothèques requises :

org.springframework.transaction-3.0.5.RELEASE.jar, org.springframework.aop-3.0.5.RELEASE.jar, org.springframework.asm-3.0.5.RELEASE.jar, org.springframework.aspects-3.0.5.RELEASE.jar, org.springframework.beans-3.0.5.RELEASE.jar, org.springframework.context-3.0.5.RELEASE.jar, org.springframework.core-3.0.5.RELEASE.jar, org.springframework.expression-3.0.5.RELEASE.jar, com.springsource.org.apache.commons.logging-1.1.1.jar, com.springsource.org.aopalliance-1.0.0.jar, commons-dbcp-1.4.jar, com.springsource.org.apache.commons.pool-1.5.3.jar, org.springframework.jdbc-3.0.5.RELEASE.jar, derbyclient.jar, derbynet.jar, org.springframework.instrument-3.0.5.RELEASE.jar, spring-aspects-3.0.5.RELEASE.jar, log4j-1.2.16.jar

La JVM est lancée avec l'option -javaagent:C:/java/api/aspectjweaver/aspectjweaver-1.6.1.jar pour activer l'agent AspectJ qui va se charger de tisser les aspects au runtime, notamment ceux concernant les transactions déclarées dans la configuration.

Lors de l'exécution de l'application, l'appel de la méthode du service provoque un rollback puisqu'une exception de type runtime est levée durant ses traitements.

Résultat :

```
2011-04-28 22:16:07,937  INFO [fr.jmdoudoux.dej.spring.MonApp] Debut de l'application
...
2011-04-28 22:16:10,000 DEBUG [org.springframework.beans.factory.support.
DefaultListableBeanFactory] Returning cached instance of singleton bean 'personneService'
2011-04-28 22:16:10,000  INFO [fr.jmdoudoux.dej.spring.MonApp] Debut invocation du service
2011-04-28 22:16:10,031 DEBUG [org.springframework.jdbc.datasource.
DataSourceTransactionManager] Creating new transaction with name
[fr.jmdoudoux.dej.spring.service.PersonneServiceImpl.ajouter]:
PROPAGATION_REQUIRED,ISOLATION_DEFAULT
2011-04-28 22:16:11,093 DEBUG [org.springframework.jdbc.datasource.
DataSourceTransactionManager] Acquired Connection [jdbc:derby://localhost:1527/MaBaseDeTest,
UserName=APP, Apache Derby Network Client JDBC Driver] for JDBC transaction
2011-04-28 22:16:11,109 DEBUG [org.springframework.jdbc.datasource.
DataSourceTransactionManager] Switching JDBC Connection [jdbc:derby://localhost:1527/
MaBaseDeTest, UserName=APP, Apache Derby Network Client JDBC Driver] to manual commit
2011-04-28 22:16:11,109 DEBUG [org.springframework.jdbc.datasource.
DataSourceTransactionManager] Initiating transaction rollback
2011-04-28 22:16:11,109 DEBUG [org.springframework.jdbc.datasource.
DataSourceTransactionManager] Rolling back JDBC transaction on Connection [jdbc:derby:
//localhost:1527/MaBaseDeTest, UserName=APP, Apache Derby Network Client JDBC Driver]
2011-04-28 22:16:11,109 DEBUG [org.springframework.jdbc.datasource.
DataSourceTransactionManager] Releasing JDBC Connection [jdbc:derby://localhost:1527/
MaBaseDeTest, UserName=APP, Apache Derby Network Client JDBC Driver] after transaction
2011-04-28 22:16:11,109 DEBUG [org.springframework.jdbc.datasource.DataSourceUtils]
 Returning JDBC Connection to DataSource
2011-04-28 22:16:11,109 ERROR [fr.jmdoudoux.dej.spring.MonApp]
 exception java.lang.UnsupportedOperationException interceptee
2011-04-28 22:16:11,109  INFO [fr.jmdoudoux.dej.spring.MonApp] Fin invocation du service
2011-04-28 22:16:11,109  INFO [fr.jmdoudoux.dej.spring.MonApp] Fin de l'application
```

Si la méthode du service ne lève pas d'exception durant son invocation, la transaction est validée par un commit.

Résultat :

```
2011-04-28 22:18:05,609  INFO
[fr.jmdoudoux.dej.spring.MonApp] Debut de l'application
...
```

```
2011-04-28 22:18:07,625  INFO
[fr.jmdoudoux.dej.spring.MonApp] Debut invocation du service
2011-04-28 22:18:07,671 DEBUG
[org.springframework.jdbc.datasource.DataSourceTransactionManager] Creating new
transaction with name [fr.jmdoudoux.dej.spring.service.PersonneServiceImpl.ajouter]:
PROPAGATION_REQUIRED,ISOLATION_DEFAULT
2011-04-28 22:18:08,703 DEBUG
[org.springframework.jdbc.datasource.DataSourceTransactionManager] Acquired
Connection [jdbc:derby://localhost:1527/MaBaseDeTest, UserName=APP, Apache
Derby Network Client JDBC Driver] for JDBC transaction
2011-04-28 22:18:08,734 DEBUG
[org.springframework.jdbc.datasource.DataSourceTransactionManager] Switching
JDBC Connection [jdbc:derby://localhost:1527/MaBaseDeTest, UserName=APP, Apache
Derby Network Client JDBC Driver] to manual commit
2011-04-28 22:18:08,734 DEBUG
[org.springframework.jdbc.datasource.DataSourceTransactionManager] Initiating
transaction commit
2011-04-28 22:18:08,734 DEBUG
[org.springframework.jdbc.datasource.DataSourceTransactionManager] Committing
JDBC transaction on Connection [jdbc:derby://localhost:1527/MaBaseDeTest,
UserName=APP, Apache Derby Network Client JDBC Driver]
2011-04-28 22:18:08,734 DEBUG
[org.springframework.jdbc.datasource.DataSourceTransactionManager] Releasing
JDBC Connection [jdbc:derby://localhost:1527/MaBaseDeTest, UserName=APP, Apache
Derby Network Client JDBC Driver] after transaction
2011-04-28 22:18:08,734 DEBUG
[org.springframework.jdbc.datasource.DataSourceUtils] Returning JDBC Connection
to DataSource
2011-04-28 22:18:08,734  INFO
[fr.jmdoudoux.dej.spring.MonApp] Fin invocation du service
2011-04-28 22:18:08,734  INFO
[fr.jmdoudoux.dej.spring.MonApp] Fin de l'application
```

93.4. La déclaration des transactions avec des annotations

L'annotation @Transactional peut être utilisée pour indiquer au conteneur les méthodes qui doivent s'exécuter dans un contexte transactionnel.

Si la déclaration des transactions se fait avec des annotations, il est tout de même nécessaire de déclarer le gestionnaire de transactions dans la configuration du contexte de Spring.

Pour permettre une utilisation de l'annotation @Transactional, il faut utiliser le tag <annotation-driven> de l'espace de nommage tx pour préciser à Spring que les annotations sont utilisées pour la définition des transactions.

Son attribut transaction-manager permet de préciser l'identifiant du bean qui encapsule le gestionnaire de transactions (TransactionManager) utilisé pour gérer les transactions : son utilisation n'est obligatoire que si l'id du gestionnaire de transactions est différent de "transactionManager".

Exemple :

```
<beans>
  <bean id="txManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="datasource" ref="dataSource"
  </bean>

  <tx:annotation-driven transaction-manager="txManager"/>

  <!-- ... -->
</beans>
```

La définition des transactions avec une annotation est plus simple à mettre en oeuvre, car il suffit d'annoter chaque méthode ou classe concernée avec @Transactional au lieu de la définir par des expressions régulières dans le fichier de configuration.

L'annotation org.springframework.transaction.annotation.Transactionnal s'utilise sur une classe ou une méthode. Sur une classe, elle s'applique automatiquement sur toutes les méthodes publiques de la classe.

L'annotation @Transactional possède plusieurs attributs :

- propagation : précise le mode de propagation de la transaction grâce à une énumération de type Propagation. La valeur par défaut est Propagation.REQUIRED
- readonly : booléen qui précise de façon informative au système de gestion des transactions sous-jacent si la transaction est en lecture seule (true) ou si elle effectue des mises à jour (false)
- isolation : précise le niveau d'isolation de la transaction grâce à une énumération de type Isolation. La valeur par défaut est Isolation.DEFAULT
- timeout : entier qui précise le timeout de la transaction

Exemple :

```
package fr.jmdoudoux.dej.spring.service;

@Service("personneService")
@Transactional
public class PersonneServiceImpl implements PersonneService {
  //...

  @Transactional(readOnly=true)
  public List<Personne> findAll() throws ServiceException {
    //...
  }

  //...
}
```

Il est fortement recommandé d'utiliser l'annotation @Transactional sur des classes et non sur des interfaces.

L'avantage de mettre en oeuvre les transactions par AOP est qu'il n'est pas nécessaire d'utiliser une API de Spring dans le code pour mettre en oeuvre les transactions. La mise en oeuvre reste aussi la même quelque soit l'implémentation du gestionnaire de transactions utilisée : la seule chose qui change c'est la configuration du transaction manager.

93.4.1. L'utilisation de l'annotation @Transactional

L'annotation @Transactional permet de délimiter une transaction (entre le début et la fin de la méthode) et de définir le comportement transactionnel d'une méthode.

L'annotation @Transactional possède plusieurs attributs :

Nom de l'attribut	Rôle	Valeur par défaut
propagation	mode de propagation de la transaction	PROPAGATION_REQUIRED
isolation	niveau d'isolation de la transaction	ISOLATIONDEFAULT
read-write	indique si la transaction est en lecture seule (false) ou lecture/écriture(true)	true
timeout		valeur par défaut de l'implémentation du système de gestion des transactions utilisé
rollbackFor	ensemble d'exceptions héritant de Throwable qui provoquent un rollback de la transaction si elles sont levées durant les traitements	
rollbackForClassname	ensemble de noms de classes héritant de Throwable pour lesquelles un rollback est effectué si des exceptions sont levées pendant les traitements	

noRollbackFor	ensemble d'exceptions héritant de Throwable qui ne provoquent pas un rollback de la transaction si elles sont levées durant les traitements	
noRollbackForClassname	ensemble de noms de classes héritant de Throwable pour lesquelles la levée d'une exception ne provoquera pas de rollback	

La simple utilisation de l'annotation @Transactional ne suffit pas car il ne représente que des métadonnées : il faut obligatoirement utiliser le tag <tx:annotation-driven> dans la configuration pour permettre à Spring d'ajouter les traitements relatifs aux aspects transactionaux sur les méthodes annotées.

Le tag <tx:annotation-driven> possède plusieurs attributs :

Nom de l'attribut	Rôle	Valeur par défaut
transaction-manager	nom du bean qui encapsule le gestionnaire de transactions (obligatoire uniquement si le nom ne correspond pas à la valeur par défaut)	transaction-manager
mode	les valeurs possibles sont proxy (utilisation de proxys) et aspectj (tissage des aspects avec AspectJ)	proxy
proxy-target-class	Permet de préciser le type de proxy utilisé (true : proxy reposant sur les interfaces, false : proxy reposant sur les classes). Ne doit être utilisé que si le mode est proxy	False
order	Permet de définir l'ordre des traitements exécutés sur les beans annotés avec @Transactional	Ordered.LOWEST PRECEDENCE

Attention : le tag <tx:annotation-driven> ne recherche les beans annotés avec @Transactional que dans le contexte dans lequel il est défini.

L'annotation @Transactional peut être utilisée sur une classe ou sur une méthode. Utilisée sur une classe, elle s'applique par défaut sur toutes les méthodes public de la classe sauf si la méthode est elle-même annotée avec @Transactional. Dans ce cas, c'est l'annotation sur la méthode qui est utilisée.

Exemple :

```
@Transactional(readOnly = true)
public class PersonneServicelmpl implements PersonneService {
  public Personne getParld(long id) {
    // traitements de la methode
  }

  @Transactional(readOnly = false, propagation = Propagation.REQUIRESNEW)
  public void modifier(Personne personne) {
    // traitements de la methode
  }
}
```

Seules les méthodes public doivent être annotées avec @Transactional lors de l'utilisation de proxys. Si des méthodes package-private, protected ou private sont annotées avec @Transactional, aucune erreur n'est produite à la compilation mais ces méthodes seront ignorées lors de l'utilisation des proxys.

Il est fortement recommandé de n'utiliser l'annotation @Transactional que dans des classes concrètes surtout dans le mode par défaut, le mode proxy.

Attention : dans le mode proxy, seules les invocations de méthodes depuis d'autres classes seront transactionnelles. Les invocations d'une méthode de la classe par une autre méthode de la classe ne sont pas transactionnelles même si la méthode invoquée est annotée avec @Transactional car ces invocations ne sont pas interceptées par le proxy.

Dans ce cas, il faut utiliser le mode AspectJ pour permettre un tissage des classes avec les aspects relatifs à la gestion des transactions pour les méthodes annotées. L'utilisation de ce mode requiert que la bibliothèque spring-aspects.jar soit ajoutée au classpath et le tissage (load-time weaving ou compile-time weaving) soit activé.

93.4.2. Le support de @Transactional par AspectJ

Il est possible d'utiliser le tissage d'aspects d'AspectJ pour intégrer le bytecode requis par le traitement des annotations @Transactional plutôt que d'utiliser des proxys gérés par le conteneur.

L'aspect à tisser est org.springframework.transaction.aspectj.AnnotationTransactionAspect contenu dans la bibliothèque spring-aspects.jar.

Il faut utiliser le tag <tx.annotation-driven> dans la configuration du contexte avec l'attribut mode="aspectj"

Le tissage peut se faire à la compilation ou au runtime.

L'annotation @Transactional peut alors être utilisée sur n'importe quelle méthode quelle que soit sa visibilité.

93.4.3. Un exemple de déclaration des transactions avec des annotations

L'exemple de cette section va utiliser Spring 3, AspectJ (en mode Load Time Weaving), JavaDB (en mode client/serveur), log4J.

La déclaration des transactions en utilisant les annotations est plus simple à mettre en oeuvre que la déclaration dans la configuration.

Il suffit d'utiliser l'annotation @Transactional sur les méthodes ou sur les classes concernées.

Exemple :

```
package fr.jmdoudoux.dej.spring.service;

import java.util.List;

import fr.jmdoudoux.dej.spring.entite.Personne;

public interface PersonneService {
  void ajouter(Personne personne);
  Personne getParId(long id);
  List<Personne> getTous();
  void modifier(Personne personne);
}
```

Le service implémente l'interface ci-dessus.

Exemple :

```
package fr.jmdoudoux.dej.spring.service;

import java.util.List;

import org.springframework.transaction.annotation.Transactional;

import fr.jmdoudoux.dej.spring.entite.Personne;

public class PersonneServiceImpl implements PersonneService {

  @Override
  @Transactional
  public void ajouter(final Personne personne) {
    throw new UnsupportedOperationException();
  }

  @Override
  @Transactional(readOnly = true)
  public Personne getParId(final long id) {
    throw new UnsupportedOperationException();
  }

  @Override
  @Transactional(readOnly = true)
  public List<Personne> getTous() {
    throw new UnsupportedOperationException();
  }

  @Override
  @Transactional
  public void modifier(final Personne personne) {
    throw new UnsupportedOperationException();
  }
}
```

Dans cette implémentation, toutes les méthodes transactionnelles lèvent une exception, ce qui permet de tester le rollback de la transaction.

La configuration du contexte est simplifiée car il suffit de déclarer le gestionnaire de transactions à utiliser et d'utiliser le tag <tx:annotation-driven>.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:aop="http:/www.springframework.org/schema/aop"
        xmlns:tx="http://www.springframework.org/schema/tx"
        xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

        <tx:annotation-driven mode="aspectj" transaction-manager="txManager" />

        <bean id="datasource" class="org.apache.commons.dbcp.BasicDataSource"
               destroy-method="close">
               <property name="driverClassName" value="org.apache.derby.jdbc.ClientDriver" />
               <property name="url" value="jdbc:derby://localhost/MaBaseDeTest" />
        </bean>

        <bean id="txManager"
          class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
               <property name="dataSource" ref="datasource" />
        </bean>

  <bean id="personneService" class="fr.jmdoudoux.dej.spring.service.PersonneServiceImpl" />
</beans>
```

L'attribut transaction-manager du tag <tx:annotation-driven> permet de préciser l'instance du gestionnaire de transactions à utiliser. Cet attribut peut être facultatif si le nom du bean du gestionnaire de transactions est "transactionManager".

Exemple :

```
package fr.jmdoudoux.dej.spring;

import org.apache.log4j.Logger;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import fr.jmdoudoux.dej.spring.entite.Personne;

import fr.jmdoudoux.dej.spring.service.PersonneService;

public class MonApp {
  private static Logger LOGGER = Logger.getLogger(MonApp.class);

  public static void main(final String[] args) throws Exception {
    LOGGER.info("Debut de l'application");

    ClassPathXmlApplicationContext appContext = new ClassPathXmlApplicationContext(
        new String[] { "appContext.xml" });
    PersonneService personneService = (PersonneService) appContext
        .getBean("personneService");
    LOGGER.info("Debutinvocation du service");
    try {
      personneService.ajouter(newPersonne());
    } catch (Exception e) {
      LOGGER.error("exception" + e.getClass().getName() + " interceptee");
    }
    LOGGER.info("Fin invocation du service");
    LOGGER.info("Fin de l'application");
  }
}
```

L'application de test charge le contexte, lui demande un instance du service et invoque sa méthode ajouter().

Le niveau de traces dans le fichier de configuration de Log4J est configuré sur debug pour permettre de voir le détail des actions réalisées par Spring pour gérer les transactions.

Exemple :

```
<?xml version="1.0" encoding= "UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration
xmlns:log4j="http://jakarta.apache.org/log4j/">
  <appender name="stdout" class="org.apache.log4j.ConsoleAppender">
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%d %p [%c] - %m%n"/>
    </layout>
  </appender>

  <root>
    <priority value="debug"/>
      <appender-ref ref="stdout"/>
  </root>
</log4j:configuration>
```

Le classpath de l'application contient les bibliothèques requises :

org.springframework.transaction-3.0.5.RELEASE.jar, org.springframework.aop-3.0.5.RELEASE.jar, org.springframework.asm-3.0.5.RELEASE.jar, org.springframework.aspects-3.0.5.RELEASE.jar, org.springframework.beans-3.0.5.RELEASE.jar, org.springframework.context-3.0.5.RELEASE.jar, org.springframework.core-3.0.5.RELEASE.jar, org.springframework.expression-3.0.5.RELEASE.jar, com.springsource.org.apache.commons.logging-1.1.1.jar, com.springsource.org.aopalliance-1.0.0.jar, commons-dbcp-1.4.jar, com.springsource.org.apache.commons.pool-1.5.3.jar, org.springframework.jdbc-3.0.5.RELEASE.jar, derbyclient.jar, derbynet.jar, org.springframework.instrument-3.0.5.RELEASE.jar, spring-aspects-3.0.5.RELEASE.jar, log4j-1.2.16.jar

La JVM est lancée avec l'option -javaagent:C:/java/api/aspectjweaver/aspectjweaver-1.6.1.jar pour activer l'agent AspectJ qui va se charger de tisser les aspects, notamment ceux concernant l'annotation @Transactional, au runtime.

Lors de l'exécution de l'application, l'appel de la méthode du service provoque un rollback puisqu'une exception de type runtime est levée durant ses traitements.

Résultat :

```
2011-04-26 22:17:48,453  INFO [fr.jmdoudoux.dej.spring.MonApp] Debut de l'application
```

```
2011-04-26 22:17:50,187  INFO [fr.jmdoudoux.dej.spring.MonApp] Debut invocation du service
...
2011-04-26 22:17:50,218 DEBUG [org.springframework.transaction.annotation
.AnnotationTransactionAttributeSource] Adding transactional method 'ajouter' with attribute:
PROPAGATION_REQUIRED,ISOLATION_DEFAULT; ''
2011-04-26 22:17:50,218 DEBUG [org.springframework.beans.factory.support.
DefaultListableBeanFactory] Returning cached instance of singleton bean 'txManager'
2011-04-26 22:17:50,250 DEBUG [org.springframework.jdbc.datasource.
DataSourceTransactionManager] Creating new transaction with name [fr.jmdoudoux.dej.spring.
service.PersonneServiceImpl.ajouter]: PROPAGATION_REQUIRED,ISOLATION_DEFAULT; ''
2011-04-26 22:17:51,296 DEBUG [org.springframework.jdbc.datasource.
DataSourceTransactionManager] Acquired Connection [jdbc:derby://localhost:1527/MaBaseDeTest,
UserName=APP, Apache Derby Network Client JDBC Driver] for JDBC transaction
2011-04-26 22:17:51,328 DEBUG [org.springframework.jdbc.datasource.
DataSourceTransactionManager] Switching JDBC Connection [jdbc:derby://localhost:1527/
MaBaseDeTest, UserName=APP, Apache Derby Network Client JDBC Driver] to manual commit
2011-04-26 22:17:51,328 DEBUG [org.springframework.jdbc.datasource.
DataSourceTransactionManager] Initiating transaction rollback
2011-04-26 22:17:51,328 DEBUG [org.springframework.jdbc.datasource.
DataSourceTransactionManager] Rolling back JDBC transaction on Connection [jdbc:derby:
//localhost:1527/MaBaseDeTest, UserName=APP, Apache Derby Network Client JDBC Driver]
2011-04-26 22:17:51,328 DEBUG [org.springframework.jdbc.datasource.
DataSourceTransactionManager] Releasing JDBC Connection [jdbc:derby://localhost:1527/
MaBaseDeTest, UserName=APP, Apache Derby Network Client JDBC Driver] after transaction
2011-04-26 22:17:51,328 DEBUG [org.springframework.jdbc.datasource.DataSourceUtils]
 Returning JDBC Connection to DataSource
2011-04-26 22:17:51,328 ERROR [fr.jmdoudoux.dej.spring.MonApp]
 exception java.lang.UnsupportedOperationException interceptee
2011-04-26 22:17:51,328  INFO [fr.jmdoudoux.dej.spring.MonApp] Fin invocation du service
2011-04-26 22:17:51,328  INFO [fr.jmdoudoux.dej.spring.MonApp] Fin de l'application
```

Si la méthode du service ne lève pas d'exception durant son invocation, la transaction est validée par un commit.

Résultat :

```
2011-04-26 22:25:17,484  INFO
[fr.jmdoudoux.dej.spring.MonApp] Debut de l'application
...
2011-04-26 22:25:19,250  INFO
[fr.jmdoudoux.dej.spring.MonApp] Debut invocation du service
2011-04-26 22:25:19,296 DEBUG
[org.springframework.transaction.annotation.AnnotationTransactionAttributeSource]
Adding transactional method 'ajouter' with attribute: PROPAGATION_REQUIRED,ISOLATION_DEFAULT;
''
2011-04-26 22:25:19,296 DEBUG
[org.springframework.beans.factory.support.DefaultListableBeanFactory]
Returning cached instance of singleton bean 'txManager'
2011-04-26 22:25:19,312 DEBUG [org.springframework.jdbc.datasource.
DataSourceTransactionManager] Creating new transaction with name
[fr.jmdoudoux.dej.spring.service.PersonneServiceImpl.ajouter]:
PROPAGATION_REQUIRED,ISOLATION_DEFAULT; ''
2011-04-26 22:25:20,390 DEBUG [org.springframework.jdbc.datasource.
DataSourceTransactionManager] Acquired Connection [jdbc:derby://localhost:1527/
MaBaseDeTest, UserName=APP, Apache Derby Network Client JDBC Driver] for JDBC transaction
2011-04-26 22:25:20,421 DEBUG
[org.springframework.jdbc.datasource.DataSourceTransactionManager] Switching
JDBC Connection [jdbc:derby://localhost:1527/MaBaseDeTest, UserName=APP, Apache
Derby Network Client JDBC Driver] to manual commit
2011-04-26 22:25:20,421 DEBUG
[org.springframework.jdbc.datasource.DataSourceTransactionManager] Initiating
transaction commit
2011-04-26 22:25:20,421 DEBUG
[org.springframework.jdbc.datasource.DataSourceTransactionManager] Committing
JDBC transaction on Connection [jdbc:derby://localhost:1527/MaBaseDeTest,
UserName=APP, Apache Derby Network Client JDBC Driver]
2011-04-26 22:25:20,421 DEBUG
[org.springframework.jdbc.datasource.DataSourceTransactionManager] Releasing
JDBC Connection [jdbc:derby://localhost:1527/MaBaseDeTest, UserName=APP, Apache
Derby Network Client JDBC Driver] after transaction
2011-04-26 22:25:20,421 DEBUG [org.springframework.jdbc.datasource.DataSourceUtils]
Returning JDBC Connection to DataSource
2011-04-26 22:25:20,421  INFO
```

```
[fr.jmdoudoux.dej.spring.MonApp] Fin invocation du service
2011-04-26 22:25:20,421  INFO
[fr.jmdoudoux.dej.spring.MonApp] Fin de l'application
```

93.5. La gestion du rollback des transactions

Spring utilise des règles particulières reposant sur les exceptions pour effectuer un rollback, au besoin, de la transaction. Par défaut, un rollback est effectué si une exception de type unchecked est levée dans les traitements de la transaction.

Ainsi par défaut, une transaction est annulée (rollback) si une exception de type RuntimeException ou Error est levée durant les traitements exécutés dans le contexte de la transaction. Donc par défaut, une transaction n'est pas annulée si une exception de type checked est levée dans les traitements.

Spring permet cependant une configuration fine des types d'exceptions qui vont provoquer un rollback de la transaction : ces règles peuvent être adaptées dans la déclaration de la transaction en précisant quelles exceptions provoquent un rollback ou non.

Il est aussi possible de forcer un rollback de la transaction par programmation en invoquant la méthode setRollback0nly() sur l'objet de type TransactionStatus.

93.5.1. La gestion du rollback dans la configuration

Si les transactions sont définies dans la configuration du contexte, les attributs rollback-for et no-rollback-for du tag <tx:method> permettent respectivement de préciser l'exception ou les types d'exceptions qui vont provoquer un rollback ou non.

Exemple :

```
<tx:advice id="txAdvice" transaction-manager="txManager">
  <tx:attributes>
    <tx:method name="get*" read-only="true" rollback-for="MonException"/>
    <tx:method name="*"/>
  </tx:attributes>
</tx:advice>
```

Dans cet exemple, un rollback de la transaction sera exécuté par Spring si une exception de type MonException est levée durant les traitements du contexte transactionnel.

Exemple :

```
<tx:advice id="txAdvice" transaction-manager="txManager">
  <tx:attributes>
    <tx:method name="get*" read-only="true" no-rollback-for="MonAutreException"/>
    <tx:method name="*"/>
  </tx:attributes>
</tx:advice>
```

Dans cet exemple, si seule une exception de type MonAutreException est levée durant les traitements, le commit de la transaction sera exécuté par Spring.

Exemple :

```
<tx:advice id="txAdvice">
  <tx:attributes>
    <tx:method name="*" rollback-for="Throwable" no-rollback-for="MonException"/>
  </tx:attributes>
</tx:advice>
```

Dans l'exemple ci-dessus, seule l'exception MonException ne va pas provoquer un rollback de la transaction.

93.5.2. La gestion du rollback via l'API

Il est aussi possible de forcer le rollback de la transaction par programmation en utilisant l'API.

Exemple :

```
public void maMethode() {
  try {
    // traitements
  } catch (MonException ex) {
    TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();
  }
}
```

Cette solution impose d'utiliser l'API de Spring ce qui n'est pas la meilleure solution : il est préférable dans la mesure du possible d'utiliser l'approche déclarative.

93.5.3. La gestion du rollback avec les annotations

L'annotation @Transactional possède plusieurs attributs permettant de gérer finement un éventuel rollback de la transaction.

L'attribut rollbackFor permet de préciser un tableau d'exceptions héritant de Throwable qui provoquent un rollback de la transaction si elles sont levées durant les traitements.

L'attribut rollbackForClassname permet de préciser un tableau de noms de classes héritant de Throwable qui si elles sont levées provoqueront un rollback de la transaction.

L'attribut noRollbackFor permet de préciser un tableau d'exceptions héritant de Throwable qui ne provoquent pas un rollback de la transaction si elles sont levées durant les traitements.

L'attribut noRollbackForClassname permet de préciser un tableau de noms de classes héritant de Throwable qui si elles sont levées ne provoqueront pas un rollback de la transaction.

Ces quatre attributs permettent de configurer de façon précise les conditions selon lesquelles un rollback de la transaction sera fait par les traitements de l'annotation @Transactional.

93.6. La mise en oeuvre d'aspects sur une méthode transactionnelle

Les transactions sont mises en oeuvre grâce à l'AOP mais il est aussi possible d'utiliser l'AOP pour ses propres besoins sur des méthodes transactionnelles.

Spring offre un moyen de configurer l'ordre d'exécution de ces aspects en implémentant l'interface Ordered.

Exemple :

```
package fr.jmdoudoux.dej.spring.aspect;

import org.aspectj.lang.ProceedingJoinPoint;
import org.springframework.util.StopWatch;
import org.springframework.core.Ordered;

public class MonitoringPerf implements Ordered {
  private int order;
  public int getOrder() {
    return this.order;
  }

  public void setOrder(int order) {
    this.order = order;
  }

  public Object executer(ProceedingJoinPoint call) throws Throwable {
    Object returnValue;
    StopWatch clock = new StopWatch(getClass().getName());
    try {
      clock.start(call.toShortString());
      returnValue = call.proceed();
    } finally {
      clock.stop();
      System.out.println(clock.prettyPrint());
    }
    return returnValue;
  }
}
```

Il faut définir l'aspect dans le fichier de configuration du contexte et préciser l'ordre d'invocation des aspects.

Le fichier de configuration si on déclare les transactions par annotations :

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

  <tx:annotation-driven mode="aspectj"
    transaction-manager="txManager" order="99" />

  <aop:config>
    <aop:aspect id="monitorerPerfAspect" ref="monitorerPerf">
      <aop:pointcut id="methodeService"
        expression="execution(* fr.jmdoudoux.dej.spring.service.*ServiceImpl.*(..))" />
      <aop:around method="executer" pointcut-ref="methodeService" />
    </aop:aspect>
  </aop:config>

  <bean id="datasource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">
    <property name="driverClassName" value="org.apache.derby.jdbc.ClientDriver" />
    <property name="url" value="jdbc:derby://localhost/MaBaseDeTest" />
  </bean>

  <bean id="txManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="datasource" />
  </bean>

  <bean id="monitorerPerf" class="fr.jmdoudoux.dej.spring.aspect.MonitorerPerf">
    <property name="order" value="1" />
  </bean>

  <bean id="personneService"
    class="fr.jmdoudoux.dej.spring.service.PersonneServiceImpl" />

</beans>
```

L'ordre d'invocation des aspects est défini grâce à la valeur fournie aux attributs order du bean qui encapsule l'aspect et de l'attribut order du tag <annotation-driven>.

Le fichier de configuration si on déclare les transactions dans le fichier de configuration :

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

  <tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
      <tx:method name="get*" read-only="true" />
      <tx:method name="*" />
    </tx:attributes>
  </tx:advice>

  <aop:config>
    <aop:pointcut id="personneServiceOperation"
      expression="execution(* fr.jmdoudoux.dej.spring.service.PersonneService.*(..))" />
    <aop:advisor advice-ref="txAdvice" pointcut-ref="personneServiceOperation"
      order="99" />

    <aop:aspect id="monitorerPerfAspect" ref="monitorerPerf">
      <aop:around method="executer" pointcut-ref="personneServiceOperation" />
    </aop:aspect>

  </aop:config>

  <bean id="datasource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">
    <property name="driverClassName" value="org.apache.derby.jdbc.ClientDriver" />
    <property name="url" value="jdbc:derby://localhost/MaBaseDeTest" />
  </bean>

  <bean id="txManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="datasource" />
  </bean>

  <bean id="monitorerPerf" class="fr.jmdoudoux.dej.spring.aspect.MonitorerPerf">
    <property name="order" value="1" />
  </bean>

  <bean id="personneService"
    class="fr.jmdoudoux.dej.spring.service.PersonneServiceImpl" />

</beans>
```

L'ordre d'invocation des aspects est défini grâce à la valeur fournie aux attributs order du bean qui encapsule l'aspect et de l'advisor qui concerne la gestion des transactions.

L'exécution de l'application affiche le temps d'exécution suite à l'invocation de la méthode

Résultat :

```
2011-04-28 22:27:47,375  INFO [fr.jmdoudoux.dej.spring.MonApp] Debut de l'application
...
```

```
2011-04-28 22:27:49,578  INFO [fr.jmdoudoux.dej.spring.MonApp] Debut invocation du service
2011-04-28 22:27:49,593 DEBUG [org.springframework.beans.factory.support.
DefaultListableBeanFactory] Returning cached instance of singleton bean 'monitorerPerf'
2011-04-28 22:27:49,640 DEBUG [org.springframework.jdbc.datasource.
DataSourceTransactionManager] Creating new transaction with name
[fr.jmdoudoux.dej.spring.service.PersonneServiceImpl.ajouter]:
 PROPAGATION_REQUIRED,ISOLATION_DEFAULT
2011-04-28 22:27:50,687 DEBUG [org.springframework.jdbc.datasource.
DataSourceTransactionManager] Acquired Connection
[jdbc:derby://localhost:1527/MaBaseDeTest, UserName=APP, Apache Derby Network Client
JDBC Driver] for JDBC transaction
2011-04-28 22:27:50,703 DEBUG [org.springframework.jdbc.datasource.
DataSourceTransactionManager] Switching JDBC Connection
[jdbc:derby://localhost:1527/MaBaseDeTest, UserName=APP, Apache Derby Network Client
JDBC Driver] to manual commit
2011-04-28 22:27:50,703 DEBUG [org.springframework.jdbc.datasource.
DataSourceTransactionManager] Initiating transaction commit
2011-04-28 22:27:50,703 DEBUG [org.springframework.jdbc.datasource.
DataSourceTransactionManager] Committing JDBC transaction on Connection [jdbc:derby:
//localhost:1527/MaBaseDeTest, UserName=APP, Apache Derby Network Client JDBC Driver]
2011-04-28 22:27:50,703 DEBUG [org.springframework.jdbc.datasource.
DataSourceTransactionManager] Releasing JDBC Connection
[jdbc:derby://localhost:1527/MaBaseDeTest, UserName=APP, Apache Derby Network Client
 JDBC Driver] after transaction
2011-04-28 22:27:50,703 DEBUG [org.springframework.jdbc.datasource.DataSourceUtils]
 Returning JDBC Connection to DataSource
2011-04-28 22:27:50,703 DEBUG [fr.jmdoudoux.dej.spring.aspect.MonitorerPerf]
temps d'execution : StopWatch
'fr.jmdoudoux.dej.spring.aspect.MonitorerPerf': running time (millis) = 1094
-----------------------------------------
ms     %     Task name
-----------------------------------------
01094  100 %  execution(PersonneService.ajouter(..))

2011-04-28 18:27:50,703  INFO [fr.jmdoudoux.dej.spring.MonApp] Fin invocation du service
2011-04-28 18:27:50,703  INFO [fr.jmdoudoux.dej.spring.MonApp] Fin de l'application
```

93.7. L'utilisation des transactions via l'API

Pour la mise en oeuvre des transactions par programmation, Spring propose deux solutions :

- utiliser la classe TransactionTemplate
- utiliser directement une implémentation de l'interface PlatformTransactionManager

L'utilisation de ces deux solutions lie fortement le code de l'application avec Spring puisqu'elles utilisent des API dédiées. Elles ne sont donc à utiliser que pour des besoins très spécifiques et une solution déclarative est préférable.

93.7.1. L'utilisation de la classe TransactionTemplate

Le principe de la classe TransactionTemplate repose sur l'utilisation de callbacks comme pour les autres templates Spring.

Il faut écrire une implémentation de TransactionCallback, généralement sous la forme d'une classe anonyme interne qui va contenir les traitements à exécuter dans le contexte transactionnel.

Il suffit alors de passer une instance de cette implémentation en paramètre de la méthode execute() de la classe TransactionTemplate.

Exemple :

```
package fr.jmdoudoux.dej.spring.service;

import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.TransactionCallback;
import org.springframework.transaction.support.TransactionTemplate;

import fr.jmdoudoux.dej.spring.entite.Personne;

public class MonServiceImpl implements MonService {

  private final TransactionTemplate transactionTemplate;

  public MonServiceImpl(final PlatformTransactionManager transactionManager) {
    this.transactionTemplate = new TransactionTemplate(transactionManager);
  }

  protected boolean maMethode(final Personne personne) {
    // traitement de la methode
    return true;
  }

  public Object maMethodeTransactionelle(final Personne personne) {
    return transactionTemplate.execute(new TransactionCallback() {

      public Object doInTransaction(final TransactionStatus status) {
        return maMethode(personne);
      }
    });
  }
}
```

L'instance de TransactionTemplate est utilisée par toutes les méthodes d'instances.

L'injection de l'instance du gestionnaire de transactions se fait en passant par le constructeur.

Si la méthode transactionnelle ne renvoie aucun résultat, il faut utiliser la classe TransactionCallbackWithoutResult.

Exemple :

```
package fr.jmdoudoux.dej.spring.service;

import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.TransactionCallbackWithoutResult;
import org.springframework.transaction.support.TransactionTemplate;

import fr.jmdoudoux.dej.spring.entite.Personne;

public class MonServiceImpl implements MonService {

  private final TransactionTemplate transactionTemplate;

  public MonServiceImpl(final PlatformTransactionManager transactionManager) {
    this.transactionTemplate = new TransactionTemplate(transactionManager);
  }

  protected void maMethode(final Personne personne) {
    // traitement de la methode
  }

  public Object maMethodeTransactionelle(final Personne personne) {
    return transactionTemplate.execute(new TransactionCallbackWithoutResult() {

      public void doInTransactionWithoutResult(final TransactionStatus status) {
        maMethode(personne);
      }
    });
  }
}
```

Il est possible de demander explicitement l'annulation de la transaction dans le code du callback en invoquant la méthode setRollbackOnly() de l'instance de TransactionStatus fournie en paramètre.

Exemple :

```
package fr.jmdoudoux.dej.spring.service;

import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.TransactionCallback;
import org.springframework.transaction.support.TransactionTemplate;

import fr.jmdoudoux.dej.spring.entite.Personne;

public class MonServiceImpl implements MonService {

  private final TransactionTemplate transactionTemplate;

  public MonServiceImpl(final PlatformTransactionManager transactionManager) {
    this.transactionTemplate = new TransactionTemplate(transactionManager);
  }

  protected boolean maMethode(final Personne personne) {
    // traitement de la methode
    return true;
  }

  public Object maMethodeTransactionelle(final Personne personne) {
    return transactionTemplate.execute(new TransactionCallback() {

      @Override
      public Object doInTransaction(final TransactionStatus status) {
        boolean resultat = false;
        try {
          resultat = maMethode(personne);
        } catch (MonException ex) {
          status.setRollbackOnly();
        }
        return resultat;
      }
    });
  }
}
```

La classe TransactionTemplate possède plusieurs méthodes pour permettre de configurer le contexte transactionnel.

Exemple :

```
package fr.jmdoudoux.dej.spring.service;

import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.TransactionDefinition;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.TransactionCallback;
import org.springframework.transaction.support.TransactionTemplate;

import fr.jmdoudoux.dej.spring.entite.Personne;

public class MonServiceImpl implements MonService {

  private final TransactionTemplate transactionTemplate;

  public MonServiceImpl(final PlatformTransactionManager transactionManager) {
    this.transactionTemplate = new TransactionTemplate(transactionManager);
    this.transactionTemplate
        .setIsolationLevel(TransactionDefinition.ISOLATION_READ_UNCOMMITTED);
    this.transactionTemplate.setTimeout(60);

  }

  // ...
}
```

Les instances de TransactionTemplate sont threadsafe mais elles contiennent la configuration. Il est donc possible de partager une instance de TransactionTemplate mais il est nécessaire d'avoir autant d'instances que de configurations différentes.

Il est donc possible de déclarer un bean de type TransactionTemplate et de le paramétrer dans la configuration puis de l'injecter dans les services qui en ont besoin.

Exemple :

```
  <bean
id="monTransactionTemplate"

class="org.springframework.transaction.support.TransactionTemplate">

<property name="isolationLevelName"
value="ISOLATION_READ_UNCOMMITTED" />

<property name="timeout" value="30" />

</bean>
```

93.7.2. L'utilisation directe d'un PlatformTransactionManager

Par programmation, il est possible d'utiliser directement une instance de type PlatformTransactionManager pour gérer les transactions.

Dans le bean, il faut permettre une injection de l'instance de PlatformTransactionManager par Spring.

L'interface TransactionDefinition définit la configuration d'une transaction.

Le plus simple est de créer une instance de la classe DefaultTransactionDefinition puis d'invoquer sa méthode setName() pour lui attribuer un nom et invoquer toutes les méthodes utiles pour configurer la transaction.

La méthode getTransaction() qui attend en paramètre une instance de TransactionDefinition renvoie une instance de l'interface TransactionStatus.

L'interface TransactionStatus définit le statut d'une transaction. Elle propose plusieurs méthodes :

Méthode	Rôle
boolean hasSavepoint()	Renvoie un booléen qui précise si la transaction a été créée comme englobée dans une transaction possédant un savepoint
boolean isCompleted()	Renvoie un booléen qui précise si la transaction est terminée (par un rollback ou un commit)
boolean isNewTransaction()	Renvoie un booléen qui précise si la transaction est nouvelle
boolean isRollbackOnly()	Renvoie un booléen qui précise si la transaction est marquée comme devant être annulée
void setRollbackOnly()	Demande l'annulation de la transaction

Exemple :

```
    DefaultTransactionDefinition def = new DefaultTransactionDefinition();
    def.setName("MaTransaction");
    def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);
    TransactionStatus status = txManager.getTransaction(def);
    try {
      // les traitements
    } catch (MonException ex) {
      txManager.rollback(status);
      throw ex;
    }
    txManager.commit(status);
```

93.8. L'utilisation d'un gestionnaire de transactions reposant sur JTA

Spring permet d'utiliser un gestionnaire de transactions implémentant l'API JTA, généralement fourni par un serveur d'applications.

La suite de cette section sera développée dans une version future de ce document

94. Spring et JMS

Chapitre 94

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

L'API JMS propose fondamentalement deux grandes fonctionnalités : produire et consommer des messages.

Dans une application Java EE, la consommation de messages est assurée par des EJB de type MDB (Message Driven Bean). Dans les autres types d'applications, la consommation de messages se fait en utilisant des MessageListener de l'API JMS.

Spring offre une abstraction pour faciliter la mise en oeuvre de l'API JMS. Il propose une API qui fournit une abstraction dans la mise en oeuvre de JMS version 1.0.2 et 1.1.

Ce chapitre contient plusieurs sections :

- Les packages de Spring JMS
- La classe JmsTemplate : le template JMS de Spring
- La réception asynchrone de messages
- L'espace de nommage jms

94.1. Les packages de Spring JMS

Les classes de Spring dédiées à la mise en oeuvre de JMS sont dans le package org.springframework.jms du fichier spring-jms.jar.

Le package org.springframework.jms.core contient les classes et interfaces de base pour utiliser JMS avec Spring. Il contient notamment un template qui est un helper prennant en charge la création et la libération des ressources et délèguant les traitements spécifiques à un callback.

Le package org.springframework.jms.connection contient des classes pour la connexion et la gestion des transactions avec JMS. Il fournit une implémentation de ConnectionFactory qui peut être utilisée dans une application standalone et une implémentation de PlatformTransactionManager pour permettre une utilisation des ressources JMS dans un contexte transactionnel.

Le package org.springframework.jms.support propose des fonctionnalités pour transformer les exceptions de type checked de JMS en une hiérarchie plus compacte d'exceptions de type unchecked. Une sous-classe de JMSException dédiée à un fournisseur est transformée en UncategorizedJmsException.

Le package org.springframework.jms.support.converter contient des utilitaires pour convertir des messages en objets. Il fournit notamment la classe MessageConverter pour convertir un message JMS en objet Java.

Le package org.springframework.jms.support.destination fournit plusieurs fonctionnalités pour gérer les destinations JMS comme par exemple un service locator pour obtenir une destination stockée dans un annuaire JNDI.

94.2. La classe JmsTemplate : le template JMS de Spring

La classe JmsTemplate est la classe de base pour faciliter l'envoi et la réception de messages JMS de façon synchrone. Cette classe se charge de gérer la création et la libération des ressources utiles pour JMS.

Le principe des templates est de fournir une classe de type helper qui facilite la mise en oeuvre de tâches communes à une fonctionnalité tout en délégant les tâches particulières à un callback qui implémente une interface particulière.

La classe JmsTemplate propose ainsi des méthodes permettant d'envoyer un message, de consommer un message de manière synchrone et de permettre un accès à la session JMS et au message producer.

Spring propose la classe JmsTemplate qui est un helper pour faciliter l'utilisation de JMS version 1.1. Sa classe fille JmsTemplate102 propose les mêmes fonctionnalités pour la version 1.0.2 de JMS.

94.2.1. L'envoi d'un message avec JmsTemplate

L'envoi d'un message se fait en utilisant la méthode send() de la classe JmsTemplate qui attend en paramètre dans ses surcharges un objet de type MessageCreator.

Une surcharge de la méthode send() attend en paramètre le nom de la destination du message et un objet de type MessageCreator() qui est une interface de callback pour créer un message.

Exemple :

```
package fr.jmdoudoux.dej.spring.jms;

import java.util.Date;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Session;

import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;

public class JmsProducer {

  private JmsTemplate jmsTemplate;

  public void envoyerMessage() {
    jmsTemplate.send(new MessageCreator() {
      public Message createMessage(final Session session) throws JMSException {
        return session.createTextMessage("Message " + new Date());
      }
    });
  }

  public JmsTemplate getJmsTemplate() {
    return jmsTemplate;
  }

  public void setJmsTemplate(final JmsTemplate jmsTemplate) {
    this.jmsTemplate = jmsTemplate;
  }
}
```

Dans l'exemple ci-dessus, une instance anonyme de la classe MessageCreator est utilisée pour créer le message : la méthode createMessage() possède en paramètre la session qui sera utilisée pour créer la nouvelle instance du message.

Pour des besoins plus pointus, il est possible d'utiliser un callback de type SessionCallback qui permet un accès à la session et au MessageProducer.

La classe JmsTemplate prend en charge la fermeture de la session JMS.

94.2.2. La réception d'un message avec JmsTemplate

Une instance de JmsTemplate peut être utilisée pour recevoir les messages de manière synchrone.

La méthode receive() attend un nouveau message sur la destination par défaut de l'instance de JmsTemplate.

Une autre surcharge de la méthode receive() attend un nouveau message sur la destination fournie en paramètre.

Exemple :

```
package fr.jmdoudoux.dej.spring.jms;

import javax.jms.Message;
import javax.jms.TextMessage;

import org.springframework.jms.core.JmsTemplate;

public class JmsConsumer {

  private JmsTemplate jmsTemplate;

  public JmsTemplate getJmsTemplate() {
    return jmsTemplate;
  }

  public void recevoirMessage() {
    Message msg = jmsTemplate.receive();
    try {
      TextMessage textMessage = (TextMessage) msg;
      if (msg != null) {
        System.out.println("Message = " + textMessage.getText());
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public void setJmsTemplate(final JmsTemplate jmsTemplate) {
    this.jmsTemplate = jmsTemplate;
  }
}
```

La propriété receiveTimeout de la classe JmsTemplate permet de préciser un timeout d'attente puisque la réception est synchrone.

Remarque : La classe JmsTemplate peut être utilisée pour envoyer des messages mais elle n'est pas recommandée pour en recevoir. Pour la réception d'un message, il est préférable d'utiliser une solution asynchrone reposant sur un MessageListenerContainer de Spring.

94.2.3. La mise en oeuvre dans une application

Cette section va écrire une petite application qui utilise les deux classes définies précédemment, Spring 3.0.5 et ActiveMQ 5.4.2.

Le context Spring doit contenir la définition des différents beans utilisés.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:context="http://www.springframework.org/schema/context"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd">

        <!-- Fabrique de connexions à ActiveMQ  -->
        <bean id="amqConnectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
               <property name="brokerURL"
                       value="tcp://localhost:61616?wireFormat.maxInactivityDuration=0" />
        </bean>

        <!-- Destination dans ActiveMQ -->
        <bean id="destination" class="org.apache.activemq.command.ActiveMQQueue">
               <constructor-arg value="local.maqueue" />
        </bean>

        <!-- Instance de JmsTemplate qui utilise ConnectionFactory et la
               Destination -->
        <bean id="producerTemplate" class="org.springframework.jms.core.JmsTemplate">
               <property name="connectionFactory" ref="amqConnectionFactory" />
               <property name="defaultDestination" ref="destination" />
        </bean>

        <bean id="consumerTemplate" class="org.springframework.jms.core.JmsTemplate">
               <property name="connectionFactory" ref="amqConnectionFactory" />
               <property name="defaultDestination" ref="destination" />
        </bean>

        <bean id="jmsProducer" class="fr.jmdoudoux.dej.spring.jms.JmsProducer">
               <property name="jmsTemplate" ref="producerTemplate" />
        </bean>

        <bean id="jmsConsumer" class="fr.jmdoudoux.dej.spring.jms.JmsConsumer">
               <property name="jmsTemplate" ref="consumerTemplate" />
        </bean>

</beans>
```

Le bean amqConnectionFactory est une instance de type ActiveMQConnectionFactory : ce bean est une fabrique de connexions à un ActiveMQ installé en local et qui utilise le port 61616.

Le bean destination est une instance de type ActiveMQQueue : ce bean encapsule une queue nommée « local.maqueue ».

Les beans producerTemplate et consumerTemplate sont des instances de type JmsTemplate.

La propriété connectionFactory est initialisée avec le bean amqConnectionFactory et la propriété defaultDestination l'est avec le bean destination.

Une instance de JmsProducer et de JmsConsumer sont déclarées avec, en dépendance, l'instance de JmsTemplate correspondante.

Pour tester les classes d'envoi et de réception, il suffit d'écrire une petite application qui charge le contexte Spring, obtient une instance de la classe et invoque l'opération voulue.

Exemple :

```
package fr.jmdoudoux.dej.spring.jms;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestProducerConsumer {
  public static void main(final String[] args) {
    ClassPathXmlApplicationContext appContext = new ClassPathXmlApplicationContext(
        new String[] {"appContext.xml" });
    System.out.println("envoi du message");
    JmsProducer jmsProducer = (JmsProducer) appContext.getBean("jmsProducer");
    jmsProducer.envoyerMessage();

    System.out.println("reception du message");
    JmsConsumer jmsConsumer = (JmsConsumer) appContext.getBean("jmsConsumer");
    jmsConsumer.recevoirMessage();
  }
}
```

Le classpath de l'application doit contenir plusieurs fichiers jar : org.springframework.jms-3.0.5.RELEASE.jar, org.springframework.beans-3.0.5.RELEASE.jar, org.springframework.context-3.0.5.RELEASE.jar, org.springframework.core-3.0.5.RELEASE.jar, com.springsource.org.apache.commons.logging-1.1.1.jar, org.springframework.asm-3.0.5.RELEASE.jar, org.springframework.expression-3.0.5.RELEASE.jar, org.springframework.transaction-3.0.5.RELEASE.jar, apache-activemq-5.4.2/activemq-all-5.4.2.jar

94.2.4. La classe CachingConnectionFactory

Initialement la classe JmsTemplate a été conçue pour être utilisée dans un conteneur Java EE qui offre une gestion des ressources JMS notamment en utilisant des pools.

Si la classe JmsTemplate est utilisée en dehors d'un conteneur Java EE ou si aucune gestion des ressources JMS n'est prise en charge par le fournisseur, il est alors intéressant d'utiliser la classe CachingConnectionFactory.

La classe CachingConnectionFactory est un wrapper qui encapsule une connexion à un MOM en proposant une reconnexion au besoin et une mise en cache de certaines ressources (connections, sessions).

Par défaut, la classe CachingConnectionFactory utilise une seule session pour créer les connections. Il est possible d'utiliser plusieurs sessions pour améliorer la montée en charge en utilisant la propriété sessionCacheSize.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:context="http://www.springframework.org/schema/context"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd">

        <!-- Fabrique de connexions à ActiveMQ  -->
        <bean id="amqConnectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
               <property name="brokerURL"
                       value="tcp://localhost:61616?wireFormat.maxInactivityDuration=0" />
        </bean>

        <!-- Cache des connexions à ActiveMQ -->
        <bean id="cachedConnectionFactory"
               class="org.springframework.jms.connection.CachingConnectionFactory">
               <property name="targetConnectionFactory" ref="amqConnectionFactory" />
               <property name="sessionCacheSize" value="3" />
        </bean>

        <!-- Destination dans ActiveMQ -->
        <bean id="destination" class="org.apache.activemq.command.ActiveMQQueue">
               <constructor-arg value="local.maqueue" />
        </bean>

        <!-- Instances de JmsTemplate qui utilise la ConnectionFactory avec
        mise en cache et la Destination -->
        <bean id="producerTemplate" class="org.springframework.jms.core.JmsTemplate">
               <property name="connectionFactory" ref="cachedConnectionFactory" />
               <property name="defaultDestination" ref="destination" />
        </bean>

        <bean id="consumerTemplate" class="org.springframework.jms.core.JmsTemplate">
               <property name="connectionFactory" ref="cachedConnectionFactory" />
               <property name="defaultDestination" ref="destination" />
        </bean>

        <bean id="jmsProducer" class="fr.jmdoudoux.dej.spring.jms.JmsProducer">
```

```
                <property name="jmsTemplate" ref="producerTemplate" />
        </bean>

        <bean id="jmsConsumer" class="fr.jmdoudoux.dej.spring.jms.JmsConsumer">
                <property name="jmsTemplate" ref="consumerTemplate" />
        </bean>

</beans>
```

La classe de test est modifiée pour permettre un arrêt et une relance d'ActiveMQ durant son exécution.

Exemple :

```
package fr.jmdoudoux.dej.spring.jms;

import java.io.IOException;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestProducerConsumer {

  public static void main(final String[] args) {
    ClassPathXmlApplicationContext appContext = new ClassPathXmlApplicationContext(
        new String[] { "appContext.xml" });

    System.out.println("envoi du message");
    JmsProducer jmsProducer = (JmsProducer) appContext.getBean("jmsProducer");
    jmsProducer.envoyerMessage();

    try {
      System.out.println("arret activeMQ");
      System.in.read();
      System.out.println("relance activeMQ");
      System.in.read();
    } catch (IOException e) {
      e.printStackTrace();
    }

    System.out.println("reception du message");
    JmsConsumer jmsConsumer = (JmsConsumer) appContext.getBean("jmsConsumer");
    jmsConsumer.recevoirMessage();
  }
}
```

Lors de l'exécution de l'application, les logs contiennent une trace de la déconnexion au broker JMS mais Spring va assurer une reconnexion automatique dès que le broker est relancé.

Résultat :

```
17 avr. 2011 17:15:21 org.springframework.context.support.AbstractApplicationContext
 prepareRefresh
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@c1b531:
startup date [Sun Apr 17 17:15:21 CEST 2011]; root of context hierarchy
17 avr. 2011 17:15:21 org.springframework.beans.factory.xml.XmlBeanDefinitionReader
 loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [appContext.xml]
17 avr. 2011 17:15:22 org.springframework.beans.factory.support.DefaultListableBeanFactory
 preInstantiateSingletons
INFO: Pre-instantiating singletons in org.springframework.beans.factory.support.
DefaultListableBeanFactory@ef5502: defining beans
[amqConnectionFactory,cachedConnectionFactory,destination,producerTemplate,consumerTemplate
,jmsProducer,jmsConsumer]; root of factory hierarchy
envoi du message
17 avr. 2011 17:15:22 org.springframework.jms.connection.SingleConnectionFactory initConnection
INFO: Established shared JMS Connection: ActiveMQConnection
{id=ID:THINKPAD_X60S-1340-1302707722546-0:1,clientId=null,started=false}
arret activeMQ
17 avr. 2011 17:15:35 org.springframework.jms.connection.SingleConnectionFactory onException
ATTENTION: Encountered a JMSException - resetting the underlying JMS Connection
```

```
javax.jms.JMSException: java.io.EOFException
   at org.apache.activemq.util.JMSExceptionSupport.create(JMSExceptionSupport.java:49)
   at org.apache.activemq.ActiveMQConnection.onAsyncException(ActiveMQConnection.java:1833)
   at org.apache.activemq.ActiveMQConnection.onException(ActiveMQConnection.java:1850)
   at org.apache.activemq.transport.TransportFilter.onException(TransportFilter.java:101)
   at org.apache.activemq.transport.ResponseCorrelator.onException(ResponseCorrelator.java:126)
   at org.apache.activemq.transport.TransportFilter.onException(TransportFilter.java:101)
   at org.apache.activemq.transport.TransportFilter.onException(TransportFilter.java:101)
   at org.apache.activemq.transport.WireFormatNegotiator.onException(WireFormatNegotiator.
   java:160)
   at org.apache.activemq.transport.InactivityMonitor.onException(InactivityMonitor.java:266)
   at org.apache.activemq.transport.TransportSupport.onException(TransportSupport.java:96)
   at org.apache.activemq.transport.tcp.TcpTransport.run(TcpTransport.java:206)
   at java.lang.Thread.run(Unknown Source)
Caused by: java.io.EOFException
   at java.io.DataInputStream.readInt(Unknown Source)
   at org.apache.activemq.openwire.OpenWireFormat.unmarshal(OpenWireFormat.java:269)
   at org.apache.activemq.transport.tcp.TcpTransport.readCommand(TcpTransport.java:227)
   at org.apache.activemq.transport.tcp.TcpTransport.doRun(TcpTransport.java:219)
   at org.apache.activemq.transport.tcp.TcpTransport.run(TcpTransport.java:202)
   ... 1 more

relance activeMQ
reception du message
17 avr. 2011 17:16:23 org.springframework.jms.connection.SingleConnectionFactory initConnection
INFO: Established shared JMS Connection: ActiveMQConnection
{id=ID:THINKPAD_X60S-1340-1302707722546-0:2,clientId=null,started=false}
Message = Message Sun Apr 17 17:15:22 CEST 2011
```

94.3. La réception asynchrone de messages

L'écriture de consommateurs de messages JMS nécessite de nombreuses lignes de code surtout si l'on souhaite que la solution soit fiable et sache monter en charge.

La classe JmsTemplate peut être utilisée pour recevoir des messages de manière synchrone, mais son utilisation dans ce cadre n'est pas recommandée car la montée en charge est très problématique.

Les EJB de type Message Bean Driven sont définis dans la version 2.0 des spécifications de EJB : ce sont des EJB stateless, supportant les transactions qui agissent en tant que listeners de messages JMS. Pour permettre une meilleure montée en charge, le serveur d'applications peut mettre en oeuvre un pool d'EJB MDB.

Cela permet au conteneur Java EE d'avoir une solution de traitement des messages asynchrones.

Les EJB de type MDB présentent plusieurs inconvénients :

- la configuration et la création des EJB est statique et ne peut pas être faite de façon dynamique
- l'écoute ne peut se faire que sur une seule destination
- l'envoi d'un message ne peut se faire que suite à la réception préalable d'un message

Spring propose plusieurs solutions pour permettre de recevoir des messages de manière asynchrone en utilisant des MessageListenerContainer.

La classe SimpleMessageListenerContainer est l'implémentation la plus simple : elle offre donc des fonctionnalités limitées. Par exemple, elle ne propose pas de support pour les transactions.

L'utilisation de la classe DefaultMessageListenerContainer a plusieurs avantages :

- de bonnes performances grâce à la mise en cache des ressources JMS (connexions, sessions, consumers)
- le nombre de consumers peut être modifié dynamiquement (méthodes setConcurrentConsumers() et setMaxConcurrentConsumers()) ce qui permet de traiter plus de messages de manière concurrente
- le support de plusieurs modes d'acquittement des messages (acknowledgement)

94.3.1. La classe DefaultMessageListenerContainer

La classe DefaultMessageListenerContainer a pour but de faciliter la réception de messages asynchrones.

Le plus simple pour utiliser la classe DefaultMessageListenerContainer est d'utiliser le namespace jms dans le fichier de configuration du contexte pour définir les différents composants utilisés. L'exemple ci-dessous utilise Apache ActiveMQ.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
      xmlns:jms="http://www.springframework.org/schema/jms"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
            http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
http://www.springframework.org/schema/jms
http://www.springframework.org/schema/jms/spring-jms-3.0.xsd">
  <!-- Fabrique de connexions à ActiveMQ -->
  <bean id="amqConnectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616?wireFormat.maxInactivityDuration=0"
    />
  </bean>

  <!-- Cache des connexions à ActiveMQ -->
  <bean id="cachedConnectionFactory"
    class="org.springframework.jms.connection.CachingConnectionFactory">
    <property name="targetConnectionFactory" ref="amqConnectionFactory" />
    <property name="sessionCacheSize" value="3" />
  </bean>

  <!-- Destination dans ActiveMQ -->
  <bean id="destination" class="org.apache.activemq.command.ActiveMQQueue">
    <constructor-arg value="local.maqueue" />
  </bean>

  <!-- Instances de JmsTemplate qui utilise la ConnectionFactory
    avec mise en cache et la Destination -->
  <bean id="producerTemplate" class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="cachedConnectionFactory" />
    <property name="defaultDestination" ref="destination" />
  </bean>

  <bean id="jmsProducer" class="fr.jmdoudoux.dej.spring.jms.JmsProducer">
    <property name="jmsTemplate" ref="producerTemplate" />
  </bean>

  <!-- Bean qui implemente l'interface MessageListener -->
  <bean id="monSimpleMessageListener"
    class="fr.jmdoudoux.dej.spring.jms.MonSimpleMessageListener">
  </bean>

  <jms:listener-container container-type="default"
    connection-factory="cachedConnectionFactory" acknowledge="auto" >

  <jms:listener id="monListener" destination="local.maqueue"
    ref="monSimpleMessageListener" method="onMessage" />
  </jms:listener-container>
</beans>
```

Un bean de type ActiveMQConnectionFactory est défini pour assurer la connexion avec le broker ActiveMQ.

Un bean qui est le message listener JMS est défini.

Un listener-container est défini en lui fournissant le cache des ConnectionFactory, le type de container default et le message listener. La propriété acknowledge permet de préciser le mode d'acquittement des messages.

Le tag <jms:listener> permet d'associer un message listener avec une destination.

Il est possible de définir plusieurs <jms:listener> qui seront gérés par le conteneur.

Exemple :

```
package fr.jmdoudoux.dej.spring.jms;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

public class MonSimpleMessageListener implements MessageListener {

  public void onMessage(final Message message) {
    try {
      System.out.println("debut reception message");
      TextMessage msg = (TextMessage) message;
      System.out.println(" Message recu : " + msg.getText());
    } catch (JMSException e) {
      e.printStackTrace();
    }
    System.out.println("fin reception message");
  }

}
```

Le message listener est un bean qui peut être implémenté de plusieurs manières :

- en implémentant l'interface javax.jms.MessageListener
- en implémentant l'interface SessionAwareMessageListener qui permet un accès à l'objet Session JMS. La gestion des exceptions est à gérer par la classe par exemple en redéfinissant la méthode handleListenerException()
- en implémentant l'interface MessageListenerAdapter qui permet de gérer les messages en masquant l'API JMS

L'utilisation du message listener container de Spring possède plusieurs avantages :

- il peut être utilisé dans différents contextes : conteneur web, conteneur Java EE, application standalone
- il peut utiliser n'importe quel MOM qui respecte les spécifications JMS. Il faut définir un bean de type connection factory et éventuellement définir quelques propriétés dans le listener-container

La classe de test envoie plusieurs messages dans la queue et attend une entrée de l'utilisateur. Cela permet au listener démarré automatiquement par le conteneur Spring de consommer les messages.

Exemple :

```
package fr.jmdoudoux.dej.spring.jms;

import java.io.IOException;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestProducerConsumerAsync {

  public static void main(final String[] args) {
    ClassPathXmlApplicationContext appContext = new ClassPathXmlApplicationContext(
        new String[] { "appContext.xml" });

    System.out.println("envoi des messages");
    JmsProducer jmsProducer = (JmsProducer) appContext.getBean("jmsProducer");
    jmsProducer.envoyerMessage();
    jmsProducer.envoyerMessage();
    jmsProducer.envoyerMessage();

    try {
      System.in.read();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

A l'exécution, les messages sont consommés par le listener.

Résultat :

```
17 avr. 2011 17:49:17 org.springframework.context.support.AbstractApplicationContext
 prepareRefresh
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@19c26f5:
startup date [Sun Apr 17 17:49:17 CEST 2011]; root of context hierarchy
17 avr. 2011 17:49:17 org.springframework.beans.factory.xml.XmlBeanDefinitionReader
 loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [appContext.xml]
17 avr. 2011 17:49:17 org.springframework.beans.factory.support.DefaultListableBeanFactory
 preInstantiateSingletons
INFO: Pre-instantiating singletons in org.springframework.beans.factory.support.
DefaultListableBeanFactory@152c4d9: defining beans
[amqConnectionFactory,cachedConnectionFactory,destination,producerTemplate,consumerTemplate,
jmsProducer,jmsConsumer,monSimpleMessageListener,monListener];
root of factory hierarchy
17 avr. 2011 17:49:17 org.springframework.context.support.
DefaultLifecycleProcessor$LifecycleGroup start
INFO: Starting beans in phase 2147483647
17 avr. 2011 17:49:17 org.springframework.jms.connection.SingleConnectionFactory initConnection
INFO: Established shared JMS Connection: ActiveMQConnection
{id=ID:THINKPAD_X60S-1390-1303228157843-0:1,clientId=null,started=false}
envoi des messages
debut reception message
 Message recu : Message Sun Apr 17 17:49:18 CEST 2011
fin reception message
debut reception message
 Message recu : Message Sun Apr 17 17:49:18 CEST 2011
fin reception message
debut reception message
 Message recu : Message Sun Apr 17 17:49:18 CEST 2011
fin reception message
```

94.3.2. L'amélioration des performances de la consommation des messages

Lors de la mise en oeuvre de JMS, l'envoi de messages est plus rapide que la réception et le traitement d'un message. Pour permettre à un système de monter en charge, il peut être utile de mettre en oeuvre plus de consumers que de producers pour des échanges au travers d'une queue.

La classe DefaultMessageListenerContainer est un conteneur pour la consommation asynchrone de messages.

Elle permet d'adapter dynamiquement le nombre de consumers utilisés en fonction du nombre de messages à traiter. La propriété concurrentConsumers permet de préciser le nombre de consumers à utiliser. La propriété maxConcurrentConsumers permet de préciser le nombre maximal de consumers utilisable. Chaque consumer possède sa propre connexion au broker.

La propriété concurrency permet de préciser le nombre de consumers et le nombre maximal de consumers en séparant les deux valeurs par un caractère tiret.

La propriété idleConsumerLimit permet de préciser le nombre de consumers inactifs qui doivent rester chargés. Ceci permet de réutiliser des consumers sans avoir à toujours en recréer.

Les propriétés concurrentConsumers et maxConcurrentConsumers peuvent être modifiées dynamiquement au runtime.

Il ne faut pas utiliser plusieurs consumers sur un topic car le message sera consommé par chaque consumer.

Pour améliorer la montée en charge, il est possible de modifier la valeur de plusieurs propriétés en fonction des besoins :

- idleTaskExecutionLimit : permet de préciser le nombre de consumers inactifs. Par défaut, la valeur est 1, ce qui implique une libération des ressources dès que le consumer ne traite plus de messages
- maxMessagesPerTask : permet de préciser le nombre maximum de messages à traiter. La valeur par défaut est -1, ce qui signifie un nombre illimité

- receiveTimeout : le timeout d'attente avant de vérifier la réception d'un message. La valeur par défaut est 1000 ms.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

      xmlns:context="http://www.springframework.org/schema/context"
      xmlns:jms="http://www.springframework.org/schema/jms"

      xmlns:p="http://www.springframework.org/schema/p"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
            http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
http://www.springframework.org/schema/jms
http://www.springframework.org/schema/jms/spring-jms-3.0.xsd">
  <!-- Fabrique de connexions à ActiveMQ -->
  <bean id="amqConnectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL"
      value="tcp://localhost:61616?wireFormat.maxInactivityDuration=0" />
  </bean>

  <!-- Cache des connexions à ActiveMQ -->
  <bean id="cachedConnectionFactory"
    class="org.springframework.jms.connection.CachingConnectionFactory">
    <property name="targetConnectionFactory" ref="amqConnectionFactory" />
    <property name="sessionCacheSize" value="3" />
  </bean>

  <!-- Destination dans ActiveMQ -->
  <bean id="destination" class="org.apache.activemq.command.ActiveMQQueue">
    <constructor-arg value="local.maqueue" />
  </bean>

  <!-- Instances de JmsTemplate qui utilise la ConnectionFactory avec mise
  en cache et la Destination -->
  <bean id="producerTemplate" class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="cachedConnectionFactory" />
    <property name="defaultDestination" ref="destination" />
  </bean>

  <bean id="jmsProducer" class="fr.jmdoudoux.dej.spring.jms.JmsProducer">
    <property name="jmsTemplate" ref="producerTemplate" />
  </bean>

  <bean id="monSimpleMessageListener"
    class="fr.jmdoudoux.dej.spring.jms.MonSimpleMessageListener">
  </bean>

  <bean id="monMessageListenerContainer"
    class="org.springframework.jms.listener.DefaultMessageListenerContainer"
    p:connectionFactory-ref="cachedConnectionFactory"
    p:destination-ref="destination"
    p:messageListener-ref="monSimpleMessageListener"
    p:concurrentConsumers="3"
    p:maxConcurrentConsumers="5"
    p:receiveTimeout="5000"
    p:idleTaskExecutionLimit="10"
    p:idleConsumerLimit="3"
  />
</beans>
```

Lors de l'exécution, les messages sont consommés en parallèle.

Résultat :

```
17 avr. 2011 18:31:31 org.springframework.context.support.AbstractApplicationContext
 prepareRefresh
```

```
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@c1b531:
startup date [Sun Apr 17 18:31:31 CEST 2011]; root of context hierarchy
17 avr. 2011 18:31:31 org.springframework.beans.factory.xml.XmlBeanDefinitionReader
 loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [appContext.xml]
17 avr. 2011 18:31:31 org.springframework.beans.factory.support.DefaultListableBeanFactory
 preInstantiateSingletons
INFO: Pre-instantiating singletons in org.springframework.beans.factory.support.
DefaultListableBeanFactory@a61164: defining beans
[amqConnectionFactory,cachedConnectionFactory,destination,producerTemplate,consumerTemplate,
jmsProducer,jmsConsumer,monSimpleMessageListener,monMessageListen
erContainer]; root of factory hierarchy
17 avr. 2011 18:31:32 org.springframework.context.support.
DefaultLifecycleProcessor$LifecycleGroup start
INFO: Starting beans in phase 2147483647
17 avr. 2011 18:31:32 org.springframework.jms.connection.SingleConnectionFactory initConnection
INFO: Established shared JMS Connection: ActiveMQConnection
{id=ID:THINKPAD_X60S-1397-1303230692234-0:1,clientId=null,started=false}
envoi des messages
debut reception message
 Message recu : Message Sun Apr 17 18:31:32 CEST 2011
debut reception message
 Message recu : Message Sun Apr 17 18:31:32 CEST 2011
debut reception message
 Message recu : Message Sun Apr 17 18:31:32 CEST 2011
fin reception message
fin reception message
fin reception message
```

Par défaut, la classe DefaultMessageListenerContainer utilise un cache pour les ressources JMS (connexions, sessions, consumers) sauf si un gestionnaire de transactions externe est utilisé. La propriété cacheLevel permet de préciser les éléments qui doivent être mis en cache (connexions uniquement, connexions et sessions ou connexions, sessions et consumers).

En précisant consumer comme valeur de la propriété cacheLevel, les connexions, les sessions et les consumers seront mis en cache.

La session est mise en cache selon son mode d'acquittement. Le consumer est mis en cache selon sa session, son sélecteur et sa destination.

La mise en cache des ressources peut permettre d'améliorer les performances de la montée en charge de l'application : par exemple ActiveMQ propose la classe org.apache.activemq.pool.PooledConnectionFactory.

Il est cependant préférable d'utiliser la classe CachingConnectionFactory de Spring qui permet en plus de mettre en cache les consumers et qui permet de se connecter à n'importe quel MOM respectant les spécifications JMS.

94.4. L'espace de nommage jms

L'espace de nommage jms permet de définir dans le contexte des objets relatifs à l'API JMS.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jms="http://www.springframework.org/schema/jms"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/jms
       http://www.springframework.org/schema/jms/spring-jms-3.0.xsd">

<!-- ... -->

</beans>
```

Le tag <listener-container> permet de configurer un container qui va gérer des listeners JMS.

Ce tag va créer un objet qui va se connecter au broker de messages et obtenir les messages pour les faire traiter par un listener.

Ce tag peut avoir un ou plusieurs tags fils <listener>.

Le tag <listener> permet de configurer un listener JMS.

Exemple :

```
<bean id="messageHandler" class="fr.jmdoudoux.dej.spring.jms.MessageHandlerImpl" />
<jms:listener-container connectionfactory="amqConnectionFactory">
  <jms:listener destination="local.maqueue" ref="messageHandler"
    method="onMessage" />
</jms:listener-container>
```

La propriété destination permet de préciser la destination (queue ou topic) qui sera écoutée.

La propriété method permet de préciser la méthode de l'objet indiqué par la propriété ref qui sera invoquée pour traiter les messages.

Le tag <jca-listener-container> permet de configurer un container JCA qui va gérer des listeners JMS.

95. Spring et JMX

Chapitre 95

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Comme pour d'autres API, Spring propose des fonctionnalités qui facilitent la mise en oeuvre de JMX sans avoir forcement à utiliser directement cette API de bas niveau.

Spring permet d'enregistrer n'importe quel bean gérer dans le contexte sous la forme d'un Model MBean simplement en utilisant une instance d'un composant de type MBeanExporter.

Le composant MBeanExporter permet d'enregistrer des model MBeans dont l'instance est un bean géré par le conteneur Spring. Cela permet d'exposer de simple POJO sans avoir à créer une interface ni respecter une convention de nommage particulière.

Ce chapitre contient plusieurs sections :

- L'enregistrement d'un bean en tant que MBean
- Le nommage des MBeans
- Les Assembler
- L'utilisation des annotations
- Le développement d'un client JMX
- Les notifications

95.1. L'enregistrement d'un bean en tant que MBean

Un objet de type org.springframework.jmx.export.MBeanExporter est responsable de l'enregistrement d'un bean géré par le conteneur Spring sous la forme d'un MBean dans un MBeanServer. Il sera alors possible d'utiliser ce bean grâce à JMX.

Exemple :

```
package fr.jmdoudoux.dej.spring.jmx;

public class MonBean {
  private String maPropriete;
  private long longueur;
  public int maValeur;

  public String getMaPropriete() {
    return maPropriete;
  }

  public void monOperation() {
    System.out.println(maPropriete);
  }

  public void setMaPropriete(final String maPropriete) {
    this.maPropriete = maPropriete;
  }
}
```

L'exemple ci-dessus va servir de base pour un MBean à exporter grâce à JMX.

95.1.1. La classe MBeanExporter

Pour exporter des beans, il suffit de déclarer dans le contexte un bean de type org.springframework.jmx.export.MBeanExporter. La propriété lazy-init doit toujours être à false pour ce type de beans.

La propriété beans de cette classe est une collection de type map qui permet de préciser l'objectName du MBean comme clé et le bean correspondant comme valeur.

Exemple :

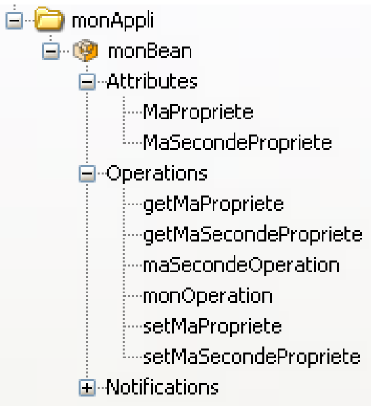
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="monBean" class="fr.jmdoudoux.dej.spring.jmx.MonBean" />

  <bean id="mbeanExporter" class="org.springframework.jmx.export.MBeanExporter"
    lazy-init="false">
    <property name="beans">
      <map>
        <entry key="monAppli:name=monBean" value-ref="monBean" />
      </map>
    </property>
  </bean>
</beans>
```

L'application n'a pas besoin d'utiliser l'API JMX pour obtenir le serveur de MBeans et enregistrer le MBean : il suffit simplement de créer le contexte Spring. Il est nécessaire de demander l'activation de JMX à la JVM en lui passant au minimum le paramètre -Dcom.sun.management.jmxremote

Exemple :

```
package fr.jmdoudoux.dej.spring.jmx;

public class MonBean {
  private String maPropriete;
  private String maSecondePropriete;
  private long longueur;
  public int maValeur;

  public String getMaPropriete() {
    return maPropriete;
  }

  public String getMaSecondePropriete() {
    return maSecondePropriete;
  }

  public void maSecondeOperation() {
    System.out.println(maPropriete);
  }

  public void monOperation() {
    System.out.println(maPropriete);
  }

  public void setMaPropriete(final String maPropriete) {
    this.maPropriete = maPropriete;
  }

  public void setMaSecondePropriete(final String maSecondePropriete) {
    this.maSecondePropriete = maSecondePropriete;
  }
}
```

Il est alors possible de voir le MBean dans un client JMX comme l'outil jconsole.



```
monAppli
  monBean
    Attributes
      MaPropriete
      MaSecondePropriete
    Operations
      getMaPropriete
      getMaSecondePropriete
      maSecondeOperation
      monOperation
      setMaPropriete
      setMaSecondePropriete
    Notifications
```

Par défaut, toutes les propriétés publiques de la classe (possédant un getter et/ou un setter) sont exposées comme propriétés du MBean et toutes les méthodes publiques sauf celles héritées de la classe Object sont exposées comme opérations du MBean.

Il est possible, pour des besoins spécifiques, de définir plusieurs MBeanExporter dans le contexte.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="monBean" class="fr.jmdoudoux.dej.spring.jmx.MonBean" />

  <bean id="monAutreBean" class="fr.jmdoudoux.dej.spring.jmx.MonAutreBean" />
  <bean id="mbeanExporter1" class="org.springframework.jmx.export.MBeanExporter"
    lazy-init="false">
    <property name="beans">
      <map>
        <entry key="monAppli:name=monBean" value-ref="monBean" />
      </map>
    </property>
  </bean>

  <bean id="mbeanExporter2" class="org.springframework.jmx.export.MBeanExporter">
    <property name="assembler" ref="assembler" />
    <property name="beans">
      <map>
        <entry key="monAppli:name=monAutreBean" value-ref="monAutreBean" />
      </map>
    </property>

  </bean>

  <bean id="assembler"
    class="org.springframework.jmx.export.assembler.MetadataMBeanInfoAssembler">
    <property name="attributeSource">

  <bean class="org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource"/>
    </property>
  </bean>
</beans>
```

La propriété booléenne autodetect de la classe MBeanExporter permet de demander l'enregistrement automatique des beans du contexte qui sont des MBeans comme définis par la spécification JMX.

La propriété registrationBehaviorName permet de préciser le comportement du MBeanExporter selon trois valeurs :

- REGISTRATION_FAIL_ON_EXISTING : si un MBean est déjà enregistré avec l'objectName, alors une exception de type InstanceAlreadyExistsException est levée. C'est le comportement par défaut
- REGISTRATION_IGNORE_EXISTING : si un MBean est déjà enregistré avec l'objectName, alors l'enregistrement du MBean est simplement ignoré sans lever d'exception
- REGISTRATION_REPLACE_EXISTING : un MBean déjà enregistré avec l'objectName est désenregistré et le nouveau MBean est enregistré dans le serveur en remplacement

95.1.2. La création d'un MBeanServer

Le composant MBeanExporter tente de trouver un MBeanServer pour y enregistrer le MBean. Si aucun MBeanServer n'existe, il est possible d'utiliser le composant MBeanServerFactoryBean pour en créer un.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="monBean" class="fr.jmdoudoux.dej.spring.jmx.MonBean" />

  <bean id="mbeanServer" class="org.springframework.jmx.support.MBeanServerFactoryBean">
    <property name="locateExistingServerIfPossible" value="true" />
  </bean>
  <bean id="mbeanExporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="monAppli:name=monBean" value-ref="monBean" />
      </map>
    </property>
  </bean>
</beans>
```

La propriété localeExistingServerIfPossible avec la valeur true permet de demander la création d'un nouveau serveur de MBeans uniquement si aucun n'est trouvé.

Si plusieurs serveurs de MBeans sont lancés, la propriété agentId permet de fournir l'identifiant du serveur qui sera retourné par la fabrique.

La propriété mbeanServer du composant MBeanExporter permet de préciser le serveur de Mbeans qui doit être utilisé pour l'enregistrement des MBeans

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="monBean" class="fr.jmdoudoux.dej.spring.jmx.MonBean" />

  <bean id="mbeanServer" class="org.springframework.jmx.support.MBeanServerFactoryBean">
    <property name="locateExistingServerIfPossible" value="true" />
  </bean>

  <bean id="mbeanExporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="monAppli:name=monBean" value-ref="monBean" />
      </map>
    </property>
    <property name="server" ref="mbeanServer" />
  </bean>
</beans>
```

95.1.3. L'accès distant au serveur de MBeans

Spring facilite la mise en oeuvre de la JSR 160 (JMX Remote API) pour permettre l'accès distant aux MBeans. Pour permettre un accès distant à un serveur de MBeans, il faut utiliser un connecteur encapsulé dans une instance obtenue en utilisant la classe ConnectorServerFactoryBean.

La classe ConnectorServerFactoryBean peut utiliser plusieurs protocoles notamment JMXMP et RMI. Par défaut, elle utilise le protocole JMXMP.

Exemple :

```
  <bean class="org.springframework.jmx.support.ConnectorServerFactoryBean"/>
```

L'url par défaut est service:jmx:jmxmp://localhost:9875

Le protocole JMXNP n'est pas supporté en standard.

Résultat :

```
Caused by: java.net.MalformedURLException: Unsupported protocol: jmxmp
      at javax.management.remote.JMXConnectorServerFactory.newJMXConnectorServer(Unknown
Source)
      at org.springframework.jmx.support.ConnectorServerFactoryBean.afterPropertiesSet
(ConnectorServerFactoryBean.java:144)
      at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.
invokeInitMethods(AbstractAutowireCapableBeanFactory.java:1477)
      at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.
initializeBean(AbstractAutowireCapableBeanFactory.java:1417)
```

Il faut ajouter le fichier jmxremote_optional.jar dans le classpath. Pour obtenir ce fichier, il faut télécharger le fichier jmx_remote-1_0_1_03-ri.zip sur le site d'Oracle.

La classe ConnectorServerFactoryBean peut aussi mettre en oeuvre d'autres protocoles : par exemple, un connecteur utilisant RMI.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="monBean" class="fr.jmdoudoux.dej.spring.jmx.MonBean" />

  <bean class="org.springframework.jmx.support.ConnectorServerFactoryBean"
    depends-on="rmiRegistry">
    <property name="objectName" value="connector:name=rmi" />
    <property name="serviceUrl"
      value="service:jmx:rmi://localhost/jndi/rmi://localhost:8099/monconnector"/>
    <property name="server" ref="mbeanServer"></property>
  </bean>

  <bean id="rmiRegistry" class="org.springframework.remoting.rmi.RmiRegistryFactoryBean">
    <property name="port" value="8099" />
  </bean>

  <bean id="mbeanServer" class="org.springframework.jmx.support.MBeanServerFactoryBean">
    <property name="locateExistingServerIfPossible" value="true" />
  </bean>
  <bean id="mbeanExporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="assembler" ref="assembler" />
    <property name="server" ref="mbeanServer" />
    <property name="beans">
      <map>
        <entry key="monAppli:name=monBean" value-ref="monBean" />
      </map>
    </property>
  </bean>

  <bean id="assembler"
    class="org.springframework.jmx.export.assembler.MetadataMBeanInfoAssembler">
    <property name="attributeSource">
      <bean
        class="org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource"
      />
    </property>
  </bean>
</beans>
```

Un bean de type RmiRegistryFactoryBean est défini dans le contexte. C'est une fabrique qui tente de trouver un registre RMI et d'en créer un si la recherche échoue. Le port utilisé par le registre est précisé grâce à la propriété port.

Un bean de type ConnectorServerFactoryBean est défini avec une dépendance sur le bean qui encapsule le registre RMI comme le recommande la documentation Spring pour garantir que le registre sera démarré avant son utilisation par le connecteur. La classe ConnectorServerFactoryBean est une fabrique d'objets de type JmxConnectorServer défini dans la JSR 160.

La classe ConnectorServerFactoryBean possède plusieurs attributs :

Attributs	Rôle
boolean daemon	Préciser si les threads démarrés pour le connecteur sont des démons
Properties environment	Fournir des propriétés utilisées pour construire l'instance du connecteur
Map<String,?> environment	Fournir des propriétés utilisées pour construire l'instance du connecteur
Object objectName	Préciser l'objectName utilisé pour enregistrer le connecteur dans le serveur de MBeans
String serviceUrl	Définir l'url du service permettant d'invoquer le connecteur
boolean threaded	Préciser si le connecteur doit être démarré dans un thread dédié

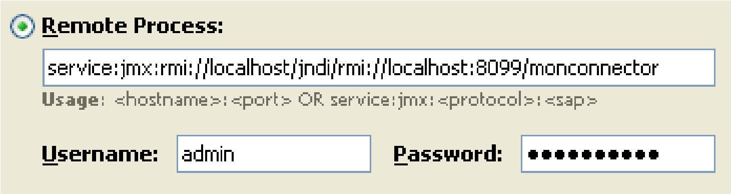
Il est important que le port précisé dans l'url et le registre RMI soient les mêmes et ne soient pas déjà utilisés sur la machine.

Pour sécuriser l'accès au serveur de MBeans, il est possible de fournir des informations à la propriété environment pour permettre l'authentification et les autorisations. L'exemple ci-dessous utilise une JVM de Sun/Oracle.

Exemple :

```
  <bean class="org.springframework.jmx.support.ConnectorServerFactoryBean"
    depends-on="rmiRegistry">
    <property name="objectName" value="connector:name=rmi" />
    <property name="serviceUrl"
      value="service:jmx:rmi://localhost/jndi/rmi://localhost:8099/monconnector"/>
    <property name="server" ref="mbeanServer"></property>
    <property name="environment">
      <map>
        <entry key="jmx.remote.x.password.file" value="C:/monapp/jmxremote.password" />
        <entry key="jmx.remote.x.access.file" value="C:/monapp/jmxremote.access"/>
      </map>
    </property>
  </bean>
```

Il est alors nécessaire de préciser l'utilisateur et son mot de passe définis dans les deux fichiers.



● Remote Process:

service:jmx:rmi://localhost/jndi/rmi://localhost:8099/monconnector

Usage: <hostname>:<port> OR service:jmx:<protocol>:<sap>

Username: admin

Password: ●●●●●●●●●●

95.1.4. Les listeners d'un Exporter

L'interface MBeanExporterListener définit les méthodes de callback d'un listener lorsqu'un MBean est enregistré ou supprimé d'un serveur de MBeans.

Méthode	Rôle
void mbeanRegistered(ObjectName objectName)	Méthode invoquée lorsqu'un MBean est correctement enregistré dans un serveur de MBeans
void mbeanUnregistered(ObjectName objectName)	Méthode invoquée lorsqu'un MBean est correctement supprimé d'un serveur de MBeans

Exemple :

```
package fr.jmdoudoux.dej.spring.jmx;

import javax.management.ObjectName;
import org.springframework.jmx.export.MBeanExporterListener;

public class MonMBeanExporterListener implements MBeanExporterListener {
  @Override
  public void mbeanRegistered(final ObjectName arg0) {
    System.out.println("Enregistrement du MBean " + arg0);
  }

  @Override
  public void mbeanUnregistered(final ObjectName arg0) {
    System.out.println("Suppression du MBean " + arg0);
  }
}
```

La propriété listeners de la classe MBeanExporter permet de définir des listeners de type MBeanExporterListener qui seront invoqués par le MBeanExporter.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="monBean" class="fr.jmdoudoux.dej.spring.jmx.MonBean" />

  <bean id="mbeanServer" class="org.springframework.jmx.support.MBeanServerFactoryBean">
    <property name="locateExistingServerIfPossible" value="true" />
  </bean>

  <bean id="monMBeanExporterListener"
    class="fr.jmdoudoux.dej.spring.jmx.MonMBeanExporterListener" />

  <bean id="mbeanExporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="monAppli:name=monBean" value-ref="monBean" />
      </map>
    </property>
    <property name="server" ref="mbeanServer" />
    <property name="listeners">
      <array>
        <ref bean="monMBeanExporterListener" />
      </array>
    </property>
  </bean>
</beans>
```

95.2. Le nommage des MBeans

Tout MBean doit avoir un nom unique dans le serveur de MBeans dans lequel il est enregistré. Le nom d'un MBean est encapsulé dans un objet de type ObjectName.

95.2.1. Les stratégies de nommage des MBeans

Un MBeanExporter utilise une stratégie de nommage pour déterminer l'ObjectName d'un MBean : il délègue la définition d'un objectName à une instance de type ObjectNamingStrategy lorsqu'il doit enregistrer une instance d'un bean dans un serveur de MBeans.

Spring propose en standard plusieurs stratégies de nommage des MBeans.

95.2.1.1. L'interface ObjectNamingStrategy

Une stratégie de nommage doit implémenter l'interface ObjectNamingStrategy.

Elle ne définit qu'une seule méthode :

Méthode	Rôle
ObjectName getObjectName(Object managedBean, String beanKey)	Obtenir l'ObjectName pour l'instance courante du bean

Cette méthode sera invoquée par le MBeanExporter pour obtenir l'ObjectName avec lequel il va enregistrer le MBean.

95.2.1.2. La classe IdentityNamingStrategy

La classe IdentityNamingStrategy implémente l'interface ObjectNamingStrategy.

Son implémentation définit un ObjectName en utilisant la valeur de hachage de l'instance du bean.

La valeur de l'ObjectName retournée est de la forme package:class=nom_de_la_classe,hashCode=valeur_de_hachage_de_l_instance

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="monBean" class="fr.jmdoudoux.dej.spring.jmx.MonBean" />

  <bean id="mbeanExporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="monBean" value-ref="monBean" />
      </map>
    </property>
    <property name="namingStrategy">
      <bean class="org.springframework.jmx.export.naming.IdentityNamingStrategy"/>
    </property>
  </bean>
</beans>
```

Avec l'exemple ci-dessus, le MBean est enregistré dans le serveur de MBeans avec l'objectName fr.jmdoudoux.dej.spring.jmx:hashCode=1a68ef9,type=MonBean

95.2.1.3. La classe KeyNamingStrategy

La classe KeyNamingStrategy implémente l'interface ObjectNamingStrategy.

Par défaut, cette implémentation donne à l'objectName la valeur fournie dans le Map comme clé de la propriété beans du MBeanExporter.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="monBean" class="fr.jmdoudoux.dej.spring.jmx.MonBean" />

  <bean id="mbeanExporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="monAppli:name=monBean" value-ref="monBean" />
      </map>
    </property>
    <property name="namingStrategy">
      <bean class="org.springframework.jmx.export.naming.KeyNamingStrategy">
      </bean>
    </property>
  </bean>
</beans>
```

Il est aussi possible de définir explicitement la valeur de la clé sous la forme de Properties en utilisant les méthodes setMappings(), setMappingLocation() ou setMappingLocations().

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="monBean" class="fr.jmdoudoux.dej.spring.jmx.MonBean" />

  <bean id="mbeanExporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="monBean" value-ref="monBean" />
      </map>
    </property>
    <property name="namingStrategy">
```

```
      <bean class="org.springframework.jmx.export.naming.KeyNamingStrategy">
        <property name="mappings">
          <props>
            <prop key="monBean">monAppli:name=monBean,type=MonBean</prop>
          </props>
        </property>
      </bean>
    </property>
  </bean>
</beans>
```

Il est possible d'utiliser la propriété mappingLocations dont la valeur précise un ou plusieurs fichiers .properties séparés par des virgules. La valeur de l'objectName sera alors recherchée dans ces fichiers qui doivent être dans le classpath en utilisant la valeur de la propriété key du bean. Si aucune clé correspondante n'est trouvée dans les fichiers properties alors c'est la valeur correspondante au bean dans la propriété mappings qui est utilisée.

95.2.1.4. La classe MetadataNamingStrategy

La classe MetadataNamingStrategy implémente l'interface ObjectNamingStrategy.

Son implémentation recherche la valeur de l'ObjectName dans les métadonnées contenues dans la classe du bean.

Cette stratégie s'utilise avec une instance de type JmxAttributeSource qui va se charger de lire les métadonnées. Cette instance peut être fournie en utilisant la surcharge du constructeur qui attend un paramètre de ce type ou utiliser la méthode setAttributeSource().

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="mbeanExporter" class="org.springframework.jmx.export.MBeanExporter"
    lazy-init="false">
    <property name="autodetect" value="true"></property>
    <property name="namingStrategy" ref="namingStrategy"></property>
    <property name="assembler" ref="assembler"></property>
  </bean>

  <bean id="attributeSource"
    class="org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource" />

  <bean id="assembler"
    class="org.springframework.jmx.export.assembler.MetadataMBeanInfoAssembler">
    <property name="attributeSource" ref="attributeSource" />
  </bean>

  <bean id="namingStrategy"
    class="org.springframework.jmx.export.naming.MetadataNamingStrategy">
    <property name="attributeSource" ref="attributeSource" />
  </bean>
</beans>
```

Dans l'exemple, ci-dessus la classe du bean à exposer comme MBean est annotée avec @ManagedResource détaillée dans les sections suivantes.

Si l'attribut objectName n'est pas précisé dans l'annotation @ManagedResource alors l'objectName utilisé sera de la forme nom_complet_du_package:type=nom_de_la_classe,name=nom_du_bean

95.2.2. L'interface SelfNaming

L'interface SelfNaming permet à un composant de déterminer dynamiquement l'objectName qui sera utilisé par l'Exporter pour exposer l'instance du bean sous la forme d'un MBean.

L'utilisation de cette interface est notamment nécessaire si plusieurs instances d'une même classe doivent être exposées comme MBean. La convention utilisée par Spring impliquerait dans ce cas des doublons dans les objectNames, ce qui est impossible.

Il faut alors que la classe implémente l'interface SelfNaming et redéfinir sa méthode getObjectName() qui renvoie une instance de type ObjectName. La valeur de l'objet retourné doit être unique pour un même serveur de MBeans.

Au moment de l'enregistrement du bean dans le serveur de MBeans, Spring invoquera cette méthode pour obtenir l'objectName sous lequel l'instance sera enregistrée.

95.3. Les Assembler

Un MBeanExporter utilise une instance de l'interface org.springframework.jmx.export.assembler.MBeanInfoAssembler qui va permettre de déterminer les fonctionnalités exposées par le MBean. Spring propose plusieurs implémentations de l'interface MBeanInfoAssembler en standard :

- SimpleReflectiveMBeanInfoAssembler : expose par introspection toutes les propriétés et les méthodes publiques
- InterfaceBaseMBeanInfoAssembler : expose les propriétés et les méthodes du MBean définies dans une interface
- MetadataMBeanInfoExporter : utilise des annotations pour déterminer les propriétés et les opérations du MBean à exposer (c'est l'implémentation par défaut)
- MethodExclusionMBeanInfoAssembler : expose toutes les propriétés et les méthodes du MBean sauf celles précisées
- MethodNameBasedMBeanInfoAssembler : expose toutes les méthodes précisées du MBean. Les méthodes de type getters/setters sont exposées comme des propriétés JMX

Pour associer un Assembler à un Exporter, il faut utiliser la propriété assembler de l'Exporter.

95.3.1. La classe MethodNameBasedMBeanInfoAssembler

L'Assembler le plus simple est implémenté par la classe MethodNameBasedMBeanInfoAssembler qui permet de définir le nom des méthodes à exposer. Les méthodes de type getter et setter exposent automatiquement la propriété correspondante.

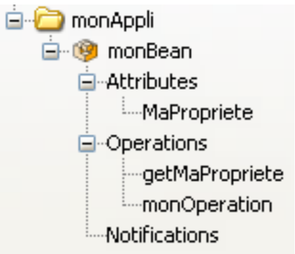
Il faut définir dans le contexte un bean de type MethodNameBasedMBeanInfoAssembler et passer à sa propriété managedMethods une collection des noms des méthodes à exposer.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="monBean" class="fr.jmdoudoux.dej.spring.jmx.MonBean" />
  <bean id="mbeanServer" class="org.springframework.jmx.support.MBeanServerFactoryBean">
    <property name="locateExistingServerIfPossible" value="true" />
  </bean>

  <bean id="mbeanExporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="assembler" ref="assembler" />
    <property name="server" ref="mbeanServer" />
    <property name="beans">
```

```
      <map>
        <entry key="monAppli:name=monBean" value-ref="monBean" />
      </map>
    </property>
  </bean>

  <bean id="assembler"
    class="org.springframework.jmx.export.assembler.MethodNameBasedMBeanInfoAssembler">
    <property name="managedMethods">
      <list>
        <value>monOperation</value>
        <value>getMaPropriete</value>
      </list>
    </property>
  </bean>
</beans>
```

Le résultat est le suivant en connectant la JVM à JConsole.



La propriété methodMapping permet pour chaque MBean de préciser les méthodes qui doivent être exposées par JMX. C'est un objet de type Properties dont la clé est l'objectName du MBean et la valeur est l'ensemble des méthodes à exposer séparées chacunes par un caractère virgule.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="monBean" class="fr.jmdoudoux.dej.spring.jmx.MonBean" />

  <bean id="mbeanServer" class="org.springframework.jmx.support.MBeanServerFactoryBean">
    <property name="locateExistingServerIfPossible" value="true" />
  </bean>

  <bean id="mbeanExporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="assembler" ref="assembler" />
    <property name="server" ref="mbeanServer" />
    <property name="beans">
      <map>
        <entry key="monAppli:name=monBean" value-ref="monBean" />
      </map>
    </property>
  </bean>

  <bean id="assembler"
    class="org.springframework.jmx.export.assembler.MethodNameBasedMBeanInfoAssembler">
    <property name="methodMappings">
      <props>
        <prop key="monAppli:name=monBean">
          getMaPropriete,monOperation
          </prop>
      </props>
    </property>
  </bean>
</beans>
```

Les noms de méthodes qui n'existent pas dans la classe sont simplement ignorés.

La propriété notificationInfoMappings permet de déclarer les notifications du MBean auxquelles il est possible de s'abonner.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="monBean" class="fr.jmdoudoux.dej.spring.jmx.MonBean" />
  <bean id="mbeanServer" class="org.springframework.jmx.support.MBeanServerFactoryBean">
    <property name="locateExistingServerIfPossible" value="true" />
  </bean>

  <bean id="mbeanExporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="assembler" ref="assembler" />
    <property name="server" ref="mbeanServer" />
    <property name="beans">
      <map>
        <entry key="monAppli:name=monBean" value-ref="monBean" />
      </map>
    </property>
  </bean>

  <bean id="assembler"
    class="org.springframework.jmx.export.assembler.MethodNameBasedMBeanInfoAssembler">
    <property name="notificationInfoMappings">
      <map>
        <entry key="monAppli:name=monBean">
          <bean class="org.springframework.jmx.export.metadata.ManagedNotification">
            <property name="name" value="Ma notification" />
            <property name="description"
              value="Modification de la valeur d'une propriété" />
            <property name="notificationTypes" value="MiseAJour" />
          </bean>
        </entry>
      </map>
    </property>
  </bean>
</beans>
```

Dans l'exemple ci-dessus, seule la notification «MiseAJour» du MBean est exposée grâce à JMX.

95.3.2. La classe MethodExclusionMBeanInfoAssembler

La classe MethodExclusionMBeanInfoAssembler permet de définir le nom des méthodes qui ne seront pas exposées. Toutes les autres méthodes publiques de la classe seront exposées automatiquement.

Il faut définir dans le contexte un bean de type MethodExclusionMBeanInfoAssembler et passer à sa propriété ignoredMethods un tableau contenant les noms des méthodes à ne pas exposer.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="monBean" class="fr.jmdoudoux.dej.spring.jmx.MonBean" />
  <bean id="mbeanServer" class="org.springframework.jmx.support.MBeanServerFactoryBean">
    <property name="locateExistingServerIfPossible" value="true" />
  </bean>
  <bean id="mbeanExporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="assembler" ref="assembler" />
    <property name="server" ref="mbeanServer" />
    <property name="beans">
      <map>
        <entry key="monAppli:name=monBean" value-ref="monBean" />
      </map>
    </property>
  </bean>

  <bean id="assembler"
    class="org.springframework.jmx.export.assembler.MethodExclusionMBeanInfoAssembler">
    <property name="ignoredMethods">
      <list>
        <value>monOperation</value>
      </list>
    </property>
  </bean>
</beans>
```

Cette définition concerne tous les MBeans qui seront exposés.

Il est aussi possible d'utiliser sa propriété ignoredMethodMappings qui est de type Properties. Elle permet de définir pour chaque MBean les méthodes à exclure. La clé est l'objecName du MBean et la valeur est une collection des noms des méthodes à ne pas exposer chacun séparé par une virgule.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="monBean" class="fr.jmdoudoux.dej.spring.jmx.MonBean" />
  <bean id="mbeanServer" class="org.springframework.jmx.support.MBeanServerFactoryBean">
    <property name="locateExistingServerIfPossible" value="true" />
  </bean>

  <bean id="mbeanExporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="assembler" ref="assembler" />
    <property name="server" ref="mbeanServer" />
    <property name="beans">
      <map>
        <entry key="monAppli:name=monBean" value-ref="monBean" />
      </map>
    </property>
  </bean>

  <bean id="assembler"
    class="org.springframework.jmx.export.assembler.MethodExclusionMBeanInfoAssembler">
    <property name="ignoredMethodMappings">
      <props>
        <prop key="monAppli:name=monBean">
          setMaPropriete,monOperation </prop>
      </props>
    </property>
  </bean>
</beans>
```

95.3.3. La classe InterfaceBasedMBeanInfoAssembler

La classe InterfaceBasedMBeanInfoAssembler permet de préciser une liste d'interfaces dont les méthodes seront exposées par les MBeans. Le mécanisme standard de JMX pour identifier un MBean et les fonctionnalités qu'il doit exposer reposent sur une interface qui doit respecter une convention de nommage précise. La classe InterfaceBasedMBeanInfoAssembler utilise une ou plusieurs interfaces, dont le nom est libre, uniquement pour déterminer les fonctionnalités à exposer.

Exemple :

```
package fr.jmdoudoux.dej.spring.jmx;

public interface IMonBean {
  public abstract String getMaPropriete();
  public abstract void monOperation();
  public abstract void setMaPropriete(final String maPropriete);
}
```

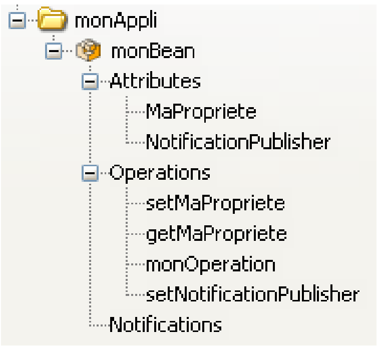
Par défaut, toutes les méthodes définies par des interfaces sont exposées.

Exemple :

```
package fr.jmdoudoux.dej.spring.jmx;

import java.util.concurrent.atomic.AtomicInteger;
import javax.management.Notification;
import org.springframework.jmx.export.notification.NotificationPublisher;
import org.springframework.jmx.export.notification.NotificationPublisherAware;

public class MonBean implements NotificationPublisherAware, IMonBean {
  private String maPropriete;
  private NotificationPublisher notificationPublisher;
  private AtomicInteger sequence = new AtomicInteger();

  @Override
  public String getMaPropriete() {
    return maPropriete;
  }

  public void maSecondeOperation() {
    System.out.println("Seconde operation");
  }

  @Override
  public void monOperation() {
    System.out.println(maPropriete);
  }

  @Override
  public void setMaPropriete(final String maPropriete) {
    this.maPropriete = maPropriete;
    notificationPublisher.sendNotification(new Notification("MiseAJour", this,
        sequence.getAndIncrement(), "Modification de la propriété avec la valeur "
           + maPropriete));
  }

  @Override
  public void setNotificationPublisher(
      final NotificationPublisher notificationPublisher) {
    this.notificationPublisher = notificationPublisher;
  }
}
```

Exemple :

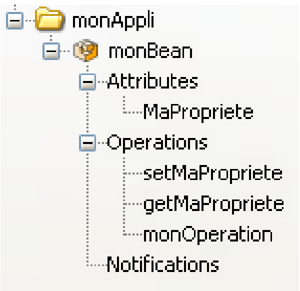
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="monBean"
    class="fr.jmdoudoux.dej.spring.jmx.MonBean" />

  <bean id="mbeanServer"
    class="org.springframework.jmx.support.MBeanServerFactoryBean">
    <property name="locateExistingServerIfPossible" value="true" />
  </bean>
  <bean id="mbeanExporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="assembler" ref="assembler" />
    <property name="server" ref="mbeanServer" />
    <property name="beans">
      <map>
        <entry key="monAppli:name=monBean" value-ref="monBean" />
      </map>
    </property>
  </bean>

  <bean id="assembler"
    class="org.springframework.jmx.export.assembler.InterfaceBasedMBeanInfoAssembler">
  </bean>
</beans>
```



Il n'est pas toujours souhaité d'exposer toutes les méthodes de toutes les interfaces. Dans l'exemple ci-dessus, c'est notamment le cas pour l'interface NotificationPublisherAware. Il est possible de préciser la ou les interfaces dont les méthodes seront exposées en utilisant la propriété managedInterfaces.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="monBean" class="fr.jmdoudoux.dej.spring.jmx.MonBean" />

  <bean id="mbeanServer" class="org.springframework.jmx.support.MBeanServerFactoryBean">
    <property name="locateExistingServerIfPossible" value="true" />
  </bean>

  <bean id="mbeanExporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="assembler" ref="assembler" />
    <property name="server" ref="mbeanServer" />
    <property name="beans">
      <map>
        <entry key="monAppli:name=monBean" value-ref="monBean" />
      </map>
    </property>
  </bean>

  <bean id="assembler"
    class="org.springframework.jmx.export.assembler.InterfaceBasedMBeanInfoAssembler">
  <property name="managedInterfaces">
    <list>
      <value>fr.jmdoudoux.dej.spring.jmx.IMonBean</value>
    </list>
  </property>
  </bean>
</beans>
```



La propriété interfaceMappings permet de définir pour chaque MBean, la ou les interfaces contenant les méthodes à exposer. Cette propriété est de type Properties : la clé est l'objectName du MBean et la valeur est une liste des interfaces séparées par des virgules.

95.3.4. La classe MetadataBasedMBeanInfoAssembler

L'assembler MetadataMBeanInfoAssembler utilise des métadonnées sur les beans pour déterminer les fonctionnalités à exposer par le MBean.

Il faut définir un bean de type MetadataMBeanInfoAssembler dans le contexte et associer sa référence à la propriété assembler de l'Exporter.

La propriété attributSource permet de préciser les métadonnées qui sont utilisées en lui passant une instance de type org.springframework.jmx.export.metadata.JmxAttributes. Spring propose en standard deux implémentations de cette interface selon la version utilisée :

- jusqu'à la version 2.5 : la classe AttributesJmxAttributeSource qui recherche des attributs
- à partir de la version 3.0 : la classe AnnotationJmxAttributeSource qui recherche des annotations

L'avantage d'utiliser les annotations est que le compilateur effectue une vérification lors de ses traitements.

95.3.4.1. L'utilisation d'attributs comme métadonnées

Jusqu'à la version 2.5 de Spring, la classe org.springframework.jmx.export.metadata.AttributesJmxAttributeSource recherchait des attributs particuliers permettant de fournir les informations nécessaires à l'exposition d'un MBean.

Spring propose plusieurs attributs :

- org.springframework.jmx.export.metadata.ManagedRessource : s'utilise sur une classe pour déclarer que ses instances seront exposées comme MBean
- org.springframework.jmx.export.metadata.ManagedAttribute : s'utilise uniquement sur des méthodes de type getter ou setter pour exposer un attribut
- org.springframework.jmx.export.metadata.ManagedOperation : s'utilise sur des méthodes qui ne sont pas de type getter ou setter pour exposer une opération
- org.springframework.jmx.export.metadata.ManagedOperationParameter : s'utilise sur des méthodes pour décrire des paramètres
- org.springframework.jmx.export.metadata.ManagedMetric
- org.springframework.jmx.export.metadata.ManagedNotification

Ces attributs possèdent des propriétés qui sont identiques à celles proposées par les annotations correspondantes décrites dans la section suivante.

Exemple :

```
package fr.jmdoudoux.dej.spring.jmx;

  /**
   * @@org.springframework.jmx.export.metadata.ManagedResource
   *   (description="Mon MBean de test avec attributs",
   *   objectName="monAppli:type=mesBeans,name=MonBeanAvecAttributs",
   *   log=true, logFile="jmx.log", currencyTimeLimit=15)
   */
  public class MonBeanAvecAttributs {

  private String maPropriete;

  /**
   * @@org.springframework.jmx.export.metadata.ManagedAttribute
   * (description="Ma propriete",
   * currencyTimeLimit=15)
   */
  public String getMaPropriete() {
    return maPropriete;
  }

  /**
   * @@org.springframework.jmx.export.metadata.ManagedOperation
   * (description="Mon operation")
   */
  public void monOperation() {
    System.out.println(maPropriete);
  }

  /**
   * @@org.springframework.jmx.export.metadata.ManagedAttribute
   * (description="Ma propriete")
   */
  public void setMaPropriete(final String maPropriete) {
    this.maPropriete = maPropriete;
  }
}
```

Java ne propose pas en standard le support de ce type d'attributs : il est nécessaire d'utiliser l'outil Apache Commons Attributes qui propose un compilateur générant du code source Java à partir des attributs qu'il va rencontrer. Ce compilateur est utilisable avec Ant et Maven.

Exemple :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>TestSpringJMX25</groupId>
  <artifactId>TestSpringJMX25</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <build>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>commons-attributes-maven-plugin</artifactId>
        <executions>
          <execution>
            <goals>
              <goal>compile</goal>
                <goal>test-compile</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.6</source>
          <target>1.6</target>
        </configuration>
      </plugin>
    </plugins>
  </build>

  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>2.5.6</version>
    </dependency>

    <dependency>
      <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>2.5.6</version>
    </dependency>

    <dependency>
      <groupId>commons-attributes</groupId>
      <artifactId>commons-attributes-api</artifactId>
      <version>2.2</version>
    </dependency>

    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-aop</artifactId>
      <version>2.5.6</version>
    </dependency>
  </dependencies>
</project>
```

Résultat :

```
C:\Documents
and Settings\jm\Documents\workspace\TestSpringJMX25>mvn clean compile
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building Unnamed - TestSpringJMX25:TestSpringJMX25:jar:0.0.1-SNAPSHOT
[INFO]    task-segment: [clean, compile]
[INFO] ------------------------------------------------------------------------
[INFO] [clean:clean {execution: default-clean}]
[INFO] Deleting directory C:\Documents and Settings\jm\Documents\workspace
\TestSpringJMX25\target
[INFO] [commons-attributes:compile {execution: default}]
[INFO] [resources:resources {execution: default-resources}]
[INFO] Copying 1 resource
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Compiling 4 source files to C:\Documents and Settings\jm\Documents\worksp
ace\TestSpringJMX25\target\classes
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 3 seconds
[INFO] Finished at: Wed Feb 06 20:46:46 CET 2013
[INFO] Final Memory: 11M/28M
[INFO]
------------------------------------------------------------------------
```

Le compilateur d'attributs génère un fichier MonBeanAvecAttributs$__attributeRepository.java dans le sous-répertoire target/generated-sources. Ce fichier sera alors compilé pour être utilisé lors de l'exécution de l'application.

Le fichier de description du contexte Spring et la classe principale qui se charge de lire et de créer ce contexte n'ont rien de particulier.

Dans l'exemple ci-dessus, un fichier de log est créé par le MBean pour journaliser ses activités.

Résultat :

```
LogMsg: Wed Feb 06 20:50:28 CET 2013 jmx.attribute.change
AttributeChangeDetected Name = MaPropriete Old value = null New value =
maValeur
```

Attention : l'utilisation des attributs comme métadonnées n'est plus utilisable à partir de Spring 3.0 : il est nécessaire d'utiliser les annotations.

95.3.4.2. L'utilisation d'annotations comme métadonnées

La classe org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource est utilisée pour rechercher des annotations particulières qui permettent de fournir les informations nécessaires à l'exposition d'un MBean.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="monBean" class="fr.jmdoudoux.dej.spring.jmx.MonBean" />

  <bean id="mbeanServer"
    class="org.springframework.jmx.support.MBeanServerFactoryBean">
    <property name="locateExistingServerIfPossible"
      value="true" />
  </bean>

  <bean id="mbeanExporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="assembler" ref="assembler" />
    <property name="server" ref="mbeanServer" />
    <property name="beans">
      <map>
        <entry key="monAppli:name=monBean" value-ref="monBean" />
      </map>
    </property>
  </bean>

  <bean id="assembler"
    class="org.springframework.jmx.export.assembler.MetadataMBeanInfoAssembler">
    <property name="attributeSource">
      <bean
        class="org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource"
      />
    </property>
  </bean>
</beans>
```

Une exception est levée au chargement du contexte si un bean doit être exposé et qu'aucune métadonnée n'est présente.

Résultat :

```
Exception in thread "main"
org.springframework.beans.factory.BeanCreationException: Error creating bean
with name 'mbeanExporter' defined in class path resource [appContext.xml]:
Invocation of init method failed; nested exception is org.springframework.jmx.export.
UnableToRegisterMBeanException:
Unable to register MBean [fr.jmdoudoux.dej.spring.jmx.MonBean@1d10a5c] with
key 'monAppli:name=monBean'; nested exception is
org.springframework.jmx.export.MBeanExportException: Could not create
ModelMBean for managed resource [fr.jmdoudoux.dej.spring.jmx.MonBean@1d10a5c]
with key 'monAppli:name=monBean'; nested exception is
org.springframework.jmx.export.metadata.InvalidMetadataException: No
ManagedResource attribute found for class: class
fr.jmdoudoux.dej.spring.jmx.MonBean
```

95.4. L'utilisation des annotations

Spring propose plusieurs annotations pour déclarer et définir des MBeans notamment :

- @ManagedResource : s'utilise sur la classe du bean pour déclarer qu'une classe est exposable sous la forme d'un MBean
- @ManagedAttribute : s'utilise sur une propriété du bean pour la définir comme exposable grâce à JMX
- @ManagedOperation : s'utilise sur une méthode du bean pour la définir comme exposable grâce à JMX

Pour exploiter ces annotations, il y a plusieurs solutions :

- définir un MBeanExporter en lui associant un MetadataMBeanInfoAssembler qui utilise un AnnotationJmxAttributeSource
- définir un AnnotationMBeanExporter
- utiliser le tag <context:mbean-export>

Un composant de type org.springframework.jmx.export.annotation.AnnotationMBeanExporter va détecter automatiquement les beans définis dans le contexte qui sont annotés avec les annotations @ManagedXXX.

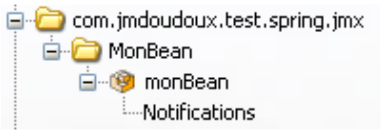
Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="monBean" class="fr.jmdoudoux.dej.spring.jmx.MonBean" />

  <bean id="mbeanServer"
    class="org.springframework.jmx.support.MBeanServerFactoryBean">
    <property name="locateExistingServerIfPossible" value="true" />
  </bean>

  <bean id="mbeanExporter"
    class="org.springframework.jmx.export.annotation.AnnotationMBeanExporter">
    <property name="server" ref="mbeanServer" />
  </bean>
</beans>
```

Par défaut, le domaine utilisé dans l'objectName des MBeans est le nom du package de la classe, le nom de l'objectname est le nom du bean utilisé dans le contexte et le type est le nom de la classe.

Résultat :

```
13 janv. 2013 17:24:41 org.springframework.context.support.AbstractApplicationContext
prepareRefresh
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@2808b3:
startup date [Sun Jan 13 17:24:41 CET 2013]; root of context hierarchy
13 janv. 2013 17:24:41 org.springframework.beans.factory.xml.XmlBeanDefinitionReader
loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [appContext.xml]
13 janv. 2013 17:24:41 org.springframework.beans.factory.support.DefaultListableBeanFactory
preInstantiateSingletons
INFO: Pre-instantiating singletons in org.springframework.beans.factory.support.
DefaultListableBeanFactory@141b571:
defining beans [monBean,mbeanServer,mbeanExporter]; root of factory hierarchy
13 janv. 2013 17:24:41 org.springframework.jmx.export.MBeanExporter afterPropertiesSet
INFO: Registering beans for JMX exposure on startup
13 janv. 2013 17:24:41 org.springframework.jmx.export.MBeanExporter autodetect
INFO: Bean with name 'monBean' has been autodetected for JMX exposure
13 janv. 2013 17:24:42 org.springframework.jmx.export.MBeanExporter registerBeanInstance
INFO: Located managed bean 'monBean': registering with JMX server as MBean
[fr.jmdoudoux.dej.spring.jmx:name=monBean,type=MonBean]
13 janv. 2013 17:24:42 org.springframework.jmx.export.MBeanExporter getMBeanInfo
ATTENTION: Bean with key 'monBean' has been registered as an MBean but has no exposed
attributes or operations
```

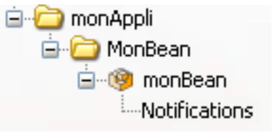


La propriété defaultDomain permet de fournir le nom du domaine utilisé dans l'object name.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="monBean" class="fr.jmdoudoux.dej.spring.jmx.MonBean" />

  <bean id="mbeanServer" class="org.springframework.jmx.support.MBeanServerFactoryBean">
    <property name="locateExistingServerIfPossible" value="true" />
  </bean>

  <bean id="mbeanExporter"
    class="org.springframework.jmx.export.annotation.AnnotationMBeanExporter">
    <property name="server" ref="mbeanServer" />
    <property name="defaultDomain" value="monAppli" />
  </bean>
</beans>
```

Le résultat est le suivant en connectant la JVM à l'outil jconsole.



L'attribut objectName de l'annotation @ManagedResource permet de préciser l'object name qui sera utilisé pour enregistrer le MBean. Cette solution n'est utilisable que pour des beans ayant la portée singleton.

Exemple :

```
package fr.jmdoudoux.dej.spring.jmx;

import org.springframework.jmx.export.annotation.ManagedResource;

@ManagedResource(objectName = "monAppli:type=mesBeans,name=MonBean")
public class MonBean {
}
```

Les classes qui doivent être exposées comme des MBeans en utilisant les annotations @ManagedXXX ne devraient pas implémenter d'interfaces ayant le suffixe MBean ou MXBean.

95.4.1. L'annotation @ManagedResource

L'annotation @ManagedResource, qui s'utilise sur une classe, permet d'indiquer que le bean doit être exposé comme un MBean.

Elle possède plusieurs attributs qui seront utilisés pour configurer le Model MBean :

Attribut	Rôle
String description	Fournir une description du MBean (optionnel)

boolean log	Préciser si certaines actions sur le beans doivent être mises dans un fichier de log (optionnel, false par défaut)
String logFile	Définir le nom du fichier de log avec son chemin (optionnel)
String objectName	Préciser l'ObjectName du MBean (optionnel)
String persistLocation	Définir le chemin d'un répertoire dans lequel les valeurs du MBean seront rendues persistantes (optionnel)
String persistName	Définir le nom du fichier dans lequel les valeurs du MBean seront rendues persistantes (optionnel)
int persistPeriod	Définir la durée en secondes applicable pour les persistPolicy NoMoreOftenThan et OnTimer (optionnel)
String persistPolicy	Préciser à quel moment les valeurs sont rendues persistantes (optionnel). Les valeurs possibles sont : OnUpdate, OnTimer, NoMoreOftenThan, Always, Never (valeur par défaut)

Son attribut par défaut est équivalent à l'attribut objectName.

L'attribut optionnel objectName permet de préciser l'object name avec lequel le MBean sera enregistré. Par défaut, l'objectName est composé de plusieurs éléments :

- le nom de domaine est le nom du package
- le type est le nom du bean défini dans le contexte
- le nom est le nom de la classe

L'attribut objectName permet de fournir explicitement l'objectName qui sera utilisé pour enregistrer le MBean.

Exemple :

```
package fr.jmdoudoux.dej.spring.jmx;

import org.springframework.jmx.export.annotation.ManagedResource;

@ManagedResource(objectName = "monAppli:type=mesBeans,name=MonBean",
  description = "Mon MBean de test")
public class MonBean {
}
```

L'utilisation de l'attribut objectName n'est possible que pour des singletons. Si plusieurs instances du bean peuvent exister dans le contexte, il faut que chaque instance ait un objectName unique. Pour cela, il faut utiliser l'interface selfNaming().

Exemple :

```
package fr.jmdoudoux.dej.spring.jmx;

import javax.management.MalformedObjectNameException;
import javax.management.ObjectName;
import org.springframework.beans.factory.BeanNameAware;
import org.springframework.jmx.export.annotation.ManagedResource;
import org.springframework.jmx.export.naming.SelfNaming;

@ManagedResource(objectName= "monAppli:type=mesBeans,name=MonBean",
  description = "Mon MBean de test")
public class MonBean implements SelfNaming, BeanNameAware {
  private String beanName;

  public String getBeanName() {
    return beanName;
  }

  @Override
  public ObjectName getObjectName() throws MalformedObjectNameException {
    final ManagedResource managedResource = this.getClass().getAnnotation(
        ManagedResource.class);
    final String name = managedResource.objectName() + ",beanName="
        + getBeanName();
    return ObjectName.getInstance(name);
  }

  @Override
  public void setBeanName(final String name) {
    beanName = name;
  }
}
```

Dans l'exemple ci-dessus, la classe du bean implémente deux interfaces :

- BeanNameAware qui va permettre d'obtenir le nom du bean tel que son instance le définit
- SelfNaming qui va permettre de définir dynamiquement l'objectName d'une instance.

L'implémentation de la méthode getObjectName() recherche par introspection l'objectName défini grâce à l'annotation @ManageResource et lui ajoute un attribut beanName dont la valeur est le nom de l'instance du bean défini dans le contexte.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="monBean1" class="fr.jmdoudoux.dej.spring.jmx.MonBean" />

  <bean id="monBean2" class="fr.jmdoudoux.dej.spring.jmx.MonBean" />

  <bean id="mbeanServer" class="org.springframework.jmx.support.MBeanServerFactoryBean">
    <property name="locateExistingServerIfPossible" value="true" />
  </bean>

  <bean id="mbeanExporter"
    class="org.springframework.jmx.export.annotation.AnnotationMBeanExporter">
    <property name="server" ref="mbeanServer" />
  </bean>
</beans>
```

95.4.2. L'annotation @ManagedAttribute

L'annotation @ManagedAttribute s'utilise sur une méthode de type getter et/ou setter pour exposer une propriété grâce au MBean. Pour exposer la propriété de manière non modifiable, il faut utiliser l'annotation @ManagedAttribute uniquement sur la méthode de type getter. Pour l'exposer de manière modifiable, il faut utiliser l'annotation sur le getter et le setter.

Elle possède plusieurs attributs qui seront utilisés par Spring pour configurer le model MBean :

Attribut	Rôle
int currencyTimeLimit	Préciser combien de temps la valeur est valide avant de devoir être rafraîchie : < 0 jamais, 0 toujours et >0 la durée en seconde
String defaultValue	Fournir la valeur par défaut (optionnel)
String description	Fournir une description de la propriété (optionnel)
int persistPeriod	Définir la durée en secondes applicable pour les persistPolicy NoMoreOftenThan et OnTimer (optionnel)
String persistPolicy	Préciser à quel moment les valeurs sont rendues persistantes (optionnel). Les valeurs possibles sont : OnUpdate, OnTimer, NoMoreOftenThan, Always, Never (valeur par défaut)

Exemple :

```
package fr.jmdoudoux.dej.spring.jmx;

import org.springframework.jmx.export.annotation.ManagedAttribute;
import org.springframework.jmx.export.annotation.ManagedResource;

@ManagedResource(objectName= "monAppli:type=mesBeans,name=MonBean",
  description = "Mon MBean de test")
public class MonBean {
  private String maPropriete;
  private String maSecondePropriete;
  public int maValeur;

  @ManagedAttribute(description="Description de ma premiere propriete.")
  public String getMaPropriete() {
    return maPropriete;
  }

  @ManagedAttribute(description="Description de ma seconde propriete.",
    currencyTimeLimit = 20, persistPolicy = "OnUpdate")
  public String getMaSecondePropriete() {
    return maSecondePropriete;
  }

  public void setMaPropriete(final String maPropriete) {
    this.maPropriete = maPropriete;
  }

  public void setMaSecondePropriete(final String maSecondePropriete) {
    this.maSecondePropriete = maSecondePropriete;
  }
}
```

95.4.3. L'annotation @ManagedOperation

L'annotation @ManagedOperation s'utilise sur une méthode pour exposer une opération grâce au MBean. Cette méthode ne doit pas être utilisée sur un getter ou un setter.

Elle possède plusieurs attributs :

Attribut	Rôle
int currencyTimeLimit	Préciser combien de temps la valeur de retour est valide avant de devoir être rafraichie : < 0 jamais, 0 toujours et >0 la durée en seconde (optionnel)
String description	Fournir une description de l'opération (optionnel)

Exemple :

```
package fr.jmdoudoux.dej.spring.jmx;

import org.springframework.jmx.export.annotation.ManagedAttribute;
import org.springframework.jmx.export.annotation.ManagedOperation;
import org.springframework.jmx.export.annotation.ManagedResource;

@ManagedResource(objectName = "monAppli:type=mesBeans,name=MonBean",
  description = "Mon MBean de test")
public class MonBean {
  private String maPropriete;
  private String maSecondePropriete;
  public int maValeur;

  @ManagedOperation(description = "Description de la seconde operation.")
  public void maSecondeOperation() {
    System.out.println(maPropriete);
  }

  @ManagedOperation(description = "Description de la premiere operation.")
  public void monOperation() {
    System.out.println(maSecondePropriete);
  }
}
```

95.4.4. Les annotations @ManagedOperationParameters et @ManagedOperationParameter

Les annotations @ManagedOperationParameters et @ManagedOperationParameter permettent de préciser des métadonnées sur les paramètres d'une opération.

L'annotation @ManagedOperationParameters est un conteneur pour une ou plusieurs annotations @ManagedOperationParameter. Elle s'utilise sur une méthode.

L'annotation @ManagedOperationParameter s'utilise sur une méthode

Elle possède plusieurs attributs qui seront utilisés pour configurer le Model MBean :

Attribut	Rôle
String name	Préciser le nom du paramètre (obligatoire)
String description	Fournir une description du paramètre (obligatoire)

Exemple :

```
package fr.jmdoudoux.dej.spring.jmx;

import org.springframework.jmx.export.annotation.ManagedOperation;
import org.springframework.jmx.export.annotation.ManagedOperationParameter;
import org.springframework.jmx.export.annotation.ManagedOperationParameters;
import org.springframework.jmx.export.annotation.ManagedResource;

@ManagedResource(objectName= "monAppli:type=mesBeans,name=MonBean",
  description = "Mon MBean de test")
public class MonBean {
  private String maPropriete;
  private String maSecondePropriete;
  public int maValeur;

  @ManagedOperation(description = "Description de la premiere operation.")
  @ManagedOperationParameters({
  @ManagedOperationParameter(name = "valeur", description = "La valeur a traiter") })
  public void monOperation(final String valeur) {
    System.out.println(maSecondePropriete);
  }

  @ManagedOperation(description = "Description de la seconde operation.")
  @ManagedOperationParameters({
    @ManagedOperationParameter(name = "min", description = "La valeur minimale"),
    @ManagedOperationParameter(name = "max", description = "La valeur maximale") })
  public void maSecondeOperation(final int min, final int max) {
    System.out.println(maPropriete);
  }
}
```

L'utilisation des annotations @ManagedOperationParameters et @ManagedOperationParameter est facultative mais dans ce cas le nom des paramètres de l'opération apparaîtront sous la forme p1, p2, ... dans le client JMX.

95.4.5. L'annotation @ManagedMetric

A partir de Spring 3.0, l'annotation @ManagedMetric permet d'exposer une propriété sous la forme d'un metric JMX.

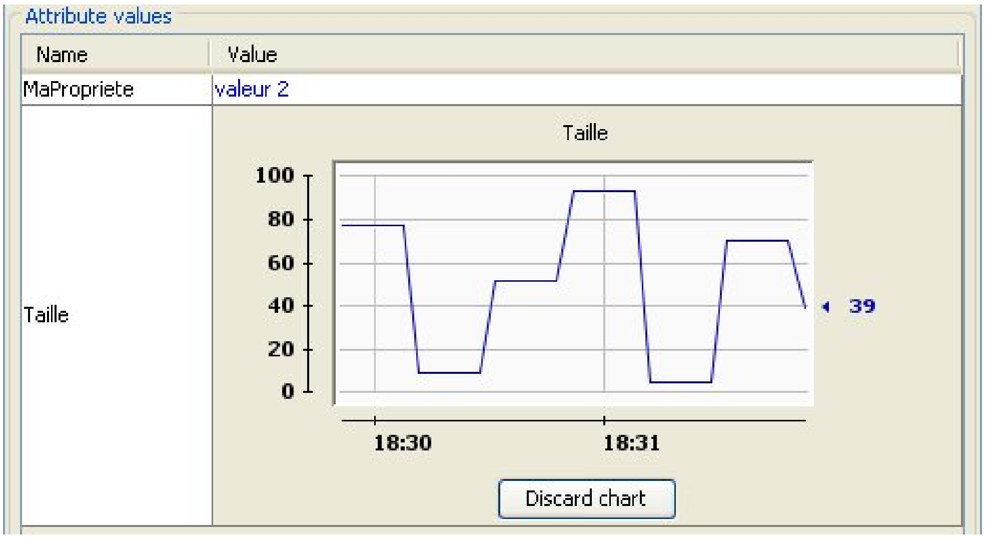
Cette annotation ne peut être utilisée que sur une méthode de type getter.

Elle possède plusieurs attributs qui seront utilisés pour configurer le Model MBean :

Attribut	Rôle
String category	Préciser la catégorie(optionnel)
int currencyTimeLimit	Préciser combien de temps la valeur de retour est valide avant de devoir être rafraîchie : < 0 jamais, 0 toujours et >0 la durée en seconde (optionnel)
String description	Fournir une description de la propriété (optionnel)
String displayName	(optionnel)
MetricType metricType	Indiquer comment la valeur évolue dans le temps (optionnel). COUNTER indique que la valeur ne fait qu'augmenter. GAUGE indique que la valeur peut varier pour augmenter ou diminuer.
int persistPeriod	Préciser la durée en secondes applicable pour les persistPolicy NoMoreOftenThan et OnTimer (optionnel)
String persistPolicy	Préciser à quel moment les valeurs sont rendues persistantes (optionnel). Les valeurs possibles sont : OnUpdate, OnTimer, NoMoreOftenThan, Always, Never (valeur par défaut)
String unit	Préciser l'unité de la valeur (optionnel)

Exemple :

```
package fr.jmdoudoux.dej.spring.jmx;

import javax.management.Notification;
import org.springframework.jmx.export.annotation.ManagedAttribute;
import org.springframework.jmx.export.annotation.ManagedMetric;
import org.springframework.jmx.export.annotation.ManagedResource;
import org.springframework.jmx.support.MetricType;

@ManagedResource(objectName = "monAppli:type=mesBeans,name=MonBean",
  description = "Mon MBean de test")
public class MonBean implements  {
  private String  maPropriete;

  @ManagedAttribute(description = "Description de ma premiere propriete.")
  public String getMaPropriete() {
    return maPropriete;
  }

  @ManagedMetric(description = "Nombre d'éléments",
    currencyTimeLimit = 20, category = "Utilisation",
    metricType = MetricType.GAUGE, displayName = "Nb éléments", unit = "éléments")
  public long getTaille() {
    return (long) (Math.random() * 100l);
  }

  public void maSecondeOperation() {
    System.out.println("Seconde operation");
  }

  @ManagedAttribute(description = "Description de ma premiere propriete.")
  public void setMaPropriete(final String maPropriete) {
    this.maPropriete = maPropriete;
  }
}
```



Name	Value
MaPropriete	valeur 2
Taille	

95.4.6. Les annotations @ManagedNotifications et @ManagedNotification

Depuis Spring 2.0, il est possible d'utiliser les annotations @ManagedNotifications et @ManagedNotification pour préciser qu'une ou plusieurs notifications peuvent être émises par le MBean.

L'annotation @ManagedNotifications est un conteneur pour une ou plusieurs annotations @ManagedNotification. Elle s'utilise sur une classe.

Elle possède un seul attribut :

Attribut	Rôle
ManagedNotification[] value	Les annotations de type @ManagedNotification associées (obligatoire)

L'annotation @ManagedNotification permet de fournir des informations sur une notification qui peut être émise grâce à JMX par le MBean.

Elle possède plusieurs attributs :

Attribut	Rôle
String name	Le nom de la notification (obligatoire)
String[] notificationTypes	(obligatoire)
String description	La description de la notification (optionnel)

Exemple :

```
package fr.jmdoudoux.dej.spring.jmx;

import java.util.concurrent.atomic.AtomicInteger;
import javax.management.Notification;
import org.springframework.jmx.export.annotation.ManagedAttribute;
import org.springframework.jmx.export.annotation.ManagedNotification;
import org.springframework.jmx.export.annotation.ManagedNotifications;
import org.springframework.jmx.export.annotation.ManagedOperation;
import org.springframework.jmx.export.annotation.ManagedResource;
import org.springframework.jmx.export.notification.NotificationPublisher;
import org.springframework.jmx.export.notification.NotificationPublisherAware;

@ManagedResource(objectName = "monAppli:type=mesBeans,name=MonBean",
  description = "Mon MBean de test")
@ManagedNotifications({
  @ManagedNotification(name = "javax.management.Notification",
    notificationTypes = { "MiseAJour" }) })
public class MonBean implements NotificationPublisherAware {
  private String                maPropriete;
  private NotificationPublisher notificationPublisher;
  private final AtomicInteger   sequence = new AtomicInteger();

  @ManagedAttribute(description = "Description de ma propriete.")
  public String getMaPropriete() {
    return maPropriete;
  }

  @ManagedOperation(description = "Description de mon operation.")
  public void monOperation() {
    System.out.println(maPropriete);
  }

  @ManagedAttribute(description = "Description de ma propriete.")
  public void setMaPropriete(final String maPropriete) {
    this.maPropriete = maPropriete;
    notificationPublisher.sendNotification(new Notification("MiseAJour", this,
        sequence.getAndIncrement(),
        "Modification de la propriété avec la valeur " + maPropriete));
  }

  @Override
  public void setNotificationPublisher(final NotificationPublisher notificationPublisher) {
    this.notificationPublisher = notificationPublisher;
  }
}
```

95.4.7. L'utilisation du tag <mbean-server>

Ce tag facilite la recherche du serveur de MBeans par défaut de la plate-forme pour Java 1.5 et supérieur, Weblogic 9 et supérieur et Websphere 5.1 et supérieur. Si aucun serveur de MBeans n'est trouvé, alors une nouvelle instance est créée et ajoutée à la plate-forme.

Par défaut, le nom du bean de type MBeanServer est mbeanServer. L'attribut id permet de définir l'identifiant du bean qui encapsule le serveur de MBeans et ainsi modifier sa valeur par défaut.

Exemple :

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context-3.0.xsd">

  <context:mbean-server id="mbeanServer" />
   ...
  </beans>
```

95.4.8. L'utilisation du tag <mbean-export>

A partir de Spring 2.5, il est possible d'utiliser le tag <context:mbean-export> pour demander au conteneur Spring d'enregistrer les beans annotés avec @ManagedXXX comme des MBeans.

Ainsi, plutôt que de définir explicitement un composant de type AnnotationMBeanExporter, si les valeurs par défaut sont suffisantes, il est possible d'utiliser le tag <context:mbean-export>.

Le tag <context:mbean-export> est équivalent à une déclaration explicite dans le contexte d'un bean de type AnnotationMBeanExporter ou d'un bean de type MBeanExporter associé à un MetadataNamingStrategy et à un AnnotationJmxAttributeSource. L'instance de type MBeanExporter créée dans le contexte a pour nom mbeanExporter.

Le tag <context:mbean-export> possède plusieurs attributs :

Attribut	Rôle
registration	Définir le comportement lors de l'enregistrement d'un MBean qui est déjà dans le serveur de MBeans. Les valeurs possibles sont failOnExisting (valeur par défaut), replaceExisting, ignoreExisting
server	Préciser le serveur de MBeans dans lequel les MBeans seront enregistrés
default-domain	Définir le nom du domaine par défaut utilisé dans l'objectName pour enregistrer les MBeans

Exemple :

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context-3.0.xsd">

  <context:mbean-export server="mbeanServer" default-domain="fr.jmdoudouxbeans"/>
   ...
  </beans>
```

La valeur replaceExisting pour l'attribut registration est particulièrement utile si les MBeans peuvent être enregistrés plusieurs fois dans le serveur de MBeans : c'est notamment le cas si le contexte Spring est recréé. Un exemple de cas concret est le redéploiement d'une application web dans un serveur Tomcat.

Les packages des beans concernés doivent être précisés dans le tag <context:component-scan>.

Exemple :

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context-3.0.xsd">

  <context:component-scan base-package=" fr.jmdoudoux.dej.spring.jmx"/>

  <context:mbean-export registration="replaceExisting"/>

</beans>
```

95.5. Le développement d'un client JMX

Spring propose deux solutions pour permettre l'accès à un serveur de MBeans distant :

- une fabrique permettant de créer de manière déclarative une instance pour se connecter au serveur de MBeans et utiliser l'API JMX
- une fabrique qui permet de créer un proxy pour accéder à un MBean

95.5.1. La connexion à un serveur de MBeans

La classe MBeanServerConnectionFactoryBean est une fabrique pour créer des instances de type MBeanServerConnection.

La propriété la plus importante est serviceUrl qui permet de préciser l'url de connexion au serveur de MBeans.

La méthode getObject() renvoie une instance de la classe javax.management.MBeanServerConnection qui encapsule une connexion sur un serveur de MBeans grâce à un connecteur de type JMXServerConnector configuré et démarré.

Il est possible de configurer un bean de type MBeanServerConnectionFactoryBean dans le contexte Spring.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="mbeanServerConnection"
    class="org.springframework.jmx.support.MBeanServerConnectionFactoryBean">
    <property name="serviceUrl"
      value="service:jmx:rmi://localhost/jndi/rmi://localhost:8099/monconnector"/>
  </bean>
</beans>
```

La propriété serviceUrl permet de préciser l'url de connexion au serveur de MBeans. L'exemple ci-dessus utilise RMI, l'exemple ci-dessous utilise le protocole JMXMP.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">
  <bean id="mbeanServerConnection"
    class="org.springframework.jmx.support.MBeanServerConnectionFactoryBean">
    <property name="serviceUrl" value="service:jmx:jmxmp://localhost:9875"/>
  </bean>
</beans>
```

Il est alors possible d'obtenir du contexte l'instance de type MBeanServerConnection et de l'utiliser pour interagir avec les MBeans enregistrés dans le serveur en utilisant l'API JMX.

Exemple :

```
package fr.jmdoudoux.dej.spring.jmx;

import java.io.IOException;
import javax.management.Attribute;
import javax.management.AttributeNotFoundException;
import javax.management.InstanceNotFoundException;
import javax.management.InvalidAttributeValueException;
import javax.management.MBeanException;
import javax.management.MBeanServerConnection;
import javax.management.MalformedObjectNameException;
import javax.management.ObjectName;
import javax.management.ReflectionException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainClient {
  public static void main(final String[] args) throws IOException {
    ApplicationContext context = new ClassPathXmlApplicationContext(
      "appContext.xml");
    MBeanServerConnection mbeanServerConnection = (MBeanServerConnection) context
       .getBean("mbeanServerConnection");
    ObjectName mbeanName;

    try {
      mbeanName = new ObjectName("monAppli:type=mesBeans,name=MonBean");
      String maPropriete = (String) mbeanServerConnection.getAttribute(
        mbeanName, "MaPropriete");
      System.out.println("maPropriete=" + maPropriete);
      mbeanServerConnection.setAttribute(mbeanName, new Attribute(
        "MaPropriete", maPropriete + " modifie"));
      mbeanServerConnection.invoke(mbeanName,
        "monOperation", new Object[] {}, new String[] {});
    } catch (MalformedObjectNameException e) {
      e.printStackTrace();
    } catch (NullPointerException e) {
      e.printStackTrace();
    } catch (AttributeNotFoundException e) {
      e.printStackTrace();
    } catch (InstanceNotFoundException e) {
      e.printStackTrace();
    } catch (MBeanException e) {
      e.printStackTrace();
    } catch (ReflectionException e) {
      e.printStackTrace();
    } catch (InvalidAttributeValueException e) {
      e.printStackTrace();
    }
    System.in.read();
  }
}
```

95.5.2. L'utilisation d'un proxy

Pour accéder à un MBean, Spring peut utiliser un proxy de type MBeanProxy sur la base d'une interface qui décrit les fonctionnalités du MBean.

Pour utiliser un proxy, le bean doit être décrit dans une interface.

Exemple :

```
package fr.jmdoudoux.dej.spring.jmx;

import org.springframework.jmx.export.annotation.ManagedAttribute;
import org.springframework.jmx.export.annotation.ManagedOperation;

public interface IMonBean {
  @ManagedAttribute(description = "Description de ma premiere propriete.")
  public abstract String getMaPropriete();

  @ManagedOperation(description = "Description de l'operation.")
  public abstract void monOperation();

  @ManagedAttribute(description = "Description de ma premiere propriete.")
  public abstract void setMaPropriete(final String maPropriete);
}
```

La classe du MBean doit implémenter l'interface.

Exemple :

```
package fr.jmdoudoux.dej.spring.jmx;

import javax.management.Notification;
import org.springframework.jmx.export.annotation.ManagedResource;
import org.springframework.jmx.export.notification.NotificationPublisher;
import org.springframework.jmx.export.notification.NotificationPublisherAware;

@ManagedResource(objectName = "monAppli:type=mesBeans,name=MonBean",
  description = "Mon MBean de test")
public class MonBean implements NotificationPublisherAware, IMonBean {
  private String maPropriete;
  private NotificationPublisher notificationPublisher;
  private int sequence;

  @Override
  public String getMaPropriete() {
    return maPropriete;
  }
  @Override
  public void monOperation() {
    System.out.println(maPropriete);
  }

  @Override
  public void setMaPropriete(final String maPropriete) {
    this.maPropriete = maPropriete;
    notificationPublisher.sendNotification(new Notification("MiseAJour", this,
        sequence++, "Modification de la propriété avec la valeur "
            + maPropriete));
  }

  @Override
  public void setNotificationPublisher(
      final NotificationPublisher notificationPublisher) {
    this.notificationPublisher = notificationPublisher;
  }
}
```

Pour le client JMX, il faut définir un bean dans le contexte Spring qui soit du type org.springframework.jmx.access.MBeanProxyFactoryBean en initialisant plusieurs propriétés :

- objectName : l'objectName sous lequel le MBean a été enregistré dans le serveur de MBeans
- proxyInterface : le nom pleinement qualifié de l'interface du MBean

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="mbeanServerConnection"
    class="org.springframework.jmx.support.MBeanServerConnectionFactoryBean">
    <property name="serviceUrl"
      value="service:jmx:rmi://localhost/jndi/rmi://localhost:8099/monconnector"/>
  </bean>

  <bean id="monBeanProxy" class="org.springframework.jmx.access.MBeanProxyFactoryBean">
    <property name="server" ref="mbeanServerConnection" />
    <property name="objectName"
      value="monAppli:type=mesBeans,name=MonBean" />
    <property name="proxyInterface"
      value="fr.jmdoudoux.dej.spring.jmx.IMonBean" />
  </bean>
</beans>
```

Le bean peut alors être injecté par le client dans l'application qui sera alors en mesure d'invoquer les méthodes du proxy.

Exemple :

```
package fr.jmdoudoux.dej.spring.jmx;

import javax.management.MBeanServerConnection;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainProxyClient {
  public static void main(final String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext(
        "appProxyContext.xml");
    MBeanServerConnection mbeanServerConnection = (MBeanServerConnection) context
        .getBean("mbeanServerConnection");
    IMonBean monBeanProxy = (IMonBean) context.getBean("monBeanProxy");
    String maPropriete = monBeanProxy.getMaPropriete();
    System.out.println(maPropriete);
    monBeanProxy.setMaPropriete(maPropriete + " modifie");
    monBeanProxy.monOperation();
  }
}
```

L'utilisation d'un proxy évite d'avoir à manipuler l'API JMX directement pour interagir avec le MBean.

Pour que le client puisse fonctionner, il faut ajouter les biliothèques suivantes au classpath : org.springframework.aop-3.0.5.RELEASE.jar, org.springframework.asm-3.0.5.RELEASE.jar, org.springframework.beans-3.0.5.RELEASE.jar, org.springframework.context-3.0.5.RELEASE.jar, org.springframework.core-3.0.5.RELEASE.jar, org.springframework.expression-3.0.5.RELEASE.jar, com.springsource.org.apache.commons.logging-1.1.1.jar et com.springsource.org.aopalliance-1.0.0.jar.

95.6. Les notifications

La spécification JMX propose la notion de notifications qui sont comme des événements émis par des MBeans et traités par des clients JMX auxquels ils sont abonnés.

95.6.1. L'émission de notifications

Spring facilite l'émission de notifications JMX au moyen de l'interface NotificationPublisher.

Cette interface ne définit qu'une méthode :

Méthode	Rôle
void sendNotification(Notification notification)	Envoyer une notification au serveur de MBeans

Pour obtenir une instance de type NotificationPublisher injectée par le contexte Spring, il faut que la classe qui encapsule le MBean implémente l'interface NotificationPublisherAware. Cette injection se fait lorsqu'une instance est enregistrée dans le serveur de MBeans par un MBeanExporter.

Chaque instance enregistrée dans un serveur de MBeans se voit injecter sa propre instance de type NotificationPublisher.

L'interface NotificationPublisherAware ne définit qu'une seule méthode :

Méthode	Rôle
void setNotificationPublisher(NotificationPublisher notificationPublisher)	Méthode de type setter pour injecter une instance de type NotificationPublisher

Exemple :

```
package fr.jmdoudoux.dej.spring.jmx;

import javax.management.Notification;
import org.springframework.jmx.export.annotation.ManagedAttribute;
import org.springframework.jmx.export.annotation.ManagedResource;
import org.springframework.jmx.export.notification.NotificationPublisher;
import org.springframework.jmx.export.notification.NotificationPublisherAware;

@ManagedResource(objectName = "monAppli:type=mesBeans,name=MonBean",
  description = "Mon MBean de test")
public class MonBean implements NotificationPublisherAware {
  private String maPropriete;
  private NotificationPublisher notificationPublisher;
  private int sequence;

  @ManagedAttribute(description = "Description de ma premiere propriete.")
  public String getMaPropriete() {
    return maPropriete;
  }

  public void setMaPropriete(final String maPropriete) {
    this.maPropriete = maPropriete;
    notificationPublisher.sendNotification(new Notification("MiseAJour", this,
        sequence++, "Modification de la propriété avec la valeur "
            + maPropriete));
  }

  @Override
  public void setNotificationPublisher(
      final NotificationPublisher notificationPublisher) {
    this.notificationPublisher = notificationPublisher;
  }
}
```

Dans l'exemple ci-dessus, chaque modification de la valeur de la propriété émet une notification.

Exemple :

```
package fr.jmdoudoux.dej.spring.jmx;

import java.io.IOException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {
  public static void main(final String[] args) throws IOException {
    ApplicationContext context = new ClassPathXmlApplicationContext(
        "appContext.xml");
    MonBean monBean = context.getBean("monBean", MonBean.class);
    System.in.read();
    monBean.setMaPropriete("valeur 1");
    monBean.setMaPropriete("valeur 2");
    try {
      Thread.sleep(2000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

95.6.2. La réception de notifications

La réception de notifications JMX se fait au moyen d'une classe qui implémente l'interface javax.management.NotificationListener.

Exemple :

```
package fr.jmdoudoux.dej.spring.jmx;

import javax.management.Notification;
import javax.management.NotificationListener;

public class MonNotificationListener implements NotificationListener {

  @Override
  public void handleNotification(final Notification notification,
      final Object handback) {
    if (notification.getType().equals("MiseAJour")) {
      System.out.println(notification.getSource() + " : "
          + notification.getType() + " - " + notification.getSequenceNumber());
    }
  }
}
```

Il suffit de récupérer l'instance de type MBeanServerConnection du contexte de Spring et d'utiliser l'API JMX pour enregistrer une nouvelle instance du listener en utilisant la méthode addNotificationListener().

Exemple :

```
package fr.jmdoudoux.dej.spring.jmx;

import java.io.IOException;
import javax.management.InstanceNotFoundException;
import javax.management.MBeanServerConnection;
import javax.management.MalformedObjectNameException;
import javax.management.ObjectName;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainClient {
  public static void main(final String[] args) throws IOException {
    ApplicationContext context = new ClassPathXmlApplicationContext(
        "appContext.xml");

    MBeanServerConnection mbeanServerConnection = (MBeanServerConnection) context
        .getBean("mbeanServerConnection");
    ObjectName mbeanName;
    try {
      mbeanName = new ObjectName("monAppli:type=mesBeans,name=MonBean");
      mbeanServerConnection.addNotificationListener(mbeanName,
          new MonNotificationListener(), null, null);
    } catch (MalformedObjectNameException e) {
      e.printStackTrace();
    } catch (NullPointerException e) {
      e.printStackTrace();
    } catch (InstanceNotFoundException e) {
      e.printStackTrace();
    }
    System.in.read();
  }
}
```

Il est possible d'enregistrer des NotificationListener dans un MBeanExporter en utilisant sa propriété notificationListenerMappings. Cette propriété attend une Map dont les clés sont des objectNames et les valeurs associées, les noms pleinement qualifiés des classes de type NotificationListener.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">
  <bean id="monBean" class="fr.jmdoudoux.dej.spring.jmx.MonBean" />
  <bean id="mbeanServer"
    class="org.springframework.jmx.support.MBeanServerFactoryBean">
    <property name="locateExistingServerIfPossible" value="true" />
  </bean>
  <bean id="monNotificationListener"
    class="fr.jmdoudoux.dej.spring.jmx.MonNotificationListener" />
  <bean id="mbeanExporter"
    class="org.springframework.jmx.export.annotation.AnnotationMBeanExporter">
    <property name="server" ref="mbeanServer" />
    <property name="notificationListenerMappings">
      <map>
        <entry key="monAppli:type=mesBeans,name=MonBean"
          value-ref="monNotificationListener" />
      </map>
    </property>
  </bean>
</beans>
```

Partie 14 : Les outils pour le développement

Le développement dans n'importe quel langage nécessite un ou plusieurs outils. D'ailleurs la multitude des technologies mises en oeuvre dans les projets récents nécessite l'usage de nombreux outils.

Ce chapitre propose un recensement non exhaustif des outils utilisables pour le développement d'applications Java et une présentation détaillée de certains d'entre eux.

Le JDK fournit un ensemble d'outils pour réaliser les développements mais leurs fonctionnalités se veulent volontairement limitées au strict miminum.

Enfin, le monde open source propose de nombreux outils très utiles.

Cette partie contient les chapitres suivants :

- Les outils du J.D.K. : indique comment utiliser les outils fournis avec le JDK
- JavaDoc : explore l'outil de documentation fourni avec le JDK
- JShell : détaille l'utilisation de l'outil JShell
- Les outils libres et commerciaux : tente une énumération non exhaustive des outils libres et commerciaux pour utiliser java
- Ant : propose une présentation et la mise en oeuvre de cet outil d'automatisation de la construction d'applications
- Maven : présente l'outil open source Maven qui facilite et automatise certaines tâches de la gestion d'un projet
- Tomcat : Détaille la mise en oeuvre du conteneur web Tomcat

- Des outils open source pour faciliter le développement : présentation de quelques outils de la communauté open source permettant de simplifier le travail des développeurs.

96. Les outils du J.D.K.

Chapitre 96

Niveau : 🟩🟩⬜⬜⬜⬜ Elémentaire

Le JDK de Sun/Oracle fournit un ensemble d'outils qui permettent de réaliser des applications. Ces outils sont peu ergonomiques car ils s'utilisent en ligne de commandes mais, en contrepartie, ils peuvent toujours être utilisés.

Ce chapitre contient plusieurs sections :

- ♦ Le compilateur javac
- ♦ L'interpréteur java/javaw
- ♦ L'outil jar
- ♦ L'outil appletviewer pour tester des applets
- ♦ L'outil javadoc pour générer la documentation technique
- ♦ L'outil Java Check Update pour mettre à jour Java
- ♦ La base de données Java DB
- ♦ L'outil JConsole

96.1. Le compilateur javac

Cet outil est le compilateur : il utilise un fichier source Java fourni en paramètre pour créer un ou plusieurs fichiers contenant le bytecode Java correspondant. Pour chaque fichier source, un fichier portant le même nom avec l'extension .class est créé si la compilation se déroule bien. Il est possible qu'un ou plusieurs autres fichiers .class soient générés lors de la compilation de la classe si celle-ci contient des classes internes. Dans ce cas, le nom du fichier des classes internes est de la forme classe$classe_interne.class. Un fichier .class supplémentaire est créé pour chaque classe interne.

96.1.1. La syntaxe de javac

La syntaxe est la suivante :

```
javac [options] [fichiers] [@fichiers]
```

Cet outil est disponible depuis le JDK 1.0

La commande attend au moins un nom de fichier contenant du code source Java. Il peut y en avoir plusieurs, en les précisant un par un, séparés par des caractères espace ou en utilisant les jokers du système d'exploitation. Tous les fichiers indiqués doivent obligatoirement posséder l'extension .java qui doit être précisée sur la ligne de commandes.

Exemple : pour compiler le fichier MaClasse.

```
javac MaClasse.java
```

Exemple : pour compiler tous les fichiers sources du répertoire javac *.java

Le nom du fichier doit correspondre au nom de la classe contenue dans le fichier source. Il est obligatoire de respecter la casse du nom de la classe même sur des systèmes qui ne sont pas sensibles à la casse comme Windows.

Depuis le JDK 1.2, il est aussi possible de fournir un ou plusieurs fichiers qui contiennent une liste des fichiers à compiler. Chacun des fichiers à compiler doit être sur une ligne distincte. Sur la ligne de commandes, les fichiers qui contiennent une liste doivent être précédés d'un caractère @

Exemple :

javac @liste

Contenu du fichier liste :
test1.java
test2.java

96.1.2. Les options de javac

Les principales options sont :

Option	Rôle
-classpath path	permet de préciser le chemin de recherche des classes nécessaires à la compilation
-d répertoire	les fichiers sont créés dans le répertoire indiqué. Par défaut, les fichiers sont créés dans le même répertoire que leurs sources.
-g	génère des informations de débogage
-nowarn	le compilateur n'émet aucun message d'avertissement
-O	le compilateur procède à quelques optimisations. La taille du fichier généré peut augmenter. Il ne faut pas utiliser cette option avec l'option -g
-verbose	le compilateur affiche des informations sur les fichiers sources traités et les classes chargées
-deprecation	donne des informations sur les méthodes dépréciées qui sont utilisées

96.1.3. Les principales erreurs de compilation

';' expected

Chaque instruction doit se terminer par un caractère ;. Cette erreur indique généralement qu'un caractère ; est manquant.

Exemple de code :

```
public class Test {
  int i
}
```

Résultat de la compilation :

```
javac Test.java
Test.java:3:
  ';' expected
  int i
       ^
1 error
```

'(' or '[' expected

Exemple de code :

```
public class Test {
  String[] chaines = new String {"aa","bb","cc"};
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:3:
  '(' or '[' expected
  String[] chaines = new String {"aa","bb","cc"};
                                ^
1 error
```

Cannot resolve symbol

Exemple de code :

```
import java.util;
public class Test {
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:1:
  cannot resolve symbol
symbol  : class util
location:
  package java
import java.util;
            ^
1 error
```

'else' without 'if'

Exemple de code :

```
public class Test {
  int i = 1;
  public void traitement() {
    if (i==0) {
      i =1;
      else i = 0;
    }
  }
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:8:
  'else' without 'if'
    else i = 0;
    ^
Test.java:11:
  '}' expected
}
 ^
2 errors
```

'}' expected

Exemple de code :

```
public class Test {
  int i = 1;
  public void traitement() {
    if (i==0) {
      i =1;
  }
}
```

Résultat de compilation :

```
C:\tmp>javac Test.java
Test.java:9: '}' expected
}
 ^
1 error
```

Variable is already defined

Une variable est définie deux fois dans la même portée.

Exemple de code :

```
public class Test {
  int i = 1;
  int i = 0;
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:4: i is already defined in Test
  int i = 0;
      ^
1 error
```

Class is already defined in a single-type import

Exemple de code :

```
import java.util.List;
import java.awt.List;
public class Test {
  List liste = null;
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:2: java.util.List is already defined in a single-type
  import
import java.awt.List;
^
1 error
```

Reference to Class is ambiguous

Exemple de code :

```
import java.util.*;
import java.awt.*;
public
  class Test {
  List liste = null;
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:5:
  reference to List is ambiguous, both class java.awt.List in java.aw
t and
  class java.util.List in java.util match
  List liste = null;
  ^
1 error
```

Variable might not have been initialized

Exemple de code :

```
public class Test {
  public void traitement() {
    String chaine;
    System.out.println(chaine);
  }
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:6:
  variable chaine might not have been initialized
    System.out.println(chaine);
                       ^
1 error
```

Class is abstract; cannot be instantiated

Il n'est pas possible d'instancier une classe abstraite.

Exemple :

```
public abstract class Test {
  public static void main(String[] argv) {
    Test test = new Test();
  }
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:5:
  Test is abstract; cannot be instantiated
    Test test = new Test();
                ^
1 error
```

Non-static variable cannot be referenced from a static context

Exemple :

```
public class Test {
  int i = 0;
  public static void main(String[] argv) {
    System.out.println(i);
  }
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:6:
  non-static variable i cannot be referenced from a
  static context
    System.out.println(i);
                       ^
1 error
```

Cannot resolve symbol

Exemple :

```
public class Test {
  public static void main(String[] argv) {
    System.out.println(i);
  }
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:5:
  cannot resolve symbol
symbol  : variable i
location:
  class Test
    System.out.println(i);
                       ^
1 error
```

Class Classe is public, should be declared in a file named Classe.java

Exemple :

```
public class Test {
}
public class TestBis {
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:6:
  class TestBis is public, should be declared in a
  file named TestBis
.java
public class TestBis {
       ^
1 error
```

'class' or 'interface' expected

Exemple :

```
public Test {
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:2:
  'class' or 'interface' expected
public Test {
       ^
1 error
```

Méthode is already defined in Test

Exemple :

```
public class Test {
  public int additionner(int a, int b) {
    return a+b;
  }
  public float additionner(int a, int b) {
    return (float)a+b;
  }
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:8: additionner(int,int) is
  already defined in Test
  public float additionner(int a, int b) {
               ^
1 error
```

Invalid method declaration; return type required

Exemple :

```
public class Test {
  public additionner(int a, int b) {
    return a+b;
  }
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:4:
  invalid method declaration; return type required
  public additionner(int a, int b) {
         ^
1 error
```

Possible loss of precision

Exemple :

```
public class Test {
  public byte traitement(int a) {
    byte b = a;
    return b;
  }
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:5:
  possible loss of precision
found   : int
required:
  byte
    byte b = a;
             ^
1 error
```

Missing method body, or declare abstract

Exemple :

```
public class Test {
  public int additionner(int a, int b); {
    return a+b;
  }
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:4:
  missing method body, or declare abstract
  public int additionner(int a, int b); {
             ^
Test.java:5:
  return outside method
    return a+b;
    ^
2 errors
```

Operator || cannot be applied to int,int

Exemple :

```
public class Test {
  int i;
  public void Test(int a, int b, int c , int d) {
    if ( a = b || c = d) {
      System.out.println("test");
    }
  }
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:6:
  incompatible types
found   : int
required: boolean
    if ( a = b || c = d) {
         ^
Test.java:6:
  operator || cannot be applied to int,int
    if ( a = b || c = d) {
               ^
2 errors
```

Incompatible types

Exemple :

```
public class Test {
  int i;
  public void Test(int a, int b) {
    if ( a = b ) {
      System.out.println("test");
    }
  }
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:6:
  incompatible types
found   : int
```

```
required: boolean
    if ( a = b ) {
         ^
1 error
```

Missing return statement

Exemple :

```
public class Test {
  int a;
  public int traitement(int b) {
    a = b;
  }
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:7:
  missing return statement
  }
  ^
1 error
```

Modifier private not allowed here

Exemple :

```
private class Test {
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:2:
  modifier private not allowed here
private class Test {
        ^
1 error
```

96.2. L'interpréteur java/javaw

Ces deux outils sont les interpréteurs de bytecode : ils lancent le JRE, chargent les classes nécessaires et exécutent la méthode main de la classe passée en paramètre.

java ouvre une console pour recevoir les messages de l'application alors que javaw n'en ouvre pas.

96.2.1. La syntaxe de l'outil java

java [options] classe [argument ...]
java [options] -jar fichier.jar [argument ...]
javaw [options] classe [argument ...]
javaw [options] -jar fichier.jar [argument ...]

Le paramètre classe doit être un fichier .class dont il ne faut pas préciser l'extension. La classe contenue dans ce fichier doit obligatoirement contenir une méthode main(). La casse du nom du fichier doit être respectée.

Cet outil est disponible depuis la version 1.0 du JDK.

Exemple:

java MaClasse

Il est possible de fournir des arguments à l'application.

96.2.2. Les options de l'outil java

Les principales options sont :

Option	Rôle
-jar archive	Permet d'exécuter une application contenue dans un fichier .jar. Dans ce cas, le fichier manifest de l'archive doit préciser la classe qui contient la méthode main(). Depuis le JDK 1.2
-Dpropriete=valeur	Permet de définir une propriété système sous la forme propriete=valeur. propriété représente le nom de la propriété et valeur représente sa valeur. Il ne doit pas y avoir d'espace entre l'option et la définition ni même dans la définition. Il faut utiliser autant d'option -D que de propriétés à définir. Depuis le JDK 1.1
-classpath chemins ou -cp chemins	permet d'indiquer les chemins de recherche des classes nécessaires à l'exécution. Chaque répertoire doit être séparé avec un point virgule. Cette option annule l'utilisation de la variable système CLASSPATH
-classic	Permet de préciser que c'est la machine virtuelle classique qui doit être utilisée. Par défaut, c'est la machine virtuelle utilisant la technologie HotSpot qui est utilisée. Depuis le JDK 1.3
-version	Affiche des informations sur l'interpréteur
-verbose ou -v	Permet d'afficher chaque classe chargée par l'interpréteur
-X	Permet de préciser des paramètres particuliers à l'interpréteur. Depuis le JDK 1.2

96.3. L'outil jar

JAR est le diminutif de Java ARchive. C'est un format de fichier qui permet de regrouper des fichiers contenant du bytecode Java (fichier .class) ou des données utilisées en tant que ressources (images, son, ...). Ce format est compatible avec le format ZIP : les fichiers contenus dans un jar sont compressés de façon indépendante du système d'exploitation.

Les jar sont utilisables depuis la version 1.1 du JDK.

96.3.1. L'intérêt du format jar

Son utilisation est particulièrement pertinente avec les applets, les beans et même les applications. En fait, le format jar est le format de diffusion des composants Java.

Les fichiers jar sont compressés par défaut ce qui est particulièrement intéressant quelque soit leurs utilisations.

Pour une applet, le browser n'effectue plus qu'une requête pour obtenir l'applet et ses ressources au lieu de plusieurs pour obtenir tous les fichiers nécessaires (fichiers .class, images, sons ...).

Un jar peut être signé ce qui permet d'assouplir et d'élargir le modèle de sécurité, notamment celui des applets qui ont des droits restreints par défaut.

Les beans doivent obligatoirement être diffusés sous ce format.

Les applications sous forme de jar peuvent être exécutées automatiquement.

Une archive jar contient un fichier manifest qui permet de préciser le contenu du jar et de fournir des informations sur celui-ci (classe principale, type de composants, signature ...).

96.3.2. La syntaxe de l'outil jar

Le JDK fourni l'outil jar pour créer des archives jar. C'est un outil utilisable avec la ligne de commandes comme tous les outils du JDK.

La syntaxe est la suivante :

jar [option [jar [manifest [fichier

Cet outil est disponible depuis la version 1.1 du JDK.

Les options sont :

Option	Rôle
c	Création d'une nouvelle archive
t	Affiche le contenu de l'archive sur la sortie standard
x	Extraction du contenu de l'archive
u	Mise à jour ou ajout de fichiers à l'archive : à partir de Java 1.2
f	Indique que le nom du fichier contenant l'archive est fourni en paramètre
m	Indique que le fichier manifest est fourni en paramètre
v	Mode verbeux pour avoir des informations complémentaires
0 (zéro)	Empêche la compression à la création
M	Empêche la création automatique du fichier manifest

Pour fournir des options à l'outil jar, il faut les saisir sans '-' et les accoler les unes aux autres. Leur ordre n'a pas d'importance.

Une restriction importante concerne l'utilisation simultanée du paramètre 'm' et 'f' qui nécessitent respectivement le nom du fichier manifest et le nom du fichier archive en paramètre de la commande. L'ordre de ces deux paramètres doit être identique à l'ordre des paramètres 'm' et 'f' sinon une exception est levée lors de l'exécution de la commande

Exemple :

```
C:\jm\Java\xagbuilder>jar cmf test.jar manif.mf *.class
java.io.IOException: invalid header field
        at java.util.jar.Attributes.read(Attributes.java:354)
        at java.util.jar.Manifest.read(Manifest.java:161)
        at java.util.jar.Manifest.<init>(Manifest.java:56)
        at sun.tools.jar.Main.run(Main.java:125)
        at sun.tools.jar.Main.main(Main.java:904)
```

Voici quelques exemples de l'utilisation courante de l'outil jar :

- Création d'un jar avec un fichier manifest créé automatiquement contenant tout les fichiers .class du répertoire courant

  jar cf test.jar *.class

- Lister le contenu d'un jar

  jar tf test.jar

- Extraire le contenu d'une archive

  jar xf test.jar

96.3.3. La création d'une archive jar

L'option 'c' permet de créer une archive jar. Par défaut, le fichier créé est envoyé sur la sortie standard sauf si l'option 'f' est utilisée. Elle précise que le nom du fichier est fourni en paramètre. Par convention, ce fichier a pour extension .jar.

Si le fichier manifest n'est pas fourni, un fichier est créé par défaut dans l'archive jar dans le répertoire META-INF sous le nom MANIFEST.MF

Exemple (code Java 1.1) : Création d'un jar avec un fichier manifest créé automatiquement contenant tous les fichiers .class du répertoire courant

```
jar cf test.jar *.class
```

Il est possible d'ajouter des fichiers contenus dans des sous-répertoires du répertoire courant : dans ce cas, l'arborescence des fichiers est conservée dans l'archive.

Exemple (code Java 1.1) : Création d'un jar avec un fichier manifest fourni contenant tous les fichiers .class du répertoire courant et tous les fichiers du répertoire images

```
jar cfm test.jar manifest.mf .class images
```

Exemple (code Java 1.1) : Création d'un jar avec un fichier manifest fourni contenant tous les fichiers .class du répertoire courant et tous les fichiers .gif du répertoire images

```
jar cfm test.jar manifest.mf *.class images/*.gif
```

96.3.4. Lister le contenu d'une archive jar

L'option 't' permet de donner le contenu d'une archive jar.

Exemple (code Java 1.1) : lister le contenu d'une archive jar

```
jar tf test.jar
```

Le séparateur des chemins des fichiers est toujours un slash quelle que soit la plate-forme car le format jar est indépendant de toute plate-forme. Les chemins sont toujours donnés dans un format relatif et non pas absolu : le chemin est donné par rapport au répertoire courant. Il faut en tenir compte lors d'une extraction.

Exemple :

```
C:\jm\bin\test\java>jar tvf test.jar
  2156 Thu Mar 30 18:10:34 CEST 2000 META-INF/MANIFEST.MF
   678 Thu Mar 23 12:30:00 CET 2000   BDD_confirm$1.class
   678 Thu Mar 23 12:30:00 CET 2000   BDD_confirm$2.class
  4635 Thu Mar 23 12:30:00 CET 2000   BDD_confirm.class
   658 Thu Mar 23 13:18:00 CET 2000   BDD_demande$1.class
   657 Thu Mar 23 13:18:00 CET 2000   BDD_demande$2.class
   662 Thu Mar 23 13:18:00 CET 2000   BDD_demande$3.class
   658 Thu Mar 23 13:18:00 CET 2000   BDD_demande$4.class
  5238 Thu Mar 23 13:18:00 CET 2000   BDD_demande.class
   649 Thu Mar 23 12:31:28 CET 2000   BDD_resultat$1.class
  4138 Thu Mar 23 12:31:28 CET 2000   BDD_resultat.class
   533 Thu Mar 23 13:38:28 CET 2000   Frame1$1.class
   569 Thu Mar 23 13:38:28 CET 2000   Frame1$2.class
   569 Thu Mar 23 13:38:28 CET 2000   Frame1$3.class
  2150 Thu Mar 23 13:38:28 CET 2000   Frame1.class
   919 Thu Mar 23 12:29:56 CET 2000   Test2.class
```

96.3.5. L'extraction du contenu d'une archive jar

L'option 'x' permet d'extraire par défaut tous les fichiers contenus dans l'archive dans le répertoire courant en respectant l'arborescence de l'archive. Pour n'extraire que certains fichiers de l'archive, il suffit de les préciser en tant que paramètres de l'outil jar en les séparant par un espace. Pour une extraction totale ou partielle de l'archive, les fichiers sont extraits en conservant la hiérarchie des répertoires qui les contiennent.

Exemple (code Java 1.1) : Extraire le contenu d'une archive

```
jar xf test.jar
```

Exemple (code Java 1.1) : Extraire les fichiers test1.class et test2.class d'une archive

```
jar xf test.jar test1.class test2.class
```

Attention : lors de l'extraction, l'outil jar écrase tous les fichiers existants sans demander de confirmation.

96.3.6. L'utilisation des archives jar

Dans une page HTML, pour utiliser une applet fournie sous forme de jar, il faut utiliser l'option archive du tag applet. Cette option attend en paramètre le fichier jar et son chemin relatif au répertoire contenant le fichier HTML.

Exemple : le fichier HTML et le fichier MonApplet.jar sont dans le même répertoire

```
<applet code=>em<MonApplet.class>/em<
        archive=">em<MonApplet.jar>/em<"
        width=>em<300>/em< height=>em<200>/em<>
</applet>
```

Avec Java 1.1, l'exécution d'une application sous forme de jar se fait grâce au JRE. Il faut fournir dans ce cas le nom du fichier jar et le nom de la classe principale.

Exemple :

```
jre -cp MonApplication.jar ClassePrincipale
```

Avec Java 1.2, l'exécution d'une application sous forme de jar impose de définir la classe principale (celle qui contient la méthode main) dans l'option Main-Class du fichier manifest. A cette condition l'option -jar de la commande java permet d'exécuter l'application.

Exemple (Java 1.2) :

```
java -jar MonApplication.jar
```

96.3.7. Le fichier manifest

Le fichier manifest contient de nombreuses informations sur l'archive et son contenu. Ce fichier est le support de toutes les fonctionnalités particulières qui peuvent être mises en oeuvre avec une archive jar.

Dans une archive jar, il ne peut y avoir qu'un seul fichier manifest nommé MANIFEST dans le répertoire META-INF de l'archive.

Le format de ce fichier est de la forme clé/valeur. Il faut mettre un ':' et un espace entre la clé et la valeur.

Exemple :

```
C:\jm\bin\test\java>jar xf test.jar META-INF/MANIFEST.MF
```

Cela crée un répertoire META-INF dans le répertoire courant contenant le fichier MANIFEST.MF :

Exemple :

```
Manifest-Version: 1.0
Name: BDD_confirm$1.class
Digest-Algorithms: SHA MD5
SHA-Digest: ntbIs5E5YNilE4mf570JoIF9akU=
MD5-Digest: R3zH0+m9lTFq+BlQvfQdHA==
Name: BDD_confirm$2.class
Digest-Algorithms: SHA MD5
SHA-Digest: 3QEF8/zmiTAP7MHFPU5wZyg9uxc=
MD5-Digest: swBXXptrLLwPMw/bpt6F0Q==
Name: BDD_confirm.class
Digest-Algorithms: SHA MD5
SHA-Digest: pZBT/o8YeDG4q+XrHRgrB08k4HY=
MD5-Digest: VFvY4sGRfjV1ciM9C+QIdg==
```

Dans le fichier manifest créé automatiquement avec le JDK 1.1, chaque fichier possède au moins une entrée de type 'Name' et des informations la concernant.

Entre les données de deux fichiers, il y a une ligne blanche.

Dans le fichier manifest créé automatiquement avec le JDK 1.2, il n'y a plus d'entrée pour chaque fichier.

Exemple :

```
Manifest-Version: 1.0
Created-By: 1.3.0 (Sun Microsystems Inc.)
```

Le fichier manifest généré automatiquement convient parfaitement si l'archive est utilisée uniquement pour regrouper les fichiers. Pour une utilisation plus spécifique, il faut modifier ce fichier pour ajouter les informations utiles.

Par exemple, pour une application exécutable (à partir de Java 1.2) il faut ajouter une clé Main-Class en lui associant le nom de la classe dans l'archive qui contient la méthode main.

96.3.8. La signature d'une archive jar

La signature d'une archive jar joue un rôle important dans les processus de sécurité de Java. La signature d'une archive permet à celui qui utilise cette archive de lui donner des droits étendus une fois que la signature a été reconnue.

Avec Java 1.1 une archive signée possède tous les droits.

Avec Java 1.2 une archive signée peut se voir attribuer des droits particuliers définis dans un fichier policy.

96.4. L'outil appletviewer pour tester des applets

L'intérêt de cet outil est qu'il permet de tester une applet avec la version courante du JDK. Un navigateur classique nécessite un plug-in pour utiliser une version particulière du JRE. Cet outil est disponible depuis la version 1.0 du JDK.

En contrepartie, l'appletviewer n'est pas prévu pour tester les pages HTML. Il charge une page HTML fournie en paramètre, l'analyse, charge l'applet qu'elle contient et exécute cette applet.

La syntaxe est la suivante : appletviewer [option] fichier

L'appletview recherche le tag HTML <APPLET>. A partir du JDK 1.2, il recherche aussi les tags HTML <EMBED> et <OBJECT>.

Il possède plusieurs options dont les principales sont :

Option	Rôle
-J	Permet de passer un paramètre à la JVM. Pour passer plusieurs paramètres, il faut utiliser plusieurs options -J. Depuis le JDK 1.1
-encoding	Permet de préciser le jeu de caractères de la page HTML

L'appletviewer ouvre une fenêtre qui possède un menu avec les options suivantes :

Option de menu	Rôle
Restart	Permet d'arrêter et de redémarrer l'applet
Reload	Permet d'arrêter et de recharger l'applet
Stop	Permet d'arrêter l'exécution de l'applet. Depuis le JDK 1.1
Save	Permet de sauvegarder l'applet en la sérialisant dans un fichier applet.ser. Il est nécessaire d'arrêter l'applet avant d'utiliser cet option. Depuis le JDK 1.1
Start	Permet de démarrer l'applet. Depuis le JDK 1.1
Info	Permet d'afficher les informations de l'applet dans une boîte de dialogue. Ces informations sont obtenues par les méthodes getAppletInfo() et getParameterInfo() de l'applet.
Print	Permet d'imprimer l'applet. Depuis le JDK 1.1
Close	Permet de fermer la fenêtre courante
Quit	Permet de fermer toutes les fenêtres ouvertes par l'appletviewer

96.5. L'outil javadoc pour générer la documentation technique

Cet outil permet de générer une documentation à partir des données insérées dans le code source. Cet outil est disponible depuis le JDK 1.0

La syntaxe de la commande est la suivante :

javadoc [options] [nom_packages] [fichiers_source] [-sous_packages pkg1:pkg2:...] [@fichiers_arguments]

L'ordre des arguments n'a pas d'importance :

- nom_packages : un ou plusieurs noms de packages séparés par des espaces. (ces packages sont recherchés en utilisant la variable -sourcepath)
- fichiers_sources : un ou plusieurs fichiers sources Java séparés par des espaces. Il est possible de préciser le chemin de chaque fichier. Il est aussi possible d'utiliser le caractère * pour désigner 0 ou n caractères quelconques.
- sous_packages : un ou plusieurs sous-packages à inclure dans la documentation
- @fichiers_arguments : un ou plusieurs fichiers qui contiennent les options à utiliser par Javadoc

Il faut fournir en paramètres de l'outil javadoc un nom de package ou un ensemble de fichiers sources Java.

Les principales options utilisables sont :

Option	Rôle
-classpath chemin	Classpath dans lequel l'outil va rechercher les fichiers sources et .class Cette option permet de remplacer le classpath standard par celui fourni.
-encoding name	Précise le jeu de caractères utilisés dans les fichiers sources
-J flag	Permet de passer un argument à la JVM dans laquelle s'exécute l'outil javadoc Exemple : -J-Xmx32m -J-Xms32m
-sourcepath chemins	Chemins dans lequel l'outil va rechercher les fichiers sources à documenter. Plusieurs chemins peuvent être précisés en utilisant le caractère ; comme séparateur. Par défaut, si ce paramètre n'est pas précisé, c'est le classpath qui est utilisé.
-verbose	Affiche des informations sur la génération en cours
-help	Affiche un résumé des options de la commande
-doclet classe	Permet de préciser un doclet personnalisé (la classe fournie en paramètre doit être pleinement qualifiée).
-docletpath	Permet de préciser le chemin du doclet personnalisé
-exclude packages	Permet de préciser une liste de packages qui ne seront pas pris en compte par l'outil. Le caractère : est utilisé comme séparateur entre chaque package
-package	Inclut seulement les membres et classes package-private, protected et public
-private	Inclut tous les membres et classes
-protected	Inclut seulement les membres et classes protected et public
-public	Inclut seulement les membres et classes public
-overview fichier	Utilise le fichier précisé comme fichier overview-summary.html dans la documentation générée
-source version	Permet de préciser la version de Java utilisée par le code source. Les valeurs possibles sont 1.3, 1.4 et 1.5. Par défaut, c'est la version du JDK qui est utilisée
-subpackages packages	Permet de préciser les packages qui seront pris en compte. Le caractère : est utilisé comme séparateur entre chaque package

L'outil javadoc utilise un doclet pour réaliser le rendu de la documentation générée : il utilise un doclet par défaut si aucun autre doclet n'est spécifié avec l'option -doclet.

Les principales options utilisables du doclet standard sont :

Option	Rôle
-author	Permet de demander la prise en compte des tags @author dans la documentation générée

-bottom	Permet d'insérer du code HTML dans le pied de chaque page
-charset nom	Permet de préciser le jeu de caractères des fichiers HTML générés
-d repertoire	Précise le répertoire dans lequel la documentation va être générée. Par défaut, c'est le répertoire courant qui est utilisé
-docencoding nom	Précise le jeu de caractères utilisés dans les fichiers générés
-doctitle titre	Permet de préciser le titre de la page index (il est possible d'utiliser du code HTML : dans ce cas entourer le titre avec des caractères " ...")
-footer	Permet d'insérer du code HTML à droite du footer par défaut
-help fichier	Permet d'utiliser un fichier d'aide personnalisé. Le fichier doit être au format HTML
-header	Permet d'insérer du code HTML à droite du header par défaut
-link url	Permet de générer des liens vers une documentation (par exemple celle du JDK). L'url peut être relative ou absolue. Exemple : **-link http://java.sun.com/j2se/1.5.0/docs/api**
-linksource	Permet d'inclure dans la documentation le code source au format HTML avec une numérotation des lignes
-nodecprecatedlist	Permet de ne pas générer le fichier deprecated-list.html
-nodeprecated	Permet de ne pas intégrer les tags @deprecated dans la documentation
-nohelp	Permet de ne pas générer le lien "Help"
-noindex	Permet de ne pas générer la page index des packages
-nonavbar	Permet de ne pas générer la barre de navigation
-nosince	Permet de ne pas intégrer les tags @since dans la documentation
-notree	Permet de ne pas générer la hiérarchie des classes
-splitindex	Permet de générer l'index sous la forme d'un fichier par lettre
-stylesheetfile fichier	Permet de préciser la feuille de style utilisée dans la documentation
-use	Permet de demander la génération des pages d'utilisation des classes et packages
-version	Permet de demander la prise en compte des tags @version dans la documentation générée
-windowtitle titre	Permet de préciser le titre des pages

Il est possible de préciser les paramètres de l'outil javadoc (sauf l'option -J) dans un ou plusieurs fichiers en passant leurs noms précédés des caractères @. Dans un tel fichier les paramètres peuvent être séparés par un espace ou un retour chariot.

Il n'est pas possible d'utiliser le caractère joker * dans les noms de fichiers ni de faire une référence à un autre fichier de paramètres avec le caractère @.

Exemple :

C:>javadoc @javadocparam

Exemple : le fichier javadocparam

```
-d documentation -use -splitindex
```

Remarque : l'outil javadoc ne sait pas travailler de façon incrémentale : toute la documentation est à regénérer à chaque fois.

Des informations supplémentaires sur les éléments à inclure dans le code source sont fournies dans le chapitre «JavaDoc».

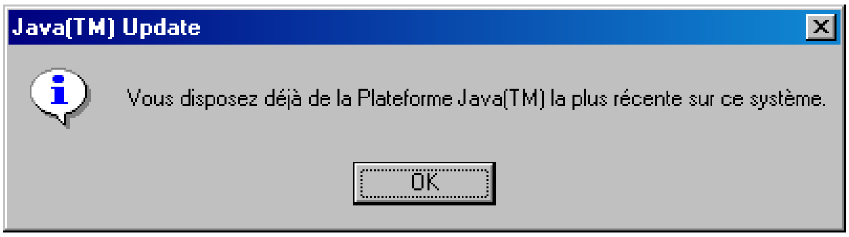
96.6. L'outil Java Check Update pour mettre à jour Java

Jucheck (Java Update Check) est un outil proposé pour permettre une mise à jour automatique de l'environnement d'exécution Java.

L'outil jusched.exe est installé par défaut et configuré pour une exécution automatique depuis la version 1.4.2 du J2SE.

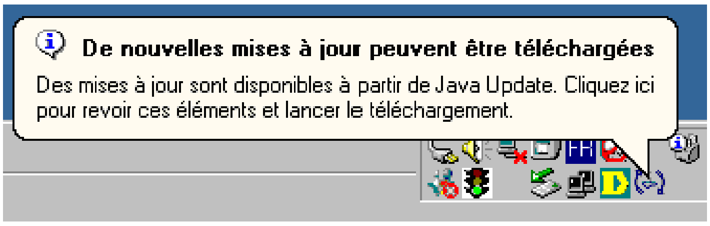
Pour lancer manuellement la mise à jour, il suffit d'exécuter le programme jucheck.exe dans le répertoire bin du JRE.

Si aucune mise à jour n'est disponible, un message est affiché :


Java(TM) Update

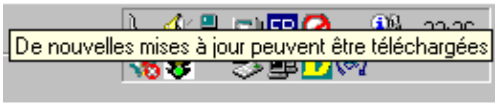
Vous disposez déjà de la Plateforme Java(TM) la plus récente sur ce système.

Sinon une bulle d'aide informe que des mises à jour peuvent être téléchargées.

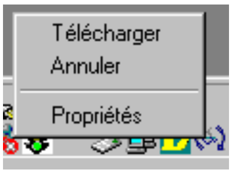

De nouvelles mises à jour peuvent être téléchargées

Des mises à jour sont disponibles à partir de Java Update. Cliquez ici pour revoir ces éléments et lancer le téléchargement.

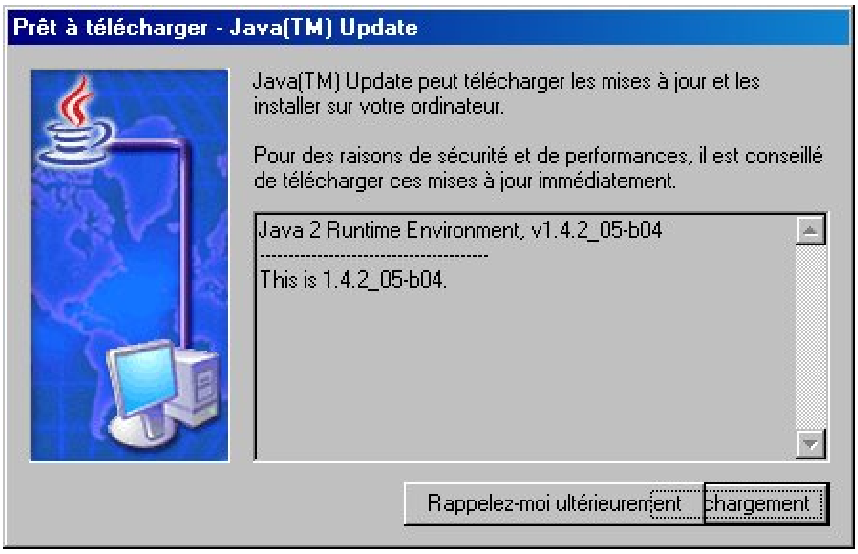
En laissant le curseur de la souris sur l'icône du programme de mise à jour, une bulle d'aide est affichée.


De nouvelles mises à jour peuvent être téléchargées

Pour télécharger les mises à jour, il suffit d'utiliser l'option " Télécharger " du menu contextuel associé à l'icône.



Une boîte de dialogue permet de demander le téléchargement des éléments dont la version est indiquée.



Prêt à télécharger - Java(TM) Update

Java(TM) Update peut télécharger les mises à jour et les installer sur votre ordinateur.

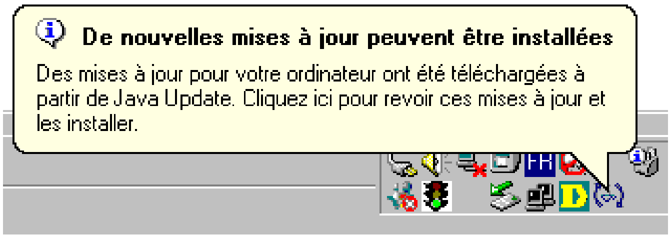
Pour des raisons de sécurité et de performances, il est conseillé de télécharger ces mises à jour immédiatement.

Java 2 Runtime Environment, v1.4.2_05-b04

This is 1.4.2_05-b04.

Cliquez sur le bouton " Téléchargement ".

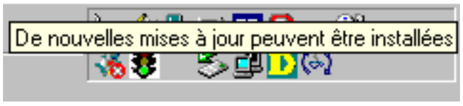
Une fois le téléchargement terminé, une bulle est affichée.



De nouvelles mises à jour peuvent être installées

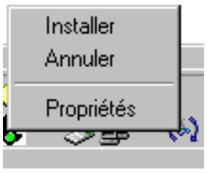
Des mises à jour pour votre ordinateur ont été téléchargées à partir de Java Update. Cliquez ici pour revoir ces mises à jour et les installer.

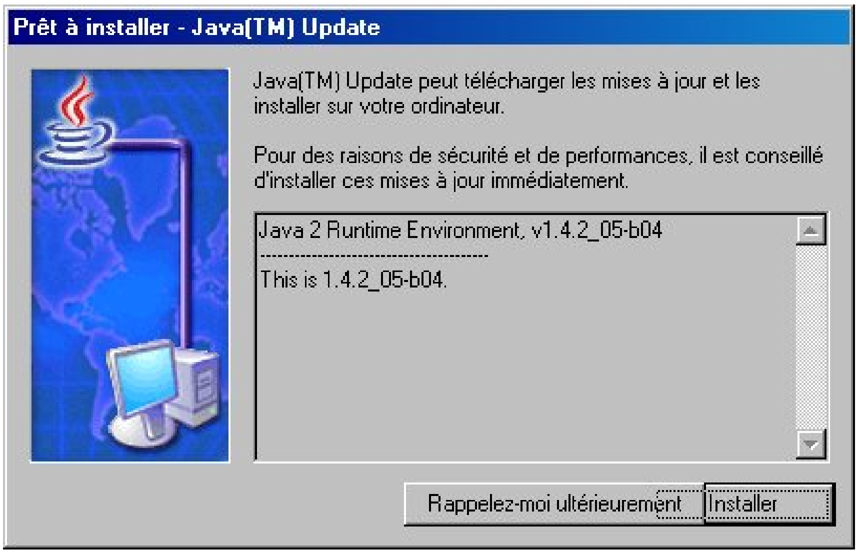
En laissant le curseur de la souris sur l'icône du programme de mise à jour, une bulle d'aide est affichée.



De nouvelles mises à jour peuvent être installées

Pour installer les mises à jour, il suffit d'utiliser l'option " Installer " du menu contextuel associé à l'icône.





Prêt à installer - Java(TM) Update

Java(TM) Update peut télécharger les mises à jour et les installer sur votre ordinateur.

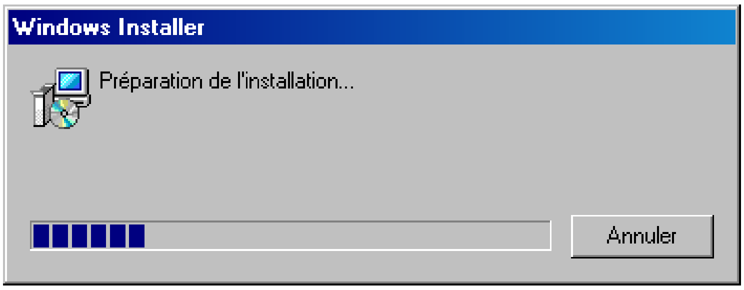
Pour des raisons de sécurité et de performances, il est conseillé d'installer ces mises à jour immédiatement.

Java 2 Runtime Environment, v1.4.2_05-b04

This is 1.4.2_05-b04.

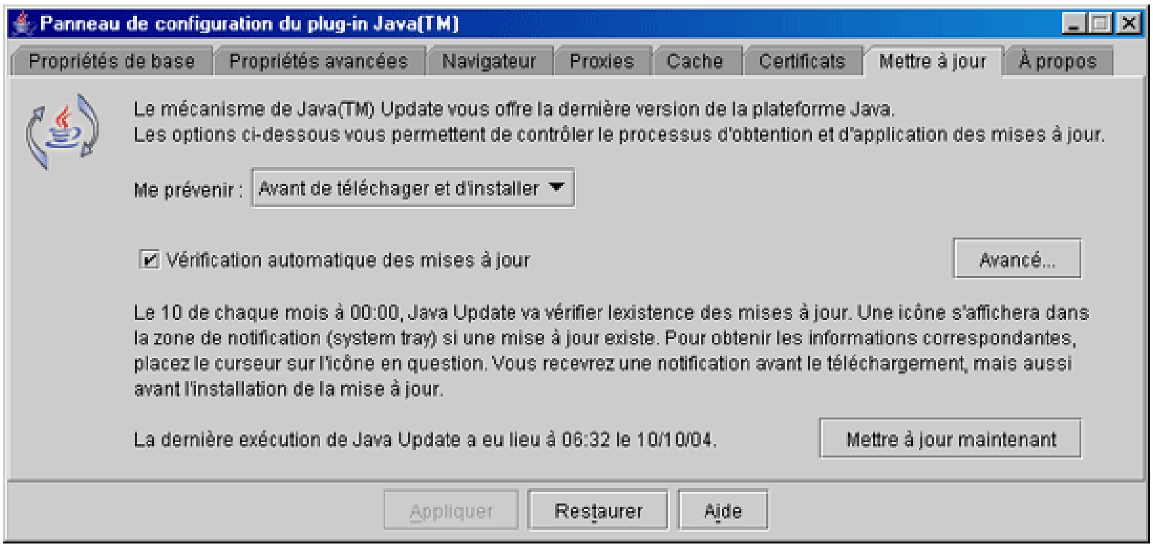
Cliquez sur le bouton " Installer ".

L'assistant se lance pour diriger les différentes étapes.



Préparation de l'installation...

L'option " Propriétés " permet d'ouvrir une boîte de dialogue pour gérer les paramètres des mises à jour dans l'onglet " Mettre à jour ".



Le mécanisme de Java(TM) Update vous offre la dernière version de la plateforme Java.
Les options ci-dessous vous permettent de contrôler le processus d'obtention et d'application des mises à jour.

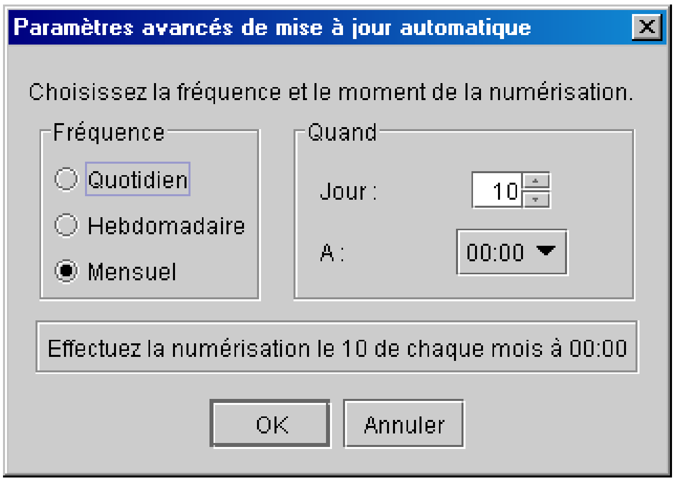
Me prévenir : Avant de télécharger et d'installer

☑ Vérification automatique des mises à jour

Le 10 de chaque mois à 00:00, Java Update va vérifier lexistence des mises à jour. Une icône s'affichera dans la zone de notification (system tray) si une mise à jour existe. Pour obtenir les informations correspondantes, placez le curseur sur l'icône en question. Vous recevrez une notification avant le téléchargement, mais aussi avant l'installation de la mise à jour.

La dernière exécution de Java Update a eu lieu à 06:32 le 10/10/04.

Le bouton " Avancé " permet de définir les paramètres de recherche automatique.



Choisissez la fréquence et le moment de la numérisation.

Fréquence
○ Quotidien
○ Hebdomadaire
● Mensuel

Quand
Jour : 10
A : 00:00

Effectuez la numérisation le 10 de chaque mois à 00:00

96.7. La base de données Java DB

Java SE 6 intègre une base de données : Java DB. C'est en fait la base de données open source Apache Derby écrite entièrement en Java. C'est une base de données légère (2 Mb) qui propose cependant des fonctionnalités intéressantes (gestion des transactions et des accès concurrents, support des triggers et des procédures stockées, ...)

Java DB est donc le nom sous lequel Sun/Oracle propose la base de données open source Derby du groupe Apache dans certains de ses outils notamment le JDK depuis sa version 6.0.

Attention : à partir des version 9 et 8u181, Java DB n'est plus fourni dans le JDK.

Java DB est stockée dans le sous-répertoire db du répertoire d'installation d'un JDK. Avec le JDK 6.0 c'est la version 10.2 de Java DB qui est fournie.

Remarque : dans cette section Java DB peut être remplacée par Derby et vice versa.

L'ajout de Java DB dans le JDK permet de rapidement écrire des applications qui utilisent une base de données et les fonctionnalités proposées par la version 4.0 de JDBC.

Java DB est idéale dans un environnement de développement car elle est riche en fonctionnalités, facile à utiliser, multiplate-forme puisqu'écrite en Java et peut être mise en oeuvre sur une simple machine de développement.

Java DB peut fonctionner selon deux modes :

- Embedded : la base de données est exécutée comme une partie de l'application
- Client/server : la base de données est exécutée de façon indépendante de l'application

Java DB peut être intégrée dans une application (mode Embedded) ce qui évite d'avoir à installer et configurer une base de données lors du déploiement de l'application.

Java DB propose plusieurs outils pour assurer sa gestion.

L'outil ij permet d'exécuter des scripts sur une base de données.

Le lancement de l'outil ij peut se faire de deux façons :

Exemple :

```
C:\Program Files\Java\jdk1.6.0\db>java -jar lib\derbyrun.jar ij
version ij 10.2
ij>
```

ou

Exemple :

```
C:\Program Files\Java\jdk1.6.0\db>java -cp .\lib\derbytools.jar org.apache.derby
.tools.ij
ij version 10.2
ij>
```

La création d'une base de données nommée MaBaseDeTest et la connexion à cette nouvelle base se font en utilisant l'outil ij.

Exemple :

```
ij> CONNECT 'jdbc:derby:MaBaseDeTest;create=true';
```

Remarque : l'outil ij peut être utilisé avec n'importe quel pilote JDBC.

Exemple : Création de la table Personne

```
ij> CREATE TABLE PERSONNE (ID INT PRIMARY KEY, NOM VARCHAR(50), PRENOM VARCHAR(5
0));
0 lignes insérés/mises à jour/supprimées

ij> select * from PERSONNE;
```

```
ID         |NOM                                               |PRENOM

--------------------------------------------------------------------------------
---------------------------------

0 lignes sélectionnées
```

Exemple : Ajout d'occurrences dans la table Personne

```
ij> INSERT INTO PERSONNE VALUES (1,'nom1','prenom1'), (2,'nom2','prenom2'), (3,'
nom3','prenom3');
3 lignes insérées/mises à jour/supprimées
ij> select * from PERSONNE;
ID         |NOM                                               |PRENOM

--------------------------------------------------------------------------------
---------------------------------
1          |nom1                                              |prenom1

2          |nom2                                              |prenom2

3          |nom3                                              |prenom3


3 lignes sélectionnées
```

La commande exit; permet de quitter l'outil ij.

Une fois la base de données créée, il est possible d'utiliser par exemple l'API JDBC pour s'y connecter et effectuer des opérations sur ses données.

Exemple : utilisation de Java DB en mode embedded

```
package fr.jmdoudoux.dej.jpa;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.Statement;

public class TestDerby {

  private static String dbURL =
    "jdbc:derby:C:/Program Files/Java/jdk1.6.0/db/MaBaseDeTest;user=APP";

  public static void main(String[] args) {

    try {
      Connection conn = null;
      Statement stmt = null;

      Class.forName("org.apache.derby.jdbc.EmbeddedDriver").newInstance();
      conn = DriverManager.getConnection(dbURL);

      stmt = conn.createStatement();
      ResultSet results = stmt.executeQuery("select * from personne");
      ResultSetMetaData rsmd = results.getMetaData();
      int nbColonnes = rsmd.getColumnCount();
      for (int i = 1; i <= nbColonnes; i++) {
        System.out.print(rsmd.getColumnLabel(i) + "\t\t");
      }

      System.out.println("\n-------------------------------------------------");

      while (results.next()) {
        System.out.println(results.getInt(1) + "\t\t"
          + results.getString(2) + "\t\t" + results.getString(3));
      }
      results.close();
      stmt.close();
      if (conn != null) {
        conn.close();
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

}
```

Remarque : par défaut dans le mode embeded, les tables sont créées dans le schéma APP.

Pour utiliser Java DB en mode client/serveur, il suffit de remplacer le pilote JDBC par org.apache.derby.jdbc.ClientDriver et de changer dans l'url le chemin local pour l'url de la base de données.

L'outil dblook permet de générer le DDL d'une base de données.

Exemple :

```
C:\Program Files\Java\jdk1.6.0\db>java -jar lib\derbyrun.jar dblook -d jdbc:derb
y:MaBaseDeTest
-- Horodatage : 2007-06-28 15:42:21.613
-- La base de données source est : MaBaseDeTest
-- L'URL de connexion est : jdbc:derby:MaBaseDeTest
-- appendLogs: false

-- ----------------------------------------------
-- Instructions DDL pour tables
-- ----------------------------------------------

CREATE TABLE "APP"."ADRESSE" ("ID_ADRESSE" INTEGER NOT NULL, "RUE" VARCHAR(250)
NOT NULL, "CODEPOSTAL" VARCHAR(7) NOT NULL, "VILLE" VARCHAR(250) NOT NULL);

CREATE TABLE "APP"."PERSONNE" ("ID" INTEGER NOT NULL, "NOM" VARCHAR(50), "PRENOM
" VARCHAR(50));

-- ----------------------------------------------
-- Instructions DDL pour clés
-- ----------------------------------------------

-- primaire/unique
ALTER TABLE "APP"."ADRESSE" ADD CONSTRAINT "SQL070628114148710" PRIMARY KEY ("ID
_ADRESSE");

ALTER TABLE "APP"."PERSONNE" ADD CONSTRAINT "SQL070627114331220" PRIMARY KEY ("I
D");
```

L'outil sysinfo permet d'obtenir des informations sur l'environnement et sur la base de données.

Exemple :

```
C:\Program Files\Java\jdk1.6.0\db>java -cp .\lib\derbytools.jar org.apache.derby
.tools.ij
ij version 10.2
ij> exit;

C:\Program Files\Java\jdk1.6.0\db>java -jar lib\derbyrun.jar sysinfo
------------------ Informations Java ------------------
Version Java :    1.6.0_01
Fournisseur Java :     Sun Microsystems Inc.
Répertoire principal Java :       C:\Program Files\Java\jre1.6.0_01
Chemin de classes Java :  lib\derbyrun.jar
Nom du système d'exploitation :         Windows XP
Architecture du système d'exploitation :         x86
Version du système d'exploitation :         5.1
Nom d'utilisateur Java :  JMD
Répertoire principal utilisateur Java :  C:\Documents and Settings\jmd
```

```
Répertoire utilisateur Java :   C:\Program Files\Java\jdk1.6.0\db
java.specification.name: Java Platform API Specification
java.specification.version: 1.6
--------- Informations Derby --------
JRE - JDBC: Java SE 6 - JDBC 4.0
[C:\Program Files\Java\jdk1.6.0\db\lib\derby.jar] 10.2.1.7 - (453926)
[C:\Program Files\Java\jdk1.6.0\db\lib\derbytools.jar] 10.2.1.7 - (453926)
[C:\Program Files\Java\jdk1.6.0\db\lib\derbynet.jar] 10.2.1.7 - (453926)
[C:\Program Files\Java\jdk1.6.0\db\lib\derbyclient.jar] 10.2.1.7 - (453926)
------------------------------------------------------
----------------- Informations sur lenvironnement local -----------------
Environnement local actuel :  [français/Luxembourg [fr_LU]]
La prise en charge de cet environnement local a été trouvée : [de_DE]
         version : 10.2.1.7 - (453926)
La prise en charge de cet environnement local a été trouvée : [es]
         version : 10.2.1.7 - (453926)
La prise en charge de cet environnement local a été trouvée : [fr]
         version : 10.2.1.7 - (453926)
La prise en charge de cet environnement local a été trouvée : [it]
         version : 10.2.1.7 - (453926)
La prise en charge de cet environnement local a été trouvée : [pt_BR]
         version : 10.2.1.7 - (453926)
------------------------------------------------------
```

Pour démarrer JavaDB en mode client-server, il faut exécuter l'application org.apache.derby.drda.NetworkServerControl.

Exemple :

```
C:\Program Files\Java\jdk1.6.0\db>java -cp .\lib\derbyrun.jar org.apache.derby.d
rda.NetworkServerControl start -h localhost
Le serveur est prêt à accepter les connexions au port 1527.
```

L'option -h permet de préciser le serveur.
L'option - p permet de préciser le port à utiliser si le port 1527 utilisé par défaut ne convient pas.

96.8. L'outil JConsole

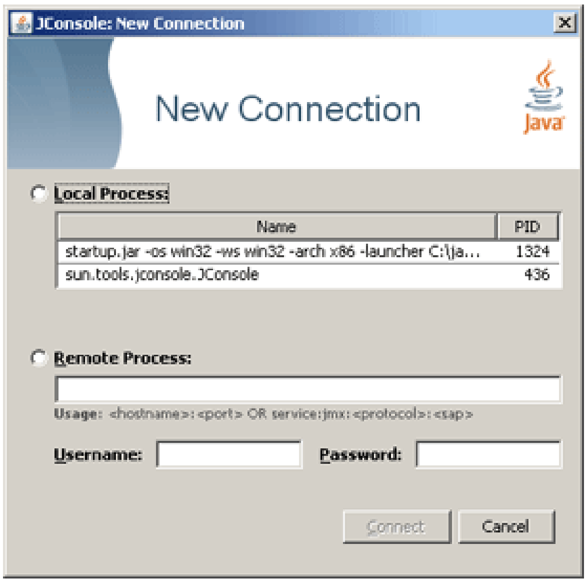
Depuis Java SE 5.0, le JDK propose l'outil JConsole qui est une interface de monitoring et de management utilisant JMX. JConsole est une application graphique qui est un client JMX permettant de mettre en oeuvre la plupart des fonctionnalités de JMX.

JConsole est dans le sous-répertoire bin du répertoire d'installation du JDK.

Remarque : Il n'est pas recommandé d'utiliser JConsole en production car son utilisation consomme beaucoup de ressources. Si son utilisation est nécessaire, il est recommandé d'exécuter JConsole sur un système distant de celui de la JVM à surveiller.

Pour utiliser JConsole , il suffit d'exécuter jconsole dans une nouvelle boîte de commande.

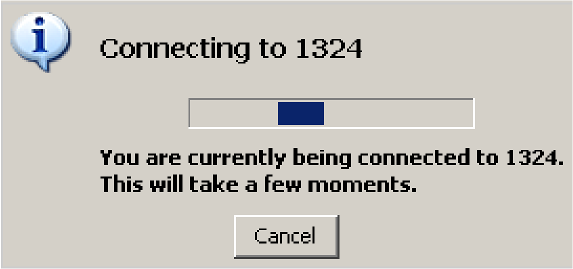
Remarque : sous Windows, JMX ne fonctionne correctement que si la partition du système est formatée en NTFS.



Name	PID
startup.jar -os win32 -ws win32 -arch x86 -launcher C:\ja...	1324
sun.tools.jconsole.JConsole	436

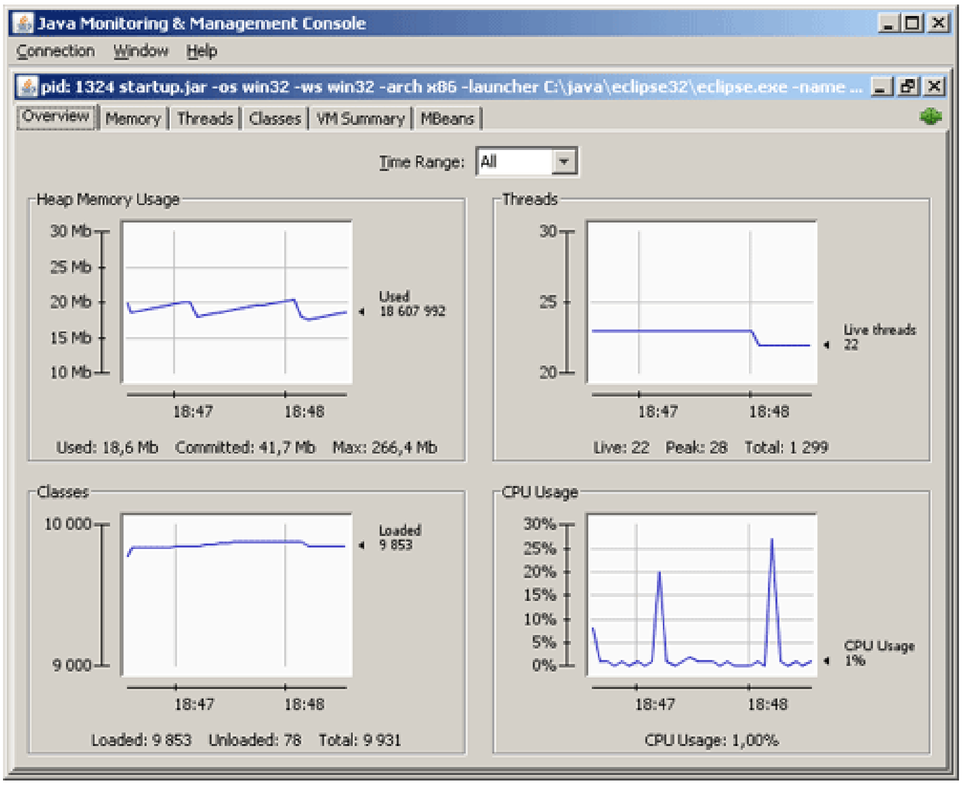
Usage: <hostname>:<port> OR service:jmx:<protocol>:<sap>

Une fenêtre de connexion permet de préciser quel sera le processus d'une JVM à utiliser. Pour une JVM locale, il suffit de sélectionner « Local Process », la JVM parmi celles proposées, et de cliquer sur « Connect ».

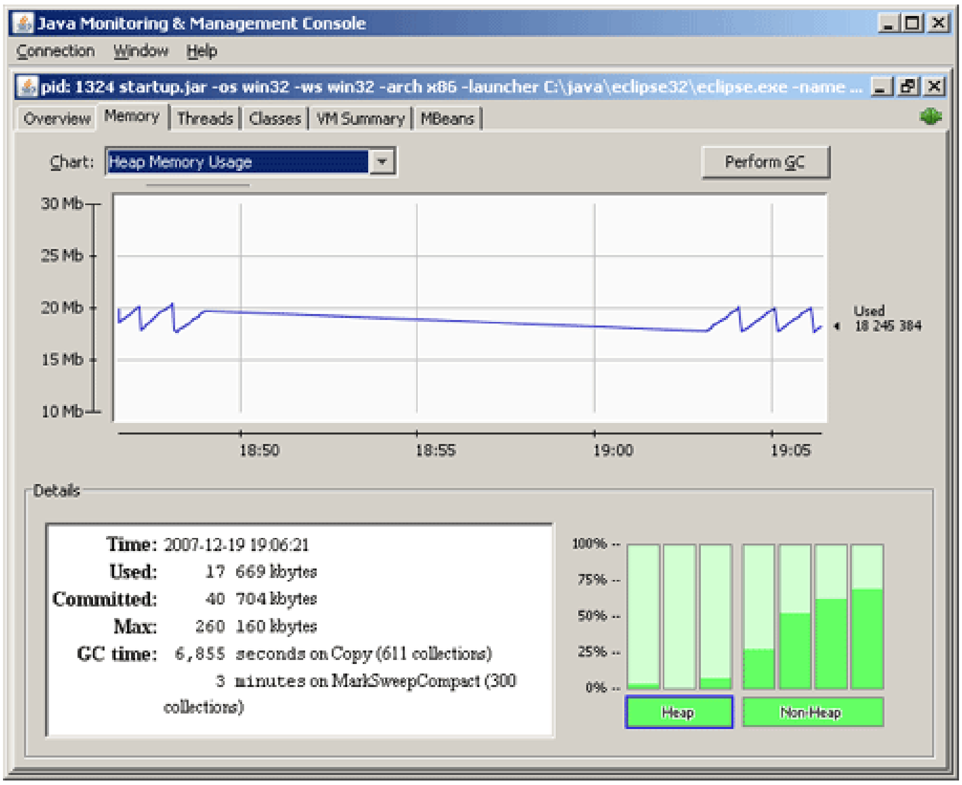


You are currently being connected to 1324.
This will take a few moments.

Pour une connexion distante, il faut saisir l'url de connexion et éventuellement le user et le mot de passe si l'authentification est activée. Une fois la connexion établie, une fenêtre contenant plusieurs onglets permet d'avoir des informations sur l'état de différentes fonctionnalités.



L'onglet « Overview » fournit un résumé graphique sur quatre données : l'usage du tas (heap) et du processeur (CPU), le nombre de threads et de classes chargées.

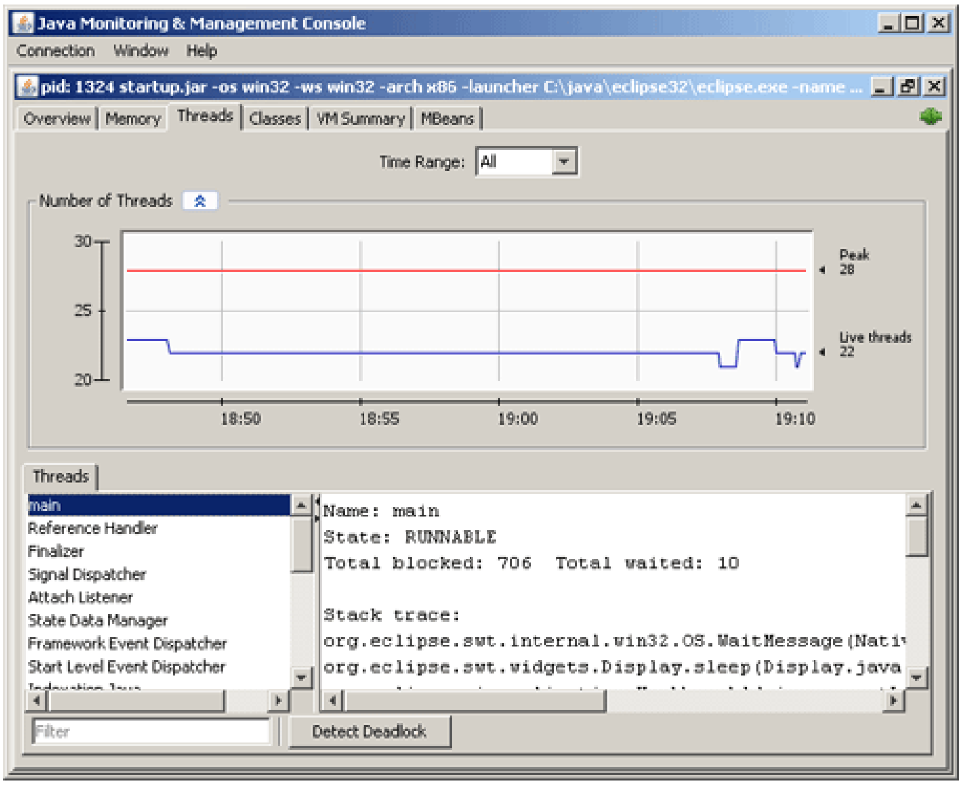


Time: 2007-12-19 19:06:21
Used: 17 669 kbytes
Committed: 40 704 kbytes
Max: 260 160 kbytes
GC time: 6,855 seconds on Copy (611 collections)
3 minutes on MarkSweepCompact (300 collections)

L'onglet « Memory » permet d'avoir des informations précises sur la mémoire.

Le contenu du graphique principal et des détails peuvent être sélectionnés dans la liste déroulante chart : Heap Memory Usage, Non-Heap Memory Usage, Eden Space, Survivor Space, Tenured Space, Code Cache, Perm Gen, Perm Gen shared-rw et Perm Gen shared-ro. Ceci permet d'avoir une vue précise sur chaque partie de la mémoire de la JVM.

Le bouton « Perform GC » permet de demander l'exécution du ramasse-miettes.

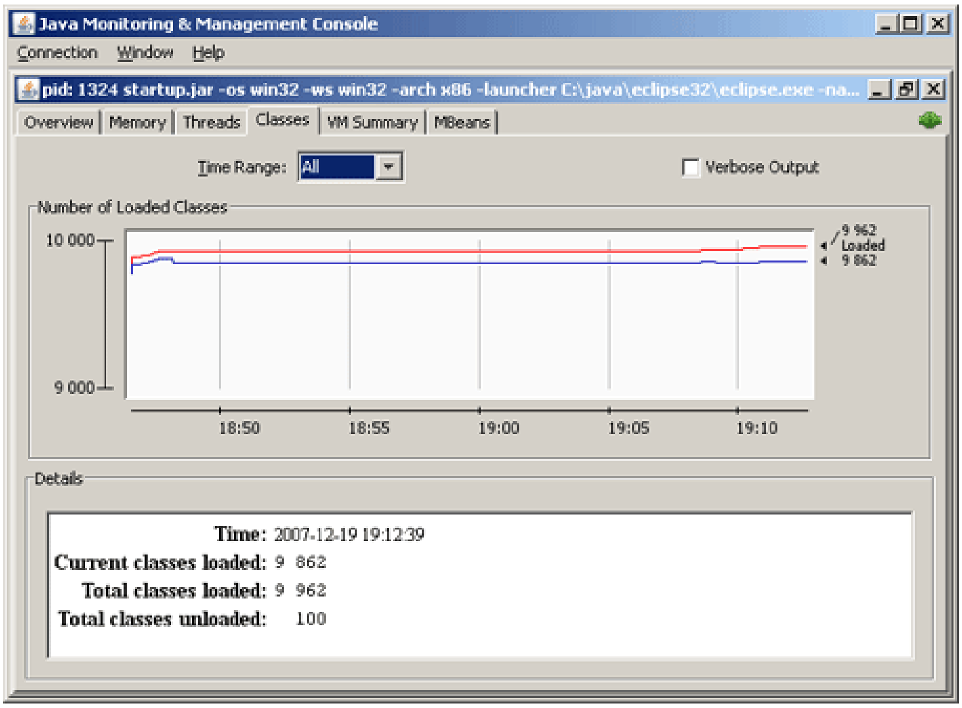


```
Name: main
State: RUNNABLE
Total blocked: 706  Total waited: 10

Stack trace:
org.eclipse.swt.internal.win32.OS.WaitMessage(Nati
org.eclipse.swt.widgets.Display.sleep(Display.java
```

L'onglet « Threads » permet d'avoir des informations sur les threads en cours d'exécution.

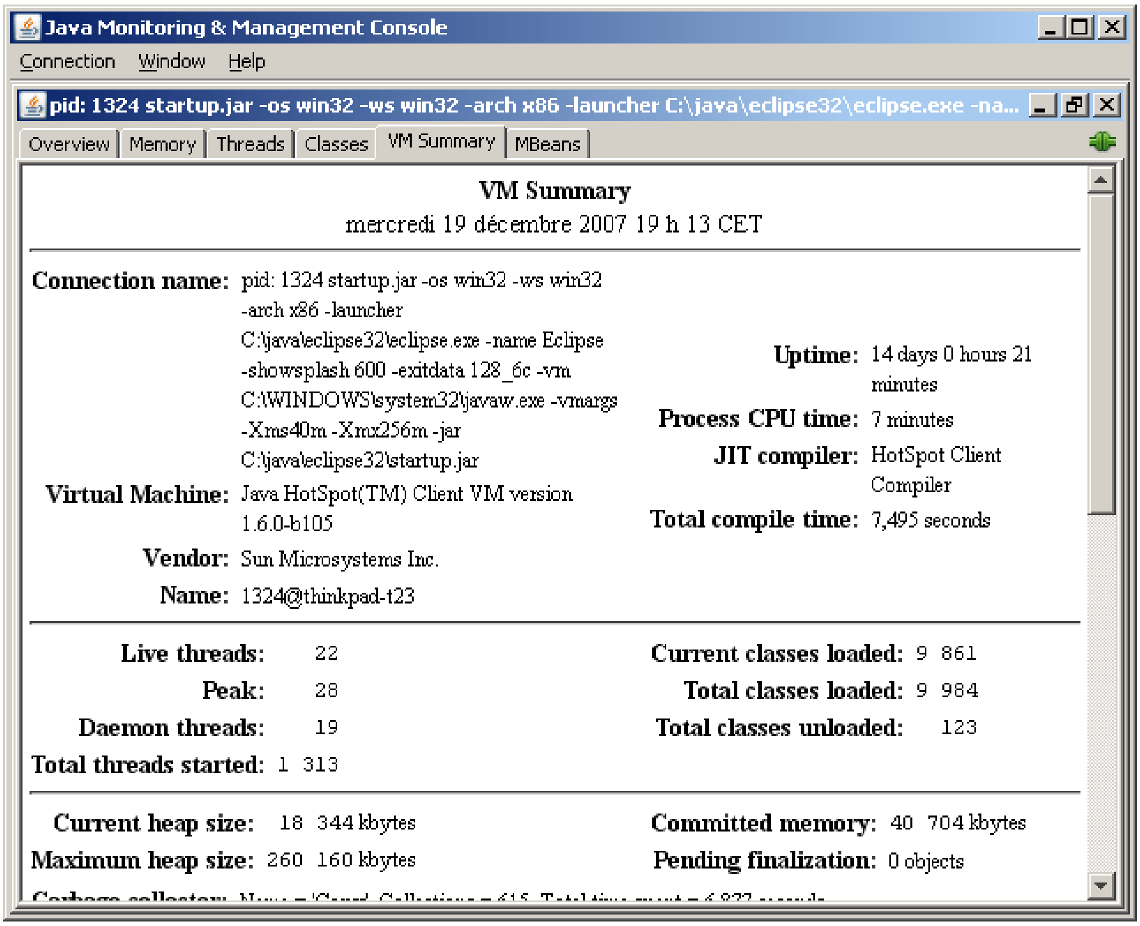
Pour avoir des informations précises sur un thread, il suffit de le sélectionner.

L'onglet « Classes » permet de connaître le nombre de classes chargées.


Java Monitoring & Management Console
Connection Window Help
pid: 1324 startup.jar -os win32 -ws win32 -arch x86 -launcher C:\java\eclipse32\eclipse.exe -na...
Overview Memory Threads Classes VM Summary MBeans
Time Range: All
Verbose Output
Number of Loaded Classes
10 000
9 000
18:50
18:55
19:00
19:05
19:10
9 962
Loaded
9 862
Details

Time: 2007-12-19 19:12:39
Current classes loaded: 9 862
Total classes loaded: 9 962
Total classes unloaded: 100

L'onglet « VM Summary » affiche de nombreuses informations sur l'état de la JVM. Celles-ci sont rafraîchies toutes les 5 secondes.


Java Monitoring & Management Console
Connection Window Help
pid: 1324 startup.jar -os win32 -ws win32 -arch x86 -launcher C:\java\eclipse32\eclipse.exe -na...
Overview Memory Threads Classes VM Summary MBeans

VM Summary
mercredi 19 décembre 2007 19 h 13 CET

Connection name: pid: 1324 startup.jar -os win32 -ws win32 -arch x86 -launcher C:\java\eclipse32\eclipse.exe -name Eclipse -showsplash 600 -exitdata 128_6c -vm C:\WINDOWS\system32\javaw.exe -vmargs -Xms40m -Xmx256m -jar C:\java\eclipse32\startup.jar
Virtual Machine: Java HotSpot(TM) Client VM version 1.6.0-b105
Vendor: Sun Microsystems Inc.
Name: 1324@thinkpad-t23

Uptime: 14 days 0 hours 21 minutes
Process CPU time: 7 minutes
JIT compiler: HotSpot Client Compiler
Total compile time: 7,495 seconds

Live threads: 22
Peak: 28
Daemon threads: 19
Total threads started: 1 313

Current classes loaded: 9 861
Total classes loaded: 9 984
Total classes unloaded: 123

Current heap size: 18 344 kbytes
Maximum heap size: 260 160 kbytes

Committed memory: 40 704 kbytes
Pending finalization: 0 objects

L'onglet « MBean » permet de gérer les MBeans de l'API JMX qui sont accessibles dans la JVM.

97. JavaDoc

Chapitre 97

Niveau : 🟩🟩⬜⬜⬜⬜ Elémentaire

Javadoc est un outil fourni avec le JDK pour permettre la génération d'une documentation technique à partir du code source.

Cet outil génère une documentation au format HTML à partir du code source Java et des commentaires particuliers qu'il contient. Un exemple concret de l'utilisation de cet outil est la documentation du JDK qui est générée grâce à Javadoc.

Cette documentation contient :

- une description détaillée pour chaque classe et ses membres public et protected par défaut (sauf les classes internes anonymes)
- un ensemble de listes (liste des classes, hiérarchie des classes, liste des éléments deprecated et un index général)
- des références croisées et une navigation entre ces différents éléments.

L'intérêt de ce système est de conserver dans le même fichier le code source et les éléments de la documentation qui lui sont associés. Il propose donc une auto-documentation des fichiers sources de façon standard.

Ce chapitre contient plusieurs sections :

- ♦ La mise en oeuvre
- ♦ Les tags définis par javadoc
- ♦ Un exemple
- ♦ Les fichiers pour enrichir la documentation des packages
- ♦ La documentation générée

97.1. La mise en oeuvre

Javadoc s'appuie sur le code source et sur un type de commentaires particuliers pour obtenir des données supplémentaires des éléments qui composent le code source.

L'outil Javadoc utilise plusieurs types de fichiers sources pour générer la documentation :

- Les fichiers sources .java
- Les fichiers de commentaires d'ensemble
- Les fichiers de commentaires des packages
- D'autres fichiers tels que des images, des fichiers HTML, ...

En fonction des paramètres fournis à l'outil, ce dernier recherche les fichiers source .java concernés. Les sources de ces fichiers sont scannées pour déterminer leurs membres, extraire les informations utiles et établir un ensemble de références croisées.

Le résultat de cette recherche peut être enrichi avec des commentaires dédiés insérés dans le code avant chaque élément qu'ils enrichissent. Ces commentaires doivent immédiatement précéder l'entité qu'ils concernent (classe, interface, méthode, constructeur ou champ). Seul le commentaire qui précède l'entité est traité lors de la génération de la documentation.

Ces commentaires suivent des règles précises. Le format de ces commentaires commence par /** et se termine par */. Il peut contenir un texte libre et des balises particulières.

Le commentaire peut être sur une ou plus généralement sur plusieurs lignes. Les caractères d'espacement (espace et tabulation) qui précèdent le premier caractère * de chaque ligne du commentaire ainsi que le caractère lui-même sont ignorés lors de la génération. Ceci permet d'utiliser le caractère * pour aligner le contenu du commentaire.

Exemple :

```
  /** Description */
```

Le format général de ces commentaires est :

Exemple :

```
/**
 * Description
 *
 * @tag1
 * @tag2
 */
```

Le commentaire doit commencer par une description de l'élément qui peut utiliser plusieurs lignes. La première phrase de cette description est utilisée par javadoc comme résumé. Cette première phrase se termine par un caractère '.' suivi d'un séparateur (espace ou tabulation ou retour chariot) ou à la rencontre du premier tag Javadoc.

Le texte du commentaire doit être au format HTML : les tags HTML peuvent donc être utilisés pour enrichir le formatage de la documentation. Il est donc aussi nécessaire d'utiliser les entités d'échappement pour certains caractères contenus dans le texte tels que < ou >. Il ne faut surtout pas utiliser les tags de titres <Hn> et le tag du séparateur horizontal <HR> car ils sont utilisés par Javadoc pour structurer le document.

Exemple :

```
  /**
   * Description de la classe avec des  <b>mots en gras</b>
   */
```

L'utilisation de balises de formatage HTML est particulièrement intéressante pour formater une description un peu longue en faisant usage notamment du tag <p> pour définir des paragraphes ou du tag <code> pour encadrer un extrait de code.

A partir du JDK 1.4, si la ligne ne commence pas par un caractère *, alors les espaces ne sont plus supprimés (ceci permet par exemple de conserver l'indentation d'un morceau de code contenu dans un tag HTML <PRE>).

Le commentaire peut ensuite contenir des tags Javadoc particuliers qui commencent obligatoirement par le caractère @ et doivent être en début de ligne. Ces tags doivent être regroupés ensemble. Un texte qui suit cet ensemble de tags est ignoré.

Les tags prédéfinis par Javadoc permettent de fournir des informations plus précises sur des composants particuliers de l'élément (auteur, paramètres, valeur de retour, ...). Ces tags sont définis pour un ou plusieurs types d'éléments.

Les tags sont traités de façon particulière par Javadoc. Il existe deux types de tags :

- Block tag : ils sont de la forme @tag
- Inline tag : ils sont de la forme{@tag}

Attention un caractère @ en début de ligne est interprété comme un tag. Si un tel caractère doit apparaître en début de ligne dans la description, il faut utiliser la séquence d'échappement HTML &#064;

Le texte associé à un block tag suit le tag et se termine à la rencontre du tag suivant ou de la fin du commentaire. Ce texte peut donc s'étendre sur plusieurs lignes.
Les tags inline peuvent être utilisés n'importe où dans le commentaire de documentation.

97.2. Les tags définis par javadoc

L'outil Javadoc traite de façon particulière les tags dédiés insérés dans le commentaire de documentation. Javadoc définit plusieurs tags qui permettent de préciser certains composants de l'élément décrit de façon standardisée. Ces tags commencent tous par le caractère arobase @.

Il existe deux types de tags :

- Block tag : ils doivent être regroupés après la description. Ils sont de la forme @tag
- Inline tag : ils peuvent être utilisés n'importe où dans le commentaire. Ils sont de la forme {@tag}

Les block tags doivent obligatoirement doivent être placés en début de ligne (après d'éventuels blancs et un caractère *).

Attention : les tags sont sensibles à la casse.

Pour pouvoir être interprétés, les tags standards doivent obligatoirement commencer en début de ligne.

Tag	Rôle	version du JDK
@author	permet de préciser le ou les auteurs de l'élément	1.0
{@code}		1.5
@deprecated	permet de préciser qu'un élément est déprécié	1.1
{@docRoot}	représente le chemin relatif du répertoire principal de génération de la documentation	1.3
@exception	permet de préciser une exception qui peut être levée par l'élément	1.0
{@inheritDoc}		1.4
{@link}	permet d'insérer un lien vers un élément de la documentation dans n'importe quel texte	1.2
{@linkplain}		1.4
{@literal}		1.5
@param	permet de documenter un paramètre de l'élément	1.0
@return	permet de fournir une description de la valeur de retour d'une méthode qui en possède une : inutile donc de l'utiliser sur une méthode qui retourne void.	1.0
@see	permet de préciser un élément en relation avec l'élément documenté	1.0
@serial		1.2
@serialData		1.2
@serialField		1.2
@since	permet de préciser depuis quelle version l'élément a été ajouté	1.1
@throws	identique à @exception	1.2
@version	permet de préciser le numéro de version de l'élément	1.0
{@value}		1.4

Ces tags ne peuvent être utilisés que pour commenter certaines entités.

Entité	Tags utilisables
Toutes	@see, @since, @deprecated, {@link}, {@linkplain}, {@docroot}
Overview (fichier overview.html)	@see, @since, @author, @version, {@link}, {@linkplain}, {@docRoot}
Package (fichier package.html)	@see, @since, @serial, @author, @version, {@link}, {@linkplain}, {@docRoot}
Classes et Interfaces	@see, @since, @deprecated, @serial, @author, @version, {@link}, {@linkplain}, {@docRoot}
Constructeurs et méthodes	@see, @since, @deprecated, @param, @return, @throws, @exception, @serialData, {@link}, {@linkplain}, {@inheritDoc}, {@docRoot}
Champs	@see, @since, @deprecated, @serial, @serialField, {@link}, {@linkplain}, {@docRoot}, {@value}

Chacun des tags sera détaillé dans les sections suivantes.

Par convention, il est préférable de regrouper les tags identiques ensemble.

97.2.1. Le tag @author

Le tag @author permet de préciser le ou les auteurs d'une entité.

La syntaxe de ce tag est la suivante :

@author texte

Le texte qui suit la balise est libre. Le doclet standard crée une section "Author" qui contient le texte du tag.

Pour préciser plusieurs auteurs, il est possible d'utiliser un seul ou plusieurs tag @author dans un même commentaire. Dans le premier cas, le contenu du texte est repris intégralement dans la section. Dans le second cas, la section contient le texte de chaque tag séparé par une virgule et un espace.

Exemple :

@author Pierre G.

@author Denis T., Sophie D.

Ce tag n'est utilisable que dans les commentaires d'ensemble, d'une classe ou d'une interface.

A partir du JDK 1.4, il est possible au travers du paramètre -tag de préciser que le tag @author peut être utilisé sur d'autres membres

Exemple :

-tag author:a:"Author:"

97.2.2. Le tag @deprecated

Le tag @deprecated permet de préciser qu'une entité ne devrait plus être utilisée même si elle fonctionne toujours : il permet donc de donner des précisions sur un élément déprécié (deprecated).

La syntaxe de ce tag est la suivante :

@deprecated texte

Il est recommandé de préciser depuis quelle version l'élément est déprécié et de fournir dans le texte libre une description de la solution de remplacement, si elle existe, ainsi qu'un lien vers une entité de substitution.

Le doclet standard crée une section "Deprecated" avec l'explication dans la documentation.

Remarque : Ce tag est particulier car il est le seul reconnu par le compilateur : celui-ci prend note de cet attribut lors de la compilation pour permettre d'en informer les utilisateurs. Lors de la compilation, l'utilisation d'entités marquées avec le tag @deprecated générera un avertissement (warning) de la part du compilateur.

Exemple Java 1.1 :

@deprecated Remplacé par setMessage

@see #setMessage

Exemple Java 1.2 :

@deprecated Remplaçé par {@link #setMessage}

97.2.3. Le tag @exception et @throws

Ces tags permettent de documenter une exception levée par la méthode ou le constructeur décrit par le commentaire.

Syntaxe :

@exception nom_exception description

Les tags @exception et @throws sont similaires.

Ils sont suivis du nom de l'exception puis d'une courte description des raisons de la levée de cette dernière. Il faut utiliser autant de tag @exception ou @throws qu'il y a d'exceptions. Ce tag doit être utilisé uniquement pour un élément de type méthode.

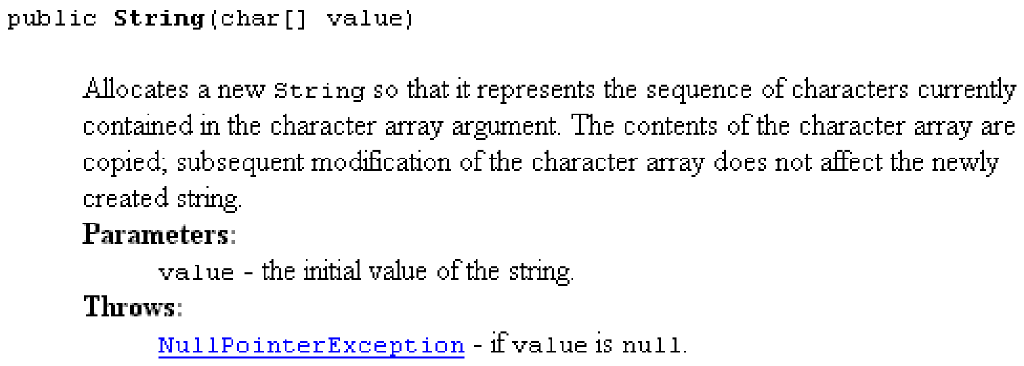
Il ne faut pas mettre de séparateur particulier comme un caractère '-' entre le nom et la description puisque l'outil en ajoute un automatiquement. Il est cependant possible d'aligner les descriptions de plusieurs paramètres en utilisant des espaces afin de faciliter la lecture.

Exemple :

@exception java.io.FileNotFoundException le fichier n'existe pas

Le doclet standard crée une section "Throws" qui regroupe les exceptions : l'outil recherche le nom pleinement qualifié de chaque exception si c'est simplement leur nom qui est précisé dans le tag.

Exemple extrait de la documentation de l'API du JDK :



`public String(char[] value)`

Allocates a new `String` so that it represents the sequence of characters currently contained in the character array argument. The contents of the character array are copied; subsequent modification of the character array does not affect the newly created string.

Parameters:

`value` - the initial value of the string.

Throws:

`NullPointerException` - if `value` is `null`.

97.2.4. Le tag @param

Le tag @param permet de documenter un paramètre d'une méthode ou d'un constructeur. Ce tag doit être utilisé uniquement pour un élément de type constructeur ou méthode.

La syntaxe de ce tag est la suivante :

@param nom_paramètre description du paramètre

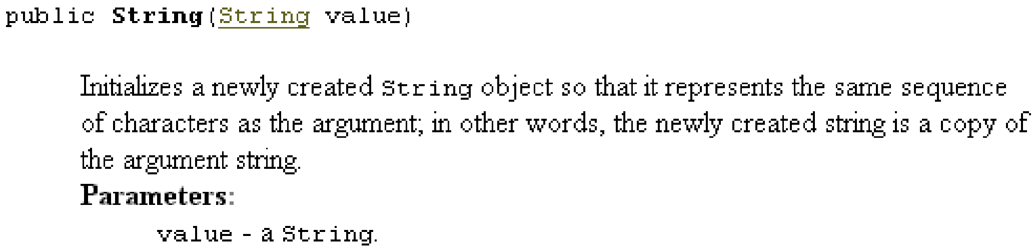
Ce tag est suivi du nom du paramètre (ne pas utiliser le type) puis d'une courte description de ce dernier. A partir de Java 5, il est possible d'utiliser le type du paramètre entre les caractères < et > pour une classe ou une méthode.

Il ne faut pas mettre de séparateur particulier comme un caractère '-' entre le nom et la description puisque l'outil en ajoute un automatiquement. Il est cependant possible d'aligner les descriptions de plusieurs paramètres en utilisant des espaces afin de faciliter la lecture.

Il faut utiliser autant de tag @param que de paramètres dans la signature de l'entité concernée. La description peut être contenue sur plusieurs lignes.

Le doclet standard crée une section "Parameters" qui regroupe les tags @param du commentaire. Il génère pour chaque tag une ligne dans cette section avec son nom et sa description dans la documentation.

Exemple extrait de la documentation de l'API du JDK :



public **String**(String value)

Initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.

Parameters:

value - a String.

Par convention les paramètres doivent être décrits dans leur ordre dans la signature de la méthode décrite

Exemple :

@param nom nom de la personne

@param message chaîne de caractères à traiter. Si cette valeur est <code>null</code> alors une exception est levée

Exemple 2 : /** * @param <E> Type des elements stockés dans la collection */
public interface List<E> extends Collection<E> { }

97.2.5. Le tag @return

Le tag @return permet de fournir une description de la valeur de retour d'une méthode qui en possède une.

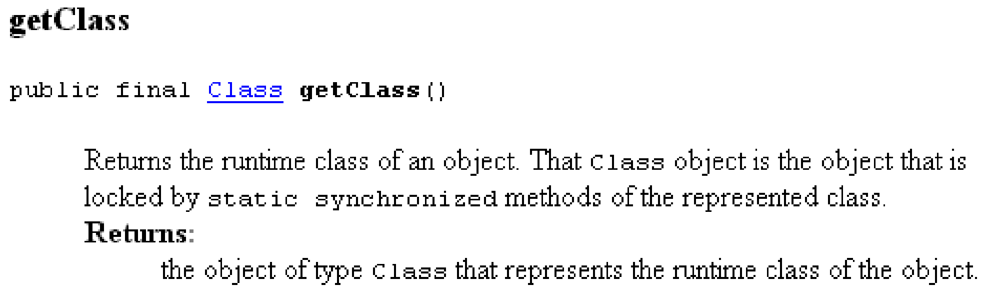
La syntaxe de ce tag est la suivante :

@return description_de_la_valeur_de retour_de_la_méthode

Il ne peut y avoir qu'un seul tag @return par commentaire : il doit être utilisé uniquement pour un élément de type méthode qui renvoie une valeur.

Avec le doclet standard, ce tag crée une section "Returns" qui contient le texte du tag. La description peut tenir sur plusieurs lignes.

Exemple extrait de la documentation de l'API du JDK :



getClass

public final Class getClass()

Returns the runtime class of an object. That Class object is the object that is locked by static synchronized methods of the represented class.

Returns:

the object of type Class that represents the runtime class of the object.

Il ne faut pas utiliser ce tag pour des méthodes ne possédant pas de valeur de retour (void).

Exemple:

@return le nombre d'occurrences contenues dans la collection

@return <code>true</code> si les traitements sont correctement exécutés sinon <code>false</code>

97.2.6. Le tag @see

Le tag @see permet de définir un renvoi vers une autre entité incluse dans une documentation de type Javadoc ou vers une url.

La syntaxe de ce tag est la suivante :

@see référence à une entité suivie d'un libellé optionnel ou lien ou texte entre double quote

@see package
@see package.Class
@see class
@see #champ
@see class#champ
@see #method(Type,Type,...)
@see class#method(Type,Type,...)
@see package.class#method(Type,Type,...)
@see <a href="..."> ... </a>
@see " ... "

Le tag génère un lien vers une entité ayant un lien avec celle documentée.

Il peut y avoir plusieurs tags @see dans un même commentaire.

L'entité vers laquelle se fait le renvoi peut être un package, une classe, une méthode ou un lien vers une page de la documentation. Le nom de la classe doit être de préférence pleinement qualifié.

Le caractère # permet de séparer une classe d'un de ses membres (champ, constructeur ou méthode). Attention : il ne faut surtout pas utiliser le caractère "." comme séparateur entre une classe ou une interface et le membre précisé.

Pour indiquer une version surchargée particulière d'une méthode ou d'un constructeur, il suffit de préciser la liste des types d'arguments de la version concernée.

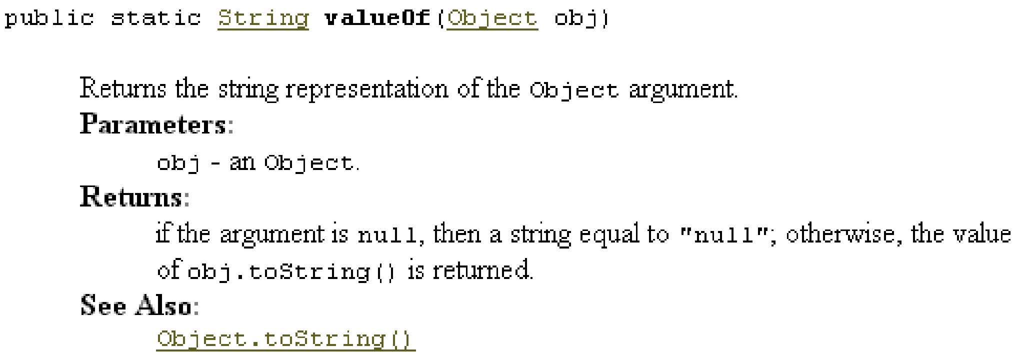
Il est possible de fournir un libellé optionnel à la suite de l'entité. Ce libellé sera utilisé comme libellé du lien généré : ceci est pratique pour forcer un libellé à la place de celui généré automatiquement (par défaut le nom de l'entité).

Si le tag est suivi d'un texte entre double cote, le texte est simplement repris avec les cotes sans lien.

Si le tag est suivi d'un tag HTML <a>, le lien proposé par ce tag est repris intégralement.

Le doclet standard crée une section "See Also" qui regroupe les tags @see du commentaire en les séparant par une virgule et un espace.

Exemple extrait de la documentation de l'API du JDK :



public static String **valueOf**(Object obj)

Returns the string representation of the `Object` argument.

Parameters:

`obj` - an `Object`.

Returns:

if the argument is `null`, then a string equal to `"null"`; otherwise, the value of `obj.toString()` is returned.

See Also:

`Object.toString()`

Remarque : pour insérer un lien n'importe où dans le commentaire, il faut utiliser le tag {@link}

Exemple :

@see String
@see java.lang.String
@see String#equals
@see java.lang.Object#wait(int)
@see MaClasse nouvelle classe
@see <a href="test.htm">Test</a>
@see "Le dossier de spécification détaillée"

Ce tag permet de définir des liens vers d'autres éléments de l'API.

97.2.7. Le tag @since

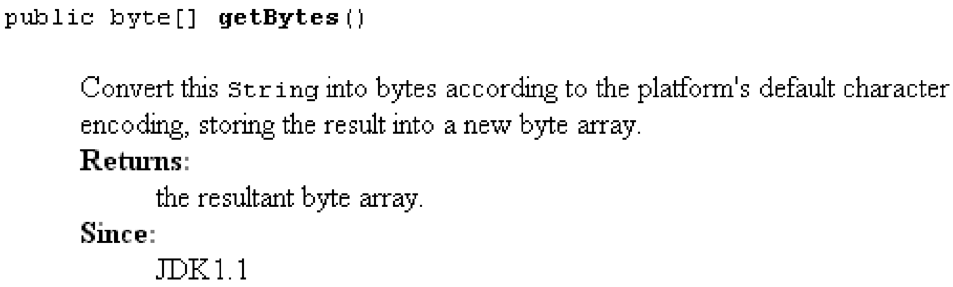
Le tag @since permet de préciser un numéro de version de la classe ou de l'interface à partir de laquelle l'élément décrit est disponible. Ce tag peut être utilisé avec tous les éléments.

La syntaxe de ce tag est la suivante :

@since texte

Le texte qui représente le numéro de version est libre. Le doclet standard crée une section "Since" qui contient le texte du tag.

Exemple extrait de la documentation de l'API du JDK :



public byte[] **getBytes**()

Convert this `String` into bytes according to the platform's default character encoding, storing the result into a new byte array.

Returns:

the resultant byte array.

Since:

JDK1.1

Par convention, pour limiter le nombre de sections Since dans la documentation, lorsqu'une nouvelle classe ou interface est ajoutée, il est préférable de mettre un tag @since sur le commentaire de la classe et de ne pas le reporter sur chacun de ses membres. Le tag @since est utilisé sur un membre uniquement lors de l'ajout du membre.

Dans la documentation de l'API Java, ce tag précise depuis qu'elle version du JDK l'entité décrite est utilisable.

Exemple :

@since 2.0

97.2.8. Le tag @version

Le tag @version permet de préciser un numéro de version. Ce tag doit être utilisé uniquement pour un élément de type classe ou interface.

La syntaxe de ce tag est la suivante :

@version texte

Le texte qui suit la balise est libre : il devrait correspondre à la version courante de l'entité documentée. Le doclet standard crée une section "Version" qui contient le texte du tag.

Il ne devrait y avoir qu'un seul tag @version dans un commentaire.

Par défaut, le doclet standard ne prend pas en compte ce tag : il est nécessaire de demander sa prise en compte avec l'option -version de la commande javadoc.

Exemple :

@version 1.00

97.2.9. Le tag {@link}

Ce tag permet de créer un lien vers un autre élément de la documentation.

La syntaxe de ce tag est la suivante :

{**@link** package.class#membre texte }

Le mode de fonctionnement de ce tag est similaire au tag @see : la différence est que le tag @see crée avec le doclet standard un lien dans la section "See also" alors que le tag {@link} crée un lien à n'importe quel endroit de la documentation.

Si une accolade fermante doit être utilisée dans le texte du tag il faut utiliser la séquence d'échappement &#125;.

Exemple :

Utiliser la {@link #maMethode(int) nouvelle méthode}

97.2.10. Le tag {@value}

Ce tag permet d'afficher la valeur d'un champ.

La syntaxe de ce tag est la suivante :

{@value}

{@value package.classe#champ_static}

Lorsque le tag {@value} est utilisé sans argument avec un champ static, le tag est remplacé par la valeur du champ.

Lorsque le tag {@value} est utilisé avec comme argument une référence à un champ static, le tag est remplacé par la valeur du champ précisé. La référence utilisée avec ce tag suit la même forme que celle du tag @see

Exemple :

{@value}

{@value #MA_CONSTANTE}

97.2.11. Le tag {@literal}

Ce tag permet d'afficher un texte qui ne sera pas interprété comme de l'HTML.

La syntaxe de ce tag est la suivante :

{**@literal** texte}

Le contenu du texte est repris intégralement sans interprétation. Notamment les caractères < et > ne sont pas interprétés comme des tags HTML.

Pour afficher du code, il est préférable d'utiliser le tag {@code}

Exemple :

{@literal 0<b>10}

97.2.12. Le tag {@linkplain}

Ce tag permet de créer un lien vers un autre élément de la documentation dans une police normale.

Ce tag est similaire au tag @link. La différence réside dans la police d'affichage.

97.2.13. Le tag {@inheritDoc}

Ce tag permet de demander explicitement la recopie de la documentation de l'entité de la classe mère la plus proche correspondante.

La syntaxe de ce tag est la suivante:

{@inheritDoc}

Ce tag permet d'éviter le copier/coller de la documentation d'une entité.

Il peut être utilisé :

- dans la description d'une entité : dans ce cas tout le commentaire de l'entité de la classe mère est repris
- dans un tag @return, @tag, @throws : dans ce cas tout le texte du tag de l'entité de la classe mère est repris

97.2.14. Le tag {@docRoot}

Ce tag représente le chemin relatif à la documentation générée.

La syntaxe de ce tag est la suivante :

{@docRoot}

Ce tag est pratique pour permettre l'inclusion de fichiers dans la documentation.

Exemple :

<a href="{@docRoot}/historique.htm">Historique</a>

97.2.15. Le tag {@code}

Ce tag permet d'afficher un texte dans des tags <code> ... </code> qui ne sera pas interprété comme de l'HTML.

La syntaxe de ce tag est la suivante :

{@code texte}

Le contenu du texte est repris intégralement sans interprétation. Notamment les caractères < et > ne sont pas interprétés comme des tags HTML.

Le tag {@code texte} est équivalent à <code>{@literal texte}</code>

Exemple :

{@code 0<b>10}

97.2.16. Le tag @snippet

Il est pratique d'inclure des fragments de code source dans les commentaires de documentation pour illustrer des cas d'utilisation.

Historiquement, cela se fait avec un tag {@code ...} ou <pre>{@code ...}</pre>.

Exemple (code Java 5.0) :

```
/**
 * Point d'entrée de l'application.
 *
 * Le code invoque l'instruction :
 * <pre>{@code
 *   System.out.println("Hello World");
 * }</pre>
 */
public static void main(String[] args) {
   // ...
}
```

Le Doclet standard génère de l'HTML qui reflète précisément le corps du tag {@code ...}, y compris l'indentation sans valider le code.

Le but de la JEP 413, ajoutée en Java 18, est de faciliter l'inclusion de fragments d'exemples de code dans la Javadoc. La JEP 413 ajoute une fonctionnalité à l'outil JavaDoc pour améliorer la prise en charge des exemples de code dans la documentation des API en utilisant le tag @snippet.

Le tag {@snippet ...} remplace les précédentes techniques de manière plus pratique et offre plus de possibilités et de flexibilité.

Le tag @snippet permet de définir un fragment de code qui sera inclus dans la documentation générée. Ce fragment peut être en ligne (inclus dans le tag lui-même) ou externe (lu à partir d'un fichier source).

Dans un fragment de code, il est possible d'utiliser des tags de marquage dans des commentaires de marquages pour formatter ou remplacer des portions de texte (@highlight et @replace) ou lier une portion texte à d'autres éléments (@link).

97.2.16.1. Les attributs

La configuration d'un fragment se fait grâce à des attributs sous la forme de paires nom=valeur qui suivent le tag @snippet, chacune séparée de la précédente par un caractère d'espacement (espace, tabulation, retour chariot, ...).

Le nom d'un attribut est un simple identifiant. Une valeur peut être entourée d'une paire de simple ou double quote. Les séquences d'échappement ne sont pas supportées dans les valeurs.

L'attribut id permet de définir un identifiant au fragment utilisable via l'API et dans le code HTML généré. Le Doclet standard n'utilise pas directement cet attribut.

Exemple (code Java 18) :

```
/**
 * {@snippet id="main" :
 *    public static void main(String[] args) {
 *      System.out.println("Hello World");
 *   }
 * }
 */
```

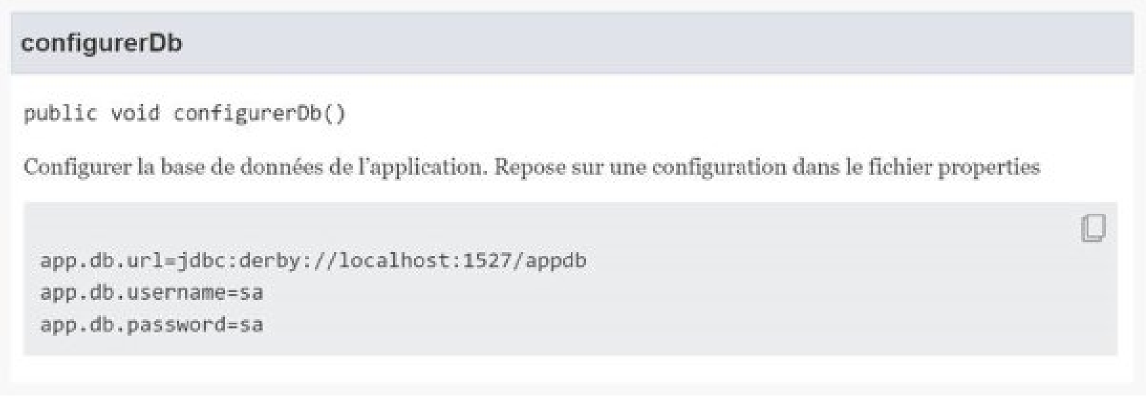
L'attribut lang permet de préciser le type de contenu du fragment (source Java ou d'un autre langage, properties, XML, Json, texte, ...). Pour un fragment en ligne (in line), la valeur par défaut est java. Pour un fragment externe, la valeur est déterminée en fonction de l'extension du fichier.

Le Doclet standard de Java 18 supporte les valeurs java et properties.

Exemple (code Java 18) :

```
/**
 * Configurer la base de données de l'application.
 * Repose sur une configuration dans le fichier properties
 * {@snippet lang="properties" :
 * app.db.url=jdbc:derby://localhost:1527/appdb
 * app.db.username=sa
 * app.db.password=sa
 * }
 */
```

La sortie générée est :



configurerDb

```
public void configurerDb()
```

Configurer la base de données de l'application. Repose sur une configuration dans le fichier properties

```
app.db.url=jdbc:derby://localhost:1527/appdb
app.db.username=sa
app.db.password=sa
```

Trois autres attributs peuvent aussi être utilisés avec le tag @snippet :

- region : le nom d'une région définie dans le fragment externe
- class : le nom d'une classe dont les sources seront le fragment
- file : le nom d'un fichier dont le contenu sera le fragment

97.2.16.2. Les fragments en ligne

Les fragments en ligne sont inclus dans le tag @snippet lui-même.

Le contenu du fragment, qui est inclus dans la documentation générée, est le texte compris entre le saut de ligne après les deux points « : » et l'accolade fermante « } ».

Dans sa forme la plus simple, le tag {@snippet ...} peut être utilisé pour contenir un fragment de texte, comme du code source ou toute autre forme de texte structuré.

Exemple (code Java 18) :

```
/**
 * Méthode de test
 * {@snippet id="main" :
 *    public static void main(String[] args) {
 *      System.out.println("Hello World");
 *   }
 * }
 */
```

Il n'est pas nécessaire d'échapper les caractères tels que <, > et & avec des entités HTML. Il n'est pas nécessaire non plus d'échapper les tags de commentaires de la documentation.

97.2.16.2.1. La gestion de l'indentation

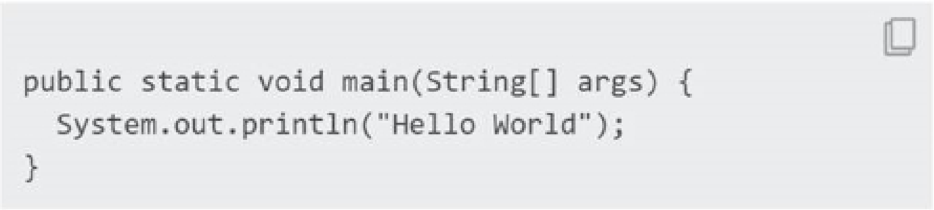
Les caractères d'espacement en tête sont supprimés du contenu à l'aide de la méthode stripIndent() de la classe String. Cela permet de remédier à un inconvénient des blocs <pre>{@code ...}</pre> qui imposait que le texte à afficher commence toujours immédiatement après les espaces et les astérisques.

Dans les tags @snippet, l'indentation dans la sortie générée est l'indentation relative à la position de l'accolade fermante dans le fichier source. Ceci est similaire à l'indentation d'un bloc de texte par rapport à la position du délimiteur de fin """.

Cela permet de contrôler l'indentation dans la sortie générée en ajustant la position de la dernière parenthèse fermante.

Exemple (code Java 18) :

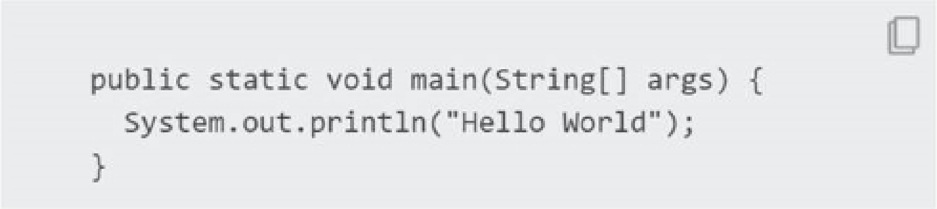
```
/**
 * {@snippet :
 *     public static void main(String[] args) {
 *       System.out.println("Hello World");
 *     }
 *     }
 */
```



```
public static void main(String[] args) {
  System.out.println("Hello World");
}
```

Exemple (code Java 18) :

```
  /**
   * {@snippet :
   *     public static void main(String[] args) {
   *       System.out.println("Hello World");
   *     }
   * }
   */
```



```
public static void main(String[] args) {
  System.out.println("Hello World");
}
```

97.2.16.2.2. Les limitations dans les fragments internes

Le contenu du fragment possède plusieurs restrictions liées au fait que le tag est inclus dans un commentaire de documentation :

- il ne peut pas contenir de séquence */, qui serait considéré comme la fin du commentaire de documentation : il n'est donc pas possible qu'il contienne un commentaire multilignes
- les paires d'accolades « { ... } » doivent être "équilibrées", ce qui implique un nombre égal d'accolades ouvrantes et fermantes correctement imbriquées, pour que l'accolade fermante du tag @snippet puisse être déterminée sans ambiguïté

97.2.16.3. Les régions

Les régions sont des portions de code dont le nom est facultatif et qui identifient le texte à utiliser par un fragment. Elles définissent également la portée des actions telles que la mise en évidence ou la modification du texte.

Le début de la définition d'une région peut se faire de plusieurs manières :

- un tag @start avec un attribut region=nom
- un tag @highlight, @replace ou @link avec un attribut region ou region=nom. Il est possible d'omettre le nom s'il n'est pas nécessaire

La fin de la définition d'une région peut se faire de plusieurs manières :

- un tag @end
- un tag @end avec l'attribut region=nom

Si un nom de région est précisé, le tag @end termine la région commencée avec ce nom. Si aucun nom n'est donné, le tag termine la région la plus proche commencée qui n'a pas déjà un tag @end correspondante.

Il n'y a aucune contrainte sur les régions créées par différentes paires de tag @start et @end correspondantes. Même si cela n'est pas recommandé, les régions peuvent même se chevaucher.

Par défaut, les commentaires de marquage ne s'appliquent que sur le contenu précédent de la même ligne. Il est parfois pratique de l'appliquer sur plusieurs lignes : dans ce cas, il faut utiliser une région.

Une région peut être anonyme ou nommée.

Pour qu'un tag s'applique à une région anonyme, il faut le placer au début de la région et utiliser un tag @end pour marquer la fin de la région.

Exemple (code Java 18) :

```
  /**
   * {@snippet :
   *  // @start region="exemple" :
   *  System.out.println("Hello World");
   *  // @end
   * }
   */
```

Il est possible d'indiquer explicitement la correspondance entre le début et la fin d'une région en attribuant un nom avec l'attribut region et en utilisant ce nom comme valeur de l'attribut region du tag @end.

Exemple (code Java 18) :

```
  /**
   * {@snippet :
   *  // @start region="exemple" :
   *  System.out.println("Hello World");
   *  // @end region="exemple"
   * }
   */
```

Si le nom de la région indiqué dans le tag @end n'est pas défini comme un nom de région valide alors une erreur est émise à la génération de la documentation.

Exemple (code Java 18) :

```
  /**
   * {@snippet :
   *  // @start region="exempledeb" :
   *  System.out.println("Hello World");
   *  // @end region="exemplefin"
   * }
   */
```

Résultat :

```
D:\java18\src\main\java\fr\jmdoudoux\dej\java18\App.java:145:
error: snippet markup: unpaired region
   *  // @end region="exemplefin"
              ^
```

Nommer une région n'a aucune incidence sur le contenu généré : sa seule utilité est la définition de la région et son identification.

Les régions peuvent être imbriquées. Les régions imbriquées ne doivent pas nécessairement être nommées, mais l'utilisation de régions nommées apporte plus de clarté.

Les régions peuvent se chevaucher : il faut alors utiliser des régions nommées pour faciliter la définition des régions.

97.2.16.4. Les fragments externes

Les fragments externes sont stockés dans un fichier externe. Contrairement aux fragments en ligne, les fragments externes n'ont pas de restrictions : ils peuvent contenir des commentaires multilignes par exemple.

Le fichier externe contenant le fragment peut être précisé de deux manières :

- avec l'attribut class dont la valeur est le nom de la classe
- avec l'attribut file dont la valeur est le chemin relatif du fichier source

L'attribut file du tag {@snippet ...} permet de préciser le chemin relatif du fichier contenant le code source qui sera ajouté dans la documentation.

Exemple (code Java 18) :

```
    /**
     * Exemple d'utilisation.
     * {@snippet file="com/jmdoudoux/dej/ExempleDoc.java"
     * }
     */
```

Exemple avec le fichier Utils.java qui contient :

Exemple (code Java 18) :

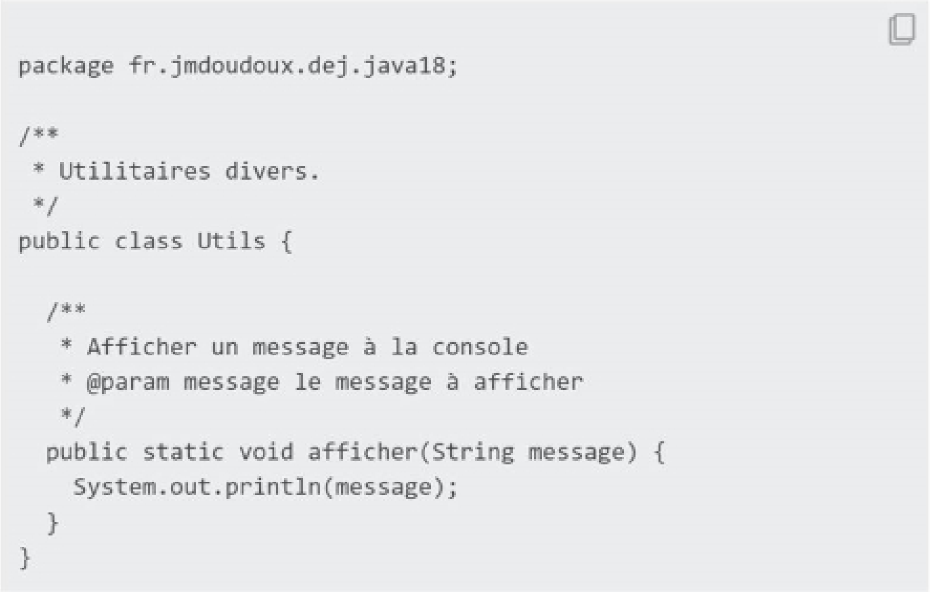
```
package fr.jmdoudoux.dej.java18;

/**
 * Utilitaires divers.
 */
public class Utils {

  /**
   * Afficher un message à la console
   * @param message le message à afficher
   */
  public static void afficher(String message) {
    System.out.println(message);
  }
}
```

L'attribut class du tag {@snippet ...} permet de préciser le nom de la classe contenant le code source qui sera ajouté dans la documentation. Dans un fragment externe, les deux points, le saut de ligne et le contenu suivant peuvent être omis.

Exemple (code Java 18) :

```
  /**
   * Méthode de test.
   * {@snippet class="fr.jmdoudoux.dej.java18.Utils" region="exemple"}
   */
```

Par défaut, tout le contenu du code source de la classe est ajouté dans la documentation générée.



```
package fr.jmdoudoux.dej.java18;

/**
 * Utilitaires divers.
 */
public class Utils {

  /**
   * Afficher un message à la console
   * @param message le message à afficher
   */
  public static void afficher(String message) {
    System.out.println(message);
  }
}
```

Les fichiers externes peuvent être placés :

- dans le sous-répertoire snippet-files du package de la classe documentée
- dans un sous-répertoire précisé grâce à l'option --snippet-path de l'outil javadoc

Lors de l'utilisation des attributs class et file, le fichier peut être placé dans une hiérarchie de répertoires ayant pour racine le sous-répertoire snippet-files du répertoire contenant le code source avec le tag {@snippet ...}. L'utilisation des sous-répertoires snippet-files est similaire à l'utilisation actuelle des sous-répertoires doc-files pour les fichiers de documentation auxiliaires. Les fichiers contenus dans un répertoire snippet-files peuvent être partagés entre les fragments d'un même paquet, et sont isolés des fragments des répertoires snippet-files d'autres packages.

Attention : certains outils peuvent considérer de manière erronée que le sous-répertoire snippet-files est inclus dans la hiérarchie des répertoires des packages. snippet-files n'est pas un identifiant Java valide et ne peut donc pas être utilisé dans le nom du package. Dans ce cas, il ne faut pas utiliser de sous-répertoires snippet-files.

Le fichier peut également être placé sur un chemin de recherche précisé par l'option --snippet-path de l'outil javadoc. Les fichiers du chemin de recherche auxiliaire existent dans un espace de nom partagé unique et peuvent être référencés depuis n'importe quel endroit de la documentation.

L'utilisation d'un fragment externe est parfois requise pour plusieurs besoins :

- pallier à des limitations des fragments en ligne comme l'impossibilité d'utiliser la séquence */ (des commentaires multilignes ou des expressions régulières par exemple)
- pour compiler et/ou tester des fragments de code à inclure

Il peut être pratique de valider le contenu d'un fragment en le compilant et éventuellement en le testant. Cela peut permettre d'éviter des erreurs liées à une faute de frappe ou à une évolution dans le code. Un des intérêts des fragments externes est qu'ils peuvent être compilés et testés par des outils externes.

97.2.16.4.1. L'inclusion d'une portion du fichier externe

Il est possible de n'ajouter qu'une portion du code source de la classe en définissant une région et en lui attribuant un nom. Les tags @start et @end dans des commentaires de marquage dans le fichier définissent les limites de la région.

Le fichier Utils.java contient :

Exemple (code Java 18) :

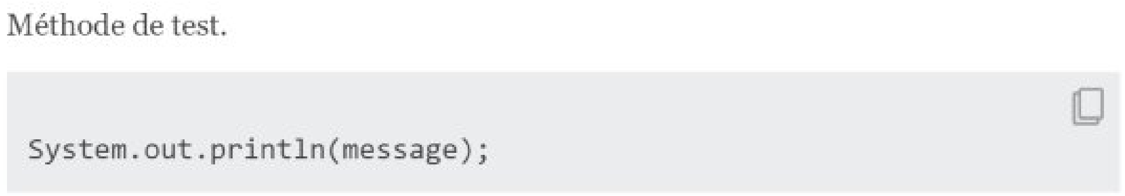
```
package fr.jmdoudoux.dej.java18;

/**
 * Utilitaires divers.
 */
public class Utils {

  /**
   * Afficher un message à la console
   * @param message le message à afficher
   */
  public static void afficher(String message) {
    // @start region="exemple" :
    System.out.println(message);
    // @end
  }
}
```

L'attribut region du tag {@snippet ...} précise le nom de la région du fichier externe à inclure.

Exemple (code Java 18) :

```
  /**
   * Méthode de test.
   * {@snippet class="fr.jmdoudoux.dej.java18.Utils" region="exemple"}
  */
```

La documentation générée contient :



Méthode de test.

```
System.out.println(message);
```

Un fichier externe peut contenir plusieurs régions avec des noms différents qui pourront être utilisées par différents tags @snippet.

Il est aussi possible de mélanger les régions au sein d'un fichier source externe, des régions utilisables pour définir les parties du fichier référençables dans un tag @snippet, et des régions utilisables avec des tags de marquage pour mettre en évidence ou modifier le texte à inclure dans la documentation générée.

97.2.16.5. Les fragments externes non Java

Les fragments externes ne sont pas restreints à n'être que du code source Java.

L'attribut file du tag @snippet permet de préciser le chemin d'un fichier texte relatif au sous-répertoire snippet-files ou au sous-répertoire précisé avec l'option --snippet-path de la commande javadoc.

Exemple (code Java 18) :

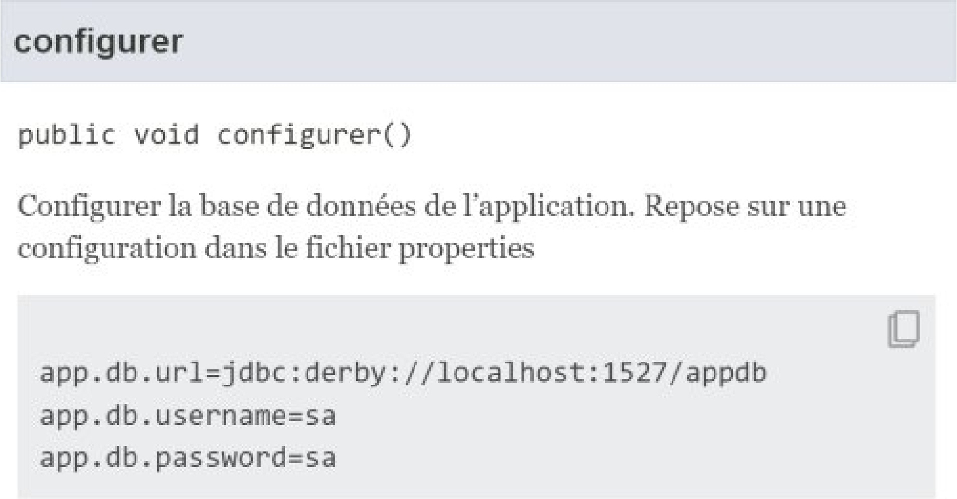
```
  /**
   * Configurer la base de données de l'application.
   * Repose sur une configuration via le fichier properties
   * {@snippet file=app-config.properties region=db }
   */
  public void configurer() {
  }
```

Avec le fichier app-config.properties qui contient :

Résultat :

```
app.titre=Mon application
# @start region=db
app.db.url=jdbc:derby://localhost:1527/db
app.db.username=root
app.db.password=root
# @end region=db
app.cache=true
```

La documentation générée contient :



configurer

```
public void configurer()
```

Configurer la base de données de l'application. Repose sur une configuration dans le fichier properties

```
app.db.url=jdbc:derby://localhost:1527/appdb
app.db.username=sa
app.db.password=sa
```

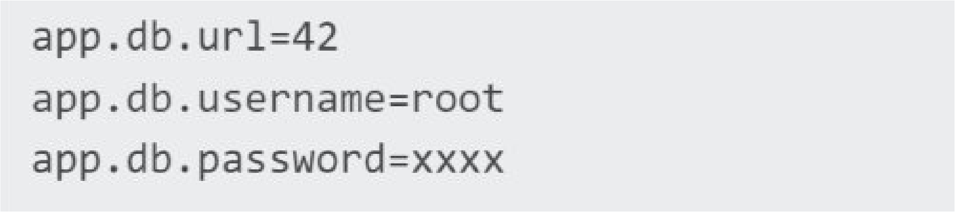
97.2.16.5.1. Le support des fichiers properties

Dans un fichier de propriétés, les commentaires de marquage utilisent la syntaxe de commentaires pour ces fichiers : les lignes commençant par un caractère dièse « # ».

Comme l'étendue par défaut des commentaires de marquage est la ligne courante, et que les fichiers de propriétés ne permettent pas de placer des commentaires sur la même ligne que le contenu sans commentaire, il faut utiliser la forme de commentaire de marquage qui se termine par « : » afin que le commentaire de marquage soit traité comme s'appliquant à la ligne suivante.

Résultat :

```
app.titre=Mon application
# @start region=db
app.db.url=jdbc:derby://localhost:1527/db
app.db.username=root
# @replace substring="root" replacement="xxxx" :
app.db.password=root
# @end region=db
app.cache=true
```



```
app.db.url=42
app.db.username=root
app.db.password=xxxx
```

97.2.16.6. Les fragments hybrides

Les fragments en ligne sont pratiques à utiliser, surtout pour de courts exemples, car ils permettent de voir le contenu du fragment dans le contexte du commentaire qui le contient.

Les fragments externes sont pratiques à utiliser car ils peuvent être compilés et même testés.

Un fragment hybride est à la fois un fragment interne et un fragment externe. Il contient le contenu du fragment dans le tag lui-même, pour la commodité de lecture du code source de la classe documentée, et il fait également référence à un fichier séparé qui contient le contenu du fragment.

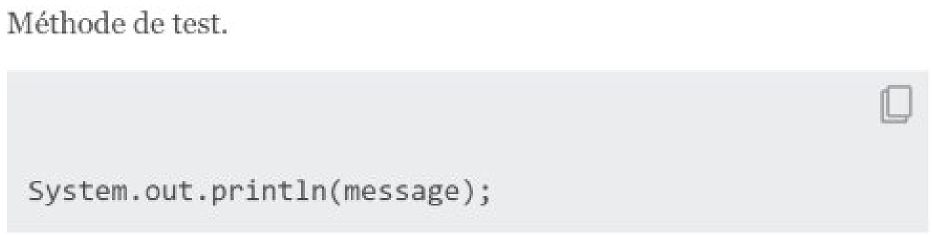
Une erreur se produit lors du traitement d'un extrait hybride si le contenu en ligne ne correspond au contenu du fragment externe.

Les fragments hybrides offrent le meilleur des deux styles mais avec quelques contraintes.

Un fragment hybride est une combinaison d'un fragment en ligne et d'un fragment externe : il a un contenu en ligne et les attributs pour spécifier un fichier externe et éventuellement une région dans ce fichier.

Exemple (code Java 18) :

```
        /**
         * Afficher un message à la console
         * @param message le message à afficher
         */
        public static void afficher(String message) {
          // @start region="exemple" :
          System.out.println(message);
          // @end
        }
```



Méthode de test.

```
System.out.println(message);
```

Avec le fichier Utils.java qui contient :

Exemple (code Java 18) :

```
package fr.jmdoudoux.dej.java18;

/**
 * Utilitaires divers.
*/
public class Utils {

  /**
   * Afficher un message à la console
   * @param message le message à afficher
   */
  public static void afficher(String message) {
   // @start region="exemple" :
   System.out.println(message);
   // @end
  }
}
```

Pour éviter une désynchronisation entre les deux contenus, le Doclet standard vérifie que le résultat du traitement du tag @snippet en tant que fragment en ligne est le même que celui du traitement en tant que fragment externe. Si ce n'est pas le cas alors une erreur est émise à la génération de la documentation.

Exemple (code Java 18) :

```
  /**
   * Méthode de test.
   * {@snippet file="fr/jmdoudoux/dej/java18/Utils.java" region="exemple"  :
   *
   *System.out.println(message);
   *
   *}
   */
```

Résultat :

```
D:\java18\src\main\java\fr\jmdoudoux\dej\java18\App.java:27:
error: contents mismatch:
         * {@snippet file="fr/jmdoudoux/dej/java18/Utils.java" region="exemple"  :
           ^
  ----------------- inline -------------------

  System.out.println(message);


  ----------------- external -----------------
  System.out.println(message);
```

97.2.16.7. Les tags de marquage

Les tags de marquage (markup tags) permettent de réaliser différentes actions selon le tag utilisé :

- mise en évidence de portion de code : @highlight
- remplacement de texte : @replace
- créer des liens vers d'autres parties de la documentation : @link

Ces tags peuvent être utilisés dans des fragments internes, externes et hybrides.

Ils s'appliquent sur une ligne ou sur une région dans le contenu d'un fragment.

Ils peuvent avoir des attributs en fonction des besoins.

Ils doivent être placé dans un commentaire pour ne pas interférer avec le code source : par exemple, après une séquence // pour du code Java ou # pour un fichier properties. Ces commentaires sont nommés commentaires de marquage (markup comments).

Plusieurs tags peuvent être inclus dans un même commentaire de marquage.

Les commentaires de fin de ligne sont pratiques à utiliser comme commentaires de marquage mais ils peuvent présenter certaines limitations :

- tous les langages ne proposent pas un support des commentaires de fin de ligne par exemple les fichiers properties
- il peut y avoir des contraintes sur l'utilisation de ces commentaires comme la taille de la ligne par exemple ou l'impossibilité d'utiliser de tels commentaires dans un bloc de texte

Pour contourner ces limitations, il y a une syntaxe spéciale pour les commentaires de marquage : si le commentaire de marquage se termine par un caractère deux points « : », il est traité comme s'il s'agissait d'un commentaire de fin de ligne sur la ligne suivante.

Ainsi, les tags de marquage s'appliquent à la ligne de source contenant le commentaire, sauf si le commentaire se termine par deux points « : », auquel cas les tags de marquage s'appliquent uniquement à la ligne suivante. Cette syntaxe peut être utile si le commentaire de marquage est particulièrement long, ou si le format syntaxique du contenu d'un extrait ne permet pas aux commentaires d'apparaître sur la même ligne que la source sans commentaire.

Les commentaires de marquage n'apparaissent pas dans la sortie générée.

Comme certains langages utilisent des méta-commentaires similaires aux commentaires de marquage, les commentaires qui commencent par @ suivi d'un nom non reconnu sont ignorés. Si le nom est reconnu, mais que des erreurs sont présentes dans le commentaire de marquage, une erreur est signalée. Dans ce cas, la sortie générée est indéfinie par rapport à la sortie générée à partir du snippet.

Un fragment peut contenir des commentaires de marquage, qui peuvent être utilisés pour modifier ce qui est affiché dans la sortie générée. Les commentaires de marquage sont des commentaires de fin de ligne dans le langage déclaré pour le fragment.

Les tags de marquage sont généralement de la forme @nom [arguments] ... La plupart des arguments sont des paires nom=valeur, auquel cas les valeurs ont la même syntaxe que celle des attributs du tag @snippet.

97.2.16.7.1. La mise en évidence (Highlighting)

Le tag de marquage @highlight permet de mettre en évidence une portion de texte sur une ou plusieurs lignes.

Plusieurs attributs permettent de préciser l'étendue du texte à prendre en compte, le texte à mettre en évidence dans cette étendue et le type de mise en évidence :

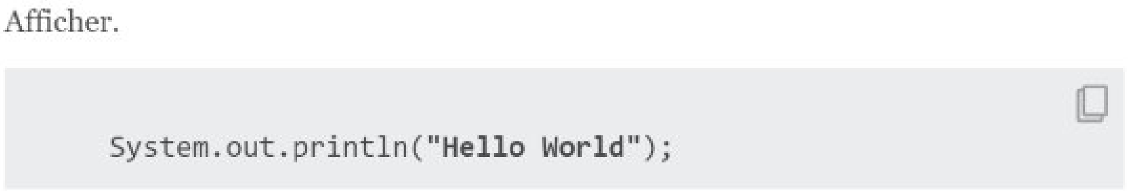
Attribut	Rôle
region region=nom	Préciser que la portée est la région indiquée jusqu'au tag @end correspondant
substring=chaine	Préciser la portion de chaîne de caractères littérale à mettre en évidence dans la portée. Le texte peut être entouré par une paire de simple ou double quotes
regex=chaine	Préciser une expression régulière pour désigner les portions de texte qui la respectent dans la portée
type=nom	Préciser le type de mise en évidence à utiliser. Trois valeurs sont utilisables, chacune correspondant à une classe CSS qui peuvent être redéfinie : • bold • italic • highlighted

Si l'attribut region n'est pas précisé alors l'étendue est juste la ligne courante ou la ligne suivante si le commentaire se termine un caractère « : ».

Exemple (code Java 18) :

```
  /**
   * Afficher.
   * {@snippet :
   *     // @highlight substring='"Hello World"' :
   *     System.out.println("Hello World");
   *}
   */
```

La documentation générée contient :



Afficher.

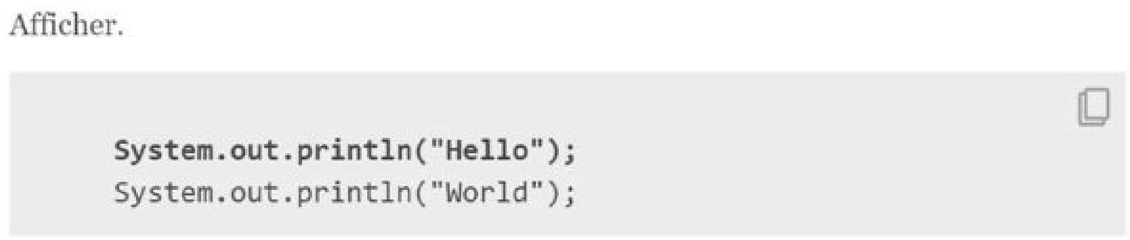
```
System.out.println("Hello World");
```

Si ni l'attribut substring ni l'attribut regex ne sont précisés alors l'intégralité de la ligne ou de la région est mise en évidence.

Exemple (code Java 18) :

```
  /**
   * Afficher.
   * {@snippet :
   *     // @highlight :
   *     System.out.println("Hello");
   *     System.out.println("World");
   *}
   */
```

La documentation générée contient :


Afficher.

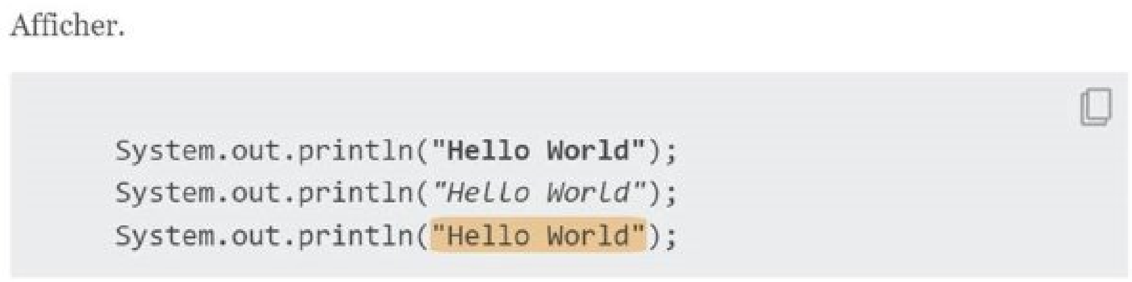
```
System.out.println("Hello");
System.out.println("World");
```

L'attribut type permet de préciser le type de mise en évidence à utiliser.

Exemple (code Java 18) :

```
/**
 * Afficher.
 * {@snippet :
 *     // @highlight substring='"Hello World"' type="bold" :
 *     System.out.println("Hello World");
 *     // @highlight substring='"Hello World"' type="italic" :
 *     System.out.println("Hello World");
 *     // @highlight substring='"Hello World"' type="highlighted" :
 *     System.out.println("Hello World");
 *}
 */
```

La documentation générée contient :


Afficher.

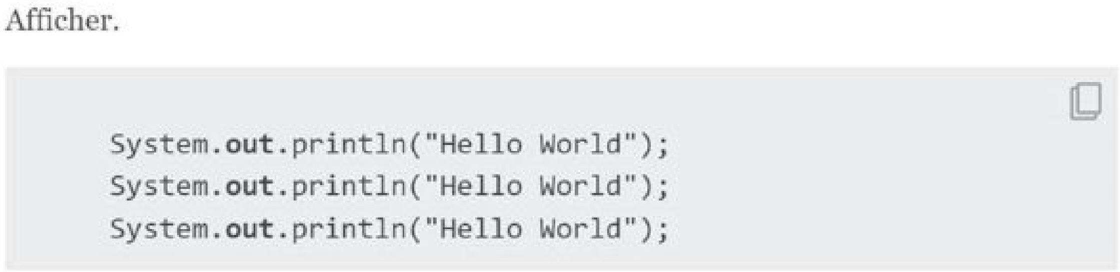
```
System.out.println("Hello World");
System.out.println("Hello World");
System.out.println("Hello World");
```

Pour appliquer la mise en évidence sur plusieurs lignes, il faut utiliser l'attribut region pour délimiter la première ligne concernée et utiliser le tag de marquage @end dans un commentaire de marquage pour indiquer la dernière ligne.

Exemple (code Java 18) :

```
/**
 * Afficher.
 * {@snippet :
 *     // @highlight substring="out" region
 *     System.out.println("Hello World");
 *     System.out.println("Hello World");
 *     System.out.println("Hello World");
 *     // @end
 *}
 */
```

La documentation générée contient :



Afficher.

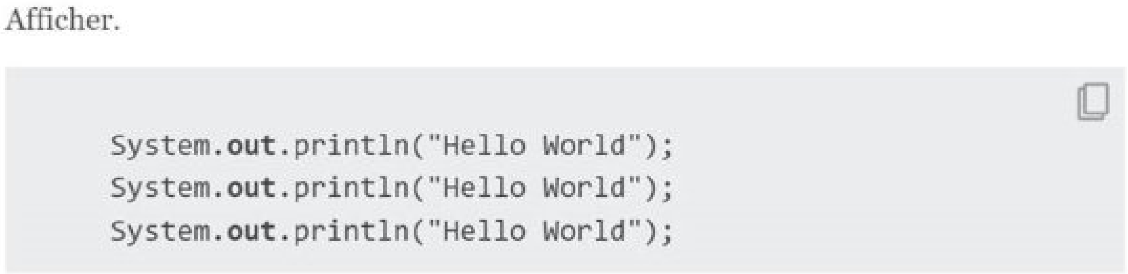
```
System.out.println("Hello World");
System.out.println("Hello World");
System.out.println("Hello World");
```

Les portions à mettre évidence peuvent aussi être définies en utilisant une expression régulière grâce à l'attribut regex.

Exemple (code Java 18) :

```
  /**
   * Afficher.
   * {@snippet :
   *     // @highlight region regex = "\bout\b"
   *     System.out.println("Hello World");
   *     System.out.println("Hello World");
   *     System.out.println("Hello World");
   *     // @end
   *}
   */
```

La documentation générée contient :



Afficher.

```
System.out.println("Hello World");
System.out.println("Hello World");
System.out.println("Hello World");
```

Si les attributs substring et regex sont utilisés simultanément alors une erreur est émise.

Exemple (code Java 18) :

```
  /**
   * Afficher.
   * {@snippet :
   *     // @highlight substring="out" region regex = "\bout\b"
   *     System.out.println("Hello World");
   *     System.out.println("Hello World");
   *     System.out.println("Hello World");
   *     // @end
   *}
   */
```

Résultat :

```
D:\java18\src\main\java\fr\jmdoudoux\dej\java18\App.java:62:
error: snippet markup: attributes "substring" and "regex" used simultaneously
   *     // @highlight substring="out" region regex = "\out\b"
                       ^
```

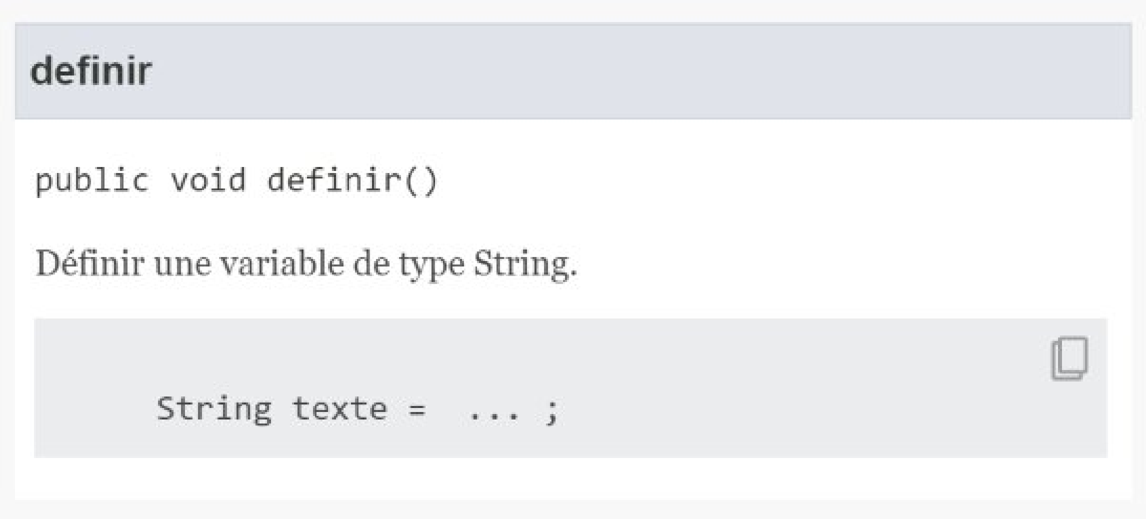
97.2.16.7.2. La modification du texte du fragment

Il peut être pratique d'écrire le contenu un fragment contenant du code validable par des outils externes, mais de l'afficher sous une forme différente qui ne compile pas. Par exemple, on peut vouloir afficher du code avec une ellipse ou un autre marqueur pour indiquer que du code supplémentaire ou différent doit être utilisé à cet endroit. Cela peut se faire en remplaçant des portions du contenu du fragment par du texte de remplacement.

Exemple (code Java 18) :

```
/**
 * Définir une variable de type String.
 * {@snippet :
 *     // @replace substring='""' replacement=" ... " :
 *     String texte = "";
 *}
 */
public void definir() {
}
```

La documentation générée contient :



definir

```
public void definir()
```

Définir une variable de type String.

```
     String texte =  ... ;
```

Le tag de marquage @replace permet de modifier le texte du fragment dans la sortie générée.

Plusieurs attributs permettent de préciser l'étendue du texte à prendre en compte, le texte à remplacer dans cette étendue et le texte de remplacement :

Attribut	Rôle
region region=nom	Préciser que la portée est la région indiquée jusqu'au tag @end correspondant
substring=texte	Préciser la portion de chaîne de caractères littérale à modifier dans la portée. Le texte peut être entouré par une paire de simple ou double quotes
regex=texte	Préciser une expression régulière pour désigner les portions de texte qui la respectent dans la portée
replacement=texte	Préciser le texte de remplacement. Si une expression régulière est utilisée pour spécifier le texte à remplacer, alors $number ou $name peuvent être utilisés pour substituer les groupes trouvés dans l'expression régulière, comme définis par la méthode replaceAll() de la classe String

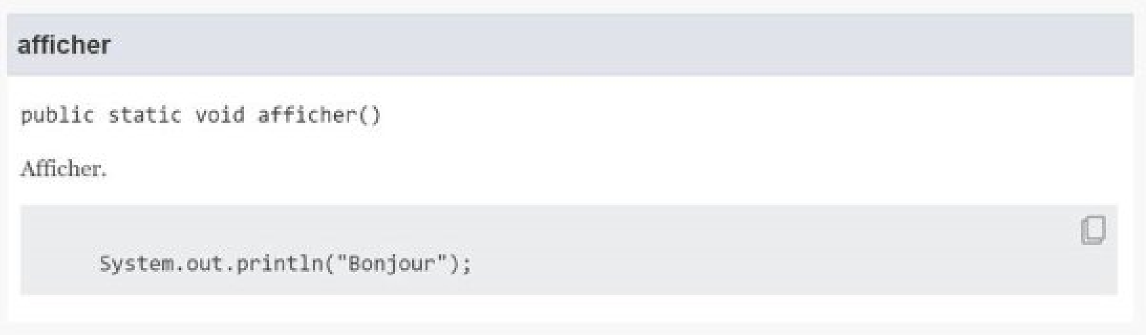
Si l'attribut region n'est pas précisé alors l'étendue est juste la ligne courante ou la ligne suivante si le commentaire se termine un caractère « : ».

Si ni l'attribut substring ni l'attribut regex ne sont précisés alors l'intégralité du fragment est remplacé.

Exemple (code Java 18) :

```
  /**
   * Afficher.
   * {@snippet :
   *     // @replace substring='"Hello World"' replacement='"Bonjour"' :
   *     System.out.println("Hello World");
   *}
   */
  public static void afficher() {
  }
```

La documentation générée contient :



afficher

```
public static void afficher()
```

Afficher.

```
    System.out.println("Bonjour");
```

Pour supprimer du texte, il suffit de mettre une chaîne de caractères vide comme valeur de l'attribut de replacement.

97.2.16.7.3. La liaison de texte avec un élément de la documentation

Le tag @link permet de lier du texte à des déclarations situées ailleurs dans la documentation.

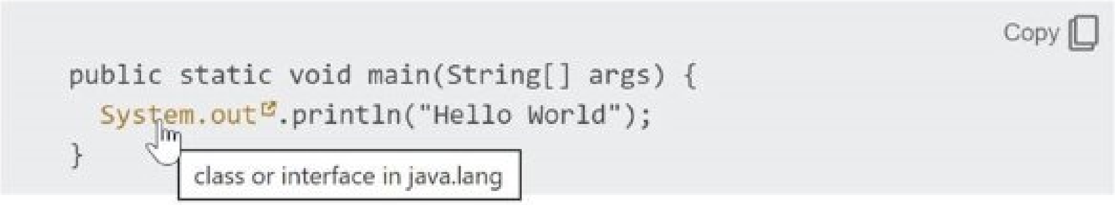
Plusieurs attributs permettent de préciser l'étendue de la portée du texte à prendre en compte, le texte à lier dans cette portée et la cible du lien :

Attribut	Rôle
region region=nom	Préciser que la portée est la région indiquée jusqu'au tag @end correspondant
substring=texte	Préciser la portion de chaîne de caractères littérale à lier dans la portée. Le texte peut être entouré par une paire de simple ou double quotes
regex=texte	Préciser une expression régulière pour désigner les portions de texte qui la respectent dans la portée
target=lien	Permettre de préciser la cible du lien. La valeur correspond à celle utilisée dans le tag @link
type=nom	Préciser le type de lien : les valeurs possibles sont link (par défaut) ou linkplain

Exemple (code Java 18) :

```
/**
 * {@snippet :
 *   public static void main(String[] args) {
 *     // @link substring="System.out" target="System#out" :
 *     System.out.println("Hello World");
 *   }
 *}
 */
```

La documentation générée contient :



```
public static void main(String[] args) {
  System.out.println("Hello World");
}
```

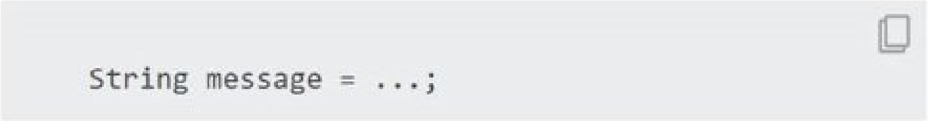
97.2.16.7.4. L'utilisation d'expressions régulières

L'attribut regex permet de préciser une expression régulière qui sera appliquée pour déterminer la portion de texte concernée.

Exemple (code Java 18) :

```
/**
 * Initialiser
 * {@snippet :
 *     String message = "Bonjour";     // @replace regex='".*"' replacement="..."
 * }
 */
```

La documentation générée contient :

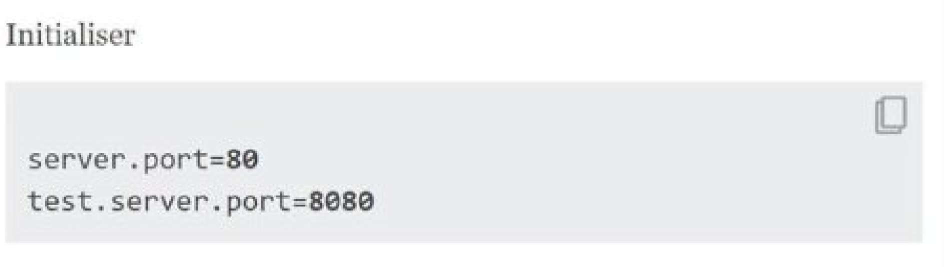


```
String message = ...;
```

Leur utilisation est parfois très pratique pour faciliter l'identification des portions à modifier.

Exemple (code Java 18) :

```
/**
 * Initialiser
 * {@snippet lang="properties" :
 * # @highlight region regex="[0-9]+":
 * server.port=80
 * test.server.port=8080
 * # @end
 * }
 */
```



Initialiser

```
server.port=80
test.server.port=8080
```

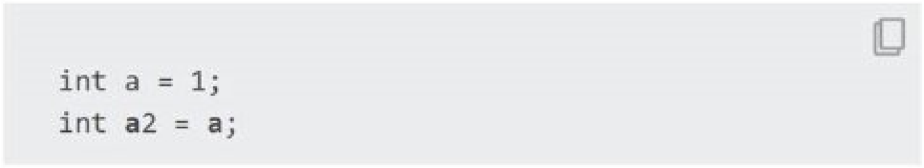
L'utilisation d'expressions régulières peut cependant s'avérer délicate pour identifier une portion spécifique d'une chaîne de caractères dans une ligne ou une région.

Exemple (code Java 18) :

```
/**
 * {@snippet :
```

```
 *   int a = 1;
 *   int a2 = a;      // @highlight regex='a'
 * }
 */
```

La documentation générée contient :



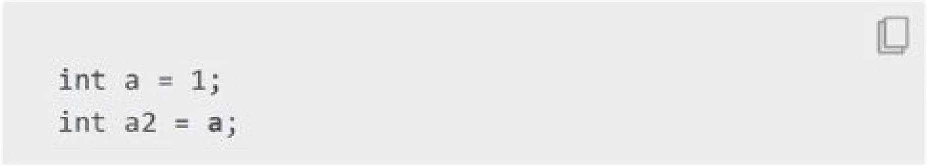
```
int a = 1;
int a2 = a;
```

Dans l'exemple précédent toutes les instances de « a » sont mises en évidence. Pour mettre en évidence uniquement la seconde instance, il faut une expression régulière qui utilise un boundary matcher.

Exemple (code Java 18) :

```
/**
 * {@snippet :
 *   int a = 1;
 *   int a2 = a;      // @highlight regex='a\b'
 * }
 */
```

La documentation générée contient :



```
int a = 1;
int a2 = a;
```

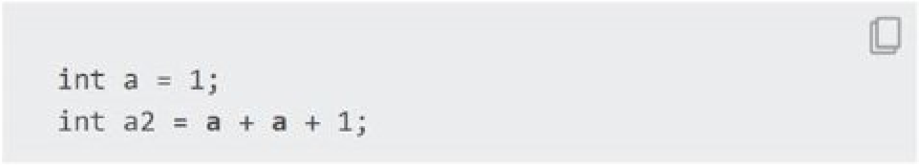
L'expression régulière de l'exemple ci-dessus utilise un word boundary pour identifier une chaîne qui est une sous-chaîne d'une autre chaîne située précédemment sur la ligne.

Il est aussi possible d'utiliser une expression régulière avec un lookahead ou un lookbehind pour identifier certaines séquences.

Exemple (code Java 18) :

```
/**
 * {@snippet :
 *   int a = 1;
 *   int a2 = a + a + 1;      // @highlight regex='a(?= \+)'
 * }
 */
```

La documentation générée contient :



```
int a = 1;
int a2 = a + a + 1;
```

Si la valeur de l'expression régulière n'est pas valide alors une erreur est émise lors de la génération de la documentation.

Exemple (code Java 18) :

```
  /**
   * {@snippet :
   *   int a = 1;
   *   int a2 = a + a + 1;      // @highlight regex='a((?= \+)'
   * }
   */
```

Résultat :

```
D:\java18\src\main\java\fr\jmdoudoux\dej\java18\App.java:154:
error: snippet markup: invalid regex
   *   int a2 = a + a + 1;      // @highlight regex='a((?= \+)'
                                                              ^
```

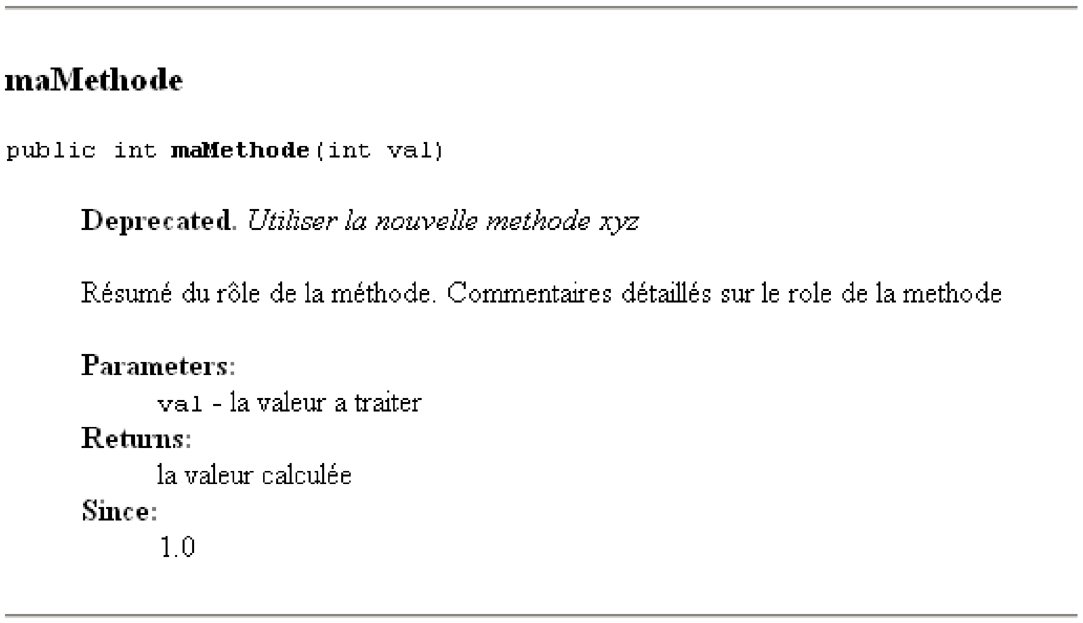
Il est recommandé de vérifier la documentation générée lors de l'utilisation d'une expression régulière afin de s'assurer que le rendu est bien celui attendu.

97.3. Un exemple

Exemple :

```
 /**
  * Résumé du rôle de la méthode.
  * Commentaires détaillés sur le role de la methode
  * @param val la valeur a traiter
  * @return la valeur calculée
  * @since 1.0
  * @deprecated Utiliser la nouvelle methode xyz
  */
   public int maMethode(int val) {
     return 0;
   }
```

Résultat :



maMethode

`public int **maMethode**(int val)`

Deprecated. *Utiliser la nouvelle methode xyz*

Résumé du rôle de la méthode. Commentaires détaillés sur le role de la methode

Parameters:
`val` - la valeur a traiter
Returns:
la valeur calculée
Since:
1.0

97.4. Les fichiers pour enrichir la documentation des packages

Javadoc permet de fournir un moyen de documenter les packages car ceux-ci ne disposent pas de code source particulier : il faut définir des fichiers dont le nom est particulier.

Ces fichiers doivent être placés dans le répertoire désigné par le package.

Le fichier package.html contient une description du package au format HTML. En plus, il est possible d'utiliser les tags @deprecated, @link, @see et @since.

Le fichier overview.html permet de fournir un résumé de plusieurs packages au format html. Ce fichier doit être placé dans le répertoire qui inclut les packages décrits.

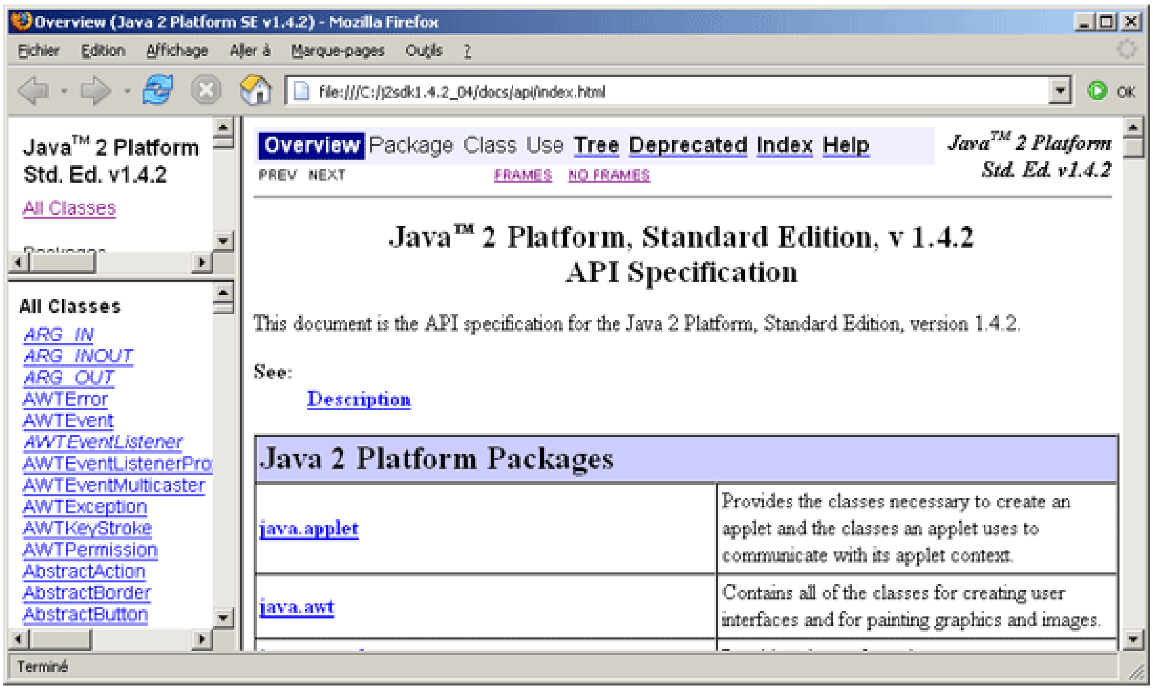
97.5. La documentation générée

Pour générer la documentation, il faut invoquer l'outil javadoc. Javadoc recrée à chaque utilisation la totalité de la documentation.

Pour formater la documentation, javadoc utilise une doclet. Une doclet permet de préciser le format de la documentation générée. Par défaut, Javadoc propose une doclet qui génère une documentation au format HTML. Il est possible de définir sa propre doclet pour changer le contenu ou le format de la documentation (pour par exemple, générer du RTF ou du XML).

La génération de la documentation avec le doclet par défaut crée de nombreux fichiers et des répertoires pour structurer la documentation au format HTML, avec et sans frame.

La documentation de l'API Java fournie par Sun/Oracle est réalisée grâce à Javadoc. La page principale est composée de trois frames :


Overview (Java 2 Platform SE v1.4.2) - Mozilla Firefox
Fichier Edition Affichage Aller à Marque-pages Outils ?
file:///C:/j2sdk1.4.2_04/docs/api/index.html
OK
Terminé

Java™ 2 Platform Std. Ed. v1.4.2

All Classes

Packages

All Classes

ARG_IN
ARG_INOUT
ARG_OUT
AWTError
AWTEvent
AWTEventListener
AWTEventListenerPro
AWTEventMulticaster
AWTException
AWTKeyStroke
AWTPermission
AbstractAction
AbstractBorder
AbstractButton

Overview Package Class Use Tree Deprecated Index Help

PREV NEXT FRAMES NO FRAMES

Java™ 2 Platform Std. Ed. v1.4.2

Java™ 2 Platform, Standard Edition, v 1.4.2
API Specification

This document is the API specification for the Java 2 Platform, Standard Edition, version 1.4.2.

See:
Description

Java 2 Platform Packages	
java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.

Par défaut, la documentation générée contient les éléments suivants :

- un fichier html par classe ou interface qui contient le détail de chaque élément de la classe ou de l'interface
- un fichier html par package qui contient un résumé du contenu du package
- un fichier overview-summary.html
- un fichier overview-tree.html
- un fichier deprecated-list.html
- un fichier serialized-form.html

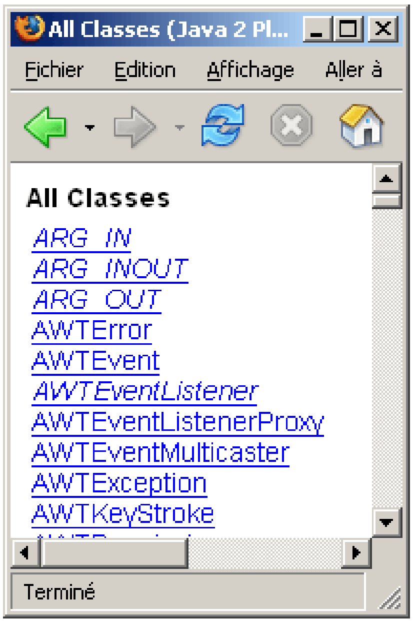
- un fichier overview-frame.html
- un fichier all-classe.html
- un fichier package-summary.html pour chaque package
- un fichier package-frame.html pour chaque package
- un fichier package-tree.html pour chaque package

Tous ces fichiers peuvent être regroupés en trois catégories :

- Les pages de base : les pages des classes et interfaces et les résumés
- Les pages des références croisées : les pages index, les pages de hiérarchie, les pages d'utilisation et les pages deprecated-list.html, constant-values.html et serialized-form.html
- Les fichiers de structure : la page principale, les frames, la feuille de style

Il y a plusieurs fichiers générés à la racine de l'application :

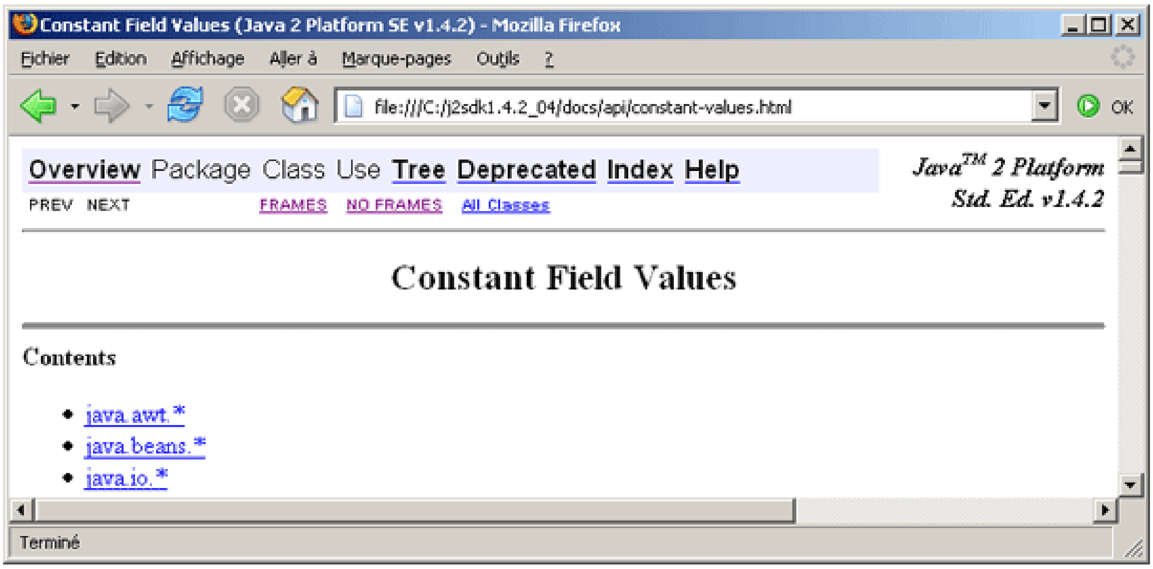
Le fichier allclasses-frame.html affiche toutes les classes, interfaces et exceptions de la documentation avec un lien pour afficher le détail. Cette page est affichée en bas à gauche dans le fichier index.html



All Classes

ARG_IN
ARG_INOUT
ARG_OUT
AWTError
AWTEvent
AWTEventListener
AWTEventListenerProxy
AWTEventMulticaster
AWTException
AWTKeyStroke

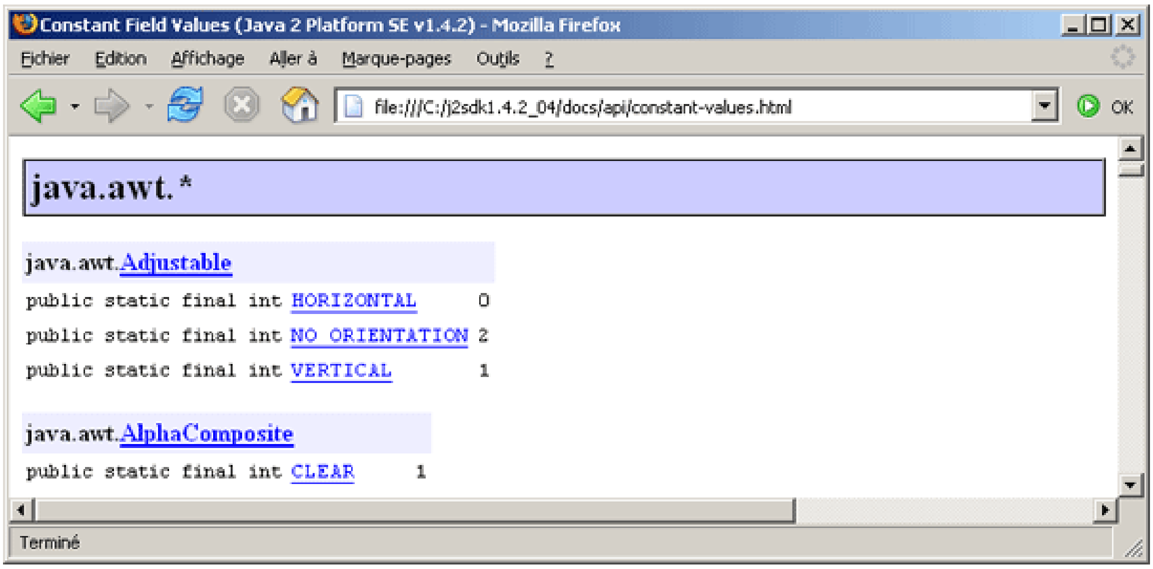
Le fichier constant-values.html affiche la liste de toutes les constantes avec leurs valeurs.



Constant Field Values

Contents

- java.awt.*
- java.beans.*
- java.io.*


Constant Field Values (Java 2 Platform SE v1.4.2) - Mozilla Firefox
Fichier
Edition
Affichage
Aller à
Marque-pages
Outils
?
file:///C:/j2sdk1.4.2_04/docs/api/constant-values.html
OK
Terminé

java.awt.*

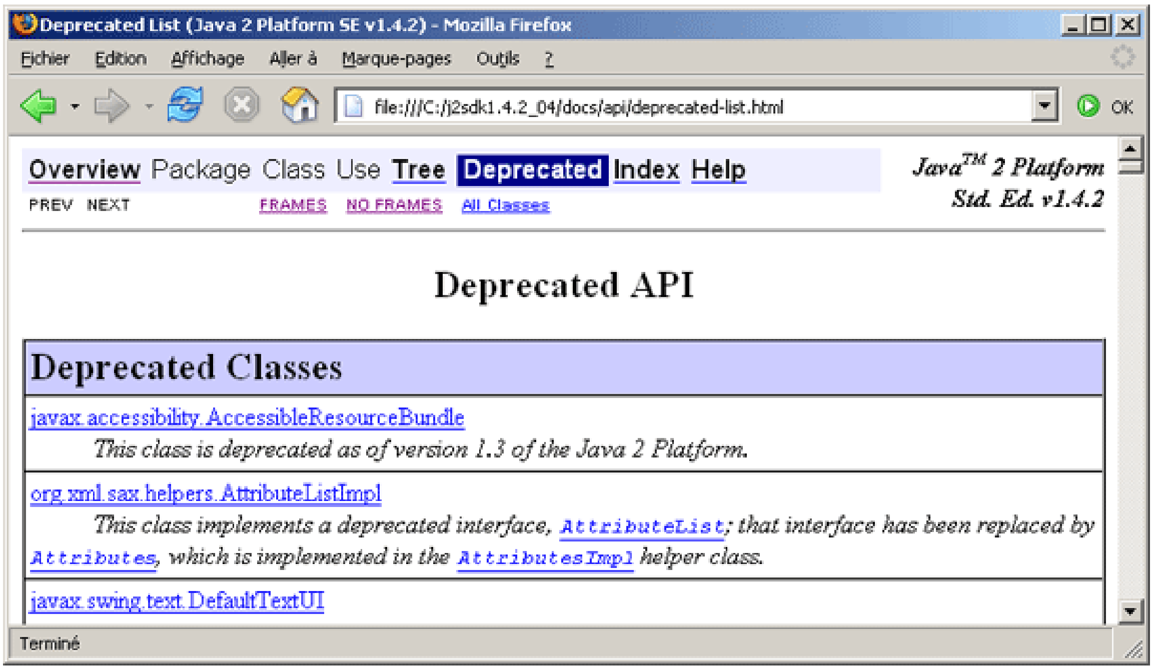
java.awt.Adjustable

public static final int HORIZONTAL	0
public static final int NO_ORIENTATION	2
public static final int VERTICAL	1

java.awt.AlphaComposite

public static final int CLEAR	1

Le fichier deprecated-list.html affiche la liste de tous les membres déclarés deprecated. Le lien Deprecated de la barre de navigation permet d'afficher le contenu de cette page.


Deprecated List (Java 2 Platform SE v1.4.2) - Mozilla Firefox
Fichier
Edition
Affichage
Aller à
Marque-pages
Outils
?
file:///C:/j2sdk1.4.2_04/docs/api/deprecated-list.html
OK
Overview Package Class Use Tree Deprecated Index Help
PREV NEXT
FRAMES NO FRAMES All Classes
Java™ 2 Platform
Std. Ed. v1.4.2
Terminé

Deprecated API

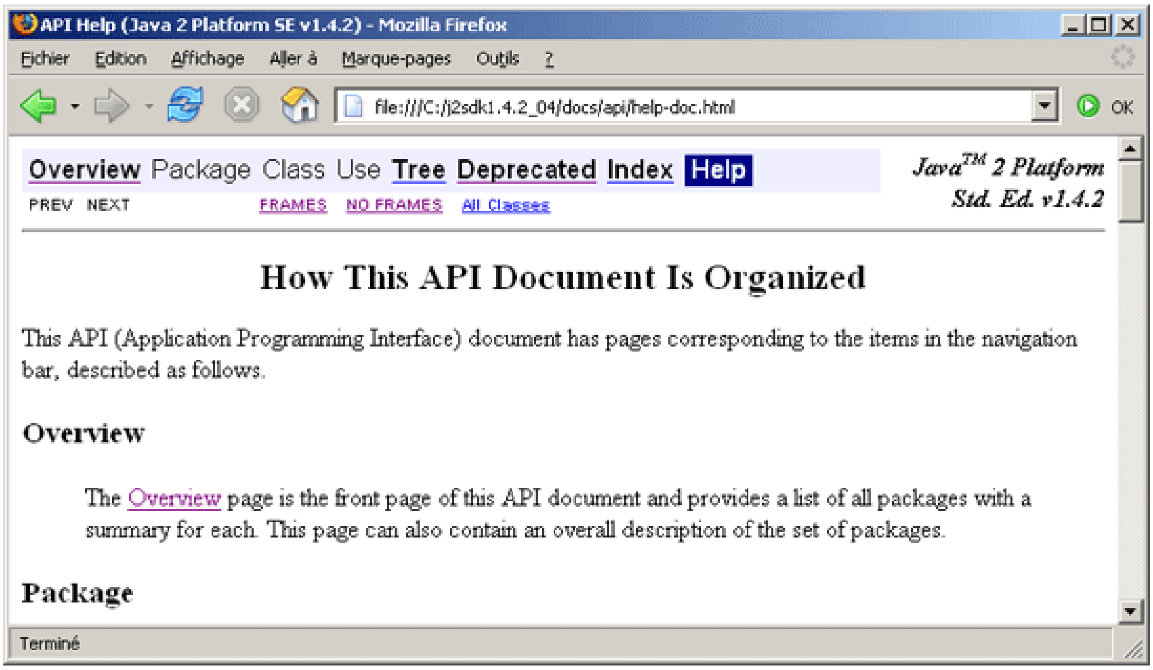
Deprecated Classes

javax.accessibility.AccessibleResourceBundle
This class is deprecated as of version 1.3 of the Java 2 Platform.

org.xml.sax.helpers.AttributeListImpl
This class implements a deprecated interface, `AttributeList`*; that interface has been replaced by* `Attributes`*, which is implemented in the* `AttributesImpl` *helper class.*

javax.swing.text.DefaultTextUI

Le fichier help-doc.html affiche l'aide en ligne de la documentation. Le lien Help de la barre de navigation permet d'afficher le contenu de cette page.



How This API Document Is Organized

This API (Application Programming Interface) document has pages corresponding to the items in the navigation bar, described as follows.

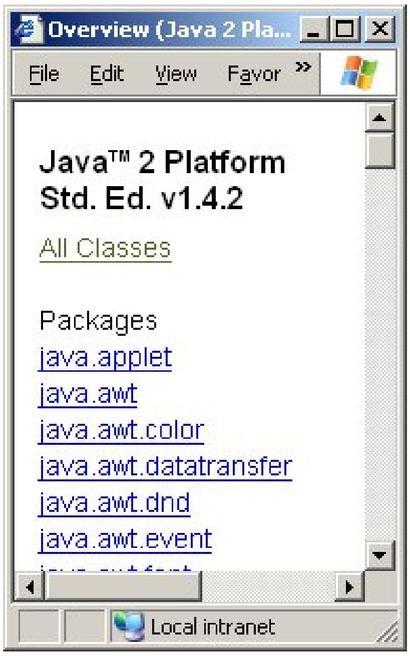
Overview

The Overview page is the front page of this API document and provides a list of all packages with a summary for each. This page can also contain an overall description of the set of packages.

Package

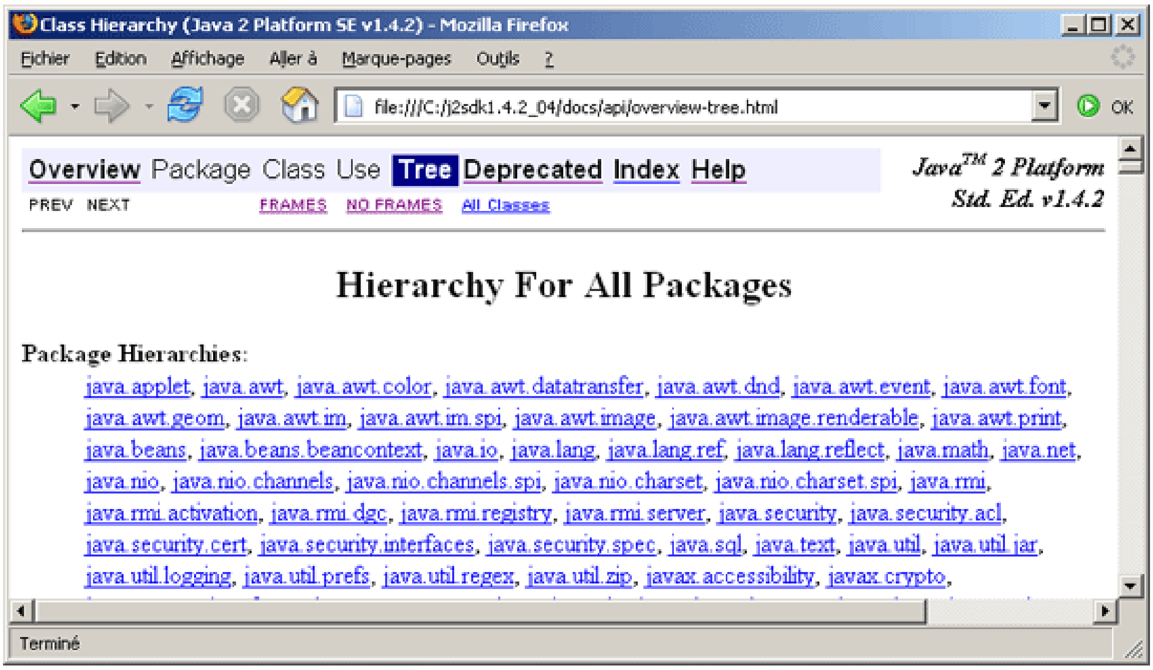
Le fichier index.html est la page principale de la documentation composée de 3 frames.

Le fichier overview-frame.html affiche la liste des packages avec un lien pour afficher la liste des membres du package. Cette page est affichée en haut à gauche dans le fichier index.html.



Le fichier overview-summary.html affiche un résumé des packages de la documentation. Cette page est affichée par défaut dans la partie centrale de la page index.html.

Le fichier overview-tree.html affiche la hiérarchie des classes et interfaces. Le lien Tree de la barre de navigation permet d'afficher le contenu de cette page.



Class Hierarchy (Java 2 Platform SE v1.4.2) - Mozilla Firefox
Fichier Edition Affichage Aller à Marque-pages Outils ?
file:///C:/j2sdk1.4.2_04/docs/api/overview-tree.html
OK
Terminé

Overview Package Class Use **Tree** Deprecated Index Help

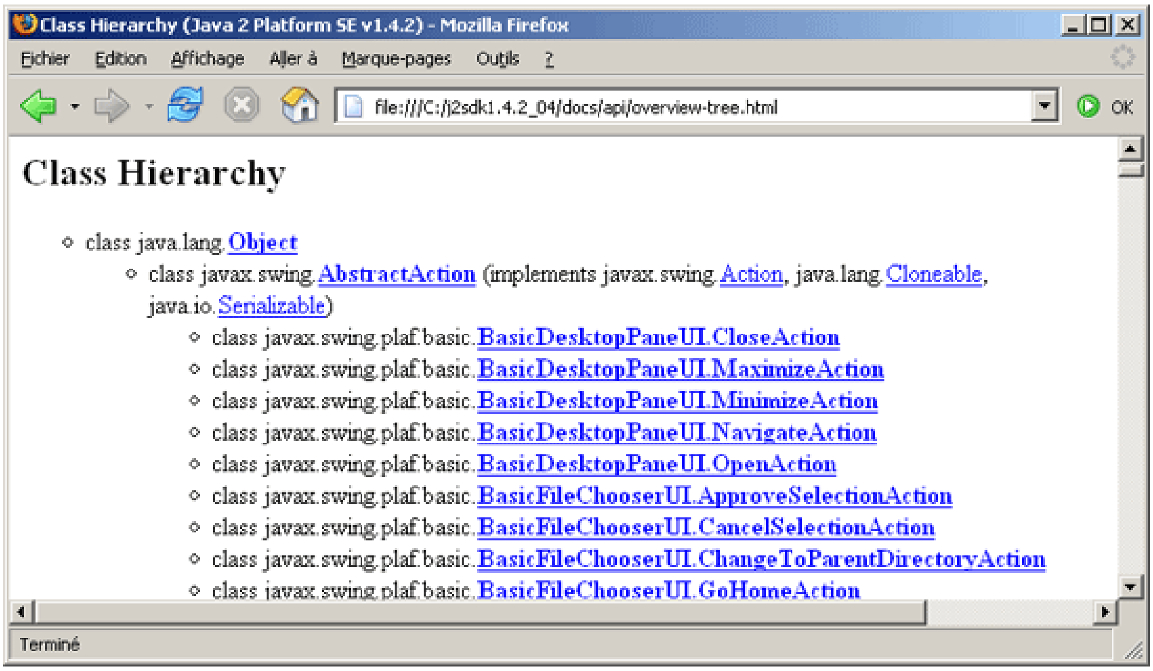
PREV NEXT FRAMES NO FRAMES All Classes

Java™ 2 Platform Std. Ed. v1.4.2

Hierarchy For All Packages

Package Hierarchies:

java.applet, java.awt, java.awt.color, java.awt.datatransfer, java.awt.dnd, java.awt.event, java.awt.font, java.awt.geom, java.awt.im, java.awt.im.spi, java.awt.image, java.awt.image.renderable, java.awt.print, java.beans, java.beans.beancontext, java.io, java.lang, java.lang.ref, java.lang.reflect, java.math, java.net, java.nio, java.nio.channels, java.nio.channels.spi, java.nio.charset, java.nio.charset.spi, java.rmi, java.rmi.activation, java.rmi.dgc, java.rmi.registry, java.rmi.server, java.security, java.security.acl, java.security.cert, java.security.interfaces, java.security.spec, java.sql, java.text, java.util, java.util.jar, java.util.logging, java.util.prefs, java.util.regex, java.util.zip, javax.accessibility, javax.crypto,



Class Hierarchy (Java 2 Platform SE v1.4.2) - Mozilla Firefox
Fichier Edition Affichage Aller à Marque-pages Outils ?
file:///C:/j2sdk1.4.2_04/docs/api/overview-tree.html
OK
Terminé

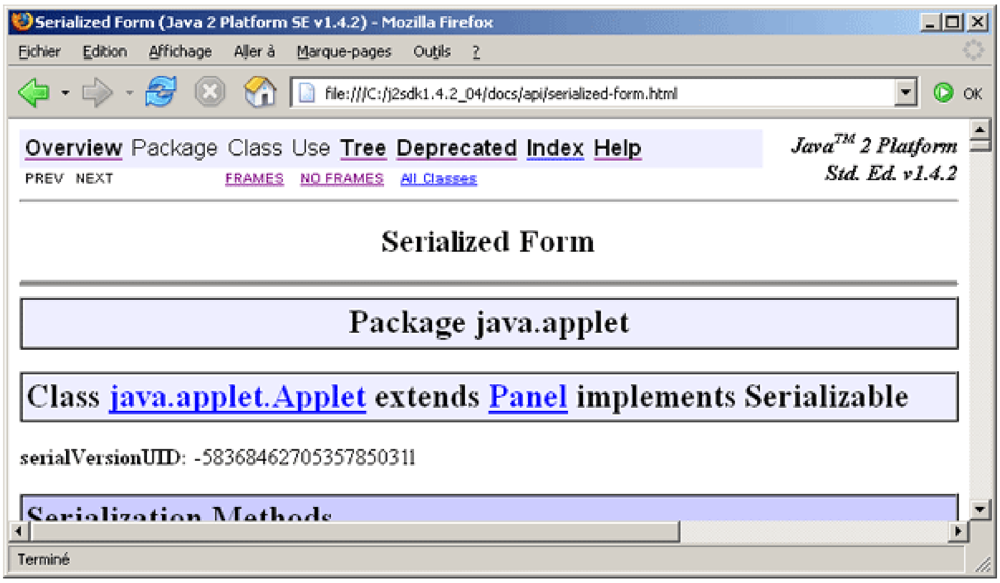
Class Hierarchy

- class java.lang.Object
  - class javax.swing.AbstractAction (implements javax.swing.Action, java.lang.Cloneable, java.io.Serializable)
    - class javax.swing.plaf.basic.BasicDesktopPaneUI.CloseAction
    - class javax.swing.plaf.basic.BasicDesktopPaneUI.MaximizeAction
    - class javax.swing.plaf.basic.BasicDesktopPaneUI.MinimizeAction
    - class javax.swing.plaf.basic.BasicDesktopPaneUI.NavigateAction
    - class javax.swing.plaf.basic.BasicDesktopPaneUI.OpenAction
    - class javax.swing.plaf.basic.BasicFileChooserUI.ApproveSelectionAction
    - class javax.swing.plaf.basic.BasicFileChooserUI.CancelSelectionAction
    - class javax.swing.plaf.basic.BasicFileChooserUI.ChangeToParentDirectoryAction
    - class javax.swing.plaf.basic.BasicFileChooserUI.GoHomeAction

Le fichier package-list est un fichier texte contenant la liste de tous les packages (non affiché dans la documentation).

Le fichier packages.html permet de choisir entre les versions avec et sans frame de la documentation.

Le fichier serialized-form.html affiche la liste des classes qui sont sérialisables.


Serialized Form (Java 2 Platform SE v1.4.2) - Mozilla Firefox
Fichier Edition Affichage Aller à Marque-pages Outils ?
file:///C:/j2sdk1.4.2_04/docs/api/serialized-form.html
OK
Overview Package Class Use Tree Deprecated Index Help
PREV NEXT
FRAMES NO FRAMES All Classes
JavaTM 2 Platform
Std. Ed. v1.4.2
Terminé

Serialized Form

Package java.applet

Class java.applet.Applet extends Panel implements Serializable

serialVersionUID: -5836846270535785031l

Serialization Methods

Le fichier stylesheet.css est la feuille de style utilisée pour afficher la documentation.

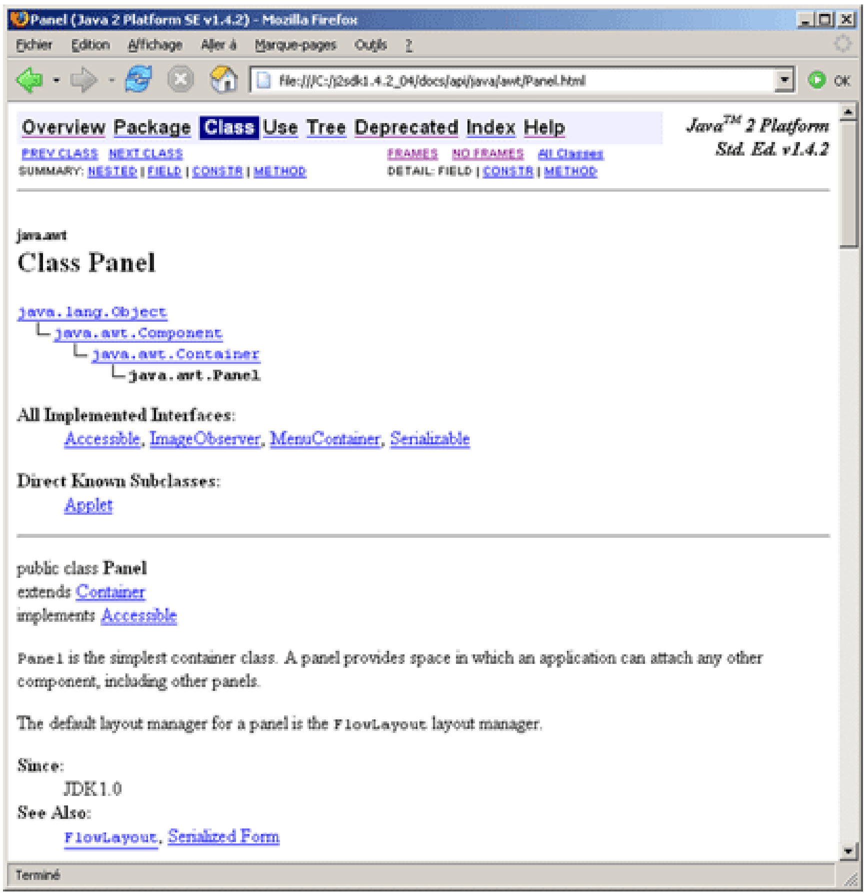
Le fichier allclasses-noframe.html affiche la page allclasses-frame.html sans frame.

Il y a un répertoire par package. Ce répertoire contient plusieurs fichiers :

- classe.html : un fichier html contenant la définition de chaque classe du package
- package-frame.html : contient la liste de toutes les interfaces, classes et exceptions du package
- package-summary.html : contient un résumé de toutes les interfaces, classes et exceptions du package
- package-tree.html : contient l'arborescence de toutes les interfaces et classes du package

Cette structure est reprise pour les sous-packages.

La page détaillant une classe possède la structure suivante :


Panel (Java 2 Platform SE v1.4.2) - Mozilla Firefox
Fichier Edition Affichage Aller à Marque-pages Outils ?
file:///C:/j2sdk1.4.2_04/docs/api/java/awt/Panel.html
OK
Overview Package Class Use Tree Deprecated Index Help
PREV CLASS NEXT CLASS
FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD
JavaTM 2 Platform
Std. Ed. v1.4.2
Terminé

java.awt
Class Panel

```
java.lang.Object
  └java.awt.Component
      └java.awt.Container
          └java.awt.Panel
```

All Implemented Interfaces:
Accessible, ImageObserver, MenuContainer, Serializable

Direct Known Subclasses:
Applet

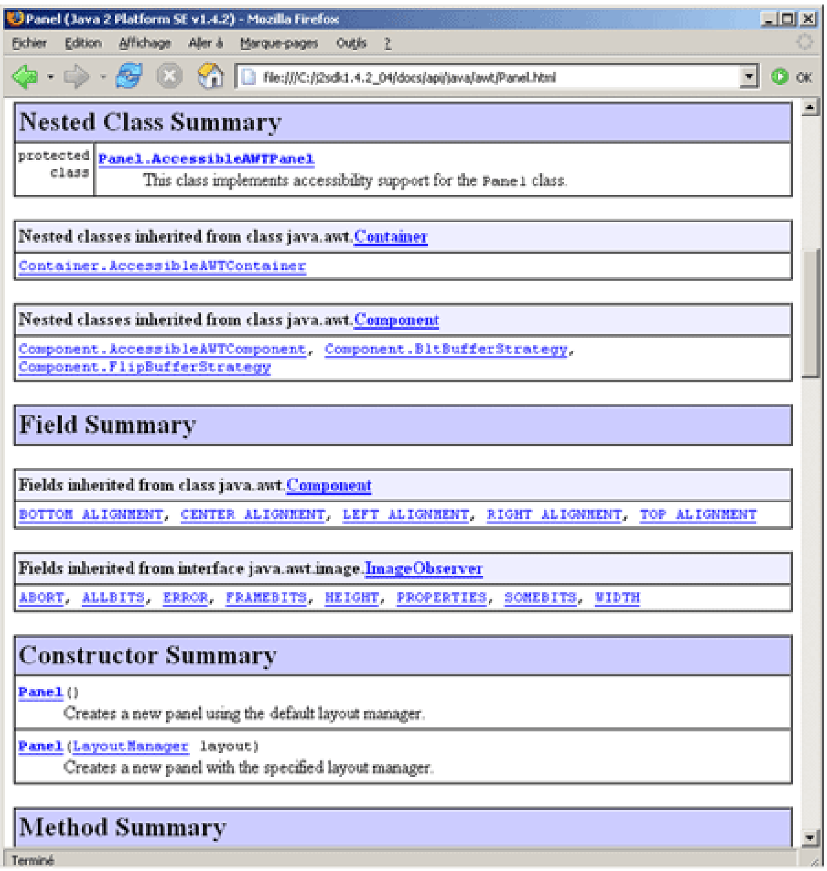
public class Panel
extends Container
implements Accessible

Panel is the simplest container class. A panel provides space in which an application can attach any other component, including other panels.

The default layout manager for a panel is the FlowLayout layout manager.

Since:
JDK1.0

See Also:
FlowLayout, Serialized Form


Panel (Java 2 Platform SE v1.4.2) - Mozilla Firefox
Fichier Edition Affichage Aller à Marque-pages Outils ?
file:///C:/j2sdk1.4.2_04/docs/api/java/awt/Panel.html
OK
Terminé

Nested Class Summary

protected class	Panel.AccessibleAWTPanel This class implements accessibility support for the Panel class.

Nested classes inherited from class java.awt.Container

Container.AccessibleAWTContainer

Nested classes inherited from class java.awt.Component

Component.AccessibleAWTComponent, Component.BltBufferStrategy, Component.FlipBufferStrategy

Field Summary

Fields inherited from class java.awt.Component

BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNMENT, RIGHT_ALIGNMENT, TOP_ALIGNMENT

Fields inherited from interface java.awt.image.ImageObserver

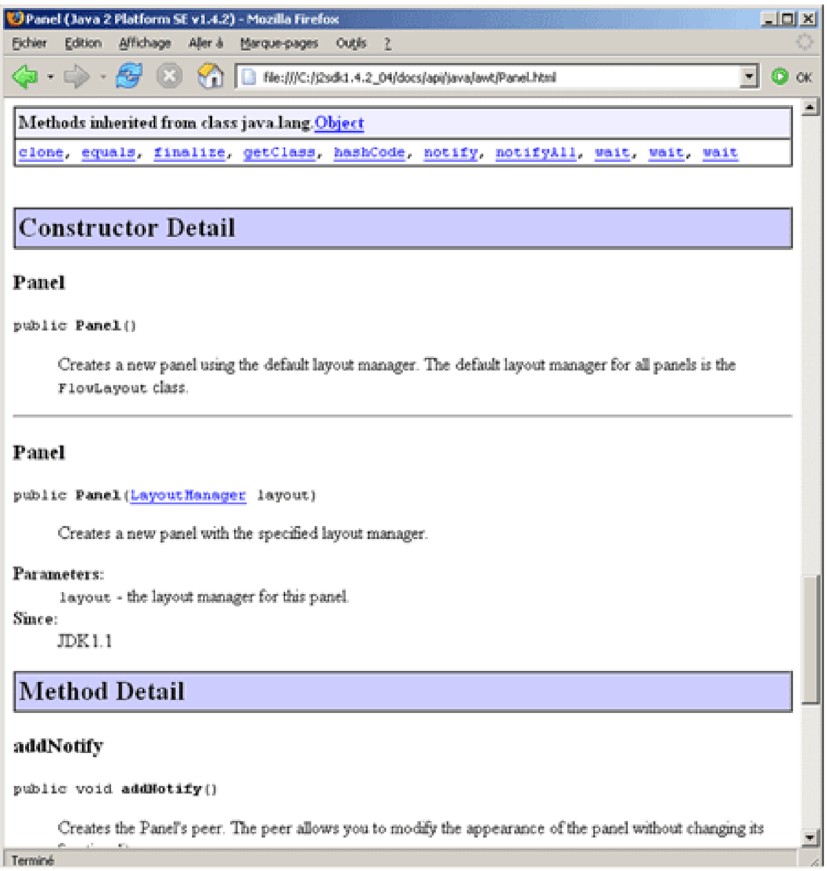
ABORT, ALLBITS, ERROR, FRAMEBITS, HEIGHT, PROPERTIES, SOMEBITS, WIDTH

Constructor Summary

Panel()
Creates a new panel using the default layout manager.

Panel(LayoutManager layout)
Creates a new panel with the specified layout manager.

Method Summary


Panel (Java 2 Platform SE v1.4.2) - Mozilla Firefox
Fichier Edition Affichage Aller à Marque-pages Outils ?
file:///C:/j2sdk1.4.2_04/docs/api/java/awt/Panel.html
OK
Terminé

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

Panel

`public Panel()`

Creates a new panel using the default layout manager. The default layout manager for all panels is the FlowLayout class.

Panel

`public Panel(LayoutManager layout)`

Creates a new panel with the specified layout manager.

Parameters:
layout - the layout manager for this panel.
Since:
JDK1.1

Method Detail

addNotify

`public void addNotify()`

Creates the Panel's peer. The peer allows you to modify the appearance of the panel without changing its

Si l'option -linksource est utilisée, les fichiers sources sont stockés dans l'arborescence du sous-répertoire src-html de la documentation.

98. JShell

Chapitre 98

Niveau : 🟩□□□□□ Fondamental

JShell, signifiant Java Shell, est défini dans la JEP 222 : c'est une implémentation d'un outil en ligne de commande fourni dans le JDK à partir de Java 9 qui propose une console interactive de type REPL (Read-Eval-Print-Loop). JShell est issu des travaux du projet Kulla d'OpenJDK.

JShell permet d'écrire du code Java (expressions, instructions, déclaration, ...) qui est évalué individuellement sans avoir obligatoirement à les placer dans une classe ou une méthode.

Ce chapitre contient plusieurs sections :

- ♦ Les outils de type REPL
- ♦ Introduction à JShell
- ♦ La saisie d'éléments dans JShell
- ♦ Les fragments
- ♦ Les commandes de JShell
- ♦ L'édition
- ♦ Les fonctionnalités de l'environnement
- ♦ Les scripts
- ♦ Les modes de feedback
- ♦ L'utilisation de classes externes
- ♦ Les options de JShell
- ♦ JShell API

98.1. Les outils de type REPL

Un REPL est un outil en ligne de commande qui permet rapidement d'écrire et d'exécuter du code. C'est un environnement de programmation en ligne de commande qui lit les instructions saisies, les évalue, affiche le résultat et ce de manière répétée.

Un outil de type REPL est une interface en ligne de commande qui permet :

- de saisir des expressions ou des traitements (Read)
- de les évaluer une fois la touche Entrée pressée (Eval)
- d'afficher le résultat (Print)
- et de recommencer (Loop)

De nombreux langages possèdent un outil de type REPL : Lisp, Python, Groovy, Ruby, ...

La plupart des langages s'exécutant dans une JVM ou non offre un outil de type RPEL (Scala, Groovy, Kotlin, Python, ...)

Un Shell Unix est aussi un outil de type REPL : il permet de lire des commandes, les évoluer, affiche le résultat et répéter ces traitements.

98.1.1. L'utilité d'un outil de type REPL

Un outil de type REPL peut avoir de nombreuses utilisations :

- expérimenter du code
- explorer une API
- faciliter la découverte du langage sans avoir à utiliser un IDE notamment dans le cadre de l'enseignement ou de l'apprentissage du langage

Il permet facilement d'explorer les fonctionnalités d'une API, ou de développer un petit prototype.

L'utilisation d'un outil de type REPL est particulièrement utile pour réduire les délais d'exécution. Un outil de type REPL permet de réduire le temps des boucles de retours notamment en permettant de voir le résultat du code saisi immédiatement. Cela augmente grandement la productivité.

98.1.2. REPL vs IDE

Les IDE proposent généralement une solution pour permettre de tester du code Java sans avoir à écrire une méthode main() dans une classe.

Un outil de type REPL n'a pas pour but d'être un IDE : il ne propose donc pas d'interface graphique, de débogueur, ...

L'utilisation d'un outil de type REPL est orientée expression : c'est une approche assez différente de l'écriture de code Java qui est orientée déclaration.

Un REPL n'est pas adapté au développement de code complexe comme une application mais il est particulièrement efficace pour écrire de petites portions de code.

98.2. Introduction à JShell

Java 9 offre un outil de type REPL (Read Evaluate Print Loop ou boucle de lecture, évaluation, impression en français) nommé JShell. Cette fonctionnalité a été développée dans le projet Kulla. JShell est intégré à Java 9 via la JEP 222 (JShell Java Enhancement Proposal) en tant qu'outil standard du JDK.

JShell est un outil en ligne de commande qui permet de saisir et d'évaluer dynamiquement des déclarations, des expressions et l'exécution de traitements sans avoir à les inclure dans une classe. C'est un outil interactif qui utilise l'API Compiler pour évaluer dynamiquement les expressions ou le code fourni : cela implique la compilation, l'exécution et le renvoie du résultat.

Il est possible de faire exécuter du code Java sans avoir à l'inclure dans une classe ou une méthode. Ceci permet de plus facilement et rapidement expérimenter du code Java.

98.2.1. Les intérêts à utiliser JShell

JShell est un outil en ligne de commande qui permet d'évaluer des traitements, des déclarations et des expressions Java très facilement. Un outil de type REPL permet de saisir de manière interactive des fragments de code arbitraires, de les évaluer et d'afficher leurs résultats.

L'évaluation interactive du code saisi est plus efficace que le développement traditionnel en Java (écriture / compilation / exécution) qui requière généralement plusieurs étapes :

- l'écriture du code dans un fichier,
- la compilation du fichier source pour créer un (ou plusieurs) fichier .class (en corrigeant les erreurs aux besoins),
- l'exécution dans une JVM de ce fichier .class éventuellement en mode debug,
- la réalisation de tests,
- selon le résultat des tests, itérer sur ce processus jusqu'à satisfaction

Ce workflow repose sur l'utilisation d'un ou plusieurs fichiers et d'un ou plusieurs outils. D'autant qu'une application Java dépend généralement de dizaines voire de centaines de bibliothèques et éventuellement de conteneurs notamment pour exécuter des applications d'entreprise utilisant Java EE.

Toutes ces étapes sont plutôt longues car elles reposent sur un ou plusieurs fichiers. De plus pour pouvoir être exécutée, la classe doit contenir une méthode public static void main() avec une signature particulière.

D'un autre côté, JShell propose une solution plus interactive. Le développeur peut rapidement saisir des portions de code qui vont être évaluées au fur et à mesure et JShell va fournir pour chacun un retour sur les résultats de l'évaluation.

Lors de la découverte d'un langage, le fait d'avoir un retour immédiat à chaque ligne de code saisie permet d'accélérer la courbe d'apprentissage. JShell peut être utilisé pour réduire la courbe d'apprentissage du langage Java ou de certaines API notamment en évitant le code inutile pour se concentrer sur le code utile.

La cible de JShell n'est pas que les étudiants qui se forment à Java mais aussi les développeurs qui pourront facilement explorer une API ou tester un algorithme.

JShell ne remplace pas un IDE mais il permet de tester facilement du code Java en ligne de commande interactive. Il est ainsi possible de tester du code sans avoir à écrire une classe avec une méthode main(). Avec JShell, les développeurs peuvent écrire du code, l'exécuter et continuer à faire évoluer leur code à la volée sans avoir à sortir pour une compilation et une exécution.

JShell est un outil qui peut donc changer la manière dont les développeurs vont apprendre et écrire du code pour tester des fonctionnalités. JShell peut offrir un réel intérêt dans certains cas notamment :

- faciliter l'apprentissage du langage Java grâce à son mode de fonctionnement
- développer rapidement des prototypes
- explorer l'utilisation et l'expérimentation d'une nouvelle API ou d'une bibliothèque : Java propose dans le JDK de nombreuses API et il existe un nombre encore plus important d'API open source
- tester rapidement des portions de code : par exemple pour valider un algorithme ou un prototype
- réaliser des démos ou des exercices

98.2.2. L'implémentation de JShell

L'implémentation de JShell s'appuie de préférence sur des fonctionnalités existantes dans le JDK lorsque celles-ci sont disponibles, notamment :

- l'état est inclus dans une instance de JVM
- l'API Compiler permet de créer le byte code du code exécuté avec quelques sous-classes prenant en charge les fragments qui ne sont pas du code Java standard
- l'API Java Debug Interface (JDI) est utilisée pour remplacer du code existant

JShell utilise aussi la bibliothèque jline2 pour gérer les saisies de la ligne de commande incluant les mises à jour des fragments et l'utilisation de l'historique.

98.2.3. Lancer et arrêter JShell

Pour lancer JShell, il suffit de lancer la commande jshell fournie par un JDK 9 minimum dans le sous-répertoire bin. Le plus simple pour ne pas être obligé de préciser le chemin complet de la commande, il faut que la variable d'environnement JAVA_HOME pointe sur le répertoire d'installation du JDK de Java 9 que le sous-répertoire JAVA_HOME/bin soit ajouté dans le PATH du système. Une fois cela fait, il suffit d'exécuter la commande jshell.

Résultat :

```
C:\java>java -version
java version "9.0.1"
Java(TM) SE Runtime Environment (build 9.0.1+11)
Java HotSpot(TM) 64-Bit Server VM (build 9.0.1+11, mixed mode)

C:\java>jshell
|  Welcome to JShell -- Version 9.0.1
|  For an introduction type: /help intro

jshell>
```

Une fois démarré, JShell affiche une invite de commande « jshell> » par défaut qui permet de saisir un fragment de code ou une commande.

Comme proposé au lancement de JShell, il est possible de demander l'affichage de l'aide en ligne en utilisant la commande /help intro.

Exemple (code Java 9) :

```
jshell> /help intro
|
|  intro
|
|  The jshell tool allows you to execute Java code, getting immediate results.
|  You can enter a Java definition (variable, method, class, etc), like:  int x = 8
|  or a Java expression, like:  x + x
|  or a Java statement or import.
|  These little chunks of Java code are called 'snippets'.
|
|  There are also jshell commands that allow you to understand and
|  control what you are doing, like:  /list
|
|  For a list of commands: /help
```

Pour quitter JShell, il suffit d'utiliser la commande /exit.

Résultat :

```
jshell> /exit
|  Goodbye

C:\java>
```

Il est aussi possible d'utiliser le raccourci clavier CTRL+D.

Il est possible de démarrer JShell en mode verbeux en utilisant l'option -v à son lancement

Résultat :

```
C:\java> jshell -v
|  Welcome to JShell -- Version 9.0.1
|  For an introduction type: /help intro

jshell> 2+3
$1 ==> 5
|  created scratch variable $1 : int
```

Lors de l'exécution en mode verbeux, JShell affiche non seulement le résultat de l'évaluation du fragment mais il fournit également une description de ce qu'il a fait, préfixée par un caractère « pipe » (une barre verticale).

Le mode verbeux est utile lors de l'apprentissage de l'utilisation de JShell. Avec de l'expérience, le mode normal ou un mode plus concis est préférable.

98.2.4. La mise en oeuvre de JShell

JShell propose des fonctionnalités pour faciliter la saisie de code et les interactions : un historique avec édition, complétion de code, ajout automatique des points virgules terminaux, imports, ...

Une session interactive en ligne de commande est ouverte accompagnée d'un message d'accueil indiquant quelle version de JShell est utilisée.

Traditionnellement, la première portion de code écrite et testée est un simple « Hello world ». Cela se fait très simplement avec JShell en saisissant l'instruction et en appuyant sur la touche Entrée

Résultat :

```
jshell> System.out.println("Hello world")
Hello world
```

Plusieurs points sont notables :

- il n'a pas été utile de créer une classe ni une méthode main()
- l'instruction saisie n'a pas l'obligation de se terminer par un point-virgule

Le point-virgule de fin d'instruction est optionnel : il n'est obligatoire que si elle est incluse dans un bloc de code ou si la ligne contient plusieurs instructions. Dans ces cas, chaque instruction doit obligatoirement se terminer par un point-virgule.

JShell propose aussi des surcharges des méthodes println() et printf() qui permettent de simplifier encore cet exemple, si ces fonctionnalités sont activées :

Résultat :

```
jshell> printf("Hello world")
Hello world
```

Avec JShell, chaque ligne de code est saisie les unes après les autres avec un affichage immédiat du résultat de son exécution.

JShell maintient un état (state) qui contient le code évalué et le résultat de son exécution.

JShell facilite l'évaluation de code notamment en permettant :

- l'évaluation d'expression et de déclarations sans avoir à les inclure dans une méthode
- la déclaration de variables sans avoir à les inclure dans une classe ou une méthode
- la définition de méthodes sans avoir à les inclure dans une classe ou une interface

Cela signifie, par exemple, qu'il est possible d'exécuter quelques lignes de Java sans avoir à écrire une classe ou une méthode.

98.3. La saisie d'éléments dans JShell

JShell permet la saisie de deux types d'éléments :

- des fragments (snippets) de code Java
- des commandes (commands) qui commencent par un caractère slash

JShell propose aussi de nombreuses fonctionnalités pour faciliter leur saisie :

- maintient un historique,
- possède un éditeur intégré simple,
- complétion en utilisant la touche tab,
- propose des commandes pour sauver et charger du code, quitter, ...
- ajoute automatiquement un point-virgule,
- ...

98.3.1. Les fonctionnalités de saisie

Un fragment (snippet) correspond à un des éléments syntaxiques du Java Language Specification (JLS) :

- une valeur primitive
- une expression
- une instruction
- la déclaration d'un champ
- la déclaration d'une méthode
- la déclaration d'une classe, d'une interface, d'une énumération ou d'un record
- la déclaration d'un import

Seuls les classes et les imports peuvent être gérés seuls : les autres types d'éléments sont associés à un contexte créé par JShell :

- les variables et méthodes sont associées comme membre static d'une classe dédiée
- les expressions et les instructions sont associées à une méthode static d'une classe dédiée

Pour permettre d'explorer et de tester du code, la déclaration d'un élément doit être capable d'évoluer dans le temps. Dans une session JShell, chaque élément possède une clé :

- le nom pour une variable ou une classe
- la signature pour une méthode (pour supporter la surcharge)

A un instant donné, un élément ne peut avoir qu'une seule déclaration.

L'état de la session est maintenu cohérent au fur et à mesure que les fragments sont évalués : chaque changement qui implique un impact est propagé suite à l'évaluation.

98.3.2. Le résultat de l'évaluation d'un fragment

Le résultat d'une évaluation correcte d'un fragment peut prendre trois valeurs :

- added (ajouté) : c'est la première déclaration de l'élément
- modified (modifié) : l'élément existe mais sa signature reste inchangée. Dans ce cas, aucun autre fragment n'est impacté
- replaced (remplacé) : l'élément existe mais sa signature a changé

Si le résultat de l'évaluation est modified ou replaced, alors la version existante du fragment est retirée de la session et la nouvelle version la remplace.

Quant un fragment est ajouté, il peut créer des références non résolues, car elle n'existe pas encore.

Quant un fragment est remplacé, il peut mettre à jour un ou plusieurs fragments qui en font usage. C'est par exemple le cas si le type de la valeur de retour change : cela peut éventuellement rendre les éléments impactés invalides.

Lorsqu'une variable est remplacée, par une action directe ou indirecte, sa valeur est réinitialisée avec sa valeur par défaut selon ton type.

Lorsqu'un élément est invalide, soit à cause d'une référence à venir, soit parce qu'elle devient invalide à la suite d'une mise à jour, la déclaration est «corralled». Une déclaration corrigée peut être utilisée dans le code d'autres déclarations. Par contre, une exception sera levée à l'exécution tant que code n'est pas corrigé pour le rendre valide.

98.4. Les fragments

Le texte saisi qui n'est pas une commande est désigné par le terme fragment (snippet).

JShell permet la déclaration de différents éléments du langage Java :

- la définition de variables,
- des expressions,
- des traitements,
- des imports,
- des types,
- des méthodes

Tous ces éléments sont désignés sous le terme fragment (snippet).

Il n'est pas nécessaire de créer une classe, ni de méthode main() : il suffit de saisir un fragment de code Java et d'appuyer sur la touche entrée pour que celui-ci soit évalué et le résultat affiché.

Si le fragment tient sur une seule ligne, il est immédiatement évalué et le résultat est affiché. Ce résultat peut être une erreur.

Résultat :

```
jshell> int i = 10;
i ==> 10
```

Si une expression est saisie, le résultat de son évaluation est affecté à une variable (scratched variable) créée pour l'occasion. Son nom commence par un $ suivi de l'identifiant du fragment.

Résultat :

```
jshell> 4+6
$2 ==> 10
```

Comme JShell permet la saisie des fragments un par un, le point-virgule de fin d'instruction est optionnel : s'il n'est pas présent, il sera automatiquement ajouté avant l'évaluation.

Si plusieurs instructions sont saisies en même temps, le ou les points virgules de séparation sont toujours obligatoire.

Résultat :

```
jshell> int i = 1; int j = 2;
i ==> 1
j ==> 2
```

Un fragment peut être une méthode, un type (classe, interface, énumération, record) : tous ces éléments requièrent généralement plusieurs lignes donc un fragment n'est pas obligatoirement sur une seule ligne.

Il est donc possible de saisir un fragment composé de plusieurs lignes, comme par exemple la définition d'une méthode. Le prompt change pour devenir ...> pour indiquer la saisie de la suite du fragment.

Résultat :

```
jshell> void afficher(String message) {
   ...> System.out.println(message);
   ...> }
```

Pour exécuter la méthode, il suffit de l'invoquer en tant que fragment.

Résultat :

```
jshell> afficher("bonjour");
bonjour
```

Pour modifier la définition d'un snippet (variable, méthode, classe, ...), il suffit de ressaisir sa définition. Attention, parfois la redéfinition peut induire des incompatibilités comme par exemple si une variable est redéfinie avec un autre type.

Les variables, les méthodes et les classes définies ne sont pas encapsulées dans une classe accessible par l'utilisateur de JShell mais sont encapsulées comme des variables statiques d'une classe générée en interne.

Toutes les variables, les méthodes et les types définies sont accessibles durant toute la session courante de JShell.

98.4.1. Les variables et les expressions

JShell permet la déclaration implicite et explicite de variables. Il est possible de saisir directement une expression Java comme par exemple une opération arithmétique, une manipulation de chaînes de caractères, la définition de variables, la création d'une instance, l'invocation d'une méthode, ...

Une fois l'expression saisie et la touche « Entrée » appuyée, l'expression est immédiatement évaluée et le résultat de cette évaluation est affiché.

Si ce n'est pas fait explicitement dans l'expression, le résultat est affecté à une variable temporaire (scratch variable) dont le nom est composé du signe dollar suivi d'un nombre unique qui correspond à l'identifiant du fragment.

Résultat :

```
jshell> 1+2
$1 ==> 3
```

Cette nouvelle variable implicitement définie peut elle-même être réutilisée dans une autre expression.

Résultat :

```
jshell> $1+2
$2 ==> 5
```

Il est bien sûr possible de définir explicitement le nom de la variable dans l'expression

Résultat :

```
jshell> int nbElements = 10
nbElements ==> 10
```

Il est possible de définir une variable sans l'initialiser.

Résultat :

```
jshell> Date date
date ==> null

jshell> int i
i ==> 0
```

L'expression peut être l'invocation d'une méthode

Résultat :

```
jshell> System.out.println($1)
3
```

L'expression peut concerner la définition et la manipulation de chaînes de caractères

Résultat :

```
jshell> String salutation = "Hello " + "world"
salutation ==> "Hello world"

jshell> salutation.toUpperCase()
$6 ==> "HELLO WORLD"
```

JShell affiche une erreur si la syntaxe est incorrecte.

Résultat :

```
jshell> salutation.toUppercase()
|  Error:
|  cannot find symbol
|    symbol:   method toUppercase()
|  salutation.toUppercase()
|  ^--------------------^
```

L'expression peut être l'invocation d'une méthode : de la même manière le résultat est implicitement affecté à une variable temporaire par défaut.

Résultat :

```
jshell> salutation.split(" ")
$7 ==> String[2] { "Hello", "world" }
```

Une expression peut être saisie sur plusieurs lignes. Si JShell détecte que l'expression n'est pas complète, un prompt différent est affiché pour indiquer de saisir la suite de l'expression.

Résultat :

```
jshell> int a =
   ...> 10
a ==> 10
```

La définition d'une variable avec le modificateur static ou final affiche un warning : la variable est définie mais le modificateur est ignoré.

Exemple (code Java 9) :

```
jshell> static int a = 10;
|  Warning:
|  Modifier 'static'  not permitted in top-level declarations, ignored
|  static int a = 10;
|  ^----^
a ==> 10

jshell> final int a = 10;
|  Warning:
|  Modifier 'final'  not permitted in top-level declarations, ignored
|  final int a = 10;
|  ^---^
a ==> 10
```

98.4.2. Les instructions de contrôle de flux

Les snippets peuvent être des instructions de contrôle de flux (if, for, while, ...). Il est possible d'écrire ces instructions sur plusieurs lignes.

JShell est suffisamment intelligent pour reconnaître les instructions multilignes et propose une invite avec un symbole ...> pour nous permettre de saisir la suite de l'instruction sur la ligne suivante.

Résultat :

```
jshell> if (note >=12) {
   ...> System.out.println("bien");
   ...> } else {
   ...> System.out.println("faible");
   ...> }
bien
```

Il est à noter que dans ce cas les instructions doivent se terminer par un point-virgule.

Résultat :

```
jshell> if (note >=12) {
   ...> System.out.println("bien")
   ...> } else {
   ...> System.out.println("faible")
   ...> }
|  Error:
|  ';' expected
|  System.out.println("bien")
|                            ^
|  Error:
|  ';' expected
|  System.out.println("faible")
|
```

L'expression peut évidemment être une boucle.

Résultat :

```
jshell> String[] lettres = {"A", "B", "C", "D"}
lettres ==> String[4] { "A", "B", "C", "D" }

jshell> for (String lettre : lettres) {
   ...> System.out.print(lettre + " ");
   ...> }
A B C D
```

Les instructions break et continue ne sont pas utilisables comme instruction de premier niveau.

Exemple (code Java 9) :

```
jshell> continue;
|  Error:
|  continue outside of loop
|  continue;
|  ^-------^

jshell> break;
|  Error:
|  break outside switch or loop
|  break;
|  ^----^
```

98.4.3. La définition et l'invocation de méthodes

Il est possible de définir une méthode directement sans avoir à l'inclure dans une classe en préambule. Il suffit de saisir la signature et le corps de la méthode.

Résultat :

```
jshell> long ajouter(int a, int b) {
   ...> return a + b;
   ...> }
|  created method ajouter(int,int)
```

Attention, les points-virgules de terminaison sont obligatoires dans les blocs de code.

Une fois la méthode définie, il est possible de l'invoquer à n'importe quel moment tant que la session n'est pas fermée.

Résultat :

```
jshell> ajouter(2, 3)
$13 ==> 5
```

Il est possible de remplacer la définition d'une méthode simplement en saisissant son code.

Résultat :

```
jshell> void saluer() {
   ...> System.out.println("Bonjour");
   ...> }
|  created method saluer()

jshell> saluer()
Bonjour

jshell> void saluer() {
   ...> System.out.println("Hello");
   ...> }
|  modified method saluer()

jshell> saluer()
Hello
```

La création d'une méthode statique n'est pas possible : le mot clé static est ignoré.

Résultat :

```
jshell> public static void afficher(String msg) { System.out.println(msg); }
|  Warning:
|  Modifier 'static'  not permitted in top-level declarations, ignored
|  public static void afficher(String msg) { System.out.println(msg); }
|  ^-----------^
|  created method afficher(String)
```

Il est possible de définir une classe abstraite.

Exemple (code Java 9) :

```
jshell> public abstract class MaClasseAbstraite {}
|  created class MaClasseAbstraite
```

Il n'est pas possible d'utiliser le modificateur synchronized.

Exemple (code Java 9) :

```
jshell> synchronized void afficher() {}
|  Error:
|  Modifier 'synchronized'  not permitted in top-level declarations
|  synchronized void afficher() {}
|  ^----------^
```

98.4.4. La création de classes et d'instances

JShell permet la création de classes, d'interfaces, d'énumérations et de records. Il suffit de saisir l'ensemble des lignes de code de la classe. Une fois l'intégralité du code saisi, JShell évalue le code saisi et affiche le résultat de cette évaluation.

Résultat :

```
jshell> class MaClasse {
   ...> public void afficher(String message) {
   ...> System.out.println(message);
   ...> }
   ...> }
|  created class MaClasse
```

La création d'une classe peut être fastidieuse : une erreur de saisie n'est détectée que lors de l'évaluation de l'intégralité du code saisi.

Pour faciliter la saisie ou la modification d'une classe, il est plus simple d'utiliser la commande /edit.

Il est aussi possible de charger et d'évaluer le code d'une classe en utilisant la commande /open.

Une fois définie, il est possible d'utiliser la classe.

Résultat :

```
jshell> MaClasse mc = new MaClasse()
mc ==> MaClasse@370736d9

jshell> mc.afficher("Bonjour");
Bonjour
```

Il est aussi possible de faire un copier/coller du code de la classe.

Résultat :

```
jshell> class Personne {  private String nom;  private String prenom;
  public Personne(String nom, String prenom) {  super();  this.nom = nom;  this.prenom
   = prenom;   } public Personne() {   super(); }  public String getNom() {    return
  nom ;   }    public void setNom(String nom) {     this.nom = nom;   }    public String
  getPrenom() {     return prenom;   }    public void setPrenom(String prenom) {
  this.prenom = prenom;   } }
|  created class Personne
```

Il n'est pas possible de déclarer des packages : tout le code saisi et exécuté dans JShell est placé dans un package transitoire interne.

Résultat :

```
jshell> package mon.package;
|  Error:
|  illegal start of expression
|  package mon.package;
|  ^
```

Il n'est pas possible d'utiliser directement this dans les fragments

Résultat :

```
jshell> System.out.println(this)
|  Error:
|  non-static variable this cannot be referenced from a static context
|  System.out.println(this)
|                     ^--^
```

98.5. Les commandes de JShell

JShell propose plusieurs commandes pour :

- obtenir des informations sur la session,
- interagir avec l'environnement,
- configurer l'environnement

Toutes les commandes commencent par un caractère « / », ce qui les distinguent des fragments qui eux seront évalués : elles sont de la forme /xxx où xxx est le nom de la commande.

L'utilisation d'un REPL est différente de celle d'un IDE. Comme les fragments sont évalués au fur et à mesure, plusieurs commandes permettent d'obtenir la liste des éléments saisis dans la session courante : variables, méthodes, types.

JShell propose de nombreuses commandes :

Commande	Rôle
/vars	Afficher la liste des variables définies dans la session courante /vars [<name or id>|-all|-start]
/methods	Afficher la liste des méthodes définies dans la session courante /methods [<name or id>|-all|-start]
/list	Afficher la liste des fragments saisis de la session courante /list [<name or id>|-all|-start]
/imports	Afficher la liste des imports définis dans la session courante /imports
/types	Afficher la liste des types définis dans la session courante

	/types [<name or id>|-all|-start]
/edit	Editer un fragment précisé par son nom ou son id /edit <name or id>
/exit	Fermer la session courante et arrêter JShell /exit
/drop	Supprimer un fragment précisé par son nom ou son id /drop <name or id>
/open	Ouvrir un fichier dont le contenu est ajouté dans la session /open <file>
/save	Enregistrer les éléments de la session dans un fichier /save [-all|-history|-start] <file>
/env	Configurer le context d'évaluation (classpath, modulepath) /env [-class-path <path>] [-module-path <path>] [-add-modules <modules>] ...
/reset	Réinitialiser la session /reset [-class-path <path>] [-module-path <path>] [-add-modules <modules>]...
/history	Afficher l'historique des fragments et commandes saisis /history
/help	Afficher l'aide en ligne /help [<command>|<subject>]
/set	Configurer l'outils /set editor|start|feedback|mode|prompt|truncation|format ...
/reload	Recharger la session : effectue un reset et réexécute les fragments de la session /reload [-restore] [-quiet] [-class-path <path>] [-module-path <path>]...

Il est possible de ne saisir que les premiers caractères de la commande tant que cela n'induit pas d'ambiguïtés.

Résultat :

```
jshell> /ex
|
Goodbye
```

Si une ambiguïté est détectée alors un message d'erreur est affiché.

Résultat :

```
jshell> /e
|  Command: '/e' is ambiguous: /edit, /exit, /env
|  Type /help for help.
```

98.5.1. La commande /help

La commande /help ou /? permet d'obtenir l'aide en ligne proposant notamment la liste des commandes utilisables dans JShell.

Résultat :

```
jshell> /help
|  Type a Java language expression, statement, or declaration.
|  Or type one of the following commands:
|  /list [<name or id>|-all|-start]
|       list the source you have typed
|  /edit <name or id>
|       edit a source entry referenced by name or id
|  /drop <name or id>
|       delete a source entry referenced by name or id
|  /save [-all|-history|-start] <file>
|       Save snippet source to a file.
|  /open <file>
|       open a file as source input
|  /vars [<name or id>|-all|-start]
|       list the declared variables and their values
...
```

Il est possible d'obtenir l'aide en ligne d'une commande en utilisant la commande /help suivi du nom de la commande.

Résultat :

```
jshell> /help set
|
|                                    /set
|                                    ====
|
|  Set the jshell tool configuration information, including:
|  the external editor to use, the startup definitions to use, a new feedback mode,
|  the command prompt, the feedback mode to use, or the format of output.
|
...
```

Il est aussi possible d'obtenir l'aide en ligne d'une option d'une commande en utilisant la commande /help suivi du nom de la commande puis de l'option.

Résultat :

```
jshell> /help set start
|
|                                 /set start
|                                 ==========
|
|  Set the startup configuration -- a sequence of snippets and commands read at startup:
|
|       /set start [-retain] <file>...
|
|       /set start [-retain] -default
|
|       /set start [-retain] -none
...
```

98.5.2. La commande /exit

La commande /exit permet de quitter JShell

Résultat :

```
C:\Program Files\Java\jdk-9\bin>jshell
```

```
|  Welcome to JShell -- Version 9-ea
|  For an introduction type: /help intro
jshell> /exit
|  Goodbye
C:\Program Files\Java\jdk-9\bin>
```

98.5.3. La commande /list

La commande /list affiche la liste des fragments saisis.

Résultat :

```
jshell> /list

   1 : 1+2
   2 : $1+3
   3 : System.out.println($2)
   4 : void afficher(String m) {
       System.out.println(m);
       }
   5 : afficher("bonjour")
```

Chacun des fragments est préfixé par son identifiant.

L'option -start affiche les fragments des scripts de démarrage. Leur identifiant commence par une lettre « s ».

Résultat :

```
jshell> /list -start

  s1 : import java.io.*;
  s2 : import java.math.*;
  s3 : import java.net.*;
  s4 : import java.nio.file.*;
  s5 : import java.util.*;
  s6 : import java.util.concurrent.*;
  s7 : import java.util.function.*;
  s8 : import java.util.prefs.*;
  s9 : import java.util.regex.*;
 s10 : import java.util.stream.*;
```

L'option -all de la commande /list affiche les fragments des scripts de démarrage et tous les fragments saisis. Par défaut, la commande /list n'affiche pas les fragments dont l'évaluation a échouée.

En utilisant l'option -all, il est possible d'obtenir l'intégralité des fragments saisis, indépendamment du résultat de leur évaluation, ainsi que les fragments exécutés par les scripts de démarrage.

L'affichage de chaque identifiant de chaque fragment dépend du type d'éléments :

- sNN pour des fragments issus des scripts de démarrage
- eNN pour de fragments dont l'évaluation à générer une erreur
- NN pour les fragments dont l'évaluation à réussie

Résultat :

```
jshell> /list -all

  s1 : import java.io.*;
  s2 : import java.math.*;
  s3 : import java.net.*;
  s4 : import java.nio.file.*;
  s5 : import java.util.*;
  s6 : import java.util.concurrent.*;
  s7 : import java.util.function.*;
  s8 : import java.util.prefs.*;
  s9 : import java.util.regex.*;
 s10 : import java.util.stream.*;
   1 : 1+2
   2 : $1+3
   3 : System.out.println($2)
   4 : void afficher(String m) {
       System.out.println(m);
       }
   5 : afficher("bonjour")
  e1 : xxx
```

La commande /list n'affiche que les fragments : pour obtenir la liste de fragments et des commandes saisis il faut utiliser la commande /history.

98.5.4. La commande /history

La commande /history affiche tous les fragments et toutes les commandes saisis dans la session courante, dans leur ordre de saisie.

Résultat :

```
jshell> /history

/list -start
1+2
$1+3
System.out.println($2)
void afficher(String m) {
System.out.println(m);
}
afficher("bonjour")
/list
/list -all
/history
```

98.5.5. La commande /imports

Par défaut, JShell permet d'utiliser les imports des principaux packages.

La commande /imports permet d'afficher la liste des imports utilisables dans l'environnement.

Résultat :

```
jshell> /imports
|    import java.io.*
|    import java.math.*
|    import java.net.*
|    import java.nio.file.*
|    import java.util.*
|    import java.util.concurrent.*
|    import java.util.function.*
|    import java.util.prefs.*
|    import java.util.regex.*
|    import java.util.stream.*
```

Il est possible d'ajouter un import dans l'environnement simplement en saisissant une expression import.

Résultat :

```
jshell> import java.time.*
```

98.5.6. La commande /vars

Elle permet d'obtenir la liste de toutes les variables définies dans la session courante ainsi que leur valeur.

Exemple (code Java 9) :

```
jshell> /vars
|    int $2 = 3
|    int $3 = 5
|    int nbElements = 10
|    String salutation = "Hello world"
|    String $6 = "HELLO WORLD"
|    String[] $7 = String[2] { "Hello", "world" }
|    int note = 12
|    String[] lettres = String[4] { "A", "B", "C", "D" }
|    long $13 = 5
|    MaClasse mc = MaClasse@370736d9
```

Il est possible de n'afficher que la variable dont l'identifiant suit la commande

Résultat :

```
jshell> /vars 1
|    int $1 = 3
```

Il est possible de n'afficher que la variable dont le nom suit la commande.

Résultat :

```
jshell> /vars i
|    int i = 10
```

L'option -all de la commande /vars affiche toutes les variables actives mais aussi celles dont la définition a échoué ou sont supprimées.

Résultat :

```
jshell> /vars -all
|    int $1 = (not-active)
|    help vars = (not-active)
|    int i = 10
```

L'option -start de la commande /vars affiche toutes les variables définies par les scripts de démarrage.

98.5.7. La commande /methods

Elle permet d'obtenir la liste des méthodes définies dans la session courante. Elle affiche le type de retour, le nom et les types des paramètres entre parenthèses de chaque méthode active de la session.

Résultat :

```
jshell> /methods
|    void afficher(String)
|    long ajouter(int,int)
```

Il est possible de n'afficher que la méthode dont le nom suit la commande.

Résultat :

```
jshell> /methods afficher
|    void afficher(String)
```

L'option -all de la commande /methods affiche toutes les méthodes actives mais aussi celles dont la définition a échoué ou sont supprimées.

L'option -start de la commande /methods affiche toutes les méthodes définies par les scripts de démarrage.

98.5.8. La commande /types

Elle permet d'obtenir la liste des classes, interfaces, énumérations et records définies dans la session courante.

Résultat :

```
jshell> /types
|    class Personne
|    class MaClasse
```

Il est possible de n'afficher que le type dont le nom suit la commande.

Résultat :

```
jshell> /types Personne
|    class Personne
```

L'option -all de la commande /types affiche tous les types actifs mais aussi ceux dont la définition a échoué ou sont supprimés.

L'option -start de la commande /types affiche toutes les méthodes définies par les scripts de démarrage.

98.5.9. La commande /save

Elle permet d'enregistrer les expressions saisies dans la session courante dans le fichier dont le nom suit la commande.

Résultat :

```
jshell> /save moncode.jsh
```

Le fichier créé n'est pas un fichier Java : il n'est pas possible de le compiler avec le compilateur javac.

Résultat :

```
C:\java>type moncode.jsh
1+2
"bonjour"
System.out.println($1 + " "+$2)
int i = 5;
long additionner(int a, int b) {
return (long) a + b;
}
additionner(1,2)
$2.toUpperCase()
```

Il est possible de modifier le fichier en utilisant un éditeur tout en prenant garde de le conserver valide.

Le fichier pourra être rechargé dans JShell en utilisant la commande /open.

Il est possible de limiter les fragments sauvegardés à ceux précisé avant le nom du fichier

Résultat :

```
jshell> /save 2-4 monfichier.jsh
```

L'option -all de la commande /save enregistre tous les fragments saisis même ceux dont la définition a échoué ou sont supprimés ainsi que les éléments des scripts de démarrage.

L'option -history de la commande /save enregistre tous les fragments et les commandes saisis.

L'option -start de la commande /save enregistre les éléments des scripts de démarrage dans un fichier.

98.5.10. La commande /open

Elle permet de lire les expressions contenues dans le fichier dont le nom suit la commande pour les ajouter dans la session en les évaluant. Le résultat de cette évaluation peut échouer et conduire à une erreur.

Résultat :

```
jshell> /open moncode.jsh
```

La commande /open peut aussi utiliser pour charger un fichier .java contenant la définition d'une classe.

Exemple (code Java 9) :

```
jshell> /types

jshell> /open Hello.java

jshell> /types
|    class Hello
```

JShell propose trois scripts prédéfinis qu'il est possible de charger avec la commande /open :

- DEFAULT : contient les imports par défaut
- PRINTING : contient des surcharges des méthodes print(), println() et printf() pour faciliter l'affichage d'informations
- JAVASE : contient les imports de tous les packages de Java SE

Résultat :

```
jshell> /open PRINTING
jshell> println("coucou")
coucou
```

98.5.11. La commande /edit

JShell permet d'éditer un nouveau fragment ou un fragment précédemment saisi pour le modifier dans un éditeur basic fournit par JShell ou par l'éditeur externe de son choix. L'édition est alors plus simple notamment si le fragment est composé de plusieurs lignes.

Par défaut, une boîte de dialogue Swing est affichée pour permettre de saisir le nouveau code du fragment : c'est une simple boîte de dialogue qui permet la saisie du code du fragment en multilignes.

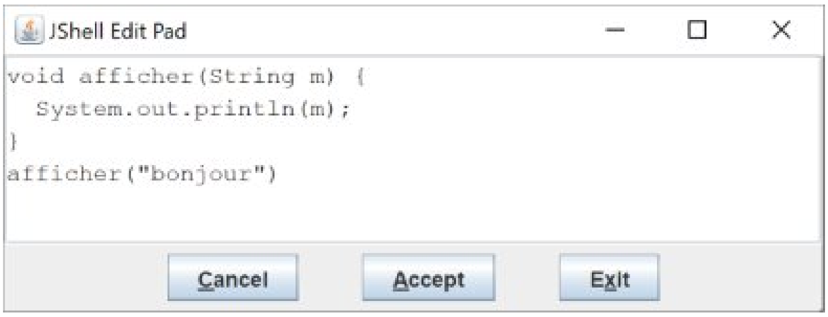
La commande /edit peut être utilisée sur un fragment valide ou invalide. Elle est particulièrement utile pour éditer un fragment contenu sur plusieurs lignes et notamment pour en corriger un dont l'évaluation a échoué dans avoir à ressaisir son contenu intégral.

Si la session ne contient aucun fragment, alors pour créer une nouvelle portion de code à partir de l'éditeur, il suffit d'invoquer la commande /edit sans argument.

Résultat :

```
jshell> /reset
|  Resetting state.
jshell> /edit
```

L'éditeur de texte se lance avec un contenu vide puisque l'état de la session a été réinitialisée. Il est alors possible de saisir les lignes de code du fragment dans l'éditeur.



```
void afficher(String m) {
  System.out.println(m);
}
afficher("bonjour")
```

Lorsque l'on clique sur « Accept », la méthode afficher() est ajoutée et elle est exécutée avec le paramètre bonjour.

Résultat :

```
jshell> /edit
|  created method afficher(String)
bonjour
```

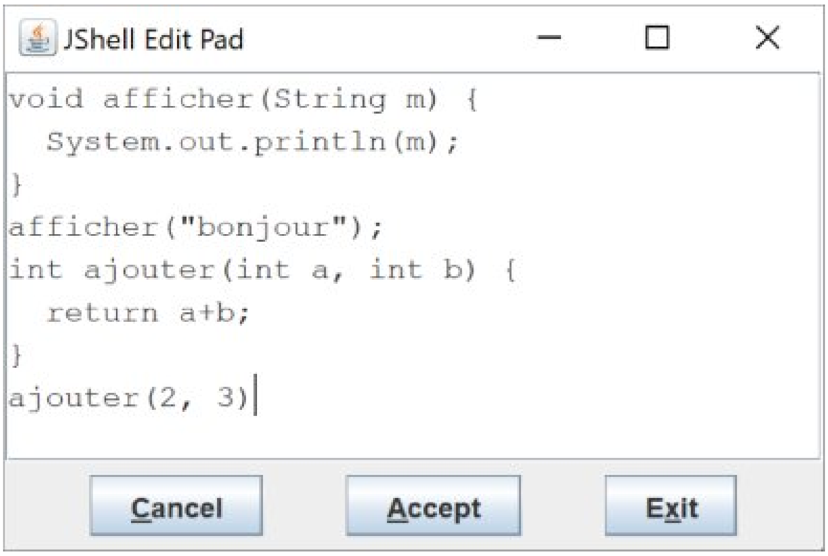
Pour valider les modifications, il suffit de cliquer sur le bouton « Accept » puis sur le bouton « Exit » pour fermer la boîte de dialogue.

Lorsque l'on quitte l'éditeur, JShell affiche un message si l'élément est modifié ou remplacé.

Résultat :

```
jshell> /edit afficher
|  modified method afficher(String)
```

Cela fonctionne aussi si la session contient déjà des fragments en invoquant la commande /edit : dans ce cas, l'éditeur s'ouvre avec l'intégralité des fragments.



```
void afficher(String m) {
  System.out.println(m);
}
afficher("bonjour");
int ajouter(int a, int b) {
  return a+b;
}
ajouter(2, 3)
```

Lors du clique sur « Accept », la nouvelle méthode est ajoutée et exécutée : les autres fragments restants inchangés, ils ne sont pas exécutés. Les fragments existants sont modifiés ou remplacés et les nouveaux fragments sont ajoutés.

Résultat :

```
jshell> /edit
|  created method ajouter(int,int)
$4 ==> 5
```

Remarque : tant que l'éditeur n'est pas fermé, il n'est pas possible de saisir de fragments ou de commande dans JShell.

Il est possible de préciser, en paramètre de la commande /edit, l'identifiant du fragment à modifier.

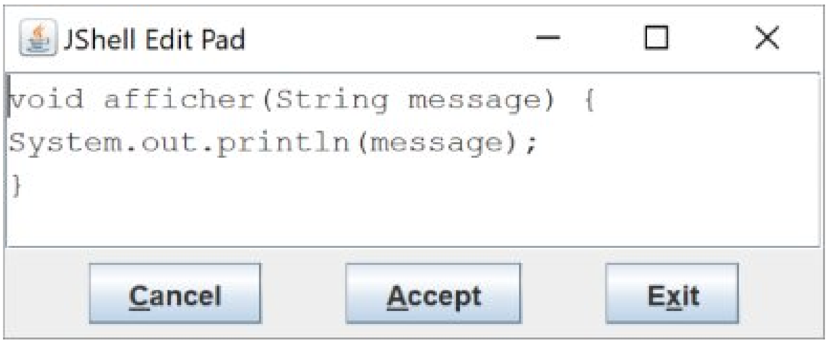
Exemple (code Java 9) :

```
jshell> /list

   1 : int valeur = 10;
   2 : String message = "Bonjour";
   3 : void afficher(String message) {
       System.out.println(message);
       }

jshell> /edit 3
```

Il est aussi possible d'indiquer le nom de l'élément à modifier à la commande /edit.

Exemple (code Java 9) :

```
jshell> /edit afficher
```

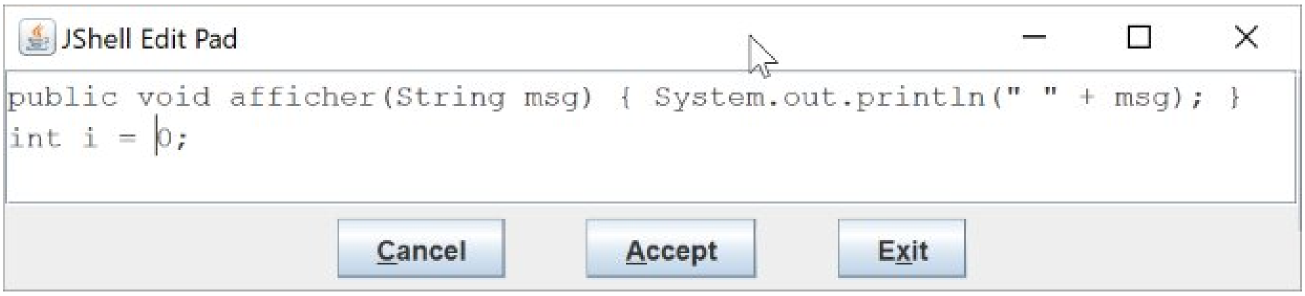


```
void afficher(String message) {
System.out.println(message);
}
```

Il est possible de passer en paramètre de la commande /edit plusieurs ID des éléments à éditer.

Exemple (code Java 9) :

```
jshell> /list

   1 : public void afficher() {}
   3 : public void afficher(String msg) { System.out.println(" " + msg); }
   4 : int i = 0;
   5 : String msg = "";

jshell> /edit 3 4
```



```
public void afficher(String msg) { System.out.println(" " + msg); }
int i = 0;
```

En cours d'édition du code, il est possible de cliquer sur le bouton « Accept » : JShell va réévaluer le code en cours d'édition sans fermer l'éditeur. Cela permet facilement de valider le code saisi sans avoir à quitter l'éditeur. Si l'évaluation du code se fait sans erreur alors un nouvel ID est affecté à la portion de code.

Exemple (code Java 9) :

```
jshell> /edit
|  created method afficher()
|  created method afficher(String)

jshell> /list

   1 : public void afficher() {}
   2 : public void afficher(String msg) { System.out.println(msg); }
```

Il est possible de configurer un éditeur de texte externe en utilisant la commande /set avec l'option editor.

98.5.12. La commande /drop

La commande /drop permet de supprimer un ou plusieurs snippets précédemment saisis.

Les snippets concernés peuvent être précisés en utilisant un identifiant, une plage d'identifiants, un nom ou une combinaison de ces éléments.

Exemple (code Java 9) :

```
jshell> /list

   1 : public void afficher() {}
   3 : public void afficher(String msg) { System.out.println(" " + msg); }
   5 : String msg = "";
   6 : int i = 10;

jshell> /drop 5
|  dropped variable msg

jshell> /list
```

```
   1 : public void afficher() {}
   3 : public void afficher(String msg) { System.out.println(" " + msg); }
   6 : int i = 10;

jshell> /drop i
|  dropped variable i

jshell> /list

   1 : public void afficher() {}
   3 : public void afficher(String msg) { System.out.println(" " + msg); }
```

98.5.13. La commande /<id>, /! et /-<n>

Plusieurs commandes permettent de réexécuter un fragment précédemment exécuté.

Commande	Rôle
/<id>	Réexécuter le fragment dont l'identifiant est précisé
/!	Réexécuter le dernier fragment
/-<n>	Réexécuter le n-ième fragment précédent

Exemple (code Java 9) :

```
jshell> /reset
|  Resetting state.

jshell> println("1")
1

jshell> println("2")
2

jshell> println("3")
3

jshell> /list

   1 : println("1")
   2 : println("2")
   3 : println("3")

jshell> /1
println("1")
1
jshell> /list

   1 : print("1")
   2 : print("2")
   3 : print("3")
   4 : print("1")

jshell> /-2
print("3")
3
jshell> /!
print("3")
3
```

98.5.14. La commande /set

Elle permet de configurer l'environnement de JShell.

98.5.14.1. L'option editor

L'option editor permet de configurer l'éditeur de texte à utiliser par la commande /edit. Elle possède plusieurs options qui peuvent se combiner :

/set editor [-retain] [-wait] <command>

/set editor [-retain] -default

/set editor [-retain] -delete

L'option -retain permet de rendre la configuration permanente. Ainsi, celle-ci sera utilisée dans les prochaines sessions ouvertes.

L'argument <command> permet de préciser la commande du système d'exportation à exécuter pour lancer l'éditeur externe. Des arguments peuvent être ajoutés après la commande en les séparant par des espaces. Lors de l'utilisation d'un éditeur externe, un fichier temporaire est créé et ce dernier est fourni en dernier argument de la commande.

Résultat :

```
jshell> /set editor notepad
|  Editor set to: notepad
```

L'option -default permet de demander l'utilisation de l'éditeur de texte fourni par défaut dans JShell.

L'option -delete permet de supprimer la configuration actuelle et de revenir à la configuration précédente.

L'option -wait permet de demander au développeur d'indiquer lorsque le mode d'édition doit se terminer. Cette option peut être utile lorsque l'éditeur utilisé ne bloque pas JShell lors de son utilisation.

Pour connaître la configuration de l'éditeur actuelle, il suffit simplement d'utiliser la commande /set editor

Résultat :

```
jshell> /set editor
|  /set editor -default
```

98.5.14.2. L'option start

L'option start permet de configurer un ensemble de fragments ou de commandes exécuté au démarrage d'une session. Des fragments et commandes précisés via cette option seront exécutés lors de la l'invocation des commandes /reset, /reload et /env.

La commande /set start possède plusieurs paramètres qui peuvent se combiner :

/set start [-retain] <file>...

/set start [-retain] -default

/set start [-retain] -none

L'option -retain permet de rendre la configuration permanente : celle-ci sera utilisée dans les prochaines sessions de JShell.

L'option -default permet de demander la configuration de démarrage par défaut : celle-ci ne concerne que l'import des quelques packages par défaut.

L'option -none permet de demander qu'aucune configuration de démarrage ne soit utilisée.

L'argument <file> peut être un fichier qui contient les fragments des commandes ou une des trois valeurs prédéfinies dans JShell :

	Rôle
DEFAULT	Importer les packages par défaut
PRINTING	Définir de méthodes pour faciliter l'affichage de données
JAVASE	Importer tous les packages de Java SE

Il est possible de préciser plusieurs fichiers ou valeur prédéfinies

Résultat :

```
jshell> /set start -retain DEFAULT PRINTING
```

Pour afficher la configuration de démarrage, il suffit d'utiliser la commande /set start

Résultat :

```
jshell> /set start
|  /set start -retain DEFAULT PRINTING
|  ---- DEFAULT ----
|  import java.io.*;
|  import java.math.*;
|  import java.net.*;
|  import java.nio.file.*;
|  import java.util.*;
|  import java.util.concurrent.*;
|  import java.util.function.*;
|  import java.util.prefs.*;
|  import java.util.regex.*;
|  import java.util.stream.*;
|  ---- PRINTING ----
|  void print(boolean b) { System.out.print(b); }
|  void print(char c) { System.out.print(c); }
|  void print(int i) { System.out.print(i); }
...
```

98.5.15. L'achèvement des commandes

JShell propose une fonctionnalité qui nous permet de lui demander de terminer la commande saisie en tapant le début de la commande puis en appuyant sur la touche TAB.

Par exemple, il suffit de saisir un caractère slash puis d'appuyer sur la touche tabulation pour obtenir la liste des commandes utilisables.

Résultat :

```
jshell> /
/!          /?          /drop       /edit       /env        /exit       /help       /history    /imports
/list       /methods    /open       /reload     /reset      /save       /set        /types      /vars

<press tab again to see synopsis>

jshell> /
```

Pour obtenir une liste détaillée, il suffit d'appuyer de nouveau sur la touche tabulation comme proposée à la fin de la liste des commandes.

Résultat :

```
jshell> /
/!        /?          /drop     /edit       /env        /exit     /help     /history    /imports
/list     /methods    /open     /reload     /reset      /save     /set      /types      /vars
```

```
<press tab again to see synopsis>

jshell> /
/!
re-run last snippet

/-<n>
re-run n-th previous snippet

/<id>
re-run snippet by id

/?
get information about jshell

/drop
delete a source entry referenced by name or id

/edit
edit a source entry referenced by name or id

/env
view or change the evaluation context

/exit
exit jshell

/help
get information about jshell

/history
history of what you have typed

/imports
list the imported items

/list
list the source you have typed

/methods
list the declared methods and their signatures

/open
open a file as source input

/reload
reset and replay relevant history -- current or previous (-restore)

/reset
reset jshell

/save
Save snippet source to a file.

/set
set jshell configuration information

/types
list the declared types

/vars
list the declared variables and their values

<press tab again to see full documentation>

jshell> /
```

Il est possible de saisir le début de la commande et d'appuyer sur la touche TAB.

Résultat :

```
jshell> /l
```

Si JShell peut terminer la saisie du code avec une unique solution, alors la commande est terminée.

Résultat :

```
jshell> /list
```

Si JShell ne peut pas terminer la saisie du code avec une unique solution, alors les différentes possibilités sont affichées.

Résultat :

```
jshell> /re
/reload    /reset

<press tab again to see synopsis>
```

Pour obtenir plus d'informations sur les commandes affichées, il suffit d'appuyer à nouveau sur la touche TAB.

Résultat :

```
jshell> /re
/reload
reset and replay relevant history -- current or previous (-restore)

/reset
reset jshell

<press tab again to see full documentation>
```

En pressant de nouveau la touche TAB, une information détaillée est affichée.

Résultat :

```
jshell> /re
/reload
Reset the jshell tool code and execution state then replay each valid snippet
and any /drop commands in the order they were entered.

/reload
    Reset and replay the valid history since jshell was entered, or
    a /reset, or /reload command was executed -- whichever is most
    recent.

/reload -restore
    Reset and replay the valid history between the previous and most
    recent time that jshell was entered, or a /reset, or /reload
    command was executed. This can thus be used to restore a previous
    jshell tool session.

/reload [-restore] -quiet
    With the '-quiet' argument the replay is not shown.  Errors will display.

Each of the above accepts context options, see:

    /help context


<press tab again to see next page>
```

L'utilisation de la touche TAB fonctionne aussi pour les options des commandes

Résultat :

```
jshell> /list -
-all       -history   -start

<press tab again to see synopsis>

jshell> /list -
```

Comme pour le nom d'une commande, l'appui une seconde fois sur la touche TAB affiche une description des options.

Si les premières lettres de l'option suffisent pour être terminé, le nom de l'option est complété.

Lors de la saisie de commande qui requière un nom de fragment, il est aussi possible d'utiliser la touche TAB pour compléter le nom d'un élément précédemment créé dans la session JShell.

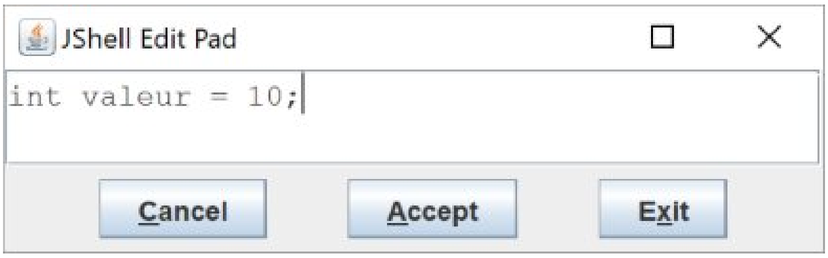
Résultat :

```
jshell> /list

jshell> int valeur = 10;
valeur ==> 10

jshell> /ed v
```

L'appui sur la touche TAB complète le nom du fragment

Résultat :

```
jshell> /ed valeur
```



```
int valeur = 10;
```

La touche TAB peut aussi être utilisée pour compléter le nom d'un fichier requis par une commande.

Résultat :

```
jshell> /open
Open a file and read its contents as snippets and commands.

/open <file>
    Read the specified file as jshell input.
```

Un appui de nouveau sur la touche TAB affiche la liste des fichiers présents dans le répertoire courant.

Résultat :

```
jshell> /open
C:\                                    Hello.java
test.jsh                               test_java9/

<press tab again to see synopsis>

jshell> /open
```

La touche TAB peut aussi être utilisée pour compléter le nom partiel d'un fichier.

Résultat :

```
jshell> /open test
test.jsh      test_java9/

jshell> /open test.jsh
```

98.5.16. L'abréviation des commandes

Il est possible de réduire la quantité de caractères à saisir en utilisant les abréviations de commandes. Cette abréviation consiste à n'utiliser sur les premières lettres tant que celle-ci constitue le début d'une unique commande.

Résultat :

```
jshell> /l
   1 : 1+2
   2 : "bonjour"
jshell> /list
   1 : 1+2
   2 : "bonjour"
```

Dans l'exemple ci-dessus, la commande /list peut être abrégée /l car c'est la seule commande qui commence par la lettre « L ».

Cette abréviation peut aussi s'utiliser sur les options des commandes

Résultat :

```
jshell> /l -a
```

La commande ci-dessus est une version abrégée de /list -all

Si l'abréviation saisie n'est pas unique alors un message d'erreur est affiché

Résultat :

```
jshell> /s
|  Command: '/s' is ambiguous: /save, /set
|  Type /help for help.
```

Dans l'exemple ci-dessus, l'abréviation /s n'est pas unique car il existe deux commandes /set et /save comme indiqué dans le message d'erreur.

En cas de doute, il est possible d'utiliser la touche tabulation : si plusieurs commandes commencent par les caractères saisis alors elles sont affichées.

98.6. L'édition

JShell utilise la bibliothèque JLine2 pour gérer la saisie des fragments et des commandes.

Il est possible d'ajouter du texte à la position du curseur simplement en le saisissant.

JShell propose de nombreuses fonctionnalités, généralement sous la forme d'appui sur une touche ou une combinaison de touches pour permettre la saisie des fragments et des commandes

Ces combinaisons peuvent utiliser, entre autres, les touches :

- Ctrl
- Meta : la touche alt sur PC
- Shift

98.6.1. La navigation dans la ligne courante

Plusieurs touches ou combinaisons de touches peuvent être utilisées pour naviguer dans la ligne courante en cours de saisie.

Touche ou combinaison de touches	Rôle
Flèche gauche Ctrl+B	Se déplacer d'un caractère à gauche
Flèche droite Ctrl+F	Se déplacer d'un caractère à droite
Retour chariot (Return)	Valider la ligne courante
Home Ctrl+A	Se déplacer au début de la ligne
Fin (End) Ctrl+E	Se déplacer au début de la ligne
Meta+B	Se déplacer au début du mot précédent
Meta+F	Se déplacer à la fin du mot suivant

Remarque : le parcours de l'historique de navigation contenant un fragment multilignes se fait ligne par ligne.

98.6.2. La navigation dans l'historique

JShell conserve un historique des fragments et des commandes saisies dans la session.

Plusieurs touches ou combinaisons de touches peuvent être utilisées pour naviguer dans l'historique.

Touche ou combinaison de touches	Rôle
Flèche haut Ctrl+B	Remplacer la ligne courante par le contenu de la ligne précédente dans l'historique
Flèche bas Ctrl+B	Remplacer la ligne courante par le contenu de la ligne suivante dans l'historique
Ctrl + flèche haut	Remplacer la ligne courante par le contenu de la première ligne du fragment ou commande précédente dans l'historique

Remarque : si un fragment est composé de plusieurs lignes alors l'utilisation des flèches haut et bas permet de naviguer dans chacune des lignes du fragments

Cette navigation permet de ressaisir un fragment ou une commande, éventuellement avec une modification, simplement en appuyant sur la touche Entrée (Enter). Cela évite d'avoir à ressaisir l'intégralité de la ligne.

Si la session est réinitialisée (par exemple en utilisant la commande /reset), l'historique est conservé et il est donc toujours possible de naviguer dedans.

98.6.3. La modification de la ligne courante

Le texte saisi est inséré à la position du curseur dans la ligne courante.

Plusieurs combinaisons de touches peuvent être utilisées lors de la saisie d'une ligne :

Combinaison de touches	Rôle
Suppr (Del ou Delete)	Supprimer le caractère après le curseur
Backspace	Supprimer le caractère avant le curseur
Ctrl+K	Supprimer le reste de la ligne à partir du curseur
Ctrl+U	Supprimer le début de la ligne jusqu'au curseur
Alt+D	Supprimer le reste du mot à partir du curseur
Ctrl+W	Supprimer le texte du curseur jusqu'à l'espace précédent
Ctrl+Y	Coller la dernière portion de code supprimée
Alt+Y	Après un Ctrl+Y, permettre de sélectionner le texte précédemment supprimé et ce de manière cyclique
Entrée (Enter)	Valider la saisie ou la modification pour évaluation

Que la ligne soit modifiée ou pas, l'appui sur la touche « Entrée » permet de demander l'évaluation du fragment.

98.6.4. La recherche et les autres fonctionnalités

JShell permet de faire une recherche dans l'historique des fragments et commandes saisis. Cela permet de plus facilement retrouver une ligne saisie sans avoir à naviguer ligne par ligne dans l'historique.

Pour demander une recherche, il faut utiliser la combinaison de touches Ctrl+R, puis de saisir tout ou partie de l'élément recherché. L'invite de commande est remplacée par celle de recherche :

Résultat :

```
(reverse-i-search)`':
```

La recherche commence à partir de la dernière ligne saisie et remonte dans l'historique.

Résultat :

```
jshell> /history

void saluer() {
System.out.println("Bonjour");
}
saluer()
void saluer() {
System.out.println("Hello");
}
saluer()
/history

(reverse-i-search)`sal': saluer()
```

La ligne courante trouvée par la recherche est affichée après le caractère deux-points. La recherche est incrémentale : les propositions s'affichent au fur et à mesure de la saisie du motif à rechercher.

Il est possible de naviguer dans les propositions en utilisant des combinaisons de touches :

- Ctrl+R : pour rechercher l'élément précédent en remontant dans l'historique
- Ctrl+S : pour rechercher l'élément suivant en descendant dans l'historique

98.6.5. La définition et l'utilisation d'une macro

Il est possible de définir une macro en utilisant deux combinaisons de touches :

Raccourci	Rôle
Ctrl+x (	Débuter l'enregistrement d'une macro
Ctrl+x)	Stopper l'enregistrement d'une macro

Pour définie une macro, il faut utiliser la combinaison de touches Ctrl+X (, saisir le texte de la macro et utiliser la combinaison de touches Ctrl+X).

Pour invoquer la macro et obtenir son contenu, il faut utiliser la combinaison de touches Ctrl+x e.

98.6.6. L'utilisation d'un éditeur

La commande /edit permet de modifier un ou plusieurs fragments en utilisant par défaut l'éditeur de texte intégré dans JShell.

Il est possible d'utiliser un éditeur de texte externe pour modifier le code d'un fragment. Il est possible de configurer JShell pour utiliser l'éditeur de texte de son choix.

JShell recherche si une des variables d'environnement ci-dessous est définie. Si c'est le cas, la valeur est utilisée pour désigner l'éditeur externe de texte à utiliser pour modifier les fragments.

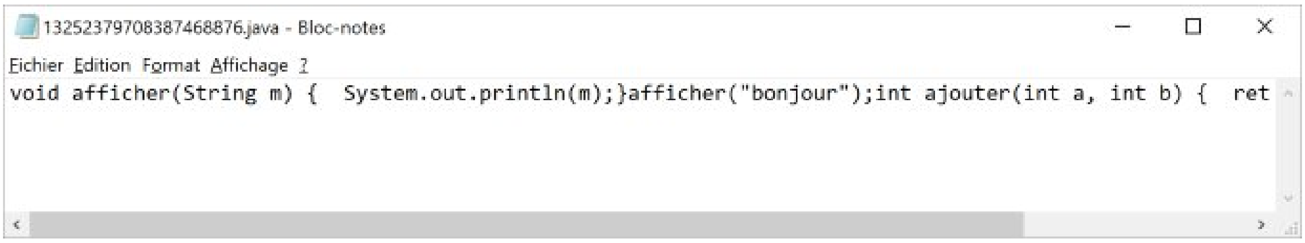
- JSHELLEDITOR
- VISUAL
- EDITOR

Si aucune n'est définie et qu'aucun éditeur n'est configuré alors c'est l'éditeur de texte par défaut fourni avec JShell qui est utilisé. Ce dernier est très simple car il ne permet que de saisir du texte multi-lignes.

La commande /set avec l'option editor permet de configurer l'éditeur externe à utiliser.

Exemple (code Java 9) :

```
jshell> /set editor notepad
|  Editor set to: notepad

jshell> /edit
```



13252379708387468876.java - Bloc-notes
Fichier Edition Format Affichage ?

```
void afficher(String m) {  System.out.println(m);}afficher("bonjour");int ajouter(int a, int b) {  ret
```

La fermeture de l'éditeur externe est obligatoire pour permettre le retour du prompt de JSHell.

98.7. Les fonctionnalités de l'environnement

JShell propose des fonctionnalités pour faciliter la saisie du code.

98.7.1. L'exploration des API et l'achèvement de code

Il suffit de saisir le début du code et d'appuyer sur la touche tab pour obtenir des suggestions contextuelles.

Si une seule suggestion est trouvée, elle est utilisée.

Si plusieurs suggestions sont trouvées alors celles-ci sont affichées.

Exemple (code Java 9) :

```
jshell> Sys<tab>
jshell> System
System

jshell> System.o<tab>
jshell> System.out
out

jshell> System.out.print<tab>
print(     printf(    println(

jshell> System.out.print
```

Il est possible d'utiliser la combinaison de touche « Shift » puis « Tab » pour afficher la liste de surcharges d'une méthode.

Exemple (code Java 9) :

```
jshell> String chaine = "Bonjour"
chaine ==> "Bonjour"

jshell> chaine.substring(<shift><tab>
Signatures:
String String.substring(int beginIndex)
String String.substring(int beginIndex, int endIndex)

<press tab again to see documentation>
```

En appuyant sur la touche « Tab » immédiatement après l'affichage de la liste, la Javadoc de chaque surcharge est affichée successivement en appuyant à chaque fois sur la touche « tab ».

Exemple (code Java 9) :

```
jshell> chaine.substring(<tab>
String String.substring(int beginIndex)
Returns a string that is a substring of this string.The substring begins with the character at
the specified index and extends to the end of this string.
Examples:
     "unhappy".substring(2) returns "happy"
     "Harbison".substring(3) returns "bison"
     "emptiness".substring(9) returns "" (an empty string)


Parameters:
beginIndex - the beginning index, inclusive.

Returns:
the specified substring.

<press tab to see next documentation>

jshell> chaine.substring(<tab>
String String.substring(int beginIndex, int endIndex)
Returns a string that is a substring of this string.The substring begins at the specified
beginIndex and extends to the character at index endIndex - 1 . Thus the length of the
substring is endIndex-beginIndex .
Examples:
     "hamburger".substring(4, 8) returns "urge"
     "smiles".substring(1, 5) returns "mile"


Parameters:
beginIndex - the beginning index, inclusive.
endIndex - the ending index, exclusive.

Returns:
the specified substring.

<press tab again to see all possible completions; total possible completions: 545>

jshell> chaine.substring(
```

Après avoir saisi la première parenthèse d'une méthode, l'utilisation de la touche Shift puis sur la touche Tab (attention, ce n'est pas une combinaison qui a un autre rôle) permet d'afficher les valeurs utilisables et le cas échant les différentes surcharges de la méthode.

Exemple (code Java 9) :

```
jshell> int debut = 10
debut ==> 10

jshell> chaine.subs<tab>
substring(

jshell> chaine.substring(<shift><tab>
debut

Signatures:
String String.substring(int beginIndex)
String String.substring(int beginIndex, int endIndex)

<press tab again to see documentation>

jshell> chaine.substring(
```

98.7.2. Les références à venir (Forward references)

Comme avec JShell le code est évalué séquentiellement au fur et à mesure de sa saisie, il est possible de faire référence à des membres ou des types qui ne sont pas encore défini. Ces références sont alors non résolues jusqu'à leur définition et sont désignées par le terme référence à venir.

JShell supporte les références à venir (forward reference) dans les classes, les méthodes et les variables.

Exemple (code Java 9) :

```
jshell> double calculerMontantTTC(double montantHT) {
   ...> return montantHT * (1 + ((double)TVA /100));
   ...> }
|  created method calculerMontantTTC(double), however, it cannot be invoked until variable
TVA is declared
```

La méthode est définie mais elle ne peut être invoquée tant que la variable TVA n'est pas définie.

Exemple (code Java 9) :

```
jshell> int TVA = 20
TVA ==> 20

jshell> calculerMontantTTC(100)
$8 ==> 120.0
```

C'est notamment le cas si le type de la redéfinition n'est pas compatible avec le précédent type. Il est important de noter que JShell nous informe par avance des incompatibilités uniquement dans le mode de feedback verbose. Dans les autres modes, l'erreur n'est affichée qu'à l'exécution.

Exemple (code Java 9) :

```
jshell> int TVA = 20
TVA ==> 20

jshell> /set feedback verbose
|  Feedback mode: verbose

jshell> BigInteger TVA = new BigInteger("20")
TVA ==> 20
|  replaced variable TVA : BigInteger
|    update modified method calculerMontantTTC(double) which cannot be invoked until
     this error is corrected:
|      incompatible types: java.math.BigInteger cannot be converted to double
|      return montantHT * (1 + ((double)TVA /100));
|                                       ^-^
|    update overwrote variable TVA : int
```

Les références à venir peuvent aussi concerner des méthodes ou des types.

Exemple (code Java 9) :

```
jshell> class MaClasse {
   ...> AutreClasse autre;
   ...> }
|  created class MaClasse, however, it cannot be referenced until class AutreClasse is declared

jshell> new MaClasse()
|  Error:
|  cannot find symbol
|    symbol:   class MaClasse
|  new MaClasse()
|      ^------^
```

```
jshell> class AutreClasse { }
|  created class AutreClasse

jshell> new MaClasse()
$3 ==> MaClasse@490ab905
```

98.7.3. La redéclaration d'une variable ou d'une méthode

JShell est un outil qui facilite l'expérimentation de code Java : il permet donc facilement de modifier ou de redéfinir une variable, une méthode ou une classe. Il suffit de ressaisir la définition du fragment de code. Si l'élément est déjà défini, sa définition est simplement remplacée.

Il est possible de redéclarer une variable, une méthode ou une classe.

Exemple (code Java 9) :

```
jshell> /list

   1 : int valeur = 10;

jshell> int valeur = 20;
valeur ==> 20

jshell> /list

   2 : int valeur = 20;
```

Il est possible que la redéfinition de l'élément soit incompatible avec la définition existante.

Par exemple, si la redéfinition d'une variable change son type.

Exemple (code Java 9) :

```
jshell> int valeur = 10;
valeur ==> 10

jshell> /list

   1 : int valeur = 10;

jshell> String valeur = "10";
valeur ==> "10"

jshell> /list

   2 : String valeur = "10";
```

Attention, la redéfinition d'un élément peut induire des erreurs à l'exécution du code qui en dépend. Donc ce type d'opération n'est pas forcement sans risque ni conséquence.

Exemple (code Java 9) :

```
jshell> BigInteger TVA = new BigInteger("20")
TVA ==> 20

jshell> calculerMontantTTC(100)
|  attempted to call method calculerMontantTTC(double) which cannot be invoked until
   this error is corrected:
|      incompatible types: java.math.BigInteger cannot be converted to double
|      return montantHT * (1 + ((double)TVA /100));
|                                       ^-^
```

Il est possible de redéfinir une méthode.

Exemple (code Java 9) :

```
jshell> void saluer(String nom) {
   ...> System.out.println("Bonjour "+nom);
   ...> }
|  created method saluer(String)

jshell> /list

   2 : String message = "Bonjour";
   7 : void saluer(String nom) {
       System.out.println("Bonjour "+nom);
       }

jshell> void saluer(String nom) {
   ...> System.out.println("Salut "+nom);
   ...> }
|  modified method saluer(String)

jshell> /list

   2 : String message = "Bonjour";
   8 : void saluer(String nom) {
       System.out.println("Salut "+nom);
       }
```

Lors de la redéfinition d'un élément, JShell affiche comme résultat « modified ... ». Dans le cas d'une méthode, la signature de la méthode ne doit pas être changée pour que cela soit une redéfinition.

Exemple (code Java 9) :

```
jshell> void saluer(String nom) {
   ...> System.out.println("Bonjour "+nom);
   ...> }
|  created method saluer(String)

jshell> /list

   1 : void saluer(String nom) {
       System.out.println("Bonjour "+nom);
       }

jshell> void saluer(int nom) {
   ...> System.out.println("Bonjour "+nom);
   ...> }
|  created method saluer(int)

jshell> /list

   1 : void saluer(String nom) {
       System.out.println("Bonjour "+nom);
       }
   2 : void saluer(int nom) {
       System.out.println("Bonjour "+nom);
       }
```

98.7.4. Les exceptions de type checked

Les exceptions de type checked levées par un fragment de code qui n'est pas dans un bloc de code sont gérées par JShell : il n'est donc pas utile de les capturer.

Il n'est donc pas nécessaire de gérer les exceptions pour les fragments évalués :

Exemple (code Java 9) :

```
jshell> Thread.sleep(5000);

jshell>
```

L'évaluation du fragment de l'exemple ci-dessus fait attendre 5 secondes avant de rendre la main.

Cela s'applique aux fragments au fur et à mesure de leur saisie et évaluation.

Exemple (code Java 9) :

```
jshell> Path path = Paths.get("fichier.txt")
path ==> fichier.txt

jshell> Stream<String> lignes = Files.lines(path)
lignes ==> java.util.stream.ReferencePipeline$Head@3327bd23

jshell> lignes.forEach(l -> System.out.println(l))
ligne1
ligne2
ligne3
```

Dans l'exemple ci-dessous, il n'est pas utile de gérer les exceptions de type IOException.

Dans un bloc de code, les exceptions de type checked doivent être gérées : la gestion des exceptions de type checked est donc toujours requise dans le code d'une méthode.

Exemple (code Java 9) :

```
jshell> class MonThread extends Thread {
   ...> public void run() {
   ...> Thread.sleep(5000);
   ...> }
   ...> }
|  Error:
|  unreported exception java.lang.InterruptedException; must be caught or declared to be thrown
|  Thread.sleep(5000);
|  ^----------------^
```

La gestion des exceptions de type checked est toujours requise dans le code d'une expression Lambda.

Exemple (code Java 9) :

```
jshell> Thread monThread = new Thread(() -> { Thread.sleep(5000); });
|  Error:
|  unreported exception java.lang.InterruptedException; must be caught or declared to be thrown
|  Thread monThread = new Thread(() -> { Thread.sleep(1000); });
|                                       ^----------------^
```

Lorsqu'une exception est levée lors de l'exécution du code d'une méthode, la stacktrace est affichée. Dans la stacktrace, la ligne du fragment concerné est identifiée sous la forme #id_fragment:numero_de_ligne.

Exemple (code Java 9) :

```
jshell> double calculerMoyenne(int... valeurs) {
   ...> long somme = 0;
   ...> for(int val : valeurs) {
   ...> somme += val;
   ...> }
   ...> return somme / valeurs.length;
   ...> }
|  created method calculerMoyenne(int...)

jshell> calculerMoyenne(new int []{5,10,15})
$22 ==> 10.0
|  created scratch variable $22 : double

jshell> calculerMoyenne(new int []{})
|  java.lang.ArithmeticException thrown: / by zero
|        at calculerMoyenne (#21:6)
|        at (#23:1)
```

98.7.5. L'ajout d'un import

JShell facilite l'ajout d'imports lors de la saisie de code : il suffit de saisir le nom d'une classe et d'utiliser la combinaison de touches « Shift + Tab » et d'appuyer sur la touche « i »

Exemple (code Java 9) :

```
jshell> LocalDateTime<Shift+tab, i>
0: Do nothing
1: import: java.time.LocalDateTime
Choice: 1
Imported: java.time.LocalDateTime

jshell> LocalDateTime.now()
$2 ==> 2018-02-20T22:51:05.310054600
```

JShell fait alors plusieurs propositions selon les classes pleinement qualifiées trouvées. Il suffit alors de saisir le chiffre correspond à l'action choisie.

Si aucune classe n'est trouvée, alors un message est affiché pour indiquer l'échec à trouver une classe à importer.

Exemple (code Java 9) :

```
jshell> LocalDataTime<Shift+tab, i>
No candidate fully qualified names found to import.
```

98.7.6. La création d'une variable à partir d'une expression

Il est possible de créer une variable à partir d'une expression. Il suffit de saisir l'expression et d'utiliser la combinaison de touches « Shift + Tab » puis d'appuyer sur la touche « v »

Exemple (code Java 9) :

```
jshell> 1 + 2 <shift+tab, v>
jshell> int | = 1 + 2
```

Le curseur, représenté par le caractère « | » (barre verticale) est placé sur la ligne à l'endroit où il faut saisir le nom de la variable. Le type est déterminé en évaluant l'expression. La valeur affectée à la variable est l'expression elle-même.

Remarque : pour que l'opération puisse être effectuée, l'expression doit être valide. Si ce n'est pas le cas, la combinaison de touches « Shift + Tab » suivi de la touche « v » n'a aucun effet.

Lors de la demande de la création d'une variable à partir d'une expression, il est possible que le type ne soit pas encore importé. Dans ce cas, la combinaison de touche « Shift + Tab » puis « v » permet de créer la variable et l'import du type.

Exemple (code Java 9) :

```
jshell> JFrame frame = new JFrame()
frame ==> javax.swing.JFrame[frame0,0,0,0x0,invalid,hidden, ... tPaneCheckingEnabled=true]

jshell> frame.getSize()<shift+tab, v>
0: Do nothing
1: Create variable
2: import: java.awt.Dimension. Create variable
Choice: 2
Imported: java.awt.Dimension

jshell> Dimension |= frame.getSize()
```

Dans l'exemple ci-dessus, il suffit de sélectionner l'option 2.

Il est aussi possible d'utiliser les imports static.

Exemple (code Java 9) :

```
jshell> import static java.lang.System.out

jshell> out.println("Bonjour")
Bonjour
```

98.8. Les scripts

Il est possible d'utiliser des scripts pour configurer une session JShell (imports, définition de fragments, ...) qui seront dès lors utilisable dans la session tant que celle-ci n'est pas terminée ou réinitialisée.

Un script JShell est un fichier texte qui contient une séquence de fragments et/ou commandes chacun sur une ligne dédiée.

Il existe trois scripts prédéfinis :

Nom	Rôle
DEFAULT	Il définit la déclaration de certains imports dans la session. C'est le script utilisé par défaut si aucun autre script n'est précisé
PRINTING	Il définit des méthodes print(), println() et printf() pour envoyer des données sur la sortie standard
JAVASE	Il importe les packages de Java SE Core (ceux définis dans le module java.se). Cela rend le démarrage de JShell très long

Un script peut être créé grâce à un éditeur de texte ou en utilisant la commande /save.

98.8.1. Les scripts de démarrage

Les scripts de démarrage sont rechargés chaque fois que la session JShell est réinitialisée : soit au démarrage de JShell et à l'utilisation des commandes /reset, /load et /env.

Il est possible d'utiliser un des scripts prédéfinis ou d'écrire ses propres scripts pour réponde à ses besoins.

Si aucun script n'est explicitement configuré, alors c'est le script DEFAULT qui est chargé.

Il est possible de configurer un ou plusieurs scripts de démarrage de différente manière.

Il est possible d'utiliser la commande /set start suivi du nom du fichier qui contient le script ou d'un des scripts prédéfinis. Pour que cela soit pris en compte, il faut réinitialiser la session.

Le script PRINTING permet d'utiliser des surcharges des méthodes print(), println() et printf(), évitant ainsi l'ajout de System.out pour afficher du texte.

Exemple (code Java 9) :

```
C:\java>jshell
|  Welcome to JShell -- Version 9.0.1
|  For an introduction type: /help intro

jshell> /set start PRINTING

jshell> /methods

jshell> /reset
|  Resetting state.

jshell> /methods
|    void print(boolean)
|    void print(char)
|    void print(int)
|    void print(long)
|    void print(float)
|    void print(double)
|    void print(char s[])
|    void print(String)
|    void print(Object)
|    void println()
|    void println(boolean)
|    void println(char)
|    void println(int)
|    void println(long)
|    void println(float)
|    void println(double)
|    void println(char s[])
|    void println(String)
|    void println(Object)
|    void printf(java.util.Locale,String,Object...)
|    void printf(String,Object...)
```

Par défaut, la configuration effectuée par la commande /set n'est effective que pour la session courante. Pour rendre la configuration courante permanente, il faut utiliser l'option -retain de la commande /set.

L'option -retain permet de rendre la configuration courante celle par défaut pour les futures sessions de JShell.

Exemple (code Java 9) :

```
jshell> /set start -retain

jshell>
```

Les scripts de démarrage sont exécutés à chaque réinitialisation de la session. Le contenu du script est chargé une seule fois au moment de l'exécution de la commande /set start puis stocké. Les scripts prédéfinis peuvent être modifiés dans les versions futures de Java.

Il est possible de configurer plusieurs scripts de démarrage avec la commande /set start.

Exemple (code Java 9) :

```
jshell> /set start -retain DEFAULT PRINTING
jshell> /exit
|  Goodbye

C:\java> jshell
|  Welcome to JShell -- Version 9.0.1
|  For an introduction type: /help intro

jshell> println("bonjour")
bonjour

jshell>
```

La commande /set start sans argument permet de visualiser les scripts de démarrage.

Exemple (code Java 9) :

```
jshell> /set start
|  /set start -retain DEFAULT PRINTING
|  ---- DEFAULT ----
|  import java.io.*;
|  import java.math.*;
|  import java.net.*;
|  import java.nio.file.*;
|  import java.util.*;
|  import java.util.concurrent.*;
|  import java.util.function.*;
|  import java.util.prefs.*;
|  import java.util.regex.*;
|  import java.util.stream.*;
|  ---- PRINTING ----
|  void print(boolean b) { System.out.print(b); }
|  void print(char c) { System.out.print(c); }
|  void print(int i) { System.out.print(i); }
|  void print(long l) { System.out.print(l); }
|  void print(float f) { System.out.print(f); }
|  void print(double d) { System.out.print(d); }
|  void print(char s[]) { System.out.print(s); }
|  void print(String s) { System.out.print(s); }
|  void print(Object obj) { System.out.print(obj); }
|  void println() { System.out.println(); }
|  void println(boolean b) { System.out.println(b); }
|  void println(char c) { System.out.println(c); }
|  void println(int i) { System.out.println(i); }
|  void println(long l) { System.out.println(l); }
|  void println(float f) { System.out.println(f); }
|  void println(double d) { System.out.println(d); }
|  void println(char s[]) { System.out.println(s); }
|  void println(String s) { System.out.println(s); }
|  void println(Object obj) { System.out.println(obj); }
|  void printf(java.util.Locale l, String format, Object... args) { System.out.printf(l,
   format, args); }
|  void printf(String format, Object... args) { System.out.printf(format, args); }
```

Il est possible d'utiliser l'option --startup de la commande jshell pour préciser un script de démarrage.

Exemple (code Java 9) :

```
C:\java>jshell --startup PRINTING
|  Welcome to JShell -- Version 9.0.1
|  For an introduction type: /help intro

jshell> /methods
|    void print(boolean)
|    void print(char)
|    void print(int)
|    void print(long)
|    void print(float)
|    void print(double)
|    void print(char s[])
|    void print(String)
|    void print(Object)
|    void println()
|    void println(boolean)
|    void println(char)
|    void println(int)
|    void println(long)
|    void println(float)
|    void println(double)
|    void println(char s[])
|    void println(String)
|    void println(Object)
|    void printf(java.util.Locale,String,Object...)
|    void printf(String,Object...)

jshell> println("bonjour")
bonjour
```

Il est possible de préciser plusieurs scripts en utilisant pour chacun une option --startup.

Exemple (code Java 9) :

```
C:\java>jshell --startup DEFAULT --startup PRINTING
|  Welcome to JShell -- Version 9.0.1
|  For an introduction type: /help intro
```

98.8.2. La création d'un script

Un script peut être créé avec un simple éditeur de texte ou créer avec JShell.

La commande /save permet de créer des scripts à partir de la session courante.

La commande la plus simple est /save suivi d'un nom de fichier. Les fragments de la session courante sont alors enregistrés dans le fichier précisé.

Exemple (code Java 9) :

```
jshell> /save test.jsh

jshell>
```

Il est possible de demander l'enregistrement dans un fichier de l'historique des fragments et des commandes saisies qu'elles soient valides ou invalides en utilisant l'option -history.

Exemple (code Java 9) :

```
jshell> /save -history test.jsh

jshell>
```

L'option -start de la commande /save permet d'enregistrer dans un fichier le script de démarrage courant.

Exemple (code Java 9) :

```
jshell> /save -start test.jsh
jshell>
```

98.8.3. Le chargement d'un script

Il est possible de demander à JShell de charger et d'exécuter un script à son lancement simplement en précisant le nom du fichier en paramètre de la commande jshell.

Exemple (code Java 9) :

```
C:\java> jshell test.jsh
bonjour
|  Welcome to JShell -- Version 9.0.1
|  For an introduction type: /help intro
```

Il est aussi possible d'utiliser la commande /open pour charger et exécuter un script

Exemple (code Java 9) :

```
C:\java> jshell
|  Welcome to JShell -- Version 9.0.1
|  For an introduction type: /help intro

jshell> print("coucou")
coucou
jshell> /open test.jsh
bonjour

jshell> /list

   1 : print("coucou")
   2 : println("bonjour")
```

La commande /open peut être utilisée pour charger l'intégralité d'un fichier .java et permettre d'ajouter dans la session la ou les classes définies dans le fichier fourni en paramètre.

Exemple (code Java 9) :

```
jshell> print("bonjour")
bonjour
jshell> /open Hello.java

jshell> /list

   1 : print("bonjour")
   2 : public class Hello {

         public static void main(String... args) {
           System.out.println("Hello");
         }
       }

jshell>
```

98.9. Les modes de feedback

Un mode de feedback est une configuration nommée qui précise quelles sont les informations restituées par JShell suite à l'évaluation d'un fragment.

98.9.1. L'utilisation d'un mode de feedback

JShell propose par défaut quatre modes de feedback qui définissent le niveau de verbosité des informations fournies par JShell suite à l'évaluation d'un fragment :

Mode	Fragments avec valeur	Déclaration de types et annotations	Mise à jour	Commandes exécutées avec succès	Prompt utilisé
verbose	name ==> value avec déclaration	Oui	Oui	Oui	\njshell>
normal	name ==> value	Oui	Non	Oui	\njshell>
Concise	name ==> value (uniquement pour les expressions)	Non	Non	Non	\njshell>
Silent		Non	Non	Non	->

La commande /set avec l'option feedback permet de sélectionner le mode de feedback à utiliser en le précisant à la suite.

Résultat :

```
jshell> /set feedback verbose
|  Feedback mode: verbose

jshell> int i = 10
i ==> 10
|  modified variable i : int
|    update overwrote variable i : int

jshell> /set feedback normal
|  Feedback mode: normal

jshell> int i = 10
i ==> 10

jshell> /set feedback concise
jshell> int i = 10
jshell> /set feedback silent
-> int i = 10
-> /set feedback normal
|  Feedback mode: normal

jshell>
```

Pour connaitre le mode de feedback actif, il suffit d'utiliser la commande /set avec l'option feedback. Elle est indiquée avec la commande qui permet de l'activer suivi de la liste des modes utilisables.

Résultat :

```
jshell> /set feedback
|  /set feedback normal
|
|  Available feedback modes:
|     concise
|     normal
|     silent
|     verbose
```

Le mode de feedback par défaut est normal.

Pour obtenir le mode de feedback courant, il suffit d'invoquer la commande /set feedback sans options

Exemple (code Java 9) :

```
jshell> /set feedback
```

```
|  /set feedback verbose
|
|  Available feedback modes:
|     concise
|     normal
|     silent
|     verbose
```

Le mode courant est indiqué sur la première ligne sous la forme de la commande utilisée pour configurer ce mode.

98.9.2. La définition d'un mode de feedback

Il est possible de définir son propre mode de feedback :

- le format de l'invite (prompt)
- le niveau d'informations que JShell affichera à la saisie de différents éléments

Les modes de feedback prédéfinis ne peuvent pas être modifiés. Cependant, il est possible de créer un mode de feedback personnalisé.

Un mode de feedback personnalisés peut être créé sur la base de la copie d'un mode existant.

Résultat :

```
jshell> /set mode monmode normal -command
|  Created new feedback mode: monmode
```

Cet exemple créé un mode personnalisé nommé monmode qui est une copie du mode normal. L'option -command demande un retour sur les actions des commandes exécutées. Si ce n'est pas souhaité, il est possible d'utiliser l'option -quiet.

Plusieurs éléments peuvent être configurés dans un mode de feedback

- Prompts (le format des invites) : normal (regular) et de suite de fragment (continuation)
- Truncation : la taille maximale des valeurs affichées
- Format (de retour) : le niveau d'informations que JShell fournit

98.9.2.1. La définition des prompts

La commande /set avec l'option prompt permet de définir les prompts.

La commande /set prompt sans autres options affiche les prompts des différents modes.

Exemple (code Java 9) :

```
jshell> /set prompt
|  /set prompt normal "\njshell> " "   ...> "
|  /set prompt silent "-> " ">> "
|  /set prompt concise "jshell> " "   ...> "
|  /set prompt verbose "\njshell> " "   ...> "
```

Il est possible de demander l'utilisation des prompts d'un mode particulier en utilisant la commande /set prompt suivi du mode souhaité.

Exemple (code Java 9) :

```
jshell> /set prompt normal
|  /set prompt normal "\njshell> " "   ...> "
```

Il est possible de définir les prompts pour un mode particulier simplement en précisant le nom du mode concerné et les valeurs des prompts normaux et de suite.

Exemple (code Java 9) :

```
jshell> /set prompt monmode "\nMon prompt> "  ".....> "
```

Le mode précisé doit exister sinon une erreur est affichée.

Exemple (code Java 9) :

```
jshell> /set prompt monautremode
"\nMon prompt> "
".....> "
|  Does not match any current feedback mode: monautremode
-- /set prompt monautremode "\nMon prompt> "  ".....> "
|  Available feedback modes:
|     concise
|     normal
|     silent
|     verbose
|  See /help /set prompt for help.
```

Pour activer un mode, il faut utiliser la commande /set prompt suivi du mode

Exemple (code Java 9) :

```
jshell> /set prompt monmode
|  /set prompt monmode "\nMon prompt> " ".....> "
jshell>
jshell> /set feedback
monmode
|  Feedback mode: monmode
Mon prompt> if (true) {
.....> i =0;
.....> }
|  Error:
|  cannot find symbol
|    symbol:
variable i
|  i =0;
|  ^
Mon prompt>
```

La configuration n'est pas impactée par la commande /reset.

La configuration demandée n'est utilisée que dans la session courante. Pour quelle soit activée pour toutes les sessions, il faut utiliser l'option -retain.

Exemple (code Java 9) :

```
jshell> /set prompt monmode
|  /set prompt monmode "\nMon prompt> " ".....> "
jshell> /set feedback
monmode
|  Feedback mode: monmode
Mon prompt> if (true) {
.....> int i =0;
.....> }
Mon prompt>
```

Il est possible d'utiliser le motif «%s» dans le prompt : il sera alors remplacé par le prochain identifiant d'un fragment.

Exemple (code Java 9) :

```
jshell> /set prompt monmode
"\nMon prompt %s> "  ".....> "
jshell> /set feedback
monmode
|  Feedback mode: monmode
Mon prompt 2> /list
   1 : if (true) {
       int i = 0;
       }
Mon prompt 2>
```

Cet identifiant ne sera attribué que si le fragment est évalué sans erreur.

Exemple (code Java 9) :

```
Mon prompt 2> test
|  Error:
|  cannot find symbol
|    symbol:
variable test
|  test
|  ^--^
Mon prompt 2> int i = 0;
i ==> 0
Mon prompt 3>
```

98.9.2.2. La troncature des valeurs affichées

Lorsque les valeurs affichées dépassent une certaine taille, celles-ci sont tronquées.

Pour connaître la taille maximale d'un mode, il faut utiliser la commande /set suivi de l'option truncation.

Exemple (code Java 9) :

```
jshell> /set truncation
normal
|  /set truncation normal 80
|  /set truncation normal 1000 expression,varvalue
```

Pour modifier sa taille maximale, il faut utiliser la commande /set suivi de l'option truncation et de la taille souhaitée. Cette taille peut être suivie d'un sélecteur pouvant appartenir à deux types :

Type de sélecteur	
Case	Permet de sélectionner un type de fragments. Les valeurs possibles sont : vardecl : déclaration d'une variable sans initialisation varinit : declaration d'une variable avec initialisation expression : expression varvalue : valeur d'une expression

	assignment : assignation une variable
Action	Permet de sélectionner les actions réalisées durant l'évaluation du fragment. Les valeurs possibles sont : added : le fragment est ajouté modified : le fragment est modifié replaced : le fragment est remplacé par un autre

Plusieurs sélecteurs peuvent être utilisés : il faut séparer chacun d'entre eux avec une virgule.

L'aide en ligne fournit une description complète des sélecteurs : celle-ci est obtenue en invoquant la commande :

/help /set truncation

Exemple (code Java 9) :

```
jshell> /set truncation
monmode
|  /set truncation monmode 80
|  /set truncation monmode 1000 expression,varvalue
jshell> /set truncation
monmode 80 expression,varvalue
jshell> /set feedback
monmode
|  Feedback mode: monmode
Mon prompt> texte
texte ==>
"012345678901234567890123456789012345678901234567 ...
5678901234567890123456789"
```

Attention si plusieurs troncatures sont définies : l'ordre de définition des différentes troncatures est important. Il faut les définir des plus globales ou plus précises.

98.9.2.3. Le format du retour de l'évaluation

Le format de retour de l'évaluation d'un fragment est défini pour chaque mode sauf pour le mode silent. Par exemple dans le mode normal, le type d'une variable n'est pas affiché ou l'utilisation d'un import n'affiche rien.

Exemple (code Java 9) :

```
jshell> import java.time.*;
jshell> 10L
$2 ==> 10
```

Le format de retour de l'évaluation d'un fragment peut être personnalisé.

Pour connaitre le format d'un mode, il faut utiliser la commande /set avec l'option format suivi du nom de mode.

Exemple (code Java 9) :

```
jshell> /set format normal
|  /set format normal action "created" added-primary
|  /set format normal action "modified" modified-primary
|  /set format normal action "replaced" replaced-primary
|  /set format normal action "overwrote" overwrote-primary
|  /set format normal action "dropped" dropped-primary
...
|  /set format normal display "{result}{pre}created scratch variable
```

```
{name} : {type}{post}" expression-added,modified,replaced-primary
|  /set format normal display "{result}{pre}value of {name} :
{type}{post}" varvalue-added,modified,replaced-primary
|  /set format normal display "{result}{pre}assigned to {name} : {type}{post}"
assignment-primary
|  /set format normal display "{result}{pre}{action} variable {name} :
{type}{resolve}{post}" vardecl,varinit
...
```

Il y a de nombreuses options de configuration pour les différents types de messages qui peuvent être affichés.

La commande /set format possède donc de nombreuses options pour configurer le message en retour de l'évaluation d'un fragment.

La syntaxe de la commande est de la forme :

/set format <mode> <field> "<format>" <selector>...

Il est possible d'obtenir le détail de ces options en utilisant la commande /help /set format car les valeurs utilisables sont riches.

98.10. L'utilisation de classes externes

Il est possible d'ajouter des classes via le classpath ou le modulepath pour être utilisées dans la session de JShell.

98.10.1. La configuration du classpath

L'option --class-path permet de définir le classpath qui sera utilisable durant la session de JShell. Le classpath peut classiquement contenir des répertoires ou des fichiers jar.

Résultat :

```
C:\java> jshell --class-path netty-all-4.1.6.Final.jar
|  Welcome to JShell -- Version 11
|  For an introduction type: /help intro

jshell>
```

Il est possible d'utiliser le caractère * comme joker pour indiquer tous les fichiers du chemin. Ils seront alors ajoutés au classpath.

Pour utiliser les classes ajoutées dans le classpath, il suffit de les importer. Il n'est donc pas possible d'utiliser des classes qui soient dans le package par défaut.

Résultat :

```
jshell> import io.netty.util.*
```

Il est aussi possible d'utiliser la commande /env pour définir le classpath durant l'exécution de JShell.

Résultat :

```
jshell> /env

jshell> /env --class-path netty-all-4.1.6.Final.jar
|  Setting new options and restoring state.
```

```
jshell> /env
|     --class-path netty-all-4.1.6.Final.jar

jshell>
```

La commande /env réinitialise l'état de la session, recharge les fragments avec le nouveau classpath.

98.10.2. La configuration du modulepath

Il est possible de définir le modulepath en utilisant l'option --module-path de JShell.

Résultat :

```
C:\java> jshell --module-path ./modules
```

Il est aussi possible d'ajouter des modules au graphe de modules en utilisant l'option --add-modules de JShell

Résultat :

```
C:\java> jshell --module-path ./modules --add-modules utils.jar
```

Il est possible d'utiliser la commande /env pour connaitre la définition de l'environnement courant.

Résultat :

```
jshell> /env
|     --module-path .\modules
```

98.11. Les options de JShell

Il est possible d'obtenir la liste complète des options en utilisant l'option --help ou -h de jshell en ligne de commande

Option	Rôle
--class-path <path>	Préciser les éléments à ajouter dans le classpath
--module-path <path>	Préciser les éléments à ajouter dans le modulepath
--add-modules <module>(,<module>)*	Ajouter un module au graphe de modules
--enable-preview	Activer les fonctionnalités en preview
--startup <file>	Remplacer les scripts de démarrage par celui précisé uniquement pour cette session
--no-startup	Ne pas exécuter de script de démarrage
--feedback <mode>	Préciser le mode de feedback à utiliser (silent, concise, normal, verbose ou un mode personnalisé précédemment défini)
-q	Activer le mode de feedback concis. Equivalent à --feedback concise
-s	Activer le mode de feedback silencieux. Equivalent à --feedback silent
-v	Activer le mode de feedback verbeux. Equivalent à --feedback verbose
-J<flag>	Passer des options à l'environnement d'exécution. Il faut utiliser une option -J pour chaque option
-C<flag>	Passer des options au compilateur. Il faut utiliser une option -C pour chaque option

--version	Afficher la version et rend la main
--show-version	Afficher la version avant de lancer JShell
--help, -?, -h	Afficher l'aide sur les options standard et rend la main
--help-extra, -X	Afficher l'aide sur les options non standard et rend la main

Il est possible de fournir en paramètre le nom d'un fichier ou un des noms de fichier prédéfinis (DEFAULT, PRINTING ou JAVASE). Le fichier indiqué est exécuté comme script de démarrage de JShell.

Résultat :

```
C:\java> type test.jsh
println("bonjour")
C:\java> jshell test.jsh
bonjour
|  Welcome to JShell -- Version 11
|  For an introduction type: /help intro

jshell>
```

Il est aussi possible d'utiliser le caractère « - » pour indiquer d'utiliser l'entrée standard : attention dans ce cas JShell est moins interactif.

Résultat :

```
C:\java> jshell -
System.out.println("hello");
hello
int i = 1;
System.out.println(i);
^C
```

98.12. JShell API

JShell n'est pas qu'un outil : c'est aussi une API qui permet d'utiliser les fonctionnalités de JShell dans une application et pas uniquement comme outil en ligne de commande. Cette API permet d'évaluer dynamiquement des fragments de code et d'obtenir le résultat correspondant.

L'API est encapsulée dans le module jdk.jshell qui contient 4 packages :

- jdk.shell
- jdk.shell.spi
- jdk.shell.execution
- jdk.shell.tool

Il est possible de créer une instance de type jdk.jshell.JShell et de l'utiliser de manière programmatique.

Pour obtenir une telle instance il faut utiliser la fabrique create() de la classe JShell. Comme elle implémente l'interface AutoClosable, il est possible de définir cette instance dans un try with resources qui gérera l'invocation de la méthode close().

Les fragments sont encapsulés dans des instances de type jdk.jshell.Snippet.

La classe jdk.jshell.SnippetEvent encapsule un événement lié à l'évaluation d'un fragment.

La méthode eval() évalue le fragment fourni en paramètre et renvoie une List de SnippetEvent qui sont des événements survenus lors de l'évaluation.

Il est aussi possible d'enregistrer un Listener de type Consumer<SnippetEvent> en utilisant la méthode onSnippetEvent()

pour exploiter les événements émis lors des évaluations de fragments.

Exemple (code Java 9) :

```
import java.util.List;

import jdk.jshell.JShell;
import jdk.jshell.Snippet;
import jdk.jshell.SnippetEvent;

public class TestJShell {

  public static void main(String[] args) {
    List<String> fragments = List.of("5",
      "int i=10;",
      "System.out.println(i);",
      "long ajouter(int a, int b) { return a+b;}",
      "ajouter(i,$1);","ii++;");
    try (JShell shell = JShell.create()) {
      shell.onSnippetEvent(TestJShell::afficherSnippetEvent);

      for (String fragment : fragments) {
        shell.eval(fragment);
      }
    }
  }

  private static void afficherSnippetEvent(SnippetEvent se) {
    Snippet snippet = se.snippet();
    System.out.println("snippet event");
    System.out.println("  status : " + se.status());
    System.out.println("  value : " + se.value());
    System.out.println("  snippet");
    System.out.println("    id : " + snippet.id());
    System.out.println("    source : " + snippet.source());
    System.out.println("    kind : " + snippet.kind());
    System.out.println("    subkind : " + snippet.subKind());
  }
}
```

Résultat :

```
snippet event
  status : VALID
  value : 5
  snippet
    id : 1
    source : 5
    kind : VAR
    subkind : TEMP_VAR_EXPRESSION_SUBKIND
snippet event
  status : VALID
  value : 10
  snippet
    id : 2
    source : int i=10;
    kind : VAR
    subkind : VAR_DECLARATION_WITH_INITIALIZER_SUBKIND
10
snippet event
  status : VALID
  value :
  snippet
    id : 3
    source : System.out.println(i);
    kind : STATEMENT
    subkind : STATEMENT_SUBKIND
snippet event
  status : VALID
  value : null
  snippet
    id : 4
    source : long ajouter(int a, int b) { return a+b;}
    kind : METHOD
    subkind : METHOD_SUBKIND
snippet event
  status : VALID
  value : 15
  snippet
    id : 5
    source : ajouter(i,$1);
    kind : VAR
    subkind : TEMP_VAR_EXPRESSION_SUBKIND
snippet event
  status : REJECTED
  value : null
  snippet
    id : 6
    source : ii++;
    kind : ERRONEOUS
    subkind : UNKNOWN_SUBKIND
```

99. Les outils libres et commerciaux

Chapitre 99

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Pour développer des composants en Java (applications clientes, applets, applications web, services web, ...), il existe une large gamme d'outils commerciaux et libres pour répondre à ce vaste marché.

Comme dans d'autres domaines, les avantages et les inconvénients de ces outils sont semblables et fonction de leur catégorie bien qu'ils ne puissent pas être complètement généralisés :

	Avantages	Inconvénient
Outils commerciaux	plus de fonctionnalité une meilleure ergonomie une hot line dédiée	le prix
Outils libres	la gratuité des mises à jour fréquentes (variable selon le projet)	pas de support officiel (aide communautaire par les forums)

Quelques outils libres n'ont que peu de choses à envier à certains de leurs homologues commerciaux : ainsi, par exemple, Tomcat du projet Jakarta est l'implémentation de référence pour ce qui concerne les servlets et les JSP.

Enfin certains éditeurs, surtout dans le domaine des IDE, proposent souvent une version limitée (dans les fonctionnalités ou dans le temps) mais gratuite qui permet d'utiliser et d'évaluer le produit.

L'évolution de ces outils suit l'évolution du marché concernant Java : développement d'applets (web client), d'applications autonomes et C/S et maintenant développement côté serveur (applications et services web).

La liste des produits de ce chapitre est loin d'être exhaustive mais représente les plus connus ou ceux que j'utilise.

Ce chapitre contient plusieurs sections :

- Les environnements de développement intégrés (IDE)
- Les serveurs d'application
- Les conteneurs web
- Les conteneurs d'EJB
- Les outils divers
- Les MOM
- Les outils concernant les bases de données
- Les outils de modélisation UML

99.1. Les environnements de développement intégrés (IDE)

Les environnements de développements intégrés regroupent dans un même outil la possibilité d'écrire du code source, de concevoir une application de façon visuelle par assemblage de beans, d'exécuter et de déboguer le code.

D'une façon générale, ils sont tous très gourmands en ressources machines : un processeur rapide, 256 Mo de RAM pour être à l'aise ... En fait la plupart de ces outils sont partiellement ou totalement écrits en Java.

Le choix d'un IDE doit tenir compte de plusieurs caractéristiques : ergonomie et convivialité pour faciliter l'utilisation, fonctionnalités de bases et avancées pour accroître la productivité, robustesse, support des standards, ... La plupart des éditeurs proposent une version gratuite qui permet d'évaluer leur produit.

99.1.1. Eclipse

Eclipse est un projet open source à l'origine développé par IBM pour ses futurs outils de développement et offert à la communauté. Le but est de fournir un outil modulaire capable non seulement de faire du développement en Java mais aussi dans d'autres langages et d'autres activités. Cette polyvalence est liée au développement de modules (plug-in) réalisés par la communauté ou des entités commerciales.

Licence : EPL (eclipse Public Licence)
Statut :
Site web : http://www.eclipse.org

Version	Date de diffusion	
3.0	juin 2004	
3.1	juin 2005	10 projets
3.2	juin 2006	nom de code Callisto, 10 projets
3.3	juin 2007	nom de code Europa, 21 projets
3.4	juin 2008	nom de code Ganymede, 23 projets
3.5	juin 2009	nom de code Galileo, 33 projets
3.6	juin 2010	nom de code Helios, 77 projets
3.7	juin 2011	nom de code Indigo, 62 projets
4.1	juin 2011	nom de code Juno, 71 projets
4.2	juin 2012	nom de code Juno, 71 projets
4.3	juin 2013	nom de code Kepler, 72 projets
4.4	juin 2014	nom de code Luna, 76 projets, support de Java 8
4.5	juin 2015	nom de code Mars, 79 projets
4.6	juin 2016	nom de code Neon, 84 projets
4.7	juin 2017	nom de code Oxgyen, 83 projets
4.8	juin 2018	nom de code Photon, 85 projets
4.9	septembre 2018	nom de code 2018-09
4.10	décembre 2018	nom de code 2018-12
4.11	mars 2019	nom de code 2019-03
4.12	juin 2019	nom de code 2019-06

4.13	septembre 2019	nom de code 2019-09
4.14	décembre 2019	nom de code 2019-12
4.15	mars 2020	nom de code 2020-03
4.16	juin 2020	nom de code 2020-06
4.17	septembre 2020	nom de code 2020-09
4.18	décembre 2020	nom de code 2020-12
4.19	mars 2021	nom de code 2021-03
4.20	juin 2021	nom de code 2021-06
4.21	septembre 2021	nom de code 2021-09
4.22	décembre 2021	nom de code 2021-12
4.23	mars 2022	nom de code 2022-03
4.24	juin 2022	nom de code 2022-06
4.25	septembre 2022	nom de code 2022-09
4.26	décembre 2022	nom de code 2022-12
4.27	mars 2023	nom de code 2023-03
4.28	juin 2023	nom de code 2023-06
4.29	septembre 2023	nom de code 2023-09
4.30	décembre 2023	nom de code 2023-12

La fondation Eclipse gère de nombreux sous-projets parmi lesquels :

Projet	Description
Eclipse	ce projet développe l'architecture et la structure de la plate-forme Eclipse.
Eclipse Tools	ce projet développe ou intègre des outils à la plate-forme pour permettre à des tiers de l'enrichir. Il possède plusieurs sous-projets tels que CDT (plug-in pour le développement en C/C++), AspectJ (AOP), GEF (Graphical Editing Framework), PHP (plug-in pour le développement en PHP), Cobol (plug-in pour le développement en Cobol), VE (Visual Editor) pour la création d'IHM.
Eclipse Technology	ce projet, divisé en trois catégories, propose d'effectuer des recherches sur des évolutions de la plate-forme et des technologies qu'elle met en oeuvre.
Web Tools Platform (WTP)	ce projet a pour but d'enrichir la plate-forme enfin de proposer un framework et des services pour la création d'outils de développement d'applications web. Il est composé de plusieurs sous-projets : WST (Web Standard Tools), JST (J2EE Standard Tools), ATF (Ajax Toolkit Framework), Dali (mapping avec JPA) et JSF (Java Server Faces)
Test and Performance Tools Platform (TPTP)	ce projet a pour but de développer une plate-forme servant de support à la création d'outils de tests et d'analyses
Business Intelligence and Reporting Tools (BIRT)	ce projet a pour but de développer une plate-forme facilitant l'intégration de générateur d'états. Il est composé de 4 sous-projets : ERD (Eclipse Report Designer), WRD (Web based Report Designer), ERE (Eclipse Report Engine) et ECE (Eclipse Charting Engine).
Eclipse Modeling	ce projet contient plusieurs sous-projets dont EMF (Eclipse Modeling Framework) et UML2 pour une implémentation d'UML reposant sur EMF
Data Tools Platform (DTP)	ce projet a pour but de manipuler des sources de données (bases de données relationnelles)

Device Software Development Platform	ce projet a pour but de créer des plugins pour faciliter le développement d'applications sur appareils mobiles
Eclipse SOA Tools Platform	ce projet a pour but de développer des outils pour faciliter la mise en oeuvre d'architecture de type SOA

Le site officiel est à l'url www.eclipse.org/

Bien que développé en Java, les performances à l'exécution d'Eclipse sont très bonnes car il n'utilise pas Swing pour l'interface homme-machine mais un toolkit particulier nommé SWT associé à la bibliothèque JFace. SWT (Standard Widget Toolkit) est développé en Java par IBM en utilisant au maximum les composants natifs fournis par le système d'exploitation sous-jacent. JFace utilise SWT et propose une API pour faciliter le développement d'interfaces graphiques.

Eclipse ne peut donc fonctionner que sur les plates-formes pour lesquelles SWT a été porté.

SWT et JFace sont utilisés par Eclipse pour développer le plan de travail (Workbench) qui organise la structure de la plate-forme et les interactions entre les outils et l'utilisateur. Cette structure repose sur trois concepts : la perspective, la vue et l'éditeur. La perspective regroupe des vues et des éditeurs pour offrir une vision particulière des développements. En standard, Eclipse propose huit perspectives.

Les vues permettent de visualiser et de sélectionner des éléments. Les éditeurs permettent de visualiser et de modifier le contenu d'un élément de l'espace de travail.

99.1.2. Netbeans

Netbeans est un environnement de développement open source écrit en Java. Le produit est composé d'une partie centrale à laquelle il est possible d'ajouter des modules.

Netbeans est un IDE open source racheté et développé par Sun Microsystems.

Licence : open source
Statut : mises à jour régulières
Site web : netbeans.org/

Version	Date de diffusion	
3.5	juin 2003	
3.6	avril 2004	
4.0	decembre 2004	
4.1	mai 2005	
5.0	janvier 2006	
5.5	octobre 2006	
6.0	décembre 2007	
6.1	avril 2008	
6.5	novembre 2008	
6.7	juin 2009	
6.8	Décembre 2009	
6.9	Juin 2010	
7.0	Avril 2011	

7.0.1	Août 2011	
7.3	Février 2013	
7.4	Octobre 2014	
8.0	Mars 2014	support Java 8, Java ME Embedded 8
8.0.2	Novembre 2014	
9	Juillet 2018	Projet Apache (Incubating), support de Java 10
10	Décembre 2018	Projet Apache (Incubating), support de Java 11, JUnit 5
11	Avril 2019	Projet Apache (Incubating), support de Java 12
11.1	Juillet 2019	Projet Apache Top Level
11.2	Octobre 2019	
11.3	Février 2020	
12.0 LTS	Juin 2020	
12.1	Septembre 2020	
12.2	Décembre 2020	
12.3	Mars 2021	
12.4	Mai 2021	
12.5	Septembre 2021	
12.6	Novembre 2021	
13	Mars 2022	
14	Juin 2022	
15	Août 2022	
16	Novembre 2022	
17	Février 2023	
18	Mai 2023	
19	Septembre 2023	

Netbeans propose des fonctionnalités permettant le développement d'applications standalone (AWT/Swing), web (Servlets, JSP, Struts, JSF), mobile (J2ME) ou d'entreprise (J2EE/Java EE).

Il fonctionne sous Windows, Linux, Mac OS X et Solaris.

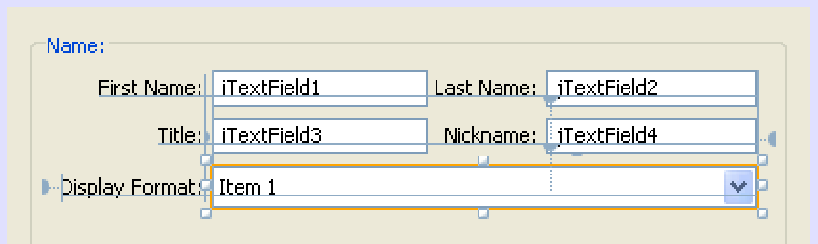
Netbeans est modulaire et propose plusieurs plugins officiels :

Mobility Pack	Pour le développement d'applications mobiles avec une conception WYSIWYG. Il existe une version pour le développement avec le profile CLDC/MIDP et une avec le profile CDC
Visual Web Pack	Pour le développement d'applications web avec une conception WYSIWYG
Enterprise Pack	Pour le développement de service web et de composants pour une architecture de type SOA avec une conception WYSIWYG
Profiler	Pour profiler une application
C/C++ Pack	Pour le développement d'applications en C/C++

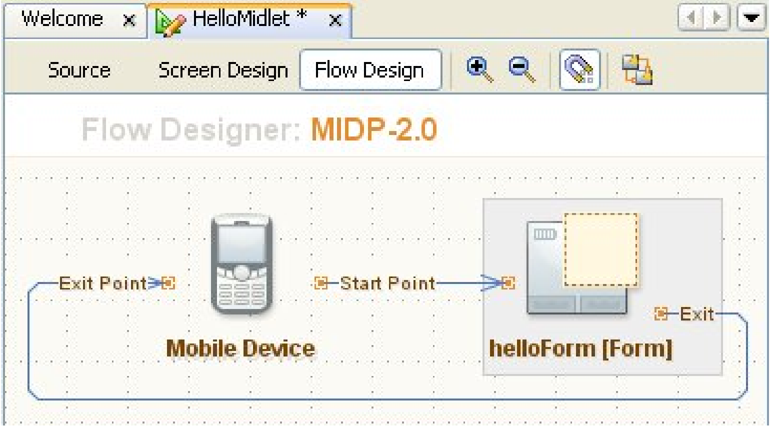
Il existe aussi de nombreux plugins développés par des tiers (une liste peut être consultée à l'url netbeans.org/catalogue/index.html)

Quelques fonctionnalités de Netbeans sont particulièrement intéressantes :

- Le développement wysiwig d'applications reposant sur Swing (projet Matisse) qui propose un positionnement aisé des composants



- Le plug-in Mobility pack qui facilite le développement des midlets en gérant leurs enchaînements graphiquement



- Le support des dernières technologies Java
- Le développement wysiwyg d'applications web avec les JSF (plug-in Visual web Pack)

99.1.3. IntelliJ IDEA

IntelliJ IDEA est un IDE développé par JetBrains.

Licence : commerciale, une version gratuite (Community Edition) existe
Statut : mises à jour régulières
Site web : www.jetbrains.com/idea/

Version	Date de diffusion	
9.0	décembre 2009	
10.0	décembre 2010	
11	décembre 2011	
12	décembre 2012	
13	décembre 2013	
14	novembre 2014	

15		
2016		
2017		
2018		
2019		
2020.1	avril 2020	

IntelliJ IDEA est proposé en deux éditions :

- Community Edition : cette édition open source sous licence Apache 2.0 propose un support pour Java SE et Groovy : assistance au codage, refactoring, debugging, support des tests (JUnit et TestNG), support de Maven et Ant
- Ultimate Edition : cette édition commerciale propose toutes les fonctionnalités de l'IDE notamment un support pour le développement d'applications d'entreprises avec Java EE

IntelliJ est disponible sous Windows, Linux et Mac OS X.

99.1.4. Oracle JDeveloper

JDeveloper est un IDE riche en fonctionnalités qui couvre de nombreux aspects du développement : modélisation UML, écriture du code, débogage, tests, profiling et déploiement d'applications.

Licence : disponible gratuitement après un enregistrement chez OTN
Statut :
Site web : https://www.oracle.com/application-development/technologies/jdeveloper.html

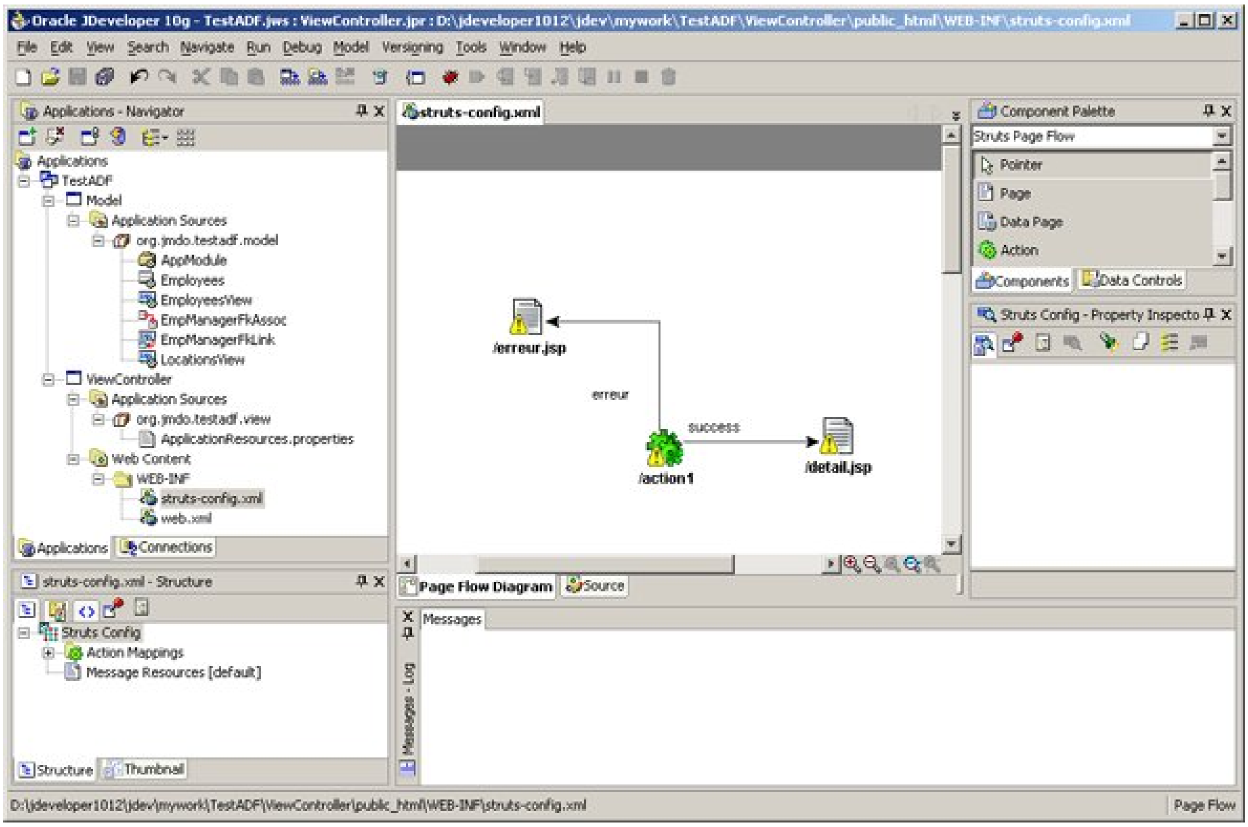
Version	Date de diffusion	
10.1.2		
10.1.3		
11g	avril 2010	
12c	juillet 2013	version 12.2.1.3.0 publiée en août 2017

Ecrit en Java, JDeveloper est disponible sur plusieurs plates-formes : Windows, Mac, Linux et plusieurs Unix.

JDeveloper propose des extensions pour enrichir l'outil en fonctionnalités notamment celles proposées par des tiers.

JDeveloper propose bien sûr une intégration facilitée de plusieurs produits d'Oracle notamment la base de données et le serveur d'applications et surtout une forte intégration et une mise en oeuvre d'Oracle ADF.

JDeveloper version 10.1.2



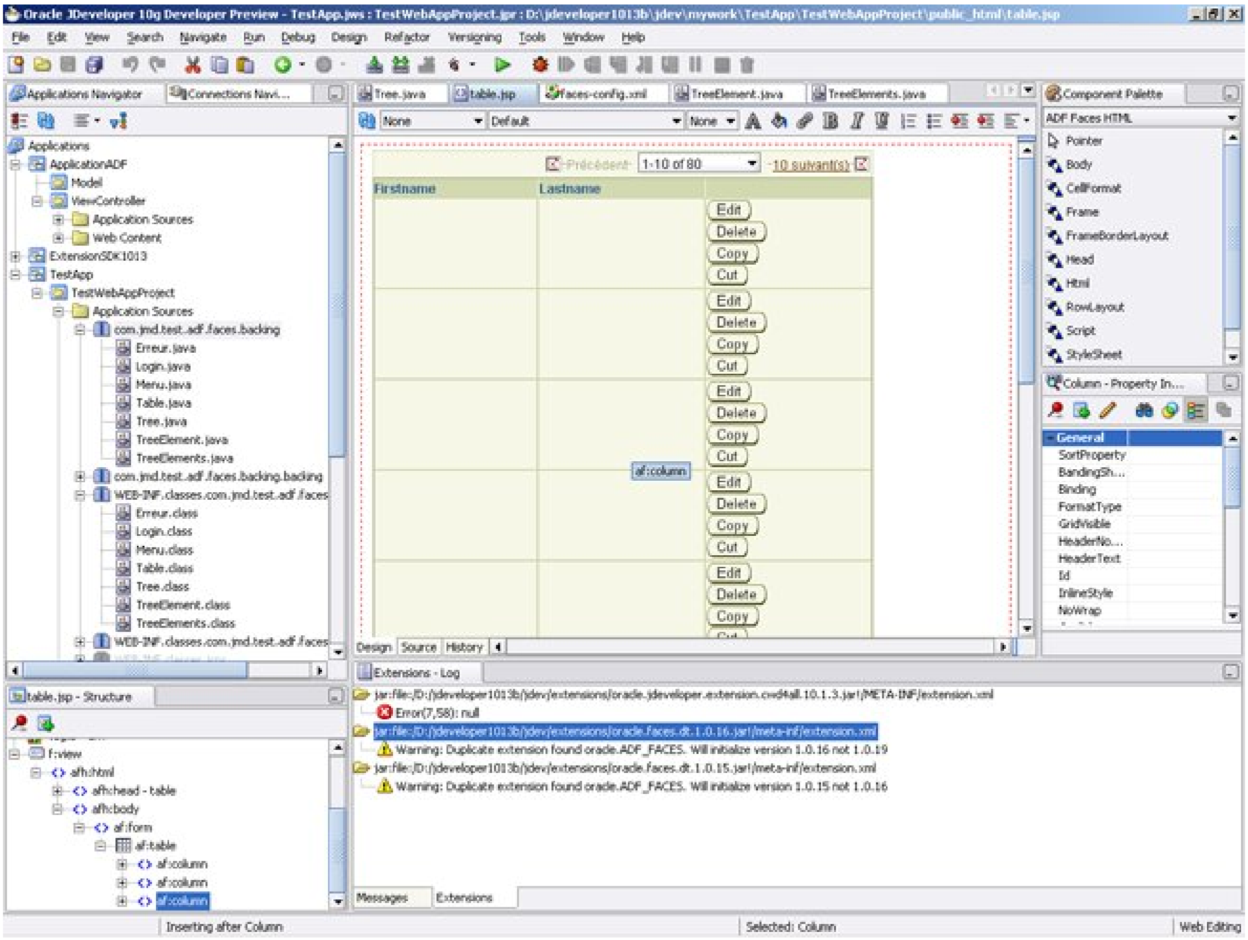
Cette version propose de nombreuses fonctionnalités dont voici quelques-unes des plus intéressantes :

- Support de J2EE 1.4
- Un serveur d'applications intégré à l'outil (OC4J)
- Support de nombreux outils open source (Ant, Junit, Struts, ...)
- Editeur de diagrammes pour les flux Struts
- Support de Toplink pour le mapping O/R
- Support d'ADF avec de nombreux assistants pour faciliter sa mise en oeuvre

JDeveloper version 10.1.3

La version 10.1.3.2 apporte de nombreuses fonctionnalités et améliorations par rapport à la version précédente, en voici quelques-unes des plus importantes :

- Une nouvelle interface plus moderne
- De nombreuses améliorations sont ajoutées dans les fonctionnalités de bases pour rattraper le retard de l'outil en la matière (assistant de code plus poussé, refactoring enrichi, historique local, ...)
- Support de nombreuses technologies (EJB 3.0, portlets, ADF, services web, XML, ...)
- Ajout de nouveaux éditeurs et designer : BEPL, ESB, XSLT
- Téléchargement des mises à jour à partir de l'outil



```
jar:file:/D:/jdeveloper1013b/jdev/extensions/oracle.jdeveloper.extension.cvs4all.10.1.3.jar!/META-INF/extension.xml
    Error(7,58): null
jar:file:/D:/jdeveloper1013b/jdev/extensions/oracle.faces.dt.1.0.16.jar!/meta-inf/extension.xml
    Warning: Duplicate extension found oracle.ADF_FACES. Will initialize version 1.0.16 not 1.0.19
jar:file:/D:/jdeveloper1013b/jdev/extensions/oracle.faces.dt.1.0.15.jar!/meta-inf/extension.xml
    Warning: Duplicate extension found oracle.ADF_FACES. Will initialize version 1.0.15 not 1.0.16
```

JDeveloper 10.1.3 est proposé en trois versions :

- Studio : propose toutes les fonctionnalités dont ADF
- J2EE : propose de nombreuses fonctionnalités sauf ADF
- Java : permet le développement avec Java et XML

JDeveloper est particulièrement intéressant pour mettre en oeuvre le framework Oracle ADF.

99.1.5. IBM Rational Application Developer for WebSphere Software

Rational Application Developer for WebSphere Software repose sur la version 3.2 d'Eclipse et propose le développement d'applications web, de portails, d'applications d'entreprises ou standalones ou de services web pour mettre en oeuvre une SOA. Cet outil s'intègre parfaitement avec les outils IBM et Rational.

Licence : commerciale
Statut :
Site web : https://www.ibm.com/products/rad-for-websphere-software

Version	Date de diffusion	
6.0	Janvier 2005	
7.5		repose sur Eclipse 3.4
8.0	Novembre 2011	repose sur Eclipse 3.6
8.5		repose sur Eclipse 3.6 avec support de Java 7
9.0	2013	repose sur Eclipse 4.2
9.7	Novembre 2018	repose sur Eclipse 4.7, support de Java EE 8

99.1.6. IBM Rational Team Concert

Rational Team Concert repose Eclipse et sur Jazz d'IBM pour la gestion collaborative et la gestion du cycle de vie des développements.

Licence : commerciale
Statut :
Site web :

Version	Date de diffusion	
1.0	Juin 2008	
2.0	Juin 2009	
3.0	Novembre 2010	
4.0	Juin 2012	
5.0	Juin 2014	
5.0.1	Septembre 2014	
6.0		

99.1.7. MyEclipse

MyEclipse est un IDE basé sur Eclipse développé par Genuitec.

Licence : commerciale
Statut :
Site web : https://www.genuitec.com/products/myeclipse/

Version	Date de diffusion	
5.1		
7.0		
7.1	avril 2009	
7.5	juin 2009	
8.0	novembre 2009	
8.6	novembre 2010	
9.1	juillet 2011	
10.0	novembre 2011	
10.6	juillet 2012	
10.7		
2013		
2014		
2015		
2016		Support de Java 8, repose sur Eclipse Mars
2017		Support Angular 2 et 4 (CI 4), repose sur Eclipse Neon
2018	août 2018	Support de Java 10, Java EE 8, repose sur Eclipse Photon
2019		Support de Java 11, repose sur Eclipse 2018-12
2020.5		Support de Java 14, repose sur Eclipse 2020-03

2020.9		Support de Quarkus, Docker, repose sur Eclipse 2020-06
2021.5		Support de Java 15 et 16, Tomcat 9, Jakarta EE 9, repose sur Eclipse 2021-03
2022.1		Support de Java 17, Jakarta EE 9.1 et Vue, repose sur Eclipse 2021-12
2023.1		Support de Java 20, Jakarta EE 10, Spring 6, Spring Boot 3, Hibernate 6

MyEclipse regroupe de nombreux plugins dont certains sont inédits comme par exemple le portage de Matisse de NetBeans sur Eclipse.

99.1.8. IBM Websphere Studio Application Developper

Websphere Studio Application Developper (WSAD) représente le nouvel outil de développement d'applications Java/web d'IBM. Il représente une fusion de nombreuses fonctionnalités des outils Visual Age for Java et Websphere Studio. Le coeur de l'outil est composé par Websphere Studio Workbench dont une partie du code a été fournie à la communauté open source pour devenir le projet Eclipse.

Licence : commerciale
Statut : remplacé par le produit Rational Application Developer for WebSphere Software

Version	Date de diffusion	
4.0		orienté développement Java/web : il ne permet pas de développement d'applications graphiques en mode RAD.

99.1.9. Embarcadero (Borland/CodeGear) JBuilder

Borland est spécialisé depuis des années dans la création d'outils de développement possédant une excellente réputation. Ainsi Jbuilder est un IDE ergonomique qui génère un code "propre". Depuis sa version 3.5, JBuilder est écrit en Java ce qui lui permet de s'exécuter sans difficulté sur plusieurs plates-formes notamment Windows, Linux ou Solaris.

Licence : commercial, une version (foundation) téléchargeable gratuitement
Statut :
Site web : https://www.embarcadero.com/products/jbuilder

Version	Date de diffusion	
2006		existe en plusieurs éditions : • foundation : téléchargeable gratuitement • developer • entreprise
2007		basée sur Eclipse, maintenue et diffusée par sa filiale CodeGear ; existe en plusieurs éditions : • JBuilder Foundation 2007 (téléchargeable gratuitement après enregistrement) • JBuilder Developper 2007 • JBuilder Professional 2007 • Jbuilder Enterprise 2007
2008	avril 2008	
2008 R2	avril 2009	

Le produit dispose de nombreuses caractéristiques qui facilitent le travail du développeur : la technologie CodeInsight facilite grandement l'écriture du code dans l'éditeur, de nombreux assistants facilitent la génération de code ...

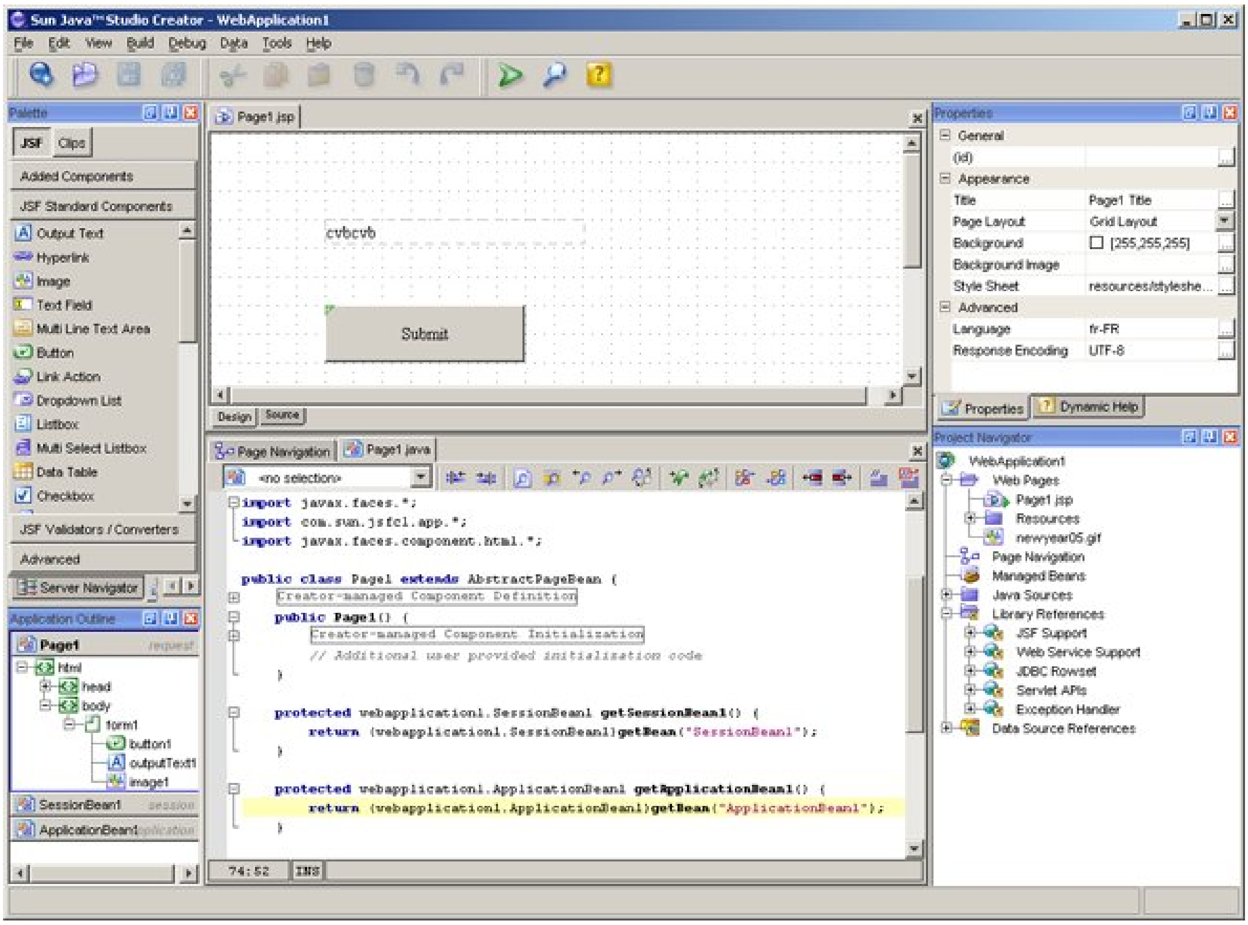
99.1.10. Sun Java Studio Creator

L'environnement de développement intégré Java Studio Creator de Sun permet de générer des applications Web à l'aide de la technologie Java notamment avec les Java ServerFaces et les portlets.

Licence : gratuit
Statut : remplacé par NetBeans
Site web :

Sun Java Studio Creator version 1.0

Cette première version de l'outil est payante. C'est un des premiers outils à exploiter les possibilités pour faciliter la mise en oeuvre des JSF.



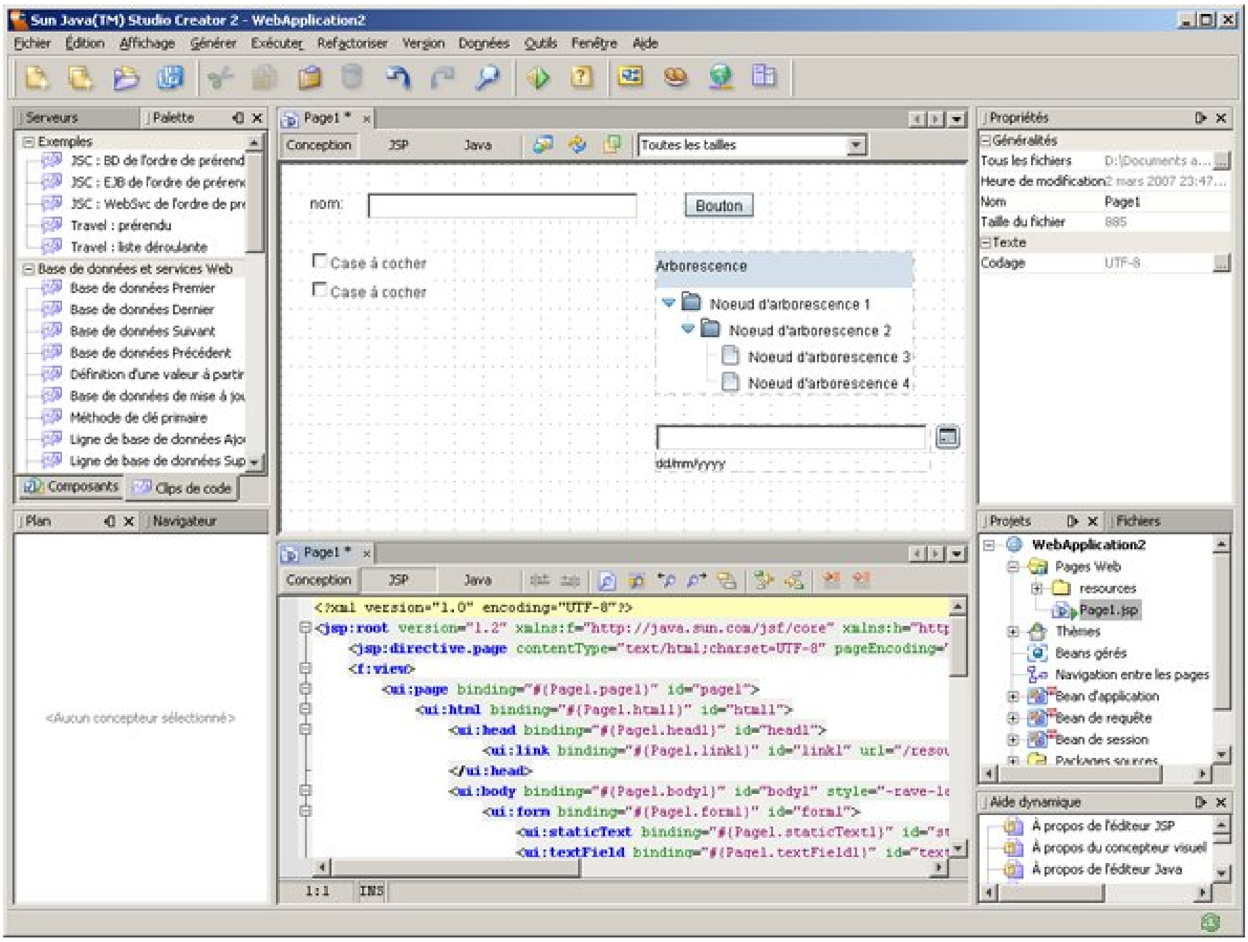
General	
(id)	
Appearance	
Title	Page1 Title
Page Layout	Grid Layout
Background	□ [255,255,255]
Background Image	
Style Sheet	resources/styleshe...
Advanced	
Language	fr-FR
Response Encoding	UTF-8

```
import javax.faces.*;
import com.sun.jsfcl.app.*;
import javax.faces.component.html.*;

public class Page1 extends AbstractPageBean {
    Creator-managed Component Definition
    public Page1() {
        Creator-managed Component Initialization
        // Additional user provided initialization code
    }

    protected webapplication1.SessionBean1 getSessionBean1() {
        return (webapplication1.SessionBean1)getBean("SessionBean1");
    }

    protected webapplication1.ApplicationBean1 getApplicationBean1() {
        return (webapplication1.ApplicationBean1)getBean("ApplicationBean1");
    }
```

Sun Java Studio Creator version 2.0

Cette seconde version est téléchargeable gratuitement après une inscription au SDN (Sun Developer Network).

Java Studio Creator 2 utilise Netbeans 4.1 comme base. Le package d'installation contient un JDK, le serveur d'applications Sun Java System Application Server 8.x et une base de données


Sun Java(TM) Studio Creator 2 - WebApplication2
Fichier Édition Affichage Générer Exécuter Refactoriser Version Données Outils Fenêtre Aide
Serveurs
Palette
Exemples
JSC : BD de l'ordre de prérend
JSC : EJB de l'ordre de prérend
JSC : WebSvc de l'ordre de pré
Travel : prérendu
Travel : liste déroulante
Base de données et services Web
Base de données Premier
Base de données Dernier
Base de données Suivant
Base de données Précédent
Définition d'une valeur à partir
Base de données de mise à jou
Méthode de clé primaire
Ligne de base de données Ajo
Ligne de base de données Sup
Composants
Clips de code
Plan
Navigateur
<Aucun concepteur sélectionné>
Page1 *
Conception
JSP
Java
Toutes les tailles
nom:
Bouton
Case à cocher
Case à cocher
Arborescence
Noeud d'arborescence 1
Noeud d'arborescence 2
Noeud d'arborescence 3
Noeud d'arborescence 4
dd/mm/yyyy
Propriétés
Projets
Fichiers
WebApplication2
Pages Web
resources
Page1.jsp
Thèmes
Beans gérés
Navigation entre les pages
Bean d'application
Bean de requête
Bean de session
Packages sources
Aide dynamique
À propos de l'éditeur JSP
À propos du concepteur visuel
À propos de l'éditeur Java
1:1
INS

Généralités
Tous les fichiers D:\Documents a...
Heure de modification 2 mars 2007 23:47...
Nom Page1
Taille du fichier 885
Texte
Codage UTF-8

```
<?xml version="1.0" encoding="UTF-8"?>
<jsp:root version="1.2" xmlns:f="http://java.sun.com/jsf/core" xmlns:h="http
    <jsp:directive.page contentType="text/html;charset=UTF-8" pageEncoding="
    <f:view>
        <ui:page binding="#{Page1.page1}" id="page1">
            <ui:html binding="#{Page1.html1}" id="html1">
                <ui:head binding="#{Page1.head1}" id="head1">
                    <ui:link binding="#{Page1.link1}" id="link1" url="/resou
                </ui:head>
                <ui:body binding="#{Page1.body1}" id="body1" style="-rave-la
                    <ui:form binding="#{Page1.form1}" id="form1">
                        <ui:staticText binding="#{Page1.staticText1}" id="st
                        <ui:textField binding="#{Page1.textField1}" id="text
```

99.1.11. JCreator

JCreator est un IDE développé par Xinox particulièrement rapide car il est écrit en code natif.

Licence : commerciale, une version gratuite (LE) est téléchargeable
Statut :
Site web :

Version	Date de diffusion	
3.5	décembre 2004	
4.0	mai 2006	Il existe deux éditions : • La version LE : téléchargeable gratuitement • La version Pro : payante
4.5	aout 2007	
5.0	mars 2010	

99.1.12. BEA Workshop

BEA Workshop est une famille d'IDE développée par BEA qui utilise Eclipse comme base.

Licence :
Statut :
Site web :

Il existe plusieurs éditions :

- BEA Workshop for Weblogic
- BEA Workshop for JSP (cette édition est téléchargeable gratuitement après inscription)
- BEA Workshop Studio
- BEA Workshop for Struts
- BEA Workshop for JSF

99.1.13. IBM Visual Age for Java

IBM proposait une famille d'outils pour le développement avec différents langages dont une version dédiée à Java.

VAJ n'est plus supporté par IBM : il est remplacé par la famille d'outils Websphere Studio Application Developper.

Visual Age for Java (VAJ) était un outil novateur dans son ergonomie et son utilisation qui étaient complètement différentes des autres EDI. Les débuts de son utilisation étaient parfois déroutants mais la persévérance permettait de révéler toute sa puissance.

La fenêtre principale (plan de travail) est séparée en deux parties :

- l'espace de travail : il contient et organise les différents éléments (projets, packages, classes, méthodes ...)
- le code source : si l'élément sélectionné dans l'espace de travail contient du code, il est visualisé et modifiable dans cette partie

Par défaut le code est éditable par méthode mais depuis la version 3.5, il est toutefois possible de visualiser le code source complet, cependant, les opérations réalisables dans ce mode sont moins nombreuses.

VAJ possédait plusieurs points forts : le regroupement de toutes les classes et leur organisation dans l'espace de travail, la compilation incrémentale à l'écriture et au débogage, le travail collaboratif avec le contrôle de version dans un référentiel (repository). Tous ces points facilitaient le développement de gros projets.

VAJ était un outil puissant particulièrement adapté aux utilisateurs chevronnés pour de gros projets.

99.1.14. Webgain Visual Café

Webgain Studio proposait un ensemble d'outils (Visual Café, Dreamweaver Ultradev, Top link, Structure Builder, Weblogic) pour la création d'applications e-business. Visual Café a été un des premier IDE de développement en Java. Visual Café existait en trois versions : standard, expert et entreprise suite.

Malheureusement cet outil n'est plus disponible.

99.2. Les serveurs d'application

Les serveurs d'applications sont des outils qui permettent l'exécution de composants Java côté serveur (servlets, JSP, EJB, ...) selon les spécifications de la plate-forme J2EE/Java EE.

99.2.1. JBoss Application Server

JBoss est un projet open source développé en Java pour fournir un serveur d'applications certifié Java EE.

Licence : open source, commerciale avec support
Statut : actif
Site web :

Version	Date de diffusion	
4.0		certifié J2EE 1.4
4.2		support EJB 3.0
5.0 GA	décembre 2008	
5.1 GA	mai 2009	certifié Java EE 5
6.0	décembre 2010	implémente le profile Web de Java EE 6
6.4	avril 2015	
7.0	mai 2016	
7.1.0	décembre 2017	
7.2	janvier 2019	
7.3	mars 2020	

JBoss est composé d'un ensemble d'outils : JBoss Server, JBoss MQ (implémentation de JMS), JBoss MX, JBoss TX (implémentation de JTA/JTS), JBoss SX , JBoss CX et JBoss CMP.

99.2.2. JBoss Wildfly

JBoss Widfly est un projet open source développé en Java pour fournir un serveur d'applications certifié Java EE.

Licence : open source
Statut : actif
Site web : www.wildfly.org/

Version	Date de diffusion	
8.0	Février 2014	certifié pour le full profile Java EE 7
8.2.0	Novembre 2014	
9	Juillet 2015	
10	Janvier 2016	
11	Octobre 2017	
12	Février 2018	
13	Mai 2018	
14	Août 2018	Certifiée Java EE 8
15	Décembre 2018	Support de Java 11
16	Février 2019	
17	Juin 2019	la version 17.0.1 est certifiée Java EE 8 et Jakarta EE 8
17.1		Certifiée Java EE 8 et Jakarta EE 8

18	Octobre 2019	
19	Mars 2020	
19.1	Mai 2020	
20.0	juin 2020	
21.0	octobre 2020	
22.0	janvier 2021	
23.0	mars 2021	
24.0	juin 2021	
25.0	octobre 2021	
26.0	décembre 2021	
26.1	avril 2022	
27	novembre 2022	Support Jakarta EE 10
28	avril 2023	
29	juillet 2023	
30	octobre 2023	

99.2.3. JOnAs

JOnAS (Java Open Application Server) est un projet open source développé par le consortium ObjectWb / OW2 (Bull, INRIA, France Telecom) dont le but est de proposer un serveur d'applications Java EE. Il se compose de nombreux éléments open source tels que JOTM pour le support des transactions, JORAM pour l'implémentation de JMS, Tomcat ou Jetty comme conteneurs Web, Speedo pour l'implémentation JDO, ...

Licence : open source
Statut : projet abandonné en 2015
Site web : https://jonas.ow2.org/

Version	Date de diffusion	
2.3.1	février 2005	certifié J2EE 1.4.
5.1	mars 2009	certifié Java EE 5
5.1.4	novembre 2010	
5.2.2	juillet 2011	
5.2.3	mars 2012	
5.3	octobre 2013	certifié Java EE 6 web profile

99.2.4. GlassFish

Le projet GlassFish est un projet communautaire dont le but est de développer un serveur d'applications open source qui implémente les spécifications de Java EE à partir de la version 5. Il est l'implémentation de référence de ces spécifications.

Licence : open source, commercial avec support
Statut : actif
Site web : https://javaee.github.io/glassfish/ jusqu'à GlassFish 5.1, https://glassfish.org/ pour les versions ultérieures

Version	Date de diffusion	
	mai 2006	support Java EE 5
2.0	septembre 2007	
2.1	janvier 2009	
3.0	décembre 2009	implémentation de référence de Java EE 6
3.0.1	juin 2010	
3.1	février 2011	cluster / haute disponibilité
3.1.1	juillet 2011	
3.1.2	février 2012	
4.0	juin 2013	support Java EE 7
4.1	septembre 2014	
5.0	septembre 2017	implémentation de référence de Java EE 8
5.1	janvier 2019	première release par la fondation Eclipse sous licence EPL
6.0	décembre 2020	Implémentation de référence de Jakarta EE 9 (packages javax.* en jakarta.*)
6.1	mai 2021	Implémentation de référence de Jakarta EE 9.1
7.0	décembre 2022	Implémentation de référence de Jakarta EE 10

Sun propose le serveur d'applications Java System Application Server Platform Edition 9 qui implémente les spécifications de Java EE 5. Ce serveur gratuit est basé sur le projet GlassFish. Il peut être téléchargé dans une archive avec le SDK.

A partir de la version 5, le développement de Glassfish est effectué par la fondation Eclipse.

99.2.5. Apache TomEE

Le projet Epache TomEE est une combinaison de plusieurs projets de la fondation Apache pour fournir un serveur d'applications Java EE articulé autour de Tomcat.

Licence : open source
Statut : actif
Site web : https://tomee.apache.org/

Version	Date de diffusion	
1.7.1	septembre 2014	Certifié Java EE 6 Web Profile
7.0.0	mai 2016	Support de Java EE 7, repose sur Tomcat 8.5
7.1.0	septembre 2018	Support de Microprofile 1.4
8.0.0	septembre 2019	Support de Java EE 8 / Jakarta EE 8, Microprofile 2.0, repose sur Tomcat 9, requiert Java 8
8.0.1	janvier 2020	

9.0.0	janvier 2023	Repose sur Tomcat 10, requiert Java 11
9.1.0	juin 2023	Support de Jakarta EE 9.1 Web Profile et MicroProfile 5

Apache TomEE utilise des composants de la foncdation Apache :

Composant	Description
Apache Tomcat	Conteneur Web : HTTP, Servlet, JSP
Apache OpenEJB	Conteneur pour Enterprise JavaBeans
Apache OpenWebBeans	Implémentation de CDI
Apache OpenJPA	Implémentation de JPA
Apache MyFaces	Implémentation de JSF
Apache ActiveMQ	Implémentation de JMS
Apache CXF	Implémentation des Web Services SOAP (JAX-WS) et RESTful (JAX-RS)
Apache Derby	Base de données relationnelles
Apache Geronimo transaction	Implémentation de JTA
Apache BVal	Implémentation de Bean validation

TomEE est diffusé en plusieurs éditions selon les besoins :

Edition	Description
Web Profile : certifié Java EE 6 Web Profile	Servlets, JSP, JSF, JTA, JPA, CDI, Bean Validation et EJB Lite
JAX-RS : certifié Java EE 6 Web Profile	Edition Web Profile, JAX-RS
Plus	Edition JAX-RS, EJB Full, JMS, JCA, JAX-WS
Plume	Edition Plus, Mojarra, EclipseLink

99.2.6. IBM Websphere Application Server

Websphere Application Server (WAS) est le serveur d'applications de la famille d'outils Websphere. Il permet le déploiement de composants Java orientés entreprise.

Licence : commerciale
Statut : actif
Site web : https://www.ibm.com/software/webservers/appserv/was/

Version	Date de diffusion	
3.5	aout 2000	support J2EE 1.2, requiert Java SE 1.2 3 éditions sont proposées : • SE (Standard Edition) • AE (Advanced Edition) • EE (Enterprise Edition)
4	aout 2001	certifié J2EE 1.2, requiert Java SE 1.3

		Elle permet la mise en oeuvre des servlets, JSP, EJB et services Web (SOAP, UDDI, WSDL, XML). Cette version est proposée en 4 éditions qui supportent tout ou partie de ces composants : • Standard Edition : pour les applications web utilisant des serlets, des JSP et XML • Advanced Edition: supporte en plus les EJB, la répartition de charges sur plusieurs machines • Advanced Single Server Edition : supporte toute les API J2EE mais uniquement sur une seule machine. Cette version ne peut pas être utilisée en production. • Enterprise Edition : supporte en plus CORBA et la connexion aux ressources de l'entreprise
5	janvier 2003	certifiée J2EE 1.3 plusieurs versions sont proposées : • Express Edition • Enterprise Edition
5.1	janvier 2004	requiert Java SE 1.4
6	décembre 2004	certifiée J2EE 1.4 Plusieurs versions sont proposées dont la Community Edition (gratuite) basée sur Apache Geronimo et une version pour z/OS support EJB 3.0
6.1	juin 2006	requiert Java SE 5
7	septembre 2008	certifié Java EE 5, requiert Java SE 6
8	juin 2011	certifié Java EE 6, requiert Java SE 6
8.5	juin 2012	peut fonctionner avec Java SE 7
8.5.5	juin 2013	
9.0	juin 2016	Java EE 7 et Java SE 8

99.2.7. BEA/Oracle Weblogic

Weblogic est une famille de produits proposés par BEA. Weblogic Server est un des leaders mondiaux des serveurs d'applications commerciaux. Weblogic a été racheté par Oracle.

Licence : commerciale
Statut : actif
Site web : https://www.oracle.com/middleware/technologies/weblogic.html

Version	Date de diffusion	
6.0	mars 2001	
7.0	juin 2002	support J2EE 1.3, Java SE 1.3
8.1	juillet 2003	support J2EE 1.3, Java SE 1.4
9.0	novembre 2006	support J2EE 1.4, Java SE 5
10.0	mars 2007	support Java EE 5, Java SE 5
10.3	aout 2008	Java EE 5, Java SE 6

11g (10.3.1)	juillet 2009	Java EE 5, Java SE 6
11g R1 (10.3.2)	novembre 2009	Java EE 5, Java SE 6
11g R1 (10.3.3)	avril 2010	Java EE 5, Java SE 6
11g R1 (10.3.4)	janvier 2011	Java EE 5, Java SE 6
11g R1 (10.3.5)	mai 2011	Java EE 5, Java SE 6
12c	décembre 2011	Java EE 6, Java SE 7
12c R2 (12.2.1.0)	octobre 2015	Java EE 6, Java SE 7
12c R2 (12.2.1.3)	août 2017	Java EE 7, Java SE 8

99.2.8. Oracle Application Server

Oracle propose un serveur d'applications certifié Java EE.

Licence : commerciale
Statut :
Site web :

Version	Date de diffusion	
10g		
11g		

99.2.9. Macromedia JRun

JRun est l'implémentation d'un serveur d'applications de Macromedia.

Licence : commerciale
Statut : ce produit n'est plus maintenu depuis 2007
Site web : http://www.adobe.com/products/jrun/

Version	Date de diffusion	
4		certifié J2EE 1.3

99.3. Les conteneurs web

Les conteneurs web sont des applications qui permettent d'exécuter du code Java utilisé pour définir des servlets et des JSP.

99.3.1. Apache Tomcat

Tomcat est un conteneur d'applications web (servlets et JSP) développé par la fondation Apache. C'est l'implémentation de référence pour les API servlets et JSP : il est donc pleinement compatible avec les spécifications J2EE de ces API.

https://jakarta.apache.org/tomcat/

L'utilisation de Tomcat est détaillée dans le chapitre «Tomcat».

99.3.2. Caucho Resin

Resin est un moteur de servlets et de JSP qui intègre un serveur web.

https://www.caucho.com/

99.3.3. Enhydra

Enhydra est un projet open source, initialement créé par Lutris Technologies, pour développer un conteneur web pour Servlets et JSP. Il fournit quelques fonctionnalités supplémentaires pour utiliser XML, mapper des données avec des objets et gérer un pool de connexions vers des bases de données

99.4. Les conteneurs d'EJB

Les conteneurs d'EJB sont des applications qui fournissent un environnement d'exécution pour les EJB.

99.4.1. OpenEJB

OpenEJB est un projet open source pour développer un conteneur d'EJB qui respecte les spécifications 2.0 des EJB. La version 1.0 est distribuée depuis février 2006.

Le site du projet est l'url https://openejb.apache.org/

99.5. Les outils divers

99.5.1. Jikes

Jikes est un compilateur Java open source écrit par IBM en code natif pour Windows et Linux. Son exécution est donc extrêmement rapide ce qui est d'autant plus appréciable lorsqu'il s'agit de compiler de très gros projets sur une machine peu véloce.

La page de l'outil est à l'url http://jikes.sourceforge.net/

Pour utiliser Jikes, il suffit de décompresser l'archive et de mettre le fichier exécutable dans un répertoire inclus dans le CLASSPATH. Enfin, il faut déclarer une variable système JIKESPATH qui doit indiquer les différents répertoires contenant les classes et les jars notamment le fichier rt.jar du JRE.

99.5.2. GNU Compiler for Java

GCJ faisait parti du projet GCC (GNU Compiler Collection). Le projet GCC propose un compilateur pour plusieurs langages (C, C++, Objective C, Java ...) permettant de produire un exécutable pour plusieurs plates-formes.

GCJ était donc un front-end pour utiliser GCC à partir du code Java. Il permet notamment de :

- compiler du code source Java en bytecode
- compiler du code source Java en un exécutable contenant du code machine dépendant d'un système d'exploitation

Pour un exécutable, le fichier final est lié avec une bibliothèque dédiée nommée libgcj qui contient, entre autres, les classes de bases et le ramasse-miettes.

La plupart des API de la plate-forme Java 2 sont supportées à l'exception notable de la bibliothèque AWT.

Son utilisation sous Windows nécessite un environnement particulier : CygWin ou MinGW (ce dernier étant retenu dans la suite de cette section).

Téléchargez sur le site https://osdn.net/projects/mingw/ les fichiers : MinGW-3.1.0-1.exe (14,5 Mo) et MSYS-1.0.9.exe (2,7 Mo). (les noms de fichiers indiqués correspondent à la version courante au moment de l'écriture de cette section).

Lancez le programme MinGW-3.1.0-1.exe

Le programme d'installation se lance et demande une confirmation de l'installation : cliquer sur « Oui ». Un assistant permet de guider les différentes étapes de l'installation :

- Cliquez sur « Next »
- Lisez la licence et cliquez sur « Yes » si vous l'acceptez
- Lisez les informations et cliquez sur « Next »
- Choisissez le répertoire d'installation et cliquez sur « Next » (pour la suite des instructions, le répertoire par défaut c:\MinGW est utilisé)
- Cliquez sur « Install »
- Une fois l'installation terminée, cliquez sur « Finish »

Lancez le programme MSYS-1.0.9.exe

Le programme d'installation se lance et demande une confirmation de l'installation : cliquer sur « Oui ». Un assistant permet de guider les différentes étapes de l'installation :

- Cliquez sur « Next »
- Lisez la licence et cliquez sur « Yes » si vous l'acceptez
- Lisez les informations et cliquez sur « Next »
- Choisissez le répertoire d'installation et cliquez sur « Next » (pour la suite des instruction, le répertoire C:\MinGW\msys\1.0 est utilisé)
- Sélectionnez l'unique composant à installer et cliquez sur « Next »
- Sélectionnez le raccourci dans le menu Programme (MinGW par défaut) et cliquez sur « Next »
- Cliquez sur « Install »
- L'installation s'exécute et lance un script dos de configuration : il suffit de répondre aux questions

Exemple :

```
C:\MinGW\msys\1.0\postinstall>..\bin\sh.exe pi.sh
This is a post install process that will try to normalize between
your MinGW install if any as well as your previous MSYS installs
if any.  I don't have any traps as aborts will not hurt anything.
Do you wish to continue with the post install? [yn ] y
Do you have MinGW installed? [yn ] y
Please answer the following in the form of c:/foo/bar.
Where is your MinGW installation? c:/Mingw
Creating /etc/fstab with mingw mount bindings.
        Normalizing your MSYS environment.
You have script /bin/awk
You have script /bin/cmd
You have script /bin/echo
You have script /bin/egrep
You have script /bin/ex
You have script /bin/fgrep
You have script /bin/printf
You have script /bin/pwd
You have script /bin/rvi
You have script /bin/rview
You have script /bin/rvim
You have script /bin/vi
You have script /bin/view
Oh joy, you do not have c:/Mingw/bin/make.exe. Keep it that way.
C:\MinGW\msys\1.0\postinstall>pause
Appuyez sur une touche pour continuer...
```

- Appuyez sur une touche pour fermer la boîte DOS
- Une fois l'installation terminée, cliquez sur « Finish »

Remarque : il est fortement recommandé de ne pas utiliser d'espace dans les noms des répertoires d'installation de MinGW et de MSYS.

Il faut pour plus de facilité d'utilisation ajouter à la variable PATH de l'environnement système les répertoires C:\MinGW\bin et C:\MinGW\msys\1.0\bin.

La version de GCC fournie avec MinGW précédemment installée est la 3.2. Pour utiliser GCJ, il faut utiliser la 3.3 et donc opérer une mise à jour.

Il faut télécharger les fichiers gcc-core-3.3.1-20030804-1.tar.gz, gcc-g++-3.3.1-20030804-1.tar.gz et gcc-java-3.3.1-20030804-1.tar.gz, les décompresser et extraire l'image tar dans le répertoire c:\MinGW.

Remarque : en standard aucun outil ne permet de traiter des fichiers gz et tar. Il faut utiliser un outil tiers.

Pour s'assurer de la bonne installation, il suffit d'ouvrir une boîte DOS et d'exécuter la commande gcj. Le message suivant doit apparaître : gcj: no input files

Voici un petit exemple très simple de mise en oeuvre de GCJ.

Exemple du code à compiler :

```
public class Bonjour {
            public static void main(String[] args) {
                        System.out.println("Bonjour");
            }
}
```

Exemple de compilation et d'exécution :

```
D:\java\test\gcj>gcj -o Bonjour Bonjour.java -O --main=Bonjour
D:\java\test\gcj>dir
 Le volume dans le lecteur D s'appelle DATA
 Le numéro de série du volume est 34B2-159D
 Répertoire de D:\java\test\gcj
01/12/2003  15:19       <DIR>          .
01/12/2003  15:19       <DIR>          ..
01/12/2003  15:19            2 747 919 Bonjour.exe
```

```
01/12/2003  15:17                  108 Bonjour.java
01/12/2003  14:07                  141 Bonjour.java.bak
               4 fichier(s)        5 496 087 octets
               2 Rép(s)     560 402 432 octets libres
D:\java\test\gcj>bonjour
Bonjour
D:\java\test\gcj>
```

L'option -o permet de préciser le nom du fichier final généré.

L'option -main= permet de préciser la classe qui contient la méthode main() à lancer par l'exécutable.

GCJ peut être utilisé pour compiler le code source en bytecode grâce à l'option -C.

Exemple :

```
D:\java\test\gcj>gcj -C Bonjour.java
D:\java\test\gcj>dir
 Le volume dans le lecteur D s'appelle DATA
 Le numéro de série du volume est 34B2-159D
 Répertoire de D:\java\test\gcj
01/12/2003  15:29       <DIR>          .
01/12/2003  15:29       <DIR>          ..
01/12/2003  15:29                  389 Bonjour.class
01/12/2003  15:19            2 747 919 Bonjour.exe
01/12/2003  15:17                  108 Bonjour.java
               4 fichier(s)        2 748 557 octets
```

Le bytecode généré est légèrement plus compact que celui généré par la commande javac du jdk 1.4.1

Exemple :

```
D:\java\test\gcj>javac Bonjour.java
D:\java\test\gcj>dir
 Le volume dans le lecteur D s'appelle DATA
 Le numéro de série du volume est 34B2-159D
 Répertoire de D:\java\test\gcj
01/12/2003  15:29       <DIR>          .
01/12/2003  15:29       <DIR>          ..
01/12/2003  15:31                  405 Bonjour.class
01/12/2003  15:19            2 747 919 Bonjour.exe
01/12/2003  15:17                  108 Bonjour.java
               4 fichier(s)        2 748 573 octets
```

L'option -d permet de préciser un répertoire qui va contenir les fichiers .class générés par l'option -C.

99.5.3. Artistic Style

Artistic Style est un outil open source qui permet d'indenter et de formater un code source C, C++ et java

Le site du projet est à l'url https://sourceforge.net/projects/astyle/

Cet outil possède de nombreuses options de formatage de fichiers sources. Les options les plus courantes pour un code source Java sont :

astyle -jp --style=java nomDuFichier.java

Par défaut, l'outil conserve le fichier original en le suffixant par .orig.

99.6. Les MOM

Les Middleware Oriented Message sont des outils qui permettent l'échange de messages entre des composants d'une application ou entre applications. Pour pouvoir les utiliser avec Java, ils doivent implémenter l'API JMS (Java Message Service).

99.6.1. Apache ActiveMQ

ActiveMQ est un projet de la foundation Apache qui propose une implémentation d'un broker JMS respectant la version 1.1 des spécifications.

Licence : open source (licence Apache 2.0)
Statut :
Site web : https://activemq.apache.org/

Il n'est pas nécessaire de déclarer les queues utilisées dans la configuration : ActiveMQ crééra automatiquement les queues qui sont utilisées.

La gestion et le monitoring d'ActiveMQ se fait grâce à JMX.

Des clients utilisant différents langages (.Net, Delphi, C/C++; ...) peuvent accéder à ActiveMQ grâce à son support de différents protocoles.

ActiveMQBrowser est un outil très pratique pour gérer le contenu des queues et topics d'une instance d'ActiveMQ.

Il faut télécharger la dernière version de l'archive à l'url https://sourceforge.net/projects/activemqbrowser/files/ et la décompresser dans un répertoire du système de fichiers.

Pour permettre la connection au Broker, il est nécessaire de le configurer.

Il faut modifier le fichier activemq.xml contenu dans le sous-répertoire conf du répertoire d'installation d'ActiveMQ. Dans le tag <broker>, il faut définir deux propriétées :

- brokerName en lui affectant un nom comme valeur
- useJmx en lui affectant la valeur true

Exemple :

```
    <broker xmlns="http://activemq.apache.org/schema/core"
brokerName="localhost_AMQ"
dataDirectory="${activemq.data}"
useJmx="true">
```

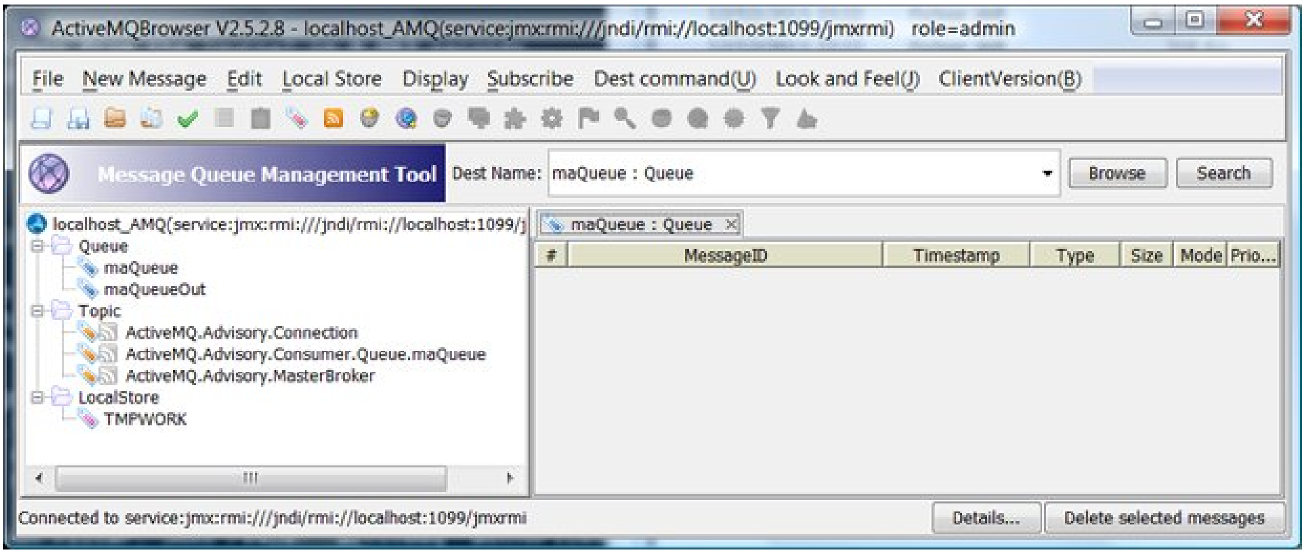
Il faut modifier le fichier activemq.bat contenu dans le sous-répertoire bin du répertoire d'installation d'ActiveMQ en définissant la variable d'environnement SUNJMX :

Résultat :

```
set
SUNJMX=-Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port=1099
-Dcom.sun.management.jmxremote.authenticate=true -Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.password.file=%ACTIVEMQ_BASE%/conf/jmx.password
-Dcom.sun.management.jmxremote.access.file=%ACTIVEMQ_BASE%/conf/jmx.access
```

Cette configuration va activer l'accès JMX sur le port 1099 avec une authentification définie dans les fichiers jmx.access (pour définir les rôles) et jmx.password (pour définir les utilisateurs/mots de passe) du sous-répertoire conf.

Pour lancer ActiveMQBrowser, il faut lancer le script run_activemq_browser.bat contenu dans le répertoire d'installation d'ActiveMQBrowser.


ActiveMQBrowser V2.5.2.8 - localhost_AMQ(service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi) role=admin
File New Message Edit Local Store Display Subscribe Dest command(U) Look and Feel(J) ClientVersion(B)
Message Queue Management Tool
Browse
Search
localhost_AMQ(service:jmx:rmi:///jndi/rmi://localhost:1099/j
Queue
maQueue
maQueueOut
Topic
ActiveMQ.Advisory.Connection
ActiveMQ.Advisory.Consumer.Queue.maQueue
ActiveMQ.Advisory.MasterBroker
LocalStore
TMPWORK
maQueue : Queue
#
MessageID
Timestamp
Type
Size
Mode
Prio...
Details...
Delete selected messages

Dest Name: maQueue : Queue

Connected to service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi

Pour utiliser ActiveMQBrowser avec la version 5.9 d'ActiveMQ, il faut effectuer plusieurs opérations :

- télécharger les fichiers activemq-all-5.9.0.jar et activemq-web-5.9.0.jar par exemple sur un repository Maven et les copier dans le répertoire d'installation d'ActiveMQ et son sous-répertoire lib
- remplacer les versions 5.6 fournies par celles téléchargées
- modifier le script de lancement d'ActiveMQBrowser

Dans le répertoire d'installation d'ActiveMQBrowser, il faut remplacer les fichiers activemq-all-5.6.0.jar et activemq-web-5.6.0.jar par leur équivalent en version 5.9.

Il faut modifier le script en remplaçant les versions 5.6 d'activemq-all.jar et activemq-web.jar par la version 5.9

Résultat :

```
echo off
start "ActiveMQBrowser"
javaw -Xms128m -Xmx512m -splash:cube.png -cp .\QBrowserV2_Neo.jar;
.\activemq-web-5.9.0.jar;.\activemq-all-5.9.0.jar;.\jide-oss-2.6.2.jar;
.\imq.jar;.\jms.jar;.\imqadmin_ja.jar;.\imqadmin.jar;.\imqutil_ja.jar;
.\imqutil.jar;.\imqjmx.jar;.\imqjmx_ja.jar com.qbrowser.ActiveMQBrowser
```

99.6.2. OpenJMS

OpenJMS est une implémentation Open Source des spécifications JMS

Le site officiel du projet est à l'url http://openjms.sourceforge.net/

Pour utiliser OpenJMS, il faut télécharger l'archive qui contient OpenJMS : par exemple le fichier openjms-0.7.7-beta-1.zip

Pour installer OpenJMS, il suffit de décompresser le fichier zip téléchargé dans un répertoire du système.

Exemple :

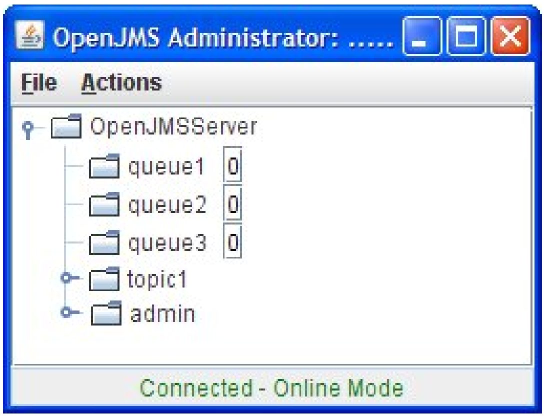
```
C:\java>jar xvf openjms-0.7.7-beta-1.zip
```

La décompression crée un répertoire nommé openjms-0.7.7-beta-1

Pour démarrer et arrêter le serveur, il faut utiliser respectivement les scripts startup et shutdown du sous-répertoire bin du répertoire d'installation.

Pour lancer la console d'administration, il faut utiliser le script admin.

Cliquez sur « Actions / Connections / OnLine ».



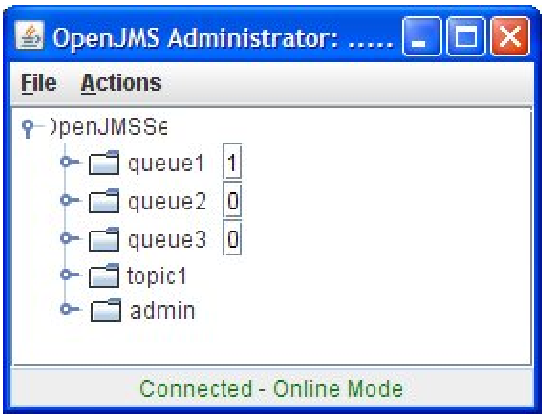
Il est alors possible d'écrire une application qui utilise JMS et les queues, par exemple l'application TestOpenJMS1 fournie dans le chapitre sur «JMS (Java Message Service)».

Les paramètres JNDI peuvent être fournis dans un fichier de configuration nommé jndi.properties :

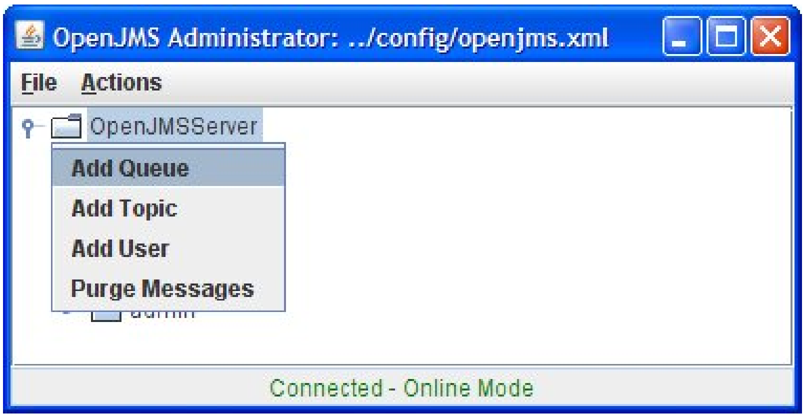
Exemple : jndi.properties

```
java.naming.provider.url=tcp://localhost:3035
java.naming.factory.initial=org.exolab.jms.jndi.InitialContextFactory
java.naming.security.principal=admin
java.naming.security.credentials=openjms
```

Dans la console d'administration, cliquez sur « Actions / Refresh ».



La configuration des queues est stockée dans le fichier openjms.xml. La console d'administration permet de gérer les destinations (ajout, suppression, ...)



99.6.3. Joram

Joram est l'acronyme de Java Open Reliable Asynchronous Messaging. C'est une implémentation open source des spécifications JMS 1.1.

La page officielle de cet outil est à l'url https://joram.ow2.io/

99.6.4. OSMQ

OSMQ™ Open Source Message Queue Open Source Message Queue (OSMQ) est un middleware orienté messages développé en open source par Boston System Group.

99.7. Les outils concernant les bases de données

99.7.1. Derby

Apache Derby Derby est un SGBDR open source écrit en Java et maintenu par le projet Apache.

Historiquement, c'est un produit développé par Cloudscape acquis par IBM (lors de son rachat d'Informix) qui en a fait don à la fondation Apache.

Derby est aussi intégré dans des produits de Sun notamment le JDK 6.0 et GlassFish sous le nom Java DB

Le site officiel est à l'url https://db.apache.org/derby/

99.7.2. SQuirrel-SQL

Squirrel-SQL est un client SQL open source écrit en Java. Il permet au travers d'une interface graphique de consulter et de manipuler une base de données pourvue d'un pilote JDBC.

Le site de l'outil est à l'url : https://squirrel-sql.sourceforge.io/

L'éditeur SQL propose une assistance à l'achévement du code (nom de table, de colonnes, ...).

Les données sont éditables dans l'interface graphique.

SQuirrel est extensible au travers de plug-in dont plusieurs sont fournis par défaut.

Pour installer Squirrel il faut télécharger le fichier squirrel-sql-<version>-install.jar qui est un setup d'installation au format IzPack.

Pour exécuter l'installation, il faut saisir la commande :

```
java -jar squirrel-sql-<version>-install.jar
```

99.8. Les outils de modélisation UML

99.8.1. Argo UML

Argo UML est un projet open source écrit en Java qui vise à développer un outil de modélisation UML 1.1. Il est possible de créer des diagrammes UML et de générer le code Java correspondant aux diagrammes de classes. Une option permet de créer les diagrammes de classes à partir du code source Java.

Cet outil n'est pas encore en version finale mais la version 0.9.5 offre de nombreuses fonctionnalités.

99.8.2. Poseidon for UML

Plusieurs éditions de Poseidon for UML sont proposées dont la version Community Edition qui est disponible gratuitement.

La version 4.1 propose de nombreuses fonctionnalités dont le respect de la version UML 2.0

99.8.3. StarUML

StarUML est historiquement un produit commercial nommé Plastic puis Agora Plastic, qui est devenu un projet open source (licence GPL) en 2005 et renommé StarUML.

Le site officiel de StarUML est à l'url https://staruml.io/

Il ne fonctionne que sous Windows mais la version 5.0 propose des fonctionnalités intéressantes :

- support de UML 2.0 et de MDA
- le support des design patterns (GoF et EJB)
- importation des fichiers Rational Rose
- exportation en XMI

- génération de code et retro conception (Java, C#, C++)
- extensible par plugins reposants sur la technologie COM

100. Ant

Chapitre 100

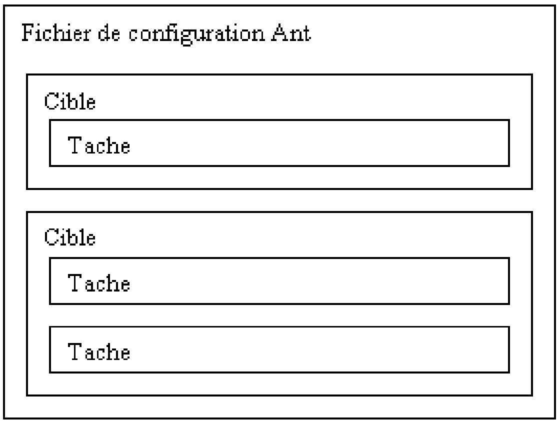
Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Ant est un projet du groupe Apache-Jakarta. Son but est de fournir un outil écrit en Java pour permettre la construction d'applications (compilation, exécution de tâches post et pré compilation, ...). Ces processus de construction d'applications sont très importants car ils permettent d'automatiser des opérations répétitives tout au long du cycle de développement de l'application (développement, tests, recettes, mises en production, ...). Le site officiel de l'outil Ant est https://ant.apache.org/.

Ant pourrait être comparé au célèbre outil make sous Unix. Il a été développé pour fournir un outil de construction indépendant de toute plate-forme. Ceci est particulièrement utile pour des projets développés sur et pour plusieurs systèmes ou, pour migrer des projets d'un système sur un autre. Il est aussi très efficace pour de petits développements.

Ant repose sur un fichier de configuration XML qui décrit les différentes tâches qui devront être exécutées par l'outil. Ant fournit un certain nombre de tâches courantes qui sont codées sous forme d'objets développés en Java. Ces tâches sont donc indépendantes du système sur lequel elles seront exécutées. De plus, il est possible d'ajouter ses propres tâches en écrivant de nouveaux objets Java respectant certaines spécifications.

Le fichier de configuration contient un ensemble de cibles (targets). Chaque cible contient une ou plusieurs tâches. Chaque cible peut avoir une dépendance envers une ou plusieurs autres cibles pour pouvoir être exécutée.



Les environnements de développement intégrés proposent souvent un outil de construction propriétaire qui est généralement moins souple et moins puissant que Ant. Ainsi des plugins ont été développés pour la majorité d'entre-eux (JBuilder, Forte, Visual Age, ...) et ainsi leur permettre d'utiliser Ant, devenu un standard de fait.

Ant possède donc plusieurs atouts : multiplate-forme, configurable grâce à un fichier XML, open source et extensible.

Pour obtenir plus de détails sur l'utilisation de l'outil Ant, il est possible de consulter la documentation de la version courante à l'url suivante : https://ant.apache.org/manual/index.html

Une version 2 de l'outil Ant est en cours de développement.

Ce chapitre contient plusieurs sections :

- L'installation de l'outil Ant
- L'exécution de l'outil Ant
- Le fichier build.xml
- Les tâches (task)

100.1. L'installation de l'outil Ant

Pour pouvoir utiliser Ant, il faut avoir un JDK 1.1 ou supérieur et installer Ant sur la machine.

100.1.1. L'installation sous Windows

Le plus simple est de télécharger la distribution binaire de l'outil Ant pour Windows : jakarta-ant-version-bin.zip sur le site de Ant.

Il suffit ensuite de :

- décompresser le fichier (un répertoire jakarta-ant-version contenant l'outil et sa documentation est créé)
- ajouter le chemin complet du répertoire bin de l'outil Ant à la variable système PATH (pour pouvoir facilement appeler Ant n'importe où dans l'arborescence du système)
- s'assurer que la variable JAVA_HOME pointe sur le répertoire contenant le JDK
- créer une variable d'environnement ANT_HOME qui pointe sur le répertoire jakarta-ant-version créé lors de la décompression du fichier
- il peut être nécessaire d'ajouter les fichiers .jar contenus dans le répertoire lib de l'outil Ant à la variable d'environnement CLASSPATH

Exemple de lignes contenues dans le fichier autoexec.bat :

```
  ...
  set JAVA_HOME=c:\jdk1.3 set
  ANT_HOME=c:\java\ant
  set PATH=%PATH%;%ANT_HOME%\bin
  ...
```

100.2. L'exécution de l'outil Ant

Ant s'utilise en ligne de commandes avec la syntaxe suivante :

ant [options] [cible]

Par défaut, Ant recherche un fichier nommé build.xml dans le répertoire courant. Ant va alors exécuter la cible par défaut définie dans le projet de ce fichier build.xml.

Il est possible de préciser le nom du fichier de configuration en utilisant l'option -buildfile et en la faisant suivre du nom du fichier de configuration.

Exemple :

```
ant -buildfile monbuild.xml
```

Il est possible de préciser une cible à exécuter. Dans ce cas, Ant exécute les cibles dont dépend la cible précisée et exécute cette dernière.

Exemple : exécuter la cible clean et toutes les cibles dont elle dépend

```
ant clean
```

Ant possède plusieurs options dont voici les principales :

Option	Rôle
-quiet	fournit un minimum d'informations lors de l'exécution
-verbose	fournit un maximum d'informations lors de l'exécution
-version	affiche la version de l'outil ant
-projecthelp	affiche les cibles définies avec leurs descriptions
-buildfile	permet de préciser le nom du fichier de configuration
-Dnom=valeur	permet de définir une propriété dont le nom et la valeur sont séparés par un caractère =

100.3. Le fichier build.xml

Le fichier build est un fichier XML qui contient la description du processus de construction de l'application.

Comme tout document XML, le fichier débute par un prologue :

<?xml version="1.0">

L'élément principal de l'arborescence du document est le projet représenté par le tag <project> qui est donc le tag racine du document.

A l'intérieur du projet, il faut définir les éléments qui le composent :

- les cibles (targets) : des étapes du projet de construction
- les propriétés (properties) : des variables qui contiennent des valeurs utilisables par d'autres éléments (cibles ou tâches)
- les tâches (tasks) : des traitements unitaires à réaliser dans une cible donnée

Pour permettre l'exécution sur plusieurs plates-formes, les chemins de fichiers indiqués dans le fichier build.xml doivent utiliser le caractère slash '/' comme séparateur, et ce, même sous Windows qui utilise le caractère anti-slash '\'.

100.3.1. Le projet

Il est défini par le tag racine <project> dans le fichier build.

Ce tag possède plusieurs attributs :

- name : précise le nom du projet
- default : précise la cible par défaut à exécuter si aucune cible n'est précisée lors de l'exécution
- basedir : précise le répertoire qui servira de référence pour l'utilisation d'une localisation relative des autres répertoires.

Exemple :

```
  <project name="mon projet" default="compile" basedir=".">
```

100.3.2. Les commentaires

Les commentaires sont inclus dans un tag <!-- -->.

Exemple :

```
<!-- Exemple de commentaires -->
```

100.3.3. Les propriétés

Le tag <property> permet de définir une propriété qui pourra être utilisée dans le projet : c'est souvent la définition d'un répertoire ou d'une variable qui sera utilisée par certaines tâches. Sa définition, en tant que propriété, permet de facilement définir sa valeur qui pourra être ensuite utilisée plusieurs fois dans le projet.

Exemple :

```
<property name= "nom_appli" value= "monAppli"/>
```

Les propriétés sont immuables et peuvent être définies de deux manières :

- avec le tag <property>
- avec l'option -D sur la ligne de commandes lors de l'appel de la commande ant

Pour utiliser une propriété sur la ligne de commandes, il faut utiliser l'option -D immédiatement suivie du nom de la propriété puis du caractère =, suivi lui-même de la valeur, le tout sans espace.

Le tag <property> possède plusieurs attributs :

- name : définit le nom de la propriété
- value : définit la valeur de la propriété
- location : permet de définir un fichier avec son chemin absolu. Peut être utilisé à la place de l'attribut value
- file : permet de préciser le nom d'un fichier qui contient la définition d'un ensemble de propriétés. Ce fichier sera lu et les propriétés qu'il contient seront définies.

L'utilisation de l'attribut file est particulièrement utile car il permet de séparer la définition des propriétés du fichier build. Le changement d'un paramètre ne nécessite alors pas de modification dans le fichier xml build.

Exemple :

```
<property file="mesproprietes.properties" />
<property name="repSources" value="src" />
<property name="projet.nom" value="mon_projet" />
<property name="projet.version" value="0.0.10" />
```

L'ordre de définition des propriétés est très important : Ant gère cet ordre. La règle est la suivante : la première définition d'une propriété est prise en compte, les suivantes sont ignorées.

Ainsi, les propriétés définies par la ligne de commandes sont prioritaires par rapport à celles définies dans le fichier build. Il est aussi préférable de mettre le tag <property> contenant un attribut file avant les tags <property> définissant des variables.

Pour utiliser une propriété définie dans le fichier, il faut utiliser la syntaxe suivante :
${nom_propriete}

Exemple :

```
${repSources}
```

Il existe aussi des propriétés prédéfinies par Ant et utilisables dans chaque fichier build :

Propriété	Rôle
basedir	chemin absolu du répertoire de travail (cette valeur est précisée dans l'attribut basedir du tag project)
ant.file	chemin absolu du fichier build en cours de traitement
ant.java.version	version de la JVM qui exécute ant
ant.project.name	nom du projet en cours d'utilisation

100.3.4. Les ensembles de fichiers

Le tag <fileset> permet de définir un ensemble de fichiers. Cet ensemble de fichiers sera utilisé dans une autre tâche. La définition d'un tel ensemble est réalisée grâce à des attributs du tag <fileset> :

Attribut	Rôle
dir	Définit le répertoire de départ de l'ensemble de fichiers
includes	Liste des fichiers à inclure
excludes	Liste des fichiers à exclure

L'expression **/ permet de désigner tous les sous-répertoires du répertoire défini dans l'attribut dir.

Exemple :

```
<fileset dir="src" includes="**/*.java">
```

100.3.5. Les ensembles de motifs

Le tag <patternset> permet de définir un ensemble de motifs pour sélectionner des fichiers.

La définition d'un tel ensemble est réalisée grâce à des attributs du tag <patternset> :

Attribut	Rôle
id	Définit un identifiant pour l'ensemble qui pourra ainsi être réutilisé
includes	Liste des fichiers à inclure
excludes	Liste des fichiers à exclure
refid	Demande la réutilisation d'un ensemble dont l'identifiant est fourni comme valeur

L'expression **/ permet de désigner tous les sous-répertoires du répertoire définit dans l'attribut dir. Le caractère ? représente un unique caractère quelconque et le caractère * représente zéro ou n caractères quelconques.

Exemple :

```
<fileset dir="src">
  <patternset id="source_code">
    <includes="**/*.java"/>
  </patternset>
</fileset>
```

100.3.6. Les listes de fichiers

Le tag <filelist> permet de définir une liste finie de fichiers. Chaque fichier est nommément ajouté dans la liste, séparé du suivant par une virgule. La définition d'un tel élément est réalisée grâce à des attributs du tag <filelist> :

Attribut	Rôle
id	Définit un identifiant pour la liste qui pourra ainsi être réutilisée
dir	Définit le répertoire de départ de la liste de fichiers
files	liste des fichiers séparés par des virgules
refid	Demande la réutilisation d'une liste dont l'identifiant est fourni comme valeur

Exemple :

```
<filelist dir="texte" files="fichier1.txt,fichier2.txt" />
```

100.3.7. Les éléments de chemins

Le tag <pathelement> permet de définir un élément qui sera ajouté à la variable classpath. La définition d'un tel élément est réalisée grâce à des attributs du tag <pathelement> :

Attribut	Rôle
location	Définit un chemin d'une ressource qui sera ajoutée
path	

Exemple :

```
<classpath>
  <pathelement location="bin/mabib.jar">
  <pathelement location="lib/">
</classpath>
```

Il est préférable, pour assurer une meilleure compatibilité entre plusieurs systèmes, d'utiliser des chemins relatifs au répertoire de base du projet.

100.3.8. Les cibles

Le tag <target> définit une cible. Une cible est un ensemble de tâches à réaliser dans un ordre précis. Cet ordre correspond à celui des tâches décrites dans la cible.

Le tag <target> possède plusieurs attributs :

- name : contient le nom de la cible. Cet attribut est obligatoire
- description : contient une brève description de la cible. Cet attribut est optionnel mais il est recommandé de l'utiliser car la plupart des IDE l'affichent lors de l'utilisation de l'outil ant
- if : permet de conditionner l'exécution à l'existence d'une propriété. Cet attribut est optionnel
- unless : permet de conditionner l'exécution à l'inexistence de la définition d'une propriété. Cet attribut est optionnel
- depends : permet de définir la liste des cibles dont dépend la cible. Cet attribut est optionnel

Il est possible de faire dépendre une cible d'une ou plusieurs autres cibles du projet. Lorsqu'une cible doit être exécutée, Ant s'assure que les cibles dont elle dépend ont été complètement exécutées préalablement. Une dépendance est définie grâce à l'attribut depends. Plusieurs cibles dépendantes peuvent être listées dans l'attribut depends. Dans ce cas, chaque cible doit être séparée de la suivante avec une virgule.

100.4. Les tâches (task)

Une tâche est une unité de traitements contenue dans une classe Java qui implémente l'interface org.apache.ant.Task. Dans le fichier de configuration, une tâche est un tag qui peut avoir des paramètres pour configurer le traitement à réaliser. Une tâche est obligatoirement incluse dans une cible.

Ant fournit en standard un certain nombre de tâches pour des traitements courants lors du développement en Java :

Catégorie	Nom de la tâche	Rôle
Tâches internes	echo	Afficher un message
	dependset	Définir des dépendances entre fichiers
	taskdef	Définir une tâche externe
	typedef	Définir un nouveau type de données
Gestion des propriétés	available	Définir une propriété si une ressource existe
	condition	Définir une propriété si une condition est vérifiée
	pathconvert	Définir une propriété avec la conversion d'un chemin de fichier spécifique à un OS
	property	Définir une propriété
	tstamp	Initialiser les propriétés DSTAMP, TSTAMP et TODAY avec la date et heure courante
	uptodate	Définir une propriété en comparant la date de modification de fichiers
Tâches Java	java	Exécuter une application dans la JVM
	javac	Compiler des sources Java
	javadoc	Générer la documentation du code source
	rmic	Générer les classes stub et skeleton nécessaires à la technologie rmi
	signjar	Signer un fichier jar
Gestion des archives	ear	Créer une archive contenant une application J2EE
	gunzip	Décompresser une archive
	gzip	Compresser dans une archive
	jar	Créer une archive de type jar
	tar	Créer une archive de type tar
	unjar	Décompresser une archive de type jar
	untar	Décompresser une archive de type tar
	unwar	Décompresser une archive de type war
	unzip	Décompresser une archive de type zip
	war	Créer une archive de type war
	zip	Créer une archive de type zip
Tâches diverses	apply	Exécuter une commande externe appliquée à un ensemble de fichiers
	cvs	Gérer les sources dans CVS
	cvspass	

	exec	Exécuter une commande externe
	genkey	Générer une clé dans un trousseau de clés
	get	Obtenir une ressource à partir d'une URL
	mail	Envoyer un courrier électronique
	replace	Remplacer une chaîne de caractères par une autre
	sql	Exécuter une requête SQL
	style	Appliquer une feuille de style XSLT à un fichier XML
Gestion des fichiers	chmod	Modifier les droits d'un fichier
	copy	Copier un fichier
	delete	Supprimer un fichier
	mkdir	Créer un répertoire
	move	Déplacer ou renommer un fichier
	touch	Modifier la date de modification du fichier avec la date courante
Gestion de l'exécution de l'outil Ant	ant	Exécuter un autre fichier de build
	antcall	Exécuter une cible
	fail	Stopper l'exécution de l'outil Ant
	parallel	Exécuter une tâche en parallèle
	record	Enregistrer les traitements de l'exécution dans un fichier journal
	sequential	Exécuter une tâche en mode séquentiel
	sleep	Faire une pause dans les traitements

Certaines de ces tâches seront détaillées dans les sections suivantes : pour une référence complète de ces tâches, il est nécessaire de consulter la documentation de l'outil Ant.

100.4.1. echo

La tâche <echo> permet d'écrire dans un fichier ou d'afficher un message ou des informations durant l'exécution des traitements.

Les données à utiliser peuvent être fournies dans un attribut dédié ou dans le corps du tag <echo>.

Cette tâche possède plusieurs attributs dont les principaux sont :

Attribut	Rôle
message	Message à afficher
file	Fichier dans lequel le message sera inséré
append	Booléen qui précise si le message est ajouté à la fin du fichier (true) ou si le fichier doit être écrasé avec le message fourni (false)

Exemple :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name="Test avec Ant" default="init" basedir=".">
  <!-- ============================================================ -->
  <!-- Initialisation -->
  <!-- ============================================================ -->
  <target name="init">
    <echo message="Debut des traitements" />
    <echo>
      Fin des traitements du projet ${ant.project.name}
    </echo>
    <echo file="${basedir}/log.txt" append="false" message="Debut des traitements" />
    <echo file="${basedir}/log.txt" append="true" >
Fin des traitements
    </echo>
  </target>
</project>
```

Résultat :

```
C:\java\test\testant>ant
Buildfile: build.xml

init:
     [echo] Debut des traitements
     [echo]
     [echo]       Fin des traitements du projet Test avec Ant
     [echo]

BUILD SUCCESSFUL
Total time: 2 seconds
C:\java\test\testant>type log.txt
Debut des traitements
Fin des traitements

C:\java\test\testant>
```

100.4.2. mkdir

La tâche <mkdir> permet de créer un répertoire avec éventuellement ses répertoires pères si ceux-ci n'existent pas.

Cette tâche possède un seul attribut:

Attribut	Rôle
dir	Précise le chemin et le nom du répertoire à créer

Exemple :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name="Test avec Ant" default="init" basedir=".">
  <!-- =================================================================== -->
  <!-- Initialisation -->
  <!-- =================================================================== -->
  <target name="init">
    <mkdir dir="${basedir}/gen" />
  </target>
</project>
```

Résultat :

```
C:\java\test\testant>ant
Buildfile: build.xml

init:
    [mkdir] Created dir: C:\java\test\testant\gen

BUILD SUCCESSFUL
Total time: 2 seconds
C:\java\test\testant>
```

100.4.3. delete

La tâche <delete> permet de supprimer des fichiers ou des répertoires.

Cette tâche possède plusieurs attributs dont les principaux sont :

Attribut	Rôle
file	Permet de préciser le fichier à supprimer
dir	Permet de préciser le répertoire à supprimer
verbose	Booléen qui permet d'afficher la liste des éléments supprimés
quiet	Booléen qui permet de ne pas afficher les messages d'erreurs
includeEmptyDirs	Booléen qui permet de supprimer les répertoires vides

Exemple :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name="Test avec Ant" default="init" basedir=".">
  <!-- ================================================================= -->
  <!-- Initialisation -->
  <!-- ================================================================= -->
  <target name="init">
    <delete dir="${basedir}/gen" />
    <delete file="${basedir}/log.txt" />
    <delete>
      <fileset dir="${basedir}/bin" includes="**/*.class" />
    </delete>
  </target>
</project>
```

Résultat :

```
C:\java\test\testant>ant
Buildfile: build.xml

init:
   [delete] Deleting directory C:\java\test\testant\gen
   [delete] Deleting: C:\java\test\testant\log.txt

BUILD SUCCESSFUL
Total time: 2 seconds
C:\java\test\testant>
```

100.4.4. copy

La tâche <copy> permet de copier un ou plusieurs fichiers dans le cas où ils n'existent pas dans la cible ou s'ils sont plus récents dans la cible.

Cette tâche possède plusieurs attributs dont les principaux sont :

Attribut	Rôle
file	Désigne le fichier à copier
todir	Permet de préciser le répertoire cible dans lequel les fichiers seront copiés
overwrite	Booléen qui permet d'écraser les fichiers cibles s'ils sont plus récents (false par défaut)

L'ensemble des fichiers concernés par la copie doit être précisé avec un tag fils <fileset>.

Exemple :

```
<project name="utilisation de hbm2java" default="init" basedir=".">

  <!-- Definition des proprietes du projet -->
  <property name="projet.sources.dir"    value="src"/>
  <property name="projet.bin.dir"        value="bin"/>

  <!-- Initialisation des traitements -->
  <target name="init" description="Initialisation">
    <!-- Copie des fichiers de mapping et parametrage -->
    <copy todir="${projet.bin.dir}" >
      <fileset dir="${projet.sources.dir}" >
        <include name="**/*.properties"/>
        <include name="**/*.hbm.xml"/>
        <include name="**/*.cfg.xml"/>
      </fileset>
    </copy>
  </target>
</project>
```

Résultat :

```
C:\java\test\testhibernate>ant
Buildfile: build.xml

init:
     [copy] Copying 3 files to C:\java\test\testhibernate\bin

BUILD SUCCESSFUL
Total time: 3 seconds
```

100.4.5. tstamp

La tâche <tstamp> permet de définir trois propriétés :

- DSTAMP : initialisée avec la date du jour au format AAAMMJJ
- TSTAMP : initialisée avec l'heure actuelle sous la forme HHMM
- TODAY : initialisée avec la date du jour au format long

Cette tâche ne possède pas d'attribut.

Exemple :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name="Test avec Ant" default="init" basedir=".">
  <!-- ================================================================ -->
  <!-- Initialisation -->
  <!-- ================================================================ -->
  <target name="init">
    <tstamp/>
    <echo message="Nous sommes le ${TODAY}" />
    <echo message="DSTAMP = ${DSTAMP}" />
    <echo message="TSTAMP = ${TSTAMP}" />
  </target>
</project>
```

Résultat :

```
C:\java\test\testant>ant
Buildfile: build.xml

init:
     [echo] Nous sommes le August 25 2004
     [echo] DSTAMP = 20040825
     [echo] TSTAMP = 1413

BUILD SUCCESSFUL
Total time: 2 seconds
```

100.4.6. java

La tâche <java> permet de lancer une machine virtuelle pour exécuter une application compilée.

Cette tâche possède plusieurs attributs dont les principaux sont :

Attribut	Rôle
classname	nom pleinement qualifié de la classe à exécuter
jar	nom du fichier de l'application à exécuter
classpath	classpath pour l'exécution. Il est aussi possible d'utiliser un tag fils <classpath> pour le spécifier
classpathref	utilisation d'un classpath précédemment défini dans le fichier de build
fork	lancer l'exécution dans une JVM dédiée au lieu de celle où s'exécute Ant
output	enregistrer les sorties de la console dans un fichier

Le tag fils <arg> permet de fournir des paramètres à l'exécution.

Le tag fils <classpath> permet de définir le classpath à utiliser lors de l'exécution

Exemple :

```
<project name="testhibernate1" default="TestHibernate1" basedir=".">

  <!-- Definition des proprietes du projet -->
  <property name="projet.sources.dir"    value="src"/>
  <property name="projet.bin.dir"        value="bin"/>
  <property name="projet.lib.dir"        value="lib"/>

  <!-- Definition du classpath du projet -->
  <path id="projet.classpath">
    <fileset dir="${projet.lib.dir}">
      <include name="*.jar"/>
    </fileset>
    <pathelement location="${projet.bin.dir}" />
  </path>

  <!-- Execution de TestHibernate1 -->
  <target name="TestHibernate1" description="Execution de TestHibernate1" >
    <java classname="TestHibernate1" fork="true">
      <classpath refid="projet.classpath"/>
    </java>
  </target>
</project>
```

100.4.7. javac

La tâche <javac> permet la compilation de fichiers sources contenus dans une arborescence de répertoires.

Cette tâche possède plusieurs attributs dont les principaux sont :

Attribut	Rôle
srcdir	précise le répertoire racine de l'arborescence du répertoire contenant les sources
destdir	précise le répertoire où les résultats des compilations seront stockés
classpath	classpath pour l'exécution. Il est aussi possible d'utiliser un tag fils <classpath> pour le spécifier
classpathref	utilisation d'un classpath précédemment défini dans le fichier de build
nowarn	précise si les avertissements du compilateur doivent être affichés. La valeur par défaut est off

debug	précise si le compilateur doit inclure les informations de débogage dans les fichiers compilés. La valeur par défaut est off
optimize	précise si le compilateur doit optimiser le code compilé qui sera généré. La valeur par défaut est off
deprecation	précise si les avertissements du compilateur concernant l'usage d'éléments deprecated doivent être affichés. La valeur par défaut est off
target	précise la version de la plate-forme Java cible (1.1, 1.2, 1.3, 1.4, ...)
fork	lance la compilation dans une JVM dédiée au lieu de celle où s'exécute Ant. La valeur par défaut est false
failonerror	précise si les erreurs de compilations interrompent l'exécution du fichier de build. La valeur par défaut est true
source	version des sources Java : particulièrement utile pour Java 1.4 et 1.5 qui apportent des modifications à la grammaire du langage Java

Exemple :

```
<project name="compiltation des classes" default="compile" basedir=".">
  <!-- Definition des proprietes du projet -->
  <property name="projet.sources.dir" value="src"/>
  <property name="projet.bin.dir" value="bin"/>
  <property name="projet.lib.dir" value="lib"/>

  <!-- Definition du classpath du projet -->
  <path id="projet.classpath">
    <fileset dir="${projet.lib.dir}">
      <include name="*.jar"/>
    </fileset>
    <pathelement location="${projet.bin.dir}" />
  </path>

  <!-- Compilation des classes du projet -->
  <target name="compile" description="Compilation des classes">
    <javac srcdir="${projet.sources.dir}"
                  destdir="${projet.bin.dir}"
                          debug="on"
                          optimize="off"
                          deprecation="on">
      <classpath refid="projet.classpath"/>
    </javac>
  </target>
</project>
```

Résultat :

```
C:\java\test\testhibernate>antBuildfile: build.xmlcompile:
    [javac] Compiling 1 source file to C:\java\test\testhibernate\bin
    [javac] C:\java\test\testhibernate\src\TestHibernate1.java:9: cannot resolve symbol
    [javac] symbol : class configuration
    [javac] location: class TestHibernate1
    [javac] Configuration config = new configuration();
    [javac] ^
    [javac] 1 error

BUILD FAILED
file:C:/java/test/testhibernate/build.xml:22: Compile failed; see the compiler e
rror output for details.

Total time: 9 seconds
```

100.4.8. javadoc

La tâche <javadoc> permet de demander la génération de la documentation au format javadoc des classes incluses dans une arborescence de répertoires.

Cette tâche possède plusieurs attributs dont les principaux sont :

Attribut	Rôle
sourcepath	précise le répertoire de base qui contient les sources dont la documentation est à générer
destdir	précise le répertoire qui va contenir les fichiers de documentation générés

Exemple :

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<project name="Test avec Ant" default="javadoc" basedir=".">
  <!-- ================================================-->
  <!-- Génération de la documentation Javadoc          -->
  <!-- ================================================-->
  <target name="javadoc">
    <javadoc sourcepath="src"
                destdir="doc" >
      <fileset dir="src" defaultexcludes="yes">
        <include name="**" />
      </fileset>
    </javadoc>
  </target>
</project>
```

Résultat :

```
C:\java\test\testant>ant
Buildfile: build.xml

javadoc:
  [javadoc] Generating Javadoc
  [javadoc] Javadoc execution
  [javadoc] Loading source file C:\java\test\testant\src\MaClasse.java...
  [javadoc] Constructing Javadoc information...
  [javadoc] Standard Doclet version 1.4.2_02
  [javadoc] Building tree for all the packages and classes...
  [javadoc] Building index for all the packages and classes...
  [javadoc] Building index for all classes...

BUILD SUCCESSFUL
Total time: 9 seconds
```

100.4.9. jar

La tâche <jar> permet la création d'une archive de type jar.

Cette tâche possède plusieurs attributs dont les principaux sont :

Attribut	Rôle
jarfile	nom du fichier .jar à créer
basedir	précise de répertoire qui contient les éléments à ajouter dans l'archive
compress	précise si le contenu de l'archive doit être compressé ou non. La valeur par défaut est true
manifest	précise le fichier manifest qui sera utilisé dans l'archive

Exemple :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <project name="Test avec Ant" default="packaging" basedir=".">

  <!-- =================================================================== -->
  <!-- Génération de l'archive jar                           -->
  <!-- =================================================================== -->
  <target name="packaging">
    <jar jarfile="test.jar" basedir="src" />
  </target>
</project>
```

Résultat :

```
C:\java\test\testant>ant
Buildfile: build.xml

packaging:
      [jar] Building jar: C:\java\test\testant\test.jar

BUILD SUCCESSFUL
Total time: 2 seconds
```

La suite de ce chapitre sera développée dans une version future de ce document

101. Maven

Chapitre 101

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Maven est un outil de construction de projets (build) open source développé par la fondation Apache, initialement pour les besoins du projet Jakarta Turbine. Il permet de faciliter et d'automatiser certaines tâches de la gestion d'un projet Java.

Le site web officiel est https://maven.apache.org

Il permet notamment :

- d'automatiser certaines tâches : compilation, tests unitaires et déploiement des applications qui composent le projet
- de gérer des dépendances vis-à-vis des bibliothèques nécessaires au projet
- de générer des documentations concernant le projet

Au premier abord, il est facile de croire que Maven fait double emploi avec Ant. Ant et Maven sont tous les deux développés par le groupe Jakarta, ce qui prouve bien que leur utilité n'est pas aussi identique que cela. Ant, dont le but est d'automatiser certaines tâches répétitives, est plus ancien que Maven. Maven propose non seulement les fonctionnalités d'Ant mais en propose de nombreuses autres.

Pour gérer les dépendances du projet vis-à-vis de bibliothèques, Maven utilise un ou plusieurs dépôts qui peuvent être locaux ou distants.

Maven est extensible grâce à un mécanisme de plugins qui permettent d'ajouter des fonctionnalités.

Ce chapitre contient plusieurs sections :

♦ Les différentes versions de Maven
♦ Maven 1
♦ Maven 2

101.1. Les différentes versions de Maven

Plusieurs versions majeures de Maven ont été diffusées :

- version 1.0, juillet 2004
- version 1.1, juin 2007
- version 2.0, octobre 2005
- version 2.1, mars 2009
- version 3.0, octobre 2010
- version 3.1, juillet 2013

Maven 2 est très différent de Maven 1.

Maven 3 est compatible avec Maven 2 et apporte notamment de meilleures performances.

101.2. Maven 1

Pour assurer la construction d'un projet, Maven propose notamment de prendre en charge :

- La compilation
- Le packaging
- La gestion des dépendances
- La génération de la documentation
- L'accès au gestionnaire de sources
- L'accès aux dépôts ou aux gestionnaires de dépendances
- Le déploiement
- ... et de très nombreuses autres tâches requises lors d'un build

Maven impose par défaut l'emploi de conventions notamment dans la structuration du projet.

Maven utilise une approche déclarative où la structure du projet et son contenu sont décrits dans un document XML. De plus il convient de se conformer à une structure de projets standards et de bonnes pratiquees. L'observation de ces normes permet de réduire le temps nécessaire pour écrire et maintenir les scripts de build car ils sont tous structurés de la même façon.

La description d'un projet est faite dans un fichier XML nommé POM (Project Object Model). Cette description contient notamment les dépendances, les spécificités de construction (compilation et packaging), éventuellement le déploiement, la génération de la documentation, l'exécution d'outils d'analyse statique du code, ...

Maven peut aussi assurer de nombreuses autres tâches car il est conçu pour utiliser des plugins : il est donc extensible. Maven est fourni avec un grand nombre de plugins standard mais il est aussi possible d'utiliser d'autres plugins qui sont stockés dans les dépôts voire même de développer ses propres plugins.

Maven permet une gestion des artefacts (dépendances, plugin-ins) qui sont stockées dans un ou plusieurs dépôts (repository).

101.2.1. L'installation

Il faut télécharger le fichier maven-1.0-rc2.exe sur le site de Maven et l'exécuter.

Un assistant permet de fournir les informations concernant l'installation :

- sur la page « Licence Agreement » : lire la licence et si vous l'acceptez cliquer sur le bouton « I Agree ».
- sur la page « Installations Options » : sélectionner les éléments à installer et cliquer sur le bouton « Next ».
- sur la page « Installation Folder » : sélectionner le répertoire dans lequel Maven va être installé et cliquer sur le bouton « Install ».
- une fois les fichiers copiés, il suffit de cliquer sur le bouton « Close ».

Sous Windows, un élément de menu nommé « Apache Software Foundation / Maven 1.0-rc2 » est ajouté dans le menu « Démarrer / Programmes ».

Pour utiliser Maven, la variable d'environnement système nommée MAVEN_HOME doit être définie avec comme valeur le chemin absolu du répertoire dans lequel Maven est installé. Par défaut, cette variable est configurée automatiquement lors de l'installation sous Windows.

Il est aussi particulièrement pratique d'ajouter le répertoire %MAVEN_HOME%/bin à la variable d'environnement PATH. Maven étant un outil en ligne de commande, cela évite d'avoir à saisir son chemin complet lors de son exécution.

Enfin, il faut créer un repository local en utilisant la commande ci-dessous dans une boîte de commandes DOS :

Exemple :

```
C:\>install_repo.bat %HOMEDRIVE%%HOMEPATH%
```

Pour s'assurer de l'installation correcte de Maven, il suffit de saisir la commande :

Exemple :

```
C:\>maven -v
 __  __
|  \/  |__ _Apache__ ___
| |\/| / _` \ V / -_) ' \  ~ intelligent projects ~
|_|  |_\__,_|\_/\___|_||_|  v. 1.0-rc2
C:\>
```

Lors de la première exécution de Maven, ce dernier va constituer le repository local (une connexion internet est nécessaire).

Exemple :

```
|  \/  |__ _Apache__ ___
| |\/| / _` \ V / -_) ' \  ~ intelligent projects ~
|_|  |_\__,_|\_/\___|_||_|  v. 1.0-rc2
Le rï¿½pertoire C:\Documents and Settings\Administrateur\.maven\repository n'exist
e pas. Tentative de crï¿½ation.
Tentative de tï¿½lï¿½chargement de commons-lang-1.0.1.jar.
.................................
.
Tentative de tï¿½lï¿½chargement de commons-net-1.1.0.jar.
................................................................................
........
.
Tentative de tï¿½lï¿½chargement de dom4j-1.4-dev-8.jar.
................................................................................
................................................................................
................................................................................
..............................................................
.
Tentative de tï¿½lï¿½chargement de xml-apis-1.0.b2.jar.
.......................................................
```

101.2.2. Les plugins

Toutes les fonctionnalités de Maven sont proposées sous la forme de plugins.

Le fichier maven.xml permet de configurer les plugins installés.

101.2.3. Le fichier project.xml

Maven est orienté projet, donc le projet est l'entité principale gérée par Maven. Il est nécessaire de fournir à Maven une description du projet (Project descriptor) sous la forme d'un document XML nommé project.xml et situé à la racine du répertoire contenant le projet.

Exemple : un fichier minimaliste

```
  <project>

    <id>P001</id>
    <name>TestMaven</name>
    <currentVersion>1.0</currentVersion>
    <shortDescription>Test avec Maven</shortDescription>
    <developers>
      <developer>
        <name>Jean Michel D.</name>
        <id>jmd</id>
        <email>jmd@test.fr</email>
      </developer>
    </developers>

    <organization>
      <name>Jean-Michel</name>
    </organization>

  </project>
```

Il est possible d'inclure la valeur d'un tag défini dans le document dans un autre tag.

Exemple :

```
  ...
  <shortDescription>${pom.name} est un test avec Maven</shortDescription>
  ...
```

Il est possible d'hériter d'un fichier project.xml existant dans lequel des caractéristiques communes à plusieurs projets sont définies. La déclaration dans le fichier du fichier père se fait avec le tag <extend>. Dans le fichier fils, il suffit de redéfinir ou de définir les tags nécessaires.

101.2.4. L'exécution de Maven

Maven s'utilise en ligne de commande sous la forme suivante :

Maven plugin:goal

Il faut exécuter Maven dans le répertoire qui contient le fichier project.xml.

Si les paramètres fournis ne sont pas corrects, une exception est levée :

Exemple :

```
C:\java\test\testmaven>maven compile
 __  __
|  \/  |__ _Apache__ ___
| |\/| / _` \ V / -_) ' \  ~ intelligent projects ~
|_|  |_\__,_|\_/\___|_||_|  v. 1.0-rc2
com.werken.werkz.NoSuchGoalException: No goal [compile]
        at com.werken.werkz.WerkzProject.attainGoal(WerkzProject.java:190)
        at org.apache.maven.plugin.PluginManager.attainGoals(PluginManager.java:
531)
        at org.apache.maven.MavenSession.attainGoals(MavenSession.java:265)
        at org.apache.maven.cli.App.doMain(App.java:466)
        at org.apache.maven.cli.App.main(App.java:1117)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.
java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAcces
sorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:324)
        at com.werken.forehead.Forehead.run(Forehead.java:551)
        at com.werken.forehead.Forehead.main(Forehead.java:581)
Total time: 4 seconds
Finished at: Tue May 18 14:17:18 CEST 2004
```

Pour obtenir une liste complète des plugins à disposition de Maven, il suffit d'utiliser la commande maven -g

Voici quelques-uns des nombreux plugins avec leurs goals principaux :

Plug in	Goal	Description
ear	ear	construire une archive de type ear
	deploy	déployer un fichier ear dans un serveur d'application
ejb	ejb	
	deploy	
jalopy	format	
java	compile	compiler des sources
	jar	créer une archive de type .jar
javadoc		
jnlp		
pdf		générer la documentation du projet au format PDF
site	generate	générer le site web du projet
	deploy	copier le site web sur un serveur web
test	match	exécuter des tests unitaires
war	init	
	war	
	deploy	

La commande maven clean permet d'effacer tous les fichiers générés par Maven.

Exemple :

```
C:\java\test\testmaven>maven clean
 __  __
|  \/  |__ _Apache__ ___
| |\/| / _` \ V / -_) ' \  ~ intelligent projects ~
|_|  |_\__,_|\_/\___|_||_|  v. 1.0-rc2
build:start:
clean:clean:
    [delete] Deleting directory C:\java\test\testmaven\target
    [delete] Deleting: C:\java\test\testmaven\velocity.log
BUILD SUCCESSFUL
Total time: 2 minutes 7 seconds
Finished at: Tue May 25 14:19:03 CEST 2004
C:\java\test\testmaven>
```

101.2.5. La génération du site du projet

Maven propose une fonctionnalité qui permet de générer automatiquement un site web pour le projet regroupant un certain nombre d'informations utiles le concernant.

Pour demander la génération du site, il suffit de saisir la commande

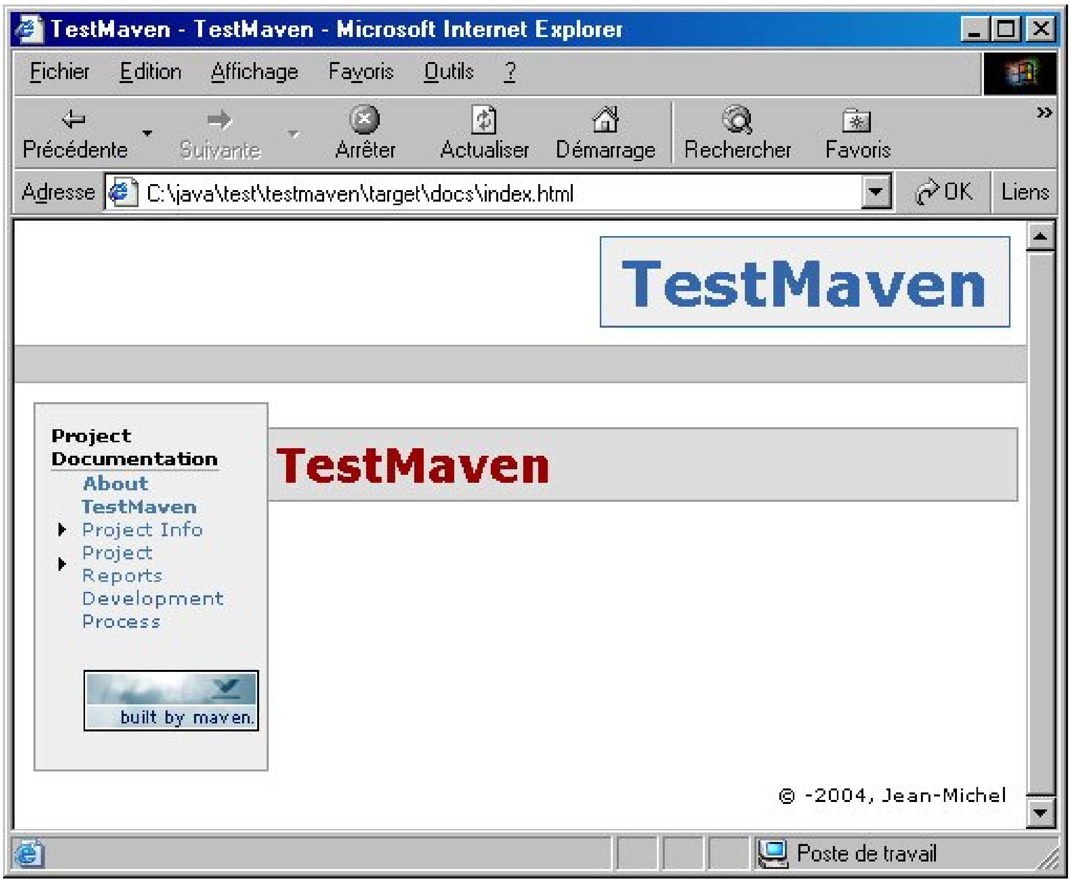
maven site:generate

Lors de l'exécution de cette commande, un répertoire target/docs est créé contenant les différents éléments du site.

Exemple :

```
C:\java\test\testmaven\target\docs>dir
```

```
 Le volume dans le lecteur C s'appelle MACHINE
 Le numéro de série du volume est 3T78-19E4
 Répertoire de C:\java\test\testmaven\target\docs
25/05/2004  14:21       <DIR>          .
25/05/2004  14:21       <DIR>          ..
25/05/2004  14:21       <DIR>          apidocs
25/05/2004  14:21                5 961 checkstyle-report.html
25/05/2004  14:21                1 637 cvs-usage.html
25/05/2004  14:21                1 954 dependencies.html
25/05/2004  14:21       <DIR>          images
25/05/2004  14:21                1 625 index.html
25/05/2004  14:21                1 646 issue-tracking.html
25/05/2004  14:21                3 119 javadoc.html
25/05/2004  14:21                9 128 jdepend-report.html
25/05/2004  14:21                2 494 license.html
25/05/2004  14:21                5 259 linkcheck.html
25/05/2004  14:21                1 931 mail-lists.html
25/05/2004  14:21                4 092 maven-reports.html
25/05/2004  14:21                3 015 project-info.html
25/05/2004  14:21       <DIR>          style
25/05/2004  14:21                2 785 task-list.html
25/05/2004  14:21                3 932 team-list.html
25/05/2004  14:21       <DIR>          xref
              14 fichier(s)          48 578 octets
               6 Rép(s)     207 151 616 octets libres
```



C:\java\test\testmaven\target\docs\index.html

TestMaven

TestMaven

© -2004, Jean-Michel

Par défaut, le site généré contient un certain nombre de pages accessibles par le menu de gauche.

La partie « Project Info » regroupe trois pages : la mailing liste, la liste des développeurs et les dépendances du projet.

La partie « Project report » permet d'avoir accès à des comptes rendus d'exécution de certaines tâches : javadoc, tests unitaires, ... Certaines de ces pages ne sont générées qu'en fonction des différents éléments produits par Maven.

Le contenu du site pourra donc être réactualisé facilement en fonction des différents traitements réalisés par Maven sur le projet.

101.2.6. La compilation du projet

Dans le fichier project.xml, il faut rajouter un tag <build> qui va contenir les informations pour la compilation des éléments du projet.

Les sources doivent être contenues dans un répertoire dédié, par exemple src

Exemple :

```
  ...
    <build>
        <sourceDirectory>
          ${basedir}/src
        </sourceDirectory>
      </build>
  ...
```

Pour demander la compilation à Maven, il faut utiliser la commande Maven java :compile :

Exemple :

```
C:\java\test\testmaven>maven java:compile
 __  __
|  \/  |__ _Apache__ ___
| |\/| / _` \ V / -_) ' \  ~ intelligent projects ~
|_|  |_\__,_|\_/\___|_||_|  v. 1.0-rc2
Tentative de tï¿½lï¿½chargement de commons-jelly-tags-antlr-20030211.143720.jar.
.....
.
build:start:
java:prepare-filesystem:
    [mkdir] Created dir: C:\java\test\testmaven\target\classes
java:compile:
    [echo] Compiling to C:\java\test\testmaven/target/classes
    [javac] Compiling 1 source file to C:\java\test\testmaven\target\classes
BUILD SUCCESSFUL
Total time: 12 seconds
Finished at: Tue May 18 14:19:12 CEST 2004
```

Le répertoire « target/classes » est créé à la racine du répertoire du projet. Les fichiers .class issus de la compilation sont stockés dans ce répertoire.

La commande maven jar permet de demander la génération du packaging de l'application.

Exemple :

```
build:start:
java:prepare-filesystem:
java:compile:
    [echo] Compiling to C:\java\test\testmaven/target/classes
java:jar-resources:
test:prepare-filesystem:
    [mkdir] Created dir: C:\java\test\testmaven\target\test-classes
    [mkdir] Created dir: C:\java\test\testmaven\target\test-reports
test:test-resources:
test:compile:
    [echo] No test source files to compile.
test:test:
    [echo] No tests to run.
jar:jar:
    [jar] Building jar: C:\java\test\testmaven\target\P001-1.0.jar
BUILD SUCCESSFUL
Total time: 2 minutes 42 seconds
Finished at: Tue May 18 14:25:39 CEST 2004
```

Par défaut, l'appel à cette commande effectue une compilation des sources, un passage des tests unitaires s'il y en a et un appel à l'outil jar pour réaliser le packaging.

Le nom du fichier jar créé est composé de l'id du projet et du numéro de version. Il est stocké dans le répertoire racine du projet.

101.3. Maven 2

Maven 2 est une version différente de Maven 1 : ces deux versions ne sont d'ailleurs pas compatibles.

La version du fichier POM qui décrit un projet est passé de la version 3.0 à 4.0 : le nom par défaut est pom.xml avec Maven 2.

Le coeur de Maven 2 utilise un conteneur d'injection de dépendances (IoC) nommé Plexus.

101.3.1. L'installation et la configuration

Maven est un outil écrit en Java : Java doit donc être installé sur la machine. Généralement, surtout sur un ordinateur qui n'est pas connecté sur un réseau d'entreprise disposant d'un gestionnaire de dépôts, une connexion à internet est nécessaire pour permettre le téléchargement des plugins requis et des dépendances.

Pour installer Maven, il faut :

- Télécharger l'archive sur le site https://maven.apache.org/download.html
- Décompresser l'archive dans un répertoire du système
- Créer la variable d'environnement M2_HOME qui pointe sur le répertoire contenant Maven
- Ajouter le chemin M2_HOME/bin à la variable PATH du système

Pour vérifier l'installation, il faut lancer la commande mvn -version

Le répertoire de Maven contient plusieurs sous-répertoires :

- bin : des scripts dont la commande mvn
- conf : contient la configuration par défaut dans le fichier settings.xml
- lib : les bibliothèques contenant le noyau de Maven

La configuration du chemin du dépôt local se fait dans le fichier settings.xml du sous-répertoire .m2 contenu dans le répertoire home de l'utilisateur.

Le tag localRepository permet de préciser le chemin absolu du dépôt local.

Exemple :

```
<settings>
  ...
<localRepository>C:\Documents and Settings\jm\.m2\repository</localRepository>
  ...
</settings>
```

Pour tester l'installation, il est possible d'exécuter la commande mvn -clean

Cette commande échoue car elle ne trouve pas de fichier POM mais auparavant, elle télécharge des plugins du dépôt central vers le dépôt local.

101.3.2. Les concepts

Maven repose sur l'utilisation de plusieurs concepts :

- Les artéfacts : composants identifiés de manière unique
- Le principe de convention over configuration : utilisation de conventions par défaut pour standardiser les projets
- Le cycle de vie et les phases : les étapes de construction d'un projet sont standardisées
- Les dépôts (local et distant)

101.3.2.1. Les artéfacts

Un artéfact est un composant packagé possédant un identifiant unique composé de trois éléments : un groupId, un artifactId et un numéro de version.

La gestion des versions est importante pour identifier quel artefact doit être utilisé : la version est utilisée comme une partie de l'identifiant d'un artéfact.

Les versions standard correspondent à des releases de l'artéfact.

Les versions en cours de développement se terminent par -SNAPSHOT : ce sont des versions intermédiaires de travail en local.

Maven va systématiquement rechercher une version plus récente pour une dépendance dont le numéro de version est un SNAPSHOT.

Le numéro de version d'un artéfact Maven se compose généralement de plusieurs informations :

- majeur
- mineur
- bug fixe
- qualificateur : permet de préciser une version antérieure à une release (exemple alpha-1, beta-1, rc-1, ...). Une version avec qualificateur est plus récente qu'une version sans
- numéro de build : une version avec numéro de build est plus ancienne qu'une version sans

Exemple

1.2.10-20131112.2132121-1

Maven propose une syntaxe particulière pour désigner potentiellement plusieurs numéros de versions

Exemple :

[1.0,) : version 1.0 ou ultérieure

(,1.0] : version antérieure ou égale à 1.0

[1.0,1.2] : entre les versions 1.0 et 1.2 incluses

(,1.2),(1.2,) : toutes les versions sauf la 1.2

[1.0,2.0) : version supérieure ou égale à 1.0 et inférieure à 2.0

101.3.2.2. Convention plutôt que configuration

Maven met en oeuvre le principe de convention over configuration pour utiliser par défaut les mêmes conventions.

Par exemple, l'arborescence d'un projet Maven est par défaut imposée par Maven. Contrairement à d'autres outils comme Ant, l'arborescence de base de chaque projet Maven est toujours la même par défaut :

Répertoire	Contenu
/src	les sources du projet (répertoire qui doit être ajouté dans le gestionnaire de sources)
/src/main	les fichiers sources principaux
/src/main/java	le code source (sera compilé dans /target/classses)
/src/main/resources	les fichiers de ressources (fichiers de configuration, images, ...). Le contenu de ce répertoire est copié dans target/classes pour être inclus dans l'artéfact généré
/src/main/webapp	les fichiers de la webapp
/src/test	les fichiers pour les tests
/src/test/java	le code source des tests (sera compilé dans /target/test-classses)
/src/test/resources	les fichiers de ressources pour les tests
/target	les fichiers générés pour les artéfacts et les tests (ce répertoire ne doit pas être inclus dans le gestionnaire de sources)
/target/classes	les classes compilées
/target/test-classes	les classes compilées des tests unitaires
/target/site	site web contenant les rapports générés et des informations sur le projet
/pom.xml	le fichier POM de description du projet

L'utilisation de ces conventions est un des points forts de Maven car elle permet aux développeurs de facilement être familiarisés avec la structure des projets qui est toujours la même.

Pour des besoins particuliers, il est possible de configurer une autre structure de répertoires mais cela n'est pas recommandé essentiellement car :

- La compréhension pour un nouveau développeur sur le projet est plus difficile
- Le fichier de configuration POM est plus complexe
- Il n'est généralement pas recommandé de sortir des standards

Maven propose aussi en standard d'autres conventions notamment :

- Chaque projet possède un artéfact principal par défaut
- Le nom des artéfacts est normalisé

101.3.2.3. Le cycle de vie et les phases

Maven 2 a standardisé le cycle de vie du projet en phases. Le cycle de vie par défaut de Maven contient plusieurs phases dont les principales sont : validate, compile, test, package, install et deploy.

Par défaut, aucun goal n'est associé aux phases du cycle de vie. En fonction du type de packaging du livrable du projet (jar, war, ejb, ejb3, ear, rar, par, pom, maven-plugin, ...) des goals différents sont associés aux différentes phases du cycle de vie Build de Maven. Le packaging par défaut est jar. Les goals exécutés pour chaque phase dépendent du type d'artéfact du projet : par exemple, la phase package exécute le goal jar:jar pour un artéfact de type jar, le goal war:war pour un artéfact de type war.

Il existe deux autres phases utiles :

- clean : effacer tous les éléments générés lors des exécutions précédentes
- site : générer un site web documentant le projet

Les phases et les goals peuvent être exécutés l'un après l'autre

Exemple :

```
mvn clean dependency:copy-dependencies package
```

Cette commande efface les fichiers générés, copie les dépendances et exécute les phases jusqu'à la phase package.

Le cycle de vie clean contient plusieurs phases :

- pre-clean
- clean
- post-clean

L'invocation d'une phase particulière du cycle de vie implique l'exécution de toutes les phases définies dans le cycle de vie qui la précède. Ainsi l'exécution de la commande mvn clean:clean exécutera les phases pre clean et clean.

Le goal clean:clean supprime tous les fichiers contenus dans le répertoire target où sont stockés les éléments produits par le build.

La phase pre clean peut permettre de réaliser des actions à exécuter avant la phase clean et la phase post clean peut permettre d'exécuter des actions après la phase clean.

101.3.2.4. Les archétypes

Un archétype est un modèle de projet. Maven propose en standard plusieurs archétypes dont les principaux sont :

Archétype	Description
maven-archetype-archetype	Un archétype pour un exemple d'archétype
maven-archetype-j2ee-simple	Un archétype pour un projet de type application J2EE simplifié
maven-archetype-mojo	Un archétype pour un exemple de plugin Maven
maven-archetype-plugin	Un archétype pour un projet de type plugin Maven
maven-archetype-plugin-site	Un archétype pour un exemple de site pour un plugin Maven
maven-archetype-portlet	Un archétype pour un projet utilisant les portlets
maven-archetype-quickstart	Un archétype pour un exemple de projet Maven
maven-archetype-simple	Un archétype pour un simple projet Maven
maven-archetype-site	Un archétype pour un site Maven
maven-archetype-site-simple	Un archétype pour un site Maven simplifié
maven-archetype-webapp	Un archétype pour un projet de type application web

D'autres achétypes peuvent être proposés par des tiers. Il est aussi possible de développer ses propres archétypes.

101.3.2.5. La gestion des dépendances

Généralement un artéfact a besoin d'autres artéfacts qui sont alors désignés comme des dépendances présentant elles-mêmes des dépendances.

Une dépendance peut être optionnelle : elle n'est requise que si une fonctionnalité particulière est utilisée. C'est par exemple le cas pour Hibernate avec le pool de connections c3p0 ou l'implémentation du cache de second niveau.

Certaines dépendances ne sont utiles que pour certaines phases, par exemple lors de la phase de tests qui devrait être la seule à avoir besoin d'un framework pour les tests unitaires ou d'un framework pour le mocking.

Certains artéfacts possèdent un support de plusieurs implémentations : c'est par exemple le cas de commons-logging qui a besoin d'une implémentation d'un moteur de logging. Il va dynamiquement utiliser celui qui sera trouvé.

La gestion des dépendances de Maven repose sur plusieurs concepts :

- les dépôts : permet de stocker les artéfacts
- la portée : permet de préciser dans quel contexte une dépendance est utilisée
- la transitivité : permet de gérer les dépendances de dépendances
- l'héritage

101.3.2.6. Les dépôts (repositories)

Maven utilise la notion de référentiel ou dépôt (repository) pour stocker les dépendances et les plugins requis pour générer les projets.

Un dépôt contient un ensemble d'artéfacts qui peuvent être des livrables, des dépendances, des plugins, ... Ceci permet de centraliser ces éléments qui sont généralement utilisés dans plusieurs projets : c'est notamment le cas pour les plugins et les dépendances.

Maven distingue deux types de dépôts : local et distant (remote). Ces dépôts peuvent être gérés à plusieurs niveaux :

- dépôt central : il stocke des dépendances et les plugins utilisables par tout le monde car disponible sur le web; ce sont généralement des artéfacts open source
- dépôt local : il stocke une copie des dépendances et plugins requis par les projets à générer en local. Ces artéfacts sont soit téléchargés des dépôts centraux soit créés avec Maven
- un référentiel au niveau entreprise ou domaine : ce référentiel permet de limiter les dépendances et les plugins ainsi que leurs versions à celles qui sont utilisables en local. Il permet de gérer et de restreindre les dépendances pour un certain niveau (entreprise, domaine, service, équipe, projet ...). Son utilisation est requise lorsque le nombre de projets et leur complexité sont importants.

Maven utilise une structure de répertoires particulière pour organiser le contenu d'un référentiel et lui permettre de retrouver les éléments requis :

Chemin_referentiel/groupId/artifactId/version

Par exemple avec Junit 3.8.2, dont les identifiants sont :

```
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.2</version>
```

La structure de répertoires dans le dépôt est :

\junit\junit\3.8.2

Le répertoire de la version contient au moins l'artefact et son POM mais il peut aussi éventuellement contenir d'autres fichiers liés contenant une archive avec les sources, la Javadoc, la valeur du message digest calculée avec SHA-1, ...

Maven recherche un élément dans un ordre précis dans les différents référentiels :

- tout d'abord dans le dépôt local
- puis un des dépôts distant configuré s'il n'est pas trouvé

Si un élément n'est pas trouvé dans le répertoire local, il sera téléchargé dans ce dernier à partir du premier dépôt distant dans lequel il est trouvé.

Par défaut, le dépôt local est contenu dans le sous-répertoire .m2\repository du répertoire personnel de l'utilisateur.

Maven gère un référentiel central qui contient les artéfacts qui peuvent être diffusés. Il existe plusieurs miroirs du référentiel central.

Il existe aussi des référentiels proposés par des tiers notamment pour diffuser leurs propres artéfacts. Certains artéfacts ne peuvent pas être inclus dans des référentiels publics pour des raisons de licences.

Il est possible d'utiliser un ou plusieurs référentiels au niveau entreprise qui permettent de stocker ses propres artéfacts ou les artéfacts dont la licence interdit le diffusion publique.

L'accès aux référentiels peut se faire en utilisant plusieurs protocoles : http, https, ftp, sftp, WebDAV, SCP, ...

101.3.2.7. La portée des dépendances

Toutes les dépendances ne doivent pas forcément être utilisées de la même manière dans le processus de build ou lors de l'exécution de l'artéfact.

Maven utilise la notion de portée (scope) pour préciser comment la dépendance sera utilisée.

Maven définit quatre portées pour les dépendances :

- compile : la dépendance est utilisable par toutes les phases et à l'exécution. C'est le scope par défaut
- provided : la dépendance est utilisée pour la compilation mais elle ne sera pas déployée car elle est considérée comme étant fournie par l'environnement d'exécution. C'est par exemple le cas des API fournies par un serveur d'applications
- runtime : la dépendance n'est pas utile pour la compilation mais elle est nécessaire à l'exécution. C'est par exemple le cas des pilotes JDBC
- test : la dépendance n'est utilisée que lors de la compilation et de l'exécution des tests. C'est par exemple le cas pour la bibliothèque utilisée pour les tests unitaires (JUnit ou TestNG pas exemple) ou pour les doublures (Easymock, Mockito, ...)

101.3.2.8. La transitivité des dépendances

Maven 2 prend en charge la gestion des dépendances transitives. Les dépendances transitives sont des dépendances requises par un artéfact, les dépendances de ces dépendances et ainsi de suite.

Exemple :

Exemple :

```
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate</artifactId>
    <version>3.3.2</version>
    <scope>compile</scope>
  </dependency>
```

Cette déclaration de dépendances indique que l'artéfact a besoin d'Hibernate. Les dépendances d'Hibernate seront déterminées par Maven en analysant son POM puis les POM des dépendances et ainsi de suite.

Les dépendances transitives engendrent parfois de petits soucis qu'il convient de régler. C'est notamment le cas lors de l'inclusion d'une version différente de celle souhaitée car cette version est obtenue par une ou plusieurs dépendances transitives ou l'inclusion de dépendances non utilisées.

La qualité des dépendances transitives est grandement liée à la déclaration des dépendances dans les fichiers POM des différents artéfacts concernés.

Pour supprimer une dépendance transitive, il est possible de déclarer l'artéfact concerné avec le scope «provided» ou de définir une exclusion dans la définition de la dépendance.

101.3.2.9. La communication sur le projet

Maven propose de générer le contenu d'un site web dont le but est de fournir des informations sur le projet.

Il est facile possible d'inclure dans le site :

- les informations générales sur le projet
- le rapport sur l'exécution des tests unitaires et de la couverture de tests
- le rapport sur l'exécution des outils de qualité statique de code (PMD, Checkstyle, ...)
- les dépendances du projet
- la Javadoc
- ...

101.3.2.10. Les plugins

Maven en lui-même n'est composé que d'un noyau très léger.

Toutes les fonctionnalités pour générer un projet sont sous la forme de plugins qui doivent être présents dans le référentiel local ou téléchargés lors de la première utilisation.

La déclaration et la configuration des plugins à utiliser se fait dans le fichier POM.

Certains plugins utilisés dans le cycle de vie par défaut n'ont pas besoin d'être définit explicitement dans le fichier POM, sauf si la configuration par défaut doit être modifiée.

Il est aussi possible de développer ses propres plugins.

Un plugin est un artéfact Maven : il est donc identifié par un groupId, un artifactId et un numéro de version (par défaut, la version la plus récente).

La personnalisation d'un projet se fait en utilisant et en configurant des plugins.

Exemple pour préciser au compilateur d'utiliser la version 1.5 de Java

Exemple :

```
...
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.0.2</version>
      <configuration>
        <source>1.5</source>
        <target>1.5</target>
      </configuration>
    </plugin>
  </plugins>
</build>
...
```

Les plugins sont des dépendances qui sont stockées dans le dépôt local après avoir été téléchargées.

Le tag <configuration> permet de fournir des paramètres qui seront utilisés par le plugin.

Le plugin Maven help possède un goal «describe» qui permet d'afficher des informations sur un plugin.

Le plugin dont on souhaite obtenir la description doit être précisé grâce à la propriété plugin fournie à la JVM.

La propriété medium fournie à la JVM permet de demander un niveau d'information supplémentaire qui offre une description de chaque goal.

Résultat :

```
C:\>mvn help:describe -Dplugin=help
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'help'.
[INFO]
------------------------------------------------------------------------
[INFO] Building Maven Default Project
[INFO]    task-segment: [help:describe] (aggregator-style)
[INFO]
------------------------------------------------------------------------
[INFO] [help:describe {execution: default-cli}]
[INFO] org.apache.maven.plugins:maven-help-plugin:2.1.1

Name: Maven Help Plugin
Description: The Maven Help plugin
provides goals aimed at helping to make
  sense out of the build environment. It includes the ability to view the
  effective POM and settings files, after inheritance and active profiles
  have been applied, as well as a describe a particular plugin goal to
  give usage information.
Group Id: org.apache.maven.plugins
Artifact Id: maven-help-plugin
Version: 2.1.1
Goal Prefix: help

This plugin has 9 goals:
help:active-profiles
  Description: Displays a list of the profiles which are currently active
  for this build.

help:all-profiles
  Description: Displays a list of available profiles under the current
  project.
  Note: it will list all profiles for a project. If a profile comes up
  with a status inactive then there might be a need to set profile activation
  switches/property.

help:describe
  Description: Displays a list of the attributes for a Maven Plugin and/or
  goals (aka Mojo - Maven plain Old Java Object).

help:effective-pom
  Description: Displays the effective POM as an XML for this build, with
  the active profiles factored in.

help:effective-settings
  Description: Displays the calculated settings as XML for this project,
  given any profile enhancement and the inheritance of the global settings
  into the user-level settings.

help:evaluate
  Description: Evaluates Maven expressions given by the user in an
  interactive mode.

help:expressions
  Description: Displays the supported Plugin expressions used by Maven.

help:help
  Description: Display help information on maven-help-plugin.
   Call
      mvn help:help -Ddetail=true -Dgoal=<goal-name>
   to display parameter details.

help:system
  Description: Displays a list of the platform details like system
  properties and environment variables.

For more information, run 'mvn help:describe [...] -Ddetail'
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
```

```
[INFO]
------------------------------------------------------------------------
[INFO] Total time: 2 seconds
[INFO] Finished at: Tue Nov 25 08:22:23 CET 2012
[INFO] Final Memory: 6M/510M
[INFO]
------------------------------------------------------------------------
```

La valeur par défaut de la propriété medium est true : elle affiche une description de chacun des goals du plugin. La valeur false affiche simplement une description du plugin.

Résultat :

```
C:\>mvn help:describe -Dplugin=help -Dmedium=false
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'help'.
[INFO]
------------------------------------------------------------------------
[INFO] Building Maven Default Project
[INFO]    task-segment: [help:describe] (aggregator-style)
[INFO]
------------------------------------------------------------------------
[INFO] [help:describe {execution: default-cli}]
[INFO] org.apache.maven.plugins:maven-help-plugin:2.1.1
Name: Maven Help Plugin
Description: The Maven Help plugin
provides goals aimed at helping to make
 sense out of the build environment. It includes the ability to view the
 effective POM and settings files, after inheritance and active profiles
 have been applied, as well as a describe a particular plugin goal to give
 usage information.

Group Id: org.apache.maven.plugins
Artifact Id: maven-help-plugin
Version: 2.1.1
Goal Prefix: help
For more information, run 'mvn
help:describe [...] -Ddetail'
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO]
------------------------------------------------------------------------
[INFO] Total time: < 1 second
[INFO] Finished at: Tue Nov 25 08:27:10 CET 2012
[INFO] Final Memory: 6M/510M
[INFO]
------------------------------------------------------------------------
```

La propriété full fournie à la JVM permet de demander un niveau d'information complet qui offre en plus la liste des paramètres de chaque goal.

Résultat :

```
C:\>mvn help:describe -Dplugin=help -Dfull=true
```

Si le plugin n'est pas dans le dépôt local, il est nécessaire de fournir toutes les informations le concernant (nom, version)

Résultat :

```
C:\>mvn help:describe -Dplugin=org.apache.maven.plugins:maven-help-plugin:2.1.1
-Dmedium=false
```

La configuration d'un plugin se fait dans le fichier POM dans le tag <project><build><pluginManagement><plugins><plugin><executions><execution><configuration> pour toutes les exécutions ou <project><build><plugins><plugin><executions><execution><configuration> pour une exécution particulière.

101.3.3. La création d'un nouveau projet

La création d'un nouveau projet Maven repose sur l'utilisation d'un archétype et du goal archetype:generate en utilisant la commande :

mvn archetype:generate options

Résultat :

```
mvn archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes
-DarchetypeArtifactId=maven-archetype-quickstart
-DgroupId=fr.jmdoudoux.dej.monapp -DartifactId=monApplication -DinteractiveMode=false
```

Cette commande va créer un répertoire pour contenir le projet et un ensemble de sous-répertoires et de fichiers selon l'archétype utilisé. Parmi les fichiers générés, il y a notamment le fichier pom.xml.

Les sous-répertoires créés suivent la structure standard des répertoires d'un projet qui est définie par convention.

Le sous-répertoire src/main/java contient les sources du projet.

Le sous-répertoire src/test/java contient les sources des tests unitaires du projet.

Un archétype est packagé dans une archive de type jar qui contient des métadonnées et un ensemble de template Velocity.

Lors de la création du premier archétype, Maven va télécharger dans son dépôt local tous les plugins qui lui sont nécessaires.

Un archétype est un template de projet qui est combiné avec certaines informations fournies pour créer un projet Maven d'un type particulier.

Les informations sont fournies sous la forme de propriétés fournies à la JVM -Dpropriete=valeur

Il est préférable d'utiliser le goal generate plutôt que l'ancien goal create.

Les archétypes sont eux même des artéfacts : ils seront téléchargés d'un référentiel distant.

101.3.4. Le fichier POM

Le fichier POM (Project Object Model) contient la description du projet Maven. C'est un fichier XML nommé pom.xml. Il contient les informations nécessaires à la génération du projet : identification de l'artéfact, déclaration des dépendances, définition d'informations relatives au projet, ...

Le fichier POM doit être à la racine du répertoire du projet.

Le tag racine est le tag <project>

Exemple :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.jmdoudoux.dej</groupId>
  <artifactId>MaWebApp</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Mon application web</name>
  <dependencies>
     <dependency>
       <groupId>junit</groupId>
       <artifactId>junit</artifactId>
       <version>4.7</version>
       <scope>test</scope>
     </dependency>
   </dependencies>
</project>
```

Le tag <projet> peut avoir plusieurs tags fils dont les principaux sont :

Tag	Rôle
<modelVersion>	Préciser la version du modèle de POM utilisée
<groupId>	Préciser le groupe ou l'organisation qui développe le projet. C'est une des clés utilisée pour identifier de manière unique le projet et ainsi éviter les conflits de noms
<artifactId>	Préciser la base du nom de l'artéfact du projet
<packaging>	Préciser le type d'artéfact généré par le projet (jar, war, ear, pom, ...). Le packaging définit aussi les différents goals qui seront exécutés durant le cycle de vie par défaut du projet. La valeur par défaut est jar
<version>	Préciser la version de l'artéfact généré par le projet. Le suffixe -SNAPSHOT indique une version en cours de développement
<name>	Préciser le nom du projet utilisé pour l'affichage
<description>	Préciser une description du projet
<url>	Préciser une url qui permet d'obtenir des informations sur le projet
<dependencies>	Définir l'ensemble des dépendances du projet
<dependency>	Déclarer une dépendance en utilisant plusieurs tags fils : <groupId>, <artifactId>, <version> et <scope>

Les informations d'un POM sont héritées d'un POM parent sauf le POM racine.

101.3.4.1. L'utilisation et la configuration des plugins

Maven possède de très nombreux plugins fournis par Maven ou par des tiers. Il est aussi possible d'écrire ses propres plugins.

Certains plugins sont utilisés par défaut par un goal : il est alors possible de changer certains de leurs paramètres d'exécution dans la section plugins.

Exemple :

```
 ...

    <build>
      ...
      <plugins>
         <plugin>
           <groupId>org.apache.maven.plugins</groupId>
           <artifactId>maven-compiler-plugin</artifactId>
             <configuration>
```

```
               <source>1.5</source>
               <target>1.5</target>
            </configuration>
         </plugin>
      </plugins>
   </build>
...
```

L'exemple ci-dessus permet de préciser au plugin de compilation d'utiliser la version 5 de Java.

Pour utiliser un plugin, il faut l'associer à un cycle de vie en précisant la phase concernée dans un tag executions/execution/phase et le goal dans un tag executions/execution/goals/goal

Exemple :

```
...
  <build>
    ...
    <plugins>
      <plugin>
      <artifactId>maven-antrun-plugin</artifactId>
      <executions>
            <execution>
              <phase>generate-sources</phase>
              <configuration>
                <tasks>
                  <!-tache Ant-->
                </tasks>
              </configuration>
              <goals>
                <goal>run</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
     </plugins>
  </build>
...
```

L'exemple ci-dessus exécute une tâche Ant en utilisant le plugin maven-antrun-plugin dans le goal run de la phase generate-sources.

101.3.4.2. Les métadonnées du projet

Il est possible d'ajouter des métadonnées dans le fichier POM : celles-ci sont essentiellement utilisées pour la génération du site de documentation du projet mais certains autres plugins peuvent aussi les utiliser pour des besoins spécifiques.

Exemple :

```
<project xmlns=http://maven.apache.org/POM/4.0.0
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xsi:schemaLocation= http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd>
  <modelVersion>4.0.0</modelVersion>
  [...]
  <name>Nom du projet</name>
  <description>Description du projet</description>
  <inceptionYear>2012</inceptionYear>
  <organization>
    <name>Nom de l'entreprise</name>
    <url>Url de l'entreprise</url>
  </organization>
  <licenses>
    <license>
      <name>Nom de la licence</name>
      <url>L'url de la licence</url>
      <comments>Commentaire concernant la licence</comments>
      <distribution>Politique de redistribution</distribution>
    </license>
  </licenses>
  <url>Url du projet</url>

  <version>1.3.0-1.2-MAVEN2TEST</version>

  <developers>
    <developer>
      <name>Jean-Michel Doudoux</name>
      <id>jmdoudoux</id>
      <email>jean-michel.doudoux@wanadoo.fr</email>
      <timezone>+1</timezone>
      <organization>Mon entreprise</organization>
      <organizationUrl>http://www.jmdoudoux.fr/</organizationUrl>
      <roles>
        <role>Tech lead</role>
        <role>développeur</role>
      </roles>
    </developer>
    ...
  </developers>
  <contributors>
    <contributor>
      <name>John Doe</name>
      <email>john_doe@inconnu.fr</email>
      <timezone>+1</timezone>
      <organization>Autre entreprise</organization>
      <organizationUrl>http://www.autreentreprise.fr/</organizationUrl>
      <roles>
        <role>Développeur</role>
      </roles>
    </contributor>
    [...]
  </contributors>
...
</project>
```

101.3.5. Le cycle de vie d'un projet

La construction d'un projet Maven repose sur la notion de cycle de vie qui définit de manière claire les différentes étapes nommées phases.

Trois cycles de vie sont définis :

- default : il permet de générer et déployer l'artéfact du projet
- clean : il permet de nettoyer les fichiers générés du projet
- site : il permet de générer un site web pour accéder à la documentation du projet

Un cycle de vie est composé de phases qui constituent les étapes de la génération de l'artéfact.

Le cycle de vie par défaut de Maven (build) propose plusieurs phases par défaut :

Phase	Rôle
validate	Valider les informations nécessaires à la génération du projet
initialize	Initialiser la génération du projet
generate-sources	Générer les sources qui doivent être incluses dans la compilation
process-sources	Traiter des sources (par exemple, application d'un filtre)
generate-resources	Générer les ressources qui doivent être incluses dans l'artéfact
process-resources	Copier les ressources dans le répertoire target

compile	Compiler le code source du projet
process-classes	Traiter les fichiers class résultant de la compilation (par exemple, pour faire du bytecode enhancement)
generate-test-sources	Générer des sources qui doivent être incluses dans les tests
process-test-sources	Traiter les sources pour les tests (par exemple, application d'un filtre)
generate-test-resources	Générer des ressources qui doivent être incluses dans les tests
process-test-resources	Copier les ressources dans le répertoire test
test-compile	Compiler le code source des tests
process-test-classes	Effectuer des actions sur les classes compilées (par exemple, pour faire du bytecode enhancement)
test	Compiler et exécuter les tests unitaires automatisés. Ces tests ne doivent pas avoir besoin de la forme diffusable de l'artéfact ni de son déploiement dans un environnement
Prepare-package (à partir de Maven 2.1)	Effectuer des actions pour préparer la génération du package
package	Générer l'artéfact sous sa forme diffusable (jar, war, ear, ...)
pre-integration-test	Effectuer des actions avant l'exécution des tests d'intégration (par exemple configurer l'environnement d'exécution)
integration-test	
(à partir de Maven 3)	Compiler et exécuter les tests d'intégration automatisés dans un environnement dédié au besoin en effectuant un déploiement
post-integration-test	Effectuer des actions après l'exécution des tests d'intégration (par exemple faire du nettoyage dans l'environnement)
verify	Exécuter des contrôles de validation et de qualité
install	Installer l'artéfact dans le dépôt local pour qu'il puisse être utilisé comme dépendance d'autres projets
deploy	Déployer l'artéfact dans un environnement dédié et copier de l'artéfact dans le référentiel distant

Ces différentes phases sont exécutées séquentiellement de la première étape jusqu'à la phase demandée à Maven. Toutes les phases jusqu'à la phase demandée seront exécutées.

Exemple :

```
mvn deploy
mvn integration-test
```

Maven définit la notion de multi-modules pour permettre de définir un projet qui possède un ou plusieurs sous-projets.

Une phase est composée d'un ou plusieurs goals. Un goal est une tâche spécifique à la génération ou la gestion du projet. Un goal peut être déclaré dans une ou plusieurs phases.

Il est possible d'invoquer directement un goal même s'il n'est pas associé à une phase.

```
mvn dependency:copy-dependencies
```

Il est possible d'invoquer des phases et des goals : leur ordre d'exécution sera celui fournit en paramètre de la commande mvn

Exemple :

```
mvn clean dependency:copy-dependencies package
```

Dans l'exemple ci-dessus, le cycle de vie clean est exécutée, puis le goal dependency :copy-dependencies puis la phase package avec toutes les phases précédentes du cycle de vie par défaut.

Si le goal est définit dans plusieurs phases alors le goal de chacune des phases sera invoqué.

Lors de l'invocation de la demande de l'exécution d'une phase à Maven, Maven va exécuter toutes les phases précédentes du cycle de vie.

Les goals exécutés par chaque phases dépendent du packaging de l'artéfact à générer. Le packaging est précisé grâce à l'élément <packaging> du POM.

Le cycle de vie clean de Maven contient plusieurs phases :

Phase	Rôle
pre-clean	Exécuter des traitements avant de nettoyer le projet
clean	Supprimer tous les fichiers par le build précédent
post-clean	Exécuter des traitements pour terminer le nettoyage du projet

Le cycle de vie site de Maven contient plusieurs phases :

Phase	Rôle
pre-site	Exécuter des traitements avant la génération du site
site	Générer le contenu du site de documentation du projet
post-site	Exécuter des traitements après la génération du site
site-deploy	Déployer le site sur un serveur web

Il est fréquemment reproché à Maven de télécharger beaucoup de fichiers à partir de dépôts distant mais Maven doit obtenir toutes les dépendances des projets, les plugins et les dépendances de ces plugins.

Tous ces éléments téléchargés sont stockés dans le dépôt local pour n'avoir à faire cette opération de téléchargement qu'une seule fois tant que l'élément est contenu dans le dépôt local.

101.3.5.1. L'ajout d'un goal à une phase

Il est possible d'ajouter de nouveaux goals à exécuter dans une phase, notamment pour permettre la configuration et l'utilisation de plugins.

Un plugin peut proposer un ou plusieurs goals. Généralement, la documentation d'un plugin doit préciser dans quelle phase un goal peut être utilisé.

La configuration d'un goal s'effectue avec un tag <execution> fils des tags <plugins>, <plugin> et <executions>. Les goals configurés viennent s'ajouter aux goals définis par défaut dans la phase concernée.

Il est possible de demander l'exécution du goal plusieurs fois avec des configurations différentes.

Pour chaque exécution, il est possible :

- de définir un identifiant avec le tag <id>.
- de préciser la phase d'exécution avec le tag <phase>
- de préciser les goals à exécuter avec le tag <goals> qui contient autant de tag fils <goal> que nécessaire
- le tag <configuration> permet de configurer le goal

Il est possible d'ajouter des goals à une phase du cycle de vie pour permettre de personnaliser les traitements exécutés lors de la génération du projet Maven.

L'exemple ci-dessous demande l'exécution du goal antrun:run qui devra s'exécuter dans la phase compile du cycle de vie du projet.

Exemple :

```
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-antrun-plugin</artifactId>
        <version>1.1</version>
        <executions>
          <execution>
            <id>id.compile</id>
            <phase>compile</phase>
            <goals>
              <goal>run</goal>
            </goals>
            <configuration>
              <tasks>
                <echo>Phase de compilation</echo>
              </tasks>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
```

101.3.6. La configuration générale de Maven

La configuration générale de Maven peut se faire à différents niveaux :

- global à Maven dans le fichier settings.xml du sous-répertoire conf de l'installation de Maven
- au niveau de l'utilisateur : dans le fichier settings.xml du sous-répertoire .m2 dans le répertoire home de l'utilisateur

Il est possible de préciser un autre fichier de configuration au niveau de l'utilisateur en utilisant l'option -s et au niveau global en utilisant l'option -gs.

Le goal help:effective-settings permet d'avoir un affichage des settings du projet. Les informations affichées sont une combinaison du contenu du fichier des settings.xml global et spécifique à l'utilisateur.

Résultat :

```
mvn help:effective-settings
```

101.3.6.1. Le fichier settings.xml

Le fichier settings.xml contient la configuration globale de Maven. C'est un document XML dont le tag racine est le tag <settings>.

Plusieurs tags fils permettent de préciser les éléments de la configuration :

Tag	Rôle
localRepository	Préciser le chemin du dépôt local. Par défaut, c'est le sous-répertoire .m2/repository du répertoire de l'utilisateur
interactiveMode	Utiliser Maven en mode interactif pour lui permettre d'obtenir des informations lorsqu'elles sont requises. Par défaut : true.

offline	Utiliser Maven en mode offline, donc déconnecter du réseau. Par défaut : false.
pluginGroups	
proxies	Définir un ou plusieurs proxy. Chaque proxy est défini dans un tag <proxy>
servers	Fournir des informations d'authentification sur une ressource. Chaque ressource est définie dans un tag <server>
mirrors	Définir des dépôts distants qui sont des miroirs. Chaque miroir est défini dans un tag <mirror>
profiles	Définir des profiles qui pourront être activés. Chaque profile est défini dans un tag <profile>
ActiveProfiles	Définir la liste des profiles qui sont activés par défaut pour tous les builds

Dans le fichier de configuration .m2/settings.xml, les informations d'authentification sur des serveurs sont définies dans le tags <servers>. Chaque serveur possède un tag fils <server> qui possède plusieurs tags fils :

Tag	Rôle
<id>	identifiant du serveur
<user>	nom de l'utilisateur
<password>	mot de passe de l'utilisateur

La configuration des dépôts utilisables se fait dans un tag <mirrors>. Chaque dépôt est défini dans un tag <mirror> qui possède plusieurs tags fils :

Tag	Rôle
<id>	identifiant du dépôt
<name>	nom du dépôt
<url>	url du dépôt
<mirrorOf>	

Exemple :

```
        <mirror>
            <id>artifactory</id>
            <name>enterprise repository</name>
            <url>http://dev-server/artifactory/libs-releases/</url>
            <mirrorOf>central</mirrorOf>
        </mirror>
```

La configuration de proxys se fait dans un tag <proxies>. Chaque proxy est défini dans un tag <proxy> qui possède plusieurs tags fils :

Tag	Rôle
<id>	L'identifiant du proxy
<protocol>	Le protocole utilisé pour accéder au proxy
<active>	
<username>	Le nom de l'utilisateur
<password>	Le mot de passe de l'utilisateur
<host>	Le host name du proxy
<port>	Le port du proxy

<nonProxyHosts>	énbsp;

Exemple :

```
<proxies>
  <proxy>
    <id>proxy</id>
    <active>yes</active>
    <protocol>http</protocol>
    <username>xxx</username>
    <password>yyy</password>
    <host>proxy-web</host>
    <port>8080</port>
    <nonProxyHosts>localhost</nonProxyHosts>
  </proxy>
</proxies>
```

Dans un dépôt, les artéfacts sont stockés en utilisant les identifiants de l'artéfact.

Une structure de répertoires est utilisée pour organiser les artéfacts en utilisant le groupId de l'artéfact de manière similaire à celle utilisée par les packages Java.

Il est donc intéressant d'utiliser un nom de domaine inversé comme groupId.

Exemple

fr.jmdoudouxmonapp

L'arborescence de l'artéfact contient aussi un sous-répertoire ayant pour nom l'artifactId.

Enfin un sous-répertoire est aussi créé pour chaque version : ce sous-répertoire contient l'artéfact et son POM. D'autres fichiers peuvent aussi être présents selon la configuration du projet (jar contenant les sources, javadoc, ...)

Le goal install-file du plugin maven-install permet d'ajouter un artéfact dans le dépôt local.

Résultat :

```
mvn install:install-file -DgroupId=group-id \
                         -DartifactId=artifact-id \
                         -Dversion=version \
                         -Dpackaging=packaging \
                         -Dfile=fichierAinstaller
```

Cette commande ajoute un nouvel artéfact dont le fichier est précisé par la propriété file dans une arborescence utilisant la valeur des propriétés groupId, artifactId et version.

Cette solution est pratique pour faire un test ou s'il n'y qu'un seul développeur sur le projet. Dans le cas contraire, il est plus intéressant d'utiliser un gestionnaire de dépôts dans lequel il faut ajouter l'artéfact.

101.3.6.2. L'utilisation d'un miroir du dépôt central

Le dépôt central de Maven par défaut est à l'url https://repo.maven.apache.org/maven2/

Il peut être utile d'utiliser un autre dépôt pour plusieurs raisons : il est plus prêt géographiquement, il possède des artéfacts qui ne sont pas disponibles dans le dépôt par défaut, ...

Des miroirs publics du dépôt par défaut sont listés dans un lien de la page
https://maven.apache.org/guides/mini/guide-mirror-settings.html.

Il est aussi possible de créer un dépôt intermédiaire gérer par un gestionnaire d'artéfacts qui va contenir une copie des artéfacts requis. L'utilisation d'un gestionnaire d'artéfacts possède plusieurs avantages :

- limiter l'utilisation de la bande passante sur internet et ainsi améliorer les performances
- restreindre les artéfacts utilisés uniquement à ceux présents dans le dépôt

Pour remplacer ou ajouter un autre dépôt, il faut modifier le fichier de configuration settings.xml

Exemple :

```
<settings>
  ...
  <mirrors>
    <mirror>
      <id>ibiblio.org</id>
      <url>http://mirrors.ibiblio.org/pub/mirrors/maven2</url>
    </mirror>
  </mirrors>
  ...
</settings>
```

Chaque dépôt est défini dans un tag <mirror>.

Exemple :

```
<settings>
  ...
  <mirrors>
    <mirror>
      <id>UK</id>
      <name>UK Central</name>
      <url>http://uk.maven.org/maven2</url>
      <mirrorOf>central</mirrorOf>
    </mirror>
  </mirrors>
  ...
</settings>
```

La valeur central du tag <mirrorOf> permet de préciser que le dépôt est un miroir du dépôt central. Maven va alors l'utiliser de préférence au dépôt central.

101.3.7. La configuration du projet

La configuration du projet se fait dans le fichier POM du projet.

Comme Maven utilise l'héritage de fichiers POM, le goal help:effective-pom permet d'avoir un affichage du POM effectif du projet. Les informations affichées sont une combinaison du contenu du fichier POM, des POM parents et des profiles qui sont actifs.

Résultat :

```
mvn help:effective-pom
```

101.3.7.1. La déclaration des dépendances

Le tag <dependencies> du fichier POM permet de déclarer les dépendances externes du projet que ce soit pour la compilation, les tests ou l'exécution.

Pour chaque dépendance, il est nécessaire de préciser le groupId, l'artifactId, la version et le scope. Les trois premières informations doivent correspondre à celles définies dans le POM de la dépendance.

Exemple :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.jmdoudouxapp</groupId>
  <artifactId>monapp</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Le scope permet de préciser dans quel cadre la dépendance va être utilisée lors de la génération du projet :

- compile : utilisée pour la compilation et l'exécution (elle sera incluse dans le livrable)
- test : utilisée pour la compilation et l'exécution des tests uniquement (ne sera pas incluse dans le livrable)
- runtime : utilisée pour l'exécution uniquement (ce sont généralement des bibliothèques)
- provided : utilisée pour la compilation mais la dépendance sera fournie par l'environnement d'exécution (elle ne sera pas incluse dans le livrable)

Maven recherche les dépendances dans la hiérarchie des dépôts en commençant par le dépôt local. Si elle n'est pas trouvée, elle est téléchargée d'un autre dépôt.

Il est possible d'ajouter un nouvel artéfact directement dans le dépôt local : il est nécessaire de créer au préalable le fichier POM correspondant. Ceci est nécessaire notamment pour les artéfacts fournis par des tiers.

Pour ajouter une dépendance, il est nécessaire de connaître le groupId, l'artifactId et la version de l'artéfact correspondant. Il existe plusieurs sites pour permettre de rechercher ces informations, par exemple :

https://search.maven.org/

https://mvnrepository.com/

https://www.ibiblio.org

Une fois les informations identifiant l'artéfact connues, il faut l'ajouter en utilisant un tag <dependency> fils du tag <dependencies>

Exemple :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>fr.jmdoudouxapp</groupId>
  <artifactId>monapp</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
        <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.17</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>
</project>
```

La gestion des dépendances transitives est une fonctionnalité intéressante de Maven. Elle permet à Maven de gérer lui-même les dépendances d'une dépendance.

Maven assure aussi la gestion des dépendances transitives en double.

La prise en charge des dépendances transitives par Maven facilite grandement le travail de gestion des dépendances. Ce mécanisme fonctionne cependant bien si les dépendances sont correctement configurées dans les fichiers pom.xml des artéfacts dépendants.

101.3.7.2. L'utilisation de profiles

Les profiles permettent de définir différentes configurations.

Il est possible de définir le profile par défaut dans le fichier de configuration settings.xml en précisant le nom du profile dans un tag <activeProfile> fils du tag <activeProfiles>.

Exemple :

```
...
      <profiles>
...
            <profile>
                  <id>dev</id>
                  <properties>
                        <ma.propriete>Ma valeur de dev</ma.propriete>
                  </properties>
            </profile>
            <profile>
                  <id>prod</id>
                  <properties>
                        <ma.propriete>Ma valeur de prod</ma.propriete>
                  </properties>
            </profile>
...
      </profiles>
...
      <activeProfiles>
            <activeProfile>dev</activeProfile>
      </activeProfiles>
...
```

Pour afficher le profile actif, il faut utiliser le goal help:active-profiles

Maven permet d'activer un profile selon la version du JDK utilisé. Dans le fichier settings.xml, il faut ajouter un tag <jdk> fils des tags profile/activation qui précise la version de Java concernée.

Exemple :

```
...

<profile>

<id>profile.java.5</id>
      <properties>
```

```
        <ma.propriete>Ma valeur pour Java 5</ma.propriete>
      </properties>
      <activation>
        <jdk>1.5</jdk>
      </activation>
    </profile>
    <profile>
      <id>profile.java.6</id>
      <properties>
        <ma.propriete>Ma valeur pour Java 6</ma.propriete>
      </properties>
      <activation>
        <jdk>1.6</jdk>
      </activation>
    </profile>
...
```

Les profiles peuvent être définis dans trois fichiers :

- settings.xml de la configuration globale (dans le sous-répertoire conf du répertoire d'installation de Maven) ou de la configuration de l'utilisateur (dans le sous-répertoire .m2 du répertoire home de l'utilisateur)
- pom.xml
- profiles.xml à la racine du projet

Exemple extrait du fichier settings.xml

Exemple :

```
...
  <profiles>
...
    <profile>
      <id>dev</id>
      <properties>
        <ma.propriete>Ma valeur de dev</ma.propriete>
      </properties>
    </profile>
    <profile>
      <id>prod</id>
      <properties>
        <ma.propriete>Ma valeur de prod</ma.propriete>
      </properties>
    </profile>
...
```

Exemple dans un fichier pom.xml

Exemple :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.jmdoudoux.dej.monapp</groupId>
  <artifactId>monApplication</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>monApplication</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <profiles>
    <profile>
      <id>dev</id>
      <properties>
        <ma.propriete>Ma valeur de dev</ma.propriete>
      </properties>
    </profile>
    <profile>
      <id>prod</id>
      <properties>
        <ma.propriete>Ma valeur de prod</ma.propriete>
      </properties>
    </profile>
  </profiles>
</project>
```

Il est possible de préciser un profile par défaut en utilisant la valeur true pour le tag <activeByDefault> fils du tag <activation>.

Exemple :

```
  <profiles>
    <profile>
      <id>dev</id>
      <activation>
        <activeByDefault>true</activeByDefault>
      </activation>
      <properties>
        <ma.propriete>Ma valeur de dev</ma.propriete>
      </properties>
    </profile>
  </profiles>
```

Pour connaitre le profile actif dans un projet, il faut invoquer le goal active-profiles du plugin help.

Résultat :

```
C:\JM\monApplication>mvn help:active-profiles
 [INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'help'.
[INFO]
------------------------------------------------------------------------
[INFO] Building monApplication
[INFO]    task-segment: [help:active-profiles] (aggregator-style)
[INFO]
------------------------------------------------------------------------
[INFO] [help:active-profiles {execution: default-cli}]
[INFO]
Active Profiles for Project 'fr.jmdoudoux.dej.monapp:monApplication:jar:1.0-SNAPSHOT':
The following profiles are active:  - dev (source: pom)
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: < 1 second
[INFO] Finished at: Thu Nov 25 08:35:26 CET 2012
[INFO] Final Memory: 6M/510M
[INFO]
------------------------------------------------------------------------
```

Il est possible d'activer un profile selon la présence d'une variable d'environnement de la JVM.

Le nom de cette propriété doit être précisé comme valeur du tag <name> fils du tag <activation><property>.

Exemple :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.jmdoudoux.dej.monapp</groupId>
  <artifactId>monApplication</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>monApplication</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <profiles>
    <profile>
      <id>dev</id>
      <activation>
        <activeByDefault>true</activeByDefault>
        <property>
          <name>profile.dev</name>
        </property>
      </activation>
      <properties>
        <ma.propriete>Ma valeur de dev</ma.propriete>
      </properties>
    </profile>
    <profile>
      <id>prod</id>
      <activation>
        <property>
          <name>profile.prod</name>
        </property>
      </activation>
      <properties>
        <ma.propriete>Ma valeur de prod</ma.propriete>
      </properties>
    </profile>
  </profiles>
</project>
```

Résultat :

```
C:\JM\monApplication>mvn -Dprofile.prod help:active-profiles
 [INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'help'.
[INFO]
------------------------------------------------------------------------
[INFO] Building monApplication
[INFO]    task-segment: [help:active-profiles] (aggregator-style)
[INFO]
------------------------------------------------------------------------
[INFO] [help:active-profiles {execution: default-cli}]
[INFO]
Active Profiles for Project 'fr.jmdoudoux.dej.monapp:monApplication:jar:1.0-SNAPSHOT':
The following profiles are active:  - prod (source: pom)
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO]
------------------------------------------------------------------------
[INFO] Total time: < 1 second
[INFO] Finished at: Thu Nov 25 08:43:52 CET 2012
[INFO] Final Memory: 6M/510M
[INFO]
------------------------------------------------------------------------
```

Il est possible :

- d'activer plusieurs profiles en définissant chacune des variables concernées.
- d'activer un profile selon la valeur d'une variable d'environnement de la JVM.

Le nom de la variable est précisé dans un tag <name> et la valeur dans un tag <value>.

Exemple :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.jmdoudoux.dej.monapp</groupId>
  <artifactId>monApplication</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>monApplication</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <profiles>
    <profile>
      <id>dev</id>
      <activation>
        <activeByDefault>true</activeByDefault>
        <property>
          <name>profile</name>
          <value>dev</value>
        </property>
      </activation>
      <properties>
        <ma.propriete>Ma valeur de dev</ma.propriete>
      </properties>
    </profile>
    <profile>
      <id>prod</id>
      <activation>
        <property>
          <name>profile</name>
          <value>prod</value>
        </property>
      </activation>
      <properties>
        <ma.propriete>Ma valeur de prod</ma.propriete>
      </properties>
    </profile>
  </profiles>
</project>
```

Résultat :

```
C:\JM\monApplication>mvn -Dprofile=prod help:active-profiles
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'help'.
[INFO]
------------------------------------------------------------------------
[INFO] Building monApplication
[INFO]    task-segment: [help:active-profiles] (aggregator-style)
[INFO]
------------------------------------------------------------------------
[INFO] [help:active-profiles {execution: default-cli}]
[INFO]
Active Profiles for Project 'fr.jmdoudoux.dej.monapp:monApplication:jar:1.0-SNAPSHOT':
```

```
The following profiles are active:  - prod (source: pom)
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO]
------------------------------------------------------------------------
[INFO] Total time: < 1 second
[INFO] Finished at: Thu Nov 25 08:51:08 CET 2012
[INFO] Final Memory: 6M/510M
[INFO]
------------------------------------------------------------------------
```

101.3.7.3. L'utilisation des propriétés

Dans le fichier POM, le tag <properties> permet de définir des propriétés. Chaque propriété est définie avec son propre tag. Le nom de la propriété correspond au nom du tag et la valeur est fournie dans le corps du tag.

Maven permet l'accès à une propriété en utilisant la syntaxe ${nom_de_la_propriete}.

Exemple :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.jmdoudoux.dej.monapp</groupId>
  <artifactId>monApplication</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>monApplication</name>
  <url>http://maven.apache.org</url>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <mapropriete>Message de test</mapropriete>
  </properties>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-antrun-plugin</artifactId>
        <version>1.1</version>
        <executions>
          <execution>
            <phase>validate</phase>
            <goals>
              <goal>run</goal>
            </goals>
            <configuration>
              <tasks>
                <echo>mapropriete = ${mapropriete}</echo>
              </tasks>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Maven considère les variables d'environnement comme des propriétés dont le nom est préfixé par «env.».

Par exemple, pour accéder à la variable M2_HOME dans le POM, il faut utiliser la syntaxe ${env.M2_HOME}

Les éléments du fichier POM peuvent être lus comme des propriétés. Le nom de ces propriétés commence par «project.»

Exemple : ${project.artifactId}

Les éléments du fichier settings peuvent être lus comme des propriétés. Le nom de ces propriétés commence par «settings.»

Exemple : ${settings.localRepository}

Les propriétés permettent aussi un accès aux variables d'environnement système et aux variables de la JVM.

Les propriétés peuvent être utilisées dans tout le POM.

Les propriétés peuvent aussi être utilisées dans les fichiers source (src/main/java) et les fichiers de ressources (src/main/resources). Les expressions correspondantes sont évaluées durant les phases process-sources et process-resources du cycle de vie par défaut de Maven.

101.3.7.4. L'ajout et l'exclusion de ressources dans l'artéfact

Il est fréquent de devoir ajouter des ressources dans un artéfact : fichiers de configuration, fichiers pour l'internationalisation, descripteurs, ...

Pour cela, Maven utilise sa structure standard de répertoires pour un projet : les ressources doivent être placées dans le sous-répertoire src/main/ressources de l'arborescence du projet. Tous les éléments (fichiers et sous-répertoires) contenus dans ce répertoire seront automatiquement ajoutés par Maven par le livrable de l'artéfact en conservant la sous-arborescence.

Résultat :

```
mon_app
|-- pom.xml
`-- src
    |-- main
        |-- java
        |   `-- com
        |       `-- jmdoudoux
        |           `-- app
        |               `-- MonApp.java
        `-- resources
            `-- META-INF
                `-- application.properties
```

Le livrable généré contient le sous-répertoire META-INF qui contient lui-même le fichier application.properties.

Résultat :

```
|-- META-INF
|   |-- MANIFEST.MF
|   |-- application.properties
|   `-- maven
|       `-- fr.jmdoudouxapp
|           `-- mon_app
|               |-- pom.properties
|               `-- pom.xml
`-- com
    `-- jmdoudoux
        `-- app
            `-- MonApp.class
```

Par défaut, lors de la génération d'un artéfact de type jar, Maven rajoute par défaut deux fichiers dans le sous-répertoire META-INF :

- pom.xml
- pom.properties

Les méta-données contenues dans le fichier pom.properties peuvent par exemple permettre d'obtenir le numéro de version de l'artéfact.

Résultat :

```
#Generated by Maven
#Wed May 09 14:15:24 CEST 2012
version=1.0-SNAPSHOT
groupId=fr.jmdoudouxmonapp
artifactId=mon_app
```

Maven créée un fichier META-INF/manifest par défaut si aucun n'est explicitement fourni dans les ressources du projet.

Résultat :

```
Manifest-Version: 1.0
Archiver-Version:
Plexus Archiver
Created-By: Apache Maven
Built-By: jm
Build-Jdk: 1.6.0_27
```

Il est aussi possible d'utiliser des ressources spécifiques aux tests unitaires. Pour ajouter ces ressources au classpath pour l'exécution des tests unitaires, il faut les mettre dans le sous-répertoire /src/test/ressources du répertoire du projet. Dans les tests unitaires, il est alors possible de lire une ressource en demandant un flux au classloader.

Il est possible d'exclure certaines ressources en les définissant dans le tag <ressource>.

Le tag fils <directory> permet de préciser le répertoire concerné.

Le tag <excludes> permet de préciser des motifs pour sélectionner les fichiers à exclure. Chaque motif est précisé dans un tag <exclude>.

Exemple :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.jmdoudoux.dej.monapp</groupId>
  <artifactId>monApplication</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>monApplication</name>
  <url>http://maven.apache.org</url>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <resources>
      <resource>
```

```
        <directory>src/main/resources</directory>
        <excludes>
          <exclude>**/*.txt</exclude>
        </excludes>
      </resource>
   </resources>
  </build>
</project>
```

101.3.7.5. La compilation d'un projet pour une version particulière de Java

Le maven-compiler-plugin permet de compiler le code source du projet et il est possible de le configurer pour utiliser une version particulière de Java.

Le tag <source> permet de préciser la version de Java avec laquelle le code source est compatible. Le tag <target> permet de préciser la version de Java qui sera utilisée pour générer le bytecode.

Exemple :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.jmdoudoux.dej</groupId>
  <artifactId>MaWebApp</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Mon application web</name>
  <dependencies>
      <dependency>
         <groupId>junit</groupId>
         <artifactId>junit</artifactId>
         <version>4.7</version>
         <scope>test</scope>
      </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.6</source>
          <target>1.6</target>
        </configuration>
      </plugin>
   </plugins>
 </build>
</project>
```

101.3.7.6. La gestion des versions des dépendances dans un POM parent

Le tag <dependencyManagement> d'un POM parent permet de gérer les numéros de versions des dépendances des projets fils.

Si une dépendance est définie dans la partie dependencyManagement d'un POM parent, alors les POM des projets fils n'ont pas l'obligation d'utiliser cette dépendance mais si c'est le cas, il n'est pas obligatoire de préciser la version de cette dépendance. Dans ce cas, c'est la version du POM parent qui sera utilisée.

Exemple :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.jmdoudoux.dej.maven</groupId>
  <artifactId>momapp</artifactId>
  <packaging>pom</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>monapp</name>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>commons-lang</groupId>
        <artifactId>commons-lang</artifactId>
        <version>2.3</version>
      </dependency>
    </dependencies>
  </dependencyManagement>
  <modules>
    <module>../monapp_service</module>
    <module>../monapp_persistance</module>
  </modules>
</project>
```

Dans le fichier POM d'un module, il n'est pas utile de préciser le numéro de version de la dépendance qui est déjà configuré dans la partie dependencyManagement du POM du projet parent.

Exemple :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">
  <parent>
    <groupId>fr.jmdoudoux.dej.maven</groupId>
    <artifactId>monapp_service</artifactId>
    <version>1.0-SNAPSHOT</version>
    <relativePath>../monapp/pom.xml</relativePath>
  </parent>

  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.jmdoudoux.dej.maven</groupId>
  <artifactId>monapp_service</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>monapp_service</name>
  <dependencies>
    <dependency>
      <groupId>commons-lang</groupId>
      <artifactId>commons-lang</artifactId>
      <scope>compile</scope>
    </dependency>
  </dependencies>
</project>
```

L'utilisation du dependencyManagement permet de centraliser les numéros de versions des dépendances à un endroit plutôt que d'avoir à répéter cette information dans plusieurs projets. Elle est particulièrement utile lors de l'utilisation de projets multi-modules.

101.3.7.7. La définition d'un fichier manifest particulier dans un jar

Par défaut, le fichier src/main/resources/META-INF/MANIFEST.MF n'est pas inclus dans l'artéfact de type jar généré.

Pour assurer sa prise en compte, il faut configurer le plugin maven-jar-plugin dans le fichier POM.

Exemple :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.jmdoudoux.dej.monapp</groupId>
  <artifactId>monApplication</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>monApplication</name>
  <url>http://maven.apache.org</url>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <mapropriete>Message de test</mapropriete>
  </properties>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
     <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <configuration>
          <archive>
            <manifestFile>src/main/resources/META-INF/MANIFEST.MF</manifestFile>
          </archive>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Le fichier ne doit contenir que les propriétés spécifiques à l'artéfact.

Résultat :

```
MaPropriete: MaValeur
```

Le fichier META-INF/MANIFEST.MF dans le fichier jar généré contient la partie générée par défaut et le contenu du fichier MANIFEST.MF contenu dans le sous-répertoire resources.

Résultat :

```
Manifest-Version: 1.0
Archiver-Version:
Plexus Archiver
Created-By: Apache Maven
Built-By: JM
Build-Jdk: 1.6.0_34
MaPropriete: MaValeur
```

Pour définir la classe principale à exécuter dans le fichier manifest, il faut préciser la classe dans un tag fils <mainClass> dans la configuration du plugin maven-jar-plugin.

Exemple :

```
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <configuration>
          <archive>
            <manifest>
              <mainClass>fr.jmdoudoux.dej.monapp.App</mainClass>
            </manifest>
          </archive>
        </configuration>
      </plugin>
```

Lors de la génération de l'artéfact, la classe principale sera précisée dans le fichier manifest.

Résultat :

```
Manifest-Version: 1.0
Archiver-Version:
Plexus Archiver
Created-By: Apache Maven
Built-By: JM
Build-Jdk: 1.6.0_34
Main-Class: fr.jmdoudoux.dej.monapp.App
```

101.3.7.8. L'utilisation de valeurs de propriétés dans les ressources

Les fichiers de ressources peuvent contenir des références sur des propriétés en utilisant la syntaxe ${...}.

Exemple : le fichier /src/main/resources/test.txt

Résultat :

```
artifactId: ${project.artifactId}
```

Par défaut, lors de la génération Maven ne fait pas la résolution des valeurs de la propriété. Il est possible de demander l'application d'un filtre qui va réaliser cette résolution en utilisant le tag <filtering> avec la valeur true.

Exemple :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.jmdoudoux.dej.monapp</groupId>
  <artifactId>monApplication</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>monApplication</name>
  <url>http://maven.apache.org</url>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <mapropriete>Message de test</mapropriete>
  </properties>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <resources>
      <resource>
        <directory>src/main/resources</directory>
        <filtering>true</filtering>
      </resource>
    </resources>
  </build>
</project>
```

Dans l'artéfact généré, la propriété est remplacée par sa valeur lors de la phase process-resources du cycle de vie par défaut du projet.

Exemple :

```
artifactId: monApplication
```

Les valeurs des propriétés peuvent être extraites d'un fichier de properties qui va contenir la définition des valeurs des différentes propriétés à utiliser.

Exemple : le fichier src/main/filters/mesproprietes.properties

Exemple :

```
ma.propriete=Ma valeur
```

Les valeurs contenues dans le fichier de properties peuvent faire référence à des propriétés de la JVM en utilisant la syntaxe ${...}.

Résultat :

```
ma.propriete=${ma.propriete}
```

Les propriétés peuvent être soit celles standard de la JVM soit des propriétés définies au lancement de la JVM.

Résultat :

```
mvn clean package "-Dma.propriete.prop=Ma valeur"
```

Dans les fichiers de ressources, il faut utiliser la syntaxe ${...} pour représenter la valeur d'une propriété.

Exemple : le fichier src/main/resources/test.txt

Résultat :

```
valeur: ${ma.propriete}
```

Pour permettre de tenir compte du fichier contenant les propriétés, il faut appliquer une configuration particulière dans le POM:

- Affecter la valeur true dans le tag build/resources/resource/filtering
- Fournir le chemin du fichier properties dans le tag build/filters/filter
- Fournir le chemin du répertoire des ressources dans le tag build/resources/resource/directory : il faut repréciser le chemin puisque la configuration n'est pas celle par défaut

Exemple :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.jmdoudoux.dej.monapp</groupId>
  <artifactId>monApplication</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>monApplication</name>
  <url>http://maven.apache.org</url>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <mapropriete>Message de test</mapropriete>
  </properties>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <filters>
      <filter>src/main/filters/mesproprietes.properties</filter>
    </filters>
    <resources>
      <resource>
        <directory>src/main/resources</directory>
        <filtering>true</filtering>
      </resource>
    </resources>
  </build>
</project>
```

Lors de la génération de l'artéfact,

Résultat :

```
valeur: Ma valeur
```

101.3.7.9. La génération d'un fichier jar contenant les sources du projet

Le plugin maven-source-plugin permet de générer un fichier jar contenant le code source du projet.

Le déploiement de ce fichier dans un dépôt distant peut permettre aux autres développeurs d'attacher le code source de l'artéfact dans leur IDE.

Le goal jar est par défaut exécuté dans la phase package du cycle de vie par défaut du projet.

Exemple :

```
<build>
  <plugins>
...

    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-source-plugin</artifactId>
      <executions>
        <execution>
          <id>attacher-sources</id>
          <goals>
            <goal>jar</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
...
  </plugins>
</build>
```

101.3.8. L'exécution de commandes

Une fois la configuration du projet effectuée, il faut demander à Maven d'effectuer certaines tâches selon les besoins en lui passant en paramètres des commandes.

Le nombre de ces commandes est important et dépend des plugins utilisés.

101.3.8.1. Un résumé des principales commandes

mvn package	Construire le projet pour générer l'artéfact
mvn site	Générer le site de documentation dans le répertoire target/site
mvn -Dtest=<unqualified-classname> test	Exécuter le test unitaire dont le nom de la classe est fourni
mvn -Dmaven.surefire.debug test	Exécuter des tests avec un débogueur distant en utilisant le port 5005. Surefire attend la connexion du débogueur
mvn help:effective-pom	Afficher le contenu du pom en incluant les POM hérités
mvn dependency:tree	Afficher une arborescence des dépendances, incluant les dépendances transitives
mvn clean	Supprimer les fichiers générés par les précédentes générations
mvn clean package	Supprimer les fichiers générés par les précédentes générations et construire le projet pour générer l'artéfact
mvn install	Générer l'artéfact et le déployer dans le dépôt local
mvn -Dmaven.test.failure.ignore=true package	Construire le projet pour générer l'artéfact même si les tests unitaires échouent
mvn eclipse:eclipse	Générer des fichiers de configuration Eclipse à partir du POM (notamment les dépendances)
mvn eclispe:clean eclipse:eclipse	Idem mais les fichiers précédemment générés sont supprimés
mvn -Dmaven.test.skip=true clean package	Supprimer les fichiers générés par les précédentes générations et construire le projet pour générer l'artéfact sans exécuter les tests unitaires
mvn javadoc:javadoc	Générer la Javadoc
mvn javadoc:jar	Générer la Javadoc dans un fichier jar
mvn --version	Afficher les informations sur la version de Maven
mvn test	Exécuter les tests unitaires
mvn compile	Compiler les sources du projet
mvn idea:idea	Générer des fichiers de configuration IntelliJ à partir du POM (notamment les dépendances)
mvn release:prepare release:perform	Livrer l'artéfact (créer un tag dans le gestionnaire de sources et supprimer SNAPSHOT dans la version)
mvn archetype:create -DgroupId=fr.jmdoudoux.dej	Créer un nouveau projet en mode interactif

-DartifactId=monAppli	

101.3.8.2. Les options de la commande mvn

Les principales options de la commande mvn sont :

Option	Rôle
-o,--offline	Mode offline
-h, --help, --usage	Afficher la liste des options utilisables
-X,--debug	Afficher les traces de debug
-v,--version	Afficher des informations sur la version de Maven
-V,--show-version	Afficher des informations sur la version de Maven sans arrêter le build
-s,--settings	Préciser un chemin alternatif pour le fichier de configuration settings.xml
-f,--file	Préciser un fichier POM alternatif
-D,--define	Définir une propriété
-B,--batch-mode	Demander l'exécution en mode batch (pas d'interaction avec l'utilisateur)
-fn,--fail-never	Demander à ce que le build n'échoue pas
-C,--strict-checksums	Demander un échec du build si un checksum ne correspond pas
-c,--lax-checksums	Demander une alerte si un checksum ne correspond pas
-P,--activate-profiles	Préciser la liste des profiles actifs
-up,--update-pluging	
-cpu,--check-plugin-updates	Demander de vérifier les mises à jour des plugins
-fae,--fail-at-end	Demander un échec du build le plus tardif possible : Maven tente d'exécuter tous les goals possibles
-npu,--no-plugin-updates	Demander de ne pas vérifier les mises à jour des plugins
--non-recursive	

101.3.8.3. Le nettoyage d'un projet

Le cycle de vie clean possède trois phases :

Phase	Rôle
pre-clean	Effectuer des tâches avant le nettoyage
clean	Supprimer les fichiers générés dans le répertoire target
post-clean	Effectuer des tâches après le nettoyage

Pour faire le ménage dans les fichiers générés, il faut invoquer la commande

Exemple :

```
mvn clean
```

C'est généralement une bonne pratique d'invoquer le cycle de vie clean avant d'invoquer la phase install du cycle de vie par défaut car demander un nettoyage implique une construction complète du projet.

Exemple :

```
mvn clean install
```

101.3.8.4. La compilation du code source

Pour demander la compilation des sources du projet, il faut demander l'exécution de la phase compile du cycle de vie par défaut.

Exemple :

```
mvn compile
```

Lors de la première exécution d'une commande maven, cette dernière doit télécharger le plugin correspondant et ses dépendances dans le dépôt local. Les exécutions suivantes utiliseront la version stockée dans le dépôt local.

Par convention, le résultat de la compilation des classes est stocké dans le sous-répertoire target/classes du projet.

101.3.8.5. La compilation du code source des tests et leur exécution

Pour demander l'exécution des tests unitaire du projet, il faut demander l'exécution de la phase test du cycle de vie par défaut.

Exemple :

```
mvn test
```

Avant de compiler et d'exécuter les tests, Maven effectue une compilation du code source de l'artéfact.

L'exécution des tests par Maven se fait grâce au plugin Surefire.

Par défaut, le plugin Surefire recherche les tests unitaires dont les classes respectent une convention de nommage particulière :

**/*Test.java
**/Test*.java
**/*TestCase.java

Pour demander uniquement la compilation des tests mais pas leur exécution, il faut invoquer la commande Maven

Exemple :

```
mvn test-compile
```

Même si cela n'est pas une bonne pratique, il est parfois utile de désactiver l'exécution des tests durant la génération du projet.

Il suffit d'utiliser l'option -DskipTests=true dans les paramètres de lancement de la commande mvn.

Pour désactiver définitivement l'exécution des tests, il faut modifier la configuration du plug-in Surefire de Maven dans le fichier pom.xml

Exemple :

```
...
  <build>
    <plugins>
```

```
      <plugin>
        <artifactId>maven-surefire-plugin</artifactId>
        <configuration>
          <skipTests>true</skipTests>
        </configuration>
      </plugin>
    </plugins>
  </build>
...
```

101.3.8.6. La création de l'artéfact

Pour demander la génération d'un artéfact, il faut demander l'exécution de la phase package du cycle de vie par défaut.

Résultat :

```
mvn package
```

L'artéfact généré est mis dans le sous-répertoire target du répertoire du projet.

101.3.8.7. L'installation de l'artéfact dans le dépôt local

Maven utilise la notion de dépôt pour stocker les artéfacts générés ou requis ainsi que les métadonnées qui leur sont associées.

Un artéfact Maven est un élément packagé livrable : par exemple, un fichier jar, war, ear, ...

Le dépôt local est, par défaut, dans le sous-répertoire .m2/repository du répertoire home de l'utilisateur.

Pour demander le déploiement d'un artéfact dans le dépôt local, il faut exécuter la commande :

Résultat :

```
mvn install
```

Maven travaille obligatoirement avec un dépôt local dans lequel il stocke les dépendances, les plugins, les artéfacts pour éviter d'avoir à toujours les télécharger d'internet.

L'installation d'un artéfact dans le dépôt local permet de l'utiliser comme dépendance dans d'autre projet.

Le dépôt Maven central est un dépôt distant qui contient un grand nombre d'artéfacts communs. Il est existe aussi des dépôts miroir.

Il est aussi possible d'utiliser des gestionnaires d'artéfacts, comme Archiva ou Artifactory, qui vont servir de dépôts intermédiaires pour limiter la quantité d'artéfacts téléchargés et restreindre l'utilisation d'artéfacts uniquement à ceux présents dans ces dépôts.

101.3.8.8. Le déploiement d'un artéfact dans un dépôt distant

Pour déployer un artéfact dans un dépôt distant, il faut configurer l'url du dépôt dans le fichier POM et les informations d'authentification dans le fichier settings.xml.

La configuration dans le fichier POM se fait dans un tag <distributionManagement>.

Chaque dépôt est configuré dans un tag <repository>

Exemple :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.jmdoudouxapp</groupId>
  <artifactId>monapp</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <distributionManagement>
    <repository>
      <id>depot-entreprise</id>
      <name>Depot d'entreprise</name>
      <url>scp://depot.entreprise.com/repository/maven2</url>
    </repository>
  </distributionManagement>
</project>
```

Le tag <repository> possède plusieurs tags fils :

Tag	Rôle
<id>	permet de faire référence au serveur défini dans le fichier .m2/settings.xml
<name>	permet de fournir un nom au serveur
<url>	permet de fournir une url utilisable par le plugin Wagon (commence généralement par scp:)

Le tag <snapshotRepository> permet de définir un dépôt qui sera utilisé pour stocker les snapshots.

Si cette configuration est commune à plusieurs projets, il est possible de définir cette configuration dans un POM parent.

Pour demander le déploiement, il suffit d'invoquer la commande mvn deploy.

101.3.8.9. La génération de la Javadoc

Le plugin maven-javadoc-plugin permet de générer la javadoc d'un artéfact et de la déployer dans un dépôt.

Il faut l'activer en précisant le goal «jar».

Exemple :

```
...
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-javadoc-plugin</artifactId>
      <executions>
        <execution>
          <id>attach-javadocs</id>
          <goals>
            <goal>jar</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
...
```

Il suffit alors d'invoquer la commande mvn clean package pour générer l'artéfact (xxx-1.0.jar), le source (xxx-1.0-sources.jar) et la javadoc(xxx-1.0-javadoc.jar). Pour générer ces éléments et les installer dans le dépôt, il suffit d'invoquer la commande mvn clean deploy.

101.3.9. L'utilisation de projets multi-modules

Maven 2.0 propose la possibilité d'utiliser des projets multi-modules.

Un projet multi-modules est un projet qui contient d'autres projets fils : c'est un regroupement logique de sous-projets.

Il contient un fichier pom.xml mais le projet ne crée pas lui-même d'artéfact.

Il est possible d'imbriquer des projets multi-modules.

Il est recommandé d'organiser les projets de manière hiérarchique : les sous-projets sont dans le projet multi-modules. Ce n'est pas une obligation mais cela facilite la compréhension et la mise en oeuvre car la hiérarchie des projets est directement reflétée dans la structure des répertoires. Cependant, ce n'est pas une obligation et une organisation des projets à plat est possible.

Résultat :

```
+-monAppli
  +- pom.xml
  +- monAppliIHM
  | +- pom.xml
  | +- src
  |   +- main
  |
+- java
  +- monAppliUtil
  | +- pom.xml
  | +- src
  |   +- main
  |
+- java
```

Le fichier POM du projet multi-module doit avoir le type de packaging pom

Exemple :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.jmdoudoux.dej</groupId>
  <artifactId>monAppli</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>
  <name>monAppli</name>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <modules>
    <module>monAppliIHM</module>
    <module>monAppliUtil</module>
  </modules>
</project>
```

Le chemin de chacun des modules est précisé dans un tag <module> fils du tag <modules>. Cela indique à Maven que les opérations ne vont pas être réalisées sur le projet mais sur les modules du projet.

Le POM de chacun des modules doit contenir un tag <parent> qui permet d'identifier le module parent.

Le POM du module monAppliIHM

Exemple :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>fr.jmdoudoux.dej</groupId>
    <artifactId>monAppli</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <groupId>fr.jmdoudoux.dej.monappli.ihm</groupId>
  <artifactId>monAppliIHM</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>monAppliIHM</name>
  <url>http://maven.apache.org</url>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>fr.jmdoudoux.dej.monappli.util</groupId>
      <artifactId>monAppliUtil</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
  </dependencies>
</project>
```

Le POM du projet monAppliUtil

Exemple :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>fr.jmdoudoux.dej</groupId>
    <artifactId>monAppli</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <groupId>fr.jmdoudoux.dej.monappli.util</groupId>
  <artifactId>monAppliUtil</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>monAppliUtil</name>
  <url>http://maven.apache.org</url>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Exemple :

```
package fr.jmdoudoux.dej.monappli.util;

public class AppUtil {
    public static String getMessage() {
        return "Hello World!";
    }
}
```

Si les modules ne sont pas dans des sous-répertoire mais dans des répertoires de même niveau que le projet parent, il est nécessaire de mettre «../» dans le chemin du module. C'est notamment le cas si les projets sont utilisés dans Eclipse. Il est aussi nécessaire dans le POM des modules d'utiliser le tag <relativePath> fils du tag <parent> pour préciser le chemin du POM parent qui devra lui aussi commencer par «../».

Résultat :

```
C:\TEMP\monAppli>mvn clean install

[INFO] Scanning for projects...
[INFO] Reactor build order:
[INFO]   monAppli
[INFO]   monAppliUtil
[INFO]   monAppliIHM
[INFO] ------------------------------------------------------------------------
[INFO] Building monAppli
[INFO]    task-segment: [clean, install]
[INFO] ------------------------------------------------------------------------
[INFO] [clean:clean {execution: default-clean}]
[INFO] [site:attach-descriptor {execution: default-attach-descriptor}]
[INFO] [install:install {execution: default-install}]
[INFO] Installing C:\TEMP\monAppli\pom.xml to
C:\java\maven_repository\com\jmdoudoux\test\monAppli\1.0-SNAPSHOT\monAppli-1.0-SNAPSHOT.pom
[INFO] ------------------------------------------------------------------------
[INFO] Building monAppliUtil
[INFO]    task-segment: [clean, install]
[INFO] ------------------------------------------------------------------------
[INFO] [clean:clean {execution: default-clean}]
[INFO] Deleting directory C:\TEMP\monAppli\monAppliUtil\target
[INFO] [resources:resources {execution: default-resources}]
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory C:\TEMP\monAppli\monAppliUtil\src\main\resources
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Compiling 1 source file to C:\TEMP\monAppli\monAppliUtil\target\classes
[INFO] [resources:testResources {execution: default-testResources}]
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory C:\TEMP\monAppli\monAppliUtil\src\test\resources
[INFO] [compiler:testCompile {execution: default-testCompile}]
[INFO] Compiling 1 source file to C:\TEMP\monAppli\monAppliUtil\target\test-classes
[INFO] [surefire:test {execution: default-test}]
[INFO] Surefire report directory: C:\TEMP\monAppli\monAppliUtil\target\surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running fr.jmdoudoux.dej.monappli.util.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.016 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] [jar:jar {execution: default-jar}]
[INFO] Building jar: C:\TEMP\monAppli\monAppliUtil\target\monAppliUtil-1.0-SNAPSHOT.jar
[INFO] [install:install {execution: default-install}]
```

```
[INFO] Installing C:\TEMP\monAppli\monAppliUtil\target\monAppliUtil-1.0-SNAPSHOT.jar
to C:\java\maven_repository\com\jmdoudoux\test\monappli\util\monAppliUtil\1.0-SNAPSH
OT\monAppliUtil-1.0-SNAPSHOT.jar
[INFO] ------------------------------------------------------------------------
[INFO] Building monAppliIHM
[INFO]    task-segment: [clean, install]
[INFO] ------------------------------------------------------------------------
[INFO] [clean:clean {execution: default-clean}]
[INFO] Deleting directory C:\TEMP\monAppli\monAppliIHM\target
[INFO] [resources:resources {execution: default-resources}]
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory C:\TEMP\monAppli\monAppliIHM\src\main\resources
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Compiling 1 source file to C:\TEMP\monAppli\monAppliIHM\target\classes
[INFO] [resources:testResources {execution: default-testResources}]
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory C:\TEMP\monAppli\monAppliIHM\src\test\resources
[INFO] [compiler:testCompile {execution: default-testCompile}]
[INFO] Compiling 1 source file to C:\TEMP\monAppli\monAppliIHM\target\test-classes
[INFO] [surefire:test {execution: default-test}]
[INFO] Surefire report directory: C:\TEMP\monAppli\monAppliIHM\target\surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running fr.jmdoudoux.dej.monappli.ihm.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.015 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] [jar:jar {execution: default-jar}]
[INFO] Building jar: C:\TEMP\monAppli\monAppliIHM\target\monAppliIHM-1.0-SNAPSHOT.jar
[INFO] [install:install {execution: default-install}]
[INFO] Installing C:\TEMP\monAppli\monAppliIHM\target\monAppliIHM-1.0-SNAPSHOT.jar
to C:\java\maven_repository\com\jmdoudoux\test\monappli\ihm\monAppliIHM\1.0-SNAPSHOT\m
onAppliIHM-1.0-SNAPSHOT.jar
[INFO]
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO] ------------------------------------------------------------------------
[INFO] monAppli .............................................. SUCCESS [1.344s]
[INFO] monAppliUtil .......................................... SUCCESS [1.765s]
[INFO] monAppliIHM ........................................... SUCCESS [0.859s]
[INFO] ------------------------------------------------------------------------
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 4 seconds
[INFO] Finished at: Tue Nov 25 09:25:51 CET 2012
[INFO] Final Memory: 15M/510M
[INFO] ------------------------------------------------------------------------
```

L'application contient une simple classe avec une méthode main() pour permettre son exécution.

Exemple :

```
package fr.jmdoudoux.dej.monappli.ihm;

import fr.jmdoudoux.dej.monappli.util.AppUtil;

public class App  {
    public static void main( String[] args ) {
        System.out.println( "Lancement MonAppli" );
        System.out.println( AppUtil.getMessage() );
    }
}
```

Tous les artéfacts doivent être dans le classpath.

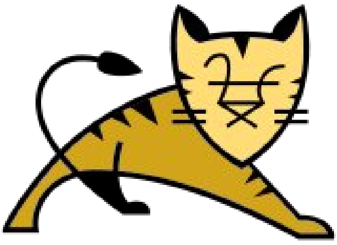
Résultat :

```
C:\TEMP\monAppli>java -cp C:\TEMP\monAppli\monAppliIHM\target\monAppliIHM-1.0-SNAPSHOT.jar;
C:\TEMP\monAppli\monAppliUtil\target\monAppliUtil-1.0-SNAPSHOT.jar
fr.jmdoudoux.dej.monappli.ihm.App
Lancement MonAppli
Hello World!
```

102. Tomcat

Chapitre 102

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur



Apache Tomcat est une implémentation open source d'un conteneur web qui permet donc d'exécuter des applications web reposant sur les technolgoies servlets et JSP.

Tomcat est diffusé en open source sous une licence Apache. C'est aussi l'implémentation de référence des spécifications servlets jusqu'à la vesion 2.4 et JSP jusqu'à la version 2.0 implémentées dans les différentes versions de Tomcat.

En tant qu'implémentation de référence de plusieurs versions des spécifications servlets/JSP, facile à mettre en oeuvre et riche en fonctionnalités, Tomcat est quasi incontournable dans les environnements de développements. Les qualités de ses dernières versions lui permettent d'être fréquemment utilisé dans des environnements de production.

Depuis la version 4, Tomcat est composé de plusieurs éléments :

- Coyote est le connecteur pour les protocoles de communications notamment HTTP, AJP, ...
- Catalina est le conteneur de servlets
- Jasper est le moteur de JSP

Ce chapitre contient plusieurs sections :

- L'historique des versions
- L'installation
- L'exécution de Tomcat
- L'architecture
- La configuration
- L'outil Tomcat Administration Tool
- Le déploiement des applications WEB
- Tomcat pour le développeur
- Le gestionnaire d'applications (Tomcat manager)
- L'outil Tomcat Client Deployer
- Les optimisations
- La sécurisation du serveur

102.1. L'historique des versions

Il existe plusieurs versions de Tomcat qui mettent en oeuvre des versions différentes des spécifications des servlets et des JSP :

Version de Tomcat	Version Servlet	Version JSP	Version EL	Version Java
3.0, 3.1, 3.2, 3.3	2.2	1.1		
4.0, 4.1	2.3	1.2		1.2

5.0	2.4	2.0	2.0	1.4
6.0	2.5	2.1	2.1	1.5
7.0	3.0	2.2	2.2	1.6
8.0	3.1	2.3	3.0	1.7
9.0	4.0	2.3	3.0	1.8

Tomcat 3.x (version initiale)

- implémente les spécifications Servlet 2.2 et JSP 1.1
- rechargement des servlets
- fonctionnalités HTTP de base.

Tomcat 4.x

- implémente les spécifications Servlet 2.3 et JSP 1.2
- nouveau conteneur de servlets Catalina
- nouveau moteur JSP Jasper
- le connecteur Coyote
- utilisation de JMX,
- application d'administration développée en Struts

Tomcat 5.x

- implémente les spécifications Servlet 2.4 et JSP 2.0
- performances améliorées
- wrappers natifs pour Windows et Unix
- amélioration du traitement des JSP

Tomcat 5.5 nécessite un J2SE 5.0 pour fonctionner. Un module dédié permet d'utiliser Tomcat 5.5 avec un JDK 1.4 mais cela n'est pas recommandé.

Tomcat 5.5 utilise le compilateur d'Eclipse pour compiler les JSP : il n'est donc plus nécessaire d'installer un JDK pour faire fonctionner Tomcat, un JRE suffit.

La configuration de Tomcat 5.5 est différente de celle de Tomcat 5.0 sur de nombreux points

Tomcat 6.x

- implémente la version 2.5 des spécifications des servlets (JSR-154)
- implémente la version 2.1 des spécifications des JSP (JSR-245)
- implémente Unified EL (Unified Expression Langage)
- utilise Java 5
- amélioration de l'usage mémoire

Tomcat 7.x

- implémente la version 3.0 des spécifications des servlets
- implémente la version 2.2 des spécifications des JSP
- implémente Unified EL 2.2 (Unified Expression Langage)
- utilise Java 5
- amélioration de l'usage mémoire

Tomcat 8.x

- implémente la version 3.1 des spécifications des servlets
- implémente la version 2.3 des spécifications des JSP
- implémente Unified EL 3.0 (Unified Expression Langage)

Tomcat 9.x

- implémente la version 4.0 des spécifications des servlets

102.2. L'installation

Tomcat est une application écrite en Java, il est possible de l'installer et de l'exécuter sous tous les environnements disposant d'une machine virtuelle Java : un JRE, ou même un JDK pour certaines anciennes versions, est un prérequis pour permettre son exécution.

Pour les versions de Tomcat nécessitant un JDK, il faut que la variable d'environnement JAVA_HOME soit définie avec comme valeur le répertoire d'installation du JDK. Ceci permet notamment à Tomcat de trouver le compilateur Java pour compiler les JSP.

L'installation de Tomcat de façon universelle se fait simplement :

- Tomcat est fourni dans une archive de type zip qu'il faut télécharger
- décompresser l'archive dans un répertoire du système
- il est préférable de définir la variable d'environnement système CATALINA_HOME qui possède comme valeur le répertoire d'installation de Tomcat

Sous Windows, Tomcat propose un package d'installation qui va permettre en plus :

- de demander et configurer le port du connecteur http à utiliser
- de créer une entrée dans le menu « démarrer / Programme » avec des raccourcis vers quelques fonctionnalités
- d'exécuter Tomcat uniquement sous la forme d'un service Windows

102.2.1. L'installation de Tomcat 3.1 sous Windows 98

Il est possible de récupérer Tomcat sur le site d'Apache Tomcat. Il faut choisir la dernière version stable de préférence, le numéro de version et le répertoire bin pour récupérer le fichier apache-tomcat-x-y-zz.zip.

Il faut ensuite décompresser le fichier dans un répertoire du système par exemple dans C:\. L'archive est décompressée dans un répertoire nommé jakarta-tomcat

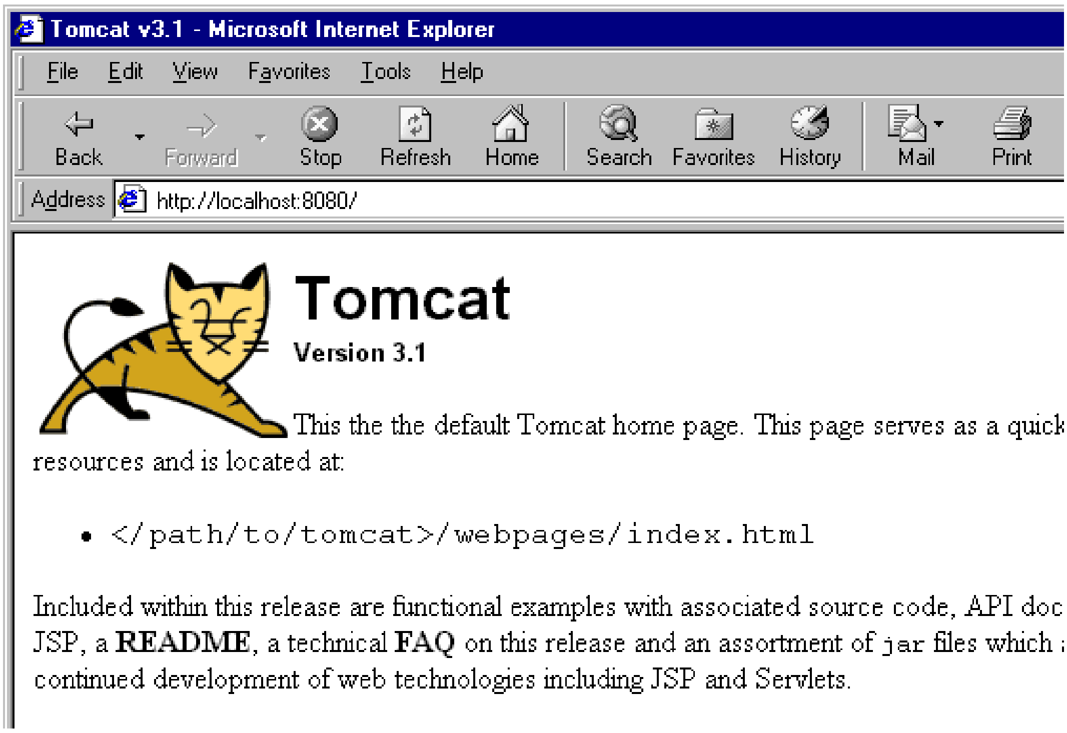
Dans une boîte DOS, assigner le répertoire contenant Tomcat à la variable d'environnement TOMCAT_HOME. Le plus simple est de l'ajouter dans le fichier autoexec.bat.

Exemple :

```
set TOMCAT_HOME=c:\jakarta-tomcat
```

Pour lancer Tomcat, il faut exécuter le fichier startup.bat dans le répertoire TOMCAT_HOME\bin

Pour vérifier que Tomcat s'exécute correctement, il faut saisir l'url http://localhost:8080 dans un browser. La page d'accueil de Tomcat s'affiche.


Tomcat v3.1 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print
Address http://localhost:8080/

Tomcat
Version 3.1

This the the default Tomcat home page. This page serves as a quick
resources and is located at:

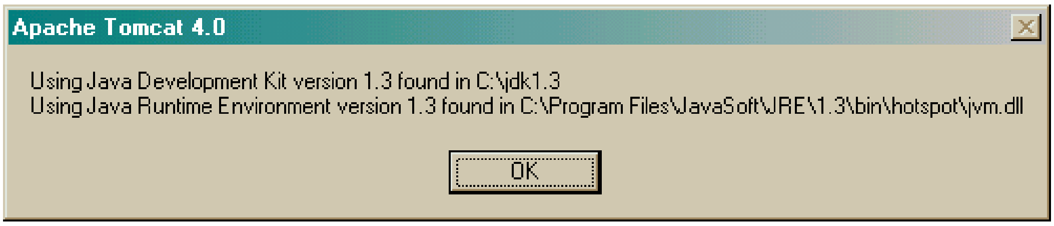
- `</path/to/tomcat>/webpages/index.html`

Included within this release are functional examples with associated source code, API doc
JSP, a **README**, a technical **FAQ** on this release and an assortment of `jar` files which
continued development of web technologies including JSP and Servlets.

Le script %TOMCAT_HOME%\bin\shutdown.bat permet de stopper Tomcat.

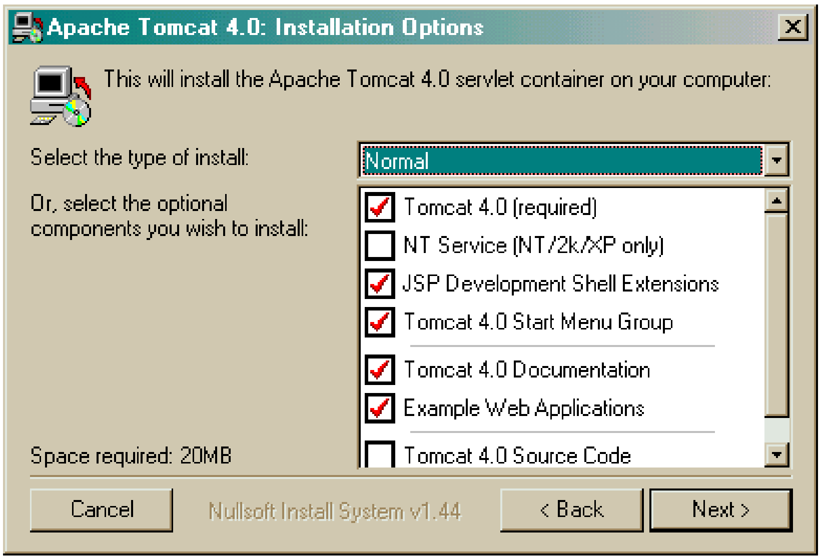
102.2.2. L'installation de Tomcat 4.0 sur Windows 98

Il suffit de télécharger et d'exécuter le programme jakarta-tomcat-4.0.exe



Using Java Development Kit version 1.3 found in C:\jdk1.3
Using Java Runtime Environment version 1.3 found in C:\Program Files\JavaSoft\JRE\1.3\bin\hotspot\jvm.dll

L'assistant affiche la licence, puis permet de sélectionner les options et le répertoire d'installation.


Apache Tomcat 4.0: Installation Options
Cancel
Nullsoft Install System v1.44
< Back
Next >

This will install the Apache Tomcat 4.0 servlet container on your computer:

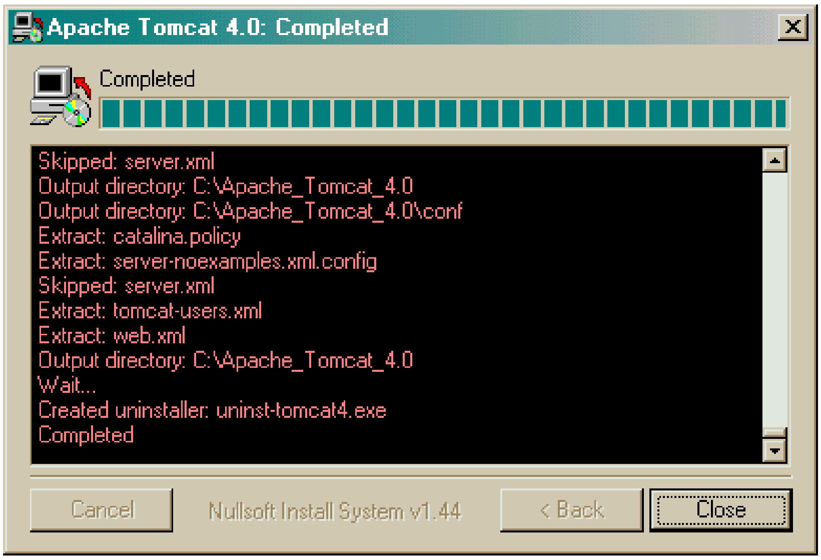
Select the type of install: Normal

Or, select the optional components you wish to install:

- ☑ Tomcat 4.0 (required)
- ☐ NT Service (NT/2k/XP only)
- ☑ JSP Development Shell Extensions
- ☑ Tomcat 4.0 Start Menu Group
- ☑ Tomcat 4.0 Documentation
- ☑ Example Web Applications
- ☐ Tomcat 4.0 Source Code

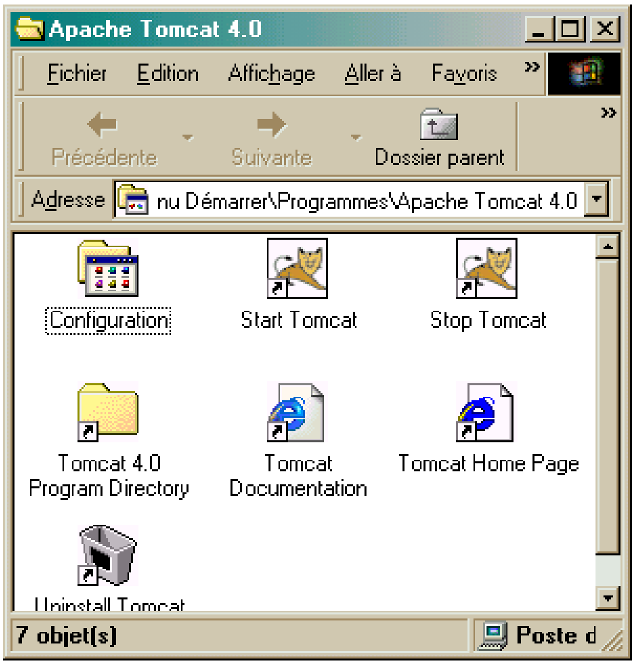
Space required: 20MB

L'assistant copie les fichiers.



```
Skipped: server.xml
Output directory: C:\Apache_Tomcat_4.0
Output directory: C:\Apache_Tomcat_4.0\conf
Extract: catalina.policy
Extract: server-noexamples.xml.config
Skipped: server.xml
Extract: tomcat-users.xml
Extract: web.xml
Output directory: C:\Apache_Tomcat_4.0
Wait...
Created uninstaller: uninst-tomcat4.exe
Completed
```

Un ensemble de raccourcis est créé dans l'option "Apache Tomcat 4.0" du menu "Démarrer/Programmes".



Il faut définir la variable d'environnement système JAVA_HOME qui doit avoir comme valeur le chemin absolu du répertoire d'installation du J2SDK.

Pour la version 4.1, il faut télécharger le fichier jakarta-tomcat-4.1.40.exe sur le site http://archive.apache.org/dist/tomcat/tomcat-4/v4.1.40/bin/

102.2.3. L'installation de Tomcat 5.0 sur Windows

Il faut télécharger le fichier jakarta-tomcat-5.0.30.exe sur le site http://archive.apache.org/dist/tomcat/tomcat-5/v5.0.30/bin/

La version 5 utilise un programme d'installation standard guidé par un assistant qui propose les étapes suivantes :

- la page d'accueil s'affiche, cliquez sur le bouton « Next »
- la page d'acceptation de la licence (« Licence agreement ») s'affiche, lire la licence et si vous l'acceptez, cliquez sur le bouton « I Agree »
- la page de sélection des composants à installer (« Choose components ») s'affiche, il faut sélectionner ou non chacun des composants ou utiliser un type d'installation qui contient une pré-configuration, cliquez sur le bouton

« Next »

- la page de sélection du répertoire d'installation (« Choose install location ») s'affiche, sélectionnez le répertoire de destination et cliquez sur le bouton « Next »
- la page de configuration (« Basic configuration ») s'affiche et permet de définir le port du connecteur http (8080 par défaut) utilisé, le nom et le mot de passe de l'administrateur. Saisissez ces informations et cliquez sur le bouton « Next »
- la page de sélection du chemin de la JVM (« Java Virtual Machine ») permet de sélectionner le chemin du JRE. Cliquez sur le bouton « Install »
- l'installation s'effectue

la page de fin d'affiche. Une case à cocher permet de demander le lancement de Tomcat. Cliquez sur le bouton « Finish ».

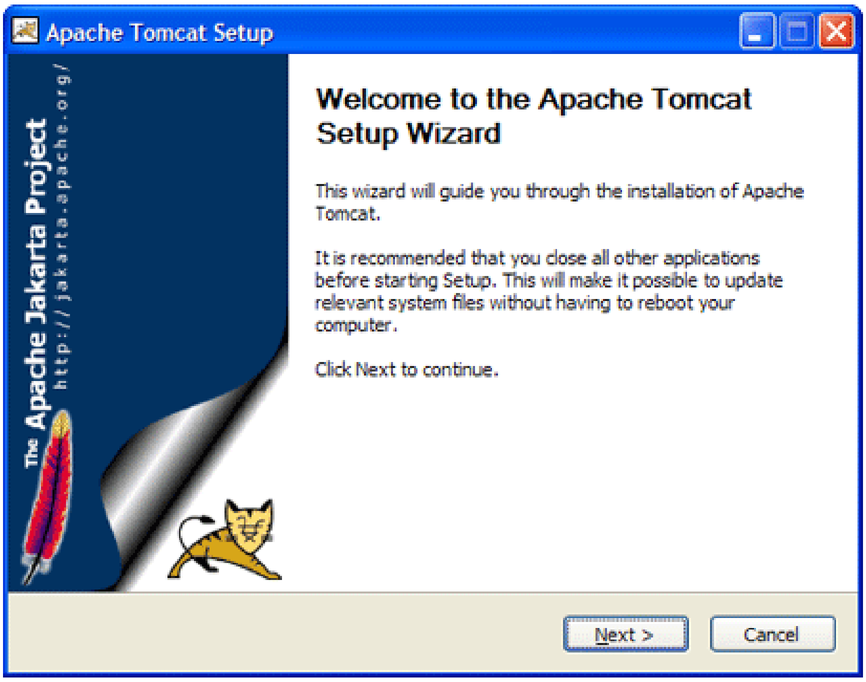
102.2.4. L'installation de Tomcat 5.5 sous Windows avec l'installer

L'url pour télécharger la version 5.5 de Tomcat est http://archive.apache.org/dist/tomcat/tomcat-5/
Attention : Tomcat 5.5 est packagé différemment de ses précédentes versions : les différents modules qui composent Tomcat sont fournis séparément. Ceci permet d'installer uniquement les modules souhaités de Tomcat notamment dans un environnement de production.
Les modules sont :

- Core : ce module contient le serveur Tomcat
- Deployer : ce module contient le TCD (Tomcat Client Deployer) qui utilise Ant pour compiler, valider et déployer une application web
- Embedded : ce module contient une version embarquée de Tomcat (pour l'intégrer dans une autre application)
- Administration web application : ce module contient l'application web d'administration de Tomcat
- JDK 1.4 Compatibility Package : ce mode doit être utilisé pour exécuter Tomcat 5.5 avec un JDK 1.4
- Documentation : ce module contient la documentation seule (ce module est inclus dans le module Core)

Téléchargez le setup de la dernière version de Tomcat 5.5 (par exemple apache-tomcat-5.5.23.exe) et exécutez ce fichier.

Attention : l'outil d'installation ne permet que l'exécution de Tomcat sous la forme d'un service Windows.



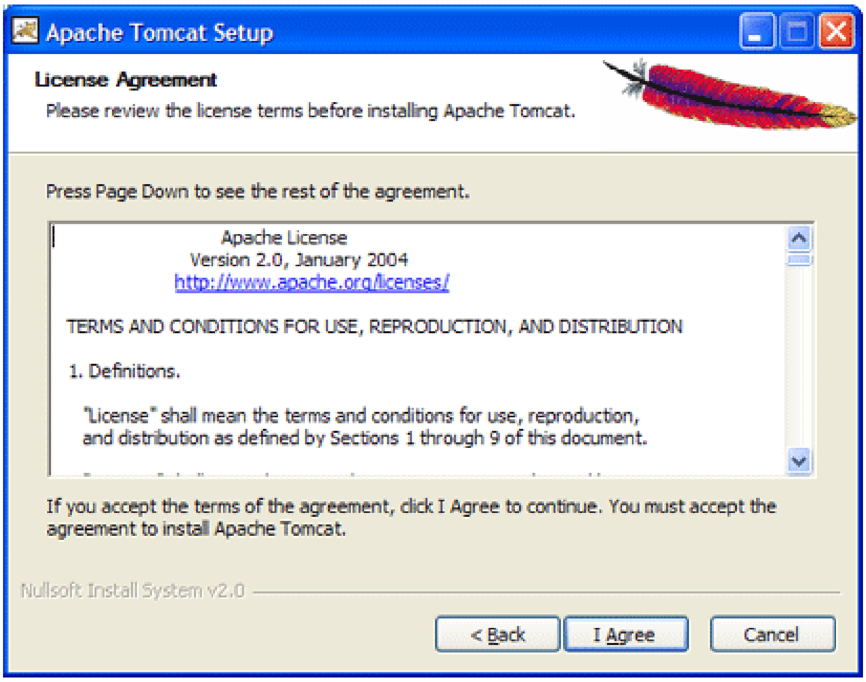
Welcome to the Apache Tomcat Setup Wizard

This wizard will guide you through the installation of Apache Tomcat.

It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

Click Next to continue.

Cliquez sur le bouton « Next ».



License Agreement

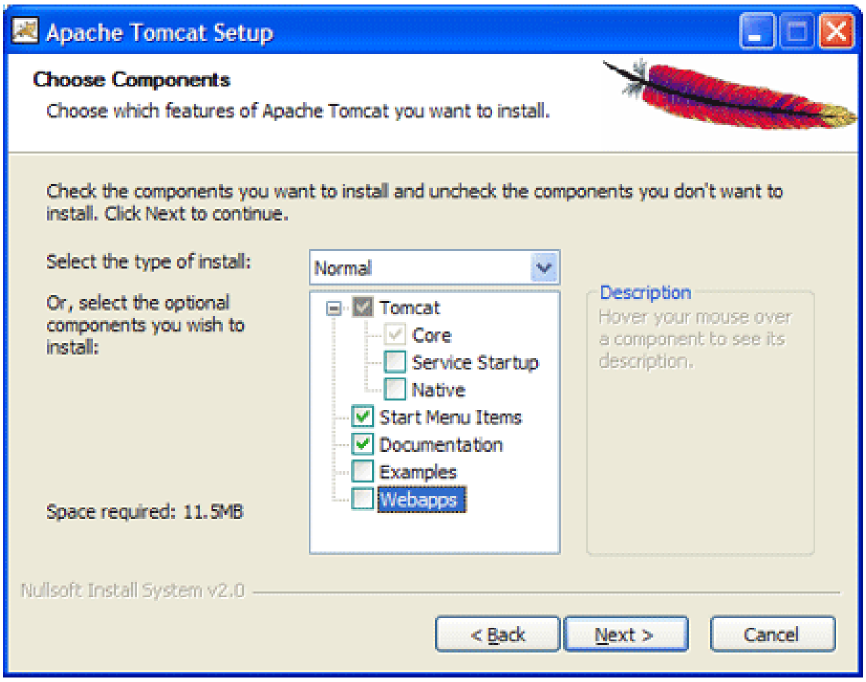
Please review the license terms before installing Apache Tomcat.

Press Page Down to see the rest of the agreement.

```
                    Apache License
               Version 2.0, January 2004
            http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.
```

If you accept the terms of the agreement, click I Agree to continue. You must accept the agreement to install Apache Tomcat.

Lisez la licence et si vous l'acceptez cliquez sur le bouton « I Agree ».



Choose Components

Choose which features of Apache Tomcat you want to install.

Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.

Select the type of install: Normal

Or, select the optional components you wish to install:

- ☑ Tomcat
  - ☑ Core
  - □ Service Startup
  - □ Native
- ☑ Start Menu Items
- ☑ Documentation
- □ Examples
- □ Webapps

Space required: 11.5MB

Description

Hover your mouse over a component to see its description.

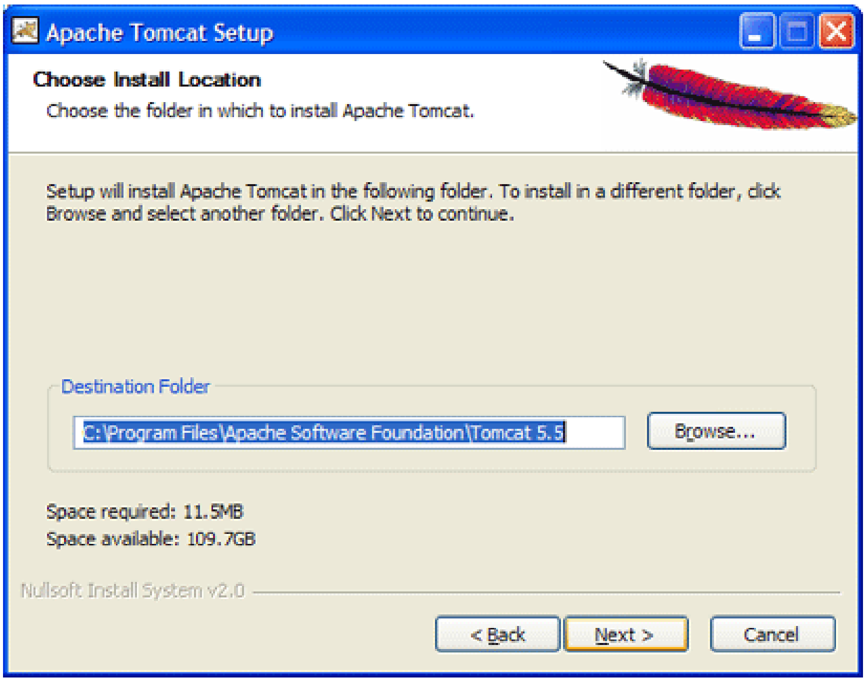
Cette page permet de sélectionner les composants à installer en sélectionnant le type d'installation. Le type custom permet une sélection de chaque composant.

Le composant « Service Startup » permet de demander le démarrage automatique du service Tomcat.

Le composant « Native » permet d'installer certaines librairies natives.

Le composant « Start Menu Items » permet de créer une entrée dans le menu « Démarrer / Programme » avec des raccourcis vers certaines fonctionnalités.

Sélectionnez le type d'installation, les composants à installer si besoin et cliquez sur le bouton « Next ».



Apache Tomcat Setup

Choose Install Location

Choose the folder in which to install Apache Tomcat.

Setup will install Apache Tomcat in the following folder. To install in a different folder, click Browse and select another folder. Click Next to continue.

Destination Folder

C:\Program Files\Apache Software Foundation\Tomcat 5.5

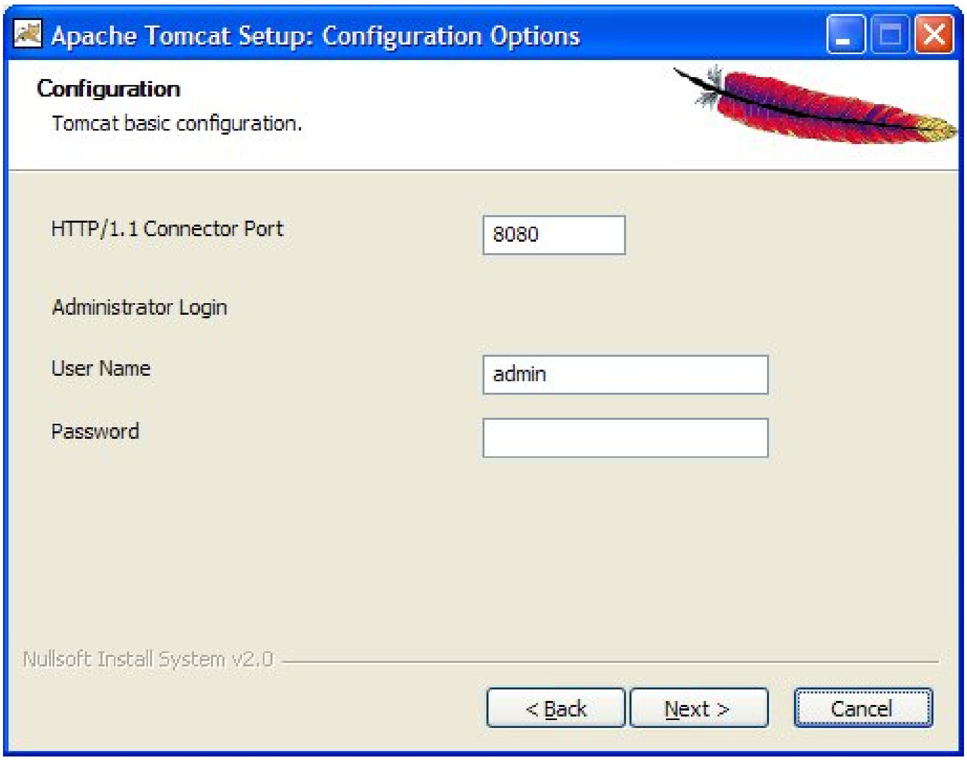
Space required: 11.5MB

Space available: 109.7GB

Nullsoft Install System v2.0

Cette page permet de sélectionner le répertoire d'installation.

Sélectionnez un autre répertoire si besoin et cliquez sur le bouton « Next ».



Apache Tomcat Setup: Configuration Options

Configuration

Tomcat basic configuration.

HTTP/1.1 Connector Port: 8080

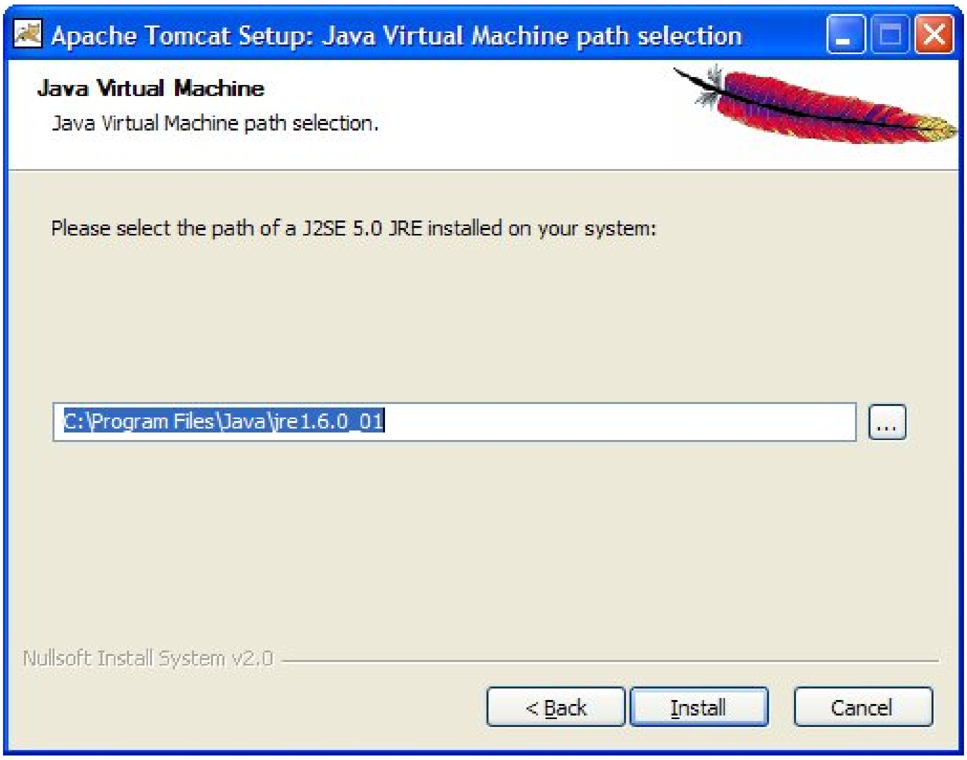
Administrator Login

User Name: admin

Password:

Nullsoft Install System v2.0

Cette page permet de préciser le port du connecteur http à utiliser (8080 par défaut) et de préciser les informations de login de l'administrateur de Tomcat.


Apache Tomcat Setup: Java Virtual Machine path selection

Java Virtual Machine

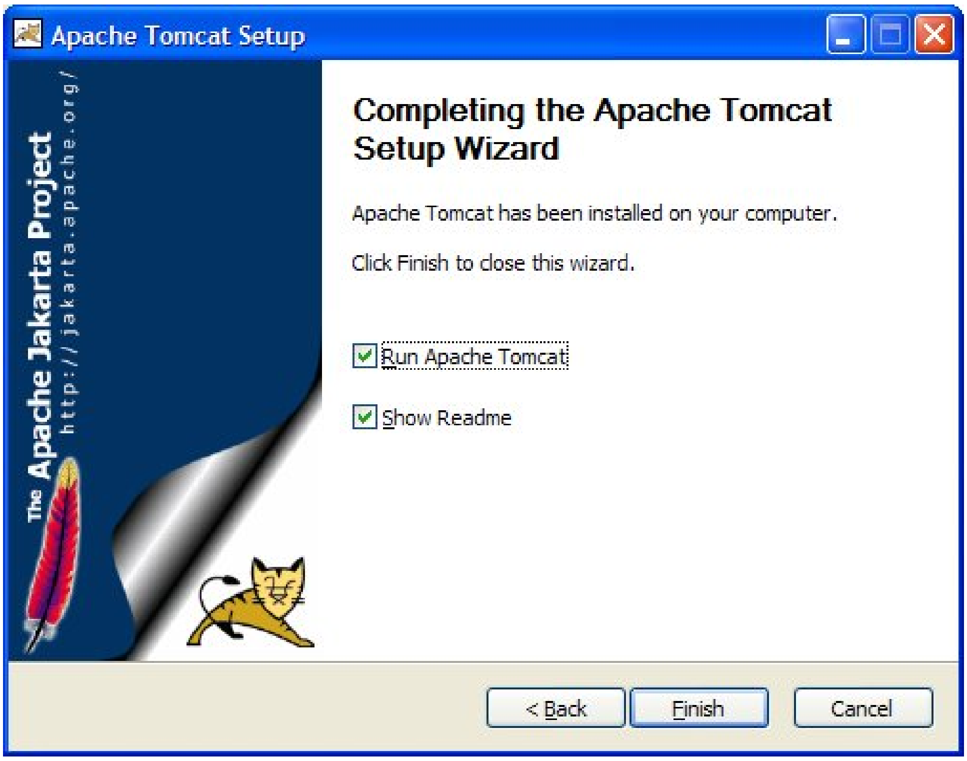
Java Virtual Machine path selection.

Please select the path of a J2SE 5.0 JRE installed on your system:

C:\Program Files\Java\jre1.6.0_01

La page suivante permet de préciser le chemin du JRE 5.0 minimum à utiliser.

Cliquez sur le bouton « Install » pour démarrer l'installation de Tomcat.


Apache Tomcat Setup

Completing the Apache Tomcat Setup Wizard

Apache Tomcat has been installed on your computer.

Click Finish to close this wizard.

☑ Run Apache Tomcat

☑ Show Readme

Cliquez sur le bouton « Finish ».

L'installation des autres modules de Tomcat se fait en les décompressant dans le répertoire d'installation mais en s'assurant que le serveur est arrêté.

102.2.5. L'installation Tomcat 6.0 sous Windows avec l'installer

La procédure est similaire à celle de la version 5.5 de Tomcat.
L'url pour télécharger la version 6.0 de Tomcat est https://tomcat.apache.org/download-60.cgi
Exécutez le fichier téléchargé, par exemple apache-tomcat-6.0.10.exe

L'installation se fait par un assistant sur les pages :

- « Welcome to Apache Tomcat Setup Wizard », cliquez sur le bouton « Next »
- « Licence Agreement », lisez la licence et si vous l'acceptez cliquez sur le bouton « I Agree »
- « Choose Components », sélectionnez les éléments à installer et cliquez sur le bouton « Next »
- « Choose Install Location », cliquez sur le bouton « Next »
- « Configuration », modifier au besoin le numéro de port du connecteur http (qui sera donc utilisé dans les url) et les informations de login de l'administrateur puis cliquez sur le bouton « Next »
- « Java Virtual Machine », cliquez sur le bouton « Install »
- « Completing the Apache Tomcat Setup Wizard », cliquez sur le bouton Finish

102.2.6. La structure des répertoires

Le répertoire d'installation de Tomcat contient plusieurs répertoires.

102.2.6.1. La structure des répertoires de Tomcat 4

L'arborescence du répertoire d'installation de Tomcat est la suivante :

- bin : contient un ensemble de scripts pour la mise en oeuvre de Tomcat
- common : le sous-répertoire lib contient les bibliothèques utilisées par Tomcat et mises à disposition de toutes les applications qui seront exécutées dans Tomcat
- conf : contient des fichiers de propriétés notamment les fichiers server.xml, tomcat-users.xml et le fichier par défaut web.xml
- logs : contient les journaux d'exécution
- server : contient des bibliothèques utilisées par Tomcat et l'application web d'administration de Tomcat
- temp : est un répertoire temporaire utilisé lors des traitements
- webapps : contient les applications web exécutées sous Tomcat
- work : contient le résultat de la compilation des JSP en servlets

102.2.6.2. La structure des répertoires de Tomcat 5

Le répertoire d'installation de Tomcat 5.x contient plusieurs répertoires :

- bin : scripts et exécutables pour gérer Tomcat
- common : bibliothèques et classes communes pour Catalina et les applications web
- conf : fichiers de configurations
- logs : journaux de Catalina et des applications web
- server : bibliothèques et classes utilisées uniquement par Catalina
- shared : bibliothèques et classes partagées par les applications web
- temp : répertoire de stockage de fichiers temporaires
- webapps : répertoire de déploiement des applications web
- work : répertoire de travail (répertoires et fichiers notamment pour la compilation des JSP)

Le répertoire conf contient en standard plusieurs fichiers de configuration :

Fichier	Rôle
catalina.policy	

catalina.properties	Configuration du chargement des classes par Tomcat
context.xml	Configuration par défaut utilisée par tous les contextes
logging.properties	Configuration des logs de Tomcat
server.xml	Configuration du serveur Tomcat
tomcat-users.xml	Contient les données utiles pour l'authentification et pour les habilitations (user et rôle)
web.xml	Descripteur de déploiement par défaut utilisé pour toutes les applications web avant de traiter le fichier des applications

Le répertoire logs est le répertoire par défaut des logs. Sa taille ne fait que croître : il est donc nécessaire de la surveiller notamment dans un environnement de production.

Remarque : l'utilisation du répertoire shared pour mettre des bibliothèques ou des classes est déconseillée. C'est une particularité de Tomcat : il est préférable d'utiliser le répertoire WEB-INF/classes pour les classes et WEB-INF/lib pour les bibliothèques de la webapp car c'est le standard.

102.2.6.3. La structure des répertoires de Tomcat 6

La structure des répertoires est similaire à celle de Tomcat 5 hormis pour les répertoires shared et server qui sont remplacés par un unique répertoire lib. Ce répertoire lib ne contient pas de sous-répertoire lib et classes : il contient directement les bibliothèques.

102.3. L'exécution de Tomcat

Le lancement de Tomcat s'effectue en utilisant un script fourni dans le sous-répertoire d'installation de Tomcat. Sous Windows, il est possible de lancer Tomcat sous la forme d'un service.

102.3.1. L'exécution sous Windows de Tomcat 4.0

Sous Windows, pour lancer Tomcat manuellement, il faut exécuter la commande startup.bat dans le sous-répertoire bin du répertoire où est installé Tomcat. La commande shutdown.bat permet inversement de stopper l'exécution de Tomcat.

Par défaut, le serveur web intégré dans Tomcat utilise le port 8080 pour recevoir les requêtes HTTP. Pour vérifier la bonne installation de l'outil, il suffit d'ouvrir un navigateur et de demander l'URL : http://localhost:8080/

102.3.2. L'exécution sous Windows de Tomcat 5.0

<u>En utilisant les scripts startup et shutdown</u>

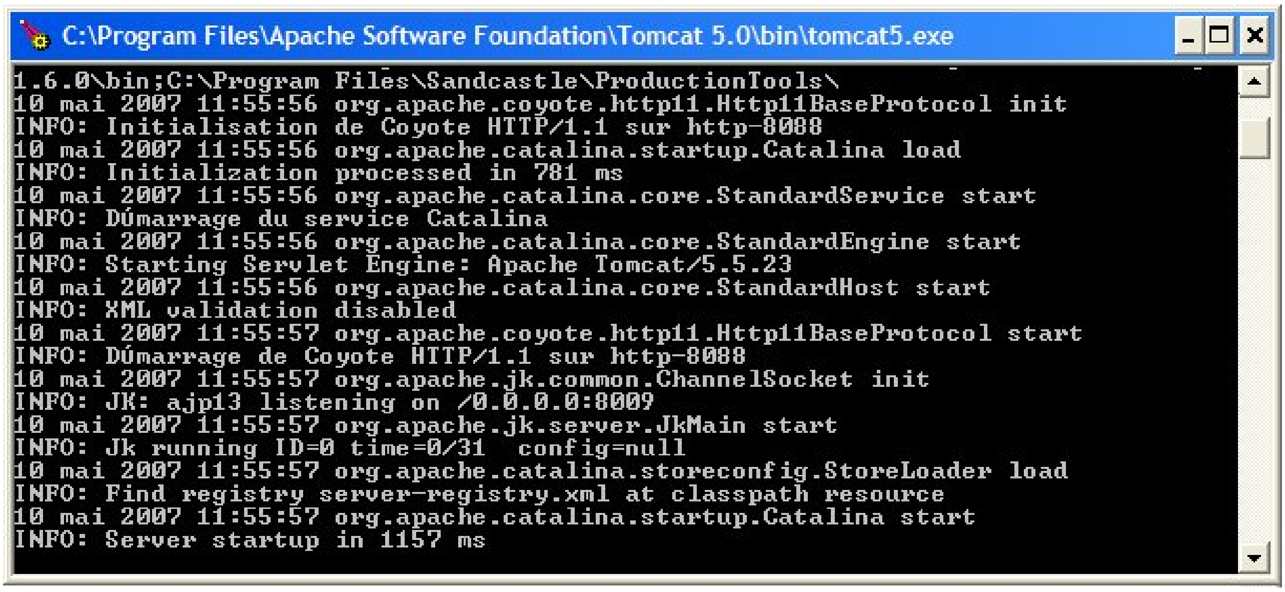
Pour lancer Tomcat, il faut d'exécuter le script startup.bat du sous-répertoire bin.

Pour arrêter Tomcat, il suffit d'exécuter le script shutdown.bat du sous-répertoire bin.

Pour une utilisation en ligne de commandes (sans IDE pour piloter Tomcat), il est pratique de créer un lien vers ces deux scripts, par exemple sur le bureau. L'avantage de les mettre sur le bureau est qu'il est possible de leur assigner des raccourcis clavier.

<u>En utilisant l'application tomcat5.exe</u>

En lançant le programme bin/Tomcat5.exe du répertoire d'installation de Tomcat, Tomcat est lancé sous la forme d'un service : les messages de la console sont affichés dans la boîte DOS associée au processus.


C:\Program Files\Apache Software Foundation\Tomcat 5.0\bin\tomcat5.exe

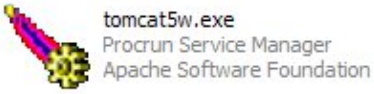
```
1.6.0\bin;C:\Program Files\Sandcastle\ProductionTools\
10 mai 2007 11:55:56 org.apache.coyote.http11.Http11BaseProtocol init
INFO: Initialisation de Coyote HTTP/1.1 sur http-8088
10 mai 2007 11:55:56 org.apache.catalina.startup.Catalina load
INFO: Initialization processed in 781 ms
10 mai 2007 11:55:56 org.apache.catalina.core.StandardService start
INFO: Dúmarrage du service Catalina
10 mai 2007 11:55:56 org.apache.catalina.core.StandardEngine start
INFO: Starting Servlet Engine: Apache Tomcat/5.5.23
10 mai 2007 11:55:56 org.apache.catalina.core.StandardHost start
INFO: XML validation disabled
10 mai 2007 11:55:57 org.apache.coyote.http11.Http11BaseProtocol start
INFO: Dúmarrage de Coyote HTTP/1.1 sur http-8088
10 mai 2007 11:55:57 org.apache.jk.common.ChannelSocket init
INFO: JK: ajp13 listening on /0.0.0.0:8009
10 mai 2007 11:55:57 org.apache.jk.server.JkMain start
INFO: Jk running ID=0 time=0/31  config=null
10 mai 2007 11:55:57 org.apache.catalina.storeconfig.StoreLoader load
INFO: Find registry server-registry.xml at classpath resource
10 mai 2007 11:55:57 org.apache.catalina.startup.Catalina start
INFO: Server startup in 1157 ms
```

Tomcat apparaît dans les processus du gestionnaire de tâches.

Pour arrêter Tomcat dans ce cas, il faut fermer la fenêtre Dos : le serveur sera arrêté proprement.

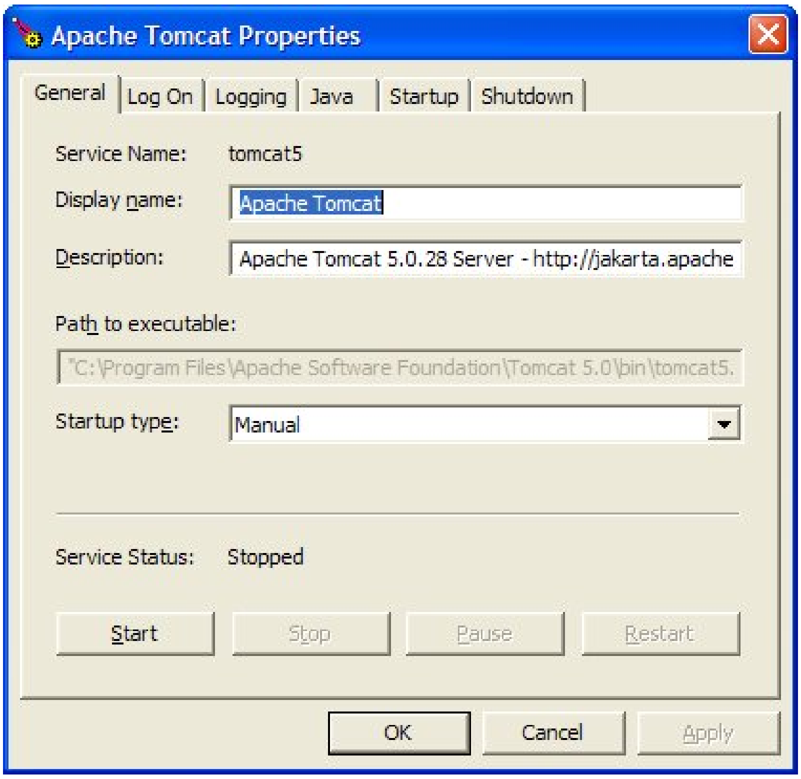
En utilisant Tomcat en tant que service Windows

Le programme d'installation de Tomcat fourni un utilitaire supplémentaire qui permet d'exécuter et de gérer Tomcat en tant que service Windows.



tomcat5w.exe
Procrun Service Manager
Apache Software Foundation

En lançant le programme bin/tomcat5w.exe du répertoire d'installation de Tomcat, une application qui permet de configurer et de gérer l'exécution de Tomcat sous la forme d'un service Windows est lancée.

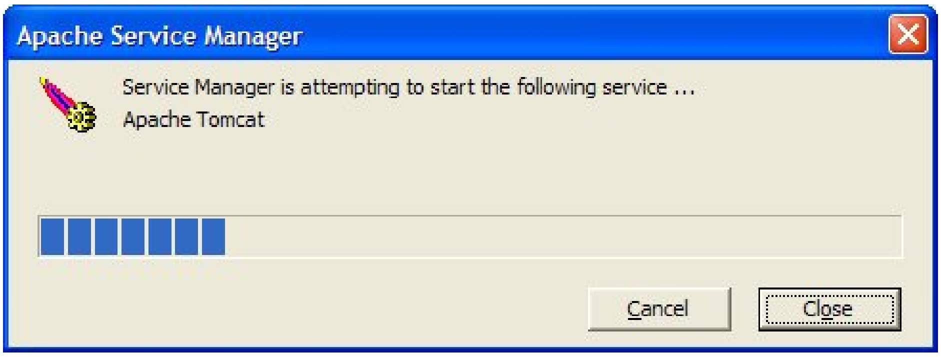


Service Name: tomcat5
Display name: Apache Tomcat
Description: Apache Tomcat 5.0.28 Server - http://jakarta.apache
Path to executable:
"C:\Program Files\Apache Software Foundation\Tomcat 5.0\bin\tomcat5.
Startup type: Manual
Service Status: Stopped

Cette application permet de gérer Tomcat en tant que service Windows.

L'onglet « General » permet de gérer l'exécution de Tomcat sous la forme d'un service.

Pour gérer le statut du service de Tomcat, il suffit d'utiliser le bouton correspondant.



Service Manager is attempting to start the following service ...
Apache Tomcat

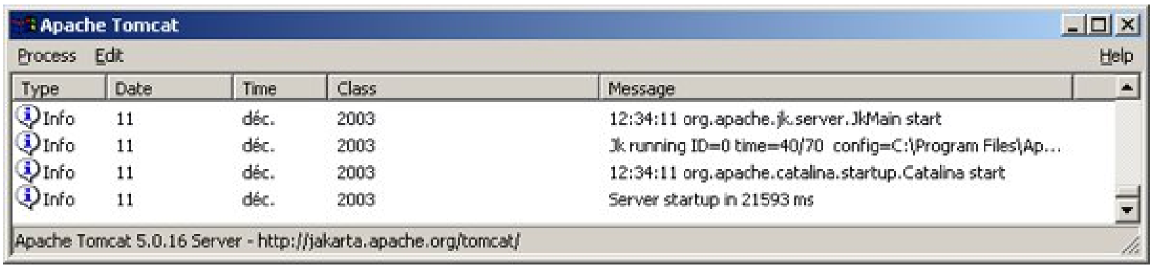
Les autres onglets permettent de préciser des paramètres d'exécution de Tomcat.

Après le démarrage de Tomcat, une icône apparaît dans la barre d'icône

Elle possède un menu contextuel qui permet de réaliser plusieurs actions :

Pour arrêter Tomcat lorsqu'il est démarré de cette façon, il faut cliquer sur le bouton « Stop » dans l'onglet « General »

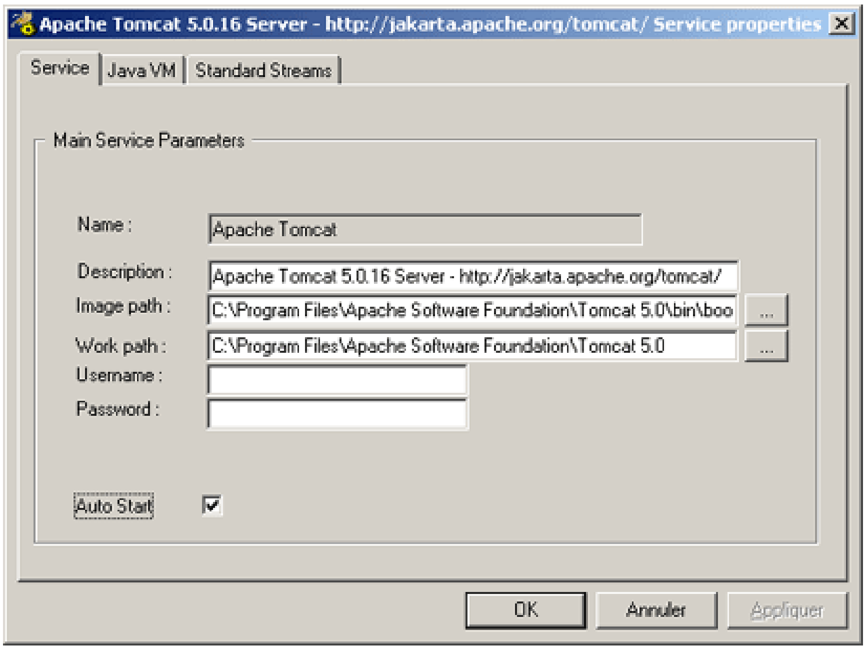
- « Open Console Monitor » : permet d'afficher un journal des messages émis par Tomcat



```
Type   Date   Time   Class   Message
Info   11     déc.   2003    12:34:11 org.apache.jk.server.JkMain start
Info   11     déc.   2003    Jk running ID=0 time=40/70  config=C:\Program Files\Ap...
Info   11     déc.   2003    12:34:11 org.apache.catalina.startup.Catalina start
Info   11     déc.   2003    Server startup in 21593 ms
```

Apache Tomcat 5.0.16 Server - http://jakarta.apache.org/tomcat/

- « About » : permet d'afficher la licence de Tomcat
- « Properties » : permet de changer les propriétés de Tomcat
- L'onglet « Service » permet de préciser les informations générales concernant l'exécution de Tomcat


Apache Tomcat 5.0.16 Server - http://jakarta.apache.org/tomcat/ Service properties
Service
Java VM
Standard Streams
Main Service Parameters
OK
Annuler
Appliquer

Name : Apache Tomcat

Description : Apache Tomcat 5.0.16 Server - http://jakarta.apache.org/tomcat/

Image path : C:\Program Files\Apache Software Foundation\Tomcat 5.0\bin\boo

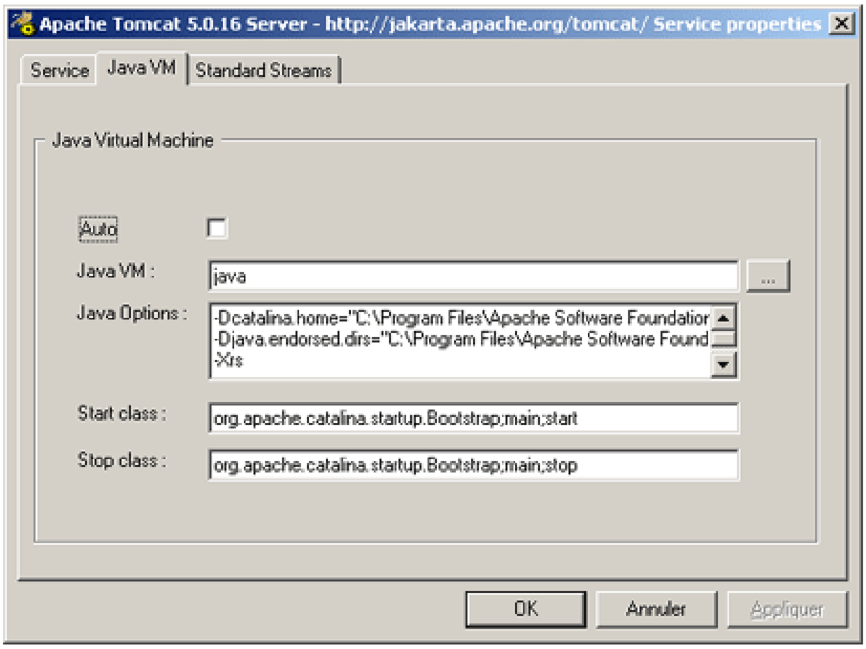
Work path : C:\Program Files\Apache Software Foundation\Tomcat 5.0

Username :

Password :

Auto Start ☑

- L'onglet Java VM permet de préciser les options utilisées lors du lancement de la JVM dans laquelle Tomcat s'exécute


Apache Tomcat 5.0.16 Server - http://jakarta.apache.org/tomcat/ Service properties
Service
Java VM
Standard Streams
Java Virtual Machine
OK
Annuler
Appliquer

Auto □

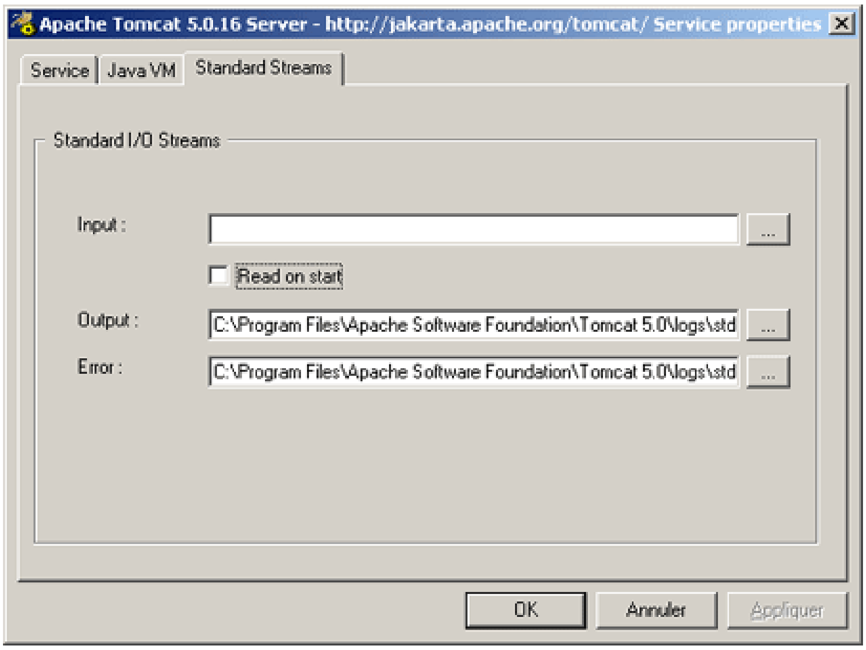
Java VM : java

Java Options : -Dcatalina.home="C:\Program Files\Apache Software Foundatior
-Djava.endorsed.dirs="C:\Program Files\Apache Software Found
-Xrs

Start class : org.apache.catalina.startup.Bootstrap;main;start

Stop class : org.apache.catalina.startup.Bootstrap;main;stop

- L'onglet « Standard Streams » permet de préciser la localisation des fichiers de logs


Apache Tomcat 5.0.16 Server - http://jakarta.apache.org/tomcat/ Service properties
Service
Java VM
Standard Streams
Standard I/O Streams
OK
Annuler
Appliquer

Input :

□ Read on start

Output : C:\Program Files\Apache Software Foundation\Tomcat 5.0\logs\std

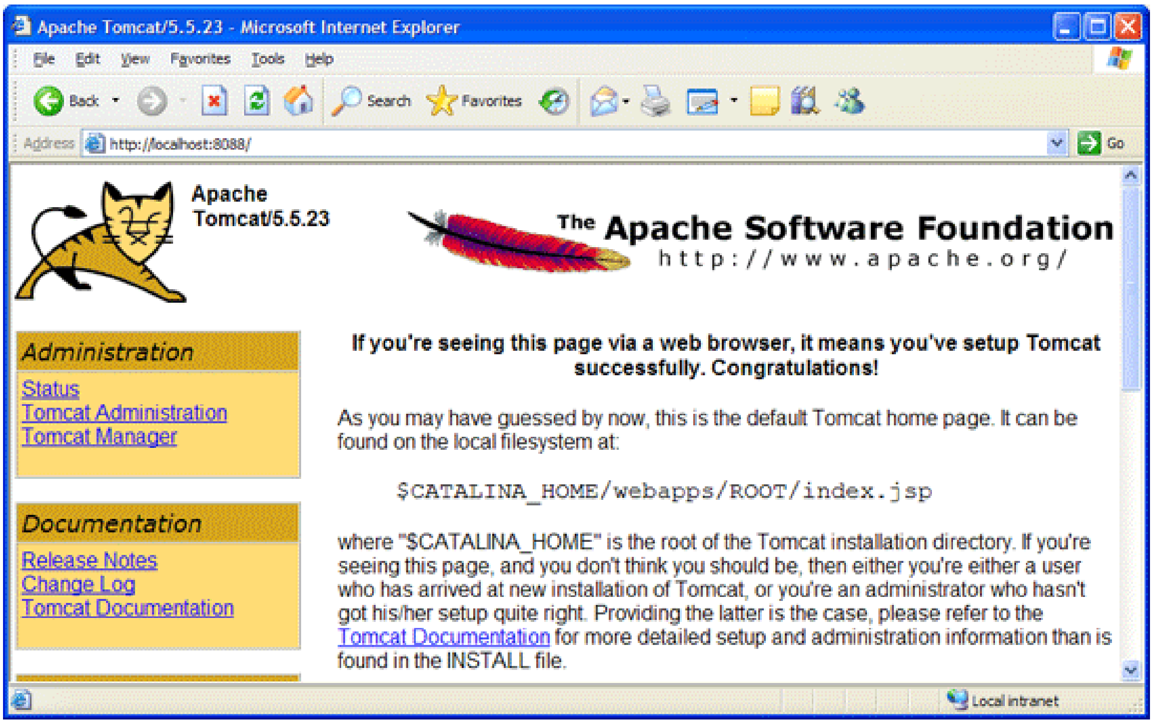
Error : C:\Program Files\Apache Software Foundation\Tomcat 5.0\logs\std

- « Shutdown » : permet d'arrêter le service Tomcat

102.3.3. La vérification de l'exécution

Pour vérifier la bonne exécution du serveur, il suffit d'ouvrir un navigateur et de saisir dans une url la machine hôte et le port d'écoute du connecteur http de Tomcat

Exemple :



If you're seeing this page via a web browser, it means you've setup Tomcat successfully. Congratulations!

As you may have guessed by now, this is the default Tomcat home page. It can be found on the local filesystem at:

```
$CATALINA_HOME/webapps/ROOT/index.jsp
```

where "$CATALINA_HOME" is the root of the Tomcat installation directory. If you're seeing this page, and you don't think you should be, then either you're either a user who has arrived at new installation of Tomcat, or you're an administrator who hasn't got his/her setup quite right. Providing the latter is the case, please refer to the Tomcat Documentation for more detailed setup and administration information than is found in the INSTALL file.

Si Tomcat ne démarre pas :

- consulter les logs pour déterminer l'origine du problème
- lancer Tomcat en utilisant le script Startup.bat dans une console DOS pour afficher les logs
- vérifier que le port utilisé n'est pas déjà utilisé par un autre service ou serveur

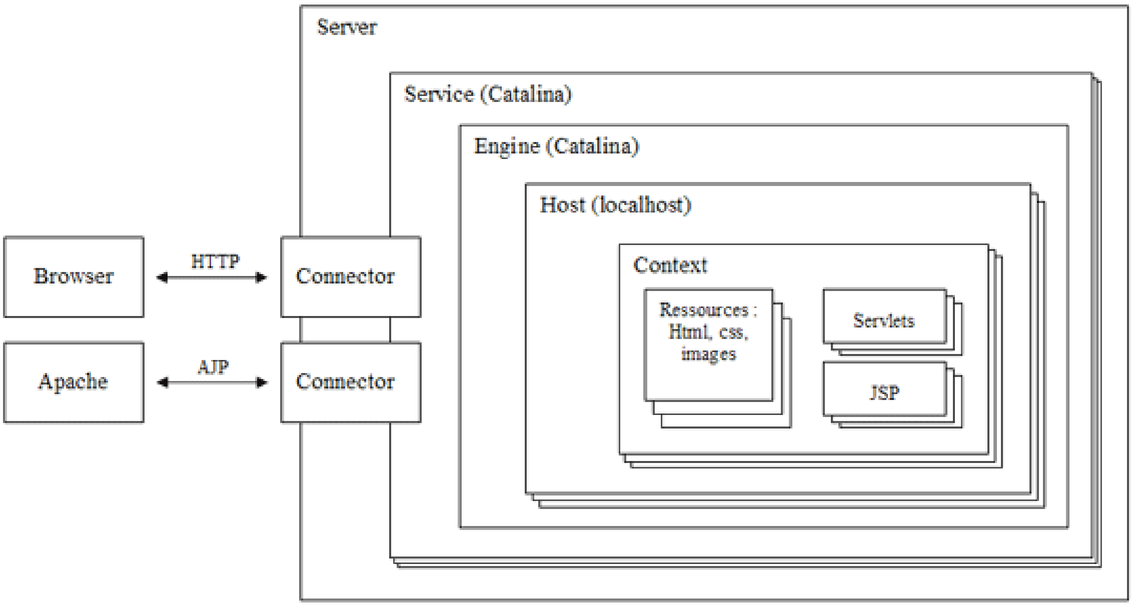
Si Tomcat est lancé mais que la page d'accueil ne s'affiche pas dans le navigateur :

- il faut vérifier l'url saisie (nom de l'hôte et surtout le numéro du port qui doit correspondre à celui configuré dans le fichier server.xml).
- si un proxy est utilisé, inhiber l'utilisation de se dernier pour l'url utilisée notamment en local

102.4. L'architecture

L'architecture de Tomcat est composée de plusieurs éléments :

- Server
  Le serveur encapsule tout le conteneur web. Il ne peut s'exécuter qu'un seul Server dans une JVM
- Service
  Un service regroupe des connectors et un unique engine
- Connector
  Un connector gère les communications avec un client. Tomcat propose plusieurs connecteurs notamment Coyote pour les communications par le protocole http, JK2 pour les communications par le protocole AJP
- Engine
  Un Engine traite les requêtes des différents Connector associés au Service : c'est le moteur de traitements des servlets.
- Host
  Un Host est un nom de domaine dont les requêtes sont traitées par Tomcat. Un Engine peut contenir plusieurs Host.
- Context
  Un contexte permet l'association d'une application web à un chemin unique pour un Host. Un Host peut avoir plusieurs contextes



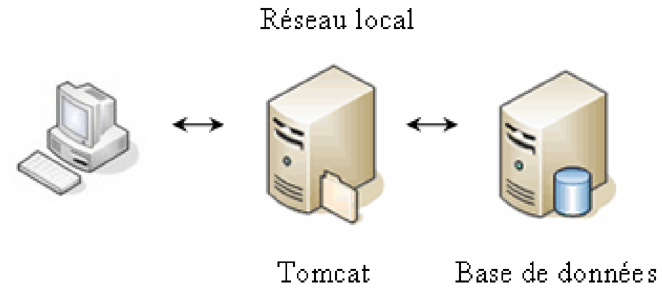
Pour assurer ces fonctionnalités, Tomcat utilise aussi différents types de composants qui prennent en charge des fonctionnalités particulières :

- Valve
  Une valve est une unité de traitements qui est utilisée lors du traitement de la requête. Son rôle est similaire à celui des filtres pour les servlets.
- Logger
  Un Logger assure la journalisation des événements des différents éléments
- Realm
  Un Realm assure l'authentification et les habilitations pour un Engine

102.4.1. Les connecteurs

Qu'il soit utilisé standalone ou en association avec un serveur web, Tomcat doit communiquer avec le monde extérieur.

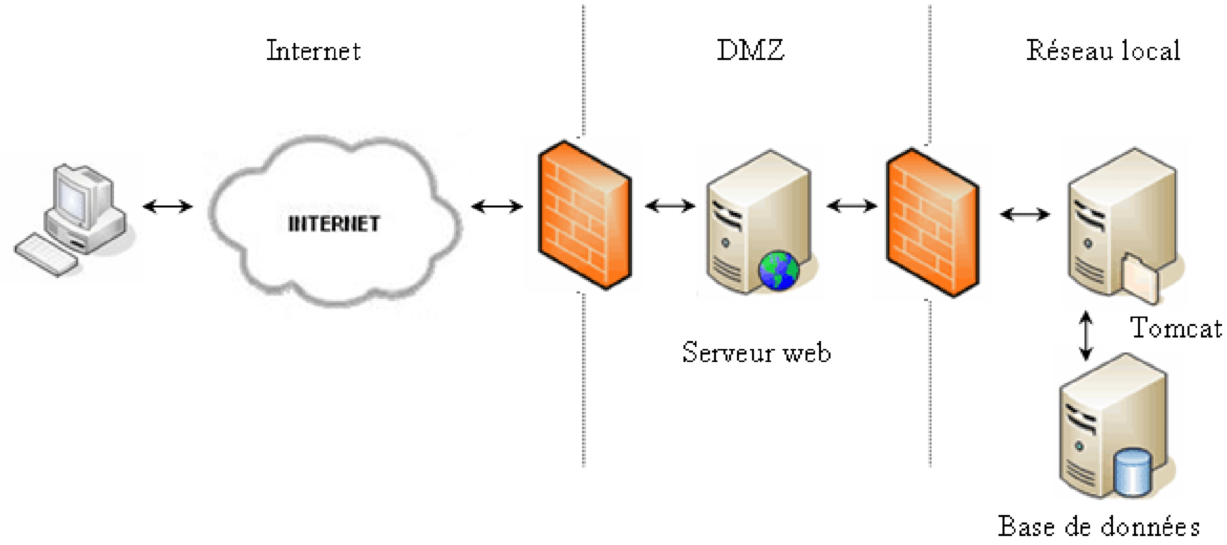
En mode Standalone, le connecteur mis en oeuvre utilise HTTP ou HTTPs pour communiquer.



En association avec un serveur web (Apache par exemple), ce dernier s'occupe des ressources statiques (page HTML, CSS, JavaScript, images,...) et Tomcat s'occupe des ressources dynamiques (JSP, servlets, ...).

Généralement pour des applications à usage externe, l'utilisation d'un serveur web et de Tomcat se fait dans une architecture réseau sécurisée grâce à une DMZ.

Exemple :



Le module mod_jk est utilisé pour assurer la communication entre le serveur Web (Apache par exemple) et Tomcat en utilisant le protocole AJP13.

102.4.2. Les services

Un service regroupe des connecteurs et l'engine. Par défaut Tomcat propose un seul service nommé Catalina.

Plusieurs services peuvent être définis dans un serveur Tomcat.

102.5. La configuration

La configuration de Tomcat est stockée dans plusieurs fichiers dans le sous-répertoire conf. Le fichier de configuration principal est le fichier server.xml.

102.5.1. Le fichier server.xml

Tomcat est configuré grâce à un fichier xml nommé server.xml dans le répertoire conf.

La structure du document xml contenu dans le fichier server.xml reprend la structure de l'architecture de Tomcat :

Exemple :

```
<Server>
 <GlobalNamingResources/>
 <Service>
  <Connector/>
  <Engine>
   <Host/>
  </Engine>
 </Service>
</Server>
```

Remarque : il n'est pas possible d'utiliser un fichier server.xml de Tomcat 4 dans Tomcat 5.

102.5.1.1. Le fichier server.xml avec Tomcat 5

Le tag <Server> est le tag racine du fichier server.xml : il encapsule le serveur Tomcat lui-même. Il possède plusieurs attributs :

Attribut	Rôle
port	port sur lequel Tomcat écoute pour son arrêt (par défaut le port 8005)
shutdown	message à envoyer sur le port pour demander l'arrêt de Tomcat (par défaut SHUTDOWN)

Remarque : Tomcat refuse toute connexion sur le port d'arrêt sauf celle issue de la machine locale (exemple : telnet localhost 8005).

Le tag <Server> peut avoir un unique tag fils <GlobalNamingResources>, au moins un tag fils <Service> et éventuellement plusieurs tags fils <Listener>.

Le tag <GlobalNamingResources/>.
Ce tag encapsule des déclarations de ressources JNDI globales au serveur.

Exemple :

```
<GlobalNamingResources>
 <Environment
  name="simpleValue"
  type="java.lang.Integer"
  value="30"/>
<Resource
 auth="Container"
 description="User database that can be updated and saved"
 name="UserDatabase"
  type="org.apache.catalina.UserDatabase"
  pathname="conf/tomcat-users.xml"
  factory="org.apache.catalina.users.MemoryUserDatabaseFactory"/>
<Resource
 name="jdbc/MonAppDS"
 type="javax.sql.DataSource"
 password=""
  driverClassName="org.apache.derby.jdbc.EmbeddedDriver"
 maxIdle="2"
 maxWait="5000"
 username="APP"
 url="jdbc:derby:C:/Program Files/Java/jdk1.6.0/db/MaBaseDeTest"
 maxActive="4"/>
</GlobalNamingResources>
```

Le tag <Service> encapsule un service. Ce tag possède un seul attribut :

Attribut	Rôle
name	obligatoire : nom identifiant le service (par défaut Catalina)

Plusieurs services peuvent être définis dans un server : dans ce cas, chaque service doit avoir un attribut name distinct.

Le tag <Service> doit avoir au moins un tag fils <Connector> et un unique tag fils <Engine>.

Le tag <Connector/> encapsule un connecteur. Un Connector se charge des échanges entre un client et le serveur pour un protocole donné. Ce tag possède plusieurs attributs :

Attribut	Rôle
className	nom de la classe d'implémentation du connector

protocol	protocole utilisé par le connecteur http ou AJP (par défaut HTTP/1.1). Pour utiliser le protocole AJP, il faut utiliser la valeur AJP/1.3
port	port d'écoute du connector (par défaut 8080 pour le protocole http et 8009 pour le protocole AJP13
redirectPort	sur un connector http, précise le port vers lequel les requêtes HTTPS seront redirigées
minSpareThreads	
maxSpareThreads	
maxThreads	nombre maximum de threads lancés pour traiter les requêtes
secure	positionné à true pour utiliser le protocole HTTPS (par défaut false)
enableLookups	effectue une résolution de l'adresse IP en nom de domaine dans les logs (par défaut true)
proxyName	
proxyPort	
acceptCount	Nombre maximum de requêtes mises en attente si aucun thread n'est libre
connectionTimeout	timeout en millisecondes durant lequel une requête peut rester sans être traitée
disableUploadTimeout	
address	adresse IP des requêtes à traiter

Le tag <Engine> encapsule le moteur de servlet. Ce tag possède plusieurs attributs :

Attribut	Rôle
name	obligatoire : nom identifiant le moteur (par défaut Catalina)
defaultHost	obligatoire : hôte utilisé par défaut si l'hôte de la requête n'est pas défini dans le serveur
jvmRoute	utilisé dans le cadre d'un cluster de serveurs Tomcat pour échanger des informations notamment celles des sessions

Le tag <Engine> doit avoir au moins un tag fils <Host> et peut avoir un tag fils unique <Logger>, <Realm>, <Valve> et <DefaultContext> :

Exemple :

```
<Engine defaultHost="localhost" name="Catalina">
 <Realm ... />
 <Host ... />
</Engine>
```

Le tag <Host> défini un hôte virtuel sur le serveur. Ce tag possède plusieurs attributs :

Attribut	Rôle
name	obligatoire : nom de l'hôte
appBase	obligatoire : chemin par défaut pour les webapps de l'hôte. Ce chemin peut être absolu ou relatif au répertoire d'installation de Tomcat
defaultHost	hôte utilisé par défaut si l'hôte de la requête n'est pas défini dans le serveur
autoDeploy	active le déploiement automatique des webapps. True par défaut

Exemple :

```
  <Host appBase="webapps" name="localhost" autoDeploy="false">
  </Host>
```

Le tag <Context> défini un contexte d'application. Ce tag possède plusieurs attributs, notamment :

Attribut	Rôle
docBase	obligatoire : chemin du répertoire ou de l'archive de l'application. Ce chemin peut être absolu ou relatif à l'attribut appBase du Host
reloadable	demande le rechargement automatique des classes modifiées (par défaut false)

102.5.1.2. Les valves

Une valve est une unité de traitements qui est utilisée lors du traitement de la requête. Son rôle est similaire à celui des filtres pour les servlets.

Une valve est déclarée dans le fichier de configuration grâce au tag <valve> qui peut être utilisé comme fils des tags <engine>, <host> et <context>.

Elle se présente sous la forme d'une classe qui implémente l'interface org.apache.catalina.Valve. Cette classe est précisée dans l'attribut className

Tomcat propose plusieurs Valves par défaut :

Valve	Rôle
org.apache.catalina.valves.AccessLogValve	Configuration des logs
org.apache.catalina.authenticator.SingleSignOn	Utilisation du SSO

102.5.2. La gestion des rôles

Par défaut dans Tomcat, les rôles sont définis dans le fichier tomcat-users.xml. Ce fichier permet de définir des rôles et de les associer à des utilisateurs.

La modification du fichier tomcat-users.xml peut se faire directement sur le fichier ou par l'application d'administration.

Exemple :

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
 <role rolename="tomcat"/>
 <role rolename="manager"/>
 <role rolename="test"/>
 <role rolename="admin"/>
 <user username="tomcat" password="tomcat" roles="tomcat"/>
 <user username="jm" password="jm" roles="test"/>
 <user username="admin" password="baron" roles="admin,manager"/>
</tomcat-users>
```

Remarque : pour utiliser l'application d'administration ou le manager, il est nécessaire de définir les utilisateurs admin et manager.

102.6. L'outil Tomcat Administration Tool

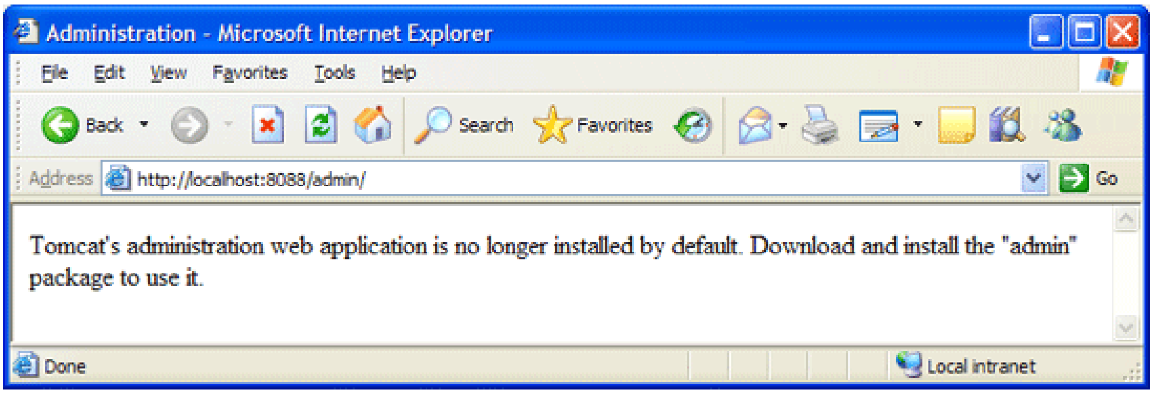
Tomcat Administration Tool est une application web qui permet de faciliter la configuration de Tomcat : il permet de modifier certains fichiers de configuration au moyen d'une interface graphique.

Remarque : cet outil n'est plus disponible à partir de Tomcat 6.

Il permet notamment de gérer :

- Les connecteurs et les ports
- Les contextes d'applications
- Les ressources
- La sécurité
- Les utilisateurs

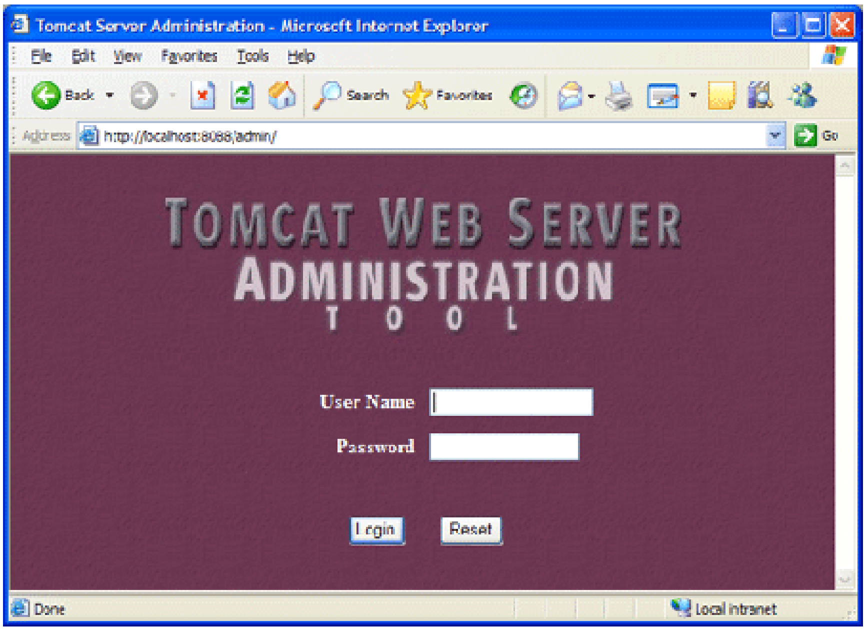
Remarque : à partir de Tomcat 5.5, cet outil n'est plus fourni en standard avec Tomcat et doit être téléchargé séparément et installé.


Administration - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:8088/admin/
Go
Done
Local intranet

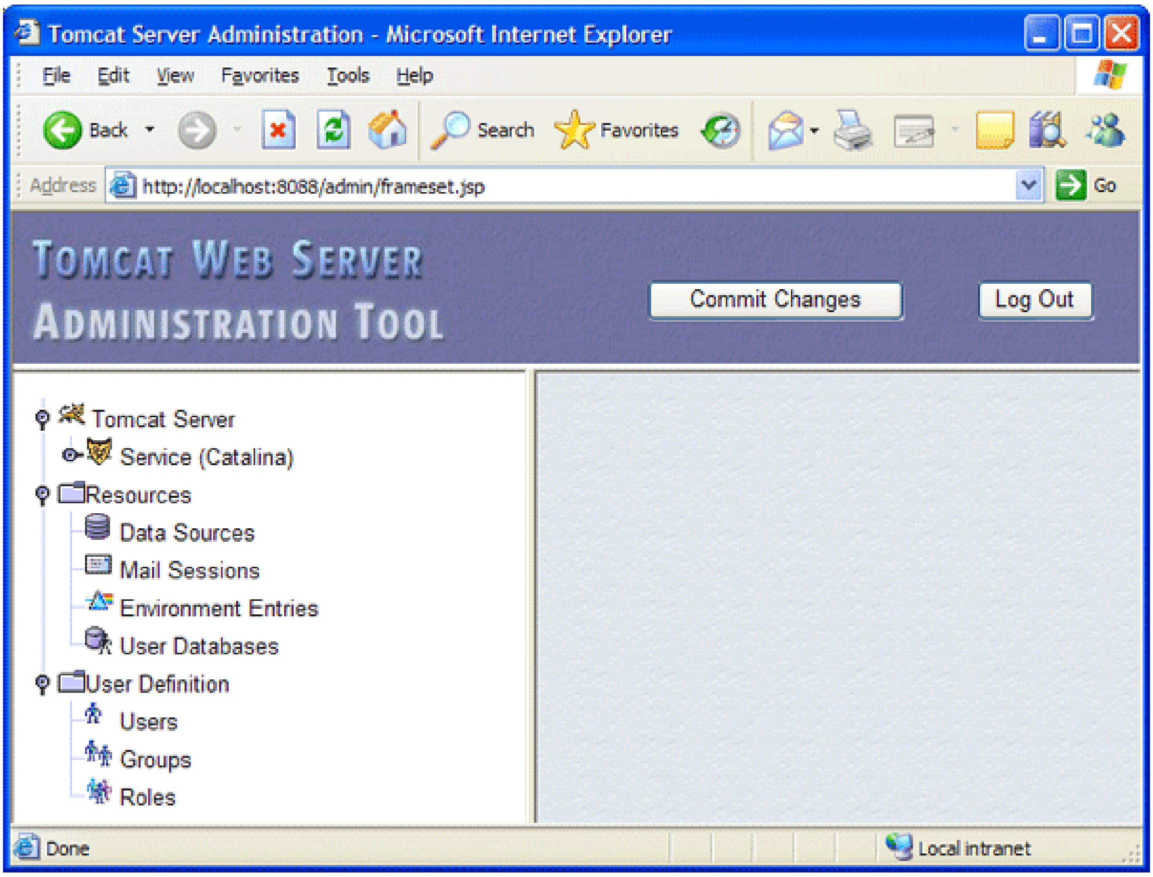
Tomcat's administration web application is no longer installed by default. Download and install the "admin" package to use it.

Dans ce cas, il faut télécharger le module et le décompresser dans le répertoire d'installation de Tomcat après avoir arrêté le serveur Tomcat.

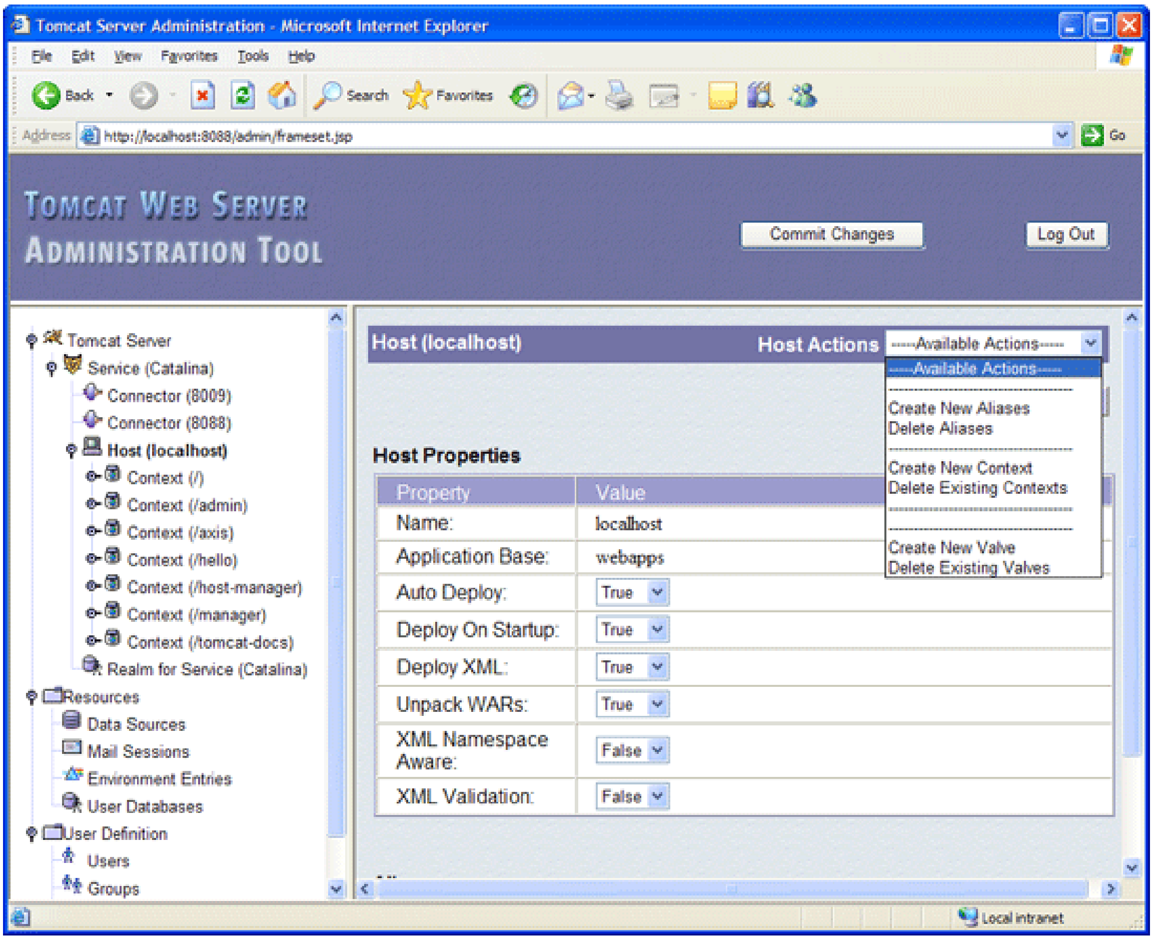
Pour lancer l'application, il suffit de cliquer sur le lien « Tomcat Administration » sur la page d'accueil de Tomcat


Tomcat Server Administration - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:8088/admin/
Go
TOMCAT WEB SERVER
ADMINISTRATION
TOOL
User Name
Password
Login
Reset
Done
Local intranet

Il faut saisir le user et le mot de passe définis pour le user admin (informations fournies au programme d'installation sous Windows)



Les différents éléments configurables sont présentés sous une forme arborescente dans la partie de gauche. La partie de droite permet de modifier les données de l'élément sélectionné. La liste déroulante « Actions » permet de réaliser des actions en fonction du contexte (création d'éléments, suppression, ...). Le bouton « Save » permet d'enregistrer les modifications en locale : il faut l'utiliser avant de changer de page.



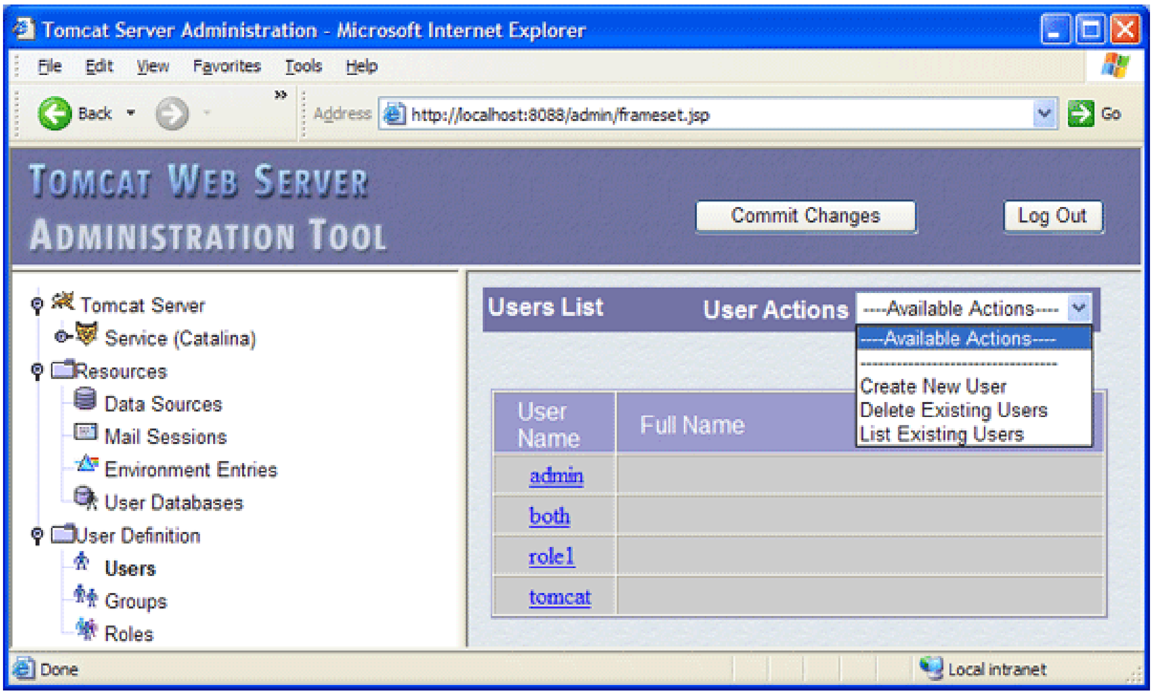
Host Properties

Property	Value
Name:	localhost
Application Base:	webapps
Auto Deploy:	True
Deploy On Startup:	True
Deploy XML:	True
Unpack WARs:	True
XML Namespace Aware:	False
XML Validation:	False

Attention : Il est très important pour valider les modifications de cliquer sur le bouton « Commit changes » : cette action rend persistantes les modifications en les écrivant dans les fichiers de configuration de Tomcat.

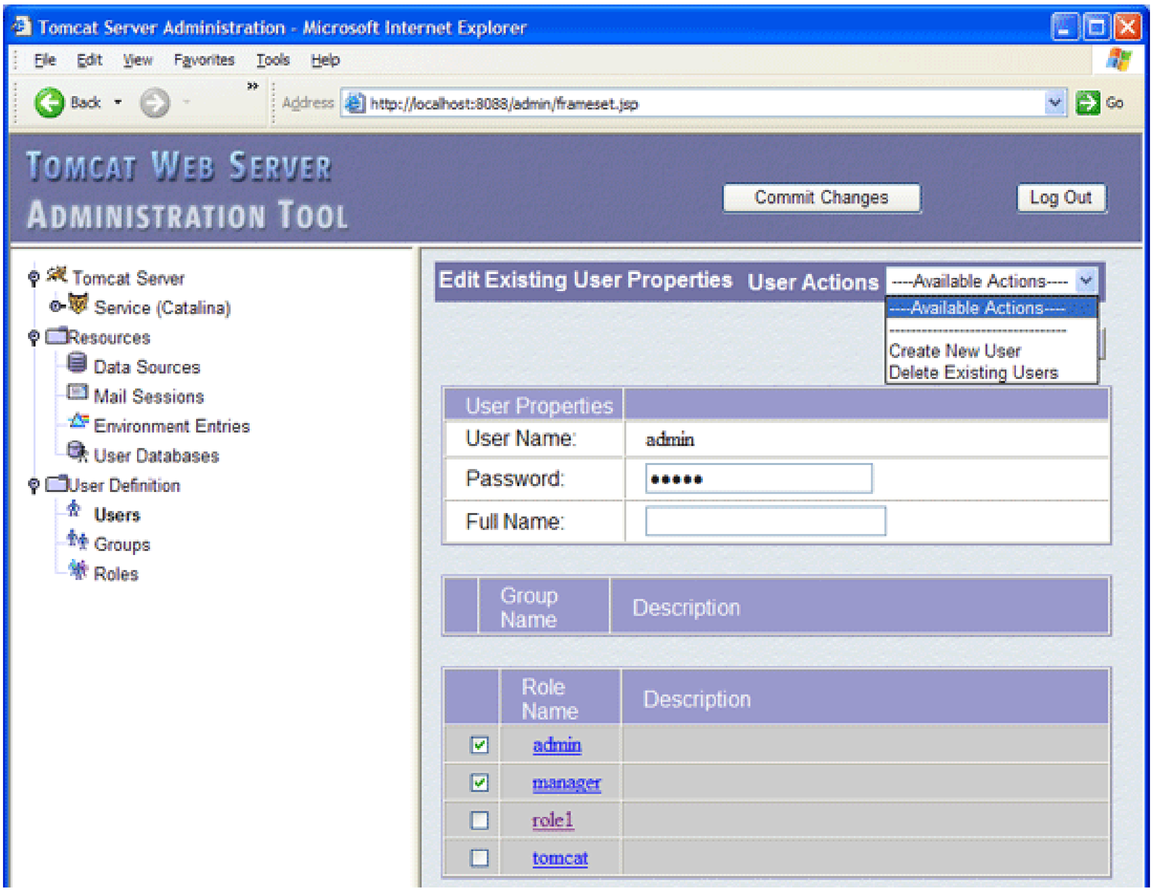
102.6.0.1. La gestion des utilisateurs, des rôles et des groupes

La partie « User Definition » de l'outil d'administration permet de gérer les users, les rôles et les groupes sans avoir à modifier directement le contenu du fichier tomcat-users.xml.



User Name	Full Name
admin	
both	
role1	
tomcat	

Pour modifier un user, il suffit de cliquer sur son lien.


Tomcat Server Administration - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Address http://localhost:8088/admin/frameset.jsp
Go
Tomcat Web Server Administration Tool
Commit Changes
Log Out
Tomcat Server
Service (Catalina)
Resources
Data Sources
Mail Sessions
Environment Entries
User Databases
User Definition
Users
Groups
Roles
Edit Existing User Properties
User Actions
----Available Actions----
----Available Actions----
Create New User
Delete Existing Users

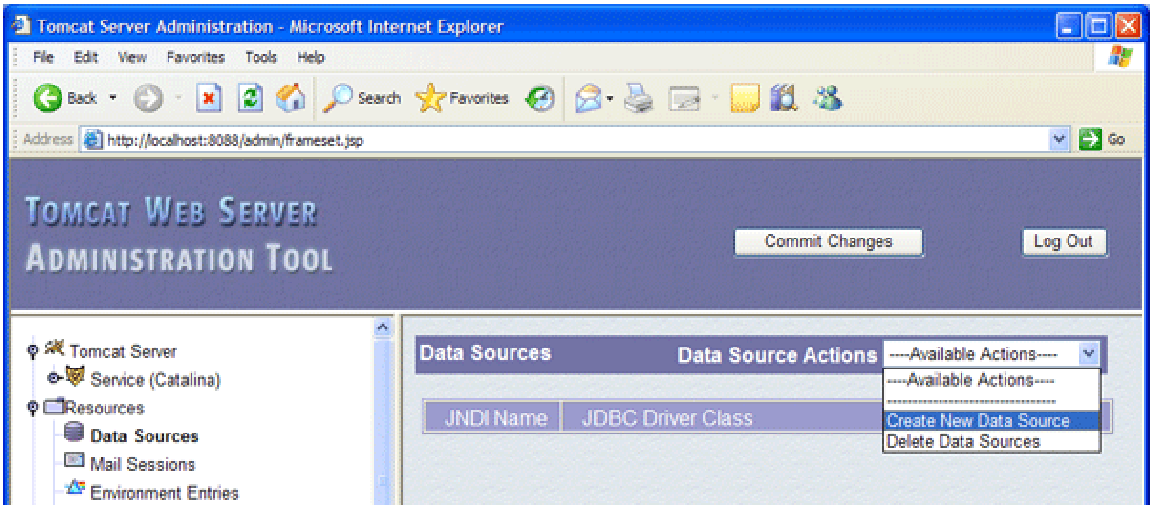
User Properties	
User Name:	admin
Password:	•••••
Full Name:	

	Group Name	Description

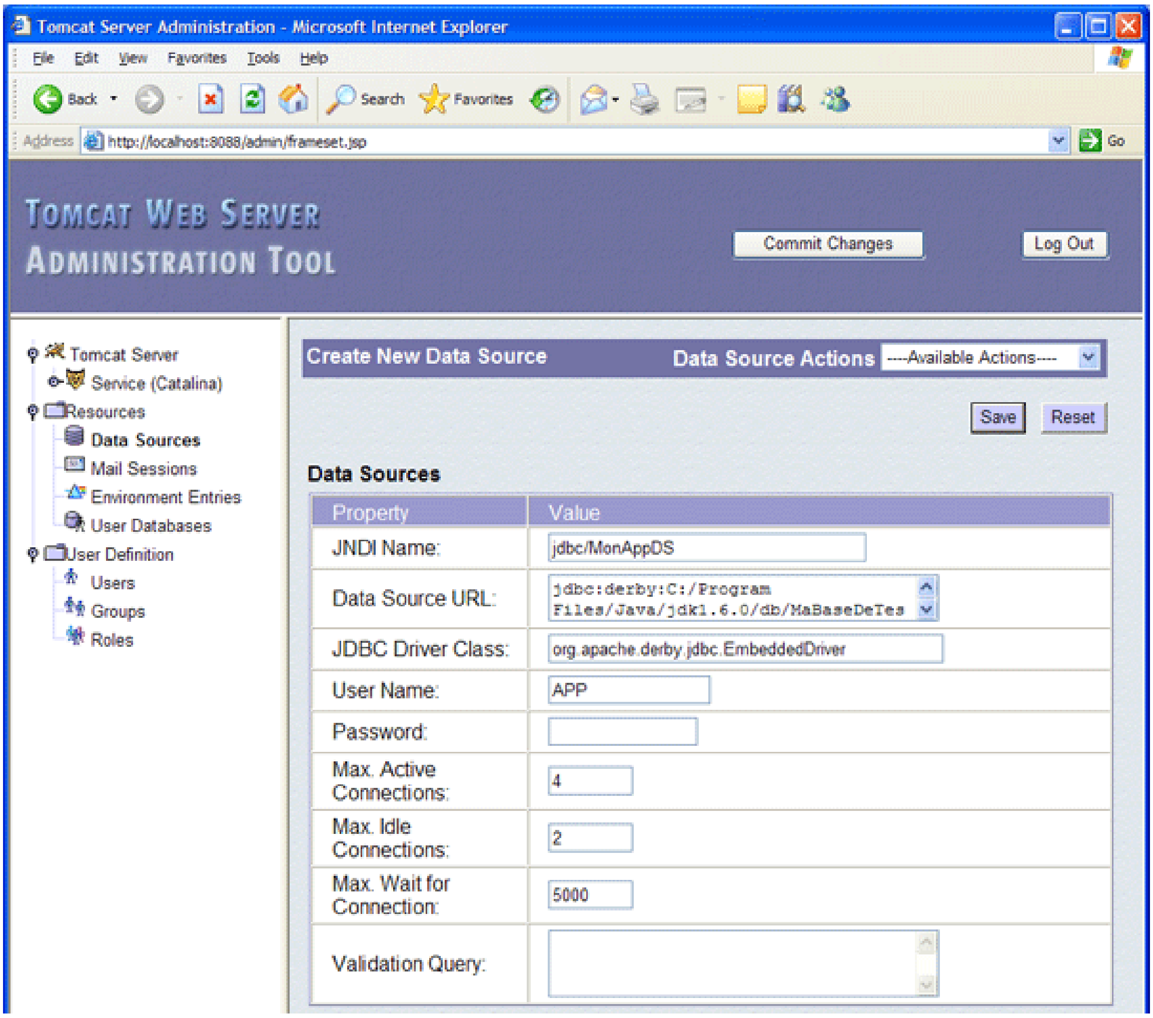
	Role Name	Description
☑	admin	
☑	manager	
□	role1	
□	tomcat	

102.6.1. La création d'une DataSource dans Tomcat

Le plus simple est d'utiliser l'application d'administration.



Cliquez sur « Data Sources » dans l'arborescence « Resources ». Dans la liste déroulante « DataSource Actions », sélectionnez « Create New Data Source ».



Data Sources

Property	Value
JNDI Name:	jdbc/MonAppDS
Data Source URL:	jdbc:derby:C:/Program Files/Java/jdk1.6.0/db/MaBaseDeTes
JDBC Driver Class:	org.apache.derby.jdbc.EmbeddedDriver
User Name:	APP
Password:	
Max. Active Connections:	4
Max. Idle Connections:	2
Max. Wait for Connection:	5000
Validation Query:	

Saisissez les informations nécessaires à la Datasource : le nom JNDI, l'url, la classe du pilote, le user, le mot de passe, ...

Cliquez sur le bouton « Save » puis sur « Commit changes » et confirmez la validation des modifications.

Dans le fichier de configuration de Tomcat server.xml, la Datasource a été ajoutée dans les ressources de GlobalNamingResources :

Exemple :

```
<GlobalNamingResources>
...
 <Resource
  name="jdbc/MonAppDS"
  type="javax.sql.DataSource"
  password=""
   driverClassName="org.apache.derby.jdbc.EmbeddedDriver"
  maxIdle="2"
  maxWait="5000"
  username="APP"
  url="jdbc:derby:C:/Program Files/Java/jdk1.6.0/db/MaBaseDeTest"
  maxActive="4"/>
 </GlobalNamingResources>
```

102.7. Le déploiement des applications WEB

Pour être exécutée, une application web doit impérativement être déployée dans un conteneur de servlets même dans un environnement de développement.

Selon les spécifications des servlets depuis la version 2.2, un conteneur doit obligatoirement être capable de déployer une application web au format war. Tomcat propose aussi un support pour déployer les applications au format unpacked et propose différentes solutions pour assurer le déploiement des applications.

102.7.1. Déployer une application web avec Tomcat 5

Une application web peut être déployée sous Tomcat 5 de plusieurs manières :

- copier l'application dans le répertoire webapps
- définir un contexte
- utiliser l'outil Tomcat Manager
- utiliser les tâches Ant du Manager
- utiliser l'outil TCD

102.7.1.1. Déployer une application au lancement de Tomcat

Alors que Tomcat est arrêté, il suffit de copier le répertoire contenant la webapp ou le fichier war qui la contient dans le sous-répertoire webapps de Tomcat et de redémarrer ce dernier.

Par défaut, l'uri de l'application utilisera le nom du répertoire ou du fichier war : Tomcat va créer un contexte pour l'application en lui associant comme chemin de contexte le nom du répertoire ou du fichier war sans son extension.

Par défaut, Tomcat décompresse le contenu d'un fichier war dans un répertoire portant le nom du fichier war sans son extension.

Pour redéployer une application sous la forme d'un fichier war, il est préférable de supprimer le répertoire contenant l'application décompressée.

Les applications du répertoire webapps sont automatiquement déployées au démarrage si l'attribut deployOnStartUp du tag Host vaut true.

102.7.1.2. Déployer une application sur Tomcat en cours d'exécution

Si l'attribut autoDeploy du tag Host vaut true, le déploiement de l'application par copie dans le répertoire webapps peut se faire alors que Tomcat est en cours d'exécution. Ce mécanisme permet aussi de recharger dynamiquement une application.

Remarque : Tomcat propose des fonctionnalités de rechargement dynamique d'une application ayant subi des modifications : il est cependant préférable de redémarrer le serveur Tomcat pour éviter certains écueils.

102.7.1.3. L'utilisation d'un contexte

Un descripteur de contexte est un document au format xml qui contient la définition d'un contexte.

Ce descripteur permet de configurer le contexte.

Ce fichier doit être placé dans le sous-répertoire /conf/{engine_name}/{host_name} où {engine_name} est le nom du moteur et {host_name} est le nom de l'hôte.

Avec la configuration par défaut de Tomcat, c'est le sous-répertoire /conf/catalina/localhost.

Le contenu de ce descripteur de contexte est détaillé dans une des sections suivantes de ce chapitre.

102.7.1.4. Déployer une application avec le Tomcat Manager

Tomcat fournit l'application web Tomcat Manager pour permettre la gestion des applications web exécutées sur le serveur sans avoir à procéder à un arrêt/redémarrage de Tomcat.

Son utilisation est détaillée dans une des sections suivantes de ce chapitre.

102.7.1.5. Déployer une application avec les tâches Ant du Manager

Tomcat propose des tâches Ant qui permettent l'utilisation dans des scripts de certaines fonctionnalités du Manager.

L'utilisation de ces tâches Ant est détaillée dans la documentation de Tomcat.

102.7.1.6. Déployer une application avec le TCD

L'outil TCD (Tomcat Client Deployer) permet de packager une application et de gérer son cycle de vie dans le serveur Tomcat. Cet outil utilise les tâches Ant du Manager.

Son utilisation est détaillée dans une des sections suivantes de ce chapitre.

102.8. Tomcat pour le développeur

102.8.1. Accéder à une ressource par son url

Une url permet d'accéder aux ressources statiques et dynamiques d'une application web. Par exemple dans une application contenue dans le sous-répertoire maWebApp du répertoire webapps de Tomcat, l'accès aux ressources se fera avec une url de la forme :

http://host[:port]/webapp/chemin/ressource

Exemple :

http://localhost:8080/maWebApp

Pour une ressource statique, il suffit de préciser le chemin dans la webapp et le nom de la ressource.

Exemple : pour le fichier index.htm à la racine de la webapp.

http://localhost:8080/maWebApp/index.htm

Exemple : pour le fichier index.htm dans le sous-répertoire admin de la webapp.

http://localhost:8080/maWebApp/admin/index.htm

Pour les ressources dynamiques de type servlet, le chemin et la ressource doivent correspondre au mapping qui est fait entre la classe et l'url dans le fichier de configuration web.xml.

Exemple : mapping de la servlet fr.jmdoudouxmawebapp.AfficherListeServ vers l'url AfficherListe

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">
 <servlet>
   <servlet-name>AffListe</servlet-name>
   <servlet-class>fr.jmdoudouxmawebapp.AfficherListeServ</servlet-class>
 </servlet>
 <servlet-mapping>
   <servlet-name>AffListe</servlet-name>
   <url-pattern>/AfficherListe</url-pattern>
 </servlet-mapping>
</web-app>
```

Exemple : appel de la servlet AfficherListeServ.

http://localhost:8080/maWebApp/AfficherListe

Attention : Tomcat 5 vérifie le fichier de configuration de chaque webapp (WEB-INF/web.xml) et impose que l'ordre des tags dans ce fichier corresponde à celui défini dans la DTD du fichier web.xml.

102.8.2. La structure d'une application web et format war

Depuis la spécification 2.2 des servlets, le contenu d'une webapp doit obligatoirement respecter une certaine structure pour organiser ses répertoires et ses fichiers :

- les ressources statiques sont stockées à la racine de la webapp ou dans un de ses sous-répertoires autre que les répertoires WEB-INF et META-INF. Ces deux répertoires ne sont accessibles que par le conteneur
- le fichier de configuration server.xml est stocké dans le répertoire WEB-INF
- les ressources utilisées par le conteneur doivent être le répertoire WEB-INF et, notamment, son sous-répertoire classes (pour les classes comme les servlets et les ressources associées comme les fichiers .properties) et son sous-répertoire lib pour les bibliothèques (pilote JDBC, API, framework, ...)

Le format war est physiquement une archive de type zip qui englobe le contenu de la webapp.

Pour déployer une webapp dans Tomcat, il suffit de copier le répertoire de la webapp (forme unpacked) ou son fichier war (forme packed) dans le sous-répertoire webapps.

Il est aussi possible de définir un contexte dont l'attribut docbase a pour valeur un répertoire quelconque du système de fichiers. Il est alors possible de développer l'application en dehors de Tomcat et d'utiliser ce répertoire de développement comme répertoire de déploiement.

Les classes et bibliothèques contenues dans les répertoires WEB-INF/classes et WEB-INF/lib sont utilisables par les classes de l'application.

102.8.3. La configuration d'un contexte

Un contexte est défini pour chaque application web exécutée sur le serveur soit explicitement dans la configuration soit implicitement avec un contexte par défaut créé par Tomcat.

102.8.3.1. La configuration d'un contexte avec Tomcat 5

Un contexte peut être défini explicitement de plusieurs manières :

- dans le fichier conf/server.xml (depuis la version 5 de Tomcat, cette utilisation n'est pas recommandée)
- dans un fichier xml de configuration du contexte
- dans le fichier META-INF/context.xml de la webapp

Les contextes peuvent être modifiés manuellement en modifiant le fichier de configuration adéquat ou en utilisant l'outil d'administration de Tomcat

Un contexte est défini grâce à un tag <Context> qui possède plusieurs attributs :

- path : précise le chemin de contexte de l'application. Il doit commencer par un / et être unique pour chaque hôte
- docbase : précise le chemin absolu ou relatif au sous-répertoire webapps du répertoire de l'application web ou le chemin de son fichier .war
- reloadable : précise si l'application doit être rechargée automatiquement en cas de modification dans les sous-répertoires WEB-INF/Classes et WEB-INF/lib
- crossContext : précise si l'application peut avoir accès au contexte des autres applications exécutées en utilisant la méthode getContext() de la classe ServletContext (false par défaut).
- cookies : précise si les cookies sont utilisés pour échanger l'id de session (true par défaut). Pour forcer l'utilisation de la réécriture d'url, il faut donner la valeur false à cet attribut
- privileged : précise si l'application peut avoir accès aux servlets du conteneur

L'implémentation par défaut de l'interface Context fournie avec Tomcat (**org.apache.catalina.core.StandardContext**) propose plusieurs attributs supplémentaires dont :

- workdir : précise le répertoire temporaire de travail
- unpackWar : précise si le fichier war doit être décompressé (true par défaut)

Un fichier de configuration du contexte peut être défini dans le répertoire conf/nom_engine/nom_hôte/. Son nom sera utilisé (dans son extension .xml) par défaut comme chemin de contexte.

La définition d'un contexte est par exemple utilisée par Sysdeo dans son plug-in Eclipse pour faciliter l'utilisation de Tomcat.

102.8.4. L'invocation dynamique de servlets

Tomcat propose une fonctionnalité particulière nommée « Invoker servlet » qui permet l'appel d'une servlet sans que celle-ci soit déclarée dans un fichier web.xml.

Cette fonctionnalité peut être pratique dans un environnement de développement. Il ne faut pas l'utiliser pour d'autre besoin que celui de tests, surtout, elle ne doit pas être activée en production.

Pour activer cette fonctionnalité, il faut décommenter la déclaration de la servlet Invoker et son mapping dans le fichier de configuration par défaut des applications web. Ce fichier est le fichier /conf/web.xml.

La déclaration est faite par le tag <servlet>.

Exemple :

```
 <servlet>
    <servlet-name>invoker</servlet-name>
   <servlet-class>
    org.apache.catalina.servlets.InvokerServlet
   </servlet-class>
   <init-param>
      <param-name>debug</param-name>
      <param-value>0</param-value>
   </init-param>
    <load-on-startup>2</load-on-startup>
 </servlet>
```

Le mapping est fait par le tag <servlet-mapping>

Exemple :

```
  <servlet-mapping>
    <servlet-name>invoker</servlet-name>
    <url-pattern>/servlet/*</url-pattern>
  </servlet-mapping>
```

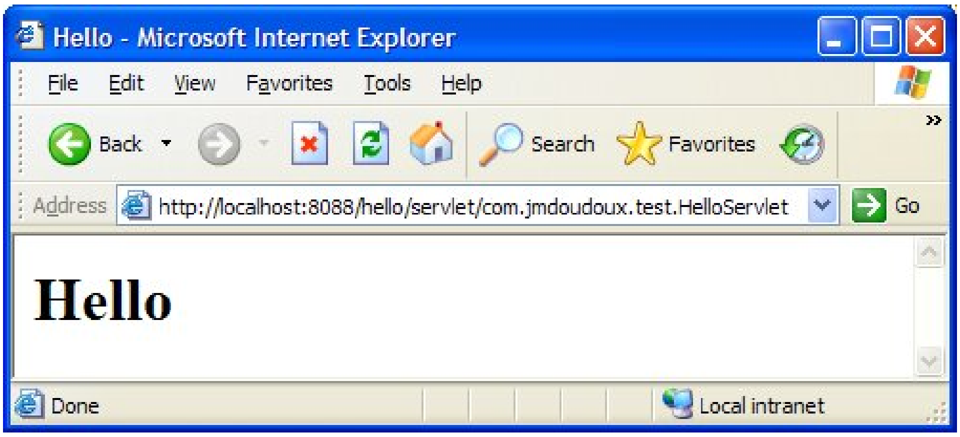
Il faut enregistrer le fichier modifié et redémarrer Tomcat.

Il suffit d'écrire le code de la servlet, de la compiler et de mettre le fichier .class correspondant dans le sous-répertoire WEB-INF/Classes d'une webapp.

Exemple :

```
package fr.jmdoudoux.dej;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class HelloServlet extends javax.servlet.http.HttpServlet implements
   javax.servlet.Servlet {

  public HelloServlet() {
   super();
  }
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
   response.setContentType("text/html");
   PrintWriter out = response.getWriter();
   out.println("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 "
     + "Transitional//EN\">\n"
     + "<HTML>\n"
     + "<HEAD><TITLE>Hello</TITLE></HEAD>\n"
     + "<BODY>\n"
     + "<H1>Hello</H1>\n"
     + "</BODY></HTML>");
  }
 }
```

Pour lancer la servlet, il suffit d'ouvrir l'url http://host:port/webapp/servlet/nom_pleinement_qualifié_de_la_classe


Hello - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
Address
Go
Done
Local intranet

http://localhost:8088/hello/servlet/com.jmdoudoux.test.HelloServlet

Hello

Attention : avec Tomcat 6, il est nécessaire de positionner l'attribut privileged à true pour le contexte de l'application.

102.8.5. Les bibliothèques partagées

Tomcat propose une solution pour partager des bibliothèques communes à toutes les applications qui s'exécutent sur le serveur.

Attention l'utilisation de cette fonctionnalité est spécifique à Tomcat. Il est en général préférable de mettre les bibliothèques dans le répertoire WEB-INF/lib de chaque application. Les bibliothèques sont dupliquées dans chaque application mais cela permet de rendre les applications moins dépendantes de Tomcat en plus d'offrir à chaque application la possibilité d'utiliser une version de bibliothèque différente.

102.8.5.1. Les bibliothèques partagées sous Tomcat 5

Deux répertoires sont fournis à cet effet :

- Le sous-répertoire common/lib : les bibliothèques sont partagées avec Tomcat et toutes les applications
- Le sous-répertoire shared/lib : les bibliothèques sont partagées par toutes les applications mais ne sont pas accessibles pour Tomcat

Des répertoires nommés classes permettent de façon similaire de partager des classes non regroupées dans une archive (jar ou zip).

Par défaut, Tomcat 5 fournit plusieurs bibliothèques partagées notamment celles des servlets, JSP et EL utilisables par toutes les webapp qu'il exécute. Ces API sont dans le répertoire "common/lib" ou "shared/lib" de Tomcat :

- ant.jar (Apache Ant 1.6)
- commons-collections*.jar (Commons Collections 2.1)
- commons-dbcp.jar (Commons DBCP 1.1)
- commons-el.jar (Commons Expression Language 1.0)
- commons-logging-api.jar (Commons Logging API 1.0.3)
- commons-pool.jar (Commons Pool 1.1)
- jasper-compiler.jar
- jsp-api.jar (JSP 2.0)
- commons-el.jar (JSP 2.0 EL)
- naming-common.jar
- naming-factory.jar
- naming-resources.jar
- servlet-api.jar (Servlet 2.4)

102.9. Le gestionnaire d'applications (Tomcat manager)

Tomcat fournit une application web pour permettre la gestion des applications web exécutées sur le serveur sans avoir à procéder à un arrêt/redémarrage de Tomcat.

Cette application permet de :

- Lister les applications déployées avec leur état et le nombre de sessions ouvertes
- Déployer une nouvelle application (deploy)
- Arrêter (stop), démarrer (start) et recharger une application (reload)
- Supprimer une application (undeploy)
- Obtenir des informations sur la JVM et l'OS

L'application Manager peut être utilisée de trois manières :

- Utilisation de l'interface graphique est associée au contexte /manager. Elle peut être lancée en utilisant le lien « Tomcat Manager » sur la page d'accueil de Tomcat ou en utilisant l'uri /manager/html
- Utilisation de requêtes http : les opérations sont fournies dans la requête. Cette solution permet son utilisation dans des scripts
- Utilisation de tâches Ant

102.9.1. L'utilisation de l'interface graphique

Le manager de Tomcat est un outil web de Tomcat qui permet de gérer les applications exécutées sous Tomcat. Elle est fournie en standard lors de l'installation de Tomcat.

L'utilisation du manager est soumise à une authentification préalable avec un utilisateur possédant le rôle de manager. Ceci est configuré dans le fichier /conf/tomcat-users.xml.

Par défaut, aucun utilisateur ne possède ce rôle : il est donc nécessaire de l'ajouter.

Sous Windows, avec le programme d'installation, l'utilisateur saisi est associé aux rôles admin et manager.

Par défaut, Tomcat utilise un MemoryRealm pour l'authentification. Dans ce cas, pour ajouter ou modifier les utilisateurs et leurs rôles, il faut modifier le fichier /conf/tomcat-users.xml.

Le tag <role> permet de définir un rôle. Par exemple, il faut ajouter le rôle manager si ce dernier n'est pas défini.

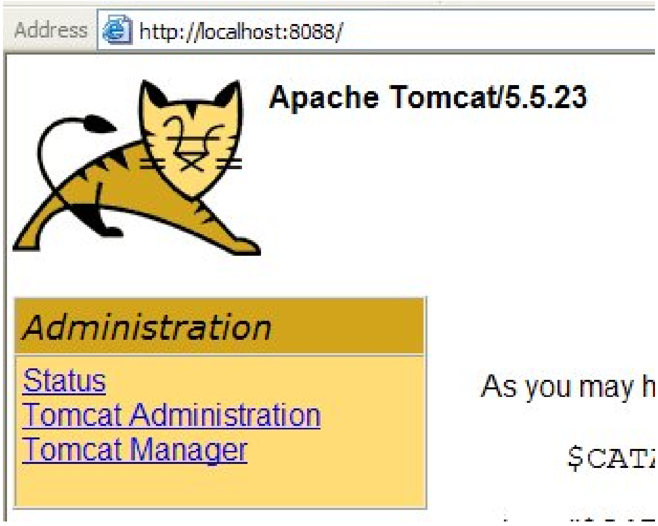
Le tag <user> permet de déclarer un utilisateur en précisant son nom avec l'attribut username, son mot de passe avec l'attribut password et en lui associant un ou plusieurs rôles avec l'attribut roles. Plusieurs rôles peuvent être donnés à un utilisateur en les séparant chacun par une virgule.

Exemple :

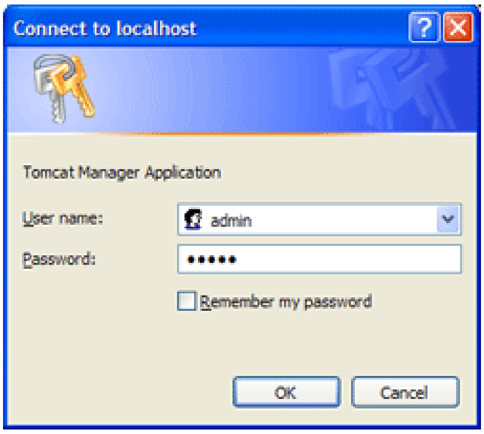
```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
 <role rolename="tomcat"/>
 <role rolename="role1"/>
 <role rolename="manager"/>
 <role rolename="admin"/>
 <user username="tomcat" password="tomcat" roles="tomcat"/>
 <user username="both" password="tomcat" roles="tomcat,role1"/>
 <user username="role1" password="tomcat" roles="role1"/>
 <user username="admin" password="admin" roles="admin,manager"/>
</tomcat-users>
```

Tous les utilisateurs qui possèdent le rôle manager peuvent employer l'application Manager.

Il faut ouvrir un navigateur sur l'url du serveur Tomcat.


Address http://localhost:8088/

Cliquez sur le lien « Tomcat Manager »


Connect to localhost

Tomcat Manager Application

User name: admin

Password: •••••

□ Remember my password

Une boîte de dialogue demande l'authentification d'un utilisateur ayant un rôle de type manager. Dans l'installation par défaut, le user admin possède les rôles manager et admin.

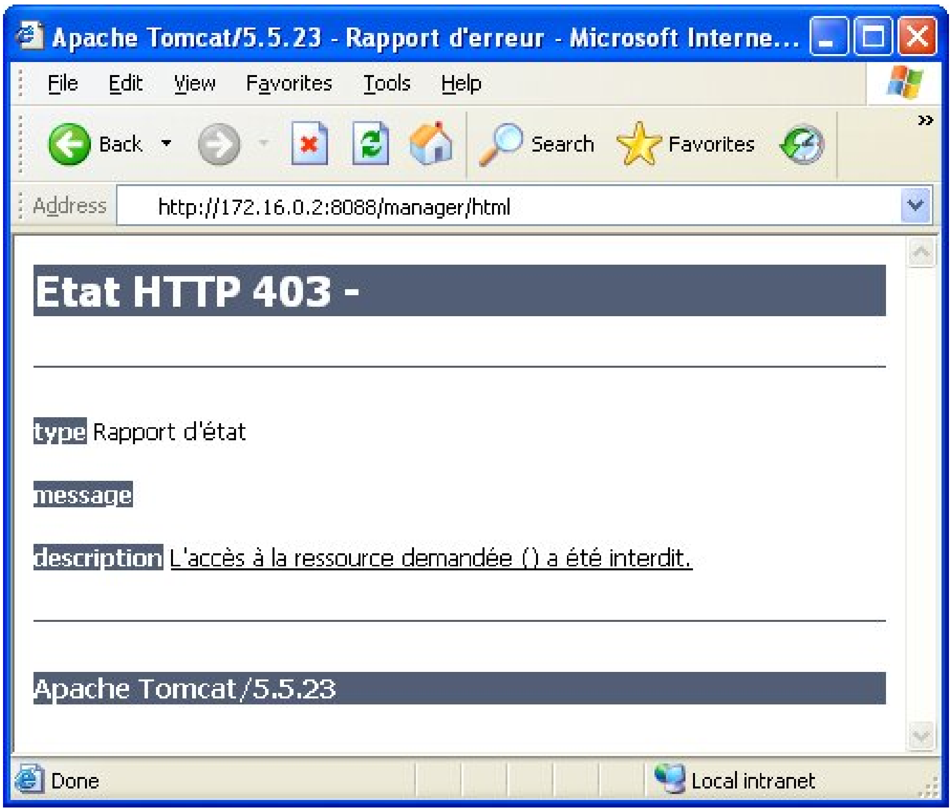
Remarque : les informations concernant les utilisateurs sont par défaut dans le répertoire $CATALINA_HOME/conf/tomcat-users.xml.

Il est possible d'utiliser une valve pour restreindre l'accès au Tomcat Manager en fonction de l'adresse IP ou du nom d'hôte de la machine.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<Context
  docBase="C:/Program Files/Apache/Tomcat 5.5/server/webapps/manager"
  privileged="true">
...
 <Valve className="org.apache.catalina.valves.RemoteAddrValve"
  allow="127.0.0.1"/>
</Context>
```

Si une machine non référencée tente d'accéder à l'application, un message d'erreur est affiché :


Apache Tomcat/5.5.23 - Rapport d'erreur - Microsoft Interne...
File Edit View Favorites Tools Help
Back
Search
Favorites
Address
http://172.16.0.2:8088/manager/html
Done
Local intranet

Etat HTTP 403 -

type Rapport d'état

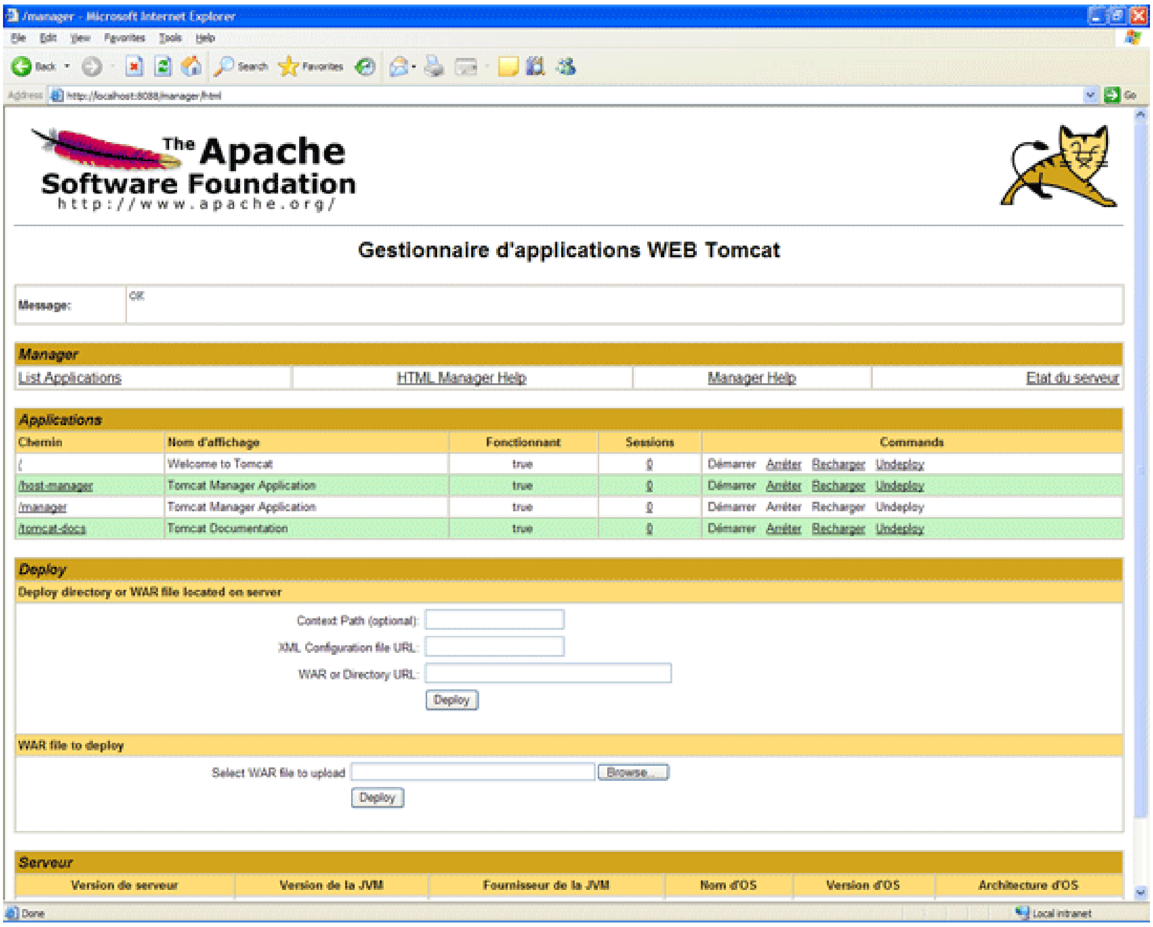
message

description L'accès à la ressource demandée () a été interdit.

Apache Tomcat/5.5.23

Ceci permet de renforcer la sécurité notamment en production.

La page principale de l'application est composée de plusieurs parties.


/manager - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
Address
http://localhost:8088/manager/html
Go
The Apache Software Foundation
http://www.apache.org/
Done
Local intranet

Gestionnaire d'applications WEB Tomcat

Message: OK

Manager

List Applications | HTML Manager Help | Manager Help | Etat du serveur

Applications

Chemin	Nom d'affichage	Fonctionnant	Sessions	Commands
/	Welcome to Tomcat	true	0	Démarrer Arrêter Recharger Undeploy
/host-manager	Tomcat Manager Application	true	0	Démarrer Arrêter Recharger Undeploy
/manager	Tomcat Manager Application	true	0	Démarrer Arrêter Recharger Undeploy
/tomcat-docs	Tomcat Documentation	true	0	Démarrer Arrêter Recharger Undeploy

Deploy

Deploy directory or WAR file located on server

Context Path (optional):

XML Configuration file URL:

WAR or Directory URL:

Deploy

WAR file to deploy

Select WAR file to upload Browse...

Deploy

Serveur

Version de serveur	Version de la JVM	Fournisseur de la JVM	Nom d'OS	Version d'OS	Architecture d'OS

La partie applications affiche la liste des applications déployées et permet de les gérer.

La partie Serveur affiche quelques informations sur les versions de Tomcat, de la JVM et de l'OS d'exécution.

La partie Deploy permet de déployer une application web soit à partir d'éléments sur le serveur ou sur le poste client.

102.9.1.1. Le déploiement d'une application

La partie « Deploy directory or WAR File located on Server » permet de déployer une application se trouvant déjà sur la machine.

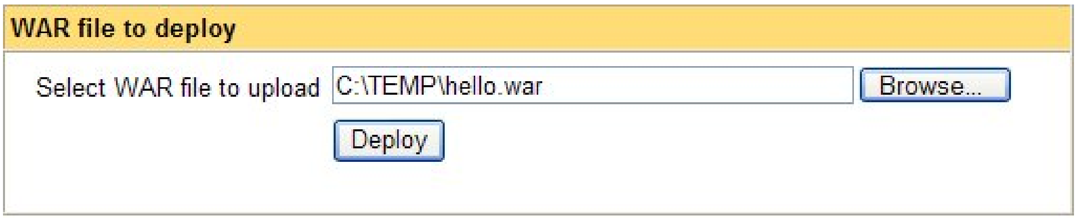
Il faut fournir trois données :

- le chemin du contexte, exemple : /MaWebApp
- le nom du fichier web.xml : web.xml en général
- le chemin où se situe l'application, exemple : C:\java\projet\MaWebApp

Puis cliquer sur le bouton « Deploy ».

La partie « War file to deploy » permet de sélectionner un fichier de type war du poste client, de le télécharger sur le serveur, de le déployer dans Tomcat et de démarrer l'application.

Exemple :

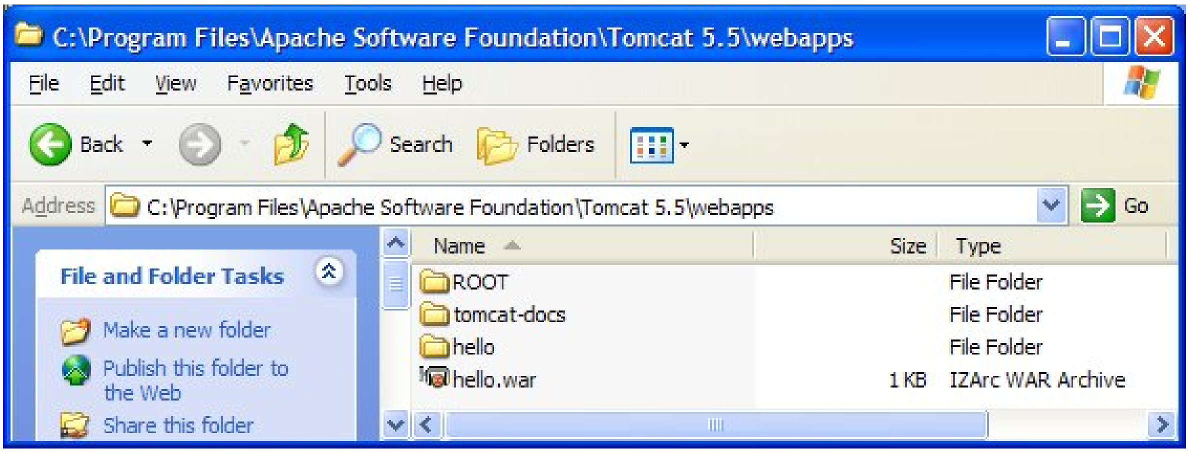


WAR file to deploy

Select WAR file to upload: C:\TEMP\hello.war

Cliquez sur « Browse », sélectionnez le fichier .war et cliquez sur « Deploy ».

Le fichier war est téléchargé dans le répertoire webapp, il est déployé par Tomcat (Tomcat est configuré par défaut pour déployer automatiquement les fichiers .war du répertoire webapp).

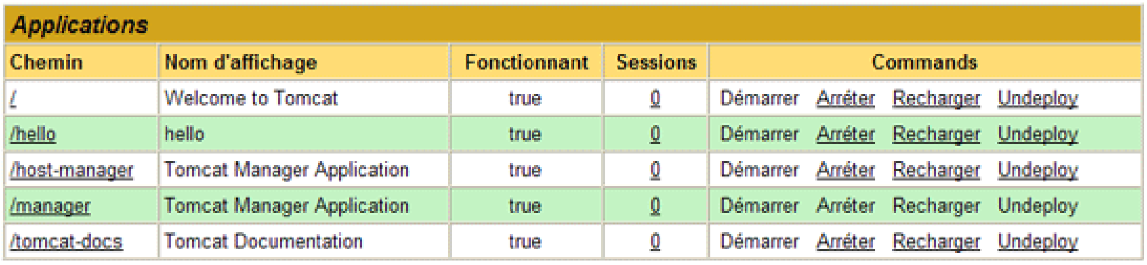


C:\Program Files\Apache Software Foundation\Tomcat 5.5\webapps

Address: C:\Program Files\Apache Software Foundation\Tomcat 5.5\webapps

Name	Size	Type
ROOT		File Folder
tomcat-docs		File Folder
hello		File Folder
hello.war	1 KB	IZArc WAR Archive

La liste des applications est enrichie de l'application déployée.



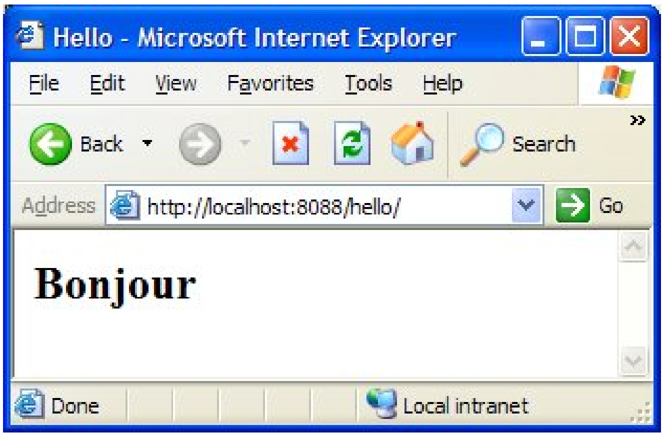
Applications

Chemin	Nom d'affichage	Fonctionnant	Sessions	Commands
/	Welcome to Tomcat	true	0	Démarrer Arrêter Recharger Undeploy
/hello	hello	true	0	Démarrer Arrêter Recharger Undeploy
/host-manager	Tomcat Manager Application	true	0	Démarrer Arrêter Recharger Undeploy
/manager	Tomcat Manager Application	true	0	Démarrer Arrêter Recharger Undeploy
/tomcat-docs	Tomcat Documentation	true	0	Démarrer Arrêter Recharger Undeploy

102.9.1.2. La gestion des applications

La partie applications permet de gérer le cycle de vie des applications déployées.

Il est possible d'accéder à l'application en cliquant sur le lien du chemin de l'application

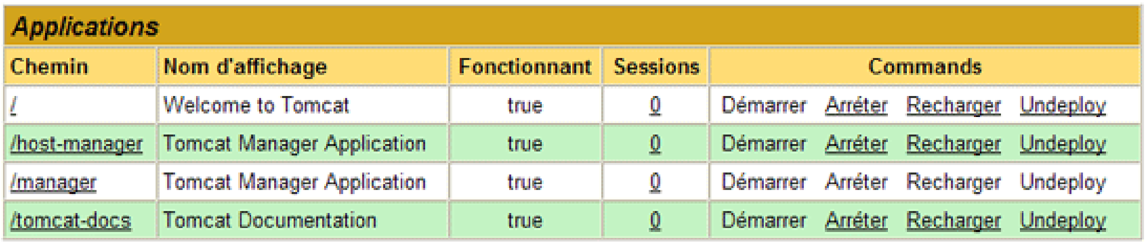


Bonjour

Il est possible de gérer le cycle de vie de l'application en utilisant les liens de commandes :

- arrêter l'application en cliquant sur le lien « Arrêter » de l'application correspondante.
- démarrer l'application en cliquant sur le lien « Démarrer » de l'application correspondante.
- recharger l'application (équivalent à un arrêt/démarrage consécutif) en cliquant sur le lien « Recharger » de l'application correspondante.
- supprimer l'application en cliquant sur le lien « Undeploy » de l'application correspondante.

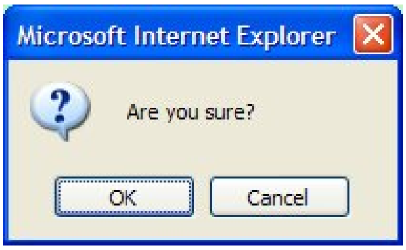
Une application arrêtée est supprimée du serveur Tomcat.



Applications

Chemin	Nom d'affichage	Fonctionnant	Sessions	Commands			
/	Welcome to Tomcat	true	0	Démarrer	Arrêter	Recharger	Undeploy
/host-manager	Tomcat Manager Application	true	0	Démarrer	Arrêter	Recharger	Undeploy
/manager	Tomcat Manager Application	true	0	Démarrer	Arrêter	Recharger	Undeploy
/tomcat-docs	Tomcat Documentation	true	0	Démarrer	Arrêter	Recharger	Undeploy

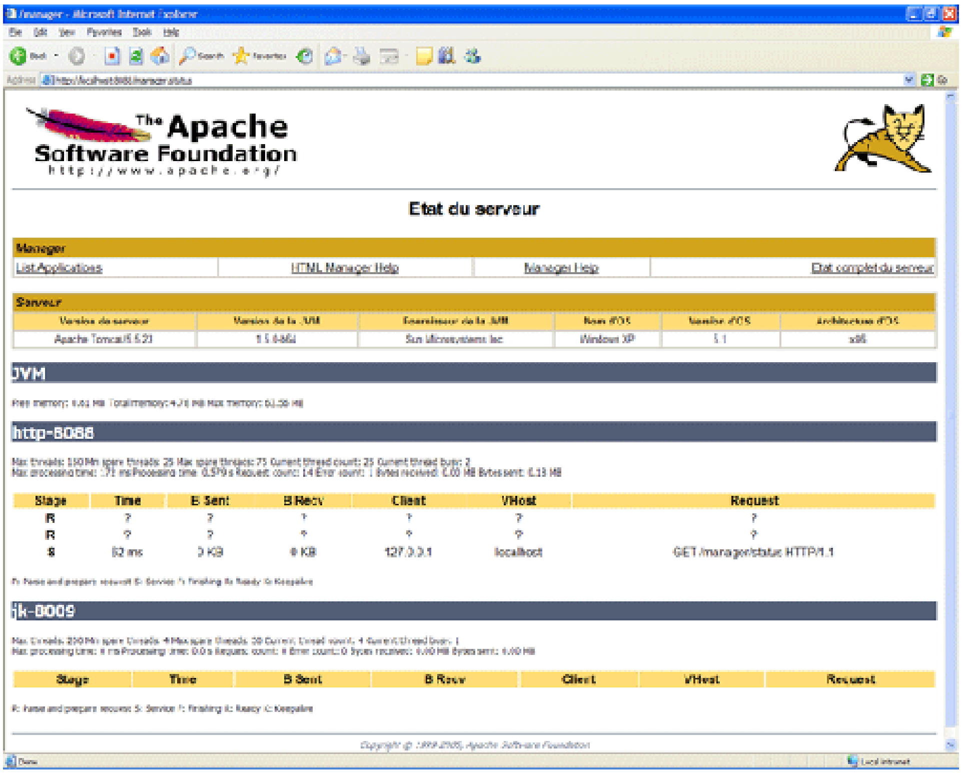
Chacune de ces actions nécessite une confirmation



Are you sure?

Cliquez sur « Etat complet du serveur » pour obtenir des informations sur l'environnement d'exécution :

- Version de Tomcat,
- Information sur la JVM (version et fournisseur),
- Informations sur le système d'exploitation et l'architecture processeur,
- Informations sur la mémoire utilisée et disponible de la JVM,
- Statistiques des ports utilisés par Tomcat



/manager - Microsoft Internet Explorer
http://localhost:8088/manager/status
The Apache Software Foundation
http://www.apache.org/

Etat du serveur

Manager			
List Applications	HTML Manager Help	Manager Help	Etat complet du serveur

Serveur

Version du serveur	Version de la JVM	Fournisseur de la JVM	Nom d'OS	Version d'OS	Architecture d'OS
Apache Tomcat/5.5.20	1.5.0_06	Sun Microsystems Inc.	Windows XP	5.1	x86

JVM

Free memory: 1.61 MB Total memory: 4.71 MB Max memory: 63.56 MB

http-8088

Max threads: 150 Min spare threads: 25 Max spare threads: 75 Current thread count: 25 Current thread busy: 2
Max processing time: 171 ms Processing time: 0.579 s Request count: 14 Error count: 1 Bytes received: 0.00 MB Bytes sent: 0.13 MB

Stage	Time	B Sent	B Recv	Client	VHost	Request
R	?	?	?	?	?	?
R	?	?	?	?	?	?
S	62 ms	0 KB	0 KB	127.0.0.1	localhost	GET /manager/status HTTP/1.1

P: Parse and prepare request S: Service F: Finishing R: Ready K: Keepalive

jk-8009

Max threads: 200 Min spare threads: 4 Max spare threads: 50 Current thread count: 4 Current thread busy: 1
Max processing time: 0 ms Processing time: 0.0 s Request count: 0 Error count: 0 Bytes received: 0.00 MB Bytes sent: 0.00 MB

Stage	Time	B Sent	B Recv	Client	VHost	Request

P: Parse and prepare request S: Service F: Finishing R: Ready K: Keepalive

Copyright © 1999-2005, Apache Software Foundation

102.9.2. L'utilisation des commandes par requêtes HTTP

Comme pour l'utilisation de l'interface, l'utilisation des commandes par requêtes http nécessite une authentification préalable.

Toutes les requêtes pour exécuter une commande sont de la forme :
http://{hôte}:{port}/manager/{commande}?{paramètres}

Hôte et port représentent la machine et le port utilisés par Tomcat. Commande est la commande à exécuter avec ses éventuels paramètres.

Certaines commandes attendent un paramètre path qui précise le chemin du contexte de l'application à utiliser. La valeur de ce paramètre commence par un /.

Remarque : il n'est pas possible d'effectuer des commandes sur l'Application Manager lui-même.

L'exécution de ces commandes renvoie une réponse ayant pour type mime text/plain. Cette réponse ne contient donc aucun tag de formatage HTML ce qui permet de l'exploiter dans des scripts par exemple.

La première ligne indique l'état de l'exécution de la commande : OK ou FAIL pour indiquer respectivement que la commande a réussi ou qu'elle a échoué. Le reste de la ligne contient un message d'information ou d'erreur.

Certaines commandes renvoient des lignes supplémentaires contenant le résultat de leurs exécutions.

102.9.2.1. La commande list

La commande list permet de demander l'affichage de la liste des applications déployées sur le serveur :

Exemple : http://localhost:8088/manager/list

```
OK - Applications listées pour l'hôte virtuel (virtual host) localhost
/admin:running:0:C:/Program Files/Apache/Tomcat 5.5/server/webapps/admin
/host-manager:running:0:C:/Program Files/Apache/Tomcat 5.5/server/webapps/Manager
/tomcat-docs:running:0:tomcat-docs
```

```
/axis:running:0:axis
/:running:0:ROOT
/manager:running:0:C:/Program Files/Apache/Tomcat 5.5/server/webapps/manager
```

Cette liste contient plusieurs informations séparées par des deux-points : le contexte de l'application, son statut (running ou stopped), le nombre de sessions ouvertes et le chemin de la webapp.

102.9.2.2. La commande serverinfo

La commande serverinfo permet d'obtenir des informations sur l'OS et la JVM.

Exemple : http://localhost:8088/manager/serverinfo

```
OK - Server info
Tomcat Version: Apache Tomcat/5.5.23
OS Name: Windows XP
OS Version: 5.1
OS Architecture: x86
JVM Version: 1.5.0-b64
JVM Vendor: Sun Microsystems Inc.
```

102.9.2.3. La commande reload

Cette commande permet de demander le rechargement d'une webapp qui est stockée dans un sous-répertoire (déploiement sous la forme étendue).

Cette commande attend un paramètre path qui doit avoir comme valeur le contexte de la webapp.

Exemple : http://localhost:8080/manager/reload?path=/hello

```
OK - Application rechargée au chemin de contexte /hello
Attention : le rechargement ne concerne que les classes. Le fichier web.xml n'est pas relu :
la prise en compte de modification dans ce fichier nécessite un arrêt/démarrage de la webapp
```

102.9.2.4. La commande resources

La commandes resources permet d'obtenir une liste des ressources JNDI globales définies dans le serveur Tomcat et pouvant être utilisées.

Exemple : http://localhost:8080/manager/resources

```
OK - Liste des ressources globales de tout type
UserDatabase:org.apache.catalina.users.MemoryUserDatabase
jdbc/MonAppDS:org.apache.tomcat.dbcp.dbcp.BasicDataSource
simpleValue:java.lang.Integer
```

Chaque ressource est précisée sur une ligne qui contient son nom et son type séparés par un deux-points.

Il est possible de préciser un type d'objet grâce au paramètre type. Dans ce cas la valeur du paramètre type doit être une classe pleinement qualifiée.

Exemple : http://localhost:8088/manager/resources?type=java.lang.Integer

```
OK - Liste des ressources globales de type java.lang.Integer
simpleValue:java.lang.Integer
```

102.9.2.5. La commande roles

Cette commande donne la liste de tous les rôles définis :

Exemple : http://localhost:8080/manager/roles

```
 OK - Liste de rôles de securité
 tomcat:
 role1:
 manager:
 admin:
```

Chaque ligne contient un rôle et sa description séparée par un caractère deux-points.

102.9.2.6. La commande sessions

Cette commande permet d'obtenir des informations sur les sessions d'un contexte.

Cette commande attend obligatoirement le paramètre path qui précise le chemin du contexte de l'application. Si ce paramètre n'est pas précisé, la commande renvoie une erreur.

Exemple : http://localhost:8080/manager/sessions

```
ECHEC - Un chemin de contexte invalide null a été spécifié
```

Le résultat de la commande contient :

- Le timeout des sessions
- Le nombre de sessions actives par tranche de timeout de 10 minutes

Exemple : http://localhost:8088/manager/sessions?path=/hello

```
 OK - Information de session pour l'application au chemin de contexte /hello
 Interval par défaut de maximum de session inactive 30 minutes
 30 - <40 minutes:2 sessions
```

102.9.2.7. La commande stop

Cette commande permet de demander l'arrêt d'une webapp.

Cette commande attend obligatoirement le paramètre path qui précise le chemin du contexte de l'application à arrêter. Si ce paramètre n'est pas précisé, la commande renvoie une erreur.

Exemple : http://localhost:8080/manager/stop

```
ECHEC - Un chemin de contexte invalide null a été spécifié
```

La commande renvoie son statut et un message d'information :

Exemple : http://localhost:8080/manager/stop?path=/hello

```
OK - Application arrétée pour le chemin de contexte /hello
```

102.9.2.8. La commande start

Cette commande permet de démarrer une webapp.

Cette commande attend obligatoirement le paramètre path qui précise le chemin du contexte de l'application à démarrer. Si ce paramètre n'est pas précisé, la commande renvoie une erreur.

Exemple : http://localhost:8080/manager/start

```
ECHEC - Un chemin de contexte invalide null a été spécifié
```

La commande renvoie son statut et un message d'information :

Exemple : http://localhost:8088/manager/start?path=/hello

```
OK - Application démarrée pour le chemin de contexte /hello
```

102.9.2.9. La commande undeploy

Cette commande permet de supprimer une webapp. Elle arrête préalablement l'application avant sa suppression.

Cette commande attend obligatoirement le paramètre path qui précise le chemin du contexte de l'application à supprimer. Si ce paramètre n'est pas précisé, la commande renvoie une erreur.

Exemple : http://localhost:8080/manager/undeploy

```
ECHEC - Un chemin de contexte invalide null a été spécifié
```

La commande renvoie son statut et un message d'information :

Exemple : http://localhost:8080/manager/undeploy?path=/hello

```
OK - Application non-déployée pour le chemin de contexte /hello
```

Attention : cette commande supprime tout ce qui concerne la webapp.

- Le répertoire de déploiement de l'application s'il existe (par exemple webapps/hello)
- Le fichier .war s'il existe (par exemple webapps/hello.war)
- Le contexte

Si le contexte est défini dans le fichier server.xml, alors la commande échoue :

Exemple : le fichier server.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Server>
...
 <Service name="Catalina">
...
  <Engine defaultHost="localhost" name="Catalina">
   <Realm className="org.apache.catalina.realm.UserDatabaseRealm"/>
   <Host appBase="webapps" name="localhost">
    <Context path="/hello"></Context>
   </Host>
  </Engine>
 </Service>
</Server>
```

Exemple : http://localhost:8088/manager/undeploy?path=/hello

```
FAIL - Context /hello is defined in server.xml and may not be undeployed
```

102.9.2.10. La commande deploy

Cette commande permet de déployer une application.

Exemple : déployer l'application dont le fichier .war est dans le répertoire de déploiement

```
http://localhost:8080/manager/deploy?path=/hello&war=hello.war
```

Exemple : déployer une application dont le répertoire de déploiement est un sous-répertoire du répertoire de déploiement de Tomcat

```
http://localhost:8088/manager/deploy?war=hello&path=/hello

OK - Application déployée pour le chemin de contexte /hello
```

Remarque : cet exemple n'est utile que si l'option autoDeploy est à False dans la configuration du Host concerné. Cette commande permet de déployer une application sous la forme d'un sous-répertoire dans le répertoire de déploiement de Tomcat, sans avoir à redémarrer Tomcat.

102.9.3. L'utilisation du manager avec des tâches Ant

Tomcat 5 propose un ensemble de tâches Ant qui permet d'exécuter des traitements du manager.

Comme pour toute tâche Ant externe, il faut déclarer chaque tâche à utiliser avec le tag taskdef.

Exemple :

```
<project name="Hello" default="list" basedir=".">
 <!-- Propriété d'accès au Manager -->
 <property name="url" value="http://localhost:8088/manager" />
 <property name="username" value="admin" />
 <property name="password" value="admin" />
 <!-- Chemin du contexte de l'application -->
 <property name="path" value="/hello" />
 <!-- Configure the custom Ant tasks for the Manager application -->
 <taskdef name="list" classname="org.apache.catalina.ant.ListTask" />
 <target name="list" description="Liste des webapp déployée">
  <list url="${url}" username="${username}" password="${password}" />
 </target>
</project>
```

Pour utiliser les tâches, il faut que le fichier catalina-ant.jar soit accessible par Ant. Pour cela, il y a deux solutions :
1) Dans la balise classpath de la balise taskdef

Exemple :

```
<project name="Hello" default="list" basedir=".">
 <property name="tomcat.home" value="C:/Program Files/Apache/Tomcat 5.5" />

 <!-- Propriété d'accès au Manager -->
 <property name="url" value="http://localhost:8088/manager" />
 <property name="username" value="admin" />
 <property name="password" value="admin" />

 <!-- Chemin du contexte de l'application -->
 <property name="path" value="/hello" />
 <!-- Configure the custom Ant tasks for the Manager application -->
  <taskdef name="list" classname="org.apache.catalina.ant.ListTask">
  <classpath>
   <path location="${tomcat.home}/server/lib/catalina-ant.jar" />
  </classpath>
 </taskdef>

 <target name="list" description="Liste des webapp déployée">
  <list url="${url}" username="${username}" password="${password}" />
 </target>
</project>
```

2) copier le fichier catalina-ant.jar, contenu dans le sous-répertoire server/lib du répertoire d'installation de Tomcat, dans le sous-répertoire lib du répertoire d'installation de Ant.

Résultat d'exécution :

```
Buildfile: C:\Documents and Settings\jmd\workspace\Tests\build.xml
list:
   [list] OK - Applications listées pour l'hôte virtuel (virtual host) localhost
   [list] /admin:running:0:C:/Program Files/Apache/Tomcat 5.5/server/webapps/admin
   [list] /host-manager:running:0:C:/Program Files/Apache/Tomcat 5.5/server/webapps/
            host-manager
   [list] /tomcat-docs:running:0:tomcat-docs
   [list] /hello:running:0:hello
   [list] /axis:running:0:axis
   [list] /:running:0:ROOT
   [list] /manager:running:0:C:/Program Files/Apache/Tomcat 5.5/server/webapps/manager
BUILD SUCCESSFUL
Total time: 173 milliseconds
```

Tomcat propose plusieurs tâches :

Classe de la tâche	Rôle
org.apache.catalina.ant.InstallTask	Déployer une application
org.apache.catalina.ant.ReloadTask	Recharger une application
org.apache.catalina.ant.ListTask	Lister les applications déployées
org.apache.catalina.ant.StartTask	Démarrer une application
org.apache.catalina.ant.StopTask	Arrêter une application
org.apache.catalina.ant.UndeployTask	Supprimer une application
org.apache.catalina.ant.SessionsTask	Obtenir des informations sur la session
org.apache.catalina.ant.RôlesTask	Obtenir des informations sur les rôles
org.apache.catalina.ant.ServerInfoTask	Obtenir des informations sur le serveur
org.apache.catalina.ant.ResourcesTask	Obtenir des informations sur les ressources JNDI
org.apache.catalina.ant.JMXGetTask	Obtenir une valeur d'un MBean
org.apache.catalina.ant.JMXQueryTask	Effectuer une requête sur le serveur JMX
org.apache.catalina.ant.JMXSetTask	Mettre à jour une valeur d'un MBean

Toutes ces tâches attendent au moins trois paramètres :

- url : url d'accès à l'application Manager
- username : utilisateur ayant le rôle admin
- password : mot de passe de l'utilisateur ayant le rôle admin

Certaines tâches attendent en plus des paramètres dédiés à leurs exécutions.

102.9.4. L'utilisation de la servlet JMXProxy

Tomcat propose une servlet qui fait office de proxy pour obtenir ou mettre à jour des données de MBean.

Par défaut, sans paramètre, la servlet affiche tous les MBeans.

Exemple : http://localhost:8088/manager/jmxproxy

```
OK - Number of results: 200

Name: Users:type=Role,rolename=role1,database=UserDatabase
modelerType: org.apache.catalina.mbeans.RoleMBean
rolename: role1

Name: Users:type=User,username="both",database=UserDatabase
modelerType: org.apache.catalina.mbeans.UserMBean
groups: [Ljava.lang.String;@16be68f
password: tomcat
roles: [Ljava.lang.String;@edf389
username: both

Name: Catalina:type=Manager,path=/axis,host=localhost
modelerType: org.apache.catalina.session.StandardManager
algorithm: MD5
randomFile: /dev/urandom
className: org.apache.catalina.session.StandardManager
distributable: false
entropy: org.apache.catalina.session.StandardManager@a4e743
maxActiveSessions: -1
maxInactiveInterval: 1800
processExpiresFrequency: 6
sessionIdLength: 16
name: StandardManager
pathname: SESSIONS.ser
activeSessions: 0
sessionCounter: 0
maxActive: 0
sessionMaxAliveTime: 0
sessionAverageAliveTime: 0
rejectedSessions: 0
expiredSessions: 0
processingTime: 0
duplicates: 0
...
```

Le paramètre qry permet de préciser une requête pour filtrer les résultats :

Exemple : http://localhost:8088/manager/jmxproxy/?qry=*%3Atype=User%2c*

```
 OK - Number of results: 4

 Name: Users:type=User,username="both",database=UserDatabase
 modelerType: org.apache.catalina.mbeans.UserMBean
 groups: [Ljava.lang.String;@1d8c528
 password: tomcat
 roles: [Ljava.lang.String;@77eaf8
 username: both

 Name: Users:type=User,username="tomcat",database=UserDatabase
 modelerType: org.apache.catalina.mbeans.UserMBean
 groups: [Ljava.lang.String;@e35bb7
 password: tomcat
 roles: [Ljava.lang.String;@9a8a68
 username: tomcat
```

```
Name: Users:type=User,username="role1",database=UserDatabase
modelerType: org.apache.catalina.mbeans.UserMBean
groups: [Ljava.lang.String;@1f4e571
password: tomcat
roles: [Ljava.lang.String;@1038de7
username: role1

Name: Users:type=User,username="admin",database=UserDatabase
modelerType: org.apache.catalina.mbeans.UserMBean
groups: [Ljava.lang.String;@5976c2
password: admin
roles: [Ljava.lang.String;@183e7de
username: admin
```

La servlet permet aussi de modifier à chaud des attributs d'un MBean grâce à trois paramètres :

- set : le nom du MBean
- att : le nom de l'attribut à modifier
- val : la nouvelle valeur de l'attribut

102.10. L'outil Tomcat Client Deployer

L'outil TCD (Tomcat Client Deployer) permet de packager une application et de gérer le cycle de vie de l'application dans le serveur Tomcat. Cet outil repose sur les tâches Ant qui utilisent le Tomcat Manager.

Il faut installer l'outil Ant :

- Télécharger Ant
- Décompresser le fichier obtenu
- Ajouter le répertoire bin d'Ant dans le path du système

La variable d'environnement JAVA_HOME doit pointer sur le répertoire d'un JDK.

Il faut installer l'outil TDC en effectuant les opérations suivantes :

- Télécharger TDC
- Décompresser le fichier obtenu

Il faut définir un fichier deployer.properties qui va contenir des informations sur l'application à gérer et sur le serveur Tomcat. Ces informations sont fournies sous la forme de propriétés :

- webapp : nom de l'application web
- path : chemin de contexte de l'application
- url : url vers le manager de Tomcat
- username : nom de l'utilisateur ayant le rôle manager
- password : mot de passe de l'utilisateur
- build : répertoire de base dans lequel l'application sera compilée. L'application sera compilée dans le répertoire ${build}/webapp/${path}

Exemple :

```
webapp=hello
path=/hello
url=http://localhost:8088/manager
username=admin
password=admin
```

Pour exécuter TCD, il faut lancer ant avec, en paramètre, la tâche à exécuter dans le répertoire qui contient le fichier build.xml.

Exemple :

```
Ant start
Ant stop
Ant reload
```

Les tâches utilisables sont :

- compile : compile et valide une application (classes et JSP). Cette tâche ne nécessite pas d'accès au serveur Tomcat. Attention : cette compilation ne peut fonctionner qu'avec la version de Tomcat correspondant à celle de l'outil
- deploy : déploie une application dans le serveur Tomcat
- start : démarre l'application
- reload : rechargement de l'application
- stop : arrêt de l'application

102.11. Les optimisations

Cette section présente rapidement quelques optimisations possibles dans la configuration de Tomcat notamment dans une optique d'exécution dans un environnement de production.

Il est préférable de mettre à false l'attribut enableLookups des tags <Connector> dans le fichier server.xml : ceci évite à Tomcat de déterminer le nom de domaine à partir de l'adresse IP des requêtes.

Il est préférable de remplacer les valeurs des attributs name des tags <Service> et <Engine> dans le fichier server.xml : ceci permet de les distinguer car par défaut, ils possèdent le même nom (Catalina).

L'attribut reloadable doit être à false pour chaque contexte : ceci évite à Tomcat de vérifier périodiquement le besoin de recharger les classes.

Il faut désactiver dans le fichier server.xml les connectors qui ne sont pas utilisés.

102.12. La sécurisation du serveur

Cette section présente rapidement quelques actions possibles pour améliorer la sécurisation d'un serveur Tomcat notamment dans une optique d'exécution dans un environnement de production. Ces actions ne concernent que Tomcat et occultent complètement la sécurisation du système et du réseau.

Il faut exécuter Tomcat avec un user qui dispose uniquement des privilèges requis pour l'exécution (par exemple, il ne faut surtout pas exécuter Tomcat avec le user root sous Unix mais créer un user tomcat dédié à son exécution).

Les utilisateurs possédant un rôle admin doivent avoir un mot de passe non trivial : il faut prohiber les user/mot de passe de type admin/admin.

Les droits d'accès aux répertoires et fichiers de Tomcat doivent être vérifiés pour ne pas permettre à quiconque de les modifier.

Il est préférable de ne pas installer l'outil d'administration : les fichiers de configuration doivent être modifiés à la main dans le système.

Il faut modifier les valeurs par défaut des attributs port et shutdown du tag <server> du fichier de configuration server.xml : ceci permet d'éviter un éventuel shutdown dû aux valeurs par défaut.

103. Des outils open source pour faciliter le développement

Chapitre 103

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

La communauté open source propose de nombreuses bibliothèques mais aussi des outils dont le but est de faciliter le travail des développeurs. Certains de ces outils sont détaillés dans des chapitres dédiés notamment Ant, Maven et JUnit. Ce chapitre va présenter d'autres outils open source pouvant être regroupés dans plusieurs catégories : contrôle de la qualité des sources et génération et mise en forme de code.

La génération de certains morceaux de code ou de fichiers de configuration peut parfois être fastidieuse voire même répétitive dans certains cas. Pour faciliter le travail des développeurs, des outils open source ont été développés par la communauté afin de générer ce code. Ce chapitre présente deux outils open source : XDoclet et Middlegen.

La qualité du code source est un facteur important pour tous développements. Ainsi certains outils permettent de faire des vérifications sur des règles de codification dans le code source. C'est le cas pour l'outil CheckStyle.

Ce chapitre contient plusieurs sections :

- ♦ CheckStyle
- ♦ Jalopy

103.1. CheckStyle

CheckStyle est un outil open source qui propose de puissantes fonctionnalités pour appliquer des contrôles sur le respect de règles de codifications.

Pour définir les contrôles réalisés lors de son exécution CheckStyle utilise une configuration qui repose sur des modules. Cette configuration est définie dans un fichier XML qui précise les modules utilisés et pour chacun d'entre-eux leurs paramètres.

Le plus simple est d'utiliser le fichier de configuration nommé sun_checks.xml fourni avec CheckStyle. Cette configuration propose d'effectuer des contrôles du respect des normes de codification proposées par Sun. Il est aussi possible de définir son propre fichier de configuration.

Le site officiel de CheckStyle est à l'URL : https://checkstyle.sourceforge.net/

La version utilisée dans cette section est la 3.4

103.1.1. L'installation

Il faut télécharger le fichier checkstyle-3.4.zip sur le site de ChekStyle et le décompresser dans un répertoire du système.

La décompression de ce fichier crée un répertoire checkstyle-3.4 contenant lui-même les bibliothèques utiles et deux répertoires (docs et contrib).

CheckStyle peut s'utiliser de deux façons :

- en ligne de commande
- comme une tâche Ant ce qui permet d'automatiser son exécution

103.1.2. L'utilisation avec Ant

Pour utiliser CheckStyle, le plus simple est d'ajouter la bibliothèque checkstyle-all-3.4.jar au classpath.

Dans les exemples de cette section, la structure de répertoires suivante est utilisée :

/bin
/lib
/outils
/outils/lib
/outils/checkstyle
/src
/temp
/temp/checkstyle

Le fichier checkstyle-all-3.4.jar est copié dans le répertoire outils/lib et le fichier sun_checks.xml fourni par CheckStyle est copié dans le répertoire outils/checkstyle.

Il faut déclarer le tag CheckStyle dans Ant en utilisant le tag <taskdef> et définir une tâche qui va utiliser le tag <CheckStyle>.

Exemple : fichier source de test

```
public class MaClasse {

  public static void main() {
    System.out.println("Bonjour");
  }

}
```

Le fichier de build ci-dessous sera exécuté par Ant.

Exemple :

```
<project name="utilisation de checkstyle" default="compile" basedir=".">
  <!-- Définition des propriétés du projet -->
  <property name="projet.sources.dir"       value="src"/>
  <property name="projet.bin.dir"           value="bin"/>
  <property name="projet.lib.dir"           value="lib"/>
  <property name="projet.temp.dir"          value="temp"/>
  <property name="projet.outils.dir"        value="outils"/>
  <property name="projet.outils.lib.dir"    value="${projet.outils.dir}/lib"/>

  <!-- Définition du classpath du projet -->
  <path id="projet.classpath">
    <fileset dir="${projet.lib.dir}">
      <include name="*.jar"/>
    </fileset>
    <fileset dir="${projet.outils.lib.dir}">
      <include name="*.jar"/>
    </fileset>
    <pathelement location="${projet.bin.dir}" />
  </path>

  <!-- Déclaration de la tâche Ant permettant l'exécution de checkstyle -->
  <taskdef resource="checkstyletask.properties"
    classpathref="projet.classpath" />
  <!-- Exécution de checkstyle -->
  <target name="checkstyle" description="CheckStyle">
    <checkstyle config="outils/checkstyle/sun_checks.xml">
      <fileset dir="${projet.sources.dir}" includes="**/*.java"/>
      <formatter type="plain"/>
    </checkstyle>
  </target>

  <!-- Compilation des classes du projet -->
  <target name="compile" depends="checkstyle" description="Compilation des classes">
    <javac srcdir="${projet.sources.dir}"
      destdir="${projet.bin.dir}"
      debug="on"
      optimize="off"
      deprecation="on">
      <classpath refid="projet.classpath"/>
    </javac>
  </target>

</project>
```

Résultat :

```
C:\java\test\testcheckstyle>ant
Buildfile: build.xml
checkstyle:
[checkstyle] C:\java\test\testcheckstyle\src\package.html:0: Missing package doc
umentation file.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:0: File does not end
with a newline.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:2: Missing a Javadoc
comment.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:2:1: Utility classes
should not have a public or default constructor.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:4:3: Missing a Javado
c comment.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:5: Line has trailing
spaces.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:5:35: La ligne contie
nt un caractÞre tabulation.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:7: Line has trailing
spaces.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:7:1: La ligne contien
t un caractÞre tabulation.
BUILD FAILED
file:C:/java/test/testcheckstyle/build.xml:27: Got 9 errors.
      Total time: 7 seconds
```

Le tag <checkstyle> possède plusieurs attributs :

Nom	Rôle
file	précise le nom de l'unique fichier à vérifier. Pour préciser un ensemble de fichiers, il faut utiliser le tag fils <fileset>
config	précise le nom du fichier de configuration des modules de ChecksStyle
failOnViolation	précise si les traitements de l'outil Ant doivent être stoppés en cas d'échec des contrôles. La valeur par défaut est true
failureProperty	précise le nom d'une propriété qui sera valorisée en cas d'échec des contrôles

Exemple :

```
...
  <!-- Exécution de checkstyle -->
  <target name="checkstyle" description="CheckStyle">
    <checkstyle config="outils/checkstyle/sun_checks.xml" failOnViolation="false">
      <fileset dir="${projet.sources.dir}" includes="**/*.java"/>
      <formatter type="plain"/>
    </checkstyle>
  </target>
...
```

Résultat :

```
C:\java\test\testcheckstyle>ant
Buildfile: build.xml
checkstyle:
[checkstyle] C:\java\test\testcheckstyle\src\package.html:0: Missing package doc
umentation file.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:2: Missing a Javadoc
comment.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:2:1: Utility classes
should not have a public or default constructor.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:4:3: Missing a Javado
c comment.
compile:
[javac] Compiling 1 source file to C:\java\test\testcheckstyle\bin
BUILD SUCCESSFUL
      Total time: 11 seconds
```

Il est possible de préciser deux types de formats de sortie des résultats lors de l'exécution de CheckStyle. Le format de sortie des résultats est précisé par un tag fils <formatter>. Ce tag possède deux attributs :

Nom	Rôle
type	précise le type. Deux valeurs sont possibles : plain (par défaut) et xml
toFile	précise un fichier dont le type déterminera le format des résultats stockés (par défaut la sortie standard de la console)

Exemple :

```
...
<!-- Exécution de checkstyle -->
<target name="checkstyle" description="CheckStyle">
  <checkstyle config="outils/checkstyle/sun_checks.xml" failOnViolation="false">
    <fileset dir="${projet.sources.dir}" includes="**/*.java"/>
    <formatter type="xml" toFile="${projet.temp.dir}/checkstyle_erreurs.xml"/>
  </checkstyle>
</target>
...
```

Il est alors possible d'appliquer une feuille de styles sur le fichier XML généré afin de créer un rapport dans un format dédié. L'exemple suivant utilise une feuille de style fournie par CheckStyle dans le répertoire contrib : cette feuille, nommée checkstyle-frames.xsl, est copiée dans le répertoire outils/checkstyle.

Exemple :

```
...
<!-- Exécution de checkstyle -->
<target name="checkstyle" description="CheckStyle">
  <checkstyle config="${projet.outils.dir}/checkstyle/sun_checks.xml" failOnViolation="false">
    <fileset dir="${projet.sources.dir}" includes="**/*.java"/>
    <formatter type="xml" toFile="${projet.temp.dir}/checkstyle/checkstyle_erreurs.xml"/>
  </checkstyle>
  <style in="${projet.temp.dir}/checkstyle/checkstyle_erreurs.xml"
    out="${projet.temp.dir}/checkstyle/checkstyle_rapport.htm"
    style="${projet.outils.dir}/checkstyle/checkstyle-frames.xsl"/>
</target>
...
```

Exemple :

```
C:\java\test\testcheckstyle>ant
```

```
Buildfile: build.xml
checkstyle:
[style] Processing C:\java\test\testcheckstyle\temp\checkstyle\checkstyle_er
reurs.xml to C:\java\test\testcheckstyle\temp\checkstyle\checkstyle_rapport.htm
[style] Loading stylesheet C:\java\test\testcheckstyle\outils\checkstyle\che
ckstyle-frames.xsl
compile:
BUILD SUCCESSFUL
Total time: 8 seconds
```

Il est possible d'utiliser d'autres feuilles de styles fournies par CheckStyle ou de définir la sienne.

103.1.3. L'utilisation en ligne de commandes

Pour utiliser CheckStyle en ligne de commandes, il faut ajouter le fichier checkstyle-all-3.4.jar au classpath par exemple en utilisant l'option -cp de l'interpréteur Java.

La classe à exécuter est com.puppycrawl.tools.checkstyle.Main

CheckStyle accepte plusieurs paramètres pour son exécution :

Option	Rôle
-c fichier_de_configuration	précise le fichier de configuration
-f format	précise le format (plain ou xml)
-o fichier	précise le fichier qui va contenir les résultats
-r	précise le répertoire dont les fichiers sources vont être récursivement traités

Exemple :

```
java -cp outils\lib\checkstyle-all-3.4.jar com.puppycrawl.tools.checkstyle.Main
  -c outils\checkstyle/sun_checks.xml -r src
```

Exemple :

```
...
C:\java\test\testcheckstyle>java -cp outils\lib\checkstyle-all-3.4.jar com.puppy
crawl.tools.checkstyle.Main -c outils\checkstyle/sun_checks.xml -r src
Starting audit...
C:\java\test\testcheckstyle\src\package.html:0: Missing package documentation fi
le.
src\MaClasse.java.bak:0: File does not end with a newline.
src\MaClasse.java:2: Missing a Javadoc comment.
src\MaClasse.java:2:1: Utility classes should not have a public or default const
ructor.
src\MaClasse.java:4:3: Missing a Javadoc comment.
Audit done.
...
```

103.2. Jalopy

Jalopy est un outil open source qui propose de formater les fichiers sources selon des règles définies.

Le site web officiel de Jalopy est http://jalopy.sourceforge.net.

Jalopy propose entre autres les fonctionnalités suivantes :

- formatage des accolades selon plusieurs formats (C, Sun, GNU)
- indentation du code

- gestion des sauts de lignes
- génération automatique ou vérification des commentaires Javadoc
- ordonnancement des éléments qui composent la classe
- ajout de texte au début et à la fin de chaque fichier
- des plugins pour une intégration dans plusieurs outils : ant, Eclipse, Jbuilder, ...
- ...

Pour connaître les règles de formatage à appliquer, Jalopy utilise une convention qui est un ensemble de paramètres.

Jalopy peut être utilisé grâce à ses plugins de plusieurs façons notamment avec Ant, en lignes de commandes ou avec certains IDE.

La version de Jalopy utilisée dans cette section est la 1.0.B10.

103.2.1. L'utilisation avec Ant

Le plus simple est d'utiliser Jalopy avec Ant pour automatiser son utilisation : pour cela, il faut télécharger le fichier jalopy-ant-0.6.2.zip et le décompresser dans un répertoire du système.

La structure de l'arborescence du projet utilisé dans cette section est la suivante :

/bin
/lib
/outils
/outils/lib
/src

Le répertoire src contient les sources Java à formater.

Les fichiers du répertoire lib de Jalopy sont copiés dans le répertoire outils/lib du projet.

Exemple : le fichier source qui sera formaté

```
public class MaClasse {public static void main() { System.out.println("Bonjour"); }}
```

Il faut définir un fichier build.xml pour Ant qui va contenir les différentes tâches du projet dont une permettant l'appel à Jalopy.

Exemple :

```
<project name="utilisation de jalopy" default="jalopy" basedir=".">
  <!-- Définition des propriétés du projet -->
  <property name="projet.sources.dir" value="src"/>
  <property name="projet.bin.dir" value="bin"/>
  <property name="projet.lib.dir" value="lib"/>
  <property name="projet.temp.dir" value="temp"/>
  <property name="projet.outils.dir" value="outils"/>
  <property name="projet.outils.lib.dir" value="${projet.outils.dir}/lib"/>

  <!-- Définition du classpath du projet -->
  <path id="projet.classpath">
    <fileset dir="${projet.lib.dir}">
      <include name="*.jar"/>
    </fileset>
    <fileset dir="${projet.outils.lib.dir}">
      <include name="*.jar"/>
    </fileset>
    <pathelement location="${projet.bin.dir}" />
  </path>

  <!-- Déclaration de la tâche Ant permettant l'exécution de Jalopy -->
  <taskdef name="jalopy"
         classname="de.hunsicker.jalopy.plugin.ant.AntPlugin"
      classpathref="projet.classpath" />

  <!-- Exécution de Jalopy -->
  <target name="jalopy" description="Jalopy" depends="compile" >
    <jalopy loglevel="info"
                threads="2"
           classpathref="projet.classpath">
      <fileset dir="${projet.sources.dir}">
        <include name="**/*.java" />
      </fileset>
    </jalopy>
  </target>

  <!-- Compilation des classes du projet -->
  <target name="compile" description="Compilation des classes">
    <javac srcdir="${projet.sources.dir}"
             destdir="${projet.bin.dir}"
               debug="on"
            optimize="off"
         deprecation="on">
      <classpath refid="projet.classpath"/>
    </javac>
  </target>

</project>
```

Exemple :

```
C:\java\test\testjalopy>ant
Buildfile: build.xml

compile:
    [javac] Compiling 1 source file to C:\java\test\testjalopy\bin

jalopy:
   [jalopy] Jalopy Java Source Code Formatter 1.0b10
   [jalopy] Format 1 source file
   [jalopy] C:\java\test\testjalopy\src\MaClasse.java:0:0: Parse
   [jalopy] 1 source file formatted

BUILD SUCCESSFUL
Total time: 9 seconds
```

Suite à l'exécution de Jalopy, le code du fichier est reformaté.

Exemple :

```
public class MaClasse {
    public static void main() {
        System.out.println("Bonjour");
    }
}
```

Il est fortement recommandé de réaliser la tâche de compilation des sources avant leur formatage car pour assurer un formatage correct les sources doivent être correctes syntaxiquement parlant.

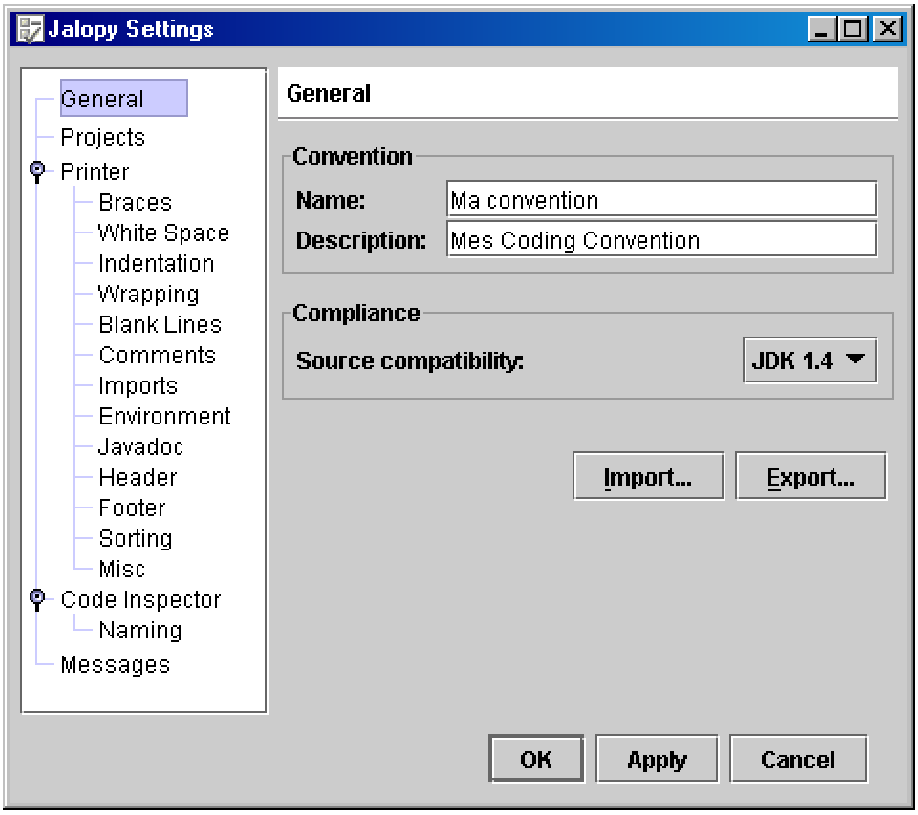
103.2.2. Les conventions

Jalopy est hautement paramétrable. Les options utilisées sont regroupées dans une convention.

Tous ses paramètres sont stockés dans le sous-répertoire .jalopy du répertoire Home de l'utilisateur.

Pour faciliter la gestion de ces paramètres, Jalopy propose un outil graphique qui permet de gérer les conventions.

Pour exécuter cet outil, il suffit de lancer le script preferences dans le répertoire bin de Jalopy (preferences.bat sous Windows et preferences.sh sous Unix).



Name: Ma convention
Description: Mes Coding Convention
Source compatibility: JDK 1.4

Toutes les nombreuses options de formatage d'une convention peuvent être réglées par cet outil. Consultez la documentation fournie avec Jalopy pour avoir le détail de chaque option.

Une fonctionnalité particulièrement utile de cet outil est de proposer une prévisualisation d'un exemple mettant en oeuvre les options sélectionnées.

Voici un exemple avec quelques personnalisations notamment, une gestion des clauses import, la génération des commentaires Javadoc :

Exemple :

```
import java.util.*;

public class MaClasse {
    public static void main() {
        List liste = new ArrayList();
        System.out.println("Bonjour");
    }

    /**
     *
     */
    private int maMethode(int valeur) {
        return valeur * 2;
    }
}
```

Résultat :

```
C:\java\test\testjalopy>ant
Buildfile: build.xml

compile:
    [javac] Compiling 1 source file to C:\java\test\testjalopy\bin

jalopy:
   [jalopy] Jalopy Java Source Code Formatter 1.0b10
   [jalopy] Format 1 source file
   [jalopy] C:\java\test\testjalopy\src\MaClasse.java:0:0: Parse
   [jalopy] C:\java\test\testjalopy\src\MaClasse.java:1:0: On-demand import "jav
a.util.List" expanded
   [jalopy] C:\java\test\testjalopy\src\MaClasse.java:1:0: On-demand import "jav
a.util.ArrayList" expanded
   [jalopy] C:\java\test\testjalopy\src\MaClasse.java:11:1: Generated Javadoc co
mment
   [jalopy] C:\java\test\testjalopy\src\MaClasse.java:18:3: Generated Javadoc co
mment
   [jalopy] 1 source file formatted

BUILD SUCCESSFUL
Total time: 10 seconds
```

Voici le source du code reformaté :

Exemple :

```
//==============================================================================
// fichier :       MaClasse.java
// projet :        $project$
//
// Modification :  date :       $Date$
//                 auteur :     $Author$
//                 revision :   $Revision$
//------------------------------------------------------------------------------
// copyright:   JMD
//==============================================================================

import java.util.ArrayList;
import java.util.List;


/**
 * DOCUMENT ME!
 *
 * @author $author$
 * @version $Revision$
 */
public class MaClasse {
  /**
   * DOCUMENT ME!
   */
  public static void main() {
    List liste = new ArrayList();
    System.out.println("Bonjour");
    System.out.println("");
  }

  /**
   * Calculer le double
   *
   * @param valeur DOCUMENT ME!
   *
   * @return DOCUMENT ME!
   */
  private int maMethode(int valeur) {
    return valeur * 2;
  }
}

// fin du fichier
```

Partie 15 : La conception et le développement des applications

Partie 15 :

La conception et
le développement
d'applications

Pour faciliter le développement d'applications, il est préférable de suivre une méthodologie pour l'analyse et d'utiliser ou de définir des normes lors du développement. L'utilisation de frameworks et de bibliothèques dans une architecture adaptée est également impératif lors de cette tâche.

La communauté Java permet d'obtenir des outils, des frameworks, des bibliothèques mais aussi de très nombreuses informations autour des technologies Java.

Cette partie contient les chapitres suivants :

- Java et UML : propose une présentation de la notation UML ainsi que sa mise en oeuvre avec Java
- Les motifs de conception (design patterns) : présente certains modèles de conception en programmation orientée objet et leur mise en oeuvre avec Java
- Des normes de développement : propose de sensibiliser le lecteur à l'importance de la mise en place de normes de développement sur un projet et propose quelques règles pour définir de telles normes
- Les techniques de développement spécifiques à Java : couvre des techniques de développement spécifiques à Java
- L'encodage des caractères : ce chapitre fournit des informations sur l'encodage des caractères dans les applications Java.
- Les frameworks : présente les frameworks et propose quelques solutions open source dans divers domaines
- La génération de documents : ce chapitre présente plusieurs API open source permettant la génération de documents dans différents formats notamment PDF et Excel
- La validation des données : la validation des données est une tâche commune, nécessaire et importante dans chaque application de gestion de données.

- L'utilisation des dates : ce chapitre détaille l'utilisation des dates en Java
- La planification de tâches : ce chapitre propose différentes solutions pour planifier l'exécution de tâches dans une application Java
- Des bibliothèques open source : présentation de quelques bibliothèques de la communauté open source particulièrement pratiques et utiles
- Apache Commons : Ce chapitre décrit quelques fonctionnalités de la bibliothèque Apache Commons

104. Java et UML

Chapitre 104

Niveau : 🟩🟩□□□□ Elémentaire

Le but d'UML est de modéliser un système en utilisant des objets. L'orientation objet de Java ne peut qu'inciter à l'utiliser avec UML. La modélisation proposée par UML repose sur 9 diagrammes.

Ce chapitre contient plusieurs sections :

- La présentation d'UML
- Les commentaires
- Les cas d'utilisations (use cases)
- Le diagramme de séquence
- Le diagramme de collaboration
- Le diagramme d'états-transitions
- Le diagramme d'activités
- Le diagramme de classes
- Le diagramme d'objets
- Le diagramme de composants
- Le diagramme de déploiement

104.1. La présentation d'UML

UML qui est l'acronyme d'Unified Modeling Language est aujourd'hui indissociable de la conception objet. UML est le résultat de la fusion de plusieurs méthodes de conception objet des pères d'UML qui étaient : Jim Rumbaugh (OMT), Grady Booch (Booch method) et Ivar Jacobson (use case).

UML a été adopté et normalisé par l'OMG (Object Management Group) en 1997.

D'une façon général, UML est une représentation standardisée d'un système orienté objet.

UML n'est pas une méthode de conception mais une notation graphique normalisée de présentation de certains concepts pour modéliser des systèmes objets. En particulier, UML ne précise pas dans quel ordre et comment concevoir les différents diagrammes qu'il défini. Cependant, UML est indépendant de toute méthode de conception et peut être utilisé avec n'importe lequel de ces processus.

Un standard de présentation des concepts permet de faciliter le dialogue entre les différents acteurs du projet : les autres analystes, les développeurs et même les utilisateurs.

UML est composé de neuf diagrammes :

- des cas d'utilisations
- de séquence
- de collaboration
- d'états-transitions
- d'activité
- de classes

- d'objets
- de composants
- de déploiement

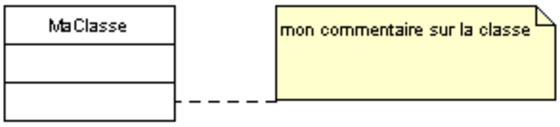
UML regroupe ces neufs diagrammes dans trois familles :

- les diagrammes statiques (diagrammes de classes, d'objet et de cas d'utilisation)
- les diagrammes dynamiques (diagrammes d'activité, de collaboration, de séquence, d'état-transitions et de cas d'utilisation)
- les diagrammes d'architecture : (diagrammes de composants et de déploiements)

104.2. Les commentaires

Utilisable dans chaque diagramme, UML propose une notation particulière pour indiquer des commentaires.

Exemple :



104.3. Les cas d'utilisations (use cases)

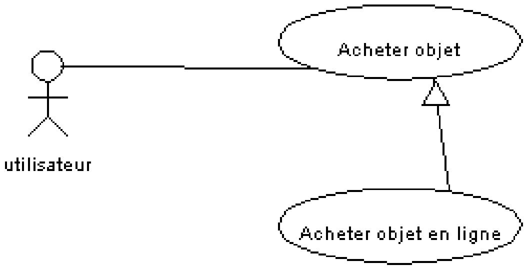
Ils sont développés par Ivar Jacobson et permettent de modéliser des processus métiers en les découpant en cas d'utilisations.

Ce diagramme permet de représenter les fonctionnalités d'un système. Il se compose :

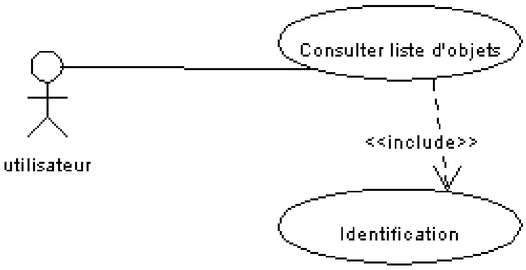
- d'acteurs : ce sont des entités qui utilisent le système à représenter
- les cas d'utilisation : ce sont des fonctionnalités proposées par le système

Un acteur n'est pas une personne désignée : c'est une entité qui joue un rôle dans le système. Il existe plusieurs types de relations qui associent un acteur et un cas d'utilisation :

- la généralisation : cette relation peut être vue comme une relation d'héritage. Un cas d'utilisation enrichit un autre cas en le spécialisant



- l'extension (stéréotype <<extend>>) : le cas d'utilisation complète un autre cas d'utilisation
- l'inclusion (stéréotype <<include>>) : le cas d'utilisation utilise un autre cas d'utilisation



Les cas d'utilisation sont particulièrement intéressants pour recenser les différents acteurs et les différentes fonctionnalités d'un système.

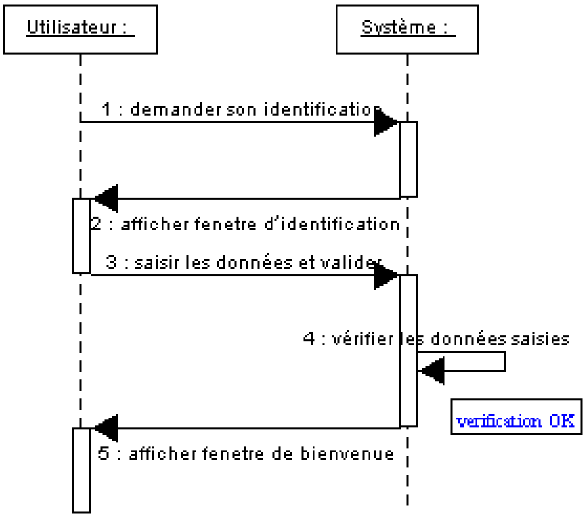
La simplicité de ce diagramme lui permet d'être rapidement compris par des utilisateurs non-informaticiens. Il est d'ailleurs très important de faire participer les utilisateurs tout au long de son évolution.

Le cas d'utilisation est ensuite détaillé en un ou plusieurs scénarios. Un scénario est une suite d'échanges entre des acteurs et le système pour décrire un cas d'utilisation dans un contexte particulier. C'est un enchaînement précis et ordonné d'opérations pour réaliser le cas d'utilisation.

Si le scénario est trop "volumineux", il peut être judicieux de découper le cas d'utilisation en plusieurs et d'utiliser les relations appropriées.

Un scénario peut être représenté par un diagramme de séquence ou sous une forme textuelle. La première forme est très visuelle

Exemple :


Utilisateur :
Système :
1 : demander son identification
2 : afficher fenetre d'identification
3 : saisir les données et valider
4 : vérifier les données saisies
verification OK
5 : afficher fenetre de bienvenue

La seconde facilite la représentation des opérations alternatives.

Les cas d'utilisation permettent de modéliser des concepts fonctionnels. Ils ne précisent pas comment chaque opération sera implémentée techniquement. Il faut rester le plus abstrait possible dans les concepts qui s'approchent de la partie technique.

Le découpage d'un système en cas d'utilisation n'est pas facile car il faut trouver un juste milieu entre un découpage fort (les scénarios sont importants) et un découpage faible (les cas d'utilisation se réduisent à une seule opération).

104.4. Le diagramme de séquence

Il permet de modéliser les échanges de messages entre les différents objets dans le contexte d'un scénario précis.

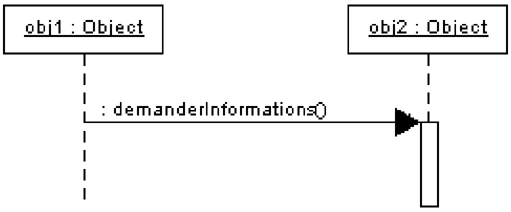
Il permet de représenter les interactions entre différentes entités. Il s'utilise essentiellement pour décrire les scénarios d'un cas d'utilisation (les entités sont les acteurs et le système) ou décrire des échanges entre objets.

Dans le premier cas, les interactions sont des actions qui sont réalisées par une entité.

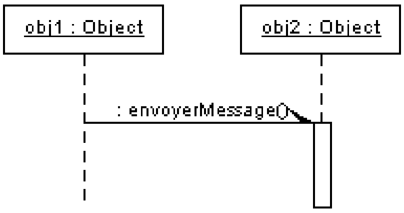
Dans le second cas, les itérations sont des appels de méthodes.

Les itérations peuvent être de deux types :

- synchrone : l'émetteur attend une réponse du récepteur

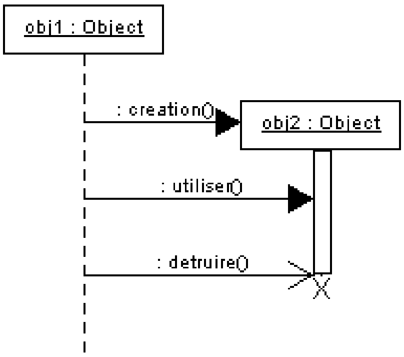


- asynchrone : l'émetteur poursuit son exécution sans attendre de réponse



Un diagramme de séquence peut aussi représenter le cycle de vie d'un objet.

Exemple :



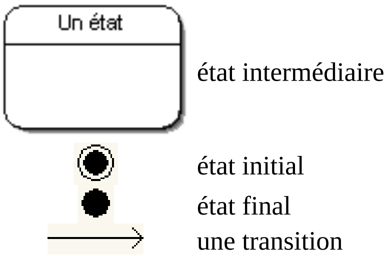
104.5. Le diagramme de collaboration

Il permet de modéliser la collaboration entre les différents objets.

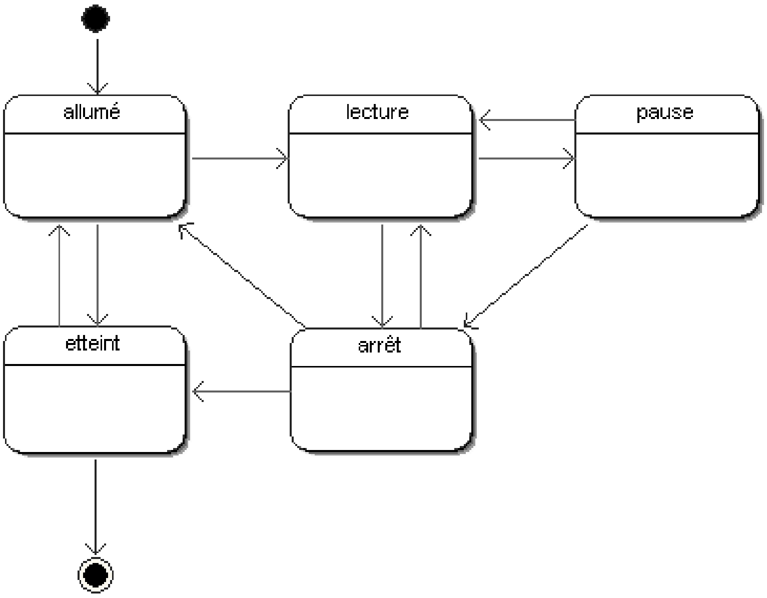
104.6. Le diagramme d'états-transitions

Un diagramme d'état permet de modéliser les différents états d'une entité, en générale une classe. L'ensemble de ces états est connu.

Ce diagramme se compose de plusieurs éléments principaux :

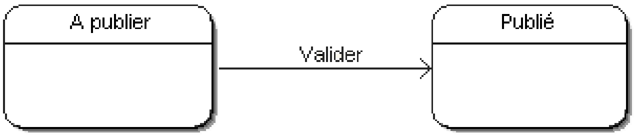


Exemple :



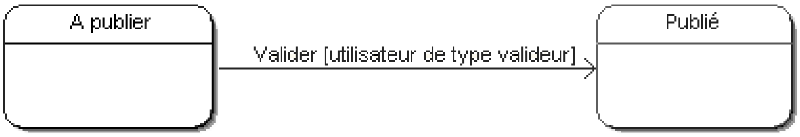
Les transitions sont des événements qui permettent de passer d'un état à un autre : chaque transition possède un sens qui précise l'état de départ et l'état d'arrivée (du côté de la flèche). Une transition peut avoir un nom qui permet de la préciser.

Exemple :



Il est possible d'ajouter une condition à une transition. Cette condition est une expression booléenne placée entre crochets qui sera vérifiée lors d'une demande de transition. Cette condition est indiquée entre crochets.

Exemple :



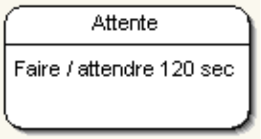
Dans un état, il est possible de préciser des actions ou des activités qui sont des traitements à réaliser. Celles-ci sont décrites avec une étiquette qui désigne le moment de l'exécution.

Une action est un traitement cours. Une activité est un traitement durant tout ou partie de la durée de maintient de l'état.

Plusieurs étiquettes standard sont définies :

- entrée (entry) : action réalisée à l'entrée dans l'état
- sortie (exit) : action réalisée à la sortie de l'état
- faire (do) : activité exécutée durant l'état

Exemple :



Il est aussi possible de définir des actions internes.

104.7. Le diagramme d'activités

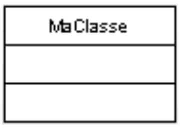
104.8. Le diagramme de classes

Ce schéma représente les différentes classes : il détaille le contenu de chaque classe mais aussi les relations qui peuvent exister entre les différentes classes.

Une classe est représentée par un rectangle séparé en trois parties :

- la première partie contient le nom de la classe
- la seconde contient les attributs de la classe
- la dernière contient les méthodes de la classe

Exemple :



Exemple :

```
public class MaClasse {
}
```

104.8.1. Les attributs d'une classe

Pour définir un attribut, il faut préciser son nom suivi du caractère ":" et du type de l'attribut.

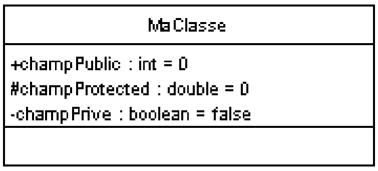
Le modificateur d'accès de l'attribut doit précéder son nom et peut prendre les valeurs suivantes :

Caractère	Rôle

+	accès public
#	accès protected
-	accès private

Une valeur d'initialisation peut être précisée juste après le type en utilisant le signe "=" suivi de la valeur.

Exemple :



Exemple :

```
public class MaClasse {

  public int champPublic = 0;
  protected double champProtected = 0;
  private boolean champPrive = false;
}
```

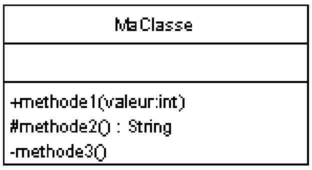
104.8.2. Les méthodes d'une classe

Les modificateurs sont identiques à ceux des attributs.

Les paramètres de la méthode peuvent être précisés en les indiquant entre les parenthèses sous la forme nom : type.

Si la méthode renvoie une valeur son type doit être précisé après un signe ":".

Exemple :



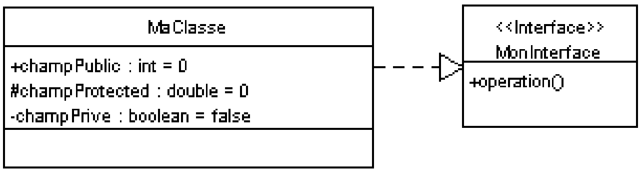
Exemple :

```
public class MaClasse {

  public void methode1(int valeur){
  }

  protected String methode2(){
  }

  private void methode3(){
  }
}
```

Il n'est pas obligatoire d'inclure dans le diagramme tous les attributs et toutes les méthodes d'une classe : seules les entités les plus significatives et utiles peuvent être mentionnées.

104.8.3. L'implémentation d'une interface

Exemple :

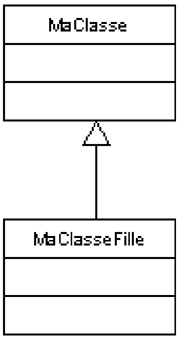


Exemple :

```
public class MaClasse implements MonInterface {
    public int champPublic = 0;
    protected double champProtected = 0;
    private boolean champPrive = false;

    public operation() {

    }
}
```

104.8.4. La relation d'héritage

Exemple :



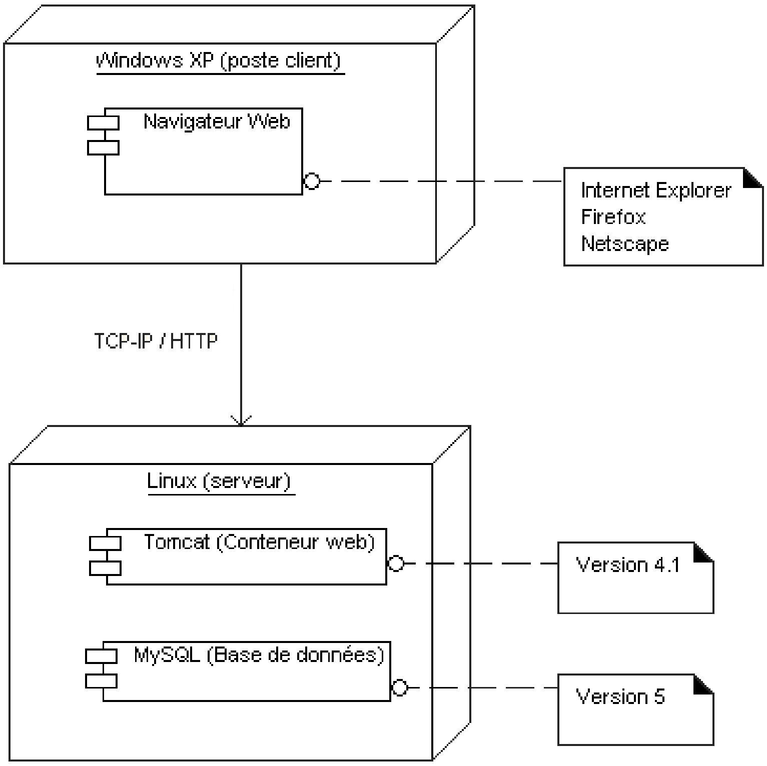
Exemple :

```
public class MaClasseFille extends MaClasse {

}
```

104.9. Le diagramme d'objets

104.10. Le diagramme de composants

104.11. Le diagramme de déploiement



105. Les motifs de conception (design patterns)

Chapitre 105

Niveau : 🟩🟩🟨⬜⬜⬜ Intermédiaire

Le nombre d'applications développées avec des technologies orientées objets augmentant, l'idée de réutiliser des techniques pour solutionner des problèmes courants a abouti aux recensements d'un certain nombre de modèles connus sous le nom de motifs de conception (design patterns).

Ces modèles sont définis pour pouvoir être utilisés avec un maximum de langages orientés objets.

Le nombre de ces modèles est en constante augmentation. Le but de ce chapitre n'est pas de tous les recenser mais de présenter les plus utilisés et de fournir un ou des exemples de leur mise en oeuvre avec Java.

Il est habituel de regrouper ces modèles communs dans trois grandes catégories :

- les modèles de création (creational patterns)
- les modèles de structuration (structural patterns)
- les modèles de comportement (behavioral patterns)

Le motif de conception le plus connu est sûrement le modèle MVC (Model View Controller) mis en oeuvre en premier avec SmallTalk.

Ce chapitre contient plusieurs sections :

♦ Les modèles de création
♦ Les modèles de structuration
♦ Les modèles de comportement

105.1. Les modèles de création

Dans cette catégorie, il existe 5 modèles principaux :

Nom	Rôle
Fabrique (Factory)	Créer un objet dont le type dépend du contexte
Fabrique abstraite (abstract Factory)	Fournir une interface unique pour instancier des objets d'une même famille sans avoir à connaître les classes à instancier
Monteur (Builder)	
Prototype (Prototype)	Création d'objet à partir d'un prototype
Singleton (Singleton)	Classe qui ne pourra avoir qu'une seule instance

105.1.1. Fabrique (Factory)

La fabrique permet de créer un objet dont le type dépend du contexte : cet objet fait partie d'un ensemble de sous-classes. L'objet retourné par la fabrique est donc toujours du type de la classe mère mais grâce au polymorphisme les traitements exécutés sont ceux de l'instance créée.

Ce motif de conception est utilisé lorsqu'à l'exécution il est nécessaire de déterminer dynamiquement quel objet d'un ensemble de sous-classes doit être instancié.

Il est utilisable lorsque :

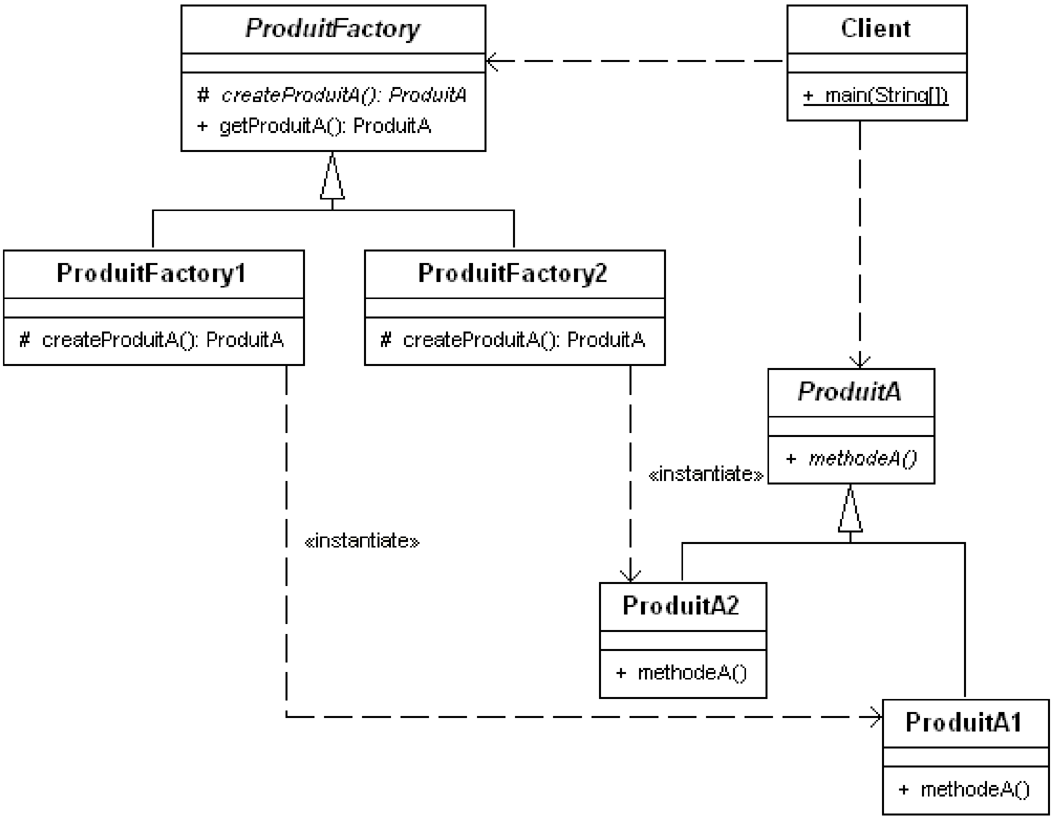
- Le client ne peut déterminer le type d'objet à créer qu'à l'exécution
- Il y a une volonté de centraliser la création des objets

L'utilisation d'une fabrique permet de rendre l'instanciation d'objets plus flexible que l'utilisation de l'opérateur d'instanciation new.

Ce design pattern peut être implémenté sous plusieurs formes dont les deux principales sont :

- Déclarer la fabrique abstraite et laisser une de ses sous-classes créer l'objet
- Déclarer une fabrique dont la méthode de création de l'objet attend les données nécessaires pour déterminer le type de l'objet à instancier

Il est possible d'implémenter la fabrique sous la forme d'une classe abstraite et de définir des sous-classes chargées de réaliser les différentes instanciations.



La classe ProduitFactory propose la méthode getProduitA() qui se charge de retourner l'instance créée par la méthode createProduitA().

Les classes ProduitFactory1 et ProduitFactory2 sont les implémentations concrètes de la fabrique. Elles redéfinissent la méthode createProduitA() pour qu'elle renvoie l'instance du produit.

La classe ProduitA est la classe abstraite mère de tous les produits.

Les classes ProduitA1 et ProduitA2 sont des implémentations concrètes de produits.

Exemple : le code sources des différentes classes

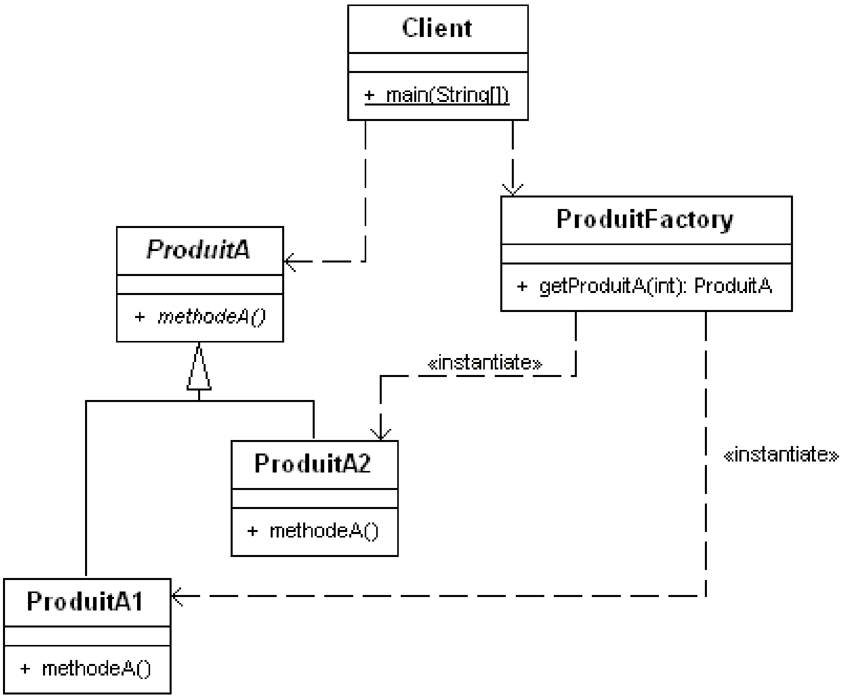
```
package fr.jmdoudoux.dej.factory1;

public class Client {

  public static void main(String[] args) {
    ProduitFactory produitFactory1 = new ProduitFactory1();
    ProduitFactory produitFactory2 = new ProduitFactory2();

    ProduitA produitA = null;

    System.out.println("Utilisation de la premiere fabrique");
    produitA = produitFactory1.getProduitA();
    produitA.methodeA();

    System.out.println("Utilisation de la seconde fabrique");
    produitA = produitFactory2.getProduitA();
    produitA.methodeA();

  }
}
package fr.jmdoudoux.dej.factory1;

public abstract class ProduitFactory {

  public ProduitA getProduitA() {
    return createProduitA();
  }

  protected abstract ProduitA createProduitA();
}

package fr.jmdoudoux.dej.factory1;

public class ProduitFactory1 extends ProduitFactory {

  protected ProduitA createProduitA() {
    return new ProduitA1();
  }
}

package fr.jmdoudoux.dej.factory1;

public class ProduitFactory2 extends ProduitFactory {

  protected ProduitA createProduitA() {
    return new ProduitA2();
  }
}

package fr.jmdoudoux.dej.factory1;

public abstract class ProduitA {

  public abstract void methodeA();
}

package fr.jmdoudoux.dej.factory1;

public class ProduitA1 extends ProduitA {

  public void methodeA() {
    System.out.println("ProduitA1.methodeA()");
  }
}

package fr.jmdoudoux.dej.factory1;

public class ProduitA2 extends ProduitA {

  public void methodeA() {
    System.out.println("ProduitA2.methodeA()");
  }
}
```

Résultat :

```
Utilisation de la premiere fabrique
ProduitA1.methodeA()
Utilisation de la seconde fabrique
ProduitA2.methodeA()
```

Il est possible d'implémenter la fabrique sous la forme d'une classe qui possède une méthode chargée de renvoyer l'instance voulue. La création de cette instance est alors réalisée en fonction de données du contexte (valeurs fournies en paramètres de la méthode, fichier de configuration, paramètres de l'application, ...).

Dans l'exemple ci-dessous, la méthode getProduitA() attend en paramètre une constante qui précise le type d'instance à créer.



Exemple : le code sources des différentes classes

```
package fr.jmdoudoux.dej.factory2;

public class Client {

  public static void main(String[] args) {
    ProduitFactory produitFactory = new ProduitFactory();

    ProduitA produitA = null;

    produitA = produitFactory.getProduitA(ProduitFactory.TYPE_PRODUITA1);
    produitA.methodeA();

    produitA = produitFactory.getProduitA(ProduitFactory.TYPE_PRODUITA2);
    produitA.methodeA();

    produitA = produitFactory.getProduitA(3);
    produitA.methodeA();

  }
}

package fr.jmdoudoux.dej.factory2;

public class ProduitFactory {
  public static final int TYPE_PRODUITA1 = 1;
  public static final int TYPE_PRODUITA2 = 2;

  public ProduitA getProduitA(int typeProduit) {
    ProduitA produitA = null;

    switch (typeProduit) {
      case TYPE_PRODUITA1:
        produitA = new ProduitA1();
        break;
      case TYPE_PRODUITA2:
        produitA = new ProduitA2();
        break;
      default:
        throw new IllegalArgumentException("Type de produit inconnu");
    }

    return produitA;
  }
}

package fr.jmdoudoux.dej.factory2;

public abstract class ProduitA {

  public abstract void methodeA();
}

package fr.jmdoudoux.dej.factory2;

public class ProduitA1 extends ProduitA {

  public void methodeA() {
    System.out.println("ProduitA1.methodeA()");
  }
}

package fr.jmdoudoux.dej.factory2;

public class ProduitA2 extends ProduitA {

  public void methodeA() {
    System.out.println("ProduitA2.methodeA()");
  }
}
```

Résultat :

```
ProduitA1.methodeA()
ProduitA2.methodeA()
java.lang.IllegalArgumentException: Type de produit inconnu
        at fr.jmdoudoux.dej.factory2.ProduitFactory.getProduitA(ProduitFactory.java:19)
        at fr.jmdoudoux.dej.factory2.Client.main(Client.java:16)
Exception in thread "main"
```

Cette implémentation est plus légère à mettre en oeuvre.

Remarque : c'est une bonne pratique de toujours respecter la même convention de nommage dans le nom des fabriques et dans le nom de la méthode qui renvoie l'instance.

105.1.2. Fabrique abstraite (abstract Factory)

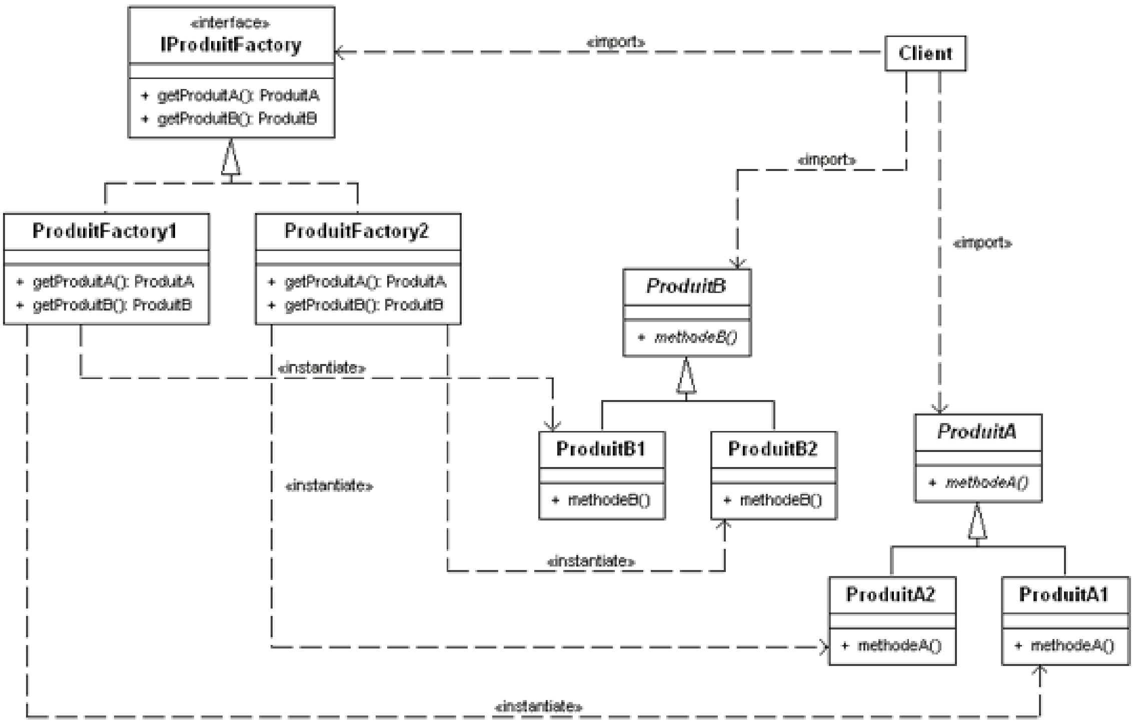
Le motif de conception Abstract Factory (fabrique abstraite) permet de fournir une interface unique pour instancier des objets d'une même famille sans avoir à connaître les classes à instancier.

L'utilisation de ce motif est pertinente lorsque :

- Le système doit être indépendant de la création des objets qu'il utilise
- Le système doit être capable de créer des objets d'une même famille

Le principal avantage de ce motif de conception est d'isoler la création des objets retournés par la fabrique. L'utilisation d'une fabrique abstraite permet de facilement remplacer une fabrique par une autre selon les besoins.

Le motif de conception fabrique abstraite peut être interprété et mis en oeuvre de différentes façons. Le diagramme UML ci-dessous propose une mise en oeuvre possible avec deux familles de deux produits.



Dans cet exemple, les classes suffixées par un chiffre correspondent aux classes relatives à une famille donnée.

Les classes misent en oeuvre sont :

- IProduitFactory : interface pour les fabriques de création d'objets. Elle définit donc les méthodes nécessaires à la création des objets
- ProduitFactory1 et ProduitFactory2 : fabriques qui réalisent la création des objets
- ProduitA et ProduitB : interfaces des deux familles de produits (En Java, cela peut être une classe abstraite ou une interface)
- ProduitA1, ProduitA2, ProduitB1 et ProduitB2 : implémentations des produits des deux familles
- Client : classe qui utilise la fabrique pour obtenir des objets

C'est une des classes filles de la fabrique qui se charge de la création des objets d'une famille. Ainsi tous les objets créés doivent hériter d'une classe abstraite qui sert de modèle pour toutes les classes de la famille.

Le client utilise une implémentation concrète de la fabrique abstraite pour obtenir une instance d'un produit créé par la fabrique.

Cette instance est obligatoirement du type de la classe abstraite dont toutes les classes concrètes héritent. Ainsi des objets concrets sont retournés par la fabrique mais le client ne peut utiliser que leur interface abstraite.

Comme il n'y a pas de relation entre le client et la classe concrète retournée par la fabrique, celle-ci peut renvoyer n'importe quelle classe qui hérite de la classe abstraite.

Ceci permet facilement :

- De remplacer une classe concrète par une autre.
- D'ajouter de nouveaux types d'objets qui héritent de la classe abstraite sans modifier le code utilisé par la fabrique.

Pour prendre en compte une nouvelle famille de produit dans le code client, il suffit simplement d'utiliser la fabrique dédiée à cette famille. Le reste du code client ne change pas. Ceci est beaucoup plus simple que d'avoir à modifier dans le code client l'instanciation des classes concrètes concernées.

Exemple :

```
package fr.jmdoudoux.dej.abstractfactory;

public class Client {

  public static void main(String[] args) {
    IProduitFactory produitFactory1 = new ProduitFactory1();
    IProduitFactory produitFactory2 = new ProduitFactory2();

    ProduitA produitA = null;
    ProduitB produitB = null;

    System.out.println("Utilisation de la premiere fabrique");
    produitA = produitFactory1.getProduitA();
    produitB = produitFactory1.getProduitB();
    produitA.methodeA();
    produitB.methodeB();

    System.out.println("Utilisation de la seconde fabrique");
    produitA = produitFactory2.getProduitA();
    produitB = produitFactory2.getProduitB();
    produitA.methodeA();
    produitB.methodeB();

  }
}

package fr.jmdoudoux.dej.abstractfactory;

public interface IProduitFactory {

  public ProduitA getProduitA();
  public ProduitB getProduitB();
}


package fr.jmdoudoux.dej.abstractfactory;

public class ProduitFactory1 implements IProduitFactory {

  public ProduitA getProduitA() {
    return new ProduitA1();
  }

  public ProduitB getProduitB() {
    return new ProduitB1();
  }
}

package fr.jmdoudoux.dej.abstractfactory;

public class ProduitFactory2 implements IProduitFactory {

  public ProduitA getProduitA() {
    return new ProduitA2();
  }

  public ProduitB getProduitB() {
    return new ProduitB2();
  }
}

package fr.jmdoudoux.dej.abstractfactory;

public abstract class ProduitA {

  public abstract void methodeA();
}
```

```
package fr.jmdoudoux.dej.abstractfactory;

public class ProduitA1 extends ProduitA {

  public void methodeA() {
    System.out.println("ProduitA1.methodeA()");
  }
}

package fr.jmdoudoux.dej.abstractfactory;

public class ProduitA2 extends ProduitA {

  public void methodeA() {
    System.out.println("ProduitA2.methodeA()");
  }
}

package fr.jmdoudoux.dej.abstractfactory;

public abstract class ProduitB {

  public abstract void methodeB();
}

package fr.jmdoudoux.dej.abstractfactory;

public class ProduitB1 extends ProduitB {

  public void methodeB() {
    System.out.println("ProduitB1.methodeB()");
  }
}

package fr.jmdoudoux.dej.abstractfactory;

public class ProduitB2 extends ProduitB {

  public void methodeB() {
    System.out.println("ProduitB2.methodeB()");
  }
}
```

Résultat :

```
Utilisation de la premiere fabrique
ProduitA1.methodeA()
ProduitB1.methodeB()
Utilisation de la seconde fabrique
ProduitA2.methodeA()
ProduitB2.methodeB()
```

Une fabrique concrète est généralement un singleton.

105.1.3. Monteur (Builder)

Cette section sera développée dans une version future de ce document

105.1.4. Prototype (Prototype)

Cette section sera développée dans une version future de ce document

105.1.5. Singleton (Singleton)

Ce motif de conception propose de n'avoir qu'une seule et unique instance d'une classe dans une application.

Le Singleton est fréquemment utilisé dans les applications car il n'est pas rare de ne vouloir qu'une seule instance pour certaines fonctionnalités (pool, cache, ...). Ce modèle est aussi particulièrement utile pour le développement d'objets de type gestionnaire. En effet ce type d'objet doit être unique car il gère d'autres objets par exemple un gestionnaire de logs.

La mise en oeuvre du design pattern Singleton doit :

- assurer qu'il n'existe qu'une seule instance de la classe
- fournir un moyen d'obtenir cette instance unique

Un singleton peut maintenir un état (stateful) ou non (stateless).

La compréhension de ce motif de conception est facile mais son implémentation ne l'est pas toujours, notamment, à cause de quelques subtilités de Java et d'une attention particulière à apporter dans le cas d'une utilisation multithreads.

Ce design pattern peut avoir plusieurs implémentations en Java.

1) une implémentation classique avec initialisation tardive

- le ou les contructeurs ont un attribut de visibilité private pour empêcher toute instanciation de l'extérieur de la classe : ne pas oublier de redéfinir le constructeur par défaut si aucun constructeur n'est explicitement défini
- l'unique instance est une variable de classe
- un getter static permet de renvoyer l'instance et de la créer au besoin
- redefinir la méthode clone pour empêcher son utilisation
- la classe est déclarée final pour empêcher la création d'une classe fille

Exemple :

```
public final class MonSingleton {

  private static MonSingleton instance;

  private MonSingleton() {
    // traitement du constructeur
  }

  public static MonSingleton getInstance() {
    if (instance == null) {
      instance = new MonSingleton();
    }
    return instance;
  }

  @Override
  public Object clone() throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
  }
}
```

Cette implémentation est simple mais malheureusement, elle n'est pas threadsafe. Dans un contexte multithreads, il est possible que les deux premiers appels concomitants puissent créer deux instances. Chaque thread reçoit alors une instance distincte ce qui ne répond pas aux contraintes du design pattern.

2) une implémentation thread-safe classique avec initialisation tardive

Le plus simple et le plus sûr pour éviter ce problème est de sécuriser l'accès au getter avec le mot clé synchronized.

Exemple :

```
public final class MonSingleton {

  private static MonSingleton instance;

  private MonSingleton() {
    // traitement du constructeur
  }

  public static synchronized MonSingleton getlnstance() {
    if (instance == null) {
      instance = new MonSingleton();
    }
    return instance;
  }

  @Override
  public Object clone() throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
  }
}
```

Cette solution est thread-safe mais elle induit un coût en terme de performance, lié à la synchronisation de la méthode, qui peut devenir génant si la méthode est appelée fréquemment de façon concomitante.

3) une implémentation classique non thread-safe avec initialisation tardive

La partie qui doit vraiment être thread safe est la création de l'instance ce qui correspond uniquement à la première invocation de la méthode. Il peut être alors tentant de ne synchroniser que la création de l'instance.

Exemple :

```
public final class MonSingleton {

  private static MonSingleton instance;

  private MonSingleton() {
    // traitement du constructeur
  }

  public static MonSingleton getInstance() {
    if (instance == null) {
      synchronized (MonSingleton.class) {
        instance = new MonSingleton();
      }
    }
    return instance;
  }

  @Override
  public Object clone() throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
  }
}
```

Le but est d'éviter de poser un verrou sur le moniteur de la classe à chaque invocation de la méthode. Malheureusement, cette solution n'est pas thread-safe.

Le thread 1 entre dans le bloc securisé et avant l'assignation de la reference créé par le constructeur à la variable instance, le scheduler passe la main au thread 2 qui teste si l'instance est null et c'est le cas donc il va attendre la sortie du bloc sécurisé du thread 1 pour exécuter à son tour le bloc de code sécurisé. Les deux threads obtiennent chacun une instance distincte.

4) une implémentation classique avec initialisation tardive non thread-safe avec double-checked

Une autre implémentation utilisée est celle nommée double-checked : elle consiste à retester si l'instance est bien null après la pose du verrou au cas ou un autre thread aurait déjà passé le premier test.

Exemple :

```
public final class MonSingleton {

  private static MonSingleton instance;

  private MonSingleton() {
    // traitement du constructeur
  }

  public static MonSingleton getInstance() {
    if (instance == null) {
      synchronized (MonSingleton.class) {
        if (instance == null) {
          instance = new MonSingleton();
        }
      }
    }
    return instance;
  }

  @Override
  public Object clone() throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
  }
}
```

Cette solution, elle aussi, peut ne pas fonctionner non plus correctement si le compilateur, par optimisation, assigne la référence alors que l'objet n'est pas encore initialisé (son constructeur n'est pas encore invoqué).

Ainsi le premier thread pourrait ne pas avoir une instance entièrement initialisée.

5) une implémentation threadsafe avec initialisation au chargement de la classe

Cette implémentation qui exploite une spécificité de Java est simple, rapide et sûre.

Exemple :

```
public final class MonSingleton {

  private static MonSingleton instance = new MonSingleton();

  public static MonSingleton getlnstance() {
    return instance;
  }

  private MonSingleton() {
  }
}
```

Cette implémentation est thread-safe car les spécifications du langage Java impose à la JVM d'avoir initialisée une variable static avant sa première utilisation.

6) une implémentation threadsafe avec initialisation tardive

L'utilisation d'une classe interne statique permet une initialisation tardive garantie par les spécifications de la JVM.

Exemple :

```
public class MonSingleton {
  private MonSingleton() {
  }

  private static class MonSingletonWrapper {
    private final static MonSingleton instance = new MonSingleton();
  }

  public static MonSingleton getInstance() {
    return MonSingletonWrapper.instance;
  }
}
```

Il existe plusieurs précautions à prendre lors de la mise en oeuvre du Singleton. Il est tentant d'utiliser des singletons mais ceux-ci peuvent être à l'origine de certaines difficultés dans des cas biens précis :

- les tests unitaires : il n'est pas facile de créer des mocks de singletons
- la distribution de l'application dans plusieurs JVM : l'utilisation du Singleton peut poser des problèmes car chaque JVM aura son propre Singleton
- le singleton peut être récupéré par le ramasse-miettes dans des JVM antérieures à la version 2. La seule solution dans ce cas est d'empécher le ramassette-miette de récupérer la mémoire des classes chargées (-Xnoclassgc), Ce problème ne concerne pas les JVM 1.3 et supérieures.
- si la classe est chargée par plusieurs classloaders alors plusieurs instances existeront (une pour chaque classloader). Ceci est dû au fait qu'une même classe chargée par deux classloaders sera présente deux fois dans la permgen.

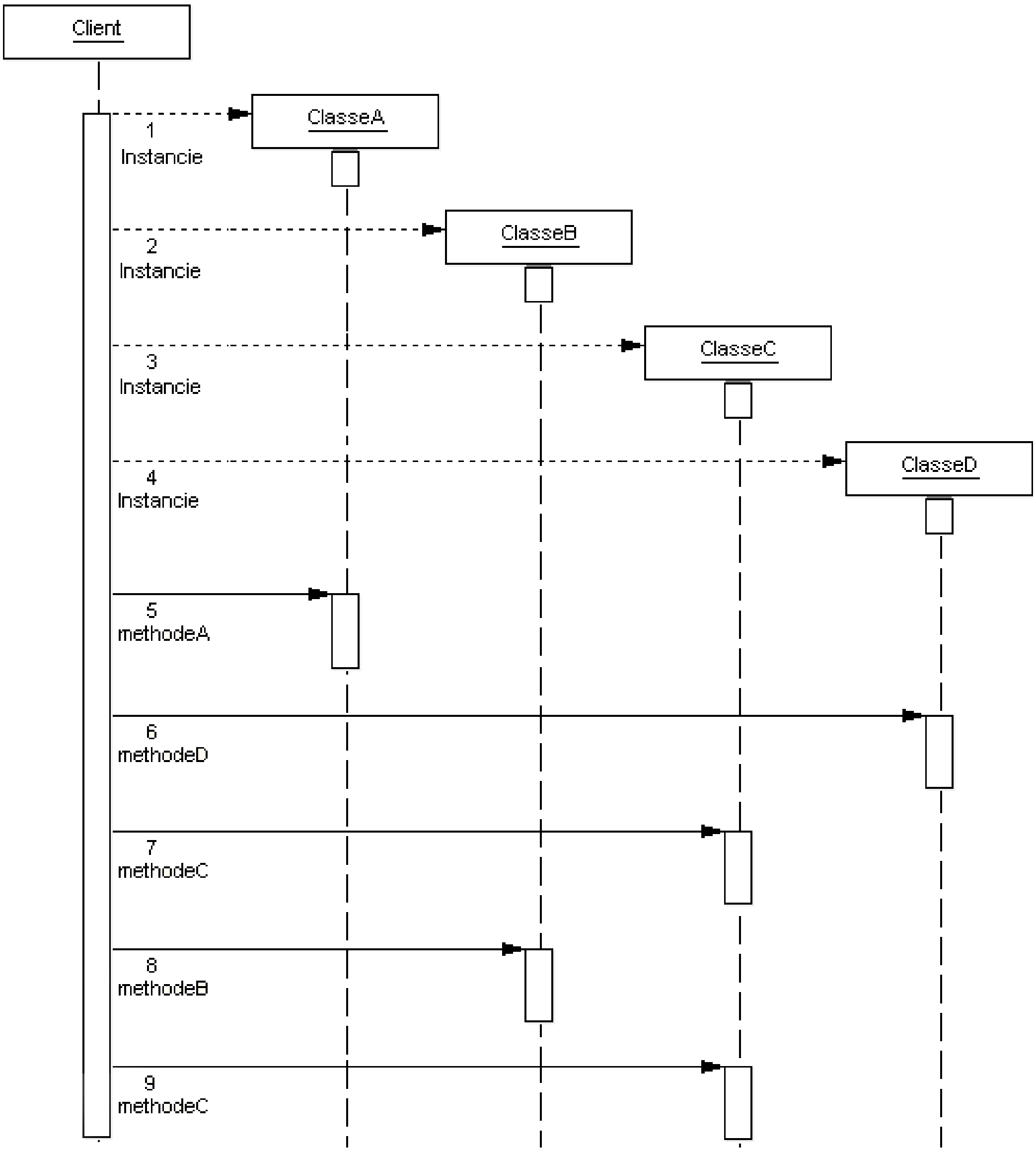
105.2. Les modèles de structuration

105.2.1. Façade (Facade)

Une bonne pratique de conception est d'essayer de limiter le couplage existant entre des fonctionnalités proposées par différentes entités. Dans la pratique, il est préférable de développer un petit nombre de classes et de proposer une classe pour les utiliser. C'est ce que propose le motif de conception façade.

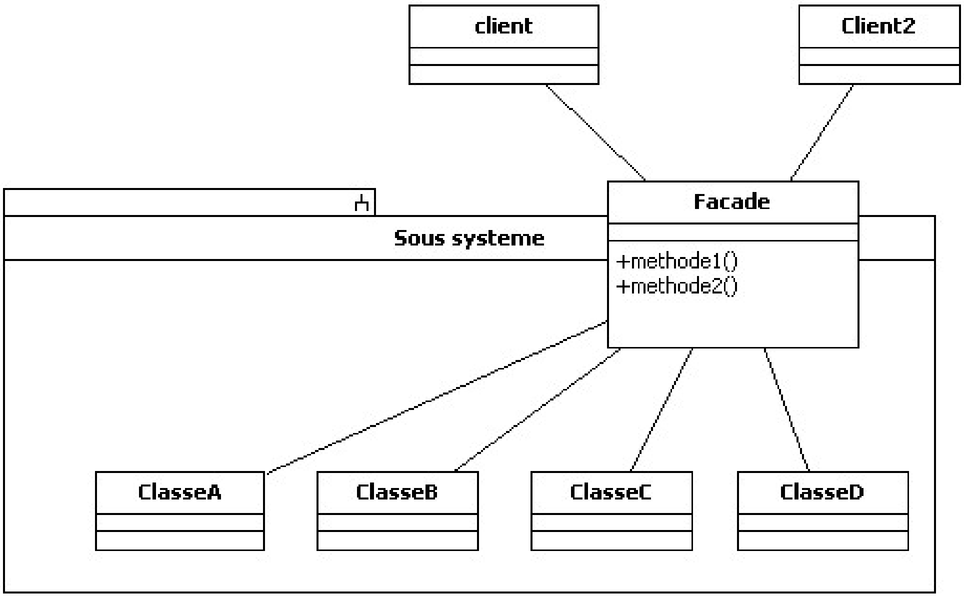
Le but est de proposer une interface facilitant la mise en oeuvre d'un ensemble de classes généralement regroupées dans un ou plusieurs sous-systèmes. Le motif Façade permet d'offrir un niveau d'abstraction entre l'ensemble de classes et celles qui souhaitent les utiliser en proposant une interface de plus haut niveau pour utiliser les classes du sous-système.

Exemple : un client qui utilise des classes d'un sous-système directement



Cet exemple volontairement simpliste va être modifié pour mettre en oeuvre le modèle de conception Façade.

Employer ce modèle aide à simplifier une grande partie de l'interface pour utiliser les classes du sous-système. Il facilite la mise en oeuvre de plusieurs classes en fournissant une couche d'abstraction supplémentaire entre ces dernières et les classes qui les utilisent. Le modèle Façade permet donc de faciliter la compréhension et l'utilisation d'un sous-système complexe que ce soit pour faciliter l'utilisation de tout ou partie du système ou pour forcer une utilisation particulière de celui-ci.



Les classes du sous-système encapsulent les traitements qui seront exécutés par des appels de méthodes de l'objet Façade. Ces classes ne doivent pas connaître ni, de surcroît, avoir de référence sur l'objet Façade.

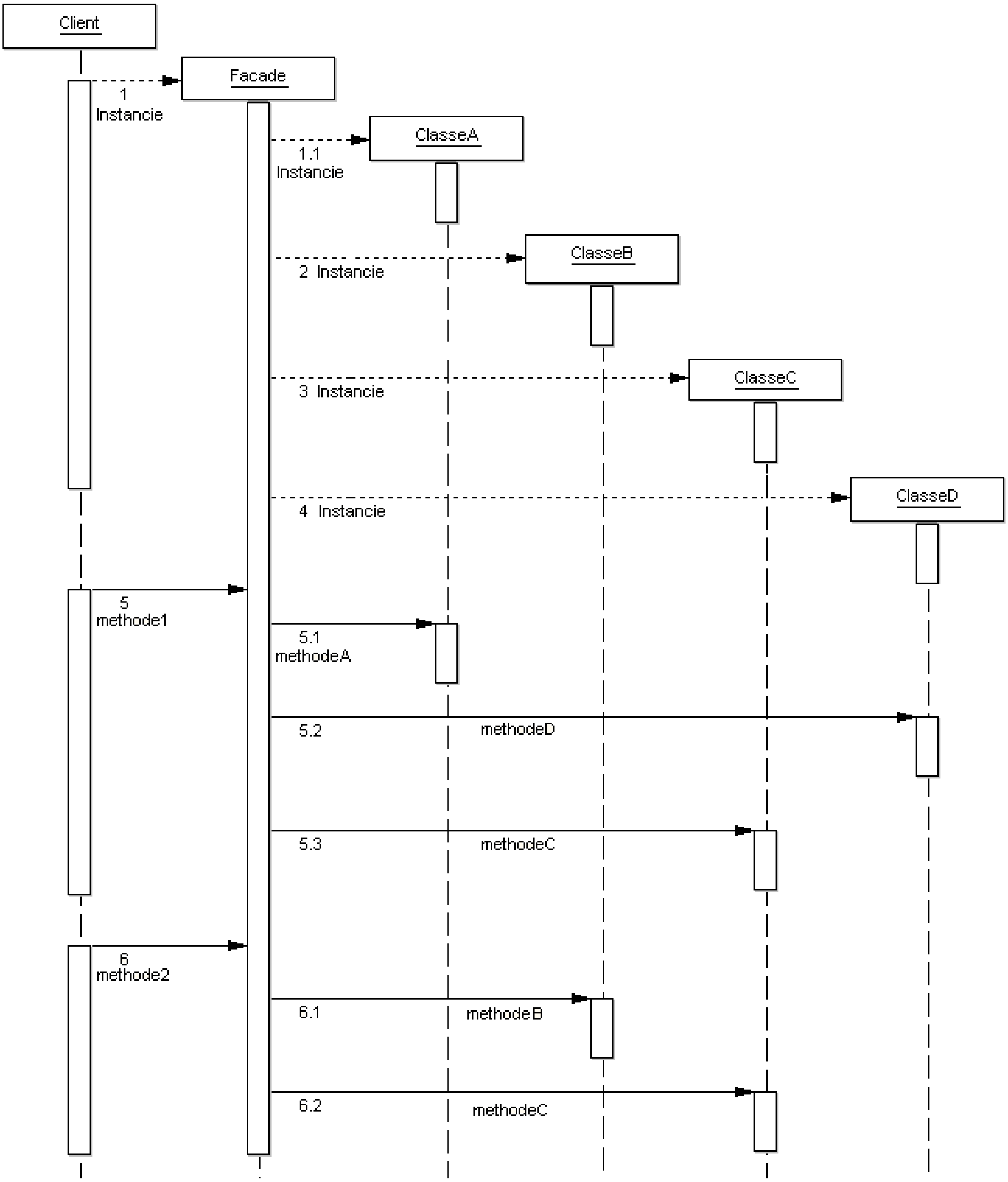
La façade propose un ensemble de méthodes qui vont réaliser les appels nécessaires aux classes du sous-système pour offrir des fonctionnalités cohérentes. Elle propose une interface pour faciliter l'utilisation du sous-système en implémentant les traitements requis pour utiliser les classes de celui-ci.

La classe qui implémente le modèle Façade encapsule les appels aux différentes classes impliquées dans l'exécution d'un traitement cohérent. Elle fait donc office de point d'entrée pour utiliser le sous-système.

Ce modèle requiert plusieurs classes :

- Le client qui va utiliser la façade
- La façade
- Les classes du sous système utilisées par la façade

Exemple :



Le code à utiliser dans la classe client est réduit ce qui va en faciliter la maintenance. La façade masque donc les complexités du sous-système utilisé et fournit une interface simple d'accès pour les clients qui l'utilisent.

Exemple :

```
public class ClientTestFacade {
  public static void main(String[] argv) {
    TestFacade facade = new TestFacade();

    facade.methode1();
    facade.methode2();
  }
}

public class TestFacade {

  ClasseA classeA;
  ClasseB classeB;
  ClasseC classeC;
  ClasseD classeD;

  public TestFacade() {
    classeA = new ClasseA();
    classeB = new ClasseB();
    classeC = new ClasseC();
    classeD = new ClasseD();
  }

  public void methode1() {
    System.out.println("Methode2 : ");
    classeA.methodeA();
    classeD.methodeD();
    classeC.methodeC();
  }

  public void methode2() {
    System.out.println("Methode1 : ");
    classeB.methodeB();
    classeC.methodeC();
  }
}

public class ClasseA {
  public void methodeA() {
    System.out.println(" - MethodeA ClasseA");
  }
}

public class ClasseB {
  public void methodeB() {
    System.out.println(" - MethodeB Classe B");
  }
}

public class ClasseC {
  public void methodeC() {
    System.out.println(" - MethodeC ClasseC");
  }
}

public class ClasseD {
  public void methodeD() {
    System.out.println(" - MethodeD ClasseD");
  }
}
```

Résultat :

```
Methode2 :
 - MethodeA ClasseA
 - MethodeD ClasseD
 - MethodeC ClasseC
Methode1 :
 - MethodeB Classe B
 - MethodeC ClasseC
```

Le modèle Façade peut être utilisé pour :

- Faciliter l'utilisation partielle d'un sous-système complexe ou de plusieurs classes
- Masquer l'existence d'un sous-système
- Ajouter des fonctionnalités sans modifier le sous-système
- Assurer un découplage entre le client et le sous-système (par exemple pour chaque couche d'une architecture logicielle N tiers)

L'utilisation d'une façade permet au client de limiter le nombre d'objets à utiliser puisqu'il se contente simplement d'appeler une ou plusieurs méthodes de la façade. Ce sont ces méthodes qui vont utiliser les classes du sous-système, masquant ainsi au client toute la complexité de leur mise en oeuvre.

Il peut être pratique de définir une façade sans état (les méthodes de la façade n'utilisent pas de membres statiques de la classe) car dans ce cas, une seule et unique instance de la façade peut être définie côté client en mettant en oeuvre le modèle de conception singleton prévu à cet effet.

Il est possible de proposer des fonctionnalités supplémentaires dans la façade qui enrichissent la mise en oeuvre du sous-système.

La façade peut aussi être utilisée pour masquer le sous-système. Elle peut encapsuler les classes du sous-système et ainsi cacher au client l'existence du sous-système. Cette mise en oeuvre facilite le remplacement du sous-système par un autre : il suffit simplement de modifier la façade pour que le client continue à fonctionner.

Il est possible que toutes les fonctionnalités proposées par les classes du sous-système ne soient pas accessibles par la façade : son but est de simplifier leurs utilisations mais pas de proposer toutes les fonctionnalités.

Ce motif de conception est largement utilisé.

105.2.2. Décorateur (Decorator)

Le motif de conception décorateur (decorator en anglais) permet d'ajouter des fonctionnalités à un objet en mettant en oeuvre une solution plus souple que l'héritage : il permet d'ajouter des fonctionnalités à une ou plusieurs méthodes existantes d'une classe dynamiquement.

La programmation orientée objet propose l'héritage pour ajouter des fonctionnalités à une classe, cependant l'héritage présente quelques contraintes et il n'est pas toujours possible de le mettre en oeuvre (par exemple si la classe est finale). L'héritage crée une nouvelle classe qui reprend les fonctionnalités de la classe mère et les modifie ou les enrichie. Mais il présente quelques inconvénients :

- Il n'est pas toujours possible (par exemple pour une classe déclarée finale)
- Cela peut faire augmenter le nombre de classes pour définir tous les cas de figure requis
- L'ajout des fonctionnalités est statique

Avec l'héritage, il serait nécessaire de définir autant de classes filles que de cas ce qui peut vite devenir ingérable. Avec l'utilisation d'un décorateur, il suffit de définir un décorateur pour chaque fonctionnalité et de les utiliser par combinaison en fonction des besoins. L'héritage ajoute des fonctionnalités de façon statique (à la compilation) alors que le décorateur ajoute des fonctionnalités de façon dynamique (à l'exécution).

Le modèle de conception décorateur apporte une solution à ces trois inconvénients et propose donc une alternative à l'héritage.

Le motif de conception décorateur permet de définir un ensemble de classes possédant une base commune mais proposant chacune des variantes sans utiliser l'héritage qui est le mécanisme de base par la programmation orientée objet. Ceci permet d'enrichir une classe avec des fonctionnalités supplémentaires.

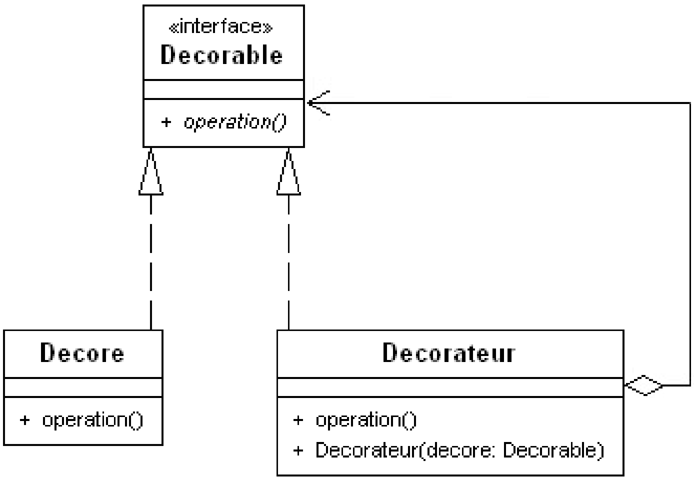
Ce motif est dédié à la création de variantes d'une classe plutôt que d'avoir une seule classe prenant en compte ces variantes. Il permet aussi de réaliser des combinaisons de plusieurs variantes.

Ce motif de conception est donc généralement utilisé lorsqu'il n'est pas possible de prédéfinir le nombre de combinaisons induites par l'ajout de nombreuses fonctionnalités ou si ce nombre est trop important. Le principe du motif de conception décorateur est d'utiliser la composition : le décorateur contient un objet décoré. L'appel d'une méthode du décorateur provoque l'exécution de la méthode correspondante du décoré et des fonctionnalités ajoutées par le décorateur.

Le motif décorateur repose sur deux entités :

- le décoré : interface ou classe qui définit les fonctionnalités de base
- le décorateur : classe enrichie qui contient les fonctionnalités de base plus celles ajoutées

Le décorateur encapsule le décoré dont l'instance est généralement fournie dans les paramètres d'un constructeur. Il est important que l'interface du décorateur reprenne celle de l'objet décoré. Pour permettre de combiner les décorations, le décoré et le décorateur doivent implémenter une interface commune.

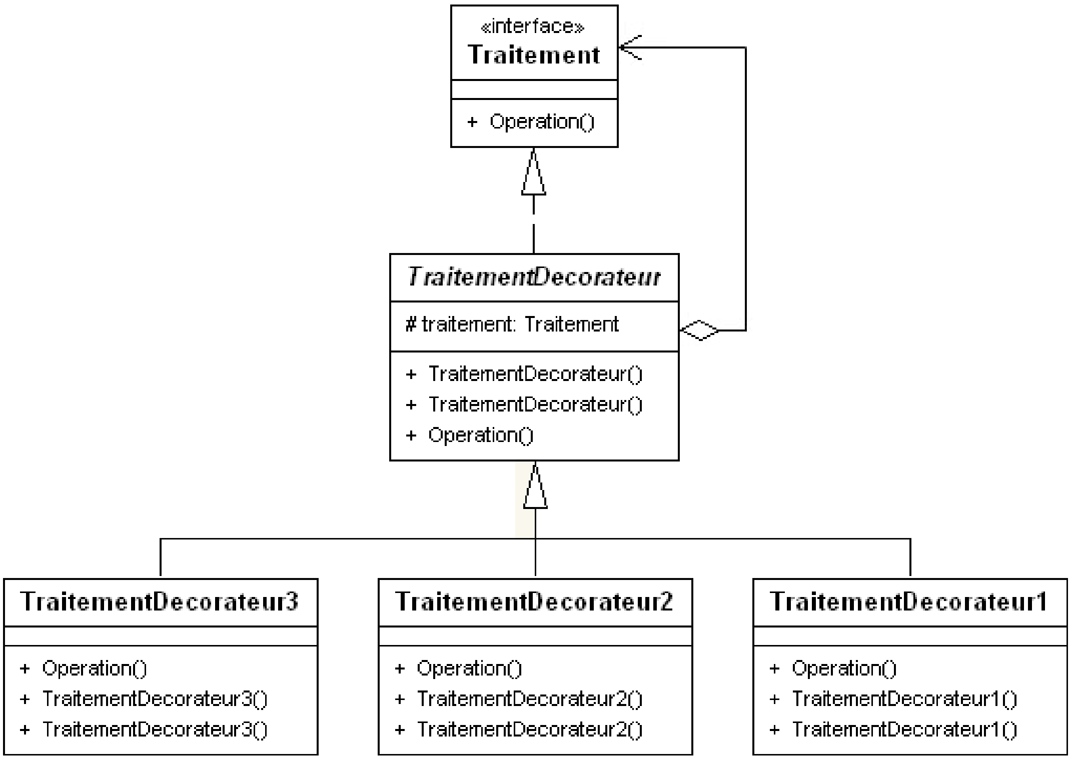


La combinaison peut alors être répétée pour construire un objet qui va contenir les différentes fonctionnalités proposées par les décorateurs utilisés.

Le motif de conception décorateur est particulièrement utile dans plusieurs cas :

- définition de fonctionnalités génériques qui peuvent prendre plusieurs formes
- définition de plusieurs fonctionnalités optionnelles

Il permet de créer un objet qui va être composé des fonctionnalités requises par ajouts successifs des différents décorateurs proposant les fonctionnalités requises.



Un des avantages de ce motif de conception est de n'avoir à créer qu'une seule classe pour proposer des fonctionnalités supplémentaires aux classes qui mettent en oeuvre ce motif. Avec l'héritage, il serait nécessaire de créer autant de classes filles que de classes concernées ou de gérer la fonctionnalité dans une classe mère en modifiant cette dernière pour prendre en compte cet ajout avec tous les risques que cela peut engendrer.

Pour mettre en oeuvre ce motif, il faut :

1) définir une interface qui va déclarer toutes les fonctionnalités des décorés.

Exemple : interface Traitement

```
package fr.jmdoudoux.dej.dp.decorateur;

public interface Traitement {
  public void Operation();
}
```

2) définir un décorateur de base qui implémente l'interface et possède une référence sur une instance de l'interface. Cette référence est le décoré qui va être enrichi des fonctionnalités du décorateur.

Exemple : classe abstraite TraitementDecorateur

```
package fr.jmdoudoux.dej.dp.decorateur;

public abstract class TraitementDecorateur implements Traitement {

  protected Traitement traitement;

  public TraitementDecorateur()
  {
  }

  public TraitementDecorateur(Traitement traitement)
  {
    this.traitement = traitement;
  }

  public void Operation() {
    if (traitement != null)
    {
      traitement.Operation();
    }
  }
}
```

3) définir les décorateurs qui héritent du décorateur de base et implémentent les fonctionnalités supplémentaires qu'ils sont chargés de proposer.

Exemple : TraitementDecorateur1

```
package fr.jmdoudoux.dej.dp.decorateur;

public class TraitementDecorateur1 extends TraitementDecorateur {

  public TraitementDecorateur1() {
    super();
  }

  public TraitementDecorateur1(Traitement traitement) {
    super(traitement);
  }

  @Override
  public void Operation() {
    if (traitement != null)
    {
      traitement.Operation();
    }
    System.out.println("TraitementDecorateur1.Operation()");
  }
}
```

Exemple : TraitementDecorateur2

```
package fr.jmdoudoux.dej.dp.decorateur;
```

```
public class TraitementDecorateur2 extends TraitementDecorateur {

  public TraitementDecorateur2() {
    super();
  }

  public TraitementDecorateur2(Traitement traitement) {
    super(traitement);
  }

  @Override
  public void Operation() {
    if (traitement != null) {
      traitement.Operation();
    }

    System.out.println("TraitementDecorateur2.Operation()");
  }
}
```

Exemple : TraitementDecorateur3

```
package fr.jmdoudoux.dej.dp.decorateur;

public class TraitementDecorateur3 extends TraitementDecorateur {

  public TraitementDecorateur3() {
    super();
  }

  public TraitementDecorateur3(Traitement traitement) {
    super(traitement);
  }

  @Override
  public void Operation() {
    if (traitement != null)
    {
      traitement.Operation();
    }
    System.out.println("TraitementDecorateur3.Operation()");
  }
}
```

Il est possible de fournir une classe d'implémentation par défaut.

Il est pratique d'utiliser le motif de conception fabrique pour construire l'objet décoré finale. Dans ce cas, une implémentation par défaut de l'interface peut être utile.

Exemple : TraitementTest.java

```
package fr.jmdoudoux.dej.dp.decorateur;

public class TraitementTest {

  public static void main(String[] args) {
    System.out.println("traitement 1 2 3");
    Traitement traitement123 = new TraitementDecorateur3(
      new TraitementDecorateur2(new TraitementDecorateur1()));
    traitement123.Operation();

    System.out.println("traitement 1 3");
    Traitement traitement13 = new TraitementDecorateur3(new TraitementDecorateur1());
    traitement13.Operation();
  }
}
```

Résultat d'exécution :

```
traitement 1 2 3
TraitementDecorateur1.Operation()
TraitementDecorateur2.Operation()
TraitementDecorateur3.Operation()
traitement 1 3
TraitementDecorateur1.Operation()
TraitementDecorateur3.Operation()
```

L'API de base de Java utilise le motif de conception décorateur notamment dans l'API IO

105.3. Les modèles de comportement

La suite de ce chapitre sera développée dans une version future de ce document

106. Des normes de développement

Chapitre 106

Niveau : 🟩🟩⬜⬜⬜⬜ Elémentaire

Le but de ce chapitre est de proposer un ensemble de conventions et de règles pour faciliter la compréhension et donc la maintenance du code.

Ces règles ne sont pas à suivre explicitement à la lettre : elles sont uniquement présentées pour inciter les développeurs à définir et à utiliser des règles dans la réalisation du code surtout dans le cadre d'un travail en équipe. Les règles proposées sont celles couramment utilisées. Il n'existe cependant pas de règle absolue et chacun pourra utiliser tout ou partie des règles proposées.

La définition de conventions et de règles est importante pour plusieurs raisons :

- La majorité du temps passé à coder est consacrée à la maintenance évolutive et corrective d'une application
- Ce n'est pas toujours, voire rarement, l'auteur du code qui effectue ces maintenances
- ces règles facilitent la lisibilité et donc la compréhension du code

Le contenu de ce document est largement inspiré par les conventions de codage historiquement proposées par Sun.

Ce chapitre contient plusieurs sections :

- ♦ Les fichiers
- ♦ La documentation du code
- ♦ Les déclarations
- ♦ Les séparateurs
- ♦ Les traitements
- ♦ Les règles de programmation

106.1. Les fichiers

Java utilise des fichiers pour stocker les sources et le bytecode des classes.

106.1.1. Les packages

Les packages permettent de grouper les classes sous une forme hiérarchisée. Le choix des critères de regroupement est laissé aux développeurs.

Il est préférable de regrouper les classes par packages selon des critères fonctionnels.

Les fichiers inclus dans un package doivent être insérer dans une arborescence de répertoires équivalente.

106.1.2. Les noms de fichiers

Chaque fichier source ne doit contenir qu'une seule classe ou interface publique. Le nom du fichier doit être identique au nom de cette classe ou interface publique en respectant la casse.

Il faut éviter d'utiliser dans ce nom des caractères accentués qui ne sont pas toujours utilisables par tous les systèmes d'exploitation.

Les fichiers sources ont pour extension .java car le compilateur javac fourni avec le J.D.K. utilise cette extension.

Exemple :

```
javac MaClasse.java
```

Les fichiers binaires contenant le bytecode ont pour extension .class car le compilateur génère un fichier avec cette extension à partir du fichier source .java correspondant. De plus, elle est obligatoire pour l'interpréteur Java qui l'ajoute automatiquement au nom du fichier fourni en paramètre.

Exemple :

```
java MaClasse
```

106.1.3. Le contenu des fichiers sources

Un fichier ne devrait pas contenir plus de 2 000 lignes de code.

Des interfaces ou classes privées ayant une relation avec la classe publique peuvent être rassemblées dans un même fichier. Dans ce cas, la classe publique doit être la première dans le fichier.

Chaque fichier source devrait contenir dans l'ordre :

1. un commentaire concernant le fichier
2. les clauses concernant la gestion des packages (la déclaration et les importations)
3. les déclarations de classes ou de l'interface

106.1.4. Les commentaires de début de fichier

Chaque fichier source devrait commencer par un commentaire multiligne contenant au minimum des informations sur le nom de la classe, la version, la date, éventuellement le copyright et tous les autres commentaires utiles :

Exemple :

```
/*
 * Nom de classe : MaClasse
 *
 * Description   : description de la classe et de son rôle
 *
 * Version       : 1.0
 *
 * Date          : 23/02/2001
 *
 * Copyright     : moi
 */
```

106.1.5. Les clauses concernant les packages.

La première ligne de code du fichier devrait être une clause package indiquant à quel paquetage appartient la classe. Le fichier source doit obligatoirement être inclus dans une arborescence correspondante au nom du package.

Il faut indiquer ensuite l'ensemble des paquetages à importer : ceux dont les classes vont être utilisées dans le code.

Exemple :

```
package monpackage;

import java.util.*;
import java.text.*;
```

106.1.6. La déclaration des classes et des interfaces

Les différents éléments qui composent la définition de la classe ou de l'interface devraient être indiqués dans l'ordre suivant :

1. les commentaires au format javadoc de la classe ou de l'interface
2. la déclaration de la classe ou de l'interface
3. les variables de classes (déclarées avec le mot clé static) triées par ordre d'accessibilité : d'abord les variables déclarées public, protected, package-private (sans modificateur d'accès) et enfin private
4. les variables d'instances triées par ordre d'accessibilité : d'abord les variables déclarées public, protected, package friendly (sans modificateur d'accès) et enfin private
5. le ou les constructeurs
6. les méthodes : elles seront regroupées par fonctionnalités plutôt que selon leur accessibilité

106.2. La documentation du code

Il existe deux types de commentaires en Java :

- les commentaires de documentation : ils permettent en respectant quelques règles d'utiliser l'outil javadoc fourni avec le J.D.K. qui formate une documentation des classes, indépendante de l'implémentation du code,
- les commentaires de traitements : ils fournissent un complément d'information dans le code lui-même.

Les commentaires ne doivent pas être entourés par de grands cadres dessinés avec des étoiles ou d'autres caractères.

Les commentaires ne devraient pas contenir de caractères spéciaux tels que le saut de page.

106.2.1. Les commentaires de documentation

Les commentaires de documentation utilisent une syntaxe particulière utilisée par l'outil javadoc pour produire une documentation standardisée des classes et interfaces au format HTML. La documentation de l'API du J.D.K. est le résultat de l'utilisation de cet outil de documentation

106.2.1.1. L'utilisation des commentaires de documentation

Cette documentation concerne les classes, les interfaces, les constructeurs, les méthodes et les champs.

La documentation est définie entre les caractères /** et */ selon le format suivant :

Exemple :

```
/**
 *  Description de la methode
 */
public void maMethode() {
```

La première ligne de commentaires ne doit contenir que /**

Les lignes de commentaires suivantes doivent obligatoirement commencer par un espace et une étoile. Toutes les premières étoiles doivent être alignées.

La dernière ligne de commentaires ne doit contenir que */ précédé d'un espace.

Un tel commentaire doit être défini pour chaque entité : une classe, une interface et chaque membre (variables et méthodes).

Javadoc définit un certain nombre de tags qu'il est possible d'utiliser pour apporter des précisions sur plusieurs informations.

Ces tags permettent de définir des caractéristiques normalisées. Il est possible d'inclure dans les commentaires des tags HTML de mise en forme (PRE, TT, EM ...) mais il n'est pas recommandé d'utiliser des tags HTML de structure tels que Hn, HR, TABLE ... qui sont utilisés par javadoc pour formater la documentation.

Il faut obligatoirement faire précéder l'entité documentée par son commentaire car l'outil associe la documentation à la déclaration de l'entité qui le suit.

106.2.1.2. Les commentaires pour une classe ou une interface

Pour les classes ou interfaces, javadoc définit les tags suivants : @see, @version et @author.

Exemple :

```
/**
 * description de la classe.
 * explication supplémentaire si nécessaire
 *
 * @version 1.0
 *
 * @see UneAutreClasse
 * @author Jean Michel D.
 */
```

106.2.1.3. Les commentaires pour une méthode

Pour les méthodes, javadoc définit les tags suivants : @see, @param, @return, @exception et @author

Exemple :

```
/**
 * description de la méthode.
 * explication supplémentaire si nécessaire
 *
 * @return      description de la valeur de retour
 * @param       arg1 description du 1er argument
 *    :           :         :
 * @param       argN description du Neme argument
 * @exception   Exception1  description de la première exception
 *    :           :         :
 * @exception ExceptionN  description de la Neme exception
 *
 * @see UneAutreClasse#UneAutreMethode
 * @author   Jean Dupond
 */
```

Remarques :

- @return ne doit pas être utilisé avec les constructeurs et les méthodes sans valeur de retour (void)
- @param ne doit pas être utilisé s'il n'y a pas de paramètres

- @exception ne doit pas être utilisé s'il n'y pas d'exception propagée par la méthode
- @author doit être omis s'il est identique à celui du tag @author de la classe

106.2.2. Les commentaires de traitements

Ces commentaires doivent ajouter du sens et des précisions au code : ils ne doivent pas reprendre ce que le code exprime mais expliquer clairement son rôle.

Tous les commentaires utiles à une meilleure compréhension du code et non inclus dans les commentaires de documentation seront insérés avec des commentaires de traitements. Il existe plusieurs styles de commentaires :

- les commentaires sur une ligne
- les commentaires sur une portion de ligne
- les commentaires multi-lignes

Il est conseillé de mettre un espace après le délimiteur de début de commentaires et avant le délimiteur de fin de commentaires lorsqu'il y en a un, afin d'améliorer sa lisibilité.

106.2.2.1. Les commentaires sur une ligne

Ces commentaires sont définis entre les caractères /* et */ sur une même ligne

Exemple :

```
if (i < 10) {
   /* commentaires utiles au code */
   ...
}
```

Ce type de commentaires doit être précédé d'une ligne blanche et doit suivre le niveau d'indentation courant.

106.2.2.2. Les commentaires sur une portion de ligne

Ce type de commentaires peut apparaître sur la ligne de code qu'elle commente mais il faut inclure un espace conséquent qui permette de séparer le code et le commentaire.

Exemple :

```
i++;                      /* commentaires utiles au code */
```

Si plusieurs lignes qui se suivent contiennent chacune un tel commentaire, il faut les aligner :

Exemple :

```
i++;                      /* commentaires utiles au code */
j++;                      /* second commentaires utiles au code */
```

106.2.2.3. Les commentaires multi-lignes

Exemple :

```
/*
 * Commentaires utiles au code
 */
```

Ce type de commentaires doit être précédé d'une ligne blanche et doit suivre le niveau d'indentation courant.

106.2.2.4. Les commentaires de fin de ligne

Ce type de commentaire peut délimiter un commentaire sur une ligne complète ou une fin de ligne.

Exemple :

```
i++;         // commentaires utiles au code
```

Ce type de commentaires peut apparaître sur la ligne de code qu'elle commente mais il faut inclure un espace conséquent qui permette de séparer le code et le commentaire.

Si plusieurs lignes qui se suivent contiennent chacune un tel commentaire, il faut les aligner :

Exemple :

```
i++;                      // commentaires utiles au code
j++;                      // second commentaires utiles au code
```

L'usage de cette forme de commentaires est fortement recommandé car il est possible d'inclure celui-ci dans un autre de la forme /* */ et ainsi mettre en commentaire un morceau de code incluant déjà des commentaires.

106.3. Les déclarations

106.3.1. La déclaration des variables

Il n'est pas recommandé d'utiliser des caractères accentués dans les identifiants de variables, cela peut éventuellement poser des problèmes dans le cas où le code est édité sur des systèmes d'exploitation qui ne les gèrent pas correctement.

Il ne doit y avoir qu'une seule déclaration d'entité par ligne.

Exemple :

```
String nom;
String prenom;
```

Cet exemple est préférable à

Exemple :

```
String nom, prenom;          //ce type de déclaration n'est pas recommandé
```

Il faut éviter de déclarer des variables de types différents sur la même ligne même si cela est accepté par le compilateur.

Exemple :

```
int age, notes[];            // ce type de déclaration est à éviter
```

Il est préférable d'aligner le type, l'identifiant de l'objet et les commentaires si plusieurs déclarations se suivent pour retrouver plus facilement les divers éléments.

Exemple :

```
String          nom         //nom de l'eleve
String          prenom      //prenom de l'eleve
int             notes[]     //notes de l'eleve
```

Il est fortement recommandé d'initialiser les variables au moment de leur déclaration.

Il est préférable de rassembler toutes les déclarations d'un bloc au début de ce bloc. (un bloc est un morceau de code entouré par des accolades).

La seule exception concerne la déclaration de la variable utilisée comme index dans une boucle.

Exemple :

```
for (int i = 0 ; i < 9 ; i++) { ... }
```

Il faut proscrire la déclaration d'une variable qui masque une variable définie dans un bloc parent afin de ne pas complexifier inutilement le code.

Exemple :

```
int taille;
...
void maMethode() {
    int taille;
}
```

106.3.2. La déclaration des classes et des méthodes

Il ne doit pas y avoir d'espaces entre le nom d'une méthode et sa parenthèse ouvrante.

L'accolade ouvrante qui définit le début du bloc de code doit être à la fin de la ligne de déclaration.

L'accolade fermante doit être sur une ligne séparée dont le niveau d'indentation correspond à celui de la déclaration.

Une exception tolérée concerne un bloc de code vide : dans ce cas les deux accolades peuvent être sur la même ligne.

La déclaration d'une méthode est précédée d'une ligne blanche.

Exemple :

```
class MaClasse extends MaClasseMere {
   String nom;
   String prenom;

   MaClasse(String nom, String prenom) {
      this.nom = nom;
      this.prenom = prenom;
   }

   void neRienFaire() {}
}
```

Il faut éviter d'écrire des méthodes longues et compliquées : le traitement réalisé par une méthode doit être simple et fonctionnel. Cela permet d'écrire des méthodes réutilisables dans la classe et facilite la maintenance. Cela permet aussi d'éviter la redondance de code.

Java propose deux syntaxes pour déclarer une méthode qui retourne un tableau : la première syntaxe est préférable.

Exemple :

```
public int[] notes() {     // utiliser cette forme
public int notes()[] {
```

Il est fortement recommandé de toujours initialiser les variables locales d'une méthode lors de leur déclaration car contrairement aux variables d'instances, elles ne sont pas implicitement initialisées avec une valeur par défaut selon leur type.

106.3.3. La déclaration des constructeurs

Elle suit les mêmes règles que celles utilisées pour les méthodes.

Il est préférable de définir explicitement le constructeur par défaut (le constructeur sans paramètre). Soit le constructeur par défaut est fourni par le compilateur et dans ce cas il serait préférable de le définir soit il existe d'autres constructeurs et dans ce cas le compilateur ne définit pas de constructeur par défaut.

Il est préférable de toujours initialiser les variables d'instance dans un constructeur soit avec les valeurs fournies en paramètres du constructeur soit avec des valeurs par défaut.

Exemple :

```
class Personne {
   String nom;
   String prenom;
   int    age;

   Personne() {
      this( "Inconnu", "inconnu", -1 );
   }

   Personne( String nom, String prenom, int age ) {
      this.name    = nom;
      this.address = prenom;
      this.age     = age;
   }
}
```

Il est possible d'appeler un constructeur dans un autre constructeur pour faciliter l'écriture.

Il est recommandé de toujours appeler explicitement le constructeur hérité lors de la redéfinition d'un constructeur dans une classe fille grâce à l'utilisation du mot clé super.

Exemple :

```
class Employe extends Personne {

   int matricule;

   Employee() {
      super();
      matricule = -1;
   }

   Employee(String nom, String prenom, int age, int matricule) {
      super(nom, prenom, age);
      this.matricule = matricule;
   }
}
```

Il est conseillé de ne mettre que du code d'initialisation des variables d'instances dans un constructeur et de mettre les traitements dans des méthodes qui seront appelées après la création de l'objet.

106.3.4. Les conventions de nommage des entités

Les conventions de nommage des entités permettent de rendre les programmes plus lisibles et plus faciles à comprendre. Ces conventions permettent notamment de déterminer rapidement quelle entité désigne un identifiant, une classe ou une méthode.

Entités	Règles	Exemple
Les packages	Toujours écrits tout en minuscules (norme Java 1.2)	com.entreprise.projet
Les classes, les interfaces et les constructeurs	La première lettre est en majuscule. Si le nom est composé de plusieurs mots, la première lettre de chaque mot doit être en majuscule, ne pas mettre de caractère underscore '_' Le nom d'une classe peut finir par Impl pour la distinguer d'une interface qu'elle implémente. Les classes qui définissent des exceptions doivent finir par Exception.	MaClasse MonInterface MaClasse()
Les méthodes	Leur nom devrait contenir un verbe. La première lettre est obligatoirement une minuscule. Si le nom est composé de plusieurs mots, la première lettre de chaque mot doit être en majuscule sans mettre de caractère underscore '_' Les méthodes pour obtenir la valeur d'un champ doivent commencer par get suivi du nom du champ. Les méthodes pour mettre à jour la valeur d'un champ doivent commencer par set suivi du nom du champ Les méthodes pour créer des objets (factory) devraient commencer par new ou create Les méthodes de conversion devraient commencer par to suivi par le nom de la classe renvoyée à la suite de la conversion	public float calculerMontant() {

Les variables	La première lettre est obligatoirement une minuscule et ne devrait pas être un caractère dollar '$' ou underscore '_' même si ceux-ci sont autorisés.
	Pour les variables d'instances non publiques, certains recommandent de commencer par un underscore pour éviter la confusion avec le nom d'une variable fournie en paramètre d'une méthode telle que le setter.
	
	Si le nom est composé de plusieurs mots, la première lettre de chaque mot doit être en majuscule, ne pas mettre de caractère underscore '_'.
	
	Les noms de variables composés d'un seul caractère doivent être évités sauf pour des variables provisoires (index d'une boucle).
	Les noms communs pour ces variables provisoires
		String nomPersonne;
		Date dateDeNaissance;
		int i;

	sont i, j, k, m et n pour les entiers et c, d et e pour les caractères.	

Les constantes	Toujours en majuscules, chaque mots est séparés par un underscore '_'. Ces variables doivent obligatoirement être initialisées lors de leur déclaration.	static final int VAL_MIN = 0;
		static final int VAL_MAX = 9;

106.4. Les séparateurs

L'usage des séparateurs tels que les retours à la ligne, les lignes blanches, les espaces, etc ... permet de rendre le code moins « dense » et donc plus lisibles.

106.4.1. L'indentation

L'unité d'indentation est constituée de 4 espaces. Il n'est pas recommandé d'utiliser les tabulations pour l'indentation.

Il est préférable d'éviter les lignes contenant plus de 80 caractères.

106.4.2. Les lignes blanches

Elles permettent de définir des sections dans le code pour effectuer des séparations logiques.

Deux lignes blanches devraient toujours séparer deux sections d'un fichier source et les définitions des classes et des interfaces.

Une ligne blanche devrait toujours être utilisée dans les cas suivants :

- avant la déclaration d'une méthode,
- entre les déclarations des variables locales et la première ligne de code,
- avant un commentaire d'une seule ligne,
- avant chaque section logique dans le code d'une méthode.

106.4.3. Les espaces

Un espace vide devrait toujours être utilisé dans les cas suivants :

- entre un mot clé et une parenthèse.

Exemple :

```
while (i < 10)
```

- après chaque virgule dans une liste d'argument
- tous les opérateurs binaires doivent avoir un blanc qui les précèdent et qui les suivent

Exemple :

```
      a = (b + c) * d
```

- chaque expression dans une boucle for doit être séparée par un espace

Exemple :

```
for (int i; i < 10; i++)
```

- les conversions de type explicites (cast) doivent être suivies d'un espace

Exemple :

```
i = ((int) (valeur + 10));
```

Il ne faut pas mettre d'espace entre un nom de méthode et sa parenthèse ouvrante.

Il ne faut pas non plus mettre de blanc avant les opérateurs unaires tels que les opérateurs d'incrément '++' et de décrément '--'.

Exemple :

```
i++;
```

106.4.4. La coupure de lignes

Il arrive parfois qu'une ligne de code soit très longue (supérieure à 80 caractères).

Dans ce cas, il est recommandé de couper cette ligne en une ou plusieurs en respectant quelques règles :

- couper la ligne après une virgule ou avant un opérateur
- aligner le début de la nouvelle ligne avec le début de l'expression coupée

Exemple :

```
  maMethode(parametre1, parametre2, parametre3,
            parametre4, parametre5);
```

106.5. Les traitements

Même s'il est possible de mettre plusieurs traitements sur une ligne, chaque ligne ne devrait contenir qu'un seul traitement :

Exemple :

```
i = getSize();
i++;
```

106.5.1. Les instructions composées

Elles correspondent à des instructions qui utilisent des blocs de code.

Les instructions incluses dans ce bloc sont encadrées par des accolades et doivent être indentées.
L'accolade ouvrante doit se situer à la fin de la ligne qui contient l'instruction composée.
L'accolade fermante doit être sur une ligne séparée au même niveau d'indentation que l'instruction composée.

Un bloc de code doit être définit pour chaque traitement même si le traitement ne contient qu'une seule instruction. Cela facilite l'ajout d'instructions et évite des erreurs de programmation.

106.5.2. L'instruction return

Elle ne devrait pas utiliser de parenthèses sauf si celles-ci facilitent la compréhension.

Exemple :

```
  return;
  return valeur;
  return (isHomme() ? 'M' : 'F');
```

106.5.3. L'instruction if

Elle devrait avoir une des formes suivantes :

Exemple :

```
  if (condition) {
     traitements;
  }

  if (condition) {
     traitements;
  } else {
     traitements;
  }

  if (condition) {
     traitements;
  } else if (condition) {
     traitements;
  } else {
     traitements;
  }
```

Même si cette forme est syntaxiquement correcte, il est préférable de ne pas utiliser l'instruction if sans accolades :

Exemple :

```
    if (i == 10) i = 0;  // cette forme ne doit pas être utilisée
```

106.5.4. L'instruction for

Elle devrait avoir la forme suivante :

Exemple :

```
for ( initialisation; condition; mise à jour) {
     traitements;
}
```

106.5.5. L'instruction while

Elle devrait avoir la forme suivante :

Exemple :

```
while (condition) {
     traitements;
}
```

106.5.6. L'instruction do-while

Elle devrait avoir la forme suivante :

Exemple :

```
do {
   traitements;
} while ( condition);
```

106.5.7. L'instruction switch

Elle devrait avoir la forme suivante :

Exemple :

```
switch (condition) {
case ABC:
   traitements;
   break;
case DEF:
   traitements;
   break;
case XYZ:
   traitements;
   break;
default:
   traitements;
   break;
}
```

Il est préférable de terminer les traitements de chaque cas avec une instruction break et de l'enlever au besoin plutôt que d'oublier une instruction break nécessaire.

Toutes les instructions switch devrait avoir un cas 'default' en fin d'instruction : le traitement de tous les cas est une bonne pratique de programmation.

Même si elle est redondante, une instruction break devrait être incluse en fin des traitements du cas 'default' afin de généraliser la première recommandation.

106.5.8. Les instructions try-catch

Elle devrait avoir la forme suivante :

Exemple :

```
try {
   traitements;
} catch (Exception1 e1) {
   traitements;
} catch (Exception2 e2) {
   traitements;
} finally {
   traitements;
}
```

106.6. Les règles de programmation

106.6.1. Le respect des règles d'encapsulation

Il ne faut pas déclarer de variables d'instances ou de classes publiques sans raison valable.

Il est préférable de restreindre l'accès à la variable avec un modificateur d'accès protected ou private et de déclarer des méthodes respectant les conventions instaurées dans les javaBeans : getXxx() ou isXxx() pour obtenir la valeur et setXxx() pour mettre à jour la valeur.

La création de méthodes sur des variables private ou protected permet d'assurer une protection lors de l'accès à la variable (déclaration des méthodes d'accès synchronized) et éventuellement un contrôle lors de la mise à jour de la valeur.

106.6.2. Les références aux variables et méthodes de classes.

Il n'est pas recommandé d'utiliser des variables ou des méthodes de classes à partir d'un objet instancié : il ne faut pas utiliser objet.methode() mais classe.methode().

Exemple à ne pas utiliser si afficher() est une méthode de classe :

```
MaClasse maClasse = new MaClasse();
maClasse.afficher();
```

Exemple à utiliser si afficher() est une méthode de classe :

```
MaClasse.afficher();
```

106.6.3. Les constantes

Il est préférable de ne pas utiliser des constantes numériques codées en dur dans le code mais de déclarer des constantes avec des noms explicites. Une exception concerne les valeurs -1, 0 et 1 dans les boucles for.

106.6.4. L'assignement des variables

Il n'est pas recommandé d'assigner la même valeur à plusieurs variables sur la même ligne :

Exemple :

```
i = j = k;                 //cette forme n'est pas recommandée
```

Il ne faut pas utiliser l'opérateur d'assignement imbriqué.

Exemple à proscrire :

```
  valeur = (i = j + k ) + m;
```

Exemple :

```
  i = j + k;
  valeur = i + m;
```

Il n'est pas recommandé d'utiliser l'opérateur d'assignation = dans une instruction if ou while afin d'éviter toute confusion.

106.6.5. L'usage des parenthèses

Il est préférable d'utiliser les parenthèses lors de l'usage de plusieurs opérateurs pour éviter des problèmes liés à la priorité des opérateurs.

Exemple :

```
if (i == j && m == n)                // à éviter
if ( (i == j) && (m == n ))          // à utiliser
```

106.6.6. La valeur de retour

Il est préférable de minimiser le nombre d'instructions return dans un bloc de code.

Exemple à eviter :

```
if (isValide()){
   return true;
} else {
  return false;
}
```

Exemple :

```
return isValide();
```

Exemple :

```
if (isValide()) {
   return x;
} else return y;
```

Exemple à utiliser :

```
return (isValide() ? x : y)
```

106.6.7. La codification de la condition dans l'opérateur ternaire ? :

Si la condition dans un opérateur ternaire ? : contient un opérateur binaire, cette condition doit être mise entre parenthèses.

Exemple :

```
( i >= 0 ) ? i : -i;
```

106.6.8. La déclaration d'un tableau

Java permet de déclarer les tableaux de deux façons :

Exemple :

```
public int[] tableau = new int[10];
public int tableau[] = new int[10];
```

L'usage de la première forme est recommandé.

107. Les techniques de développement spécifiques à Java

Chapitre 107

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Le développement en Java requiert la mise en oeuvre de quelques techniques particulières dédiées à ce langage.

Ce chapitre couvre des techniques de développement spécifiques à Java. Ces techniques ne concernent que Java et plus particulièrement certaines de ses particularités.

Ce chapitre contient plusieurs sections :

- ♦ L'écriture d'une classe dont les instances seront immuables
- ♦ La redéfinition des méthodes equals() et hashCode()
- ♦ Le clonage d'un objet

107.1. L'écriture d'une classe dont les instances seront immuables

Un objet immuable est un objet dont on ne peut plus modifier l'état une fois l'instance créée.

Pour rendre un objet immuable, il faut respecter plusieurs consignes lors de l'écriture de sa classe :

- elle doit être final pour empêcher la création d'une classe fille qui permettrait de modifier son état en ajoutant ou en redéfinissant des méthodes
- tous les champs doivent être private pour empêcher l'accès aux données sans passer par une méthode de la classe
- tous les champs devraient être final pour éviter toute modification après leur initialisation
- elle ne doit pas proposer de setter ni de méthodes qui pourraient modifier l'état de l'objet
- il faut toujours renvoyer une nouvelle instance pour une méthode qui modifie les données de la classe
- il ne faut pas implémenter l'interface Cloneable
- il faut que les getter d'une propriété de type objet renvoient une version immuable ou une autre instance qui soit une copie

Les objets immuables possèdent plusieurs avantages :

- ils ne peuvent avoir qu'un seul état
- ils sont faciles à créer, à tester et à utiliser
- ils sont thread-safe
- leur valeur de hachage reste fixe et peut même être mise en cache une fois calculée dans le constructeur : ils sont donc à privilégier dans les collections de type Map et Set

Toutes les classes de type wrapper du package java.lang sont immuables : Boolean, Byte, Character, Double, Float, Integer, Long et Short.

La classe String est sûrement la classe immuable la plus connue et la plus utilisée.

Exemple :

```
package fr.jmdoudoux.dej;

public class TestString {
  public static void main(String[] args) {
    String chaine = new String("Bonjour");
    System.out.println(chaine);
    chaine.replaceAll("jour", "soir");
    System.out.println(chaine);
  }
}
```

Résultat :

```
Bonjour
Bonjour
```

Il est cependant parfois nécessaire d'avoir une classe immuable et la même classe modifiable : l'exemple le plus connu est la classe String et les classes StringBuffer et StringBuilder.

Il est généralement recommandé d'utiliser des objets immuables le plus souvent possible.

Il est très important que les objets renvoyés par les getters ne puissent pas être modifiés sinon la classe n'est plus immuable.

Exemple :

```
package fr.jmdoudoux.dej;

import java.util.Date;

public final class Personne {
  private final String nom;
  private final String prenom;
  private final Date dateNaiss;

  public Personne(String nom, String prenom, Date dateNaiss) {
    super();
    this.nom = nom;
    this.prenom = prenom;
    this.dateNaiss = dateNaiss;
  }

  public String getNom() {
    return nom;
  }

  public String getPrenom() {
    return prenom;
  }

  public Date getDateNaiss() {
    return dateNaiss;
  }

  @Override
  public String toString() {
    StringBuilder result = new StringBuilder("nom=");
    result.append(nom);
    result.append(", prenom=");
    result.append(prenom);
    result.append(", dateNaiss=");
    result.append(dateNaiss);
    return result.toString();
  }
}
```

Le code de cette classe peut laisser à penser que les objets de cette classe seront immuables : ce n'est pas le cas.

Exemple :

```
package fr.jmdoudoux.dej;

import java.util.Date;

public class TestImmuable {

  public static void main(String[] args) {
    Date dateNaiss = new Date();
    Personne personne = new Personne("nom1", "prenom1", dateNaiss);
    System.out.println(personne);
    Date nouvelleDateNaiss = personne.getDateNaiss();
    nouvelleDateNaiss.setMonth(nouvelleDateNaiss.getMonth() + 1);
    System.out.println(personne);
  }
}
```

Résultat :

```
nom=nom1,prenom=prenom1, dateNaiss=Sat Dec 03 20:58:49 CET 2011
nom=nom1,prenom=prenom1, dateNaiss=Tue Jan 03 20:58:49 CET 2012
```

La classe n'est pas immuable puisqu'il a été possible de modifier une de ses propriétés et comme cet objet n'est pas immuable, ses propriétés sont modifiables.

Il est donc nécessaire que les getters renvoient une instance immuable ou une autre instance de la classe qui encapsule les mêmes propriétés.

Exemple :

```
package fr.jmdoudoux.dej;

import java.util.Date;

public final class Personne {
  private final String nom;
  private final String prenom;
  private final Date dateNaiss;

  // ...
  public Date getDateNaiss() {
    return new Date(dateNaiss.getTime());
  }

  // ...
}
```

Si l'on exécute de nouveau la classe de test, l'objet reste inchangé.

Résultat :

```
nom=nom1, prenom=prenom1, dateNaiss=Sat Dec 03 20:58:49 CET 2011
nom=nom1, prenom=prenom1, dateNaiss=Sat Dec 03 20:58:49 CET 2011
```

Malgré cette modification, l'objet n'est toujours pas immuable.

Exemple :

```
package fr.jmdoudoux.dej;

import java.util.Date;

public class TestImmuable {

  public static void main(String[] args) {
    Date dateNaiss = new Date();
    Personne personne = new Personne("nom", "prenom", dateNaiss);
    System.out.println(personne);
    dateNaiss.setMonth(dateNaiss.getMonth() + 1);
    System.out.println(personne);
  }
}
```

Il suffit qu'une référence sur l'objet passée en paramètre lors de la création de l'objet soit conservée pour que l'objet ne soit toujours pas immuable.

Résultat :

```
nom=nom, prenom=prenom, dateNaiss=Sat Dec 03 21:09:26 CET 2011
nom=nom, prenom=prenom, dateNaiss=Tue Jan 03 21:09:26 CET 2012
```

Si un objet fourni en paramètre du constructeur n'est pas immuable, alors il est nécessaire d'en conserver une copie profonde (deep copy), un clone ou une version immuable.

Exemple :

```
package fr.jmdoudoux.dej;

import java.util.Date;

public final class Personne {
  private final String nom;
  private final String prenom;
  private final Date dateNaiss;

  public Personne(String nom, String prenom, Date dateNaiss) {
    super();
    this.nom = nom;
    this.prenom = prenom;
    this.dateNaiss = new Date(dateNaiss.getTime());
  }

  // ...

  public Date getDateNaiss() {
    return new Date(dateNaiss.getTime());
  }

  // ...
}
```

Si l'on exécute de nouveau la classe de test, l'objet reste inchangé et il est bien immuable..

Résultat :

```
nom=nom1, prenom=prenom1, dateNaiss=Sat Dec 03 20:58:49 CET 2011
nom=nom1, prenom=prenom1, dateNaiss=Sat Dec 03 20:58:49 CET 2011
```

Dans tous les cas en Java, il est possible de passer outre les mécanismes de protection standard utilisés pour garantir l'immutabilité des objets d'une classe en utilisant l'introspection.

Exemple :

```
package fr.jmdoudoux.dej;

import java.lang.reflect.Field;

public class MaClasse {
  public static void modifierChaine(String chaine, String valeur) {
    try {
      Field stringValue = String.class.getDeclaredField("value");
      stringValue.setAccessible(true);
      stringValue.set(chaine, valeur.toCharArray());
    } catch (Exception ex) {
    }
  }
}
```

L'utilisation de l'introspection permet de modifier n'importe quelle valeur d'une propriété private, comme pour une chaîne de caractères dans l'exemple ci-dessus.

Exemple :

```
package fr.jmdoudoux.dej;

import java.util.Date;

public class TestImmuable {
  public static void main(String[] args) {
    Personne personne = new Personne("nom1", "prenom1", new Date());
    System.out.println(personne);
    MaClasse.modifierChaine(personne.getNom(), "nom2");
    System.out.println(personne);
  }
}
```

Résultat :

```
nom=nom1, prenom=prenom1, dateNaiss=Sat Dec 03 22:32:27 CET 2011
nom=nom2, prenom=prenom1, dateNaiss=Sat Dec 03 22:32:27 CET 2011
```

Pour empêcher l'utilisation de l'introspection, il faut utiliser un gestionnaire de sécurité, qui par défaut va limiter l'accès aux membres d'une classe.

Par exemple en ajoutant l'option -Djava.security.manager à la JVM, une exception de type AccessControlException va être levée lors de la tentative d'accès à un membre d'un objet par introspection.

Résultat :

```
nom=nom1, prenom=prenom1, dateNaiss=Sat Dec 03 23:14:41 CET 2011
nom=nom1, prenom=prenom1, dateNaiss=Sat Dec 03 23:14:41 CET 2011
java.security.AccessControlException:
access denied (java.lang.RuntimePermission accessDeclaredMembers)
      at java.security.AccessControlContext.checkPermission(AccessControlContext.java:323)
      at java.security.AccessController.checkPermission(AccessController.java:546)
      at java.lang.SecurityManager.checkPermission(SecurityManager.java:532)
      at java.lang.SecurityManager.checkMemberAccess(SecurityManager.java:1662)
      at java.lang.Class.checkMemberAccess(Class.java:2157)
      at java.lang.Class.getDeclaredField(Class.java:1879)
      at fr.jmdoudoux.dej.MaClasse.modifierChaine(MaClasse.java:9)
      at fr.jmdoudoux.dej.TestImmuable.main(TestImmuable.java:10)
```

107.2. La redéfinition des méthodes equals() et hashCode()

La classe Object possède deux méthodes qui sont relatives à l'identité des objets : equals() et hashCode().

La méthode equals() permet de tester l'égalité de deux objets d'un point de vue sémantique.

La méthode hashCode() permet de renvoyer la valeur de hachage de l'objet sur lequel elle est invoquée.

Les spécifications imposent une règle à respecter lors de la redéfinition de ces méthodes : si une classe redéfinit la méthode equals() alors elle doit aussi redéfinir la méthode hashCode() et inversement. Le comportement de ces deux méthodes doit être symétrique : si les méthodes hashCode() et equals() sont redéfinies alors elles doivent utiliser, de préférence, toutes les deux les même champs car deux objets qui sont égaux en utilisant la méthode equals() doivent obligatoirement avoir tous les deux la même valeur de retour lors de l'invocation de leur méthode hashCode(). L'inverse n'est pas forcément vrai.

Le hashcode ne fournit pas un identifiant unique pour un objet : de toute façon le hashcode d'un objet est de type int, ce qui limiterait le nombre d'instances possibles d'une classe.

Deux objets pouvant avoir le même hashcode, il faut alors utiliser la méthode equals() pour déterminer s'ils sont identiques.

107.2.1. Les contraintes pour redéfinir equals() et hashCode()

Les méthodes equals() et hashCode() sont étroitement liées.

La redéfinition des méthodes equals() et hashcode() doit respecter quelques contraintes qui sont précisées dans la documentation de la classe Object :

- Symétrie : pour deux références a et b, si a.equals(b) alors il faut obligatoirement que b.equals(a)
- Réflexivité : pour toute référence non null, a.equals(a) doit toujours renvoyer true
- Transitivité : si a.equals(b) et b.equals(c) alors a.equals(c)
- Consistance avec la méthode hashCode() : si deux objets sont égaux en invoquant la méthode equals() alors leur méthode hashCode() doit renvoyer la même valeur pour les deux objets
- Pour toute référence non null, a.equals(null) doit toujours renvoyer false

Aucune spécification n'est imposée concernant l'implémentation des méthodes equals() et hashCode() pour leur permettre d'être consistantes.

Cependant, l'implémentation de la méthode hashcode() doit être consistante avec la méthode equals() : si la méthode equals() renvoie true pour deux objets alors la méthode hashCode() invoquée sur les deux objets doit renvoyer la même valeur. L'inverse n'est pas vrai, deux objets dont la méthode hashCode() renvoie la même valeur, n'implique pas obligatoirement que l'invocation de la méthode equals() sur les deux objets renvoie true.

Il est donc nécessaire de redéfinir les méthodes hashCode() et equals() de manière coordonnée si l'une ou l'autre est redéfinie. Pour garantir le contrat entre les méthodes equals() et hashCode() et leur efficacité maximale, il est préférable que leur implémentation utilise les mêmes champs de la classe.

Pour les classes qui implémentent l'interface Comparable, il est aussi important de maintenir une cohérence entre les méthodes equals() / hashCode() et la méthode compareTo(). Les spécifications précisent que si la méthode compareTo() renvoie 0 alors la méthode equals() doit renvoyer true et inversement. Cela implique aussi que si equals() renvoie false, alors la méthode compareTo() doit renvoyer une valeur différente de 0.

107.2.2. La méthode equals()

L'opérateur == vérifie si deux objets sont identiques : il compare que les deux objets possèdent la même référence mémoire et sont donc en fait le même objet.

Deux objets identiques sont égaux mais deux objets égaux ne sont pas forcement identiques.

La méthode equals() vérifie l'égalité de deux objets : son rôle est de vérifier si deux instances sont sémantiquement équivalentes même si ce sont deux instances distinctes.

Chaque classe peut avoir sa propre implémentation de l'égalité mais généralement deux objets sont égaux si tout ou partie de leurs états sont égaux.

Exemple :

```
public class TestEquals {
  public static void main(String[] args) {
    String chaine1 = new String("test");
    String chaine2 = new String("test");
    boolean isSame = (chaine1 == chaine2);
    System.out.println(isSame);
    boolean isEqual = (chaine1.equals(chaine2));
    System.out.println(isEqual);
  }
}
```

Résultat :

```
false
true
```

Deux mêmes objets sont égaux s'ils possèdent la même référence évidement mais deux objets distincts peuvent aussi être égaux si l'invocation de la méthode equals() du premier avec le second en paramètre renvoie true.

107.2.2.1. L'implémentation par défaut de la méthode equals()

L'implémentation par défaut de la méthode equals() dans la classe Object est la suivante :

Exemple :

```
public boolean equals(Object obj) {
  return (this == obj);
}
```

Par défaut, l'implémentation de la méthode equals() héritée de la classe Object teste donc l'égalité de l'adresse mémoire des objets.

Il y a un contrat à respecter entre les méthodes equals() et hashCode() : comme précisé dans la javadoc, si l'invocation de la méthode equals() avec deux instances renvoie true alors l'invocation de la méthode hashCode() de ces deux instances doit renvoyer la même valeur. Cette implémentation respecte ce contrat.

Cette implémentation par défaut de la méthode equals(), héritée de la classe Object, a le mérite de fonctionner pour tous les objets mais son mode de fonctionnement n'est pas toujours souhaité pour tous les objets.

Exemple :

```
import java.util.Date;

public class Personne {

  private String nom;
  private String prenom;
  private long id;
  private Date dateNaiss;
  private boolean adulte;

  public Personne(String nom, String prenom, long id, Date dateNaiss,
    boolean adulte) {
    super();
    this.nom = nom;
    this.prenom = prenom;
    this.id = id;
    this.dateNaiss = dateNaiss;
    this.adulte = adulte;
  }
}
```

Si l'on crée deux instances de cette classe avec les même paramètres et que l'on teste l'égalité sur les deux instances, le résultat est false puisque ce sont deux instances distinctes.

Exemple :

```
public class TestEqualsPersonne {

  public static void main(String[] args) {
    Personne p1 = new Personne("nom1", "prenom1", 1, null, true);
    Personne p2 = new Personne("nom1", "prenom1", 1, null, true);
    System.out.println(p1.equals(p2));
  }
}
```

Résultat :

```
false
```

Logiquement, on pourrait espérer que ce test renvoie true mais pour cela il faut redéfinir la méthode equals().

107.2.2.2. La redéfinition de la méthode equals()

La redéfinition de la méthode equals est un besoin fréquent mais il n'est pas toujours facile d'écrire une implémentation correcte qui tienne compte de la sémantique de la classe.

La méthode equals() permet de vérifier si l'objet qui lui est fourni en paramètre est égal à l'objet sur lequel la méthode est invoquée. Sa signature est la suivante :
public boolean equals(Object obj)

L'implémentation par défaut de cette méthode héritée de la classe Object vérifie simplement si les références des deux objets sont les mêmes (this == obj). Comme la classe Object ne possède pas de champs, c'est le seul test qu'elle peut réaliser pour tester l'égalité.

La redéfinition de la méthode equals() permet de fournir des règles particulières pour le test d'égalité des objets d'une classe. Dans ce cas, son implémentation utilise généralement un test d'égalité reposant sur tout ou partie des champs de la classe qui sont pertinents pour sa discrimination.

L'implémentation de la méthode equals() est à la charge du développeur qui doit définir ce qu'est l'égalité entre deux objets de cette classe. La classe Object propose une implémentation par défaut qui test simplement l'égalité sur les références des deux objets. Comme toutes les classes héritent de la classe Object, si leur méthode equals() n'est pas redéfinie, alors deux objet sont égaux si et seulement si ces objets ont les mêmes références.

Il est donc généralement nécessaire de redéfinir la méthode equals() pour lui donner un rôle sémantique par rapport aux champs de la classe. Par exemple :

- deux objets de type String sont égaux si les deux chaînes possèdent la même séquence de caractères
- deux objets de type Integer sont égaux si leur valeur est égale

Il est préférable d'utiliser les champs qui concernent l'état de l'objet : ceci implique généralement de ne pas prendre en compte les champs static et les champs transient.

L'implémentation de la méthode equals() n'est pas toujours facile et dépend de la classe. Si la classe est immuable alors l'implémentation de la méthode equals() peut utiliser la comparaison de l'état de l'objet avec l'état de l'objet fourni en paramètre.

L'implémentation de la méthode equals() pour une classe qui n'est pas immuable est plus difficile car il faut décider si l'égalité va se faire sur tout ou partie de l'état de l'objet ou sur l'identité de l'objet (l'implémentation de la classe Object utilise la référence par exemple). Ce choix dépend de l'utilisation qui sera faite des instances de la classe.

L'implémentation de la méthode equals() peut parfois être complexe selon les besoins. Par exemple, la méthode equals()

de l'interface List vérifie que l'autre objet est aussi de type List, que les deux collections possèdent le même nombre d'éléments, qu'ils contiennent les mêmes éléments en utilisant leur méthode equals() et que ces éléments sont dans le même ordre.

107.2.2.3. Les contraintes et quelques recommandations

L'implémentation d'une redéfinition de la méthode equals() doit respecter plusieurs caractéristiques :

- être réflexive : pour tout objet x, x.equals(x) doit retourner true. Un objet doit être égal à lui-même
- être symétrique : pour tout objet x et y, si x.equals(y) renvoie true alors y.equals(x) doit renvoyer true. Si un objet est égal à un autre alors l'autre doit être égal à l'objet
- être transitive : pour tout objet x,y et z, si x.equals(y) renvoie true et y.equals(z) renvoie true alors x.equals(z) doit renvoyer true. Si un premier objet est égal à un second et que le second est égal à un troisième alors le premier doit être égal au troisième
- être cohérent : pour tout objet x et y égaux, plusieurs invocations de la méthode x.equals(y), sans modification de x ou y, renvoient de façon consistante la même valeur
- ne jamais être égal à null : pour tout objet x non null, x.equals(null) doit toujours renvoyer false
- pour respecter les spécifications, l'implémentation de la méthode equals() doit être en relation avec celle de la méthode hashCode() pour garantir que deux objets égaux renvoient le même hash code. Cependant, l'inverse n'est pas obligatoirement vrai

Le non respect des règles qui définissent le contrat de la méthode equals() peut induire des bugs difficiles à identifier car ce sont des problèmes de conception.

Il n'existe pas de solution unique pour redéfinir la méthode equals() tant que les contraintes imposées par la spécification sont respectées. La plupart des IDE propose même une fonctionnalité pour générer cette méthode à partir de tout ou partie des champs de la classe.

Exemple :

```
import java.util.Date;

public class Personne {

  private String nom;
  private String prenom;
  private long id;
  private Date dateNaiss;
  private boolean adulte;

  public Personne(String nom, String prenom, long id, Date dateNaiss,
    boolean adulte) {
    super();
    this.nom = nom;
    this.prenom = prenom;
    this.id = id;
    this.dateNaiss = dateNaiss;
    this.adulte = adulte;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
     return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Personne other = (Personne) obj;
    if (adulte != other.adulte)
      return false;
    if (dateNaiss == null) {
      if (other.dateNaiss != null)
        return false;
    } else if (!dateNaiss.equals(other.dateNaiss))
      return false;
    if (id != other.id)
      return false;
    if (nom == null) {
      if (other.nom != null)
        return false;
    } else if (!nom.equals(other.nom))
      return false;
    if (prenom == null) {
      if (other.prenom != null)
        return false;
    } else if (!prenom.equals(other.prenom))
      return false;
    return true;
  }
  // ...
}
```

Lors de la redéfinition de la méthode equals(), il faut bien faire attention à respecter la signature de la méthode qui attend en paramètre une instance de type Object sinon c'est une surcharge qui se compilera sans soucis mais qui ne sera pas invoquée pour tester l'égalité.

Pour éviter ce problème, il faut utiliser l'annotation @Override sur la redéfinition de la méthode, assurant ainsi une erreur à la compilation si la méthode n'est pas une redéfinition d'une méthode héritée.

La redéfinition de la méthode equals() n'est pas obligatoire et n'est pas toujours forcement nécessaire notamment :

- si le test de l'égalité avec d'autres objets n'est pas nécessaire
- si l'implémentation par défaut héritée de la classe Object est suffisante
- si chaque instance de la classe est unique ou s'il n'existe qu'une seule instance de la classe (singleton)
- si l'implémentation héritée de la classe mère est suffisante : il est cependant nécessaire de s'assurer que cela soit bien le cas

Si le test de l'égalité d'une classe n'a pas de sens, il est préférable de redéfinir la méthode equals() pour qu'elle lève une exception de type UnsupportedOperationException. Ceci permet d'éviter d'avoir le comportement d'une des classes mère, qui peut être celui de la classe Object.

La méthode equals() est fréquemment utilisée notamment dans la plupart des implémentations de collections pour savoir si un objet est déjà présent dans la collection ou non. Il est donc important que la redéfinition de la méthode equals() soit optimisée et efficace surtout si le nombre d'instances dans la collection est important.

Pour optimiser ces performances, il est par exemple possible de suivre quelques recommandations :

- pour assurer la contrainte symétrique, un des premiers test de la redéfinition de la méthode equals() devrait être de tester l'égalité de l'instance avec celle fournie en paramètre de la méthode car il est inutile de faire d'autres tests si c'est la même instance
- il faut rapidement tester si l'instance passée en paramètre est null pour renvoyer directement false
- comme l'instance fournie en paramètre est de type Object, il faut tester si la classe de l'instance courante est identique à la classe de l'objet fourni en paramètre de la méthode ou si celui-ci est une instance de la classe courante : si ce n'est pas le cas, il faut renvoyer directement false sinon il est possible de caster le paramètre dans le type de la classe courante pour permettre des tests sur les valeurs des champs
- il peut être intéressant de faire les comparaisons des valeurs des champs les plus rapides en premier comme par les champs de type int. Si la valeur est différente, il n'est pas nécessaire de tester les valeurs des autres champs

Par contrat, la méthode equals() attend un objet de type Object : il est donc préférable avant de tester l'égalité des membres de la classe de s'assurer de l'égalité du type de la classe avec celui de celle fournie en paramètre.

Il y a deux manières de vérifier l'égalité de la classe avant de vérifier l'égalité des membres :

- utiliser l'opérateur instanceof
- utiliser la méthode equals() sur les classes des deux objets obtenues en invoquant leur méthode getClass()

Chacune de ces solutions a son utilité selon les circonstances et l'utilisation de l'une ou l'autre dépend des besoins.

Il est généralement préférable de tester que les objets soient du même type en testant l'égalité de l'invocation de leur méthode getClass(). Ce test permet de renvoyer false si l'instance fournie en paramètre est une sous-classe de l'instance courante. Ce type de test n'est pas obligatoire mais dans ce cas, les classes qui peuvent être passées en paramètre de la méthode equals() doivent faire de même pour respecter la règle de symétrie et de réflexivité.

Cependant, pour certains cas particuliers, il peut être souhaitable de tester que les objets soient du même type en utilisant l'opérateur instanceof. Un exemple de cas particulier concerne les entités utilisées avec Hibernate : comme ce dernier peut créer des proxys, il est préférable d'utiliser l'opérateur instanceof.

Attention cependant, ce n'est généralement pas une bonne idée d'utiliser l'opérateur instanceof lorsque la méthode equals() doit être redéfinie car généralement cela peut violer la règle de symétrie que doit respecter l'implémentation de la méthode equals().

Le test sur l'égalité des classes des deux instances permet de pouvoir étendre la classe sans avoir à redéfinir la méthode equals() pour respecter la règle concernant la symétrie.

107.2.3. La méthode hashCode()

La méthode hashCode() retourne valeur de hachage calculée sur l'instance d'un objet.

La valeur du hash code est essentiellement utilisée par les collections de type Hashxxx (java.util.Hashtable, java.util.HashMap, java.util.HashSet et leurs sous-classes, ...) qui utilisent la valeur de hachage pour améliorer leur performance.

La valeur de hachage peut également être utilisée dans un autre contexte que celui des collections : par exemple, pour améliorer les performances en Java SE 7, le compilateur transforme les instructions switch utilisant des chaînes de caractères en une série d'instructions if qui testent d'abord la valeur de hash.

La définition de la méthode hashCode() dans la classe Object possède la signature suivante :

Exemple :

```
public native int hashCode();
```

Cette méthode est déclarée native car c'est l'implémentation de la JVM qui peut obtenir l'adresse mémoire de l'objet. Par défaut, la méthode hashCode(), définie dans la classe Object, utilise l'adresse mémoire de l'instance pour créer la valeur de type int du hashcode de l'instance.

Il est cependant possible de redéfinir cette méthode puisque toutes les classes héritent de la classe Object.

107.2.3.1. L'implémentation par défaut

La classe Object propose une implémentation par défaut de la méthode hashCode() qui renvoie la référence de l'objet sous la forme d'une valeur de type int. Il est possible sur certaines plates-formes que la valeur de la référence soit supérieure à la capacité d'un entier de type int : c'est pour cette raison que deux objets distincts peuvent avoir le même hashcode.

Si la méthode hashCode() est redéfinie, il est possible d'obtenir la valeur du hashcode par défaut telle qu'elle serait renvoyée par l'implémentation de la méthode hashCode() fournie par la classe Object en utilisant la méthode System.identityHashCode().

Exemple :

```
public class TestHashcode {

  public static void main(String[] args) {
    String chaine = "ma chaine";
    System.out.println("chaine.hashCode() = " + chaine.hashCode());
    int identityHashcode = System.identityHashCode(chaine);
    System.out.println("chaine identityHashcode = " + identityHashcode);
    Object monObjet = new Object();
    System.out.println("monObjet.hashCode() = " + monObjet.hashCode());
    identityHashcode = System.identityHashCode(monObjet);
    System.out.println("monObjet identityHashcode = " + identityHashcode);
  }
}
```

Résultat :

```
chaine.hashCode() = -921457200
chaine identityHashcode = 4072869
monObjet.hashCode() = 1671711
monObjet identityHashcode = 1671711
```

107.2.3.2. La redéfinition de la méthode hashCode()

Comme précisé pour la méthode equals() de la classe Object dans la Javadoc, il est nécessaire de redéfinir la méthode hashCode() si la méthode equals() est redéfinie car il faut respecter le contrat qui précise que deux objets égaux doivent avoir le même hashcode.

Généralement, la redéfinition de la méthode equals() utilise tout ou partie des attributs de la classe pour tester l'égalité de deux objets. Il est généralement pratique d'utiliser les mêmes attributs dans le calcul du hashcode afin de garantir que deux objets égaux ont le même hashcode.

La problématique est que la valeur de retour de la méthode hashCode() est de type int : il est donc nécessaire d'appliquer un algorithme qui va déterminer une valeur de type int à partir des champs à utiliser. Il est nécessaire que cet algorithme assure que la valeur de hachage calculée soit toujours la même avec les mêmes attributs. Généralement, cet algorithme calcule une valeur de type int pour chaque attributs et combine ces valeurs en utilisant un multiplicateur (généralement un nombre premier) pour déterminer la valeur de hachage.

Il n'existe pas de solution unique pour redéfinir la méthode hashCode() tant que les contraintes imposées par la spécification sont respectées.

Exemple :

```
import java.util.Date;

public class Personne {

  private String nom;
  private String prenom;
  private long id;
  private Date dateNaiss;
  private boolean adulte;

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + (adulte ? 1231 : 1237);
    result = prime * result + ((dateNaiss == null) ? 0 : dateNaiss.hashCode());
    result = prime * result + (int) (id ^ (id >>> 32));
    result = prime * result + ((nom == null) ? 0 : nom.hashCode());
    result = prime * result + ((prenom == null) ? 0 : prenom.hashCode());
    return result;
  }
}
```

107.2.3.3. Les contraintes et les recommandations

La redéfinition de la méthode hashCode() doit explicitement respecter plusieurs règles :

- la valeur renvoyée doit être constante lors de plusieurs invocations sur un même objet durant la durée de vie de l'application. Cette valeur n'a pas d'obligation d'être constante sur plusieurs exécutions de l'application
- deux objets égaux (l'invocation de la méthode equals() sur une instance avec l'autre en paramètre renvoie true) doivent obligatoirement avoir le même hash code
- si deux objets ne sont pas égaux en invoquant la méthode equals(), alors l'invocation de la méthode hashCode() de chacun des objets n'a pas l'obligation de renvoyer des valeurs entières différentes
- plus la dispersion des valeurs de hachage calculées est importante, meilleures seront les performances lorsque l'objet sera utilisé comme dans une collection de type HashXXX

L'implémentation par défaut de la méthode hashCode(), héritée de la classe Object et qui utilise la référence de l'objet, respecte ces règles :

- la référence de l'objet de change pas durant la même exécution de l'application
- par défaut, la méthode equals() vérifie l'égalité des références des deux objets : donc deux objets égaux renverront la même valeur de hachage

Deux objets égaux doivent avoir la même valeur de hachage tant qu'ils restent égaux mais deux objets non égaux n'ont pas l'obligation d'avoir des valeurs de hachage distinctes. Pour respecter ces deux règles, il est nécessaire de redéfinir la méthode hashCode() lorsque la méthode equals() est redéfinie.

De plus, par défaut, la méthode hashCode() renvoie la valeur de type int déterminée à partir de l'adresse mémoire de l'instance. Cela permet d'avoir une bonne répartition des valeurs retournées par la méthode hashCode() mais ne permet pas de retourner la même valeur pour deux instances dont la méthode equals() est redéfinie pour tester l'égalité des valeurs de leur propriété. Il faut donc redéfinir la méthode hashCode() en conséquence si la méthode equals() est redéfinie et assurer ainsi une cohérence entre leurs implémentations.

La façon la plus simple de garantir que deux objets égaux possèdent la même valeur de hachage est d'utiliser les mêmes attributs de la classe dans l'implémentation des méthodes equals() et hashCode().

La redéfinition de la méthode hashCode() doit éviter au maximum de renvoyer la même valeur pour deux instances même si cela est quasi impossible puisque les valeurs possibles sont celles du type int du hashcode et qu'elles doivent être calculées le plus rapidement possible.

Le simple fait que l'implémentation de la méthode hashCode() renvoie une valeur fixe pour toutes les instances est une implémentation qui respecte les règles : deux objets égaux auront forcément le même hashcode et la valeur du hashcode d'un objet sera obligatoirement consistante lors de plusieurs invocations de la méthode hashCode(). Cependant, cette implémentation implique de très mauvaises performances lors de l'utilisation dans des collections de type HashXXX.

La valeur de hachage des différents objets doit être assez significative et représentative dans la plage des valeurs permises par un entier de type int. Pour atteindre cet objectif, quelques règles peuvent être utilisées :

- initialiser la valeur de retour avec un entier premier
- utiliser une formule mathématique dédiée pour chaque type primitif pour déterminer une valeur entière : deux nombres premiers pour les booléens, décalage de bits pour les types plus grands qu'un entier, ...
- faire des combinaisons en ajoutant la valeur de chaque attribut multipliée par un nombre premier

Une implémentation de la méthode hashCode() utilise donc fréquemment un ou deux nombres premiers et une expression mathématique dans l'algorithme de calcul. Généralement, l'algorithme utilise une combinaison des valeurs de hachage des différents attributs qui composent la classe.

Il n'est pas forcement nécessaire d'utiliser tous les attributs mais il faut dans ce cas sélectionner les attributs qui permettront d'être le plus discriminant. Il faut cependant garantir le respect de la règle de cohérence entre la méthode hashCode() et equals().

Les spécifications n'imposent aucun algorithme pour l'implémentation de la méthode hashCode() de la classe Object. Il n'est donc pas possible de se baser sur la valeur de hachage par défaut entre deux JVM de deux fournisseurs.

Il est très important d'optimiser le calcul de la valeur retournée par la méthode hashCode().

Ainsi si le calcul de la valeur du hashcode est complexe ou pour améliorer les performances, il est possible de mettre en cache la valeur de hachage calculée. Deux cas de figure sont à prendre en compte :

- l'objet est immuable : dans ce cas, c'est très facile car le hashcode peut être calculé une seule et unique fois lorsque l'objet est initialisé
- l'objet n'est pas immuable : il est alors nécessaire de recalculer la valeur du hashcode stocké à chaque modification de la valeur d'un attribut. Il faut cependant dans ce cas avoir la maitrise de tous les cas où la valeur d'un attribut peut être modifiée afin d'être en mesure de recalculer la nouvelle valeur de hachage

Le stockage de la valeur de hachage est donc une solution utilisable sans soucis pour un objet immuable.

Exemple :

```
import java.util.Date;

public class PersonneImmuable {
  private final String nom;
  private final String prenom;
  private final long id;
  private final Date dateNaiss;
  private final boolean adulte;
  private final int cacheHashCode;

  public PersonneImmuable(String nom, String prenom, long id, Date dateNaiss,
      boolean adulte) {
    super();
    this.nom = nom;
    this.prenom = prenom;
    this.id = id;
    this.dateNaiss = dateNaiss;
    this.adulte = adulte;
    this.cacheHashCode = calculerHashCode();
  }

  @Override
  public int hashCode() {
    return cacheHashCode;
  }

  private int calculerHashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + (adulte ? 1231 : 1237);
    result = prime * result + ((dateNaiss == null) ? 0 : dateNaiss.hashCode());
    result = prime * result + (int) (id ^ (id >>> 32));
    result = prime * result + ((nom == null) ? 0 : nom.hashCode());
    result = prime * result + ((prenom == null) ? 0 : prenom.hashCode());
    return result;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    PersonneImmuable other = (PersonneImmuable) obj;
    if (hashCode() != other.hashCode()) {
      return false;
    }
    if (adulte != other.adulte)
      return false;
    if (dateNaiss == null) {
      if (other.dateNaiss != null)
        return false;
    } else if (!dateNaiss.equals(other.dateNaiss))
      return false;
    if (id != other.id)
      return false;
    if (nom == null) {
      if (other.nom != null)
        return false;
    } else if (!nom.equals(other.nom))
      return false;
    if (prenom == null) {
      if (other.prenom != null)
        return false;
    } else if (!prenom.equals(other.prenom))
      return false;
    return true;
  }

  // ...
}
```

La valeur de hachage mise en cache peut être utilisée pour optimiser l'algorithme de la méthode equals() : si la valeur de hachage est différente alors les objets ne sont pas égaux. Attention cependant, l'inverse n'est pas vrai : si les valeurs de hachage sont égales alors les objets ne sont peut être pas égaux.

Pour une classe qui n'est pas immuable, la mise en cache de la valeur de hash est beaucoup moins triviale car la valeur doit être recalculée à chaque fois que la valeur d'un attribut qui entre dans le calcul de la valeur de hachage est modifiée.

La mise en cache de la valeur de hachage n'est peut être pas une bonne idée si le nombre d'instances est très important car cela risque d'occuper beaucoup de place dans le heap.

107.2.4. Des exemples de redéfinition des méthodes hashCode() et equals()

Cette section propose plusieurs implémentations des méthodes hashCode() et equals() utilisant des outils pour leur génération ou leur mise en oeuvre.

107.2.4.1. L'utilisation d'un IDE

Les IDE fournissent des fonctionnalités pour générer les méthodes hashCode() et equals() à partir de tout ou partie des attributs de la classe. Il ne faut cependant pas oublier de les regénérer si un attribut est ajouté ou retiré à la classe.

L'exemple ci-dessous démontre, pour une classe donnée, une implémentation possible des méthodes equals() et hashCode() générées grâce à l'IDE Eclipse.

Exemple :

```
import java.util.Date;

public class Personne {
  private final String nom;
  private final String prenom;
  private final long id;
  private final Date dateNaiss;
  private final boolean adulte;

  public Personne(String nom, String prenom, long id, Date dateNaiss,
      boolean adulte) {
    super();
    this.nom = nom;
    this.prenom = prenom;
    this.id = id;
    this.dateNaiss = dateNaiss;
    this.adulte = adulte;
  }

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + (adulte ? 1231 : 1237);
    result = prime * result + ((dateNaiss == null) ? 0 : dateNaiss.hashCode());
    result = prime * result + (int) (id ^ (id >>> 32));
    result = prime * result + ((nom == null) ? 0 : nom.hashCode());
    result = prime * result + ((prenom == null) ? 0 : prenom.hashCode());
    return result;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Personne other = obj;
    if (adulte != other.adulte)
      return false;
    if (dateNaiss == null) {
      if (other.dateNaiss != null)
        return false;
    } else if (!dateNaiss.equals(other.dateNaiss))
      return false;
    if (id != other.id)
      return false;
    if (nom == null) {
      if (other.nom != null)
        return false;
    } else if (!nom.equals(other.nom))
      return false;
    if (prenom == null) {
      if (other.prenom != null)
        return false;
    } else if (!prenom.equals(other.prenom))
      return false;
    return true;
  }
}
```

L'implémentation des méthodes equals() et hashCode() utilisent toutes les deux les mêmes attributs de la classe.

L'implémentation de la méthode equals() effectuent plusieurs tests pour vérifier l'égalité de l'instance avec celle fournie en paramètre :

- le test commence par vérifier l'égalité de la référence de l'objet avec celle de l'objet passé en paramètre : les deux objets sont égaux si ce sont les mêmes. Ce test peut améliorer les performances si les deux objets sont égaux
- le test suivant vérifie si l'objet passé en paramètre null pour renvoyer false dans ce cas
- le test suivant vérifie si les deux objets sont de même type. Si ce n'est pas le cas, la méthode renvoie false sinon est effectue un cast vers le type de l'instance. Ce test n'est pas obligatoire mais il est préférable de tester la ou les classes qui peuvent être utilisées pour le test d'égalité
- enfin l'égalité de chaque attribut est testée. Il n'est pas obligatoire de tester tous les attributs mais les attributs utilisés doivent être assez discriminants pour vérifier l'égalité des deux objets. Il est important de noter quelques optimisations faites dans ces tests : les tests sont réalisés d'abord sur les données de type primitives car ce sont les plus rapides. Les tests sur les attributs de type objet commencent par vérifier que si l'attribut est null il l'est aussi dans l'objet comparé avant de tester l'égalité elle-même.

L'implémentation de la méthode hashCode() utilise une formule mathématique reposant sur des nombres premiers et la valeur de hachage des attributs qui sont des objets. Elle utilise les mêmes attributs que ceux qui sont utilisés dans l'implémentation de la méthode equals() de la classe.

107.2.4.2. L'utilisation des helpers de Commons Lang

Pour faciliter l'implémentation des méthodes equals() et hashCode(), il est possible d'utiliser respectivement les helpers EqualsBuilder et HashCodeBuilder de la bibliothèque Apache Commons Lang.

Exemple :

```
import java.util.Date;
import org.apache.commons.lang3.builder.EqualsBuilder;
import org.apache.commons.lang3.builder.HashCodeBuilder;

public class Personne {

  private final String nom;
  private final String prenom;
  private final long id;
  private final Date dateNaiss;
  private final boolean adulte;

  public Personne(String nom, String prenom, long id, Date dateNaiss,
     boolean adulte) {
    super();
    this.nom = nom;
    this.prenom = prenom;
    this.id = id;
    this.dateNaiss = dateNaiss;
    this.adulte = adulte;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Personne other = (Personne) obj;
    return new EqualsBuilder().append(adulte, other.adulte)
        .append(dateNaiss, other.dateNaiss).append(id, other.id)
        .append(nom, other.nom).append(prenom, other.prenom).isEquals();
  }

  @Override
  public int hashCode() {
    return new HashCodeBuilder(17, 31).append(nom).append(prenom).append(id)
        .append(dateNaiss).append(adulte).toHashCode();
  }
}
```

Pour la méthode equals(), il faut créer une instance de la classe EqualsBuilder : cette classe permet de vérifier l'égalité des champs des deux objets fournis lors de l'invocation de la méthode append().

Si la classe est une classe fille, il faut aussi invoquer la méthode appendSuper() en lui passant en paramètre le résultat de l'invocation de la méthode super.equals().

Le résultat du calcul de la valeur de hachage est obtenu en invoquant la méthode isEquals().

Pour la méthode hashCode(), il faut créer une instance de la classe HashCodeBuilder en passant au constructeur deux nombres premiers choisis aléatoirement.

Il faut invoquer la méthode append() pour chaque champ qui doit entrer dans le calcul de la valeur de hachage en lui passant en paramètre la valeur du champ.

Si la classe est une classe fille, il faut aussi invoquer la méthode appendSuper() en lui passant en paramètre le résultat de l'invocation de la méthode super.hashCode().

Le résultat du calcul de la valeur de hachage est obtenu en invoquant la méthode toHashCode().

107.2.5. L'intérêt de redéfinir les méthodes hashCode() et equals()

La méthode hashCode() est essentiellement utilisée par les collections pour optimiser le classement et la recherche de leurs éléments.

Il faut s'assurer que les valeurs de hash des objets qui sont utilisées comme clés dans une Map soient suffisamment diversifiées pour ne pas avoir de problèmes de performances notamment si le nombre d'occurrences dans la collection est important.

Il est aussi très important que la valeur du hashcode ne change pas pour une instance qui est utilisée comme clé dans une collection de type Map.

Pour s'éviter des ennuis difficilement détectables, il faut absolument utiliser des objets immuables comme clés dans une collection de type Map. Le comportement des objets de type Map n'est pas spécifié si un objet utilisé comme clé est modifié en impliquant une modification de la valeur de son hash code.

107.2.5.1. L'utilisation par certaines collections

Lors de l'utilisation d'objets dans les collections, il est important de redéfinir de manière adéquate les méthodes equals() et hashCode().

Exemple :

```
public class Valeur {

  private final int valeur;

  public Valeur(int valeur) {
    super();
    this.valeur = valeur;
  }

  public int getValeur() {
    return valeur;
  }
}
```

La classe de test insère plusieurs instances d'un objet dans une collection de type HashSet. Une nouvelle instance de la classe avec un attribut identique est recherchée dans la collection et supprimée.

Exemple :

```
import java.util.HashSet;

public class TestValeur {

  public static void main(String[] args) {
    Set<Valeur> hs = new HashSet<Valeur>();

    Valeur valeur1 = new Valeur(1);
    Valeur valeur2 = new Valeur(2);
    Valeur valeur3 = new Valeur(3);

    hs.add(valeur1);
    hs.add(valeur2);
    hs.add(valeur3);

    valeur2 = new Valeur(2);
    System.out.println("hs.size()=" + hs.size());
    System.out.println("hs.contains(valeur2)=" + hs.contains(valeur2));
    System.out.println("hs.remove(valeur2)=" + hs.remove(valeur2));
    System.out.println("hs.size()=" + hs.size());
  }
}
```

Résultat :

```
hs.size()=3
hs.contains(valeur2)=false
hs.remove(valeur2)=false
hs.size()=3
```

La nouvelle instance n'est pas retrouvée dans la collection car la méthode equals() n'est pas redéfinie : c'est donc celle héritée de la classe Object qui est utilisée. Comme celle-ci compare les références et qu'elles sont différentes, l'objet n'est pas trouvé.

Le même exemple est utilisé mais maintenant les méthodes equals() et hashCode() des objets insérés dans la collection sont redéfinies.

Exemple :

```
public class Valeur {
  private final int valeur;

  public Valeur(int valeur) {
    super();
    this.valeur = valeur;
  }

  public int getValeur() {
    return valeur;
  }

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + valeur;
    return result;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Valeur other = (Valeur) obj;
    if (valeur != other.valeur)
      return false;
    return true;
  }
}
```

Résultat :

```
hs.size()=3
hs.contains(valeur2)=true
hs.remove(valeur2)=true
hs.size()=2
```

Pour retrouver une instance dans une collection, le plus simple est de parcourir tous les éléments jusqu'à ce que l'on trouve l'élément ou que tous les éléments aient été parcourus. Malheureusement, cette solution n'est pas la plus performante puisque son temps d'exécution maximum est proportionnel au nombre d'éléments dans la collection.

Pour optimiser cette recherche, les collections utilisent la valeur de la méthode hashCode() pour regrouper les éléments ayant la même valeur ou appartenant à une plage de valeurs. Le test d'égalité est ainsi fait sur l'ensemble des éléments qui font partis du même groupe.

Il est donc nécessaire que la répartition des éléments selon leurs valeurs de hachage soit diversifiée et que les groupes soient de tailles similaires : si tous les éléments ont la même valur de hachage, cela revient à parcourir tous les éléments. Si la répartition en groupe est mal organisée, par exemple par manque de dispersion dans les valeurs de hachage, l'algorithme de recherche sera meilleur mais pas encore assez performant notamment si la taille de la collection est importante.

Lors de la recherche d'un élément, sa valeur de hachage est utilisée pour déterminer le groupe d'appartenance, l'objet est recherché dans ce groupe plutôt que dans toute la collection. L'algorithme de recherche est ainsi optimisé.

Il est donc important que les éléments qui servent de clés dans une collection de type HashXXX possèdent une valeur de hachage qui soit suffisamment discriminante pour permettre une bonne répartition dans les groupes.

Il est nécessaire de garder à l'esprit que si l'implémentation de la méthode hashCode() renvoie un valeur différente selon l'état de l'objet, cela peut poser des problèmes lors de l'utilisation comme clé dans une collection de type Hashxxx. Dans ce cas, il ne faut pas que la valeur du hashcode d'un objet utilisé comme clé change car les collections de type Hashxxx présument que la valeur de hachage d'un objet utilisé comme clé ne change pas. Il est donc préférable d'utiliser, comme clé, un objet de type String ou un Wrapper qui sont des objets immuables dont la valeur de hachage ne change pas.

Par exemple, une collection de type HashTable utilise le hashcode des objets servant de clés pour déterminer dans quel groupe l'objet sera rangé. Une HashTable est créée avec un nombre arbitraire de groupes. Pour déterminer dans quel groupe insérer un objet, elle utilise le reste de la division de la valeur de hash de l'objet par le nombre de groupes. Si tous les objets ont la même valeur de hachage, tous les objets sont insérés dans le même groupe ce qui inhibe les avantages de la répartition des objets en groupes.

Lors de la recherche d'un élément à partir d'une clé, son hash code est utilisé pour déterminer dans quel groupe la recherche doit être faite. Les performances sont améliorées puisque la recherche se fait uniquement dans les éléments du groupe plutôt que sur toutes les clés de la collection.

Il est donc très important que la valeur de hachage d'un élément inséré dans une collection ne change pas sinon il y a un risque que l'objet ne soit plus retrouvé car il pourrait ne plus être dans le groupe correspondant au hashcode utilisé à son insertion. Il est cependant possible de demander un recalcul de la répartition des objets dans les groupes en utilisant la méthode rehash().

107.2.5.2. Les performances en définissant correctement la méthode hashCode()

Il est très important de redéfinir correctement la méthode hashCode() pour améliorer les performances lors de l'utilisation de ces objets notamment avec des collections de type HashXXX. Pour le vérifier, deux classes vont être utilisées avec deux implémentations différentes de la méthode hashCode(). Des instances de ces classes vont être insérées dans des collections de type HashTable et HashMap.

Exemple :

```
import java.util.Hashtable;
import java.util.concurrent.TimeUnit;

public class TestHashTablePeformance {

  public static void main(String[] args) {
    testAvecHashTable();
  }

  public static void testAvecHashTable() {
    Hashtable<Personne, String> hashTable = new Hashtable<Personne, String>();
    Personne personne = null;
    long debut = System.nanoTime();
    for (int i = 0; i < 20000; i++) {
      personne = new Personne("nom" + i, "prenom" + i, i, null, true);
      hashTable.put(personne, "nom" + i + " prenom" + i);
    }
    long fin = System.nanoTime();
    System.out.println("HashTable temps d'insertion  = "
        + TimeUnit.NANOSECONDS.toMillis(Math.abs(fin - debut)) + " ms");
    debut = System.nanoTime();
    personne = new Personne("nom12345", "prenom12345", 12345, null, true);

    if (hashTable.containsKey(personne)) {
      System.out.println(hashTable.get(personne));
    }
    fin = System.nanoTime();
    System.out.println("HashTable temps de recherche = "
        + TimeUnit.NANOSECONDS.toMillis(Math.abs(fin - debut)) + " ms");
  }
}
```

Exemple :

```
import java.util.HashMap;
import java.util.concurrent.TimeUnit;

public class TestHashMapPeformance {

  public static void main(String[] args) {
    testAvecHashMap();
  }

  public static void testAvecHashMap() {
    HashMap<Personne, String> hashMap = new HashMap<Personne, String>();
    Personne personne = null;
    long debut = System.nanoTime();
    for (int i = 0; i < 20000; i++) {
      personne = new Personne("nom" + i, "prenom" + i, i, null, true);
      hashMap.put(personne, "nom" + i + " prenom" + i);
    }
    long fin = System.nanoTime();
    System.out.println("HashMap temps d'insertion  = "
        + TimeUnit.NANOSECONDS.toMillis(Math.abs(fin - debut)) + " ms");
    debut = System.nanoTime();
    personne = new Personne("nom12345", "prenom12345", 12345, null, true);
    if (hashMap.containsKey(personne)) {
      System.out.println(hashMap.get(personne));
    }
    fin = System.nanoTime();
    System.out.println("HashMap temps de recherche = "
        + TimeUnit.NANOSECONDS.toMillis(Math.abs(fin - debut)) + " ms");
  }
}
```

Les deux classes effectuent les mêmes traitements en remplissant une collection avec 20000 occurrences d'une classe dont l'id est la clé puis recherchent une occurrence particulière dans la collection.

Dans les deux exécutions suivantes, la classe Personne utilisée implémente la méthode hashCode() en renvoyant la même valeur quelque soit l'instance.

Exemple :

```
import java.util.Date;

public class Personne {
  private final String nom;
  private final String prenom;
  private final long id;
  private final Date dateNaiss;
  private final boolean adulte;

  @Override
  public int hashCode() {
    return 123;
  }

  public Personne(String nom, String prenom, long id, Date dateNaiss,
      boolean adulte) {
    super();
    this.nom = nom;
    this.prenom = prenom;
    this.id = id;
    this.dateNaiss = dateNaiss;
    this.adulte = adulte;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Personne other = (Personne) obj;
    if (adulte != other.adulte)
      return false;
    if (dateNaiss == null) {
      if (other.dateNaiss != null)
        return false;
    } else if (!dateNaiss.equals(other.dateNaiss))
      return false;
    if (id != other.id)
      return false;
    if (nom == null) {
      if (other.nom != null)
        return false;
    } else if (!nom.equals(other.nom))
      return false;
    if (prenom == null) {
      if (other.prenom != null)
        return false;
    } else if (!prenom.equals(other.prenom))
      return false;
    return true;
  }
}
```

Les performances des exécutions sont particulièrement mauvaises.

Résultat :

```
HashTable
temps d'insertion  = 15517 ms
HashTable
temps de recherche = 4 ms
```

Résultat :

```
HashMap temps d'insertion  = 14271 ms
HashMap temps de recherche = 4 ms
```

Si toutes les instances des clés renvoient les mêmes valeurs de hachage cela fonctionne, mais tous les objets sont dans le même groupe et la recherche de l'objet concerné doit se faire en invoquant la méthode equals() sur chaque objet jusqu'à trouver le bon. Cette recherche est aussi nécessaire lors de l'insertion pour vérifier si la clé n'est pas déjà présente dans la collection. Ainsi à chaque nouvelle insertion, toutes les occurrences des clés doivent être testées ce qui dégrade fortement les performances.

Dans les deux exemples suivants, l'implémentation de la méthode hashCode() de la classe Personne permet une meilleure répartition des valeurs calculées selon les valeurs des propriétés de l'instance.

Exemple :

```
import java.util.Date;

public class Personne {
  private final String nom;
  private final String prenom;
  private final long id;
  private final Date dateNaiss;
  private final boolean adulte;

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + (adulte ? 1231 : 1237);
    result = prime * result + ((dateNaiss == null) ? 0 :
    dateNaiss.hashCode());
    result = prime * result + (int) (id ^ (id >>> 32));
    result = prime * result + ((nom == null) ? 0 : nom.hashCode());
    result = prime * result + ((prenom == null) ? 0 : prenom.hashCode());
    return result;
  }

  public Personne(String nom, String prenom, long id, Date dateNaiss,
      boolean adulte) {
    super();
    this.nom = nom;
    this.prenom = prenom;
    this.id = id;
    this.dateNaiss = dateNaiss;
    this.adulte = adulte;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Personne other = (Personne) obj;
    if (adulte != other.adulte)
      return false;
    if (dateNaiss == null) {
      if (other.dateNaiss != null)
        return false;
    } else if (!dateNaiss.equals(other.dateNaiss))
      return false;
    if (id != other.id)
      return false;
    if (nom == null) {
      if (other.nom != null)
        return false;
    } else if (!nom.equals(other.nom))
      return false;
    if (prenom == null) {
      if (other.prenom != null)
        return false;
    } else if (!prenom.equals(other.prenom))
      return false;
    return true;
  }
}
```

Les performances sont grandement améliorées avec une implémentation correcte de la méthode hashCode().

Résultat :

```
HashTable temps d'insertion  = 88 ms
HashTable temps de recherche = 0 ms
```

Résultat :

```
HashMap temps d'insertion  = 83 ms
HashMap temps de recherche = 0 ms
```

107.2.6. Des implémentations particulières des méthodes hashCode() et equals()

Cette section détaille quelques implémentations particulières des méthodes equals() et hashCode().

107.2.6.1. Les méthodes hashCode() et equals() dans le JDK

Les classes de l'API Java, notamment celles qui sont immuables, redéfinissent leurs méthodes hashCode() avec des algorithmes dédiés.

Comme les objets de type String et Integer sont immuables et que leurs méthodes equals() et hashCode() sont redéfinies, des instances de ces objets peuvent parfaitement servir de clés dans une collection de type HashTable, HashMap ou HashSet. Ceci est vrai aussi pour toutes les classes de type wrapper de valeurs primitives.

Les classes du JDK redéfinissent ou non leur méthode equals() selon leur besoin :

- la valeur de hachage des classes de type wrapper Short, Byte, Character et Integer est simplement leur valeur correspondante sous la forme d'un entier de type int
- la classe StringBuffer ne redéfinit pas la méthode equals().
- depuis la version 1.3 de Java, la classe String calcule une seule fois sa valeur de hachage et la met en cache pour la retourner simplement par la méthode hashCode(). Ceci est possible car la classe String est immuable.

Attention à bien consulter la Javadoc pour être sûr de l'implémentation de la méthode equals() et s'éviter ainsi des problèmes. Par exemple, l'implémentation de la méthode equals() pour la classe BigDecimal teste l'égalité de la valeur encapsulée mais aussi le nombre de décimales. Le test ne se fait pas uniquement sur la valeur.

107.2.6.2. Les méthodes equals() et hashCode() dans une classe fille

Il faut généralement redéfinir la méthode equals() et donc la méthode hashCode() dans une classe fille si celle-ci contient des attributs supplémentaires.

Par exemple, si la classe Joueur hérite de la classe Personne et définit deux attributs supplémentaires nommés classement et nationalité, il faut redéfinir les méthodes equals() et hashCode() pour tenir compte de ces deux champs.

Exemple :

```
import java.util.Date;

public class Personne {

  private final String nom;
  private final String prenom;
  private final long id;
  private final Date dateNaiss;
  private final boolean adulte;

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + (adulte ? 1231 : 1237);
    result = prime * result + ((dateNaiss == null) ? 0 :
    dateNaiss.hashCode());
    result = prime * result + (int) (id ^ (id >>> 32));
    result = prime * result + ((nom == null) ? 0 : nom.hashCode());
    result = prime * result + ((prenom == null) ? 0 : prenom.hashCode());
    return result;
  }

  public Personne(String nom, String prenom, long id, Date dateNaiss,
    boolean adulte) {
    super();
    this.nom = nom;
    this.prenom = prenom;
    this.id = id;
    this.dateNaiss = dateNaiss;
    this.adulte = adulte;
  }
  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Personne other = (Personne) obj;
    if (adulte != other.adulte)
      return false;
    if (dateNaiss == null) {
      if (other.dateNaiss != null)
        return false;
    } else if (!dateNaiss.equals(other.dateNaiss))
      return false;
    if (id != other.id)
      return false;
    if (nom == null) {
      if (other.nom != null)
         return false;
    } else if (!nom.equals(other.nom))
      return false;
    if (prenom == null) {
      if (other.prenom != null)
         return false;
    } else if (!prenom.equals(other.prenom))
      return false;
    return true;
  }
}
```

Exemple :

```
import java.util.Date;

public class Joueur extends Personne {
  private final String nationnalite;
  private final int classement;

  public Joueur(String nom, String prenom, long id, Date dateNaiss,
    boolean adulte, String nationnalite, int classement) {
    super(nom, prenom, id, dateNaiss, adulte);
    this.nationnalite = nationnalite;
    this.classement = classement;
  }

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = super.hashCode();
    result = prime * result + classement;
    result = prime * result + ((nationnalite == null) ? 0 : nationnalite.hashCode());
    return result;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (!super.equals(obj))
      return false;
    if (getClass() != obj.getClass())
      return false;
    Joueur other = (Joueur) obj;
    if (classement != other.classement)
      return false;
    if (nationnalite == null) {
      if (other.nationnalite != null)
        return false;
    } else if (!nationnalite.equals(other.nationnalite))
        return false;
      return true;
  }
}
```

Il est aussi important de bien veiller à respecter les spécifications de la méthode equals() notamment lors de la définition d'une classe fille.

Il est aussi préférable d'invoquer la méthode equals() de la classe mère et de tenir compte de son résultat dans l'implémentation de la méthode equals() de la classe fille.

Il y a deux façons de tester le type des objets comparés dans une implémentation de la méthode equals() :

- l'opérateur instanceof qui permet une comparaison entre instances d'une classe ou de ses classes filles
- la méthode getClass() qui ne permet une comparaison que pour des objets de même type

Généralement le test en utilisant la méthode getClass() est plus robuste.

Le test sur l'égalité des classes en utilisant l'opérateur instanceof peut poser problème avec l'héritage. Les exemples ci-dessous vont définir deux classes, dont une hérite de l'autre, qui utilisent l'opérateur instanceof dans leur redéfinition de la méthode equals().

Exemple :

```
public class ClasseMere {

  protected String champsA;

  public ClasseMere(String champsA) {
    super();
    this.champsA = champsA;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
       return false;
    if (!(obj instanceof ClasseMere))
      return false;
    ClasseMere other = (ClasseMere) obj;
    if(champsA == null) {
      if (other.champsA != null)
        return false;
    }
    else if (!champsA.equals(other.champsA))
         return false;
    return true;
  }

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((champsA == null) ? 0 : champsA.hashCode());
    return result;
  }
}
```

Exemple :

```
public class ClasseFille extends ClasseMere {
  protected String champsB;

  public ClasseFille(String champsA, String champsB) {
    super(champsA);
    this.champsB = champsB;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (!(obj instanceof ClasseFille))
      return false;
    if (!super.equals(obj))
      return false;
    ClasseFille other = (ClasseFille) obj;
    if (champsB == null) {
      if (other.champsB != null)
        return false;
    }
    else if (!champsB.equals(other.champsB))
        return false;
      return true;
  }

  @Override
  public int hashCode() {

    final int prime = 31;
    int result = super.hashCode();
    result = prime * result + ((champsB == null) ? 0 : champsB.hashCode());
    return result;
  }
}
```

Exemple :

```
public class TestEqualsClasseMereFille {

  public static void main(String[] args) {

    ClasseMere classeMere = new ClasseMere("champsA");
    ClasseFille classeFille = new ClasseFille("champsA","champsB");

    System.out.println("classeMere.equals(new ClasseMere(\"champsA\"))="
      + classeMere.equals(new ClasseMere("champsA")));

    System.out.println("classeFille.equals(new ClasseFille"
      + "(\"champsA\", \"champsB\"))="
      + classeFille.equals(new ClasseFille("champsA", "champsB")));

    System.out.println("classeMere.equals(classeFille)="
      + classeMere.equals(classeFille));

    System.out.println("classeFille.equals(classeMere)="
      + classeFille.equals(classeMere));
  }
}
```

Exemple :

```
classeMere.equals(new
ClasseMere("champsA"))=true
classeFille.equals(new ClasseFille("champsA", "champsB"))=true
classeMere.equals(classeFille)=true
classeFille.equals(classeMere)=false
```

Lorsque les objets sont de même type, toutes les règles d'implémentation de la méthode equals() sont respectées.

Cependant, lorsque la classe d'un des deux objets hérite de l'autre, la règle concernant la symétrie n'est pas respectée car un objet de type ClasseFille est bien une instance de type ClasseMere mais un objet de type ClasseMere n'est pas une instance de type ClasseFille.

Dans ce cas de figure, il est préférable de comparer l'égalité des classes plutôt que d'utiliser l'opérateur instanceof.

Généralement la règle de symétrie que doit respecter la méthode equals() est violée si une classe fille utilise la méthode getClass() dans l'implémentation de sa méthode et que la méthode equals() de sa classe mère utilise l'opérateur instanceof. La règle de symétrie est aussi violée dans le cas inverse, si une classe fille utilise l'opérateur instanceof dans l'implémentation de sa méthode et que la méthode equals() de sa classe mère utilise la méthode getClass(). Il faut conserver la même stratégie utilisée dans les classes mères et filles.

Si l'implémentation de la méthode equals() utilise l'opérateur instanceof alors il est préférable que la classe soit final ou que la méthode equals() soit final. Dans ce dernier cas, la classe pourra être dérivée mais sa méthode equals() ne pourra pas être redéfinie : la classe fille pourra alors ajouter de nouveaux comportements mais ne pourra pas ajouter de propriétés qui soient discriminantes lors du test de l'égalité d'instances.

L'implémentation de la méthode equals() qui teste l'égalité des types en utilisant la méthode getClass() est plus robuste car elle permet un respect des règles que doit mettre en oeuvre la méthode equals().

Sémantiquement les deux approches sont différentes : l'utilisation d'instanceof permet des tests entre une classe et ses classes filles. Le test avec des classes filles est parfois souhaitable notamment si la classe fille ne possède aucun nouveau champs et ajoute simplement des méthodes.

Le choix de l'approche à utiliser dépend donc de la sémantique de la classe et de la façon dont elle peut être dérivée.

107.2.6.3. La redéfinition des méthodes equals() et hashCode() pour des entités

La redéfinition des méthodes hashCode() et equals() de classes de type entité utilisées avec des solutions ORM comme Hibernate est assez délicate car plusieurs points sont à prendre en compte.

Les valeurs d'une entité peuvent être modifiées lors de la persistance de son état dans la base de données notamment lorsque l'entité n'existe pas encore dans la base de données : l'identifiant de l'entité est généralement créé par la base de données en utilisant un champ auto-incrémenté.

Il est donc généralement préférable de ne pas utiliser la propriété qui est l'identifiant de l'entité dans la redéfinition de la méthode hashCode(). Son utilisation est possible dans la méthode equals(). Il est fortement recommandé de ne pas modifier la valeur du hash code d'une instance : cependant, par définition les entités sont modifiables, même le champ servant d'identifiant lorsque l'entité est sauvegardée dans la base de données pour la première fois.

Ceci rend l'implémentation de la méthode hashCode() particulièrement délicate car il faudrait utiliser des propriétés qui seraient susceptibles de ne pas être modifiées. Ce n'est pas grave si plusieurs instances possèdent le même hash code mais il est très important que l'implémentation de la méthode equals() soit la plus réaliste et la plus précise possible.

Le lazy loading utilise des proxys qui sont des sous-classes de l'entité concernée. Dans ce cas, le test de l'égalité des types des classes en utilisant la méthode getClass() sur les deux instances dans la redéfinition de la méthode equals() va toujours renvoyer false.

De plus, les proxys ont des propriétés dont la valeur peut changer : ces propriétés ont leur valeur par défaut tant que les données ne sont pas chargées de la base de données. Généralement, ce chargement ce fait lors de l'invocation du premier getter sur le proxy de l'entité.

Pour limiter les risques de problèmes, il est généralement préférable d'utiliser dans la redéfinition des méthodes hashCode() et equals() les getters de propriétés plutôt que les propriétés elles-mêmes.

Pour des entités, il est généralement préférable de ne pas utiliser, dans l'implémentation des méthodes hashCode() et equals(), les collections qui encapsulent des relations mère/fille.

107.3. Le clonage d'un objet

Le clonage d'un objet permet de créer une nouvelle instance qui soit une copie de l'état de l'objet original : il permet donc de réaliser une copie d'un objet en le dupliquant. Cette duplication crée une nouvelle instance et copie les propriétés par valeur.

La copie d'un objet peut se faire de deux manières principales :

- copie de surface (shallow copy) : toutes les valeurs des propriétés de type primitif sont copiées dans leurs propriétés correspondantes. Pour les propriétés de type objet, ce sont leurs références qui sont copiées.
- copie profonde (deep copy) : toutes les valeurs des propriétés de type primitif sont copiées dans leurs propriétés correspondantes. Pour les propriétés de type objet, c'est une copie de l'objet qui est associée à la propriété de la copie. Cette copie respecte l'encapsulation.

Il n'y a pas de règles absolues dans l'utilisation de la copie de surface ou la copie profonde : la mise en oeuvre de l'une ou de l'autre dépend essentiellement des besoins et du contexte d'utilisation.

Si l'instance à copier ne contient que des propriétés de types primitifs alors il faut utiliser une copie de surface. Si l'instance contient des références vers d'autres objets alors l'utilisation d'une copie de surface ou profonde dépend des besoins.

La copie profonde n'est pas toujours triviale à implémenter notamment si le graphe d'objets est complexe. Par exemple, s'il contient une référence circulaire ou si plusieurs références pointent sur la même instance. Il est parfois plus facile d'utiliser la sérialisation pour créer une copie profonde d'un graphe d'objets surtout si ce graphe est complexe.

Certains objets n'ont pas besoin d'être copiés : c'est le cas des objets immuables. Ainsi, il n'est pas nécessaire de cloner des objets de type String.

Il faut faire attention lors de la copie de certains objets : par exemple, il est primordial d'empêcher le clonage d'une instance dont la classe est une implémentation du motif de conception singleton. Comme, il ne devrait y avoir qu'une seule instance de cette classe, son implémentation ne devrait pas permettre sa copie. Si tel n'est pas le cas, il ne faut pas copier cette instance. Si une des classes mère redéfinie la méthode clone(), il est nécessaire de la redéfinir dans la classe du singleton pour par exemple qu'elle lève une exception de type CloneNotSupportedException.

Le but de clonage est de réaliser une copie d'un objet : en Java, il n'y a pas forcement de garantie que cette copie soit exacte. L'utilisation de la copie d'un objet doit donc être utilisée avec discernement et en tout état de cause.

Par défaut, le clonage d'un objet en Java se fait grâce à une copie de surface qui se fait champ par champ. Pour une copie profonde, il est nécessaire d'écrire du code pour réaliser l'opération.

La copie d'un objet ne doit pas être utilisée pour créer et initialiser de nouvelles instances. Par défaut, le constructeur n'est pas invoqué lors de la copie.

107.3.1. L'interface Cloneable

L'interface Cloneable est un marqueur que toute classe doit implémenter pour permettre à ses instances de pouvoir être clonées par le mécanisme standard. Elle ne définit aucune méthode : elle permet simplement de préciser que les instances peuvent être clonées.

La classe Object n'implémente pas l'interface Cloneable : l'invocation de la méthode clone() sur une instance de type Object lèvera donc toujours une exception de type CloneNotSupportedException.

107.3.2. Le clonage par copie de surface

Par défaut en Java, il n'est pas possible de créer une copie d'un objet. L'API Java propose un support de la fonctionnalité de clonage par copie de surface d'un objet en utilisant la méthode clone().

Par défaut, aucun objet n'est clonable. Pour pouvoir être clonée, la classe d'un objet doit respecter deux conditions :

- implémenter l'interface Cloneable
- redéfinir la méthode clone()

La méthode clone() est définie dans la classe Object mais elle est déclarée avec le modificateur protected.

Il est donc nécessaire de la redéfinir dans une classe en la déclarant avec le modificateur public pour permettre son accès. L'implémentation de la méthode clone() est à la discrétion du développeur. Pour une copie de surface, il est possible d'invoquer la méthode clone() de la classe Object en invoquant la méthode parent dans chaque classe. Il est aussi possible d'utiliser des traitements particuliers à la copie.

La méthode clone() de la classe Object vérifie que la classe implémente l'interface Cloneable : si ce n'est pas le cas alors elle lève une exception de type CloneNotSupportedException.

La méthode clone() de la classe Object renvoie un objet de type Object. A partir de Java 5, il est possible d'utiliser le support de la covariance pour que la méthode clone() renvoie le type de l'instance copiée.

107.3.2.1. La redéfinition de la méthode clone()

La méthode clone() de la classe Object est définie avec les modificateurs native et protected.

L'implémentation de la méthode clone() dans la classe Object effectue plusieurs opérations :

- elle vérifie que la classe de l'instance implémente l'interface Cloneable sinon elle lève une exception de type CloneNotSupportedException
- elle crée une nouvelle instance du même type
- elle effectue une copie de surface (shallow copy) des valeurs des propriétés : elle copie toutes les valeurs des membres de type primitifs et les références de membres qui sont des objets. Donc si la propriété est un objet alors l'instance clonée va contenir la même référence sur l'objet de l'instance d'origine.

Ce comportement par défaut peut être modifié en redéfinition la méthode clone() dans la classe concernée.

Pour pouvoir réaliser un clonage, il faut redéfinir la méthode clone() héritée de la classe Object. Cette méthode est déclarée protected dans la classe Object et doit être redéfinie avec la visibilité public.

Exemple :

```
public class MaClasse implements Cloneable {

  private int    monEntier;
  private String maChaine;

  public int getMonEntier() {
    return monEntier;
  }

  public void setMonEntier(int monEntier) {
    this.monEntier = monEntier;
  }

  public String getMaChaine() {
    return maChaine;
  }

  public void setMaChaine(String maChaine) {
    this.maChaine = maChaine;
  }

  @Override
  public Object clone() throws CloneNotSupportedException {
    Object resultat = null;
    try {
      resultat = super.clone();
    } catch (CloneNotSupportedException cnse) {
      cnse.printStackTrace();
    }
    return resultat;
  }
}
```

L'implémentation de la méthode clone() doit respecter plusieurs règles qui sont définies par convention :

- l'instance renvoyée doit être différente de l'instance sur laquelle la méthode clone() est invoquée
- le type de l'instance renvoyée devrait être la même que celui de l'instance sur laquelle la méthode clone() est invoquée mais ce n'est pas une obligation
- l'invocation de la méthode equals() de l'instance sur laquelle la méthode clone() est invoquée en lui passant en paramètre l'instance renvoyée par la méthode clone() devrait renvoyer true mais ce n'est pas une obligation. Pour cela, il est nécessaire de redéfinir la méthode equals() car son implémentation par défaut vérifie l'égalité des références
- l'objet retourné par la méthode clone() et l'instance sur laquelle elle est invoquée ne doivent pas contenir de référence commune sur des objets mutables : il est nécessaire que les attributs qui sont des objets mutables soient eux-mêmes clonés. Les deux instances doivent être indépendantes

Si la classe ne possède pas de propriétés qui soient des objets, alors le plus simple est d'invoquer l'implémentation de la classe Object.

Pour effectuer le clonage, il suffit d'invoquer la méthode clone() de l'objet concerné.

Exemple :

```
public class TestClone {

  public static void main(String[] args) {
    MaClasse original = new MaClasse();
    original.setMaChaine("maChaine");
    original.setMonEntier(10);
    try {
      MaClasse clone = (MaClasse) original.clone();
      System.out.println("machaine=" + clone.getMaChaine());
      System.out.println("monEntier=" + clone.getMonEntier());
    } catch (CloneNotSupportedException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
machaine=maChaine
monEntier=10
```

A partir de Java 5, il est inutile de caster le résultat de l'invocation de la méthode clone() car avec le support de la covariance, la redéfinition de ma méthode clone() peut renvoyer le type de la classe.

Exemple (Java 5) :

```
public class MaClasse implements Cloneable {

  @Override
  public MaClasse clone() throws CloneNotSupportedException {
    return (MaClasse) super.clone();
  }
}
```

Attention : lors de l'invocation de la méthode clone() de la classe Object, le constructeur de la classe n'est pas invoqué lors de la création de la nouvelle instance.

Exemple :

```
public class MaClasse implements Cloneable {

  private int    monEntier;
  private String maChaine;

  public MaClasse() {
    System.out.println("MaClasse constructeur");
  }
  // ...
}
```

Exemple :

```
public class TestClone {
  public static void main(String[] args) {
    MaClasse original = new MaClasse();
    original.setMaChaine("maChaine");
    original.setMonEntier(10);
    try {
      MaClasse clone = (MaClasse) original.clone();
      System.out.println("machaine=" + clone.getMaChaine());
      System.out.println("monEntier=" + clone.getMonEntier());
    } catch (CloneNotSupportedException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
MaClasse constructeur
machaine=maChaine
monEntier=10
```

Comme l'interface Cloneable ne définit aucune méthode, une classe qui l'implémente n'a pas d'obligation à redéfinir la méthode clone() : c'est juste une convention qui ne sera pas vérifiée par le compilateur.

Si la méthode redéfinit la méthode clone() mais n'implémente pas l'interface Cloneable alors une exception de type CloneNotSupportedException est levée.

Exemple :

```
public class MaClasse {

  @Override
  public MaClasse clone() throws CloneNotSupportedException {
    return (MaClasse) super.clone();
  }
}
```

Exemple :

```
public class TestClone {

  public static void main(String[] args) {
    MaClasse original = new MaClasse();
    try {
      MaClasse clone = original.clone();
    } catch (CloneNotSupportedException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
java.lang.CloneNotSupportedException: MaClasse
                at java.lang.Object.clone(Native Method)
                at MaClasse.clone(MaClasse.java:4)
                at TestClone.main(TestClone.java:6)
```

L'utilisation de la méthode clone() possède un gros inconvénient : elle n'est généralement pas définit dans une interface ou une classe abstraite fille. Ceci empêche l'invocation de la méthode clone() en cas d'utilisation du polymorphisme. Dans ce cas, il faut que le type de l'objet à copier contienne la méthode clone(). Ceci réduit la possibilité de mettre en oeuvre le principe d'abstraction qui suggère d'utiliser le type le plus générique possible.

Par exemple, il n'est pas possible d'invoquer la méthode clone() sur un objet de type List car l'interface List ne définit pas la méthode clone(). Cependant les classes ArrayList ou LinkedList redéfinissent la méthode clone() avec le modificateur public.

Dans ce cas, il est impératif d'utiliser le type concret de la classe au lieu d'une de ses interfaces ou classes abstraites : ceci va à l'encontre de la mise en oeuvre du principe de l'abstraction.

107.3.2.2. L'implémentation personnalisée de la méthode clone()

Il est possible de gérer la copie en écrivant son propre code.

Exemple :

```
package fr.jmdoudoux.dej.clone;

public class MaClasse implements Cloneable {
  private int    monEntier;
  private String maChaine;

  public MaClasse() {
    System.out.println("MaClasse constructeur");
  }

  public int getMonEntier() {
    return monEntier;
  }

  public void setMonEntier(int monEntier) {
    this.monEntier = monEntier;
  }

  public String getMaChaine() {
    return maChaine;
  }
  public void setMaChaine(String maChaine) {
    this.maChaine = maChaine;
  }

  @Override
  public Object clone() throws CloneNotSupportedException {
    MaClasse result = new MaClasse();
    result.setMaChaine(this.maChaine);
    result.setMonEntier(this.monEntier);
    return result;
  }
}
```

107.3.3. Le clonage par copie profonde

Pour réaliser une copie profonde (deep copy), son implémentation doit réaliser une copie des références de manière récursive.

Pour effectuer une copie profonde, il est nécessaire de parcourir tous les objets qui composent le graphe et créer une copie de chacun d'entre-eux.

Exemple :

```
package fr.jmdoudoux.dej.clone;

public class MaClasse implements Cloneable {

  private int    monEntier;
  private String maChaine;
  private Groupe groupe;
  public MaClasse() {
    System.out.println("MaClasse constructeur");
  }

  public int getMonEntier() {
    return monEntier;
  }

  public void setMonEntier(int monEntier) {
    this.monEntier = monEntier;
  }

  public String getMaChaine() {
    return maChaine;
  }

  public void setMaChaine(String maChaine) {
    this.maChaine = maChaine;
  }

  public Groupe getGroupe() {
    return groupe;
  }

  public void setGroupe(Groupe groupe) {
    this.groupe = groupe;
  }

  @Override
  public MaClasse clone() throws CloneNotSupportedException {
    MaClasse result = new MaClasse();
    result.setMaChaine(this.maChaine);
    result.setMonEntier(this.monEntier);
    Groupe original = this.getGroupe();
    Groupe groupe = new Groupe(original.getId(), original.getNom());
    result.setGroupe(groupe);
    return result;
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.clone;

public class TestDeepCloneAvecSerialisation {

  public static void main(String[] args) {
    Groupe groupe = new Groupe(1, "groupe 1");
    groupe.setId(1);
    groupe.setNom("groupe 1");
    MaClasse maClasse1 = new MaClasse();
    maClasse1.setMaChaine("maChaine1");
    maClasse1.setMonEntier(10);
    maClasse1.setGroupe(groupe);
    System.out.println(groupe);
    try {
      MaClasse copie1 = maClasse1.clone();
      System.out.println(copie1.getMaChaine());
      System.out.println(copie1.getGroupe());
      System.out.println(copie1.getGroupe().getNom());
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
MaClasse constructeur
fr.jmdoudoux.dej.clone.Groupe@42719c
MaClasse constructeur
maChaine1
```

```
fr.jmdoudoux.dej.clone.Groupe@30c221
groupe 1
```

Lors de l'implémentation d'un mécanisme de copie profonde, il est nécessaire de faire attention aux références circulaires.

Il faut aussi faire attention à certains types d'objets qui ne doivent pas être clonés car ils encapsulent des ressources.

Lors de l'utilisation d'une copie profonde, il est donc nécessaire d'avoir une bonne connaissance des objets qui composent le graphe.

107.3.4. Le clonage en utilisant la sérialisation

Lorsque le graphe d'objet à copier est important ou complexe, il n'est pas toujours facile d'écrire le code nécessaire à la réalisation de l'opération. Java propose en standard la sérialisation qui permet de stocker l'état d'un graphe d'objets et d'en recréer des instances avec le même état.

Il est donc possible d'utiliser la sérialisation comme mécanisme pour l'implémentation d'une copie profonde. Cette solution est une dès plus simple à mettre en oeuvre.

Cependant, il faut que les types des objets liés dans le graphe puissent être sérialisés. Si tel est le cas, alors la sérialisation permet de facilement réaliser une copie profonde d'un objet.

Exemple :

```
package fr.jmdoudoux.dej.clone;

import java.io.Serializable;

public class Groupe implements Serializable {

  private static final long serialVersionUID = 1L;
  private long              id;
  private String            nom;

  public Groupe(long id, String nom) {
    super();
    this.id = id;
    this.nom = nom;
  }

  public long getId() {
    return id;
  }

  public void setId(long id) {
    this.id = id;
  }

  public String getNom() {
    return nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.clone;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

public class MaClasse implements Serializable {
  private static final long serialVersionUID = 1L;
  private int               monEntier;
  private String            maChaine;
  private Groupe            groupe;

  public MaClasse() {
    System.out.println("MaClasse constructeur");
  }

  public int getMonEntier() {
    return monEntier;
  }

  public void setMonEntier(int monEntier) {
    this.monEntier = monEntier;
  }

  public String getMaChaine() {
    return maChaine;
  }

  public void setMaChaine(String maChaine) {
    this.maChaine = maChaine;
  }

  public Groupe getGroupe() {
    return groupe;
  }

  public void setGroupe(Groupe groupe) {
    this.groupe = groupe;
  }

  public MaClasse dupliquer() throws Exception {
    ObjectOutputStream oos = null;
    ObjectInputStream ois = null;
    try {
      ByteArrayOutputStream baos = new ByteArrayOutputStream();
      oos = new ObjectOutputStream(baos);
      oos.writeObject(this);
      ByteArrayInputStream bais = new ByteArrayInputStream(baos.toByteArray());
      ois = new ObjectInputStream(bais);
      return (MaClasse) ois.readObject();
    } finally {
      if (oos != null)
        oos.close();
      if (ois != null)
        ois.close();
    }
  }
}
```

Si tous les objets du graphe à dupliquer sont Serializable alors il est possible d'utiliser l'API de sérialisation pour créer une copie des objets du graphe. Pour limiter la dégradation des performances lors de la copie, il est possible de réaliser l'opération dans un flux en mémoire grâce à un objet de type ByteArrayOutputStream et un objet de type ByteArrayInputSteam.

Exemple :

```
package fr.jmdoudoux.dej.clone;

public class TestDeepCloneAvecSerialisation {

  public static void main(String[] args) {
    Groupe groupe = new Groupe(1, "groupe 1");
    groupe.setId(1);
    groupe.setNom("groupe 1");
    MaClasse maClasse1 = new MaClasse();
    maClasse1.setMaChaine("maChaine1");
    maClasse1.setMonEntier(10);
    maClasse1.setGroupe(groupe);
    System.out.println(groupe);
    try {
      MaClasse copie1 = maClasse1.dupliquer();
      System.out.println(copie1.getMaChaine());
      System.out.println(copie1.getGroupe());
      System.out.println(copie1.getGroupe().getNom());
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

L'utilisation de la sérialisation pour la copie d'un objet possède plusieurs inconvénients :

- la copie ne concerne que les éléments qui sont pris en compte par la sérialisation : par exemple, les champs transient ne sont pas copiés
- le processus de sérialisation/désérialisation est lent et coûteux car le processus requiert la transformation du graphe d'objets en données binaires puis l'opération inverse
- elle n'invoque pas le constructeur lors de la création des nouvelles instances

107.3.5. Le clonage de tableaux

Les tableaux sont des objets qui peuvent être clonés. L'implémentation de leur méthode clone() ne lève pas d'exception de type CloneNotSupportedException puisque tous les tableaux peuvent être clonés.

A partir de Java 5, il est aussi inutile de caster le résultat de l'invocation de la méthode clone() sur un tableau.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.clone;

public class TestCloneTableau {

  public static void main(String[] args) {
    MaClasse maClasse1 = new MaClasse();
    maClasse1.setMaChaine("maChaine1");
    maClasse1.setMonEntier(10);
    MaClasse maClasse2 = new MaClasse();
    maClasse2.setMaChaine("maChaine2");
    maClasse2.setMonEntier(20);
    MaClasse[] tableauOriginal = { maClasse1, maClasse2 };
    MaClasse[] tableauClone = tableauOriginal.clone();
    System.out.println("tableauOriginal=" + tableauOriginal);
    System.out.println("tableauClone=" + tableauClone);
    System.out.println("tableauOriginal[0]=" +tableauOriginal[0]);
    System.out.println("tableauClone[0]=" + tableauClone[0]);
  }
}
```

Résultat :

```
tableauOriginal=[Lfr.jmdoudoux.dej.clone.MaClasse;@187aeca
tableauClone=[Lfr.jmdoudoux.dej.clone.MaClasse;@e48e1b
tableauOriginal[0]=fr.jmdoudoux.dej.clone.MaClasse@12dacd1
tableauClone[0]=fr.jmdoudoux.dej.clone.MaClasse@12dacd1
```

Lors du clonage d'un tableau, c'est une copie de surface qui est effectuée. Un nouveau tableau est créé mais les objets contenus dans les tableaux sont les mêmes.

Il faut écrire du code pour effectuer une copie profonde du tableau.

Exemple :

```
package fr.jmdoudoux.dej.clone;

public class TestCloneTableau {

  public static void main(String[] args) {
    MaClasse maClasse1 = new MaClasse();
    maClasse1.setMaChaine("maChaine1");
    maClasse1.setMonEntier(10);
    MaClasse maClasse2 = new MaClasse();
    maClasse2.setMaChaine("maChaine2");
    maClasse2.setMonEntier(20);
    MaClasse[] tableauOriginal = { maClasse1, maClasse2 };
    MaClasse[] tableauClone = new MaClasse[tableauOriginal.length];
    for (int i = 0; i < tableauOriginal.length; i++) {
      MaClasse maClasse = new MaClasse();
      maClasse1.setMaChaine(tableauOriginal[i].getMaChaine());
      maClasse1.setMonEntier(tableauOriginal[i].getMonEntier());
      tableauClone[i] = maClasse;
    }
    System.out.println("tableauOriginal=" + tableauOriginal);
    System.out.println("tableauClone=" + tableauClone);
    System.out.println("tableauOriginal[0]=" + tableauOriginal[0]);
    System.out.println("tableauClone[0]=" + tableauClone[0]);
  }
}
```

Résultat :

```
tableauOriginal=[Lfr.jmdoudoux.dej.clone.MaClasse;@18a992f
tableauClone=[Lfr.jmdoudoux.dej.clone.MaClasse;@4f1d0d
tableauOriginal[0]=fr.jmdoudoux.dej.clone.MaClasse@1fc4bec
tableauClone[0]=fr.jmdoudoux.dej.clone.MaClasse@dc8569
```

107.3.6. La duplication d'un objet en utilisant un constructeur ou une fabrique

Il est possible d'utiliser des solutions plus classiques et moins spécifiques pour dupliquer des objets. Ces solutions nécessitent d'écrire et de maintenir plus de code mais elles permettent d'avoir un contrôle très fin sur la duplication et sont performantes à l'exécution.

107.3.6.1. La duplication d'un objet en utilisant un constructeur

Il est possible de créer une copie d'un objet en utilisant un constructeur qui attend en paramètre un objet du type de la classe elle-même. Le constructeur peut alors copier les valeurs des propriétés de l'objet passé paramètre dans les propriétés correspondantes.

Exemple :

```
public class MaClasse implements Cloneable {

  private int    monEntier;
  private String maChaine;

  public MaClasse(MaClasse original) {
    this.monEntier = original.monEntier;
    this.maChaine = original.maChaine;
  }

  public int getMonEntier() {
    return monEntier;
  }

  public void setMonEntier(int monEntier) {
    this.monEntier = monEntier;
  }
  public String getMaChaine() {
    return maChaine;
  }

  public void setMaChaine(String maChaine) {
    this.maChaine = maChaine;
  }
}
```

Exemple :

```
public class TestCloneConstructeur {

  public static void main(String[] args) {
    MaClasse original = new MaClasse();
    original.setMaChaine("maChaine");
    original.setMonEntier(10);
    MaClasse clone = new MaClasse(original);
    System.out.println("machaine=" + clone.getMaChaine());
    System.out.println("monEntier=" + clone.getMonEntier());
  }
}
```

Résultat :

```
machaine=maChaine
monEntier=10
```

Cette solution n'est pas toujours utilisable car elle nécessite la définition d'un constructeur dédié.

107.3.6.2. La duplication d'un objet en utilisant une fabrique

Il est possible de créer une copie d'un objet en utilisant une fabrique qui attend en paramètre l'instance originale. La fabrique peut alors copier les valeurs des propriétés de l'objet passé paramètre dans les propriétés correspondantes d'une nouvelle instance.

Exemple :

```
public class MaClasse implements Cloneable {

  private int    monEntier;
  private String maChaine;

  public MaClasse() {
    System.out.println("MaClasse constructeur");
  }

  public static MaClasse dupliquer(MaClasse original) {
    MaClasse resultat = new MaClasse();
    resultat.monEntier = original.monEntier;
    resultat.maChaine = original.maChaine;
    return resultat;
  }

  public int getMonEntier() {
    return monEntier;
  }

  public void setMonEntier(int monEntier) {
    this.monEntier = monEntier;
  }

  public String getMaChaine() {
    return maChaine;
  }

  public void setMaChaine(String maChaine) {
    this.maChaine = maChaine;
  }
}
```

Exemple :

```
public class TestCloneFabrique {

  public static void main(String[] args) {
    MaClasse original = new MaClasse();
    original.setMaChaine("maChaine");
    original.setMonEntier(10);
    MaClasse clone = MaClasse.dupliquer(original);
    System.out.println("machaine=" + clone.getMaChaine());
    System.out.println("monEntier=" + clone.getMonEntier());
  }
}
```

Résultat :

```
MaClasse constructeur
MaClasse constructeur
machaine=maChaine
monEntier=10
```

107.3.7. La duplication en utilisant des bibliothèques tierces

Il est possible d'utiliser des bibliothèques tierces pour faciliter ou réaliser la copie d'un objet.

107.3.7.1. La duplication en utilisant Apache Commons

La classe org.apache.commons.lang3.ObjectUtils propose la méthode clone() qui permet de cloner un objet. Elle facilite l'utilisation du clonage de l'objet passer en paramètre en invoquant sa méthode clone() par introspection.

L'objet passé en paramètre doit donc implémenter l'interface Cloneable.

Elle effectue aussi une copie si l'objet fourni en paramètre est un tableau d'objets.

Elle lève une exception de type CloneFailedException si son exécution échoue.

Elle renvoie null si null est passé en paramètre.

107.3.7.2. La duplication en utilisant Kryo

Kryo est une bibliothèque open source, diffusée sous la licence New BSD, qui permet de sérialiser un graphe d'objets. Elle permet aussi de cloner un objet sans en sérialiser les données.

Pour utiliser Kryo, il suffit d'ajouter la bibliothèque kryo.jar et ses dépendances asm, minlog et objenesis. Le plus simple est de déclarer la dépendance dans un projet Maven :

Exemple :

```
    <dependency>
        <groupId>com.esotericsoftware</groupId>
        <artifactId>kryo</artifactId>
        <version>3.0.0</version>
    </dependency>
```

La méthode copy() permet de réaliser une copie profonde (deep copy) de l'objet fourni en paramètre.

La méthode copyShallow() permet de réaliser une copie de surface (shallow copy) de l'objet fourni en paramètre.

Les références multiples à un même objet et les références circulaires sont gérées par Kryo.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.TestKryo;

import com.esotericsoftware.kryo.Kryo;

public class TestKryo {

  public static void main(String[] args) {
    Kryo kryo = new Kryo();
    Groupe groupe = new Groupe();
    groupe.setId(1);
    groupe.setNom("groupe 1");
    MaClasse maClasse1 = new MaClasse();
    maClasse1.setMaChaine("maChaine1");
    maClasse1.setMonEntier(10);
    maClasse1.setGroupe(groupe);
    System.out.println(groupe);

    MaClasse copie1 = kryo.copyShallow(maClasse1);
    System.out.println(copie1.getMaChaine());
    System.out.println(copie1.getGroupe());
    System.out.println(copie1.getGroupe().getNom());

    MaClasse copie2 = kryo.copy(maClasse1);
    System.out.println(copie2.getMaChaine());
    System.out.println(copie2.getGroupe());
    System.out.println(copie2.getGroupe().getNom());
  }
}
```

Résultat :

```
MaClasse constructeur
fr.jmdoudoux.dej.TestKryo.Groupe@fced4
MaClasse constructeur
maChaine1
fr.jmdoudoux.dej.TestKryo.Groupe@fced4
groupe 1
MaClasse constructeur
maChaine1
fr.jmdoudoux.dej.TestKryo.Groupe@272961
groupe 1
```

La classe des objets clonés n'ont pas besoin d'implémenter d'interface particulière.

107.3.7.3. La duplication en utilisant XStream

XStream est une API open source qui permet de sérialiser/désérialiser des objets.

Le site du projet est à l'url https://x-stream.github.io/

Pour utiliser XStream, il suffit d'ajouter la bibliothèque xstream.jar et sa dépendance xpp3_min.jar. Le plus simple est de déclarer la dépendance dans un projet Maven :

Exemple :

```
    <dependency>
        <groupId>xstream</groupId>
        <artifactId>xstream</artifactId>
        <version>1.2.2</version>
    </dependency>
```

Pour la mise en oeuvre, il suffit de créer une instance de type XStream et d'invoquer ses méthodes toXML() et fromXML().

Exemple :

```
package fr.jmdoudoux.dej.clone;

public class MaClasse {
  private int    monEntier;
  private String maChaine;
  private Groupe groupe;

  public MaClasse() {
    System.out.println("MaClasse constructeur");
  }

  public int getMonEntier() {
    return monEntier;
  }

  public void setMonEntier(int monEntier) {
    this.monEntier = monEntier;
  }

  public String getMaChaine() {
    return maChaine;
  }

  public void setMaChaine(String maChaine) {
    this.maChaine = maChaine;
  }

  public Groupe getGroupe() {
    return groupe;
  }

  public void setGroupe(Groupe groupe) {
    this.groupe = groupe;
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.clone;

public class Groupe {
  private long   id;
  private String nom;

  public Groupe() {
  }

  public Groupe(long id, String nom) {
    super();
    this.id = id;
    this.nom = nom;
  }

  public long getId() {
    return id;
  }

  public void setId(long id) {
    this.id = id;
  }

  public String getNom() {
    return nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.clone;

import com.thoughtworks.xstream.XStream;

public class TestXStream {

  public static void main(String[] args) {
    XStream xstream = new XStream();
    Groupe groupe = new Groupe(1, "groupe 1");
    groupe.setId(1);
    groupe.setNom("groupe 1");
    MaClasse maClasse = new MaClasse();
    maClasse.setMaChaine("maChaine1");
    maClasse.setMonEntier(10);
    maClasse.setGroupe(groupe);
    System.out.println(groupe);
    MaClasse copie = (MaClasse) xstream.fromXML(xstream.toXML(maClasse));
    System.out.println(copie.getMaChaine());
    System.out.println(copie.getGroupe());
    System.out.println(copie.getGroupe().getNom());
  }
}
```

Résultat :

```
MaClasse constructeur
fr.jmdoudoux.dej.clone.Groupe@35bb0f
MaClasse constructeur
maChaine1
fr.jmdoudoux.dej.clone.Groupe@31688f
groupe 1
```

La classe des objets clonés n'ont pas besoin d'implémenter d'interface particulière par contre elles doivent impérativement posséder un constructeur sans argument sinon une exception est levée.

Résultat :

```
MaClasse constructeur
fr.jmdoudoux.dej.clone.Groupe@18bd7f1
MaClasse constructeur
Exception in thread
"main" com.thoughtworks.xstream.converters.ConversionException:
Cannot construct fr.jmdoudoux.dej.clone.Groupe as it does not have a no-args
constructor
---- Debugging information
----
message             : Cannot construct
fr.jmdoudoux.dej.clone.Groupe as it does not have a no-args constructor
cause-exception     :
com.thoughtworks.xstream.converters.reflection.ObjectAccessException
cause-message       : Cannot construct fr.jmdoudoux.dej.clone.Groupe
as it does not have a no-args constructor
class               : fr.jmdoudoux.dej.clone.MaClasse
required-type       : fr.jmdoudoux.dej.clone.XStream.Groupe
path                : /fr.jmdoudoux.dej.clone.MaClasse/groupe
line number         : 4
-------------------------------
```

108. L'encodage des caractères

Chapitre 108

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Un caractère est une unité minimale abstraite de texte qui n'a pas forcément toujours la même représentation graphique.

La plate-forme Java utilise Unicode pour son support des caractères mais il est fréquent de devoir traiter des données textuelles encodées différemment en entrée ou en sortie d'une application. Java propose plusieurs classes et méthodes pour permettre la conversion de nombreux encodages de caractères de et vers Unicode.

Les applications Java qui doivent traiter des données non encodées en Unicode, sont lues avec l'encodage adéquat, stockées et traitées en Unicode et exportent le résultat de Unicode vers l'encodage initial ou l'encodage cible.

La version 5.0 de Java propose un support de la version 4.0 d'Unicode.

La JSR 204 « Unicode Supplementary Character Support » définit le support des caractères étendus d'Unicode dans la plate-forme Java. Ceci permet le support des caractères au-delà des 65546 possibles sur un stockage dans 2 octets.

Ce chapitre contient plusieurs sections :

- L'utilisation des caractères dans la JVM
- Les jeux d'encodages de caractères
- Unicode
- L'encodage de caractères
- L'encodage du code source
- L'encodage de caractères avec différentes technologies

108.1. L'utilisation des caractères dans la JVM

Une chaîne de caractères est stockée en interne dans la JVM en UTF-16. L'encodage des caractères est uniquement à réaliser en entrée ou en sortie de la JVM (fichiers, base de données, flux, ...).

108.1.1. Le stockage des caractères dans la JVM

En interne, Java utilise le jeu de caractères Unicode et stocke les caractères encodés en UTF-16. Cependant pour le stockage persistant ou l'échange de données, il peut être nécessaire d'utiliser différents jeux d'encodages de caractères. Java propose des mécanismes au travers d'API pour permettre ces conversions.

Le type de données primitif char qui stocke un caractère a une représentation sous la forme d'un entier de 16 bits non signé pour pouvoir contenir un caractère encodé en UTF-16. Toutes les classes qui encapsulent un ou plusieurs caractères encodent ceux-ci en UTF-16 en interne dans la JVM.

108.1.2. L'encodage des caractères par défaut

Pour modifier l'encodage utilisé par défaut pour la lecture et l'écriture dans des flux, il faut modifier la valeur de la propriété file.encoding de la JVM.

Il est possible de fournir la valeur désirée en paramètre de la JVM.

Exemple :

```
java.exe "-Dfile.encoding=UTF-8" -jar monapp.jar
```

La propriété peut aussi être modifiée par programmation en utilisant la méthode setProperty() de la classe System.

Exemple :

```
System.setProperty( "file.encoding", "UTF-8" );
```

108.2. Les jeux d'encodages de caractères

Il existe de nombreux jeux d'encodages de caractères. Une liste complète des jeux d'encodages de caractères est consultable à l'url https://www.iana.org/assignments/character-sets

Un jeu de caractères est un ensemble de caractères. Dans un jeu, chaque caractère est associé à une valeur unique.

Les jeux de caractères les plus utilisés dans les pays occidentaux sont notamment ISO-8859-1, ISO-8859-15, UTF-8, Windows CP-1252, ...

108.3. Unicode

Unicode est un ensemble de caractères pouvant contenir tous les caractères utilisés dans le monde. Unicode peut contenir jusqu'à 1 million de caractères, mais tous les caractères ne sont pas utilisés.

L'ensemble des caractères est divisé en blocs.

Unicode est géré par un consortium : la version courante d'Unicode est la 5.

Unicode attribue à chaque caractère un identifiant nommé code point. Unicode utilise la notation hexadécimale prefixée par « U+ » pour représenter un code point : exemple avec le caractère A qui possède le numéro U+0041.

Les 127 premiers caractères d'Unicode correspondent exactement à l'ensemble des caractères Ascii.

108.3.1. L'encodage des caractères Unicode

Les caractères Unicode peuvent être encodés avec plusieurs encodages de la norme UTF (Unicode Transformation Format)

UTF-32 est l'encodage le plus simple d'Unicode puisqu'il utilise 32 bits (4 octets) pour stocker chaque caractère mais c'est aussi l'encodage le plus coûteux en mémoire.

UTF-16 utilise un encodage sur 16 bits (2 octets) ou 2 fois 16 bits pour stocker les caractères Unicode. Ainsi les valeurs comprises entre U+0000 et U+FFFF sont encodées uniquement sur 16 bits. Les valeurs au-delà sont stockées sur 2 fois 16 bits.
Son principal avantage est qu'il est capable de stocker la plupart des caractères courants avec un seul entier de 16 bits.

Remarque : les fichiers encodés en UTF-16 ne sont généralement pas échangeables entre différents systèmes car deux conventions d'ordonnancement des octets sont utilisées.

UTF-8 utilise un encodage sur 1 à 4 octets pour stocker les caractères Unicode selon leurs valeurs :

- entre U+0000 et U+007F : elles sont stockées sur un seul octet
- entre U+0080 et U+07FF : elles sont stockées sur deux octets
- entre U+0800 et U+FFFF : elles sont stockées sur trois octets
- entre U+10000 to U+10FFFF : elles sont stockées sur quatre octets

Avec UTF-8 chaque caractère est encodé sur un nombre variable d'octets. L'avantage d'UTF-8 est qu'il est compatible avec l'Ascii puisque les premiers caractères sont ceux de la table Ascii et qu'ils sont codés sur un seul octet en UTF-8. Ceci rend UTF-8 assez largement utilisé.

L'encodage/décodage en UTF-8 est assez coûteux car complexe puisque les caractères sont encodés sur un nombre variable d'octets.

UTF-7 encode un caractère Unicode grâce à des séquences de caractères Ascii 7 bits. Cet encodage est utilisé par certains protocoles de messagerie.

Le tableau ci-dessous montre les valeurs des octets de différents encodages de quelques caractères Unicode.

Symbole	A	Z	0	9	€	é	@
Code point	U+0041	U+005A	U+0030	U+0039	U+20AC	U+00E9	U+0040
UTF-8	41	5A	30	39	E2 82 AC	C3 A9	40
UTF-16 Litte endian	41 00	5A 00	30 00	39 00	AC 20	E9 00	40 00
UTF-16 Big endian	00 41	00 5A	00 30	00 39	20 AC	00 E9	00 40
UTF-32 Little endian	41 00 00 00	5A 00 00 00	30 00 00 00	39 00 00 00	AC 20 00 00	E9 00 00 00	40 00 00 00
UTF-32 Big endian	00 00 00 41	00 00 00 5A	00 00 00 30	00 00 00 39	00 00 20 AC	00 00 00 E9	00 00 00 40

108.3.2. Le marqueur optionnel BOM

Le début d'un fichier encodé en UTF peut contenir un marqueur optionnel nommé BOM (Byte Order Marker). Ce marqueur a deux utilités :

- Permettre de préciser que le texte est encodé en UTF-8, UTF-16 ou UTF-32
- Pour UTF-16- et UTF-32, il permet de préciser l'ordre des octets (little-endian ou big-endian)

En UTF-8, les trois premiers octets du BOM sont EF BB BF.

Lorsque l'on édite un fichier encodé en UTF-8 contenant un BOM avec un éditeur utilisant l'encodage iso-8859-1, les premiers octets affichés sont `ï¿½»ï¿½`.

En UTF-16, les deux premiers octets du BOM peuvent avoir deux valeurs :

- FE FF : pour un big-endian
- FF FE : pour un little-endian

En UTF-32, les quatre premiers octets du BOM peuvent avoir deux valeurs :

- 00 00 FE FF : pour un big-endian
- FF FE 00 00 : pour un little-endian

108.4. L'encodage de caractères

L'environnement d'exécution Java supporte en standard plusieurs jeux d'encodages de caractères dont :

- US-ASCII : encodage des caractères sur 7 bits
- ISO-8859-1 (latin 1) : encodage des caractères sur 8 bits dans un seul octet pour la plupart des caractères des langues européennes de l'ouest excepté le caractère €
- ISO-8859-15 : comme l'ISO-8859-1 sauf 8 caractères qui sont différents dont le caractère €
- UTF-8 : encodage des caractères sur 8 bits dans 1 à 4 octets. Les caractères ASCII correspondent aux premiers caractères en UTF-8. Peut commencer par un ensemble d'octets optionnels nommés BOM (EF BB BF)
- UTF-16 : encodage des caractères sur 16 bits dans 2 ou 4 octets. Il permet l'encodage de tous les caractères utilisés dans le monde (deux encodages existent : UTF-16 BE et UTF-16 LE)
- Cp1252 : variante utilisée par Microsoft Windows du latin 1
- ...

Les implémentations de l'environnement d'exécution Java proposent généralement un ensemble beaucoup plus complet de jeux d'encodages de caractères.

Plusieurs classes qui manipulent des caractères permettent de convertir les caractères au format Unicode en utilisant le jeu d'encodages de caractères souhaité notamment :

- java.lang.String
- java.io.InputStreamReader
- java.io.OutputStreamWriter
- et les classes du package java.nio.charset

Les jeux d'encodages de caractères supportés par la plate-forme Java dépendent de leurs implémentations. Les jeux de caractères supportés en standard sont stockés dans le fichier rt.jar

Attention : la désignation des jeux de caractères dans les API java.io et java.lang est différente de la désignation de ceux de l'API java.nio. Exemple :

Description	Nom pour l'API java.io et java.lang	Nom pour l'API java.nio
MS-DOS Latin-2	Cp852	IBM852
ISO Latin 1	ISO8859_1	ISO-8859-1
ISO Latin 2	ISO8859_2	ISO-8859-2
ISO Latin 15	ISO8859_15	ISO-8859-15
ASCII	ASCII	US-ASCII
UTF 8	UTF8	UTF-8
UTF 16	UTF-16	UTF-16
UTF 32	UTF_32	UTF-32
Windows Latin 1	Cp1252	windows-1252

Les jeux de caractères supportés par la version internationale sont dans le fichier charsets.jar du sous-répertoire lib du répertoire d'installation du JRE.

Le plus important lorsque l'on manipule des données de type texte en Java est de s'assurer que les caractères seront encodés ou décodés avec le bon jeu de caractères d'encodage.

Une fois que la conversion est faite en écriture et en lecture, le support des caractères Unicode se fera de façon transparente entre l'application Java et les ressources externes.

Par exemple, pour écrire des données de type texte dans un fichier, il suffit de préciser le jeu de caractères d'encodage. Lors de la lecture de ce fichier, il suffit de préciser le même jeu de caractères d'encodage pour obtenir les données.

Chaque fichier encodé est composé d'un ensemble d'un ou plusieurs octets. Java travaille en interne en stockant les données de types caractères ou chaîne de caractères en Unicode en utilisant l'encodage UTF-16.

L'encodage se fait toujours du type String vers le type byte[].

Le décodage se fait toujours du type byte[] vers le type String.

108.4.1. Les classes du package java.lang

La classe String permet aussi des conversions d'Unicode vers un type d'encodage et vice versa.

Un constructeur de la classe String permet de créer une chaîne de caractères à partir d'un tableau d'octets et du nom de l'encodage utilisé.

Exemple :

```
byte[] someBytes = ...;
String encodingName = "Shift_JIS";
String s = new String ( someBytes, encodingName );
```

Pour obtenir un tableau d'octets qui contient le contenu d'une chaîne de caractères encodée selon un encodage particulier, il faut utiliser la méthode getBytes() de la classe String

Exemple :

```
// Using String.getBytes to encode String to bytes
String s = ...;
byte [] b = s.getBytes( "8859_1" /* encoding */ );
```

La méthode getBytes() de la classe String utilise par défaut l'encodage du système d'exploitation sur lequel la JVM est exécutée.

Une surcharge de la méthode getBytes() permet de préciser l'encodage à utiliser.

108.4.2. Les classes du package java.io

Les classes qui héritent des classes Reader et Writer proposent des fonctionnalités pour permettre des opérations de lecture et d'écriture de caractères dans un flux.

La classe InputStreamReader permet de lire des données encodées avec de nombreux types d'encodage pour obtenir des données stockées dans la JVM en Unicode.

Exemple :

```
// Pour lire un fichier en UTF8 :
new InputStreamReader(new FileInputStream("monfichier;txt"), "utf8")
```

La classe OutputStreamReader permet d'écrire des données en Unicode encodées vers de nombreux types d'encodage.

Les classes qui héritent de la classe Reader décodent des octets en String en fonction de l'encodage précisé.

Les classes qui héritent de la classe Writer encodent des String en octets en fonction de l'encodage précisé.

Les classes FileReader et FileWriter permettent de lire et d'écrire des caractères dans un flux.

Exemple :

```
// FileWriter
Writer w = new BufferedWriter(new OutputStreamWriter(new FileOutputStream(file), "UTF-8"));
// FileReader
Reader r = new BufferedReader(new InputStreamReader(new FileInputStream(file), "UTF-8"));
```

108.4.3. Le package java.nio

Le package java.nio.charset propose plusieurs classes pour réaliser des conversions de caractères.

La méthode canEncode() de la classe CharsetEncoder permet de vérifier si une chaîne de caractères peut être encodée avec un jeu de caractères d'encodage.

Exemple : vérifier si une chaîne de caractères peut être encodée en latin-1

```
String str = "abcdef"
CharsetEncoder encoder = Charset.forName("iso-8859-1").newEncoder();
boolean ok = encoder.canEncode(str);
str = "1000€";
encoder = Charset.forName("iso-8859-1").newEncoder();
ok = encoder.canEncode(str);
```

La méthode availableCharsets() de la classe java.nio.Charset permet de connaître la liste des encodages supportés.

La classe java.nio.Charset permet aussi de faire des conversions. Son grand avantage est de ne pas avoir à rechercher la classe correspondant à l'encodage utilisée à chaque appel comme c'est le cas avec les méthodes de la classe String.

Exemple :

```
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.charset.Charset;

byte[] b = ...;
Charset def = Charset.defaultCharset() ;
Charset cs = Charset.forName("Shift_JIS");
ByteBuffer bb = ByteBuffer.wrap( b );
CharBuffer cb = cs.decode( bb );
String s = cb.toString();
```

Le package java.nio.charset.spi propose des classes pour définir ses propres jeux d'encodage de caractères.

108.5. L'encodage du code source

La plupart des fichiers sources sont encodés en ASCII, ISO-8859-1 ou autres mais dans tous les cas, ils sont transformés en UTF-16 avant la compilation.

Le code source peut être écrit directement en utilisant un format UTF, par exemple UTF-8. Il suffit alors de préciser au compilateur le jeu de caractères d'encodage utilisé.

Attention : si le code est écrit en UTF-8, il faut s'assurer que l'éditeur n'inclut pas le BOM (Byte Order Mark) au début du fichier (par exemple, c'est ce que fait l'outil Notepad sous Windows), sinon le compilateur refusera de compiler le code source

Exemple :

```
public class Test {

  public static void main(String[] args) {
    System.out.println("€");
  }
}
```

Avec l'outil Notepad, il faut enregistrer le fichier au format utf-8 et compiler la classe en précisant que l'encodage est UTF-8.

Exemple :

```
C:\temp>"C:\Program
Files\Java\jdk1.6.0_07\bin\javac" -encoding utf-8 Test.java
Test.java:1: illegal
character: \65279
?public class Test {
^
1
error
```

En fait, Notepad a ajouté les octets du BOM au début du fichier

Résultat :

```
ï¿½"ï¿½public class Test {
...
```

Sans ces octets, le code source se compile parfaitement sous réserve de bien préciser la valeur utf-8 au paramètre -encoding du compilateur javac.

Il est aussi possible d'utiliser l'outil native2ascii, dont le nom est relativement inadéquat, fourni avec le jdk . Cet outil lit le code source et le convertit en ascii en échappant les caractères non ascii avec leur représentation hexadécimale sous la forme \unnnn, où nnnn représente le code Unicode du caractère. Il n'est alors plus nécessaire d'utiliser le paramètre encoding. L'avantage de cette solution est que le code source est lisible sur tous les systèmes puisqu'il est encodé en Ascii.

108.6. L'encodage de caractères avec différentes technologies

L'encodage de caractères est généralement nécessaire et cela avec plusieurs technologies utilisées en Java.

108.6.1. L'encodage de caractères dans les fichiers

Généralement, les fichiers texte ne contiennent aucunes indications sur l'encodage utilisé.

Certaines normes proposent cependant des fonctionnalités optionnelles pour fournir l'information.

Exemple : HTML

```
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
```

Exemple : XML

```
<?xml version="1.0" encoding="ISO8859-1" ?>
```

108.6.2. L'encodage de caractères dans une application web

Pour utiliser l'encodage UTF-8 dans une application web, il faut prendre plusieurs précautions.

Dans les JSP, il faut définir l'encodage utilisé

Exemple :

```
<%@ page contentType="text/html; charset=UTF-8"%>
<%@ page pageEncoding="UTF-8"%>
```

Dans une servlet, il est possible d'utiliser la méthode setCharacterEncoding() de la classe HttpRequest pour préciser l'encodage des données de la requête. Cet appel doit être fait avant l'utilisation de la méthode getParameter() pour que les données soit correctement décodées.

108.6.3. L'encodage de caractères avec JDBC

Avec JDBC, il est parfois nécessaire de préciser l'encodage utilisé dans les données échangées. Dans ce cas, l'attribut à utiliser dépend de la base de données concernée et il faut consulter la documentation du pilote JDBC utilisé.

Exemple :

```
jdbc:mysql://localhost/mabase?useUnicode=true&characterEncoding=utf8
```

109. Les frameworks

Chapitre 109

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Le développement en Java impose certaines contraintes :

- nécessité de maitriser de nombreux concepts généraux (notamment celui de la POO, les design patterns, etc ...) et spécifiques aux plates-formes Java (standard, entreprise et mobile) selon les besoins
- nécessité de maitriser de nombreuses API qui sont de bas niveau
- quasi-absence d'outils de type RAD

Ces facteurs allongent la durée d'apprentissage des développeurs et complexifient l'architecture des applications.

De ce fait, le concept de framework est apparu, pour l'essentiel, massivement soutenu par la communauté open-source qui est très prolifique dans le monde Java.

Ce chapitre contient plusieurs sections :

- ♦ La présentation des concepts
- ♦ Les frameworks pour les applications web
- ♦ L'architecture pour les applications web
- ♦ Le modèle MVC type 1
- ♦ Le modèle MVC de type 2
- ♦ Les frameworks de mapping Objet/Relationel
- ♦ Les frameworks de logging

109.1. La présentation des concepts

Le développement d'applications peut être facilité grâce à l'utilisation :

- de la POO et des design patterns lors de la conception
- de boîtes à outils : ce sont des classes qui proposent des utilitaires basiques
- de frameworks

109.1.1. La définition d'un framework

Le terme framework est fréquemment utilisé dans des contextes différents mais il peut être traduit par cadre de développement.

Les frameworks se présentent sous diverses formes, qui peuvent inclure tout ou partie des éléments suivants :

- un ensemble de classes généralement regroupées sous la forme de bibliothèques pour proposer des services plus ou moins sophistiqués
- un cadre de conception reposant sur les design patterns pour proposer tout ou partie d'un squelette d'application
- des recommandations sur la mise en oeuvre et des exemples d'utilisation

- des normes de développement
- des outils facilitant la mise en oeuvre

L'objectif d'un framework est de faciliter la mise en ouvre des fonctionnalités de son domaine d'activité. Il doit permette au développeur de se concentrer sur les tâches spécifiques de l'application à développer plutôt que sur les tâches techniques récurrentes telles que :

- l'architecture de base de l'application
- l'accès aux données
- l'internationalisation
- la journalisation des événements (logging)
- la sécurité (authentification et gestion des rôles)
- le paramétrage de l'application
- ...

La mise en oeuvre d'un framework permet notamment :

- de capitaliser le savoir-faire sans "réinventer la roue"
- d'accroître la productivité des développeurs une fois le framework pris en main
- d'homogénéiser les développements des applications en assurant la réutilisation de composants fiables
- donc de faciliter la maintenance notamment évolutive des applications

Cependant, cette mise en oeuvre peut se heurter à certaines difficultés :

- le temps de prise en main du framework par les développeurs peut être plus ou moins long en fonction de différents facteurs (complexité du framework, richesse de sa documentation, expérience des développeurs, ...)
- les évolutions du framework qu'il faut répercuter dans les applications existantes

109.1.2. L'utilité de mettre en oeuvre des frameworks

Le développement d'applications d'entreprise avec Java (J2EE) s'avère relativement complexe. Il est nécessaire d'assimiler de nombreux concepts et API pour pouvoir développer et déployer une application J2EE. Par exemple pour le développement d'applications web, les API et les spécifications de J2EE ne proposent qu'un support de bas niveau au travers des API Servlets et JSP.

Le développement d'une application de taille moyenne ou complexe va ainsi rencontrer d'énormes difficultés. Pour faciliter le développement et augmenter la productivité, des frameworks ont été développés.

L'intérêt de la mise en oeuvre d'un ou plusieurs frameworks a fait ses preuves depuis longtemps. Une des grandes difficultés est de sélectionner le ou les frameworks à utiliser.

L'intérêt majeur des frameworks est de proposer une structure identique pour toutes les applications qui l'utilisent et de fournir des mécanismes plus ou moins sophistiqués pour assurer des tâches communes à toutes les applications.

Il est possible de développer son propre framework mais cela représente un investissement long, risqué et donc coûteux qu'il sera quasi impossible de rentabiliser d'autant que de nombreux frameworks sont présents sur le marché dont quelques frameworks open source particulièrement matures. Par exemple, le framework Struts du groupe Apache Jakarta est devenu un quasi-standard adopté par la majorité des acteurs proposant un IDE et sert de base au développement d'autres frameworks open source ou commerciaux.

Il existe de nombreux frameworks open source dont la possibilité de mise en oeuvre concrète est très variable en fonction de plusieurs facteurs : fonctionnalités proposées, maturité du projet, évolutions constantes, documentation proposée, ...

Plusieurs d'entre-eux sont devenus de véritables succès grâce à leur adoption par de nombreux développeurs et à l'apport de nombreux contributeurs (exemple : Struts, Log4J ou Spring). Ils permettent d'avoir des frameworks relativement complets, fiables et fonctionnels. En plus, comme tout projet open source, les sources sont disponibles ce qui permet éventuellement de faire des modifications pour répondre à ses propres besoins ou ajouter des fonctionnalités.

La diversité de ces frameworks permet de répondre à de nombreux besoins. Le revers de la médaille est la difficulté de choisir celui ou ceux qui y répondront au mieux.

Pour choisir un framework, les caractéristiques suivantes doivent être prises en compte :

- adoption par la communauté
- la qualité de la documentation
- le support (commercial ou communautaire)
- le support par les outils de développement

Le plus gros défaut des frameworks est lié à leur complexité : il faut un certain temps d'apprentissage pour avoir un minimum de maitrise et d'efficacité dans leur utilisation.

Le choix d'un framework est très important car l'utilisation d'un autre framework en remplacement impose souvent un travail important essentiellement lié à la prise en main du nouveau framework et aux adaptations ou à la réécriture partielle de morceaux de l'application.

109.1.3. Les différentes catégories de framework

Généralement, le coeur d'une application repose sur une architecture proposée par un framework mais il est aussi nécessaire de prévoir d'autres frameworks pour réaliser certaines tâches généralement techniques :

- logging
- mapping O/R
- automatisation des tests
- ...

Ainsi, les frameworks peuvent être regroupés en plusieurs catégories :

- Technique : propose des services techniques récurrents
- Structurel : propose la mise en place d'une architecture applicative
- Métier : propose des services fonctionnels
- Tests : propose des services pour automatiser les tests unitaires

109.1.4. Les socles techniques

Généralement, pour le développement d'applications, il est nécessaire de mettre en place une architecture et d'utiliser plusieurs frameworks dédiés à la mise en oeuvre des fonctionnalités auxquelles ils répondent. Cet ensemble d'entités est désigné par socle technique.

Un socle technique est donc composé d'une architecture et d'un ensemble de frameworks pour faciliter le développement des couches qui composent l'architecture (IHM, objets métiers, persistance des données, ...) et pour proposer des services techniques transverses (logging, ...)

Les attentes dans la définition d'un socle technique sont nombreuses :

- sélection des frameworks à mettre en oeuvre
- sélection des outils à utiliser
- définition de l'architecture applicative et de ses couches
- création d'un projet "vide" par assemblage des frameworks et des classes de bases des différentes couches
- validation par un prototype
- définition de normes de développement
- apprentissage du socle (plusieurs semaines de pratique sont nécessaires pour être généralement pleinement opérationnel)
- rédaction de la documentation technique
- ...

Généralement, un socle technique doit être défini spécifiquement pour les besoins de l'application ou des applications. Cependant, il est possible de s'appuyer sur un socle existant pour ne pas avoir à complètement réinventer la roue. Il existe des socles techniques open source.

109.2. Les frameworks pour les applications web

Les applications web sont généralement le point d'entrée principal pour les applications développées en utilisant la plate-forme J2EE.

Pourtant le développement d'applications web est relativement compliqué à concevoir et à implémenter pour plusieurs raisons :

- la nature du protocole http (basé sur un modèle simple de request/response sans état)
- les nombreuses technologies à mettre en oeuvre (HTML, XHTML, CSS, JavaScript, ...) ainsi que leurs différentes versions
- le support de ces technologies par les différents navigateurs est particulièrement différent
- HTML est un langage pauvre qui ne permet que le développement de formulaires assez rudimentaires

A ces inconvénients indépendants de la technologie côté serveur pour les mettre en oeuvre viennent s'ajouter des raisons spécifiques à la plate-forme J2EE :

- les API Servlet et JSP sont de bas niveau
- le manque de modèles événementiels (cet argument explique le développement de la technologie JSF)

Dès que l'on développe des applications web uniquement avec les API servlet et JSP, il apparait évident que de nombreuses parties dans des applications différentes sont communes mais doivent être réécrites ou réutilisées à chaque fois. Ceci est en grande partie lié au fait que les API servlets et JSP sont des API de bas niveau. Elles ne proposent par exemple rien pour automatiser l'extraction des données de la requête HTTP et mapper leur contenu dans un objet de façon fiable, assurer les transitions entre les pages selon les circonstances, ...

Les frameworks de développement web permettent généralement une séparation logique d'une application selon le concept proposé par le modèle MVC (Modèle/Vue/Contrôleur). Le framework le plus utilisé dans cette catégorie est Struts.

Certains éditeurs proposent des frameworks qui s'appuient sur un framework open source et facilitent leur utilisation en proposant des fonctionnalités dédiées dans leur IDE (Oracle ADF avec Jdeveloper, Beehive avec Weblogic Workshop, ...).

L'utilisation d'un framework web permet de faciliter le développement et la maintenance évolutive d'une application web. Les frameworks utilisent ou peuvent être complétés par des moteurs de templates qui facilitent la génération de page web à partir de modèles.

Les frameworks les plus récents (tel que Java Server Faces) mettent en oeuvre l'utilisation de composants côté serveur pour faciliter les développements (modèle événementiel et développement graphique)

Récemment, une nouvelle forme d'applications web est apparue : les applications riches. Elles peuvent prendre plusieurs formes et notamment dans les développements J2EE utiliser AJAX (reposant sur DHML pour des échanges asynchrones avec le serveur) ou Lazlo (reposant sur flash). Ces technologies rendent les applications plus riches et plus conviviales pour les rapprocher de ce que les utilisateurs connaissent avec le client lourd.

109.3. L'architecture pour les applications web

109.3.1. Le modèle MVC

Le modèle MVC (Model View Controler) a été initialement développé pour le langage Smalltalk dans le but de mieux structurer une application avec une interface graphique.

Ce modèle est un concept d'architecture qui propose une séparation en trois entités des données, des traitements et de l'interface :

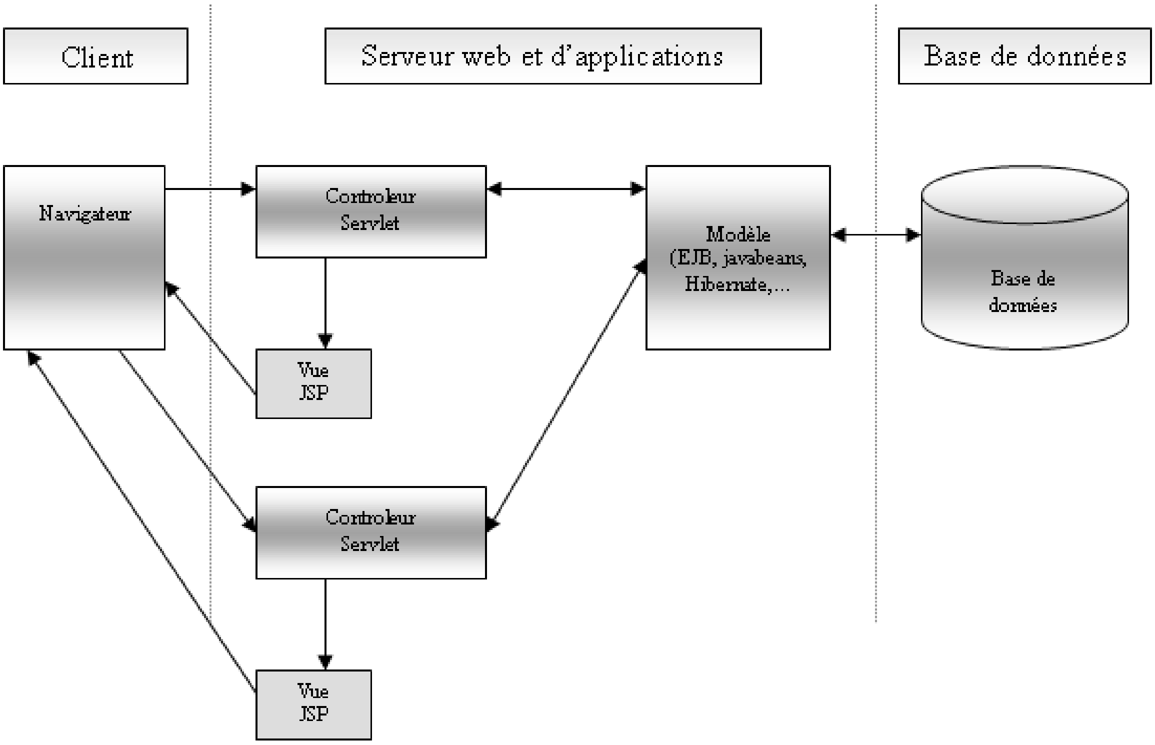
- le Modèle représente les données de l'application généralement stockées dans une base de données
- la Vue correspond à l'IHM (Interface Homme Machine)
- le Contrôleur assure les échanges entre la vue et le modèle notamment grâce à des composants métiers

Initialement utilisé pour le développement des interfaces graphiques, ce modèle peut se transposer pour les applications web sous la forme d'une architecture dite 3-tiers : la vue est mise en oeuvre par des JSP, le contrôleur est mis en oeuvre par des servlets et des Javabeans. Différents mécanismes peuvent être utilisés pour accéder aux données.

L'utilisation du modèle MVC rend un peu plus compliqué le développement de l'application qui le met en oeuvre mais il permet une meilleure structuration de celle-ci.

109.4. Le modèle MVC type 1

Dans ce modèle, chaque requête est traitée par un contrôleur sous la forme d'une servlet. Celle-ci traite la requête, fait appel aux éléments du model si nécessaire et redirige la requête vers une JSP qui se charge de créer la réponse à l'utilisateur.



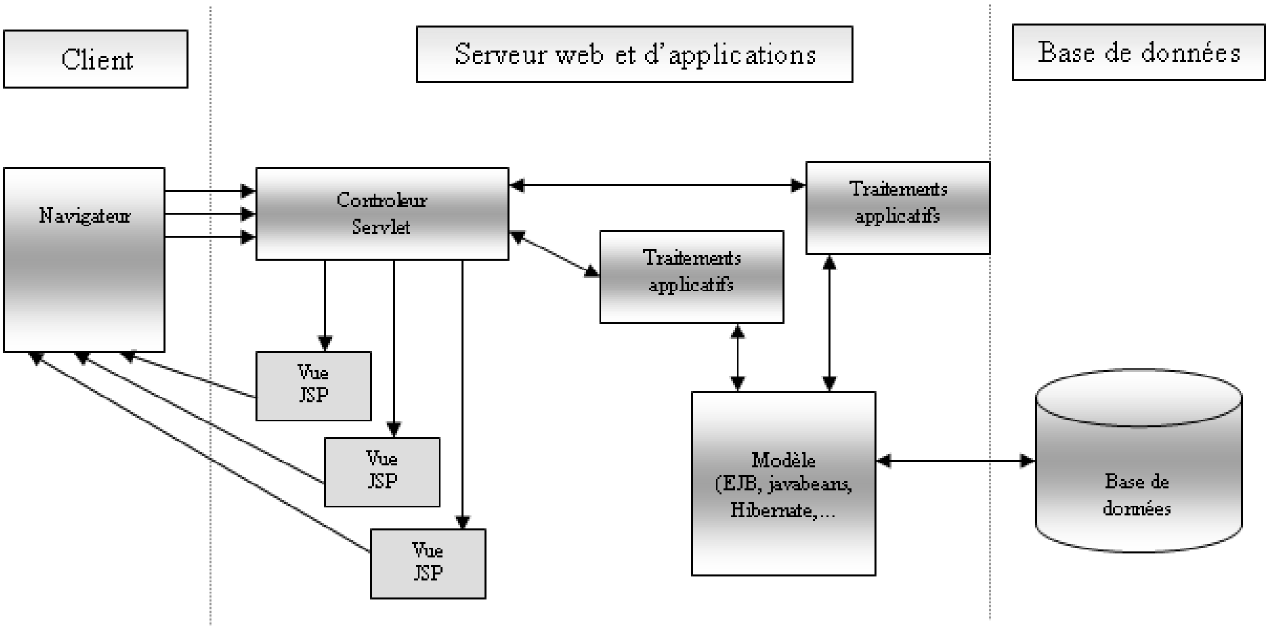
L'inconvénient est donc une multiplication du nombre de servlets nécessaires à l'application : l'implémentation de plusieurs servlets nécessite beaucoup de code à produire d'autant que chaque servlet doit être déclarée dans le fichier web.xml.

109.5. Le modèle MVC de type 2

Le principal défaut du modèle MVC est le nombre de servlets à développer pour une application.

Pour simplifier les choses, le modèle MVC model 2 ou MVC2 propose de n'utiliser qu'une seule et unique servlet comme contrôleur. Cette servlet se charge d'assurer le workflow des traitements en fonction des requêtes http reçues.

Le modèle MVC 2 est donc une évolution du modèle 1 : une unique servlet fait office de contrôleur et gère toutes les requêtes à traiter en fonction d'un paramétrage généralement sous la forme d'un fichier au format XML.



Le modèle MVC II conserve les principes du modèle MVC, mais il divise le contrôleur en deux parties en imposant un point d'entrée unique à toute l'application (première partie du contrôleur) qui déterminera à chaque requête reçue les traitements applicatifs à invoquer dynamiquement (seconde partie du contrôleur).

Une application web implémentant le modèle MVC de type II utilise une servlet comme contrôleur traitant les requêtes. En fonction de celles-ci, elle appelle les traitements dédiés généralement encapsulés dans une classe.

Dans ce modèle, le cycle de vie d'une requête est le suivant :

1. Le client envoie une requête à l'application et cette requête est prise en charge par la servlet faisant office de contrôleur
2. La servlet analyse la requête et appelle la classe dédiée contenant les traitements
3. Cette classe exécute les traitements nécessaires en fonction de la requête, notamment, en faisant appel aux objets métiers
4. En fonction du résultat, la servlet redirige la requête vers la page JSP
5. La JSP génère la réponse qui est renvoyée au client

109.5.1. Les différents types de framework web

Il existe deux grandes familles de framework web pour les applications J2EE : ceux reposant sur les requêtes et ceux reposant sur les composants.

Cette dernière famille est assez récente : elle repose aussi sur HTTP (donc sur les requêtes) mais propose de faciliter le développement et surtout de rendre plus riches les applications en utilisant des composants dont le rendu est généré par le serveur avec une gestion des événements. Ce type de framework va devenir la référence dans un futur proche et est proposé comme la nouvelle génération de framework pour le développement web.

En mars 2002, Tapestry a été le premier framework de ce type diffusé en open source. En Mars 2004, le JCP propose les Java Server Faces qui sont le standard de framework web intégré dans Java EE 5.

Les frameworks reposant sur le traitement de requêtes utilisent un cadre de développement exploitant les servlets et les JSP pour faciliter le traitement de certaines tâches répétitives (extraction des données de la requête, enchaînement des pages, utilisation de bibliothèques de tags spécifiques pour l'IHM, ...). Ces framework reposent généralement sur le modèle MVC2.

Exemple : Struts, WebWork, Spring MVC, ...

109.5.2. Des frameworks pour le développement web

Les frameworks présentés dans cette section regroupent les principaux frameworks open source et les JSF. Les frameworks open source sont nombreux, notamment : Spring MVC, Struts , Tapestry, Turbine, VRaptor, Wicket, ...

Seuls les principaux sont rapidement présentés dans les sections suivantes. Certains font l'objet d'un chapitre dédié.

109.5.2.1. Apache Struts

Struts est un projet du groupe Jakarta de la fondation Apache. La page officielle de Struts est à l'url : https://struts.apache.org/.

Struts est le framework open source le plus populaire : son utilisation est largement répandue.

Son architecture met en oeuvre le modèle MVC 2. Il utilise les servlets, les JSP, XML, les resourcesbundles (pour l'internationalisation) et des bibliothèques du projet Jakarta Commons.

Struts se concentre sur le contrôleur et la vue. Il ne propose rien concernant le modèle ce qui laisse le développeur libre pour mettre en oeuvre des Javabeans, des EJB ou toute autre solution pour la persistance des données.

La partie vue peut utiliser les tags proposés par Struts qui sont assez fournis mais qui nécessitent souvent l'ajout de bibliothèques supplémentaires pour gagner en productivité (exemple : Struts-Layout). Il est aussi possible d'utiliser ses propres tags et/ou d'utiliser la JSTL (Java Standard Tag Libraries).

Struts possède cependant quelques faiblesses : de conception assez anciennes, il est techniquement inférieur à d'autres frameworks notamment ceux reposant sur les composants.
Il est aussi légitime de s'interroger sur le futur de Struts, notamment avec l'arrivée des JSF qui vont devenir le standard de la plate-forme J2EE, et ce d'autant plus que le créateur de Struts participe activement aux spécifications des JSF par le JCP.

Les atouts de Struts sont :

- une bibliothèque de tags mature, robuste et complète.
- l'introspection des objets relatifs aux formulaires
- les formulaires de type DynaActionForm
- la validation est extensible, elle peut être faite côté client ou côté serveur.
- la gestion centralisée des exceptions
- la décomposition de l'application en modules logiques reposant sur le modèle MVC 2
- le support de l'internationalisation

Avantages	Inconvénients
Est quasiment un standard de fait	Son avenir
Nombreuses sources d'informations	Peu d'évolutions
Bibliothèques de tags HTML	une architecture vieillissante
Intégration dans les IDE	Beaucoup de classes et de code à produire (partiellement automatisable avec des outils dédiés)
Open source	

Struts est détaillé dans le chapitre «Struts».

109.5.2.2. Spring MVC

Le framework web Spring MVC fait partie intégrante du framework Spring. Ce dernier propose une architecture complète pour le développement d'applications J2EE dont le module MVC assure le développement de la partie IHM pour les applications web. L'utilisation de ce module peut être remplacée par d'autres frameworks notamment Struts grâce à la facilité de Spring à s'interfacer avec ces produits.

Le succès de Spring repose principalement sur la facilité qu'il a à permettre de développer des applications d'entreprise sans EJB tout en proposant les mêmes fonctionnalités que ces derniers grâce notamment à l'utilisation du design pattern IoC et à la programmation orientée aspect.

Le page officielle des projets Spring est à l'url : https://spring.io/

109.5.2.3. Tapestry

Tapestry est un projet open source développé par la fondation Apache.

La page officielle de ce projet est à l'url : https://tapestry.apache.org.

L'intérêt pour le projet s'est accru depuis la diffusion des JSF car ils utilisent tous les deux les composants serveurs.

La partie vue n'utilise pas de JSP mais des fichiers HTML enrichis avec des tags particuliers.

109.5.2.4. Java Server Faces

Les JSF (Java Server Faces) constituent un framework de composants graphiques, hébergés côté serveur et utiles pour le développement d'applications Web. C'est un framework basé sur des spécifications du JCP. Il dispose à ce titre d'une implémentation de référence.

Les Java Server Faces sont développés par le JCP sous la JSR 127 pour la version 1.0, la JSR 252 pour la version 1.2 et la JSR 314 pour la version 3.0.

La page officielle du projet est à l'url : https://www.oracle.com/java/technologies/javaserverfaces.html

Les JSF sont construits sur un modèle de développement orienté événementiel à l'image de SWING. Il se compose d'un ensemble d'API servant, notamment,

- à représenter les composants,
- à gérer les états et les événements
- à proposer des dispositifs de validation

Les valeurs ajoutées par ce framework sont :

- spécifications standardisées
- architecture de gestion des états des composants et des données correspondantes
- extensible pour créer de nouveaux composants
- support pour les différents types de clients (seul le renderer HTML est proposé dans l'implémentation de référence)
- séparation entre le comportement et la présentation : les applications Web basées sur la technologie JSP permettent cette séparation mais elle reste très partielle. Une page JSP ne peut pas mapper plusieurs requêtes http sur un composant graphique ou gérer un élément graphique en tant qu'objet sans état côte serveur

Ces spécifications possèdent une implémentation de référence (RI) ainsi que des implémentations open source (MyFaces, ...) et commerciales (ADF Faces, ...).

Avantages	Inconvénients
Standard du JCP	Maturité

Utilisation de composants et de la gestion d'événements	Manque de composants évolués (à développer ou à obtenir d'un tiers)
Possibilité de créer ses propres composants	Développement très orienté sur la vue (celle-ci contient la définition des validations et des méthodes à appeler pour les traitements)
Intégration forte dans certains IDE (Sun Studio Creator, ...) pour permettre des développements de type RAD	Peu de retour sur les performances (cycle de traitement des requêtes avec de nombreuses étapes)
Possibilité d'utiliser plusieurs types de rendus grâce à des toolkits (seul HTML est proposé en standard dans l'implémentation de référence)	
De base, pas besoin d'hériter de classes ou d'implémenter d'interfaces dédiées : de simples beans suffisent	

Les JSF sont détaillées dans le chapitre «JSF (Java Server Faces)».

109.5.2.5. Struts 2

La version 2 du framework Struts est constituée d'une fusion des framework Struts et WebWork.

Strut 2 requiert Java 5, Servlet 2.4 et JSP 2.0.

La page officielle du projet est à l'url : https://struts.apache.org/2.x/

109.5.2.6. Struts Shale

Le projet Struts Shale repose sur les JSF pour proposer de faciliter le développement d'applications web.

Ce projet est incompatible avec le framework Struts 1.x. Il a été abandonné en mai 2009.

La page officielle du projet est à l'url : https://shale.apache.org

109.5.2.7. Expresso

Expresso était un framework open source développé par la société JCorporate.

C'était un framework pour le développement d'applications Web J2EE qui agrège de nombreux autres frameworks chacun dédié à une tâche particulière.

Il repose sur Struts depuis sa version 4.0.

Une des particularités de ce framework est de proposer un ensemble de fonctionnalités assez complètes :

- outils de mapping objet-relationnel pour la persistance des données
- gestion d'un pool de connexions aux bases de données
- workflow
- identification et authentification pour la sécurité
- ...

La version 5.0 de ce framework est disponible depuis octobre 2002. La version 5.5, publiée en mai 2004 repose sur Struts 1.1. La version 5.6 est publiée en janvier 2005.

109.5.2.8. Jena

La page officielle de ce projet est à l'url : https://jena.apache.org

109.5.2.9. Turbine

La page officielle du projet est à l'url https://turbine.apache.org/

109.5.2.10. Wicket

La page officielle du projet est à l'url https://wicket.apache.org/

109.6. Les frameworks de mapping Objet/Relationel

Les frameworks de mapping Objet/relationel sont détaillés dans le chapitre «La persistance des objets».

109.7. Les frameworks de logging

Le logging est important dans toutes les applications pour faciliter le débogage lors du développement et conserver une trace de l'exécution lors de l'exploitation en production.

Une API très répandue est celle développée par le projet open source Log4j du groupe Jakarta.

Concient de l'importance du logging, Sun a développé et intégré au JDK 1.4 une API dédiée. Cette API est plus simple à utiliser et elle offre moins de fonctionnalités que Log4j mais elle a l'avantage d'être fournie directement avec le JDK.

Face au dilemme du choix de l'utilisation de ces deux API, le groupe Jakarta a développé une API qui permet d'utiliser indifféremment l'une ou l'autre selon leur disponibilité.

L'utilisation de ces entités est détaillée dans le chapitre «Le logging».

Il existe aussi d'autres frameworks de logging dont l'utilisation est largement moins répandue.

110. La génération de documents

Chapitre 110

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Il est fréquent qu'une application de gestion doive produire des documents dans différents formats. Ce chapitre présente plusieurs solutions open source pour permettre la génération de documents, notamment aux formats PDF et Excel.

Ce chapitre contient plusieurs sections :

- ♦ Apache POI
- ♦ iText

110.1. Apache POI

POI est l'acronyme de Poor Obfuscation Implementation. C'est un projet open source du groupe Apache, sous licence Apache V2, dont le but est de permettre la manipulation de fichiers de la suite bureautique Office de Microsoft, dans des applications Java mais sans utiliser Office.

L'implémentation de POI est intégralement réalisée en pur Java.

La manipulation ne peut se faire que sur des documents reposant sur le format Microsoft OLE2 (Object Linking and Embedding) Compound Document ce qui inclus les documents de la suite Office mais aussi les applications qui utilisent les ensembles de propriétés MFC pour sérialiser leurs documents.

Ce projet contient plusieurs composants :

- POIFS (Poor Obfuscation Implementation File System) : manipulation de fichiers utilisant le format Microsoft OLE 2 Compound Document
- HSSF (Horrible SpreadSheet Format) : manipulation des fichiers Excel (XLS) en lecture et écriture.
- HWPF (Horrible Word Processor Format) : manipulation de fichiers Word en lecture et certaines fonctionnalités en écriture.
- HPSF (Horrible Slide Layout Format) : manipulation de fichiers PowerPoint en lecture et écriture pour certains fonctionnalités mais pas toutes
- HDGF : lecture et uniquement extraction de texte de fichiers Visio
- HPSF : API pour manipuler les propriétés d'un fichier au format OLE 2 en lecture et en écriture

La version 3.0.1 a été diffusée en juillet 2007.

La version 3.1 a été diffusée fin juin 2008.

La version 3.5 en cours de développement devrait apporter le support des formats Office Open XML proposés depuis la version 2007 d'Office.

Ce projet est particulièrement intéressant car il permet la manipulation de documents au format Office sans que celui-ci soit installé et cela, même sur des systèmes d'exploitation non Microsoft Windows.

Le site officiel du projet est à l'url https://poi.apache.org/

La version utilisée dans cette section est la 3.1.

Le téléchargement de l'archive contenant la version binaire de POI se fait à l'url :

https://poi.apache.org/download.html

Il faut ensuite décompresser l'archive poi-bin-3.1-FINAL-20080629.zip obtenue dans un répertoire du système.

Pour utiliser PIO, il suffit d'ajouter le fichier poi-3.1-FINAL-20080629.jar au classpath de l'application.

110.1.1. POI-HSSF

HSSF permet la manipulation de document Excel de la version 97 à la version 2007 uniquement pour le format OLE2 (fichier avec l'extension .xls). Le format OOXML d'Excel 2007 n'est pas encours supporté (fichier avec l'extension .xlsx)

HSSF est une solution riche en fonctionnalités et fiable pour la manipulation de documents Excel en Java.

Un document Excel est composé de plusieurs éléments : un Dossier (Workbook) qui contient une ou plusieurs Feuilles (WorkSheets) étant elle-mêmes constituées de Lignes (Rows) comportant des cellules (Cells).

Les classes principales de l'API HSSF proposent d'encapsuler chacun de ces éléments. HSSF propose deux API pour manipuler un document Excel :

- user API : API la plus riche qui permet la lecture et l'écriture mais qui consomme beaucoup de ressources car le document est intégralement représenté dans un graphe d'objets (le pendant pour le traitement de documents XML pourrait être DOM). Les classes de cette API sont regroupées dans le package org.apache.poi.hssf.usermodel
- event API : API pour la lecture uniquement qui consomme peut de ressources (le pendant pour le traitement de documents XML pourrait être SAX). Les classes de cette API sont regroupées dans le package org.apache.poi.hssf.eventmodel et org.apache.poi.hssf.eventusermodel

La liste des packages de HSSF comprend notamment :

Package	Rôle
org.apache.poi.hssf.eventmodel	Classes pour gérer les événements émis lors de la lecture d'un document
org.apache.poi.hssf.eventusermodel	Classes pour lire un document
org.apache.poi.hssf.extractor	Classes pour extraire le texte d'un document
org.apache.poi.hssf.record.formula	Classes pour le support des formules dans les cellules
org.apache.poi.hssf.usermodel	Classes pour la manipulation de documents
org.apache.poi.hssf.util	Utilitaires pour faciliter la mise en oeuvre de certaines fonctionnalités

110.1.1.1. L'API de type usermodel

L'API de HSSF permet de créer, lire et modifier les documents Excel. Pour cela, elle contient de nombreuses classes dont les principales sont :

- POIFSFileSystem : classe qui permet d'accéder à un document existant
- HSSFWorkbook : classe qui encapsule un document
- HSSFSheet : classe qui encapsule une feuille d'un document
- HSSFRow : classe qui encapsule une ligne d'une feuille
- HSSFCell : classe qui encapsule une cellule d'une ligne

Cette API est riche en fonctionnalités mais elle consomme beaucoup de ressources notamment pour les gros de fichiers car ceux-ci sont intégralement représentés en mémoire dans une arborescence d'objets.

Parmi les nombreuses fonctionnalités proposées par cette API, il y a :

- lecture et écriture de document
- création et modification des différentes entités qui composent un document (document, feuille, ligne, cellule, ...)
- support de fonctionnalités avancées sur la feuille : sélection, zoom, support des panneaux, ...
- support des types de données d'une cellule (numérique et date, chaîne de caractère, formule)
- formatage des cellules (alignement, police, couleur, bordures, formats de données proposés en standard ou personnalisés,
- fonctionnalités avancées sur les cellules : taille, taille optimale, fusion, commentaires, ...
- paramètre d'impression d'une page (sélection de la zone d'impression, faire tenir sur une page, bas de page, ...)
- support graphique : dessin de primitives, d'images, ...

Seules quelques-unes de ces fonctionnalités sont détaillées dans les sections suivantes. Consultez la documentation de l'API pour obtenir des détails sur la mise en oeuvre des autres fonctionnalités.

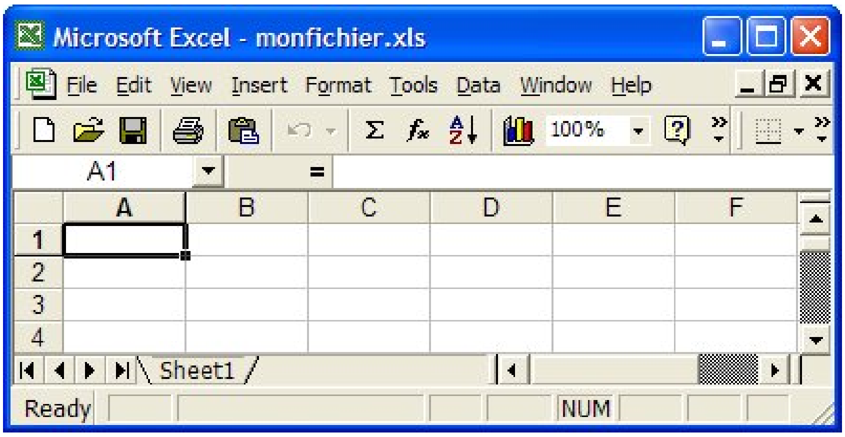
110.1.1.1.1. La création d'un nouveau document

Il suffit d'instancier un objet de type HSSFWorkBook et d'invoquer sa méthode write() pour créer le fichier.

Exemple :

```
package fr.jmdoudoux.dej.poi;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import org.apache.poi.hssf.usermodel.HSSFWorkbook;

public class TestPOI1 {

  public static void main(String[] args) {
    HSSFWorkbook wb = new HSSFWorkbook();
    FileOutputStream fileOut;
    try {
      fileOut = new FileOutputStream("monfichier.xls");
      wb.write(fileOut);
      fileOut.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

A l'exécution de cet exemple, un document Excel vierge est créé.

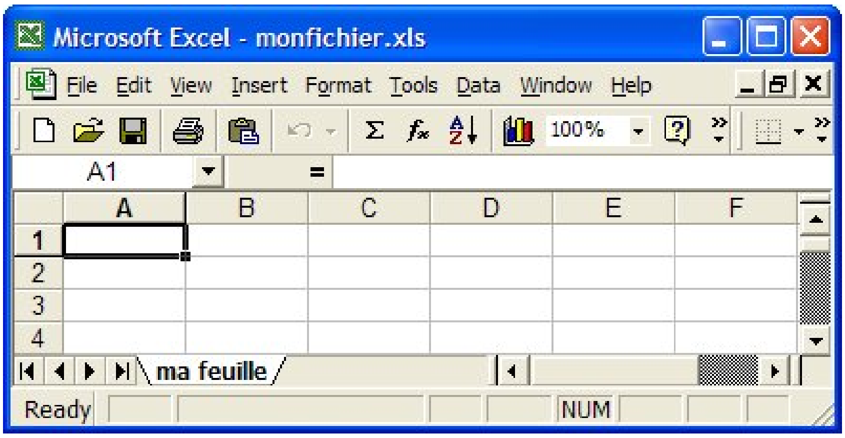

Microsoft Excel - monfichier.xls
File Edit View Insert Format Tools Data Window Help
100%
A1
=
A B C D E F
1 2 3 4
Sheet1
Ready
NUM

110.1.1.1.2. La création d'une nouvelle feuille

Une feuille est encapsulée dans la classe HSSFSheet. Pour créer une nouvelle feuille dans un document, il faut invoquer la méthode createSheet() de la classe HSSFWorkBook.

Exemple :

```
package fr.jmdoudoux.dej.poi;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;

public class TestPOI2 {

  public static void main(
      String[] args) {

    HSSFWorkbook wb = new HSSFWorkbook();
    HSSFSheet sheet = wb.createSheet("ma feuille");

    FileOutputStream fileOut;
    try {
      fileOut = new FileOutputStream("monfichier.xls");
      wb.write(fileOut);
      fileOut.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```


Microsoft Excel - monfichier.xls
File Edit View Insert Format Tools Data Window Help
100%
A1
=
A B C D E F
1
2
3
4
ma feuille
Ready
NUM

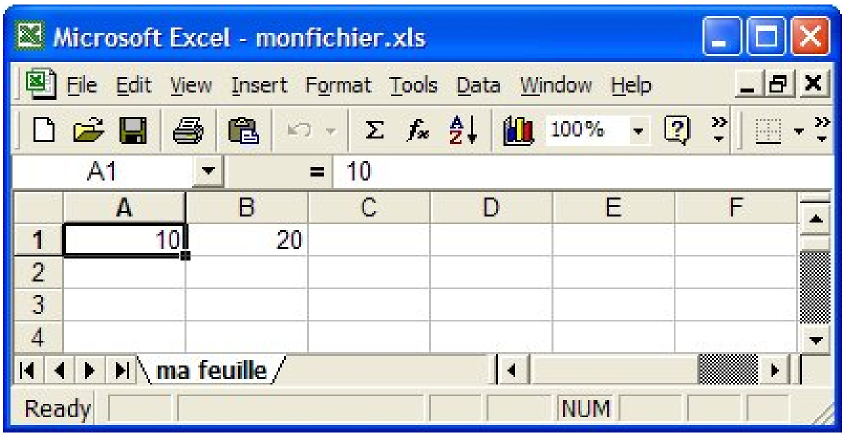
110.1.1.1.3. La création d'une nouvelle cellule

Une cellule d'une feuille est contenue dans une ligne qui est encapsulée dans un objet de type HSSFRow. Pour instancier un objet de ce type, il faut invoquer la méthode createRow() de la classe HSSFSheet. Cette méthode attend en paramètre le numéro de la ligne concernée sous la forme d'un entier de type int sachant que la première ligne possède l'index 0.

Une cellule est encapsulée dans la classe HSSFCell. Pour instancier un objet de ce type, il faut invoquer la méthode createCell() de la classe HSSFRow. Cette méthode attend en paramètre le numéro de la cellule concernée sous la forme d'un entier de type short sachant que la première cellule possède l'index 0.

Exemple :

```
package fr.jmdoudoux.dej.poi;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import org.apache.poi.hssf.usermodel.HSSFCell;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;

public class TestPOI3 {

  public static void main(
      String[] args) {

    HSSFWorkbook wb = new HSSFWorkbook();
    HSSFSheet sheet = wb.createSheet("ma feuille");

    HSSFRow row = sheet.createRow(0);
    HSSFCell cell = row.createCell((short)0);
    cell.setCellValue(10);

    row.createCell((short)1).setCellValue(20);

    FileOutputStream fileOut;
    try {
      fileOut = new FileOutputStream("monfichier.xls");
      wb.write(fileOut);
      fileOut.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```


Microsoft Excel - monfichier.xls
File Edit View Insert Format Tools Data Window Help
100%
A1
= 10
A
B
C
D
E
F
1
10
20
2
3
4
ma feuille
Ready
NUM

Remarque : seules les lignes ayant au moins une cellule sont ajoutées à la feuille. Seules les cellules non vides sont ajoutées à une ligne.

La méthode setCellValue() possède plusieurs surcharges pour fournir une valeur à la cellule selon plusieurs formats : int, boolean, double et des objets de type Calendar, Date et chaîne de caractères.

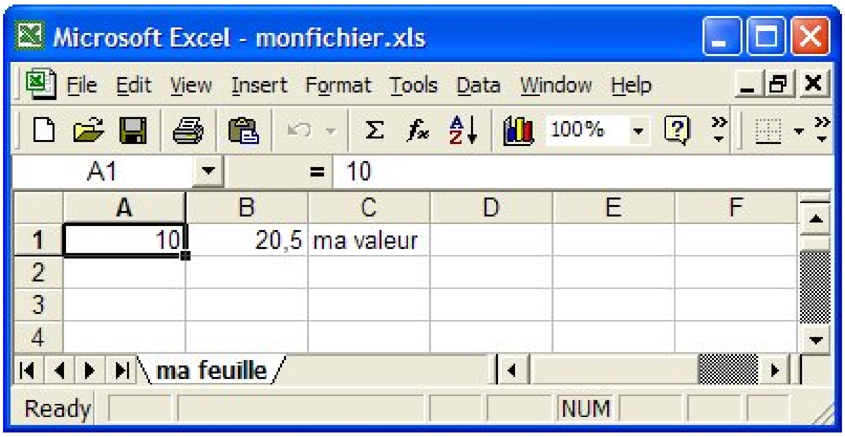
Remarque : pour les chaînes de caractères, la surcharge attendant en paramètre un objet de type String est deprecated au profit de la surcharge attendant en paramètre un objet de type HSSFRichTextString.

La méthode setCellType() permet de préciser le type des données de la cellule. Elle attend en paramètre une des constantes définies dans la classe HSSFCell : CELL_TYPE_BLANK, CELL_TYPE_BOOLEAN, CELL_TYPE_ERROR, CELL_TYPE_FORMULA, CELL_TYPE_NUMERIC, ou CELL_TYPE_STRING

Exemple :

```
package fr.jmdoudoux.dej.poi;
```

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import org.apache.poi.hssf.usermodel.HSSFCell;
import org.apache.poi.hssf.usermodel.HSSFRichTextString;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;

public class TestPOI4 {

  public static void main(
      String[] args) {

    HSSFWorkbook wb = new HSSFWorkbook();
    HSSFSheet sheet = wb.createSheet("ma feuille");

    HSSFRow row = sheet.createRow(0);
    HSSFCell cell = row.createCell((short)0);
    cell.setCellValue(10);

    row.createCell((short)1).setCellValue(20.5);

    row.createCell((short)2, HSSFCell.CELL_TYPE_STRING)
      .setCellValue(new HSSFRichTextString("ma valeur"));;

    FileOutputStream fileOut;
    try {
      fileOut = new FileOutputStream("monfichier.xls");
      wb.write(fileOut);
      fileOut.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```


Microsoft Excel - monfichier.xls
File Edit View Insert Format Tools Data Window Help
100%
A1
= 10
ma feuille
Ready
NUM

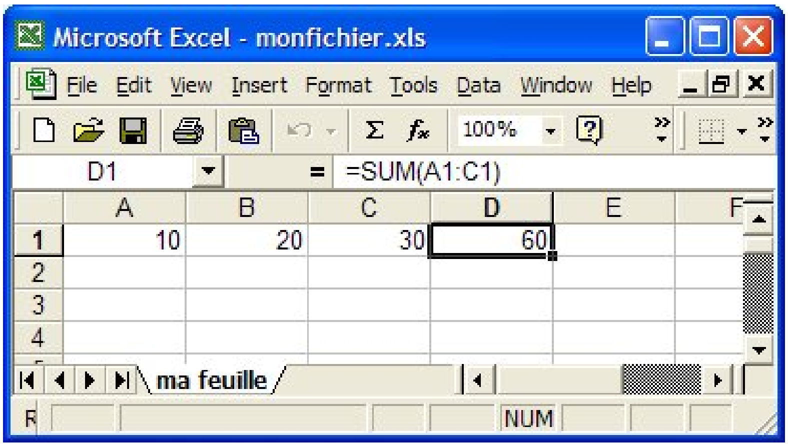
	A	B	C	D	E	F
1	10	20,5	ma valeur			
2						
3						
4						

Une des grandes forces d'Excel est de permettre l'application de formules plus ou moins complexes sur les données.

Pour assigner une formule à une cellule, il faut lui assigner le type FORMULA. La méthode setCellFormula() permet de définir la formule qui sera associée à la cellule.

Exemple :

```
package fr.jmdoudoux.dej.poi;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import org.apache.poi.hssf.usermodel.HSSFCell;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;

public class TestPOI13 {

  public static void main(
      String[] args) {

    HSSFWorkbook wb = new HSSFWorkbook();
    HSSFSheet sheet = wb.createSheet("ma feuille");

    HSSFRow row = sheet.createRow(0);
    HSSFCell cell = null;

    cell = row.createCell((short) 0);
    cell.setCellValue(10);

    cell = row.createCell((short) 1);
    cell.setCellValue(20);

    cell = row.createCell((short) 2);
    cell.setCellValue(30);

    cell = row.createCell((short) 3);
    cell.setCellType(HSSFCell.CELL_TYPE_FORMULA);
    cell.setCellFormula("SUM(A1:C1)");

    FileOutputStream fileOut;
    try {
      fileOut = new FileOutputStream("monfichier.xls");
      wb.write(fileOut);
      fileOut.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```



D1 = =SUM(A1:C1)

	A	B	C	D	E
1	10	20	30	60	
2					
3					
4					

110.1.1.1.4. Le formatage d'une cellule

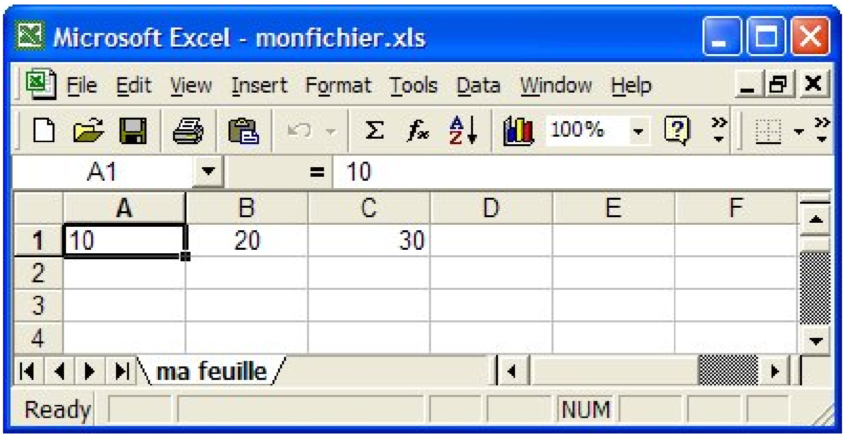
Le formatage d'une cellule se fait à l'aide d'un objet de type HSSFCellStyle. La classe HSSFCellStyle permet de définir le format des données, d'aligner les valeurs dans la cellule, de définir ses bordures, ...

La méthode setCellStyle() de la classe HSSFCell permet d'associer un style à la cellule.

Pour définir un style qui permet d'aligner les données, il faut utiliser la méthode setAlignment() de la classe HSSFCellStyle. Celle-ci attend en paramètre une des constantes suivantes : HSSFCellStyle.ALIGN_CENTER, HSSFCellStyle.ALIGN_CENTER_SELECTION, HSSFCellStyle.ALIGN_FILL, HSSFCellStyle.ALIGN_GENERAL, HSSFCellStyle.ALIGN_JUSTIFY, HSSFCellStyle.ALIGN_LEFT, HSSFCellStyle.ALIGN_RIGHT

Exemple :

```
package fr.jmdoudoux.dej.poi;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import org.apache.poi.hssf.usermodel.HSSFCell;
import org.apache.poi.hssf.usermodel.HSSFCellStyle;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;

public class TestPOI5 {

  public static void main(
      String[] args) {

    HSSFWorkbook wb = new HSSFWorkbook();
    HSSFSheet sheet = wb.createSheet("ma feuille");

    HSSFRow row = sheet.createRow(0);
    HSSFCell cell = null;
    HSSFCellStyle cellStyle = null;

    cell = row.createCell((short) 0);
    cell.setCellValue(10);
    cellStyle = wb.createCellStyle();
    cellStyle.setAlignment(HSSFCellStyle.ALIGN_LEFT);
    cell.setCellStyle(cellStyle);

    cell = row.createCell((short) 1);
    cell.setCellValue(20);
    cellStyle = wb.createCellStyle();
    cellStyle.setAlignment(HSSFCellStyle.ALIGN_CENTER);
    cell.setCellStyle(cellStyle);

    cell = row.createCell((short) 2);
    cell.setCellValue(30);
    cellStyle = wb.createCellStyle();
    cellStyle.setAlignment(HSSFCellStyle.ALIGN_RIGHT);
    cell.setCellStyle(cellStyle);

    FileOutputStream fileOut;
    try {
      fileOut = new FileOutputStream("monfichier.xls");
      wb.write(fileOut);
      fileOut.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```



	A	B	C	D	E	F
1	10	20	30			
2						
3						
4						

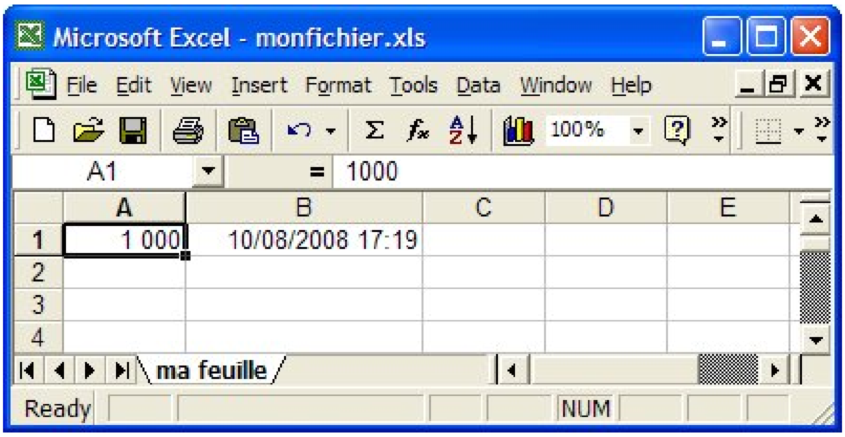
Pour préciser le format des données de la cellule, il faut utiliser la méthode setDataFormat() de la classe HSSFCellStyle. Elle attend en paramètre un entier qui précise le type selon les valeurs gérées par la classe HSSFDataFormat.

La classe HSSFDataFormat propose plusieurs méthodes statiques pour obtenir un des formatages prédéfinis.

Pour utiliser un format personnalisé, il faut invoquer la méthode createDataFormat() de la classe HSSFWorkbook de l'instance qui encapsule le document pour obtenir une instance de la classe HSSFDataFormat. L'invocation de la méthode getFormat() de cette instance permet de définir son format personnalisé.

Exemple :

```
package fr.jmdoudoux.dej.poi;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Date;

import org.apache.poi.hssf.usermodel.HSSFCell;
import org.apache.poi.hssf.usermodel.HSSFCellStyle;
import org.apache.poi.hssf.usermodel.HSSFDataFormat;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;

public class TestPOI6 {

  public static void main(
      String[] args) {

    HSSFWorkbook wb = new HSSFWorkbook();
    HSSFSheet sheet = wb.createSheet("ma feuille");

    HSSFRow row = sheet.createRow(0);
    HSSFCell cell = null;
    HSSFCellStyle cellStyle = null;

    cell = row.createCell((short) 0);
    cell.setCellValue(1000);
    cellStyle = wb.createCellStyle();
    cellStyle.setDataFormat(HSSFDataFormat.getBuiltinFormat("#,##0"));
    cell.setCellStyle(cellStyle);

    cell = row.createCell((short) 1);
    cell.setCellValue(new Date());
    cellStyle = wb.createCellStyle();
    HSSFDataFormat hssfDataFormat = wb.createDataFormat();
    cellStyle.setDataFormat(hssfDataFormat.getFormat("dd/mm/yyyy h:mm"));
    cell.setCellStyle(cellStyle);

    FileOutputStream fileOut;
    try {
      fileOut = new FileOutputStream("monfichier.xls");
      wb.write(fileOut);
      fileOut.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```


Microsoft Excel - monfichier.xls
File Edit View Insert Format Tools Data Window Help
100%
A1
= 1000
A
B
C
D
E
1
1 000
10/08/2008 17:19
2
3
4
ma feuille
Ready
NUM

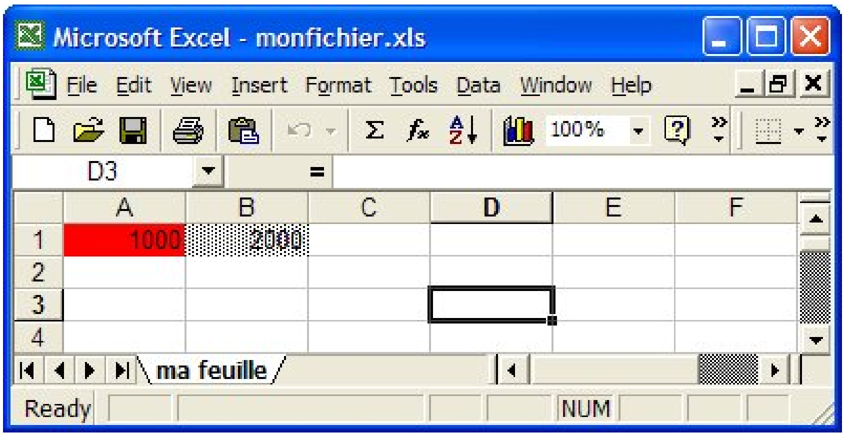
Pour modifier le fond et l'apparence d'une cellule, il faut utiliser les méthodes setFillBackgroundColor(), setFillForegroundColor() et setFillPattern() de la classe HSSFCellStyle.

Les méthodes setFillBackgroundColor() et setFillForegroundColor() attendent en paramètre un entier de type short : la classe HSSFColor possède de nombreuses classes filles pour faciliter l'utilisation d'une couleur.

La méthode setFillPattern() attend en paramètre un entier de type short : la classe HSSFCellStyle propose de nombreuses constantes pour faciliter l'utilisation d'un motif de remplissage.

Exemple :

```
package fr.jmdoudoux.dej.poi;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import org.apache.poi.hssf.usermodel.HSSFCell;
import org.apache.poi.hssf.usermodel.HSSFCellStyle;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;
import org.apache.poi.hssf.util.HSSFColor;

public class TestPOI7 {

  public static void main(
      String[] args) {

    HSSFWorkbook wb = new HSSFWorkbook();
    HSSFSheet sheet = wb.createSheet("ma feuille");

    HSSFRow row = sheet.createRow(0);
    HSSFCell cell = null;
    HSSFCellStyle cellStyle = null;

    cell = row.createCell((short) 0);
    cell.setCellValue(1000);
    cellStyle = wb.createCellStyle();
    cellStyle.setFillForegroundColor(HSSFColor.RED.index);
    cellStyle.setFillPattern(HSSFCellStyle.SOLID_FOREGROUND);
    cell.setCellStyle(cellStyle);

    cell = row.createCell((short) 1);
    cell.setCellValue(2000);
    cellStyle = wb.createCellStyle();
    cellStyle.setFillForegroundColor(HSSFColor.YELLOW.index);
    cellStyle.setFillPattern(HSSFCellStyle.ALT_BARS);
    cellStyle.setFillForegroundColor(HSSFColor.WHITE.index);
    cell.setCellStyle(cellStyle);

    FileOutputStream fileOut;
    try {
      fileOut = new FileOutputStream("monfichier.xls");
      wb.write(fileOut);
      fileOut.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

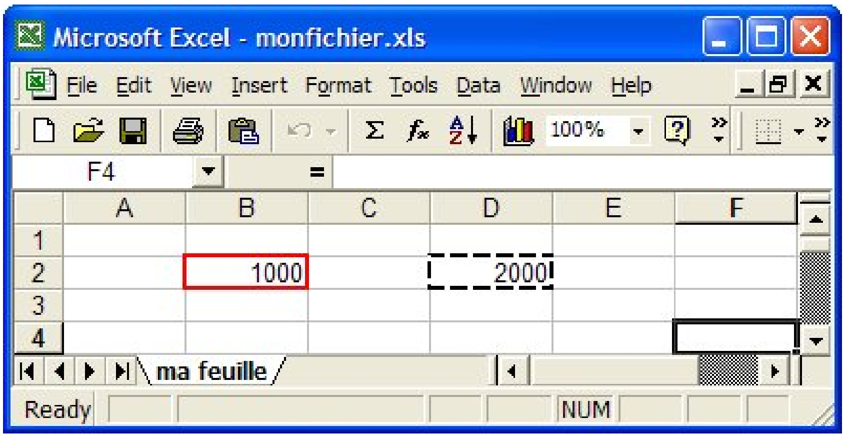

Microsoft Excel - monfichier.xls
File Edit View Insert Format Tools Data Window Help
100%
D3
=
A
B
C
D
E
F
1
1000
2000
2
3
4
ma feuille
Ready
NUM

Pour mettre des bordures sur les côtés des cellules, la classe HSSFCellStyle propose plusieurs méthodes :

- setBorderXxx : permet de préciser la forme de la bordure en utilisant les constantes définies dans la classe HSSFCellStyle
- setXxxBorderColor : permet de préciser la couleur de la bordure

Exemple :

```
package fr.jmdoudoux.dej.poi;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import org.apache.poi.hssf.usermodel.HSSFCell;
import org.apache.poi.hssf.usermodel.HSSFCellStyle;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;
import org.apache.poi.hssf.util.HSSFColor;

public class TestPOI8 {

  public static void main(
      String[] args) {

    HSSFWorkbook wb = new HSSFWorkbook();
    HSSFSheet sheet = wb.createSheet("ma feuille");

    HSSFRow row = sheet.createRow(1);
    HSSFCell cell = null;
    HSSFCellStyle cellStyle = null;

    cell = row.createCell((short) 1);
    cell.setCellValue(1000);
    cellStyle = wb.createCellStyle();
    cellStyle.setBorderBottom(HSSFCellStyle.BORDER_MEDIUM);
    cellStyle.setBottomBorderColor(HSSFColor.RED.index);
    cellStyle.setBorderLeft(HSSFCellStyle.BORDER_MEDIUM);
    cellStyle.setLeftBorderColor(HSSFColor.RED.index);
    cellStyle.setBorderRight(HSSFCellStyle.BORDER_MEDIUM);
    cellStyle.setRightBorderColor(HSSFColor.RED.index);
    cellStyle.setBorderTop(HSSFCellStyle.BORDER_MEDIUM);
    cellStyle.setTopBorderColor(HSSFColor.RED.index);
    cell.setCellStyle(cellStyle);

    cell = row.createCell((short) 3);
    cell.setCellValue(2000);
    cellStyle = wb.createCellStyle();
    cellStyle.setBorderBottom(HSSFCellStyle.BORDER_MEDIUM_DASHED);
    cellStyle.setBottomBorderColor(HSSFColor.BLACK.index);
    cellStyle.setBorderLeft(HSSFCellStyle.BORDER_MEDIUM_DASHED);
    cellStyle.setLeftBorderColor(HSSFColor.BLACK.index);
    cellStyle.setBorderRight(HSSFCellStyle.BORDER_MEDIUM_DASHED);
    cellStyle.setRightBorderColor(HSSFColor.BLACK.index);
    cellStyle.setBorderTop(HSSFCellStyle.BORDER_MEDIUM_DASHED);
    cellStyle.setTopBorderColor(HSSFColor.BLACK.index);
    cell.setCellStyle(cellStyle);

    FileOutputStream fileOut;
    try {
      fileOut = new FileOutputStream("monfichier.xls");
      wb.write(fileOut);
      fileOut.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

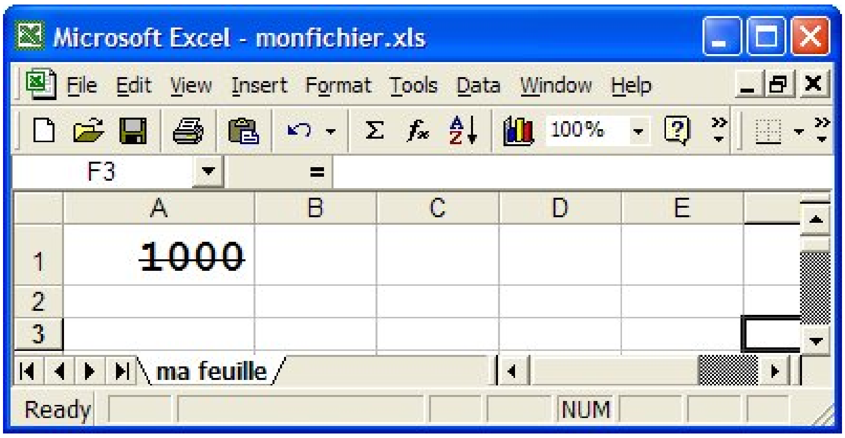

Microsoft Excel - monfichier.xls
File Edit View Insert Format Tools Data Window Help
100%
F4
=
A B C D E F
1
2
3
4
1000
2000
ma feuille
Ready
NUM

Pour utiliser une police de caractères, il faut utiliser la méthode setFont() de la classe HSSFCellStyle qui attend en paramètre un objet de type HSSFFont encapsulant une police de caractères.

Pour obtenir une instance de la classe HSSFFont, il faut invoquer la méthode createFont() de la classe HSSFWorkbook. La classe HSSFFont possède plusieurs méthodes pour définir les caractéristiques de la police de caractères : famille, taille, gras, souligné, italique, barré, ...

Exemple :

```
package fr.jmdoudoux.dej.poi;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import org.apache.poi.hssf.usermodel.HSSFCell;
import org.apache.poi.hssf.usermodel.HSSFCellStyle;
import org.apache.poi.hssf.usermodel.HSSFFont;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;

public class TestPOI9 {

  public static void main(
      String[] args) {

    HSSFWorkbook wb = new HSSFWorkbook();
    HSSFSheet sheet = wb.createSheet("ma feuille");

    HSSFRow row = sheet.createRow(0);
    HSSFCell cell = null;
    HSSFCellStyle cellStyle = null;

    HSSFFont fonte = wb.createFont();
    fonte.setFontHeightInPoints((short) 18);
    fonte.setFontName("Courier New");
    fonte.setBoldweight(HSSFFont.BOLDWEIGHT_BOLD);
    fonte.setStrikeout(true);

    cell = row.createCell((short) 0);
    cell.setCellValue(1000);
    cellStyle = wb.createCellStyle();
    cellStyle.setFont(fonte);
    cell.setCellStyle(cellStyle);

    FileOutputStream fileOut;
    try {
      fileOut = new FileOutputStream("monfichier.xls");
      wb.write(fileOut);
      fileOut.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```


Microsoft Excel - monfichier.xls
File Edit View Insert Format Tools Data Window Help
100%
F3
=
A
B
C
D
E
1
~~1000~~
2
3
ma feuille
Ready
NUM

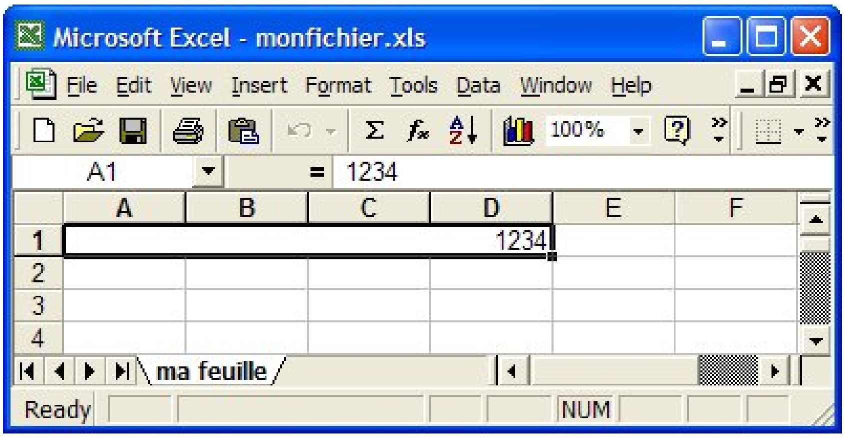
110.1.1.1.5. La fusion de cellules

Pour fusionner un ensemble de cellules, il faut invoquer la méthode addMergedRegion() de la classe HSSFSheet. Elle attend en paramètre un objet de type org.apache.poi.hssf.util.Region qui permet de définir l'ensemble des cellules à fusionner.

Un des constructeurs de la classe Region attend en paramètre quatre entiers qui correspondent respectivement au numéro et à la colonne de la première ligne, au numéro et à la colonne de la dernière ligne de l'ensemble des cellules.

Exemple :

```
package fr.jmdoudoux.dej.poi;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import org.apache.poi.hssf.usermodel.HSSFCell;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;
import org.apache.poi.hssf.util.Region;

public class TestPOI10 {
  public static void main(
      String[] args) {

    HSSFWorkbook wb = new HSSFWorkbook();
    HSSFSheet sheet = wb.createSheet("ma feuille");

    HSSFRow row = sheet.createRow(0);
    HSSFCell cell = null;

    cell = row.createCell((short) 0);
    cell.setCellValue(1234);

    sheet.addMergedRegion(new Region(0,(short)0,0,(short)3));

    FileOutputStream fileOut;
    try {
      fileOut = new FileOutputStream("monfichier.xls");
      wb.write(fileOut);
      fileOut.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```


Microsoft Excel - monfichier.xls
File Edit View Insert Format Tools Data Window Help
100%
A1
= 1234
A B C D E F
1 2 3 4
1234
ma feuille
Ready
NUM

110.1.1.1.6. La lecture et la modification d'un document

Pour lire un document, il faut utiliser la classe POIFSFileSystem. Un des constructeurs de cette classe attend en paramètre un objet de type InputStream qui encapsule un flux vers le document à lire.

Il suffit de fournir l'instance de la classe POIFSFileSystem en paramètre du constructeur de la classe HSSFWorkbook pour lire le document et produire une arborescence d'objets qui encapsule son contenu.

Il est ensuite possible d'utiliser ces objets pour modifier le contenu du document et l'enregistrer une fois les modifications terminées.

Les méthodes getNumericCellValue() et getRichStringCellValue() de la classe HSSFCell permettent d'obtenir la valeur de la cellule selon son type.

La méthode setCellType() de la classe HSSFCell permet de préciser le type du contenu de la cellule (CELL_TYPE_BLANK, CELL_TYPE_BOOLEAN, CELL_TYPE_ERROR, CELL_TYPE_FORMULA, CELL_TYPE_NUMERIC, CELL_TYPE_STRING)

Plusieurs surcharges de la méthode setValue() de la classe HSSFCell permettent de fournir la valeur de la cellule.

Exemple :

```
package fr.jmdoudoux.dej.poi;
```

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import org.apache.poi.hssf.usermodel.HSSFCell;
import org.apache.poi.hssf.usermodel.HSSFRichTextString;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;
import org.apache.poi.poifs.filesystem.POIFSFileSystem;

public class TestPOI11 {

  public static void main(
      String[] args) {

    try {
      FileOutputStream fileOut;

      POIFSFileSystem fs = new POIFSFileSystem(new FileInputStream("monfichier.xls"));
      HSSFWorkbook wb = new HSSFWorkbook(fs);
      HSSFSheet sheet = wb.getSheetAt(0);
      HSSFRow row = sheet.getRow(0);

      HSSFCell cell = row.getCell((short) 0);
      if (cell != null)
          row.removeCell(cell);
      cell = row.createCell((short) 0);
      cell.setCellType(HSSFCell.CELL_TYPE_STRING);
      cell.setCellValue(new HSSFRichTextString("données modifiées"));

      fileOut = new FileOutputStream("monfichier.xls");
      wb.write(fileOut);
      fileOut.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Pour obtenir une donnée, il est préférable de s'assurer du type de données associé à la cellule en utilisant la méthode getCellType().

Une exception est levée si le type de données demandé ne correspond pas à la méthode invoquée, par exemple : appel de la méthode getCellValue() sur une cellule de type STRING.

Remarque : Excel stocke les dates sous une forme numérique. Pour les identifier, il faut regarder le format des données.

110.1.1.1.7. Le parcours des cellules d'une feuille

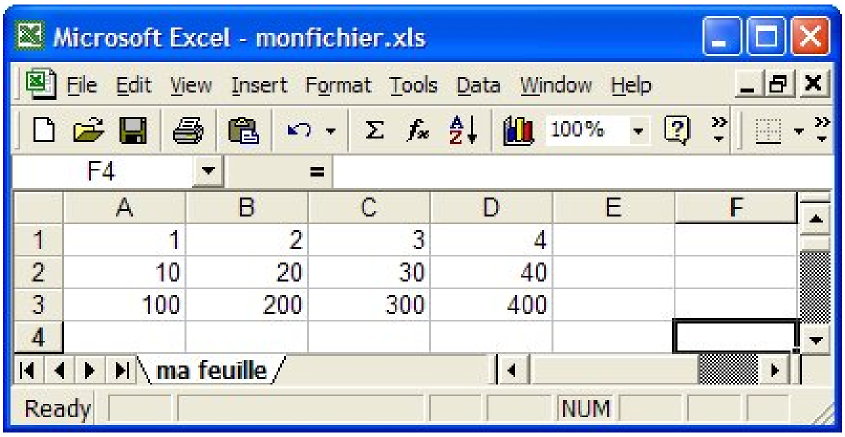
La classe HSSFSheet propose la méthode rowIterator() qui renvoie un objet de type Iterator permettant de parcourir les lignes de la feuille.

Attention : seules les lignes non vides sont contenues dans l'iterator. Ainsi il n'y pas de corrélation entre le numéro de la ligne de l'iterator et le numéro de la ligne dans la feuille. Pour connaître le numéro de la ligne dans la feuille, il faut utiliser la méthode getRowNum() de la classe HSSFRow.

La classe HSSFRow propose la méthode cellIterator() qui renvoie un objet de type Iterator permettant de parcourir les cellules de la ligne.

Attention : seules les cellules non vides sont contenues dans l'iterator. Ainsi il n'y pas de corrélation entre le numéro de la cellule de l'iterator et le numéro de la cellule dans la ligne. Pour connaître le numéro de la colonne dans la ligne, il faut utiliser la méthode getCellNum() de la classe HSSFCell.

L'exemple ci-dessous va utiliser le fichier suivant :


Microsoft Excel - monfichier.xls
File Edit View Insert Format Tools Data Window Help
100%
F4
=
ma feuille
Ready
NUM

	A	B	C	D	E	F
1	1	2	3	4		
2	10	20	30	40		
3	100	200	300	400		
4						

L'application va parcourir les cellules et afficher le total de chaque ligne et le total de toutes les cellules.

Exemple :

```
package fr.jmdoudoux.dej.poi;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Iterator;

import org.apache.poi.hssf.usermodel.HSSFCell;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;
import org.apache.poi.poifs.filesystem.POIFSFileSystem;

public class TestPOI12 {

  public static void main(
      String[] args) {

    try {
      POIFSFileSystem fs = new POIFSFileSystem(new FileInputStream("monfichier.xls"));
      HSSFWorkbook wb = new HSSFWorkbook(fs);
      HSSFSheet sheet = wb.getSheetAt(0);
      HSSFRow row = null;
      HSSFCell cell = null;
      double totalLigne = 0.0;
      double totalGeneral = 0.0;
      int numLigne = 1;

      for (Iterator rowIt = sheet.rowIterator(); rowIt.hasNext();) {
        totalLigne = 0;
        row = (HSSFRow) rowIt.next();
        for (Iterator cellIt = row.cellIterator(); cellIt.hasNext();) {
          cell = (HSSFCell) cellIt.next();
          totalLigne += cell.getNumericCellValue();
        }
        System.out.println("total ligne "+numLigne+" = "+totalLigne);
        totalGeneral += totalLigne;
        numLigne++;
      }
      System.out.println("total general "+totalGeneral);
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
total ligne 1 = 10.0
total ligne 2 = 100.0
total ligne 3 = 1000.0
total general 1110.0
```

Il est aussi possible d'utiliser les generics de Java 5.

Exemple :

```
package fr.jmdoudoux.dej.poi;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Iterator;

import org.apache.poi.hssf.usermodel.HSSFCell;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;
import org.apache.poi.poifs.filesystem.POIFSFileSystem;

public class TestPOI12 {

  public static void main(
      String[] args) {

    try {
      POIFSFileSystem fs = new POIFSFileSystem(new FileInputStream("monfichier.xls"));
      HSSFWorkbook wb = new HSSFWorkbook(fs);
      HSSFSheet sheet = wb.getSheetAt(0);
      HSSFRow row = null;
      HSSFCell cell = null;
      double totalLigne = 0.0;
      double totalGeneral = 0.0;
      int numLigne = 1;

      for (Iterator<HSSFRow> rowIt = (Iterator<HSSFRow>) sheet.rowIterator();
        rowIt.hasNext();) {
        totalLigne = 0;
        row = rowIt.next();
        for (Iterator<HSSFCell> cellIt = (Iterator<HSSFCell>) row.cellIterator();
          cellIt.hasNext();) {
          cell = cellIt.next();
          totalLigne += cell.getNumericCellValue();
        }
        System.out.println("total ligne "+numLigne+" = "+totalLigne);
        totalGeneral += totalLigne;
        numLigne++;
      }
      System.out.println("total general "+totalGeneral);
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

110.1.1.1.8. La génération d'un document Excel dans une servlet

Il peut être utile de faire générer un document Excel par une servlet pour que celle-ci retourne le document dans sa réponse.

Il est nécessaire de correctement positionner le type mime sur "application/vnd.ms-excel" qui désigne l'application Excel.

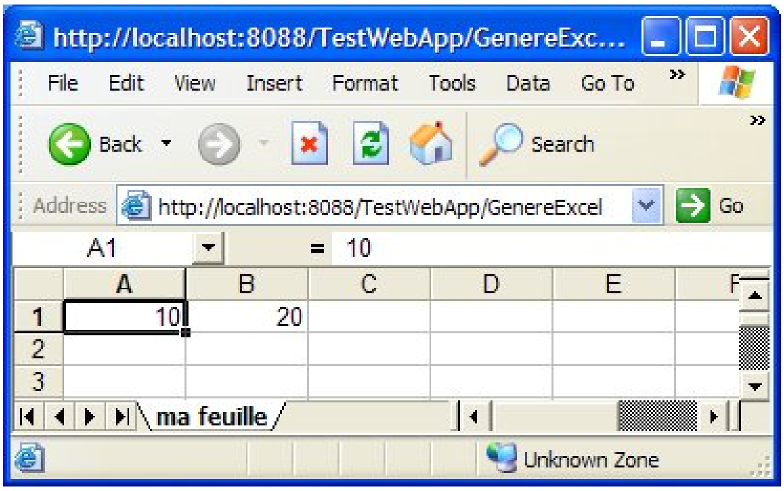
Il est aussi utile de définir la propriété "Content-disposition" pour faciliter l'enregistrement du document par l'utilisateur.

Enfin, il faut simplement fournir en paramètre de la méthode write() de la classe HSSFWorkbook le flux de sortie de la réponse HTTP de la servlet.

Exemple :

```
package fr.jmdoudoux.dej.poi;
```

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.OutputStream;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.poi.hssf.usermodel.HSSFCell;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;

public class GenereExcel extends javax.servlet.http.HttpServlet
  implements javax.servlet.Servlet {

  public GenereExcel() {
    super();
  }

  protected void doGet(
      HttpServletRequest request,
      HttpServletResponse response) throws ServletException, IOException {
    try {
      OutputStream out = response.getOutputStream();

      response.setContentType("application/vnd.ms-excel");

      response.setHeader("Content-disposition", "inline; filename=monfichier.xls");
      HSSFWorkbook wb = new HSSFWorkbook();

      HSSFSheet sheet = wb.createSheet("ma feuille");

      HSSFRow row = sheet.createRow(0);
      HSSFCell cell = row.createCell((short) 0);
      cell.setCellValue(10);

      row.createCell((short) 1).setCellValue(20);

      wb.write(out);
      out.flush();
      out.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```



110.1.1.2. L'API de type eventusermodel

L'utilisation de cette API est particulièrement adaptée à la lecture de gros fichiers Excel car elle ne charge pas le document en mémoire mais émet des événements lors de sa lecture. Cette API permet uniquement la lecture de document.

Sa mise en oeuvre n'est cependant pas triviale et nécessite quelques notions sur la structure de bas niveau du document Excel.

Consultez la documentation de POI-HSSF pour le détail de sa mise en oeuvre.

110.2. iText

iText est une API open source qui permet la génération de documents PDF, RTF et HTML. Elle est diffusée sous deux licences : MPL et LGPL.

Le site officiel de cette API est à l'url : https://itextpdf.com

iText contient de très nombreuses classes permettant de réaliser des actions basiques comme avancées, notamment pour la génération de documents de type PDF.

iText est une API qui permet d'intégrer dans une application la génération dynamique de documents : ceci est particulièrement utile lorsque le contenu du document dépend d'informations fournies par l'utilisateur ou encore quand ce contenu est calculé.

La possibilité d'iText d'exporter un document créé avec l'API dans différents formats peut être pratique. Le document exporté peut en outre être envoyé vers différents flux (fichier, réponse http d'une servlet, console, ...).

iText permet aussi la mise en oeuvre de fonctionnalités avancées sur un document PDF :

- définition de marque-pages, filigranes, ...
- signature numérique
- remplissage de formulaires
- diviser un document ou assembler plusieurs documents
- ...

iText requiert un JDK 1.4 minimum et l'API BouncyCastle pour certaines fonctionnalités.

110.2.1. Un exemple très simple

L'exemple proposé va créer un document PDF qui contient "Hello World".

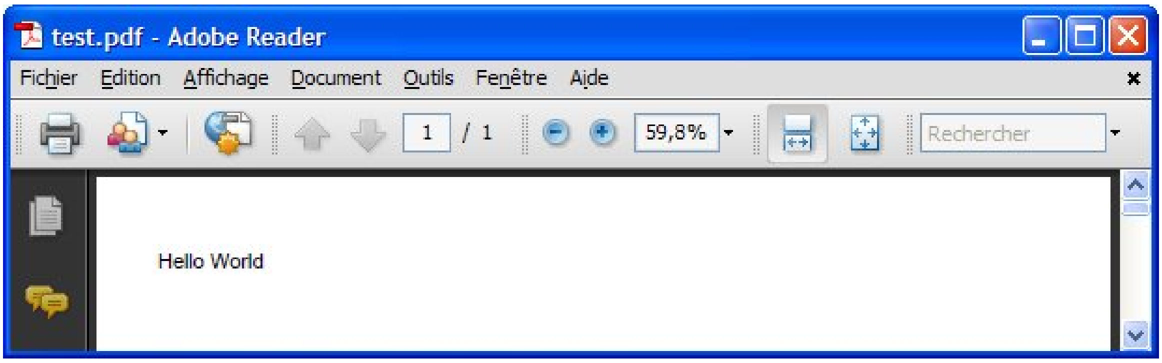
La création d'un document PDF avec iText se fait en cinq étapes :

- Instanciation d'un objet de type Document
- Instanciation d'un objet de type PdfWriter pour exporter le document
- Appel de la méthode open() du document
- Création et ajout des éléments qui composent le document
- Appel de la méthode close() pour exporter le document

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.io.FileOutputStream;
import java.io.IOException;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText1 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document,
          new FileOutputStream("c:/test.pdf"));
      document.open();
      document.add(new Paragraph("Hello World"));
    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

Pour compiler et exécuter cet exemple, il faut ajouter la bibliothèque iText-2.1.3.jar au classpath.



Hello World

110.2.2. L'API de iText

L'API de iText contient des objets qui proposent des fonctionnalités pour la création de documents notamment au format PDF. Ils encapsulent par exemple le document et chacun des éléments qui peuvent le composer tels que les objets de type Font, Paragraph, Chapter, Anchor, Image, List, Table, ...

L'API de iText est composée de nombreuses classes : seules les principales sont présentées dans cette section. Le site officiel et la Javadoc de l'API fournissent des informations détaillées sur l'ensemble des fonctionnalités de chacune des classes.

110.2.3. La création d'un document

La création d'un document avec iText se fait en plusieurs étapes :

- instanciation d'un objet de type Document
- instanciation d'un objet de type Writer pour exporter le document
- appel de la méthode open() du document
- création et ajout des éléments qui composent le document
- appel de la méthode close() pour exporter le document

110.2.3.1. La classe Document

La classe Document est un conteneur pour le contenu d'un document.

L'objet Document possède plusieurs constructeurs :

Constructeur	Rôle
Document()	constructeur par défaut
Document(Rectangle pageSize)	constructeur qui précise la taille des pages du document
Document(Rectangle pageSize, int marginLeft, int marginRight, int marginTop, int marginBottom)	constructeur qui précise la taille des pages du document et leurs marges

La taille des pages peut être définies en utilisant deux des surcharges du constructeur ou en utilisant la méthode setPageSize().

Un objet de type Rectangle permet de définir la taille des pages du document.

L'unité de mesure dans un document est le point. Il y a 72 points dans un pouce et un pouce vaut 2,54 cm. Ainsi par exemple, la taille d'une page A4 vaut :

largeur : (21 / 2,54) * 72 = 595 points

hauteur : (29,7 / 2,54) * 72 = 842 points

La classe PageSize définit de nombreuses constantes de type Rectangle pour les tailles de pages standards (A0 à A10, LETTER, LEGAL, ...).

Par défaut, la taille utilisée est PageSize.A4.

La plupart des tailles prédéfinies sont au format portrait. Pour utiliser une taille au format paysage, il faut utiliser la méthode rotate() de la classe Rectangle.

Exemple :

```
...
      Document document = new Document(PageSize.A4.rotate());
...
```

La marge par défaut est de 36 points. La marge par défaut peut être précisée dans la surcharge du constructeur dédiée ou en utilisant la méthode setMargins(). Durant la création du contenu d'un document, il est possible d'utiliser la méthode setMargins() pour modifier les marges : cette modification ne sera effective qu'à partir de la page suivante.

La mise en oeuvre d'un document impose quelques contraintes :

- les métadonnées doivent impérativement être associées au document avant l'appel de la méthode open()
- il n'est possible d'ajouter le contenu du document qu'une fois que la méthode open() a été invoquée
- la modification de l'en-tête et du pied page n'est effective qu'à partir de la page suivante

La classe Document possède plusieurs méthodes pour associer des métadonnées au document :

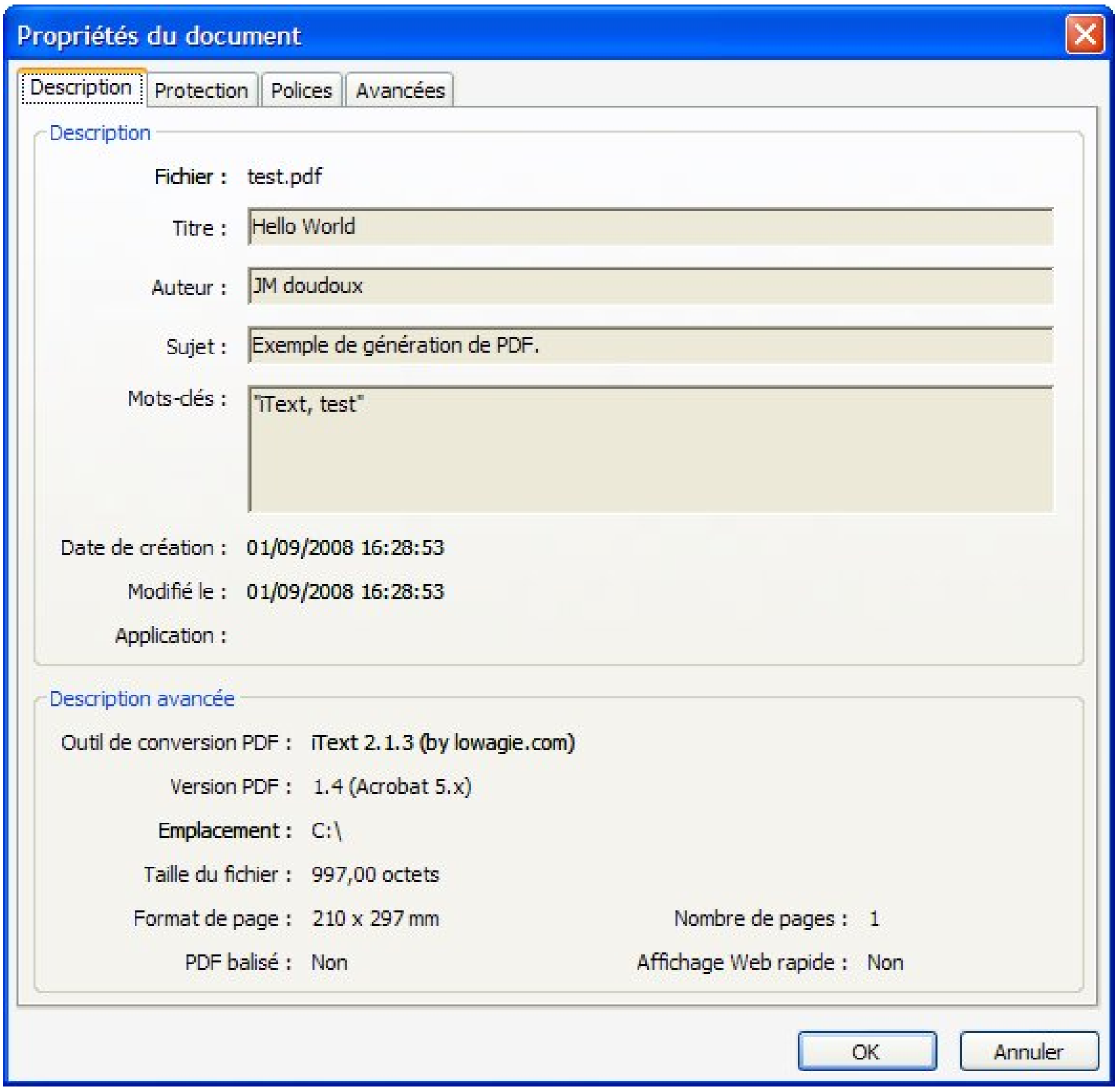
Méthode	Rôle
boolean addTitle(String title)	ajout du titre du document fourni en paramètre
boolean addSubject(String subject)	ajout du sujet du document fourni en paramètre

boolean addKeywords(String keywords)	ajout du mot clé fourni en paramètre
boolean addAuthor(String author)	ajout de l'auteur fourni en paramètre
boolean addCreator(String creator)	ajout du créateur fourni en paramètre
boolean addProducer()	ajout iText comme outil de production du document
boolean addCreationDate()	ajout de la date actuelle comme date de création
boolean addHeader(String name, String content)	ajout d'une métadonnée personnalisée

Remarque : l'utilisation de la méthode addHeader() n'a pas d'effet si le document est exporté en PDF.

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText3 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document,
          new FileOutputStream("c:/test.pdf"));
      document.addTitle("Hello World");
      document.addAuthor("JM doudoux");
      document.addSubject("Exemple de génération de PDF.");
      document.addKeywords("iText, test");
      document.open();
      document.add(new Paragraph("Hello World"));
    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```



Description

Fichier : test.pdf

Titre : Hello World

Auteur : JM doudoux

Sujet : Exemple de génération de PDF.

Mots-clés : "iText, test"

Date de création : 01/09/2008 16:28:53

Modifié le : 01/09/2008 16:28:53

Application :

Description avancée

Outil de conversion PDF : iText 2.1.3 (by lowagie.com)

Version PDF : 1.4 (Acrobat 5.x)

Emplacement : C:\

Taille du fichier : 997,00 octets

Format de page : 210 x 297 mm

Nombre de pages : 1

PDF balisé : Non

Affichage Web rapide : Non

Important : l'ajout de métadonnées doit obligatoirement se faire avant l'appel à la méthode open().

Avant de pouvoir ajouter du contenu au document, il faut obligatoirement invoquer la méthode open() de l'instance de la classe Document. Dans le cas contraire, une exception de type DocumentException est levée.

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText2 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document,
          new FileOutputStream("c:/test.pdf"));
      document.add(new Paragraph("Hello World"));
    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }
    document.close();
  }
}
```

Résultat :

```
com.lowagie.text.DocumentException: The document is not open yet; you can only add Meta
information.
               at com.lowagie.text.Document.add(Unknown Source)
               at fr.jmdoudoux.dej.itext.TestIText2.main(TestIText2.java:20)
Exception in thread "main" java.lang.RuntimeException: The document is not open.
               at com.lowagie.text.pdf.PdfWriter.getDirectContent(Unknown Source)
               at com.lowagie.text.pdf.PdfDocument.newPage(Unknown Source)
               at com.lowagie.text.pdf.PdfDocument.close(Unknown Source)
               at com.lowagie.text.Document.close(Unknown Source)
               at fr.jmdoudoux.dej.itext.TestIText2.main(TestIText2.java:27)
```

La méthode add() permet d'ajouter un élément au contenu du document.

Il est important d'invoquer la méthode close() du document une fois celui-ci complet : la méthode close() va demander la fermeture du ou des flux d'exportation du document.

Attention : la classe Document encapsule le contenu du document mais ne contient aucune information sur son rendu. Le rendu est assuré par différents writers (par exemple un document peut avoir plusieurs pages en PDF mais une seule en HTML) : ainsi il ne faut pas utiliser la méthode getPageNumber() de la classe Document.

110.2.3.2. Les objets de type DocWriter

Pour exporter le document, il faut lui associer un ou plusieurs DocWriters. Chaque DocWriter permet l'exportation du document dans un format particulier.

Trois classes héritent de la classe DocWriter. PdfWriter, HtmlWriter et RtfWriter2 exportent respectivement au format Pdf, Html et Rtf.

Pour obtenir une instance d'un objet héritant du type DocWriter, il faut utiliser sa méthode statique getInstance() qui attend en paramètre l'instance du document et le flux vers lequel le document sera exporté. Ce flux peut être de différents types selon les besoins : FileOutputStream pour un fichier, ServletOutputStream pour une réponse d'une servlet, ByteArrayOutputStream pour stocker le document en mémoire, ...

Il est possible d'affecter plusieurs DocWriter à un document utilisant des flux différents.

Il est nécessaire de conserver l'instance retournée par la méthode getInstance() pour mettre en oeuvre quelques fonctionnalités avancées de l'exportation.

110.2.3.2.1. La classe PdfWriter

La classe PdfWriter permet l'exportation d'un document au format PDF.

Elle propose de nombreuses méthodes permettant de définir des caractéristiques spécifiques au format PDF.

La méthode setViewerPreferences() permet de préciser le mode d'affichage du document par défaut. Elle attend en paramètre un entier pour lequel plusieurs constantes sont définies.

Plusieurs constantes peuvent être combinées pour préciser le mode d'affichage des éléments du panneau de navigation.

Constante	Rôle
PdfWriter.PageModeFullScreen	affichage en plein écran

PdfWriter.PageModeUseAttachments	afficher les pièces jointes
PdfWriter.PageModeUseNone	n'afficher aucun élément du panneau de navigation
PdfWriter.PageModeUseOC	afficher les calques
PdfWriter.PageModeUseOutlines	afficher l'arborescence
PdfWriter.PageModeUseThumbs	affichage des vignettes

Plusieurs autres constantes peuvent être combinées pour préciser le mode d'affichage des pages.

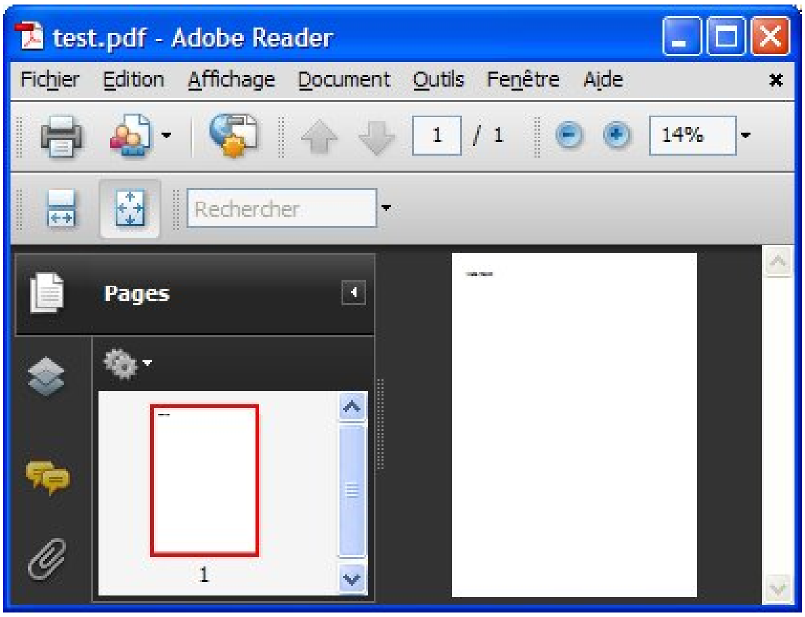
Constante	Rôle
PdfWriter.PageLayoutSinglePage	affichage d'une seule page à la fois
PdfWriter.PageLayoutOneColumn	affichage des pages dans une colonne
PdfWriter.PageLayoutTwoColumnLeft	affichage des pages dans deux colonnes de gauche à droite
PdfWriter.PageLayoutTwoColumnRight	affichage des pages dans deux colonnes (de droite à gauche)

D'autres constantes peuvent être combinées pour afficher ou non quelques éléments de l'interface graphique d'Adobe Reader.

Constante	Rôle
PdfWriter.HideToolBar	permet de spécifier si la barre d'outils est affichée
PdfWriter.HideMenuBar	permet de spécifier si la barre de menu est affichée
PdfWriter.HideWindowUI	permet de spécifier si les contrôles de navigation sont affichés

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText4 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter writer = PdfWriter.getInstance(document,
          new FileOutputStream("c:/test.pdf"));
      writer.setViewerPreferences(PdfWriter.PageLayoutSinglePage
          | PdfWriter.PageModeUseThumbs);

      document.open();
      document.add(new Paragraph("Hello World"));
    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }
    document.close();
  }
}
```



110.2.4. L'ajout de contenu au document

IText propose de nombreuses classes qui encapsulent des éléments qui pourront être ajoutés au contenu d'un document.

Cependant, toutes ces classes ne sont pas supportées par tous les DocWriters : si une classe n'est pas supportée par le DocWriter alors celle-ci est ignorée lors de l'exportation du document qui la contient.

110.2.4.1. Les polices de caractères

Par défaut, un document peut utiliser 14 polices de caractères standards : Courier, Courier Bold, Courier Italic, Courier Bold and Italic, Helvetica, Helvetica Bold, Helvetica Italic, Helvetica Bold and Italic, Times Roman, Times Roman Bold, Times Roman Italic, Times Roman Bold and Italic, Symbol et ZapfDingBats.

Chaque variante de type bold, italic et bold italic pour Courier, Helvetica et Times Roman sont proposées chacune sous la forme d'une police dédiée.

Une police de caractères est encapsulée dans un objet de type Font.

La classe Font encapsule les caractéristiques de la police de caractères : la famille, la taille, le style et la couleur. Elle possède de nombreux constructeurs pour définir ces différentes informations.

Elle propose des constantes pour :

- la famille : COURIER, HELVETICA, SYMBOL, TIMES_ROMAN, ZAPFDINGBATS
- la taille : DEFAULTSIZE
- le style : BOLD, BOLDITALIC, ITALIC, NORMAL, STRIKETHRU, UNDERLINE

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.awt.Color;
import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.Font;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText5 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter writer = PdfWriter.getInstance(document,
          new FileOutputStream("c:/test.pdf"));
      document.open();
      document.add(new Paragraph("Hello World",
            new Font(Font.COURIER, 28, Font.BOLD, Color.RED)));
    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

iText propose une fabrique pour instancier des polices de caractères.

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.awt.Color;
import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.Font;
import com.lowagie.text.FontFactory;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText6 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();
      document.add(new Paragraph("Hello World", FontFactory.getFont(
          FontFactory.COURIER,
            28f,
            Font.BOLD,
            Color.RED)));
    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

Tous les objets retournés par la fabrique héritent de la classe BaseFont. La classe BaseFont propose plusieurs surcharges de la méthode createFont() pour instancier un objet de type Font.

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.awt.Color;
import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.Font;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.pdf.BaseFont;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText7 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      BaseFont fonte = BaseFont.createFont(
          BaseFont.COURIER,
          BaseFont.CP1252,
          BaseFont.NOT_EMBEDDED);
      Font maFonte = new Font(fonte);
      maFonte.setColor(Color.RED);
      maFonte.setStyle(Font.BOLD);
      maFonte.setSize(38.0f);

      document.add(new Paragraph("Hello World", maFonte));
    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

Il est possible d'utiliser n'importe quelle fonte true type présente sur le système.

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.awt.Color;
import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.Font;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.pdf.BaseFont;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText8 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      BaseFont fonte = BaseFont.createFont(
          "C:/Windows/FONTS/ARIAL.TTF",
          BaseFont.CP1252,
          BaseFont.NOT_EMBEDDED);
      Font maFonte = new Font(fonte);
      maFonte.setColor(Color.RED);
      maFonte.setStyle(Font.BOLD);
      maFonte.setSize(38.0f);

      document.add(new Paragraph("Hello World", maFonte));
    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

La méthode createFont() possède plusieurs surcharges. Celle utilisée dans l'exemple attend en paramètre le chemin du fichier qui contient la police true type, l'encodage utilisé (plusieurs constantes sont définies : CP1250, CP1252, CP1257, MACROMAN, WINANSI) et un booléen qui précise si la police doit être incluse dans le PDF (deux constantes sont définies : EMBEDDED et NOT_EMBEDDED).

PDF fourni en standard 3 polices de caractères texte avec les styles normal, gras, italique et gras/italique (Courier, Helvetica et Times) et deux polices de symboles (Symbol et Zapf Dingbats). Il est donc inutile d'inclure ces polices dans le fichier PDF.

Il est possible d'utiliser la méthode register() de la classe FontFactory() pour enregistrer une police True Type en précisant le chemin du fichier de la police en paramètre. Une surcharge de cette méthode attend en plus en paramètre un nom d'alias pour accéder à la police.

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.awt.Color;
import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.Font;
import com.lowagie.text.FontFactory;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.pdf.BaseFont;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText9 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      FontFactory.register("C:/Windows/FONTS/ARIAL.TTF");

      Font fonte = FontFactory.getFont("arial", BaseFont.WINANSI, 38);

      Font maFonte = new Font(fonte);
      maFonte.setColor(Color.RED);
      maFonte.setStyle(Font.BOLD);

      document.add(new Paragraph("Hello World", maFonte));
    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

110.2.4.2. Le classe Chunk

La classe Chunk encapsule une portion de texte du document affiché avec une certaine police de caractères. C'est la plus petite unité de texte utilisable.

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.awt.Color;
import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Chunk;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.Font;
import com.lowagie.text.FontFactory;
import com.lowagie.text.PageSize;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText10 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Chunk chunk = new Chunk("Hello world",
            FontFactory.getFont(FontFactory.COURIER, 20, Font.BOLD, Color.BLUE));

      document.add(chunk);

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

La méthode setUnderline() permet pour les documents PDF d'avoir un contrôle précis sur les caractéristiques du soulignement de la portion de texte. Le premier paramètre précise l'épaisseur du trait et le second précise la position du trait.

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.awt.Color;
import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Chunk;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.Font;
import com.lowagie.text.FontFactory;
import com.lowagie.text.PageSize;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText11 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Chunk chunk = new Chunk("Hello world",
            FontFactory.getFont(FontFactory.COURIER, 20, Font.BOLD, Color.BLUE));
      chunk.setUnderline(0.2f,-2f);
      document.add(chunk);

      chunk = new Chunk("Hello world",
            FontFactory.getFont(FontFactory.COURIER, 20, Font.BOLD, Color.BLUE));
      chunk.setUnderline(2f,5f);
      document.add(chunk);

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

Une surcharge de cette méthode permet de fournir des précisions sur l'apparence du trait.

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.awt.Color;
import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Chunk;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.Font;
import com.lowagie.text.FontFactory;
import com.lowagie.text.PageSize;
import com.lowagie.text.pdf.PdfContentByte;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText12 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Chunk chunk = new Chunk("Hello world",
            FontFactory.getFont(FontFactory.COURIER, 20, Font.BOLD, Color.BLUE));
      chunk.setUnderline(Color.BLUE, 5.0f, 0.0f, 0.0f, -0.2f,
            PdfContentByte.LINE_CAP_ROUND);
      document.add(chunk);

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }
    document.close();
  }
}
```

La méthode setBackground() permet de modifier la couleur de fond de la portion de texte.

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.awt.Color;
import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Chunk;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.Font;
import com.lowagie.text.FontFactory;
import com.lowagie.text.PageSize;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText13 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Chunk chunk = new Chunk("Hello world",
            FontFactory.getFont(FontFactory.COURIER, 20, Font.BOLD, Color.WHITE));
      chunk.setBackground(Color.BLUE);
      document.add(chunk);


    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

110.2.4.3. La classe Phrase

La classe Phrase encapsule une série d'objets de type Chunk qui forme une ou plusieurs lignes dont l'espacement est défini.

Elle possède de nombreux constructeurs.

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Chunk;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.Phrase;
import com.lowagie.text.pdf.PdfWriter;
```

```
public class TestIText13 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Phrase phrase = new Phrase(new Chunk("Hello world "));
      phrase
          .add(new Chunk(
              " test de phrase dont la longueur dépasse largement une seule ligne"));
      phrase.add(new Chunk(" grace à un commentaire assez long"));
      document.add(phrase);

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

La propriété leading de la classe Phrase permet de définir l'espacement entre deux lignes.

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Chunk;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.Phrase;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText14 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Phrase phrase = new Phrase(new Chunk("Hello world "));
      phrase.setLeading(20f);
      phrase
          .add(new Chunk(
              " test de pharse dont la longueur dépasse"+
                          " largement une seule ligne"));
      phrase.add(new Chunk(" grace à un commentaire assez long"));
      document.add(phrase);

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

110.2.4.4. La classe Paragraph

La classe Paragraph encapsule un ensemble d'objets de type Chunk et/ou Phrase pour former un paragraphe. Chacun de ces objets peut avoir des polices de caractères différentes.

Un paragraphe commence systématiquement sur une nouvelle ligne.

La propriété leading permet de préciser l'espacement entre deux lignes.

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText15 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      document.add(new Paragraph("ligne 1"));
      document.add(new Paragraph("ligne 2"));
      document.add(new Paragraph("ligne 3"));

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

La méthode setAlignment() permet de définir l'alignement du paragraphe grâce à plusieurs constantes : Element.ALIGN_LEFT, Element.ALIGN_CENTER, Element.ALIGN_RIGHT et Element.ALIGN_JUSTIFIED

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.Element;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText16 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();
      document.add(new Paragraph("ligne 1"));
      Paragraph paragraph = new Paragraph("ligne 2");
      paragraph.setAlignment(Element.ALIGN_CENTER);
      document.add(paragraph);
      document.add(new Paragraph("ligne 3"));

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

Il est possible de préciser une indentation à gauche et/ou à droite respectivement grâce aux méthodes setIndentationLeft() et setIndentationRight().

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText17 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Paragraph paragraph = new Paragraph(
          "ligne 1 test de phrase dont la longueur dépasse"+
                  " largement une seule ligne grace à un commentaire assez long");
      paragraph.setIndentationLeft(20f);
      document.add(paragraph);
      paragraph = new Paragraph(
          "ligne 2 test de phrase dont la longueur dépasse"+
                  " largement une seule ligne grace à un commentaire assez long");
      paragraph.setIndentationRight(20f);
      document.add(paragraph);

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

Les méthodes setSpacingBefore() et setSpacingAfter() permettent respectivement de préciser l'espace avant et après le paragraphe.

110.2.4.5. La classe Chapter

La classe Chapter encapsule un chapitre. Elle hérite de la classe Section.

Un chapitre commence sur une nouvelle page et possède un numéro affiché par défaut.

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Chapter;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText18 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Chapter chapter = new Chapter(new Paragraph("Premier chapitre"), 1);

      Paragraph paragraph = new Paragraph("ligne 1 test de phrase");
      chapter.add(paragraph);
      paragraph = new Paragraph("ligne 2 test de phrase");
      chapter.add(paragraph);
      document.add(chapter);

      chapter = new Chapter(new Paragraph("Second chapitre"), 1);
      paragraph = new Paragraph("ligne 3 test de phrase");
      chapter.add(paragraph);
      document.add(chapter);

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

Pour ne pas afficher le numéro, il faut invoquer la méthode setNumberDepth() avec la valeur 0 en paramètre.

110.2.4.6. La classe Section

La classe Section encapsule une section qui est un sous-ensemble d'un chapitre.

Pour ajouter une section, il faut utiliser la méthode addSection() qui retourne une instance de la Section.

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.io.FileOutputStream;
import java.io.IOException;
import com.lowagie.text.Chapter;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.Section;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText19 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Chapter chapter = new Chapter(new Paragraph("Mon chapitre"), 1);

      Section section = chapter.addSection(new Paragraph("Premiere section "), 2);
      section.setChapterNumber(1);

      Paragraph paragraph = new Paragraph("ligne 1 test de phrase");
      section.add(paragraph);
      paragraph = new Paragraph("ligne 2 test de phrase");
      section.add(paragraph);

      section = chapter.addSection(new Paragraph("Seconde section "), 2);
      section.setChapterNumber(2);

      paragraph = new Paragraph("ligne 3 test de phrase");
      section.add(paragraph);

      document.add(chapter);

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

La propriété numberDepth permet de préciser la profondeur de la section.

La méthode setChapterNumber() permet de préciser le numéro de la profondeur de la section.

110.2.4.7. La création d'une nouvelle page

Pour créer une nouvelle page, il faut invoquer la méthode newPage() de la classe Document.

Attention, l'appel à la méthode newPage() dans une page vide n'a aucun effet.

Pour créer une page vide, il faut ajouter une ligne

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Chapter;
import com.lowagie.text.Chunk;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.Section;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText21 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Paragraph paragraph = new Paragraph("ligne 1 test de phrase");
      document.add(paragraph);
      document.newPage();
      document.add(Chunk.NEWLINE);
      document.newPage();

      paragraph = new Paragraph("ligne 2 test de phrase");
      document.add(paragraph);
    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

Dans une section, il faut lui ajouter un objet de type Chunk.NEXTPAGE.

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Chapter;
import com.lowagie.text.Chunk;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.Section;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText20 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Chapter chapter = new Chapter(new Paragraph("Mon chapitre"), 1);

      Section section = chapter.addSection(new Paragraph("Premiere section "), 2);
      section.setChapterNumber(1);

      Paragraph paragraph = new Paragraph("ligne 1 test de phrase");
      section.add(paragraph);
      paragraph = new Paragraph("ligne 2 test de phrase");
      section.add(paragraph);
      section.add(Chunk.NEXTPAGE);

      section = chapter.addSection(new Paragraph("Seconde section "), 2);
      section.setChapterNumber(2);
      paragraph = new Paragraph("ligne 3 test de phrase");
      section.add(paragraph);

      document.add(chapter);
    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

110.2.4.8. La classe Anchor

La classe Anchor encapsule un lien hypertexte. Elle hérite de la classe Phrase.

La méthode setReference() permet de préciser l'url externe du lien.

La méthode setName() permet de préciser l'ancre pour un lien interne.

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Anchor;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText22 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Anchor anchor = new Anchor("mon site web");
      anchor.setReference("http://www.jmdoudoux.fr/");

      document.add(anchor);

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

110.2.4.9. Les classes List et ListItem

La classe List encapsule une liste d'éléments de type ListeItem.

La classe Liste possède plusieurs constructeurs. La liste peut être ordonnée ou non selon le premier paramètre fourni au constructeur utilisé : true indique une liste ordonnée.

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.List;
import com.lowagie.text.ListItem;
import com.lowagie.text.PageSize;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText23 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      List liste = new List(true, 20);
      liste.add(new ListItem("Element 1"));
      liste.add(new ListItem("Element 2"));
      liste.add(new ListItem("Element 3"));
      document.add(liste);

      liste = new List(false, 30);
      liste.add(new ListItem("Element 1"));
      liste.add(new ListItem("Element 2"));
      liste.add(new ListItem("Element 3"));
      document.add(liste);

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

La méthode setLettered() permet de préciser si la numérotation d'une liste ordonnée est littérale : la première valeur est dans ce cas A.

La méthode setNumbered() précise si la numérotation d'une liste ordonnée est numérique : la première valeur est dans ce cas 1.

Dans une liste ordonnée, la méthode setFirst() permet d'indiquer la valeur du premier élément pour une numérotation numérique ou littérale.

Dans une liste non ordonnée, la méthode setListeSymbol() permet de préciser le ou les caractères utilisés comme puces.

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.awt.Color;
import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Chunk;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.Font;
import com.lowagie.text.FontFactory;
import com.lowagie.text.List;
import com.lowagie.text.ListItem;
import com.lowagie.text.PageSize;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText24 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      List liste = new List(20);
      liste.setListSymbol(new Chunk("B",
            FontFactory.getFont(FontFactory.ZAPFDINGBATS, 20, Font.BOLD, Color.BLUE)));
      liste.add(new ListItem("Element 1"));
      liste.add(new ListItem("Element 2"));
      liste.add(new ListItem("Element 3"));
      document.add(liste);

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

110.2.4.10. La classe Table

La classe com.lowagie.test.Table encapsule un tableau utilisé comme une matrice. Chaque cellule est encapsulée dans un objet de type Cell.

Il est possible de définir le nombre de lignes et de colonnes en utilisant la surcharge du constructeur adéquat.

Il est impératif de définir le nombre de colonnes ; le nombre de lignes peut croître selon les besoins.

La méthode addCell() permet de fournir la valeur de la cellule courante. Par défaut, c'est la cellule de la première ligne, première colonne. L'appel à la méthode déplace la cellule courante dans la même ligne sur la colonne suivante si elle existe sinon sur la première colonne de la ligne suivante.

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.Table;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText25 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Table tableau = new Table(2,2);
      tableau.addCell("1.1");
      tableau.addCell("1.2");
      tableau.addCell("2.1");
      tableau.addCell("2.2");

      document.add(tableau);


    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

La classe Table possède une surcharge de la méthode addCell() qui attend en second paramètre un objet de type Point qui permet d'indiquer une cellule bien précise dans le tableau.

La méthode setAutoFillEmptyCell() attend un booléen qui permet de préciser si les cellules non renseignées doivent être automatiquement créées vides.

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.awt.Point;
import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.Table;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText26 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Table tableau = new Table(2,2);
      tableau.addCell("1.0", new Point(1,0));
      tableau.addCell("2.1", new Point(2,1));

      document.add(tableau);


    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

La classe Table possède de nombreuses méthodes pour modifier son rendu par exemple : setBorderWidth(), setBorderColor(), setBackGroundColor(), setPadding(), ...

Pour obtenir plus de souplesse dans le rendu d'une cellule, il est possible d'instancier une occurrence de la classe Cell et d'invoquer les méthodes qu'elle propose pour configurer son apparence.

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.awt.Color;
import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Cell;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.Element;
import com.lowagie.text.PageSize;
import com.lowagie.text.Table;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText27 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Table tableau = new Table(2,2);
      tableau.setAutoFillEmptyCells(true);
      tableau.setPadding(2);

      Cell cell = new Cell("1.1");
      cell.setHorizontalAlignment(Element.ALIGN_CENTER);
      cell.setBackgroundColor(Color.YELLOW);
      tableau.addCell(cell);

      tableau.addCell("1.2");
      tableau.addCell("2.1");
      tableau.addCell("2.2");

      document.add(tableau);

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

Il est possible de définir une en-tête pour les colonnes en ajoutant au début des cellules un appel à la méthode setHeader() avec le paramètre true. Une fois toutes les en-têtes définies, il faut invoquer la méthode endHeaders() de la classe Table.

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Cell;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.Element;
import com.lowagie.text.PageSize;
import com.lowagie.text.Table;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText28 {

  public static void main(String[] args) {
    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Table tableau = new Table(2, 2);
      tableau.setAutoFillEmptyCells(true);
      tableau.setPadding(2);

      Cell cell = new Cell("colonne 1");
      cell.setHeader(true);
      cell.setHorizontalAlignment(Element.ALIGN_CENTER);
      tableau.addCell(cell);

      cell = new Cell("colonne 2");
      cell.setHeader(true);
      cell.setHorizontalAlignment(Element.ALIGN_CENTER);
      tableau.addCell(cell);
      tableau.endHeaders();

      cell = new Cell("1.1");
      cell.setHorizontalAlignment(Element.ALIGN_CENTER);
      tableau.addCell(cell);

      tableau.addCell("1.2");
      tableau.addCell("2.1");
      tableau.addCell("2.2");

      document.add(tableau);

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

110.2.5. Des fonctionnalités avancées

iText propose de nombreuses fonctionnalités avancées pour générer un document.

110.2.5.1. Insérer une image

iText propose le support de plusieurs formats d'images : JPEG, GIF, PNG, BMP, TIFF, WMF et les objets de type java.awt.Image.

La classe Image est une classe abstraite dont hérite chaque classe qui encapsule un type d'images supporté par iText.

La méthode getInstance() de la classe Image permet d'obtenir une instance d'une image. De nombreuses surcharges sont proposées pour fournir par exemple un chemin sur le système de fichiers ou une url.

Exemple :

```
package fr.jmdoudoux.dej.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.Image;
import com.lowagie.text.PageSize;
import com.lowagie.text.pdf.PdfWriter;
```

```
public class TestIText29 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Image image = Image.getInstance("c:/monimage.jpg");
      document.add(image);

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

La méthode setAlignment() peut être utilisée pour préciser l'alignement de l'image en passant en paramètre une des constantes : LEFT, MIDDLE ou RIGHT

La classe Image propose plusieurs méthodes pour permettre de redimentionner l'image : scaleAbsolute(), scaleAbsoluteWidth(), scaleAbsoluteHeight(), scalePercent() et scaleToFit().

Les méthodes setRotation() et setRotationDegrees() permettent de faire une rotation de l'image.

111. La validation des données

Chapitre 111

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

La validation des données est une tâche commune, nécessaire et importante dans chaque application. De plus, ces validations peuvent être faites dans les différentes couches d'une application :

- Présentation
- Service
- Métier
- DAO
- Dans la base de données par des contraintes d'intégrités

Certains frameworks notamment pour les couches présentation et DAO proposent leurs propres solutions de validations de données. Pour les autres couches, soit un autre framework soit une solution maison sont utilisées. Toutes ces solutions proposent des implémentations différentes pour déclarer et valider des contraintes mais aussi pour signaler les violations de contraintes (Exception, objets dédiés, ...).

Ceci entraîne fréquemment une duplication du code et/ou une redondance des contrôles effectués avec les risques que cela peut engendrer :

- temps requis
- source d'erreurs
- difficultés de maintenance

Il y a aussi le risque d'oublier la déclaration de contraintes dans une couche.

Une solution est de mettre ces traitements de validation dans les entités du domaine ce qui les complexifie.

De plus, certaines de ces validations sont fréquemment utilisées et sont donc standard (vérifier la présence d'une valeur, vérifier une taille, vérifier la valeur sur une plage de dates ou une plage numérique, vérifier la valeur sur une expression régulière, ...).

Il est aussi généralement nécessaire de développer des validations spécifiques.

Pour répondre à ces différents besoins, des frameworks ont été développés pour :

- fournir des validateurs classiques
- permettre de définir ses propres validateurs
- faciliter l'application de ces validateurs sur des données.

Ce chapitre contient plusieurs sections :

- ♦ Quelques recommandations sur la validation des données
- ♦ L'API Bean Validation (JSR 303)
- ♦ D'autres frameworks pour la validation des données

111.1. Quelques recommandations sur la validation des données

Les validations de données sont utiles dans plusieurs endroits d'une application et ce quel que soit le type d'applications :

- couche présentation : pour informer le plus rapidement possible l'utilisateur de la saisie d'une donnée erronée, généralement ce sont des contrôles de surface
- couche service : validation des données reçues et qui n'ont pas été forcément validées par une IHM
- couche métier : validation des données traitées qui peuvent nécessiter un accès aux données
- couche accès aux données (DAO) : validation des données avant leur envoi dans la base de données

Il est important de valider les contraintes le plus tôt possible dans les couches de l'application pour éviter des appels inutiles, fréquemment au travers du réseau, aux fonctionnalités de la couche en amont.

Il est aussi très important de répéter les validations de données dans les couches sous-jacentes car il ne faut pas présumer de ce que fait la couche en amont. Par exemple, même si les données dont validées dans l'IHM d'une application invoquant un service, il faut revalider ces données dans la couche service car si le service est invoqué par une application qui ne fait pas le contrôle, les données risquent d'être non valides. Même si cela augmente les traitements cela rend les applications plus sûres.

Dans tous les cas, les contrôles doivent être faits dans la couche la plus basse possible, ce qui comprend aussi les contrôles d'intégrité dans la base de données.

Il existe des contrôles spécifiques à la couche présentation : par exemple, la double saisie d'un mot de passe et la comparaison des deux valeurs saisies.

Il existe une frontière très mince entre les règles métiers, les traitements métiers et la validation des données. Il peut être tentant de mettre certaines de ces fonctionnalités dans la validation des données mais il ne faut pas tout mettre dans la validation et maintenir un rôle à la couche métier. Les traitements de validation des données doivent rester simples et ne pas devenir trop complexes ni nécessiter plusieurs entités (propriétés, objets, ressources, ...).

111.2. L'API Bean Validation (JSR 303)

L'API Bean Validation est issue des travaux de la JSR 303 : https://jcp.org/en/jsr/detail?id=303

Cette JSR 303 propose de standardiser un framework de validation des données d'un bean.

L'intérêt de cette API est de proposer une approche cohérente sous la forme d'un standard pour la validation des données d'un bean.

Il y a besoin d'un standard pour plusieurs raisons :

- il existe déjà plusieurs frameworks open source de validation de données
- les validations se font dans toutes les couches, pas toujours de façon cohérente et fréquemment par duplication de code
- plusieurs technologies des plates-formes Java ont besoin d'un framework de validation : JPA, JSF, ...

Généralement ces validations ont lieu avec plus ou moins de redondance dans les différentes couches d'une application.

Fréquemment, les contraintes sont exprimées sur les entités du domaine ainsi la JSR propose de déclarer les contraintes dans les beans qui encapsulent les entités du domaine.

Inclure ces validations dans les entités du domaine permet de centraliser ces traitements plutôt que de les dupliquer ou les répartir dans les différentes couches.

111.2.1. La présentation de l'API

L'API Bean Validation standardise la définition, la déclaration et la validation de contraintes sur les données d'un ou plusieurs beans.

La déclaration de contraintes se fait dans le bean qui encapsule les données. L'expression de ces contraintes se fait à l'aide d'annotations ou d'un descripteur au format XML ce qui permet de réduire la quantité de code à produire. La manière privilégiée pour déclarer les contraintes est d'utiliser les annotations mais il est aussi possible d'utiliser un descripteur au format XML.

L'API propose une ensemble de contraintes communes fournies en standard et permet de définir ses propres contraintes.

La validation de ces contraintes se fait grâce à un valideur fourni par l'API.

Elle propose aussi des fonctionnalités avancées comme la composition de contraintes, la validation partielle en utilisant la notion de groupes de contraintes, la définition de contraintes personnalisées et la recherche des contraintes définies.

111.2.1.1. Les objectifs de l'API

La JSR 303 tente de combiner les meilleures fonctionnalités de différents frameworks dans une spécification qui peut être implémentée par différents fournisseurs et qui propose :

- de fournir un ensemble de contraintes standard
- de déclarer les contraintes sans avoir à écrire de code explicitement
- de valider un objet par rapport à ses contraintes et à ses valeurs grâce à un moteur de validation
- de définir des contraintes personnalisées pour rendre le framework extensible
- de fournir une API de recherche des contraintes sur un type

Le but de cette JSR est de standardiser les fonctionnalités de validation des données des beans en utilisant des annotations plutôt que d'avoir à écrire du code pour réaliser ces validations.

Il ne s'agit pas de fournir une solution permettant de définir des contraintes dans toutes les couches de l'application (notamment elle ne couvre pas directement les contraintes dans la base de données car celles-ci sont spécifiques) mais de proposer des contraintes au niveau des entités du domaine. Ce choix repose sur le fait que ces contraintes sont généralement liées à l'entité elle-même.

La JSR 303 a plusieurs objectifs :

- Proposer une spécification relative à la validation de données dans les applications Java
- Fournir une API qui soit indépendante d'une architecture
- Etre utilisable dans toutes les couches Java d'une application : elle est utilisable côté client ou serveur
- Standardiser la déclaration des contraintes en privilégiant les annotations au niveau de la classe (généralement un bean qui encapsule une entité du domaine)
- Définir des contraintes communes fournies en standard
- Fournir un mécanisme standard pour valider les contraintes
- Etre facile à utiliser et extensible
- Fournir une API qui permette de rechercher les contraintes exploitables notamment par des frameworks

111.2.1.2. Les éléments et concepts utilisés par l'API

L'API Java Bean Validation utilise plusieurs éléments et concepts lors de sa mise en oeuvre :

- une contrainte est une restriction appliquée sur la valeur d'un champ ou d'une propriété d'une instance d'un bean.
- la déclaration d'une contrainte assigne une contrainte à un bean, un champ ou une propriété en utilisant une annotation ou grâce à un fichier XML.
- une implémentation de l'interface ConstraintValidator encapsule les traitements de validation d'une donnée ou du bean.

La définition d'une contrainte se fait par une annotation en précisant le type sur lequel elle s'applique, ses attributs et la classe qui encapsule les traitements de validation.

Les groupes permettent de n'appliquer qu'un sous-ensemble des contraintes d'un bean. La déclaration d'une contrainte peut être associée à un ou plusieurs groupes.

La validation d'une contrainte applique les traitements de validation d'une données sur l'instance courante.

Les spécifications doivent être implémentées par un fournisseur pour pouvoir être utilisées. Le projet Hibernate Validator est l'implémentation de référence de ces spécifications.

L'interpolation du message contient les traitements pour créer le message d'erreur fourni à l'utilisateur.

Une séquence permet de définir l'ordre dans lequel les contraintes de validation vont être évaluées.

Les spécifications proposent une API qui permet d'obtenir des métadonnées sur les contraintes d'un type. Cette API est particulièrement utile pour l'intégration dans d'autres frameworks.

Les spécifications proposent aussi une API nommée BootStrap qui fournit des mécanismes pour obtenir une instance de la fabrique de type ValidatorFactory. Cette API permet notamment de choisir l'implémentation à utiliser.

111.2.1.3. Les contraintes et leur validation avec l'API

Une contrainte est composée de deux éléments :

- une annotation : utilisée par le développeur pour déclarer ses contraintes
- une classe de type Validator : utilisée par l'API pour valider les données selon les annotations utilisées

La JSR-303 définit un ensemble de contraintes que chaque implémentation doit fournir. Cependant, cet ensemble ne concerne que des contraintes standard qui ne sont en général pas suffisantes pour répondre à tous les besoins. La spécification prévoit donc la possibilité de développer ses propres contraintes personnalisées.

Ceci peut se faire de deux façons :

- par combinaison d'autres contraintes sous la forme d'une composition de contraintes qui est un ensemble de contraintes utilisées comme une seule
- par la définition de ses propres contraintes

La validation des contraintes se fait par introspection à la recherche des annotations du type des contraintes utilisées dans le bean. Pour chaque annotation, la classe de type Validator associée est instanciée et utilisée par le framework pour valider la valeur de la donnée.

L'API peut prendre en charge, à la demande lors de la validation du bean, le parcours des objets dépendant de ce bean pour les valider également si des contraintes leur sont associées.

La validation des données peut être invoquée automatiquement par les frameworks qui proposent un support pour l'API Bean Validator : c'est notamment le cas pour JSF 2.0 et JPA 2.0.

111.2.1.4. La mise en oeuvre générale de l'API

La JSR 303 propose de standardiser les validations avec les spécifications d'une API composée de plusieurs parties :

- des annotations sur les entités du domaine ce qui permet de centraliser ces validations sans alourdir les beans avec beaucoup de code
- une API pour valider les contraintes
- une API dédiée à l'obtention des métadonnées des contraintes

La plupart des frameworks de validations sont relatifs à un framework particulier pour une ou deux couches données :

Struts, Hibernate, ... L'API Bean Validator est conçue pour être utilisée dans toutes les couches écrites en Java d'une application.

L'API Bean Validation est incluse dans Java EE 6 car elle est utilisée par JSF 2.0 et JPA 2.0. L'API peut cependant être utilisée dans Java SE à partir de la version 5.

L'API Bean Validation est conçue pour être indépendante de la technologie qui l'utilise aussi bien côté client (Swing, ...) que serveur (JPA, JSF, ...).

Le package de cette API est javax.validation.

L'implémentation de référence est proposée par Hibernate Validator 4.

Pour mettre en oeuvre l'API, il n'est pas nécessaire d'utiliser des classes de l'implémentation : seules les classes et interfaces de l'API doivent être importées dans le code source. Ceci rend l'utilisation d'une autre implémentation très facile.

Il est nécessaire d'ajouter au classpath les dépendances de l'implémentation utilisée : par exemple, avec l'implémentation de référence.

111.2.1.5. Un exemple simple de mise en oeuvre

Cet exemple va définir un bean, ajouter une contrainte de type non null sur un champ et créer une petite application de test qui va instancier le bean avec un champ null et appliquer les validations des contraintes sur le bean.

La JSR 303 permet d'annoter une classe ou un attribut d'une classe ou le getter de cet attribut.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import java.util.Date;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Past;
import javax.validation.constraints.Size;

public class PersonneBean {

  private String nom;
  private String prenom;
  private Date dateNaissance;

  public PersonneBean(String nom, String prenom, Date dateNaissance) {
    super();
    this.nom = nom;
    prenom = prenom;
    this.dateNaissance = dateNaissance;
  }

  @NotNull
  @Size(max=50)
  public String getNom() {
    return nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

  @NotNull
  @Size(max=50)
  public String getPrenom() {
    return prenom;
  }

  public void setPrenom(String prenom) {
    prenom = prenom;
  }

  @Past
  public Date getDateNaissance() {
    return dateNaissance;
  }

  public void setDateNaissance(Date dateNaissance) {
    this.dateNaissance = dateNaissance;
  }
}
```

L'API propose aussi un mécanisme pour valider les contraintes et exploiter les éventuelles violations.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidation {

  public static void main(String[] args) {

    PersonneBean personne = new PersonneBean(null, null, new GregorianCalendar(
        2065, Calendar.JANUARY, 18).getTime());

    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();

    Set<ConstraintViolation<PersonneBean>> constraintViolations =
      validator.validate(personne);

    if (constraintViolations.size() > 0 ) {
      System.out.println("Impossible de valider les donnees du bean : ");
      for (ConstraintViolation<PersonneBean> contraintes : constraintViolations) {
        System.out.println(contraintes.getRootBeanClass().getSimpleName()+
           "." + contraintes.getPropertyPath() + " " + contraintes.getMessage());
      }
    } else {
      System.out.println("Les donnees du bean sont valides");
    }
  }
}
```

Résultat :

```
Impossible de valider les donnees du bean :
PersonneBean.dateNaissance doit être dans le passé
PersonneBean.nom ne peut pas être nul
PersonneBean.prenom ne peut pas être nul
```

111.2.2. La déclaration des contraintes

La déclaration de contraintes se fait dans des classes ou des interfaces avec des annotations ce qui est la manière recommandée ou par une description dans un fichier XML.

Une contrainte peut être appliquée sur un type (classe ou interface), un champ ou une propriété respectant les conventions des Java beans.

Remarque : les champs statiques ne peuvent pas être validés en utilisant l'API.

La valeur fournie à l'objet de type ConstraintValidator qui va valider les contraintes dépend de l'entité annotée avec la contrainte :

- si la contrainte est définie sur la classe ou une interface de la classe alors c'est l'instance de classe qui est fournie comme valeur
- si la contrainte est définie sur un champ alors c'est la valeur du champ qui est fournie
- si la contrainte est définie sur le getter d'une propriété alors c'est la valeur de retour de ce getter qui est fournie

Remarque : il faut définir les contraintes soit sur le champ soit sur la propriété correspondante mais pas sur les deux à la fois sinon la validation se fera deux fois. Il est préférable de rester consistant et d'utiliser les annotations toujours sur les champs ou toujours sur les getter.

Chaque déclaration d'une contrainte peut redéfinir le message fourni en cas de violation.

111.2.2.1. La déclaration des contraintes sur les champs

L'application de contraintes sur un champ permet de réaliser la validation de la donnée par l'implémentation de l'API de façon indépendante de la forme d'accès à ce champ.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import java.util.Date;

import javax.validation.constraints.NotNull;

public class PersonneBean {

  @NotNull
  private String nom;
  @NotNull
  private String prenom;
  @Past
  private Date dateNaissance;

  public PersonneBean(String nom, String prenom, Date dateNaissance) {
    super();
    this.nom = nom;
    prenom = prenom;
    this.dateNaissance = dateNaissance;
  }

  public String getNom() {
    return nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

  public String getPrenom() {
    return prenom;
  }

  public void setPrenom(String prenom) {
    prenom = prenom;
  }

  public Date getDateNaissance() {
    return dateNaissance;
  }

  public void setDateNaissance(Date dateNaissance) {
    this.dateNaissance = dateNaissance;
  }
}
```

L'application de ces contraintes peut se faire sur un champ quel que soit sa visibilité (private, protected ou public) mais ne peut pas se faire sur un champ static.

111.2.2.2. La déclaration des contraintes sur les propriétés

Il est possible de définir les contraintes sur une propriété : dans ce cas, seul le getter doit être annoté.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import java.util.Date;

import javax.validation.constraints.NotNull;

public class PersonneBean {

  private String nom;
  private String Prenom;
  private Date dateNaissance;

  public PersonneBean(String nom, String prenom, Date dateNaissance) {
    super();
    this.nom = nom;
    Prenom = prenom;
    this.dateNaissance = dateNaissance;
  }

  @NotNull
  public String getNom() {
    return nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

  @NotNull
  public String getPrenom() {
    return Prenom;
  }

  public void setPrenom(String prenom) {
    Prenom = prenom;
  }

  @Past
  public Date getDateNaissance() {
    return dateNaissance;
  }

  public void setDateNaissance(Date dateNaissance) {
    this.dateNaissance = dateNaissance;
  }
}
```

La validation de la donnée par l'implémentation de l'API utilise alors obligatoirement le getter pour obtenir la valeur de la donnée.

111.2.2.3. La déclaration des contraintes sur une classe

La déclaration d'une contrainte peut être faite sur une classe ou une interface. Dans ce cas, la validation se fait sur l'état de la classe ou de la classe qui implémente l'interface.

C'est l'instance de la classe qui sera fournie comme valeur à valider au ConstraintValidator.

Une telle validation peut être requise si elle nécessite l'état de plusieurs données de la classe pour être réalisée.

111.2.2.4. L'héritage de contraintes

Lorsqu'un bean hérite d'un autre bean qui contient une définition de contraintes, celles-ci sont héritées.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import java.util.Date;

import javax.validation.constraints.Min;

public class DeveloppeurSeniorBean extends PersonneBean {

  private int experience;

  public DeveloppeurSeniorBean(String nom, String prenom, Date dateNaissance, int experience) {
    super(nom, prenom, dateNaissance);
    this.experience = experience;
  }

  @Min(value=5)
  public int getExperience() {
    return experience;
  }

  public void setExperience(int experience) {
    this.experience = experience;
  }
}
```

Lors de la validation du bean, les contraintes du bean sont vérifiées mais aussi celles de la classe mère.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidation {

  public static void main(String[] args) {
    DeveloppeurSeniorBean personne = new DeveloppeurSeniorBean(null, "", new GregorianCalendar(
        1965, Calendar.JANUARY, 18).getTime(), 3);

    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();

    Set<ConstraintViolation<DeveloppeurSeniorBean>> constraintViolations =
      validator.validate(personne);
    if (constraintViolations.size() > 0 ) {
      System.out.println("Impossible de valider les donnees du bean : ");
      for (ConstraintViolation<DeveloppeurSeniorBean> contraintes : constraintViolations) {
        System.out.println(contraintes.getRootBeanClass().getSimpleName()+
          "." + contraintes.getPropertyPath() + " " + contraintes.getMessage());
      }
    } else {
      System.out.println("Les donnees du bean sont valides");
    }
  }
}
```

Résultat :

```
Impossible de valider les donnees du bean :
DeveloppeurSeniorBean.experience doit être plus grand que 5
DeveloppeurSeniorBean.nom ne peut pas être nul
```

Les contraintes sont héritées d'une classe mère mais elles peuvent être redéfinies. Si une méthode est redéfinie, les contraintes de la méthode de la classe mère s'appliquent aussi sauf si une contrainte existante est aussi redéfinie.

111.2.2.5. Les contraintes de validation d'un ensemble d'objets

L'API propose une validation d'un objet mais permet aussi la validation d'un graphe d'objets composé de l'objet et de tout ou partie de ses objets dépendants.

L'annotation @Valid utilisée sur une dépendance d'un bean permet de demander au moteur de validation de valider aussi la dépendance lors de la validation du bean.

Ce mécanisme est récursif : une dépendance annotée avec @Valid peut elle-même contenir des dépendances annotées avec @Valid. Ainsi, l'ensemble des beans dépendants qui seront validés en même temps que le bean est défini en utilisant l'annotation @Valid sur chacune des dépendances concernées.

Une dépendance annotée avec @Valid est ignorée par le moteur si sa valeur est nulle.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import javax.validation.Valid;
import javax.validation.constraints.NotNull;

public class Comite {

  @NotNull
  @Valid
  private PersonneBean president;

  @Valid
  private PersonneBean tresorier;

  @Valid
  private PersonneBean secretaire;

  public Comite(PersonneBean president, PersonneBean tresorier,
      PersonneBean secretaire) {
    super();
    this.president = president;
    this.tresorier = tresorier;
    this.secretaire = secretaire;
  }

  public PersonneBean getPresident() {
    return president;
  }
  public PersonneBean getTresorier() {
    return tresorier;
  }

  public PersonneBean getSecretaire() {
    return secretaire;
  }
}
```

La validation du bean échoue si la validation d'une de ses dépendances échoue.

La dépendance peut aussi être une collection typée de beans. Cette collection peut être :

- un tableau
- une implémentation de java.lang.Iterable (collection, List, Set, ...)
- une implémentation de java.util.Map

Exemple :

```
package fr.jmdoudoux.dej.validation;

import java.util.ArrayList;
import java.util.List;

import javax.validation.Valid;
import javax.validation.constraints.NotNull;

public class Groupe {

  @NotNull
  private String nom;

  private List<PersonneBean> membres = new ArrayList<PersonneBean>();

  public Groupe(String nom) {
    super();
    this.nom = nom;
    membres = new ArrayList<PersonneBean>();
  }

  public String getNom() {
    return nom;
  }

  @NotNull
  @Valid
  public List<PersonneBean> getMembres() {
    return membres;
  }

  public void ajouter(PersonneBean personne) {
    membres.add(personne);
  }

  public void supprimer(PersonneBean personne) {
    membres.remove(personne);
  }
}
```

Si une telle collection est marquée avec l'annotation @Valid, alors toutes les occurrences de la collection seront validées lorsque le bean qui encapsule la collection sera validé.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidationGroupe {

  public static void main(String[] args) {


    Groupe groupe = new Groupe("Mon groupe");

    PersonneBean personne = new PersonneBean(null, null, new GregorianCalendar(
        2065, Calendar.JANUARY, 18).getTime());

    groupe.ajouter(personne);

    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();

    Set<ConstraintViolation<Groupe>> constraintViolations =
      validator.validate(groupe);

    if (constraintViolations.size() > 0 ) {
      System.out.println("Impossible de valider les donnees du bean : ");
      for (ConstraintViolation<Groupe> contraintes : constraintViolations) {
        System.out.println(contraintes.getRootBeanClass().getSimpleName()+
          "." + contraintes.getPropertyPath() + " " + contraintes.getMessage());
      }
    } else {
      System.out.println("Les donnees du groupe sont valides");
    }

  }
}
```

Résultat :

```
Impossible de valider les donnees du bean :
Groupe.membres[0].nom ne peut pas être nul
Groupe.membres[0].dateNaissance doit être dans le passé
Groupe.membres[0].prenom ne peut pas être nul
```

Les occurrences null dans une collection sont ignorées lors de la validation.

Dans le cas d'une collection de type Map, seules les valeurs sont validées (Map.Entry) : les clés ne le sont pas.

Lors de la validation, l'annotation @Valid est traitée récursivement dans les dépendances tant que cela ne provoque pas une boucle infinie : le moteur de validation doit ignorer une instance qui a déjà été validée lors du traitement d'un même graphe d'objets.

111.2.3. La validation des contraintes

Bean Validation propose une API pour permettre la validation des contraintes sur les données de façon indépendante de la couche dans laquelle elle est mise en oeuvre.

L'interface Validator définit les fonctionnalités d'un valideur.

Pour valider les contraintes sur les données d'un bean, il faut obtenir une instance de l'interface Validator, utiliser cette instance pour valider les données d'un bean. Les éventuelles erreurs détectées par cette validation sont retournées sous la forme d'un Set d'objets de type ConstraintViolation.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidation {

  public static void main(String[] args) {

    PersonneBean personne = new PersonneBean("nom1", "prenom1", new GregorianCalendar(
        1965, Calendar.JANUARY, 18).getTime());

    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();

    Set<ConstraintViolation<PersonneBean>> constraintViolations =
      validator.validate(personne);

    if (constraintViolations.size() > 0 ) {
      System.out.println("Impossible de valider les donnees du bean : ");
      for (ConstraintViolation<PersonneBean> contraintes : constraintViolations) {
        System.out.println(contraintes.getRootBeanClass().getSimpleName()+
          "." + contraintes.getPropertyPath() + " " + contraintes.getMessage());
      }
    } else {
      System.out.println("Les donnees du bean sont valides");
    }
  }
}
```

111.2.3.1. L'obtention d'un valideur

Pour obtenir une instance de Validator fournie par une implémentation de l'API, il faut utiliser une fabrique de type ValidatorFactory.

Le plus simple pour obtenir une instance de cette fabrique est d'utiliser la méthode statique buildDefaultValidatorFactory() de la classe Validation.

Il est alors possible d'utiliser la méthode getValidator() de la fabrique pour obtenir une instance de type Validator.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidation {

  public static void main(String[] args) {
...

    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();

...
  }
}
```

111.2.3.2. L'interface Validator

L'interface javax.Validation.Validator est l'élément principal de l'API de validation des contraintes.

L'interface Validator propose des méthodes pour demander la validation de données notamment :

Méthode	Rôle
Set<ConstraintViolation<T>> validate(T, Class< ?>...)	Demander la validation des données d'un bean et éventuellement de ses dépendances
Set<ConstraintViolation<T>> validateProperty(T, String, Class< ?>...)	Demander la validation de la valeur d'une propriété d'un bean. Cette méthode est utile pour la validation partielle d'un bean
Set<ConstraintViolation<T>> validateValue(T, String, Object, Class< ?>...)	Demander la validation d'une valeur par rapport à une propriété particulière d'un bean

Si la collection est vide, c'est que la validation a réussi sinon la validation a échoué et la collection contient alors la ou les raisons de l'échec sous la forme d'une occurrence pour chaque contrainte qui n'a pas été validée.

Toutes les méthodes attendent aussi un paramètre de type varargs qui peut être utilisé pour préciser les groupes à valider. Si aucun groupe n'est précisé, c'est le groupe par défaut (javax.validation.Default) qui est utilisé.

111.2.3.3. L'utilisation d'un valideur

La méthode validate() permet de demander la validation des données d'un bean et éventuellement de ses dépendances.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidationGroupe {

  public static void main(String[] args) {

    Groupe groupe = new Groupe("Mon groupe");

    PersonneBean personne = new PersonneBean(null, null, new GregorianCalendar(
        2065, Calendar.JANUARY, 18).getTime());

    groupe.ajouter(personne);

    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();

    Set<ConstraintViolation<Groupe>> constraintViolations =
      validator.validate(groupe);

    if (constraintViolations.size() > 0 ) {
      System.out.println("Impossible de valider les donnees du bean : ");
      for (ConstraintViolation<Groupe> contraintes : constraintViolations) {
        System.out.println(contraintes.getRootBeanClass().getSimpleName()+
          "." + contraintes.getPropertyPath() + " " + contraintes.getMessage());
      }
    } else {
      System.out.println("Les donnees du groupe sont valides");
    }

  }
}
```

La méthode validateProperty() permet de valider la valeur d'une propriété d'un bean.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidationProperty {

  public static void main(String[] args) {

    MonBean monBean = new MonBean(new GregorianCalendar(1980,
                                                        Calendar.DECEMBER,
                                                        25).getTime());

    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();

    Set<ConstraintViolation<MonBean>> constraintViolations =
      validator.validateProperty(monBean,
                                 "maValeur");
    validator.validate(monBean);

    if (constraintViolations.size() > 0) {
      System.out.println("Impossible de valider les donnees du bean : ");
      for (ConstraintViolation<MonBean> contraintes : constraintViolations) {
        System.out.println("  "
            + contraintes.getRootBeanClass().getSimpleName() + "."
            + contraintes.getPropertyPath() + " " + contraintes.getMessage());
      }
    } else {
      System.out.println("Les donnees du bean sont validees");
    }
  }
}
```

La méthode validateValue() permet de valider la valeur d'une propriété particulière d'un bean.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;
import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidationValue {

  public static void main(String[] args) {
    Date valeur = new GregorianCalendar(1980, Calendar.DECEMBER, 25).getTime();

    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();

    Set<ConstraintViolation<MonBean>> constraintViolations =
      validator.validateValue(MonBean.class,
                              "maValeur",
                              valeur);
    if (constraintViolations.size() > 0) {
      System.out.println("Impossible de valider la valeur de la donnee du bean : ");
      for (ConstraintViolation<MonBean> contraintes : constraintViolations) {
        System.out.println("  "
            + contraintes.getRootBeanClass().getSimpleName() + "."
            + contraintes.getPropertyPath() + " " + contraintes.getMessage());
      }
    } else {
      System.out.println("La valeur de la donnees du bean est validee");
    }
  }
}
```

Remarque : la validation des dépendances déclarées avec l'annotation @Valid n'est effective qu'avec la méthode validate().

111.2.3.4. L'interface ConstraintViolation

L'interface ConstraintViolation<T> encapsule les informations relatives à l'échec de la validation d'une contrainte.

Elle propose plusieurs méthodes pour obtenir ces données :

Méthode	Rôle
String getMessage()	Renvoie le message d'erreur interpolé
String getMessageTemplate()	Renvoie le message d'erreur non interpolé (généralement la valeur de l'attribut message de la contrainte)
T getRootBean()	Renvoie le bean racine qui a été validé (c'est l'objet qui a été passé en paramètre de la méthode validate() de la classe Validator)
Class<T> getRootBeanClass()	Renvoie la classe du bean racine qui a été validé
Object getLeafBean()	Renvoie l'objet sur lequel la contrainte est appliquée
Object getInvalidValue()	Renvoie la valeur qui a fait échouer la contrainte (la valeur passée en paramètre de la méthode isValid() de la classe ConstraintValidator)
ConstraintDescriptor<?> getConstraintDesrcriptor()	Renvoie un objet qui encapsule la contrainte

111.2.3.5. La mise en oeuvre des groupes

Comme les contraintes sont définies au niveau des entités du domaine et que les validations peuvent se faire dans toutes les couches de l'application, il faut que les contraintes puissent s'appliquer partout.

Ce n'est pas toujours le cas : c'est possible pour des contrôles de surface mais pour des contraintes plus compliquées ce n'est pas toujours réalisable (par exemple si un accès à la base de données est nécessaire, ...).

De plus, toutes les contraintes d'un bean ne peuvent pas être validée en même temps. Par exemple, un bean qui encapsule les données d'un assistant comportant plusieurs pages. Pour valider les données de la première page avant de passer à la seconde, une validation de l'intégralité des contraintes du bean n'est pas possible puisque les données des autres pages ne sont pas encore renseignées.

Enfin, certaines contraintes ne peuvent être réalisées dans toutes les couches car elles sont trop coûteuses par exemple en ressources ou en temps de traitement.

L'API Bean Validation résoud ces problématiques au travers de la notion de groupes qui contiennent les contraintes à valider.

Les groupes (groups) permettent de restreindre l'ensemble des contraintes qui seront testées durant une validation.

Les groupes sont des types (interfaces ou classes) ce qui permet un typage fort, de faire de l'héritage, de les documenter avec Javadoc et autorise le refactoring grâce à un IDE. Il est possible de définir une hiérarchie de groupes, le plus simple étant d'utiliser une interface de type marqueur.

Exemple :

```
package fr.jmdoudoux.dej.validation;

public interface AssistantEtape1 {

}

package fr.jmdoudoux.dej.validation;

public interface AssistantEtape2 {

}

package fr.jmdoudoux.dej.validation;

public interface AssistantEtape3 {

}
```

La notion de groupe permet de donner une flexibilité à la validation en proposant d'indiquer quelles contraintes doivent être vérifiées lors de la validation. Ainsi, le ou les groupes à valider sont précisés au moment de la demande de validation des contraintes du bean.

L'attribut groups de l'annotation d'une contrainte permet de faire une validation partielle du bean : elle précise le ou les groupes qui sont concernés lors d'une validation.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import javax.validation.constraints.NotNull;

public class DonneesAssistantBean {

  /**
   *  donnees saisies à l'étape 1 de l'assitant
   */
  private String donnees1;

  /**
   *  donnees saisies à l'étape 2 de l'assitant
   */
  private String donnees2;

  /**
   *  donnees saisie à l'étape 3 de l'assitant
   */
  private String donnees3;

  @NotNull(groups={AssistantEtape1.class, AssistantEtape2.class, AssistantEtape3.class})
  public String getDonnees1() {
    return donnees1;
  }

  public void setDonnees1(String donnees1) {
    this.donnees1 = donnees1;
  }

  @NotNull(groups={AssistantEtape2.class, AssistantEtape3.class})
  public String getDonnees2() {
    return donnees2;
  }

  public void setDonnees2(String donnees2) {
    this.donnees2 = donnees2;
  }

  @NotNull(groups={AssistantEtape3.class})
  public String getDonnees3() {
    return donnees3;
  }

  public void setDonnees3(String donnees3) {
    this.donnees3 = donnees3;
  }
}
```

Les groupes qui doivent être utilisés lors de la validation sont précisés grâce au paramètre parameter de type varargs des méthodes validate(), validateProperty() et validateValue() de la classe Validator.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidationDonneesAssistantBean {

  public static void main(String[] args) {
    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();
    DonneesAssistantBean donnees = new DonneesAssistantBean();

    donnees.setDonnees1("valeur donnees1");
    System.out.println("Validation des données de l'étape 1");
    validerDonnees(validator, donnees, AssistantEtape1.class);

    donnees.setDonnees2("valeur donnees2");
    System.out.println("Validation des données de l'étape 2");
    validerDonnees(validator, donnees, AssistantEtape2.class);

    donnees.setDonnees3("valeur donnees3");
    System.out.println("Validation des données de l'étape 3");
    validerDonnees(validator, donnees, AssistantEtape3.class);
  }

  private static void validerDonnees(Validator validator,
                                     DonneesAssistantBean donnees,
                                     Class<?>... groupes) {
    Set<ConstraintViolation<DonneesAssistantBean>> constraintViolations;
    constraintViolations = validator.validate(donnees, groupes);

    if (constraintViolations.size() > 0) {
      System.out.println("Impossible de valider les donnees du bean : ");
      for (ConstraintViolation<DonneesAssistantBean> contrainte : constraintViolations) {
        System.out.println("  " + contrainte.getRootBeanClass().getSimpleName()
            + "." + contrainte.getPropertyPath() + " "
            + contrainte.getMessage());
      }
    } else {
      System.out.println("Les donnees du bean sont validees");
    }
  }
}
```

Chaque contrainte qui n'a pas de groupe explicite est associée au groupe par défaut (javax.validation.Default).

Il est aussi possible d'utiliser les groupes pour les évaluer un par un en conditionnant l'évaluation du suivant au succès de l'évaluation du précédent.

111.2.3.6. Définir et utiliser un groupe implicite

Il est possible d'associer plusieurs contraintes à un groupe sans avoir à déclarer le groupe explicitement dans la déclaration de chaque contrainte.

Chaque contrainte du groupe par défaut contenue dans une interface I est automatiquement associée au groupe I.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import java.util.Date;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Past;

public interface Tracabilite {
  @NotNull
  @Past
  Date getDateCreation();

  @NotNull
  @Past
  Date getDateModif();

  @NotNull
  Long getUtilisateur();

}
```

Exemple :

```
package fr.jmdoudoux.dej.validation;

import java.util.Date;

import javax.validation.constraints.NotNull;

public class Operation implements Tracabilite {
  private Date dateCreation;
  private Date dateModification;
  private Long utilisateur;
  private String designation;

  public Operation(Date dateCreation, Date dateModification, Long utilisateur,
      String designation) {
    super();
    this.dateCreation = dateCreation;
    this.dateModification = dateModification;
    this.utilisateur = utilisateur;
    this.designation = designation;
  }

  @NotNull
  public String getDesignation() {
    return this.designation;
  }

  @Override
  public Date getDateCreation() {
    return this.dateCreation;
  }

  @Override
  public Date getDateModif() {
    return this.dateModification;
  }

  @Override
  public Long getUtilisateur() {
    return utilisateur;
  }
}
```

Ceci est pratique pour permettre la validation partielle d'un bean basée sur les fonctionnalités définies dans une interface.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import java.util.Date;
import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidationOperation {

  public static void main(String[] args) {

    Operation operation = new Operation(new Date(), new Date(), 1234l, null);

    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();

    System.out.println("Validation sur le groupe par defaut");
    Set<ConstraintViolation<Operation>> constraintViolations =
      validator.validate(operation);


    if (constraintViolations.size() > 0 ) {
      System.out.println("Impossible de valider les donnees du bean : ");
      for (ConstraintViolation<Operation> contraintes : constraintViolations) {
        System.out.println(contraintes.getRootBeanClass().getSimpleName()+
          "." + contraintes.getPropertyPath() + " " + contraintes.getMessage());
      }
    } else {
      System.out.println("Les donnees du groupe sont valides");
    }

    constraintViolations = validator.validate(operation, Tracabilite.class);

    System.out.println("Validation sur le groupe Tracabilite : ");

    if (constraintViolations.size() > 0 ) {
      System.out.println("Impossible de valider les donnees du bean : ");
      for (ConstraintViolation<Operation> contraintes : constraintViolations) {
        System.out.println(contraintes.getRootBeanClass().getSimpleName()+
          "." + contraintes.getPropertyPath() + " " + contraintes.getMessage());
      }
    } else {
      System.out.println("Les donnees du groupe sont valides");
    }
  }
}
```

Résultat :

```
Validation sur le groupe par defaut
```

```
Impossible de valider les donnees du bean :
Operation.designation ne peut pas être nul
Validation sur le groupe Tracabilite :
Les donnees du groupe sont valides
```

111.2.3.7. La définition de l'ordre des validations

Par défaut, une donnée est validée sans tenir compte d'un ordre vis-à-vis des groupes auxquelles la contrainte est associée.

Il peut cependant être utile de vouloir contrôler l'ordre d'évaluation des contraintes : par exemple s'il est utile de voir évaluer certaines contraintes avant d'autres.

Pour définir cet ordre particulier dans la validation des groupes, il faut créer un groupe qui va définir une séquence ordonnée d'autres groupes. La définition de cette séquence ce fait avec l'annotation @GroupSequence

Exemple :

```
package fr.jmdoudoux.dej.validation;

import javax.validation.GroupSequence;

@GroupSequence( { MaContrainte1.class, MaContrainte2.class, MaContrainte2.class })
public @interface MonGroupeDeSequence {
}
```

Lors de l'évaluation des groupes de la séquence, dès que la validation d'un groupe échoue, les autres groupes de la séquence ne sont pas évalués.

Il faut faire attention de ne pas créer une dépendance cyclique entre la définition d'une séquence et les groupes qui composent cette séquence aussi bien directement qu'indirectement sinon une exception de type GroupDefinitionException est levée.

L'interface d'un groupe de séquences ne devra pas avoir de superinterface.

111.2.3.8. La redéfinition du groupe par défaut

L'annotation @GroupSequence sert aussi à redéfinir le groupe par défaut d'une classe. Il suffit de l'utiliser sur une classe pour remplacer le groupe par défaut (Default.class).

Comme les séquences ne peuvent pas avoir de dépendances circulaires, il n'est pas possible d'inclure le groupe Default dans une séquence.

Par contre, comme les contraintes d'une classe sont associées automatiquement au groupe, il faut obligatoirement ajouter le groupe (la classe elle-même) dans la séquence car les contraintes contenues dans la classe doivent être incluses dans la séquence qui redéfinit le groupe par défaut. Si ce n'est pas le cas, une exception de type GroupDefinitionException est levée lors de la validation de la classe ou de la recherche des contraintes qu'elle contient.

111.2.4. Les contraintes standard

Les spécifications de la JSR 303 définissent un petit ensemble de contraintes que chaque implémentation doit fournir. Celles-ci peuvent être utilisées telles quelles ou dans une composition.

Il est aussi possible pour une implémentation de fournir d'autres contraintes.

La JSR 303 propose en standard plusieurs annotations pour des actions de validations communes.

Annotation	Rôle
@Null	Vérifier que la valeur du type concerné soit null
@NotNull	Vérifier que la valeur du type concerné soit non null
@AssertTrue	Vérifier que la valeur soit true
@AssertFalse	Vérifier que la valeur soit false
@DecimalMin	Vérifier que la valeur soit supérieure ou égale à celle fournie sous la forme d'une chaîne de caractères encapsulant un BigDecimal
@DecimalMax	Vérifier que la valeur soit inférieure ou égale à celle fournie sous la forme d'une chaîne de caractères encapsulant un BigDecimal
@Digits	Vérifier qu'un nombre n'a pas plus de chiffres avant et après la virgule que ceux précisés en paramètre
@Size	Vérifier que la taille de la donnée soit comprise en les valeurs min et max fournies
@Min	Vérifier que la valeur du type soit un nombre entier dont la valeur doit être supérieure ou égale à la valeur fournie en paramètre
@Max	Vérifier que la valeur du type soit un nombre entier dont la valeur doit être inférieure ou égale à la valeur fournie en paramètre
@Pattern	Vérifier la conformité d'une chaîne de caractères avec une expression régulière
@Valid	Demander la validation des objets dépendant de l'objet à valider
@Future	Vérifier que la date soit dans le futur (postérieure à la date courante)
@Past	Vérifier que la date soit dans le passé (antérieure à la date courante)

Ces contraintes sont dans le package javax.validation.constraints.

Les exemples des sections suivantes vont utiliser la classe ci-dessous pour valider les données du bean d'exemple.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidationMonBean {

  public static void main(String[] args) {

    MonBean monBean = new MonBean("test");

    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();

    Set<ConstraintViolation<MonBean>> constraintViolations =
      validator.validate(monBean);

    if (constraintViolations.size() > 0 ) {
      System.out.println("Impossible de valider les donnees du bean : ");
      for (ConstraintViolation<MonBean> contraintes : constraintViolations) {
        System.out.println("  "+contraintes.getRootBeanClass().getSimpleName()+
          "." + contraintes.getPropertyPath() + " " + contraintes.getMessage());
      }
    } else {
      System.out.println("Les donnees du bean sont validees");
    }
  }
}
```

111.2.4.1. L'annotation @Null

Cette contrainte impose que la valeur du type concerné soit null. Elle peut s'appliquer sur n'importe quel type d'objet.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import javax.validation.constraints.Null;

public class MonBean {

  @Null
  private String maValeur;

  public MonBean(String maValeur) {
    super();
    this.maValeur = maValeur;
  }

  public String getMaValeur() {
    return maValeur;
  }

  public void setMaValeur(String maValeur) {
    this.maValeur = maValeur;
  }
}
```

111.2.4.2. L'annotation @NotNull

Cette contrainte impose que la valeur du type concerné ne soit pas null. Elle peut s'appliquer sur n'importe quel type d'objet.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import javax.validation.constraints.NotNull;

public class MonBean {

  @NotNull
  private String maValeur;

  public MonBean(String maValeur) {
    super();
    this.maValeur = maValeur;
  }

  public String getMaValeur() {
    return maValeur;
  }

  public void setMaValeur(String maValeur) {
    this.maValeur = maValeur;
  }
}
```

111.2.4.3. L'annotation @AssertTrue

Cette contrainte impose que la valeur du type concerné soit true ou null (la donnée est valide si sa valeur est null).

Exemple :

```
package fr.jmdoudoux.dej.validation;

import javax.validation.constraints.AssertTrue;

public class MonBean {

  @AssertTrue
  private boolean maValeur;

  public MonBean(boolean maValeur) {
    super();
    this.maValeur = maValeur;
  }

  public boolean getMaValeur() {
    return maValeur;
  }

  public void setMaValeur(boolean maValeur) {
    this.maValeur = maValeur;
  }
}
```

Elle ne peut s'appliquer que sur un type booléen (Boolean et boolean) sinon une exception de type UnexpectedTypeException est levée à la validation ou lors de la recherche des métadonnées.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import javax.validation.constraints.AssertTrue;

public class MonBean {

  @AssertTrue
  private String maValeur;

  public MonBean(String maValeur) {
    super();
    this.maValeur = maValeur;
  }

  public String getMaValeur() {
    return maValeur;
  }

  public void setMaValeur(String maValeur) {
    this.maValeur = maValeur;
  }
}
```

Résultat :

```
Exception in thread "main" javax.validation.UnexpectedTypeException: No validator could
be found for type: java.lang.String
        at org.hibernate.validator.engine.ConstraintTree.verifyResolveWasUnique(ConstraintTree.
java:236)
        at org.hibernate.validator.engine.ConstraintTree.findMatchingValidatorClass(ConstraintT
ree.java:219)
        at org.hibernate.validator.engine.ConstraintTree.getInitializedValidator(ConstraintTree
.java:167)
        at org.hibernate.validator.engine.ConstraintTree.validateConstraints(ConstraintTree.jav
a:113)
        at org.hibernate.validator.metadata.MetaConstraint.validateConstraint(MetaConstraint.ja
va:121)
        at org.hibernate.validator.engine.ValidatorImpl.validateConstraint(ValidatorImpl.java:3
34)
        at org.hibernate.validator.engine.ValidatorImpl.validateConstraintsForRedefinedDefaultG
roup(ValidatorImpl.java:278)
        at org.hibernate.validator.engine.ValidatorImpl.validateConstraintsForCurrentGroup(Vali
datorImpl.java:260)
```

```
        at org.hibernate.validator.engine.ValidatorImpl.validateInContext(ValidatorImpl.java:21
3)
        at org.hibernate.validator.engine.ValidatorImpl.validate(ValidatorImpl.java:119)
        at fr.jmdoudoux.dej.validation.TestValidationMonBean.main(TestValidationMonBean.java:
20)
```

111.2.4.4. L'annotation @AssertFalse

Cette contrainte impose que la valeur du type concerné soit false ou null (la donnée est valide si sa valeur est null).

Exemple :

```
package fr.jmdoudoux.dej.validation;

import javax.validation.constraints.AssertTrue;

public class MonBean {

  @AssertFalse
  private boolean maValeur;

  public MonBean(boolean maValeur) {
    super();
    this.maValeur = maValeur;
  }

  public boolean getMaValeur() {
    return maValeur;
  }

  public void setMaValeur(boolean maValeur) {
    this.maValeur = maValeur;
  }
}
```

Elle ne peut s'appliquer que sur un type booléen (Boolean et boolean) sinon une exception de type UnexpectedTypeException est levée à la validation ou lors de la recherche des métadonnées.

111.2.4.5. L'annotation @Min

Cette contrainte impose que la valeur du type soit un nombre dont la valeur doit être supérieure ou égale à la valeur fournie en paramètre sous la forme d'un entier de type long. La donnée est valide si sa valeur est null.

Elle ne peut s'appliquer que sur les types : BigDecimal, BigInteger, byte, short, int, long et leurs wrappers respectifs. Le fournisseur n'a pas l'obligation de proposer une implémentation du valideur de la contrainte pour les types double et float.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import javax.validation.constraints.Min;

public class MonBean {

  @Min(value=10)
  private int maValeur;

  public MonBean(int maValeur) {
    super();
    this.maValeur = maValeur;
  }

  public int getMaValeur() {
    return maValeur;
  }

  public void setMaValeur(int maValeur) {
    this.maValeur = maValeur;
  }
}
```

111.2.4.6. L'annotation @Max

Cette contrainte impose que la valeur du type soit un nombre dont la valeur doit être inférieure ou égale à la valeur fournie en paramètre sous la forme d'un entier de type long. La donnée est valide si sa valeur est null.

Elle ne peut s'appliquer que sur les types : BigDecimal, BigInteger, byte, short, int, long et leurs wrappers respectifs. Le fournisseur n'a pas l'obligation de proposer une implémentation du valideur de la contrainte pour les types double et float.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import javax.validation.constraints.Min;

public class MonBean {

  @Max(value=20)
  private int maValeur;

  public MonBean(int maValeur) {
    super();
    this.maValeur = maValeur;
  }

  public int getMaValeur() {
    return maValeur;
  }

  public void setMaValeur(int maValeur) {
    this.maValeur = maValeur;
  }
}
```

111.2.4.7. L'annotation @DecimalMin

Cette contrainte impose que la valeur du type soit un nombre dont la valeur doit être supérieure ou égale à la valeur fournie en paramètre sous la forme d'une chaîne de caractères qui puisse être transformée en BigDecimal. La donnée est valide si sa valeur est null.

Elle ne peut s'appliquer que sur les types : BigDecimal, BigInteger, String, byte, short, int, long et leurs wrappers respectifs. Les types double et float ne sont pas obligatoirement supportés à cause des problèmes d'arrondis mais une implémentation peut proposer une solution par approximation de la valeur selon des règles qui lui sont propres.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import javax.validation.constraints.DecimalMin;

public class MonBean {

  @DecimalMin(value="10.5")
  private int maValeur;

  public MonBean(int maValeur) {
    super();
    this.maValeur = maValeur;
  }

  public int getMaValeur() {
    return maValeur;
  }

  public void setMaValeur(int maValeur) {
    this.maValeur = maValeur;
  }
}
```

Si le type de données est String alors la valeur contenue doit pouvoir être convertie en BigDecimal sinon la validation échoue.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidationMonBean {

  public static void main(String[] args) {

    MonBean monBean = new MonBean("test");

...

  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.validation;

import javax.validation.constraints.DecimalMin;

public class MonBean {

  @DecimalMin(value="10.5")
  private String maValeur;

  public MonBean(String maValeur) {
    super();
    this.maValeur = maValeur;
  }

  public String getMaValeur() {
    return maValeur;
  }

  public void setMaValeur(String maValeur) {
    this.maValeur = maValeur;
  }
}
```

Résultat :

```
Impossible de valider les donnees du bean :
  MonBean.maValeur doit être plus grand que 10.5
```

111.2.4.8. L'annotation @DecimalMax

Cette contrainte impose que la valeur du type soit un nombre dont la valeur doit être supérieure ou égale à la valeur fournie en paramètre sous la forme d'une chaîne de caractères qui puisse être transformée en BigDecimal. La donnée est valide si sa valeur est null.

Elle ne peut s'appliquer que sur les types : BigDecimal, BigInteger, String, byte, short, int, long et leurs wrappers respectifs. Les types double et float ne sont pas obligatoirement supportés à cause des problèmes d'arrondis mais une implémentation peut proposer une solution par approximation de la valeur selon des règles qui lui sont propres.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import javax.validation.constraints.DecimalMax;

public class MonBean {

  @DecimalMin(value="99.9")
  private int maValeur;

  public MonBean(int maValeur) {
    super();
    this.maValeur = maValeur;
  }

  public int getMaValeur() {
    return maValeur;
  }

  public void setMaValeur(int maValeur) {
    this.maValeur = maValeur;
  }
}
```

Si le type de données est String alors la valeur contenue doit pouvoir être convertie en BigDecimal sinon la validation échoue.

111.2.4.9. L'annotation @Size

Cette contrainte impose que la taille du type soit un nombre dont la valeur doit être comprise entre les valeurs de type int fournies aux attributs min (valeur par défaut 0) et max (valeur par défaut Integer.MAX_VALUE) incluses.

Les types supportés sont :

- String : c'est la taille de la chaîne qui est évaluée
- Tableau : c'est la taille du tableau qui est évaluée
- Collection : c'est la taille de la collection qui est évaluée
- Map : c'est la taille de la Map qui est évaluée

La donnée est valide si sa valeur est null.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import javax.validation.constraints.Size;

public class MonBean {

  @Size(min=10, max=20)
  private String maValeur;

  public MonBean(String maValeur) {
    super();
    this.maValeur = maValeur;
  }

  public String getMaValeur() {
    return maValeur;
  }
  public void setMaValeur(String maValeur) {
    this.maValeur = maValeur;
  }
}
```

111.2.4.10. L'annotation @Digits

L'élément annoté doît être la représentation d'un nombre dont la partie entière et la mantisse ne dépassent pas le nombre maximum de chiffres imposé par les attributs integer et fraction.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import javax.validation.constraints.Digits;

public class MonBean {

  @Digits(integer=5, fraction=2)
  private String maValeur;

  public MonBean(String maValeur) {
    super();
    this.maValeur = maValeur;
  }

  public String getMaValeur() {
    return maValeur;
  }

  public void setMaValeur(String maValeur) {
    this.maValeur = maValeur;
  }
}
```

111.2.4.11. L'annotation @Past

Cette contrainte impose que la date vérifiée soit dans le passé.

La date actuelle est celle de la JVM. Le calendrier utilisé est celui correspondant au TimeZone et à la Locale courante.

Les types de données utilisables avec cette annotation sont Date et Calendar.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import java.util.Date;

import javax.validation.constraints.Past;

public class MonBean {

  @Past
  private Date maValeur;

  public MonBean(Date maValeur) {
    super();
    this.maValeur = maValeur;
  }

  public Date getMaValeur() {
    return maValeur;
  }

  public void setMaValeur(Date maValeur) {
    this.maValeur = maValeur;
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.validation;

import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidationMonBean {

  public static void main(String[] args) {
    MonBean monBean = new MonBean(new GregorianCalendar(1980,
      Calendar.DECEMBER, 25).getTime());
...

  }
}
```

111.2.4.12. L'annotation @Future

Cette contrainte impose que la date vérifiée soit dans le futur.

La date actuelle est celle de la JVM. Le calendrier utilisé est celui correspondant au TimeZone et à la Locale courante.

Les types de données utilisables avec cette annotation sont Date et Calendar.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import java.util.Date;

import javax.validation.constraints.Past;

public class MonBean {

  @Futur
  private Date maValeur;

  public MonBean(Date maValeur) {
    super();
    this.maValeur = maValeur;
  }

  public Date getMaValeur() {
    return maValeur;
  }

  public void setMaValeur(Date maValeur) {
    this.maValeur = maValeur;
  }
}
```

111.2.4.13. L'annotation @Pattern

Cette contrainte permet de valider une valeur par rapport à une expression régulière. La donnée est valide si sa valeur est null. Le format de l'expression régulière est celui utilisé par la classe java.util.regex.Pattern.

Elle ne peut s'appliquer que sur une donnée de type String.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import java.util.Date;
import javax.validation.constraints.Pattern;

public class UtilisateurBean extends PersonneBean {

  private String digiCode;

  public UtilisateurBean(String nom, String prenom, Date dateNaissance, String digiCode) {
    super(nom, prenom, dateNaissance);
    this.digiCode = digiCode;
  }

  @Pattern(regexp="\\d\\d\\d[A-F]",
    message="Le digicode doit contenir 3 chiffres et une lettre entre A et F")
  public String getDigiCode() {
    return digiCode;
  }

  public void setDigiCode(String digiCode) {
    this.digiCode = digiCode;
  }
}
```

L'attribut regex permet de préciser l'expression régulière sur laquelle la donnée sera validée.

L'attribut flags est un tableau de l'énumération Flag qui précise les options à utiliser par la classe Pattern. Les valeurs de l'énumération sont : UNIX_LINES, CASE_INSENSITIVE, COMMENTS, MULTILINE, DOTALL, UNICODE_CASE et CANON_EQ.

111.2.5. Le développement de contraintes personnalisées

L'API Bean Validation propose des contraintes standard mais celles-ci ne peuvent pas répondre à tous les besoins en particulier pour des contraintes spécifiques. L'API propose donc de pouvoir développer et utiliser ses propres contraintes personnalisées.

La création d'une contrainte requiert plusieurs étapes :

- créer l'annotation pour la contrainte
- implémenter la contrainte sous la forme d'un Validator
- définir le message d'erreur par défaut

111.2.5.1. La création de l'annotation

Une annotation est considérée comme une contrainte de validation si elle est annotée avec l'annotation javax.validation.Constraint et si sa retention policy est RUNTIME.

L'annotation est définie comme n'importe quelle annotation en utilisant l'annotation @interface et un définissant une méthode pour chaque attribut.

L'annotation de la contrainte doit être annotée avec des méta-annotations comme pour la définition de toutes annotations :

- @Target({METHOD, FIELD, ANNOTATION_TYPE }) : permet de préciser le type d'entité sur lequel l'annotation peut être utilisée
- @Retention(RUNTIME) : permet de préciser comment sera exploitée l'annotation. Dans le cas d'une contrainte de l'API Bean Validation, il faut obligatoirement utiliser la valeur RUNTIME pour permettre à l'API d'utiliser l'introspection à l'exécution

- @Constraint(validatedBy = CheckCaseValidator.class): permet de préciser la ou les classes de type Validator qui encapsulent les traitements de validation grâce à l'attribut validatedBy
- @Documented : Permet de préciser que l'utilisation de cette annotation sera incluse dans la Javadoc

L'utilisation des 3 premières méta-annotations est obligatoire selon les spécifications de l'API Java Bean Validation.

Exemple :

```
@java.lang.annotation.Documented
@ConstraintValidator(value = CarteBleueValidator.class)
@java.lang.annotation.Target(value = {java.lang.annotation.ElementType.FIELD})
@java.lang.annotation.Retention(value = java.lang.annotation.RetentionPolicy.RUNTIME)
public @interface CarteBleue
{
    String message() default "";
    String[] groups() default {};
    String bankName() default "";
}
```

Les annotations standards @Target et @Retention permettent respectivement de préciser le type sur lequel l'annotation peut s'appliquer et la portée d'application de l'annotation qui doit obligatoirement être RUNTIME pour permettre à l'API de fonctionner à l'exécution.

Une annotation relative à une contrainte doit obligatoirement être annotée avec l'annotation @Constraint. Son attribut validatedBy permet de préciser la ou les classes de type ConstraintValidator qui lui sont associées et qui contiennent les traitements de validation à instancier.

La spécification de l'API Bean Validation impose que chaque annotation d'une contrainte définisse obligatoirement trois attributs :

- message : préciser le message d'erreur en cas de violation de la contrainte (type String)
- groups : définir le ou les groupes de validation auxquels la contrainte appartient. La valeur par défaut doit être un tableau vide de type Class<?>
- payload : fournir des données complémentaires généralement utilisées lors de l'exploitation des violations de contraintes

L'attribut message de type String permet de créer le message qui indiquera pourquoi la validation a échouée.

Il est préférable d'utiliser un ResourceBundle pour stocker les messages. Dans ce cas, la valeur de l'attribut message doit contenir la clé entourée d'accolades. Par convention, le nom de la clé doit être composé du nom pleinement qualifié de la classe concaténé avec .message.

Exemple :

```
String message() default "{com.acme.constraint.MyConstraint.message}";
```

L'attribut groups de type Class< ?>[] permet de définir les groupes de contraintes qui seront utilisés lors de la validation. La valeur par défaut doit être un tableau vide : dans ce cas c'est le groupe par défaut qui est utilisé.

Exemple :

```
    Class<?>[] groups() default {};
```

Les groupes ont deux utilités principales :

- réaliser une validation partielle en précisant quelles seront les contraintes à utiliser
- définir l'ordre dans lequel les contraintes seront validées

L'attribut payload de type Class< ? extends Payload>[] permet de déclarer des types qui seront associés à la contrainte. La valeur par défaut est un tableau vide.

Exemple :

```
    Class<? extends Payload>[] payload() default {};
```

Chaque classe qui est fournie en tant que payload doit implémenter l'interface Payload. Ces données sont typiquement non portables. L'utilisation d'un type permet un typage fort de l'information. Un exemple d'utilisation de ces données peut être un niveau de gravité qui permettra à la couche présentation de préciser la sévérité de la contrainte violée, chaque gravité étant représentée dans sa propre classe.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import javax.validation.Payload;

public class Gravite {
    public static class Info implements Payload {};
    public static class Attention implements Payload {};
    public static class Erreur implements Payload {};
}
```

Il suffit alors de préciser la ou les classes comme valeur de l'attribut payload

Exemple :

```
package fr.jmdoudoux.dej.validation;

import javax.validation.constraints.NotNull;

public class DonneesBean {

  private String valeur1;
  private String valeur2;

  public DonneesBean(String valeur1, String valeur2) {
    super();
    this.valeur1 = valeur1;
    this.valeur2 = valeur2;
  }

  @NotNull(message="La saisie de la valeur est obligatoire", payload=Gravite.Erreur.class)
  public String getValeur1() {
    return valeur1;
  }

  public void setValeur1(String valeur1) {
    this.valeur1 = valeur1;
  }

  @NotNull(message="La saisie de la valeur est recommandée", payload=Gravite.Info.class)
  public String getValeur2() {
    return valeur2;
  }

  public void setValeur2(String valeur2) {
    this.valeur2 = valeur2;
  }
}
```

Ces classes peuvent être retrouvées dans un objet de type ConstraintDescriptor encapsulé dans les objets de type ConstraintViolation.

Exemple :

```
package fr.jmdoudoux.dej.validation;
```

```
import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Payload;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidationDonneesBean {

  public static void main(String[] args) {
    DonneesBean donneesBean = new DonneesBean(null, null);

    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();

    Set<ConstraintViolation<DonneesBean>> constraintViolations =
      validator.validate(donneesBean);

    if (constraintViolations.size() > 0 ) {
      System.out.println("Impossible de valider les donnees du bean : ");
      for (ConstraintViolation<DonneesBean> contrainte : constraintViolations) {
        String severite = "";
        for (Class<? extends Payload> gravite : contrainte.getConstraintDescriptor()
          .getPayload()) {
          severite = gravite.getSimpleName();
          break;
        }

        System.out.println(severite + "\t "+contrainte.getRootBeanClass().getSimpleName()+
          "." + contrainte.getPropertyPath() + " " + contrainte.getMessage());
      }
    } else {
      System.out.println("Les donnees du bean sont validees");
    }
  }
}
```

Résultat :

```
Impossible de valider les donnees du bean :
Info     DonneesBean.valeur2 La saisie de la valeur est recommandée
Erreur   DonneesBean.valeur1 La saisie de la valeur est obligatoire
```

Il est possible de définir des attributs spécifiques aux besoins de la contrainte.

Le nom des attributs de l'annotation d'une contrainte est soumis à des restrictions :

- les noms message, groups et payload sont réservés et ne peuvent donc pas être utilisés pour des attributs personnalisés
- les noms ne peuvent pas commencer par valid

Exemple : la définition d'une contrainte avec un attribut avec une valeur par défaut

```
package fr.jmdoudoux.dej.validation;

import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;

@Target( { METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
```

```
@Constraint(validatedBy = CasseValidator.class)
@Documented
public @interface Casse {

    String message() default "La casse de la donnée est erronée";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    boolean majuscule() default false;
}
```

Exemple : définition d'une contrainte avec un attribut obligatoire

```
package fr.jmdoudoux.dej.validation;

import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;

@Target( { METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = CasseValidator.class)
@Documented
public @interface Casse {

    String message() default "La casse de la donnée est erronée";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    boolean majuscule();
}
```

111.2.5.2. La création de la classe de validation

Chaque contrainte doit être associée avec au moins une classe qui encapsule la logique de validation de la valeur de la donnée associée à la contrainte. Cette association est précisée grâce à l'attribut validatedBy de l'annotation @Constraint que chaque annotation d'une contrainte utilise.

Cette classe doit implémenter l'interface ConstraintValidator qui requiert deux types paramétrés avec des generics :

- Le premier type est l'interface de l'annotation qui est utilisée pour invoquer le valideur
- Le second précise le type de la valeur qui sera validée par la contrainte

Si une annotation peut être utilisée sur différents types d'éléments alors il faut créer une implémentation de type ConstraintValidator pour chacun de ces types puisque le type de la donnée à valider est fourni en tant que paramètre générique.

Cette interface définit deux méthodes :

- void initialize(A constraintAnnotation)
- boolean isValid(T value, ConstraintValidatorContext context)

La méthode initialize() est invoquée une fois que le validateur est instancié : elle permet de l'initialiser. Elle reçoit en paramètre l'annotation de la contrainte ce qui permet notamment d'extraire les valeurs des attributs à utiliser pour la validation. L'implémentation doit garantir que cette méthode est invoquée avant toute utilisation de la contrainte.

La méthode isValid() contient les traitements de validation de la valeur de la donnée. Le paramètre value contient la valeur de l'objet à valider. Le paramètre context encapsule les informations sur le contexte dans lequel la validation se fait. Le code de cette méthode doit être thread-safe (elle doit obligatoirement fonctionner dans un environnement multithread) et ne doit pas modifier la valeur de l'objet fourni en paramètre. Elle renvoie un booléen qui précise si la validation a réussie ou non.

Si une exception est levée dans les méthodes initialize() ou isValid() alors celle-ci est propagée sous la forme d'une exception de type ValidationException.

La spécification recommande comme une bonne pratique dans le traitement de validation de considérer la valeur null comme valide. Ceci permet de ne pas faire double emploi avec la contrainte @NotNull qui doit être utilisée si la valeur est invalide lorsqu'elle est null.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class CasseValidator implements ConstraintValidator<Casse, String> {

  private boolean majuscule;

  public void initialize(Casse constraintAnnotation) {
    this.majuscule = constraintAnnotation.majuscule();
  }

  public boolean isValid(String object,
                         ConstraintValidatorContext constraintContext) {

    if (object == null)
      return true;

    if (majuscule) {
      return object.equals(object.toUpperCase());
    } else {
      return object.equals(object.toLowerCase());
    }
  }
}
```

L'interface ConstraintValidationContext encapsule des données relatives au contexte qui peuvent être exploitées lors de la validation de la contrainte sur une valeur donnée. Un objet de type ConstraintViolation peut ainsi être généré dans le cas où la donnée est invalide. Cette interface définit plusieurs méthodes notamment :

Méthode	Rôle
void disableDefaultConstraintViolation()	Désactiver la génération par défaut de l'objet de type ConstraintViolation
String getDefaultConstraintMessageTemplate()	Retourner le message par défaut non interpolé
ConstraintViolationBuilder buildConstraintViolationWithTemplate(String messageTemplate)	

Une contrainte est associée à une ou plusieurs implémentations de l'interface ConstraintValidator. Une implémentation doit être fournie pour chaque type de données sur lequel la contrainte peut être appliquée. Lors de l'évaluation d'une contrainte, la seule implémentation utilisée est celle correspondant au type de la donnée à valider.

La contrainte doit proposer une implémentation pour le type de l'entité sur laquelle elle est appliquée (classe ou interface, type de la donnée ou renvoyé par le getter). L'implémentation à utiliser est déterminée dynamiquement par le moteur de validation : une exception de type UnexpectedTypeException est levée si l'implémentation correspondante n'est pas trouvée ou si plusieurs le sont.

Toutes les implémentations utilisables lors de la validation de la contrainte doivent être déclarées dans l'attribut validatedBy

Exemple :

```
@Target({ METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = {MaContrainteValidatorPourString.class,
                           MaContrainteValidatorPourDate.class})
@Documented
public @interface MaContrainte {

    String message() default "{fr.jmdoudoux.dej.validation.MaContrainte.message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    String parametre();

    @Target({ METHOD, FIELD, ANNOTATION_TYPE})
    @Retention(RUNTIME)
    @Documented
    @interface List {
        MaContrainte[] value();
    }

}
```

Exemple :

```
package fr.jmdoudoux.dej.validation;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;


public class MaContrainteValidatorPourString implements
             ConstraintValidator<MaContrainte, String>  {

  @Override
  public void initialize(MaContrainte arg0) {
    // TODO A coder
  }

  @Override
  public boolean isValid(String arg0, ConstraintValidatorContext arg1) {
    // TODO A coder
    return false;
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.validation;

import java.util.Date;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;


public class MaContrainteValidatorPourDate implements
  ConstraintValidator<MaContrainte, Date>  {

  @Override
  public void initialize(MaContrainte arg0) {
  // TODO A coder
  }

  @Override
  public boolean isValid(Date arg0, ConstraintValidatorContext arg1) {
    // TODO A coder
    return false;
  }
}
```

111.2.5.3. Le message d'erreur

Il est nécessaire de définir un message d'erreur par défaut qui sera utilisé s'il y a une violation de la contrainte lors de la validation d'une valeur d'un bean.

La valeur du message peut être en dur mais il est recommandé d'utiliser un ResourceBundle pour permettre notamment d'internationaliser le message.

L'API propose un mécanisme d'interpolation pour permettre une détermination dynamique du message grâce à :

- L'utilisation d'un ResourceBundle
- Des placeholders qui seront remplacés par la valeur correspondante d'un attribut de la contrainte

Pour que l'API recherche le message dans un ResourceBundle, il faut mettre comme valeur de message la clé correspondante entourée par des accolades. Par défaut, l'API recherche les messages dans un fichier nommé ValidationMessages.properties dans le classpath.

Par défaut, le fichier de ce ResourceBundle se nomme ValidationMessages.properties et doit être placé dans un répertoire du classpath. Par convention, il est recommandé que la clé soit composée du nom pleinement qualifié de la contrainte suivi de « .message ».

Exemple :

```
fr.jmdoudoux.dej.validation.Casse.message=La casse de la donnée est erronée
```

Pour préciser la clé du ResourceBundle à utiliser il suffit, dans la valeur la propriété message, de mettre la clé entourée par des accolades.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;

@Target( { METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = CasseValidator.class)
@Documented
public @interface Casse {

    String message() default "{fr.jmdoudoux.dej.validation.Casse.message}";

    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};

    boolean majuscule() default false;
}
```

111.2.5.4. L'utilisation d'une contrainte

L'utilisation d'une contrainte personnalisée se fait comme avec des annotations standard : il suffit d'utiliser l'annotation de la contrainte dans le bean sur une des entités sur laquelle elle peut s'appliquer (classe, méthode ou champ).

Exemple :

```
package fr.jmdoudoux.dej.validation;

public class TestBean {

  private String codePays;

  public TestBean(String codePays) {
    super();
    this.codePays = codePays;
  }

  @Casse(majuscule=true)
  public String getCodePays() {
    return codePays;
  }

  public void setCodePays(String codePays) {
    this.codePays = codePays;
  }
}
```

La validation des beans annotées se fait avec la même API pour les annotations standard ou personnalisées. L'implémentation par défaut de l'API instancie les classes de type ConstraintValidators en utilisant l'introspection.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidationTestBean {

  public static void main(String[] args) {

    TestBean bean = new TestBean("fr");

    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();

    Set<ConstraintViolation<TestBean>> constraintViolations =
      validator.validate(bean);

    if (constraintViolations.size() > 0 ) {
      System.out.println("Impossible de valider les donnees du bean : ");
      for (ConstraintViolation<TestBean> contraintes : constraintViolations) {
        System.out.println(contraintes.getRootBeanClass().getSimpleName()
          + "." + contraintes.getPropertyPath()
          + " " + contraintes.getMessage());
      }
    } else {
      System.out.println("Les donnees du groupe sont valides");
    }
  }
}
```

Si une contrainte est appliquée sur une entité différente, une exception de type UnexpectedTypeException est levée.

Si la définition d'une contrainte est invalide, une exception de type ConstraintDefinitionException est levée lors de la validation ou lors de la recherche des métadonnées.

111.2.5.5. Application multiple d'une contrainte

Il peut être utile de vouloir appliquer plusieurs fois la même contrainte sur une même donnée avec des propriétés différentes. Ce n'est bien sûr pas utile sur les contraintes @Null ou @NotNull mais cela peut être utile sur la contrainte @Pattern par exemple.

L'utilisation d'une même contrainte plusieurs fois sur une même entité peut aussi être utile par exemple pour appliquer la contrainte sur différents groupes avec différentes propriétés.

C'est une recommandation de la spécification d'associer à une contrainte une annotation correspondante qui gère une version multiusage de l'annotation. L'implémentation de cette recommandation devrait se faire au travers de la définition d'un annotation interne nommée List.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;

@Target({ METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = MaContrainteValidator.class)
@Documented
public @interface MaContrainte {

    String message() default "{fr.jmdoudoux.dej.validation.MaContrainte.message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    String parametre();

    @Target({ METHOD, FIELD, ANNOTATION_TYPE})
    @Retention(RUNTIME)
    @Documented
    @interface List {
        MaContrainte[] value();
    }
}
```

Pour appliquer plusieurs fois la même contrainte, il faut utiliser l'annotation interne List en lui passant comme valeur d'attribut un tableau des contraintes à appliquer.

Exemple :

```
package fr.jmdoudoux.dej.validation;

public class TestBean {

  private String codePays;

  public TestBean(String codePays) {
    super();
    this.codePays = codePays;
  }

  @MaContrainte.List( {
    @MaContrainte(parametre="param1", message="message d'erreur concernant param1"),
    @MaContrainte(parametre="param2", message="message d'erreur concernant param2"),
    @MaContrainte(parametre="param3", message="message d'erreur concernant param3")
  })
  public String getCodePays() {
    return codePays;
  }

  public void setCodePays(String codePays) {
    this.codePays = codePays;
  }
}
```

111.2.6. Les contraintes composées

Il est fréquemment utile de pouvoir regrouper un ensemble de contraintes sous la forme d'une composition réutilisable. Par exemple, lorsqu'un même champ est utilisé dans deux beans distincts, il n'est pas souhaitable d'avoir à dupliquer toutes les contraintes sur les champs des deux beans pour des raisons évidentes de facilité de maintenance.

Il est possible de définir des contraintes composées (Compound Constraints).

La composition de contraintes permet de rassembler plusieurs contraintes pour en former une seule. La composition de contraintes peut avoir plusieurs utilités :

- créer une version spécifique de la contrainte
- créer une combinaison de plusieurs annotations
- exposer plusieurs contraintes sous forme d'une seule
- faciliter la réutilisation de contraintes en évitant ainsi la duplication

Pour créer une composition, il faut annoter la composition avec les annotations des contraintes qui vont la composer et l'annotation @Constraint.

Une composition doit aussi définir les attributs message, groups et payload et des attributs dédiés.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Size;

@NotNull
```

```
@Size(min = 11, max = 11,
  message="La taille du numéro de sécurité sociale est invalide")
@Pattern(regexp = "[12]\\d\\d[01]\\d\\d\\d\\d\\d\\d\\d",
  message="Le format du numéro de sécurité sociale est invalide")
@Target( { METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = {})
@Documented
public @interface NumeroSecuriteSociale {

  String message() default "Le numéro de sécurité sociale est invalide";

  Class<?>[] groups() default {};

  Class<? extends Payload>[] payload() default {};

}
```

Lorsque la contrainte sera évaluée, toutes les contraintes qui la composent le seront aussi.

Par défaut, chaque contrainte dont la validation échoue génère une erreur.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidationAssureBean {

  public static void main(String[] args) {


    AssureBean assureBean = new AssureBean("nom1",
      "prenom1",
      new GregorianCalendar(1964, Calendar.FEBRUARY, 5).getTime(),
      "3650900000");

    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();

    Set<ConstraintViolation<AssureBean>> constraintViolations =
      validator.validate(assureBean);

    if (constraintViolations.size() > 0 ) {
      System.out.println("Impossible de valider les donnees du bean : ");
      for (ConstraintViolation<AssureBean> contraintes : constraintViolations) {
        System.out.println("  "+contraintes.getRootBeanClass().getSimpleName()+
          "." + contraintes.getPropertyPath() + " " + contraintes.getMessage());
      }
    } else {
      System.out.println("Les donnees du bean sont validees");
    }
  }
}
```

Résultat :

```
Impossible de valider les donnees du
bean :

AssureBean.numSecSoc Le format du numéro de sécurité sociale est invalide
  AssureBean.numSecSoc La taille du numéro de sécurité sociale est invalide
```

Il peut être souhaité de n'avoir qu'une seul message d'erreur si au moins une contrainte n'est pas validée. L'annotation @ReportAsSingleViolation permet de préciser que si au moins une contrainte de la composition n'est pas validée alors une seule violation est reportée et celles de la composition ne sont pas remontées.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;
import javax.validation.ReportAsSingleViolation;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Size;

@NotNull
@Size(min = 11, max = 11,
  message="La taille du numéro de sécurité sociale est invalide")
@Pattern(regexp = "[12]\\d\\d[01]\\d\\d\\d\\d\\d\\d\\d",
  message="Le format du numéro de sécurité sociale est invalide")
@Target( { METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = {})
@Documented
@ReportAsSingleViolation
public @interface NumeroSecuriteSocial {

  String message() default "Le numéro de sécurité sociale est invalide";

  Class<?>[] groups() default {};

  Class<? extends Payload>[] payload() default {};

}
```

Résultat :

```
Impossible de valider les donnees du
bean :
  AssureBean.numSecSoc Le numéro de sécurité sociale est invalide
```

Il est possible qu'un attribut d'une composition redéfinisse un ou plusieurs attributs des annotations utilisées dans la composition : dans ce cas, il faut l'annoter avec @OverrideAttribute ou @OverrideAttribute.List pour un tableau d'attributs.

L'annotation redéfinie est précisée par les attributs constraint, qui définit le type, et par name qui identifie l'attribut modifié.

Les types des attributs dans la composition et dans la ou les contraintes doivent être identiques.

Une exception de type ConstraintDefinitionException est levée lors de la validation de la contrainte ou lors de la recherche de ses métadonnées si la définition n'est pas valide.

Exemple :

```
package fr.jmdoudoux.dej.validation;
```

```
import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.OverridesAttribute;
import javax.validation.Payload;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

@NotNull
@Size(message="La taille du numéro de sécurité sociale est invalide")
//@Pattern(regexp = "[12]\\d\\d[01]\\d*",
//  message="Le format du numéro de sécurité sociale est invalide")
@Target( { METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = {})
@Documented
public @interface NumeroSecuriteSociale {

  String message() default "Le numéro de sécurité sociale est invalide";

  Class<?>[] groups() default {};

  Class<? extends Payload>[] payload() default {};

  @OverridesAttribute.List( {
    @OverridesAttribute(constraint=Size.class, name="min"),
    @OverridesAttribute(constraint=Size.class, name="max") } )
  int taille() default 11;
}
```

111.2.7. L'interpolation des messages

Chaque contrainte doit définir un message par défaut sous la forme d'une propriété nommée message qui doit avoir une valeur par défaut et qui décrit la raison de l'échec de la validation de la contrainte.

Ce message peut être redéfini au moment de l'utilisation de la contrainte.

L'interface MessageInterpolator définit les méthodes pour transformer un message pour qu'il soit compréhensible par un utilisateur.

Une instance de MessageInterpolator se charge de faire l'interpolation du message : cette interpolation consiste à déterminer le message en effectuant une résolution des chaînes de caractères entre accolades qui font office de paramètres.

Le message est une chaîne de caractères qui peut contenir des paramètres entourés par des accolades. Les chaînes de caractères entre accolades dans le message peuvent avoir plusieurs significations :

- être la clé d'un message dans le ResourceBundle : le contenu entre accolades sera remplacé par la valeur correspondante à la clé
- être le nom d'un attribut de l'annotation : le contenu entre accolades sera remplacé par la valeur correspondant à l'attribut

Résultat :

```
La valeur doit être comprise entre les valeurs {min} et {max}
{fr.jmdoudoux.dej.validation.monmessage}
```

Comme les accolades ont une signification particulière, elles doivent être échappées avec un caractère backslash pour être utilisées sous une forme littérale dans le message. Ce caractère lui-même doit être échappé avec un double backslash pour ne pas être interprété.

Il est possible de créer sa propre implémentation de MessageInterpolator pour des besoins spécifiques et de la fournir en paramètre lors de l'instanciation de la fabrique ValidatorFactory.

111.2.7.1. L'algorithme d'une interpolation par défaut

Par défaut, MessageInterpolator suit l'algorithme suivant :

- étape 1 : les paramètres sont recherchés dans un ResourceBundle applicatif nommé ValidationMessage.properties. Si la propriété est trouvée alors elle est remplacée dans le message. Cette étape est effectuée récursivement (car un paramètre peut contenir un paramètre) jusqu'à ce que plus aucun paramètre ne soit trouvé
- étape 2 : les paramètres sont recherchés dans un ResourceBundle fournit par l'implémentation. Si la propriété est trouvée alors elle est remplacée dans le message. Cette étape n'est pas effectuée récursivement
- étape 3 : si l'étape 2 a effectué un remplacement alors l'étape 1 est de nouveau effectuée
- étape 4 : les paramètres sont extraits et ceux dont la valeur correspond à un des attributs de la contrainte sont remplacés par la valeur de cet attribut

Lors des recherches dans le ResourceBundle applicatif ou fourni par l'implémentation de la locale utilisée est :

- soit la locale fournie en paramètre de la méthode interpolate()
- soit la locale par défaut retournée par la méthode getDefault() de la classe Locale.

111.2.7.2. Le développement d'un MessageInterpolator spécifique

Il est possible de développer et d'utiliser son propre MessageInterpolator pour par exemple prendre en compte une Locale particulière ou obtenir les valeurs des paramètres d'une ressource particulière.

Il faut créer une classe qui implémente l'interface MessageInterpolator. Cette interface définit plusieurs méthodes :

Méthode	Rôle
String interpolate(String messageTemplate, Context context)	Interpoler le message final avec la locale par défaut
String interpolate(String messageTemplate, Context context, Locale locale)	Interpoler le message final avec la locale fournie en paramètre
ConstraintDescriptor<?> getConstraintDescriptor()	Renvoyer la contrainte dont le message est interpolé
Object getValidatedValue()	Renvoyer la valeur en cours de validation

Un objet de type Contexte encapsule des informations relatives à l'interpolation.

La méthode interpolate() de l'instance de MessageInterpolator est invoquée pour chaque contrainte dont la validation échoue.

Une implémentation de MessageInterpolator devrait être thread safe.

Pour associer un MessageInterpolator spécifique à un Validator, il faut utiliser la méthode messageInterpolator() de la classe Configuration en lui passant l'instance de MessageInterpolator à utiliser. Cet objet Configuration doit ensuite être fourni pour obtenir l'instance de ValidatorFactory.

Il n'y a qu'une seule instance de MessageInterpolator pour un Validator. Il est possible de remplacer cette instance pour une instance de Validator donnée en utilisant la méthode ValidatorFactory.usingContext().messageInterpolator().

Pour obtenir le MessageInterpolator par défaut, il faut invoquer la méthode Configuration.getDefaultMessageInterpolator().

111.2.8. Bootstrapping

Le bootstrapping propose plusieurs solutions pour obtenir une instance d'une fabrique de type ValidatorFactory qui va permettre de créer une instance de type Validator. Ces mécanismes permettent de découpler l'application de l'implémentation de l'API Bean Validation du fournisseur utilisé.

Le bootstrapping permet de :

- gérer plusieurs implémentations
- choisir l'implémentation à utiliser
- configurer l'implémentation utilisée
- s'intégrer dans les conteneurs notamment ceux de Java EE 6

Les mécanismes de bootstrap mettent en oeuvre plusieurs interfaces :

- Validation : c'est le point d'entrée de l'API pour utiliser une implémentation
- ValidationProvider : interface qui définit les fonctionnalités utilisables lors du bootstrap
- ValidationProviderResolver : permet de rechercher la liste des implémentations utilisables dans le contexte d'exécution
- Configuration : encapsule les données de configuration lors de la création de l'instance de type ValidatorFactory
- ValidatorFactory : fabrique qui est instanciée par le mécanisme de bootstrap. Le rôle de cette fabrique est de créer des instances de type Validator

Le fichier META-INF/validation.xml peut aussi contenir des données de configuration pour le mécanisme de bootstrap.

La façon la plus facile pour obtenir une instance de la classe Validator est d'utiliser la méthode statique buidDefaultValidatorFactory() de la classe Validation et d'utiliser la fabrique pour créer une instance du type Validator.

Il existe plusieurs autres façons pour obtenir la fabrique :

- Utilisation du mécanisme proposé par le Java Service Provider mis en oeuvre par le fournisseur de l'implémentation
- Utilisation des méthodes statiques de la classe Validation

111.2.8.1. L'utilisation du Java Service Provider

Une implémentation de l'API Bean Validation peut être découverte grâce à l'utilisation du Java Service Provider.

Pour cela le fournisseur doit fournir un fichier nommé javax.validation.spi.ValidationProvider dans le répertoire META-INF/services du package (jar, war, ...) de l'implémentation.

Ce fichier doit contenir le nom pleinement qualifié de la classe de l'implémentation de ValidationProvider proposée par le fournisseur.

111.2.8.2. L'utilisation de la classe Validation

La classe Validation est le point d'entrée pour utiliser l'API bootstraping. Elle propose plusieurs méthodes pour obtenir de façon plus ou moins directe une instance du type ValidatorFactory.

La méthode buildDefaultValidatorFactory() permet d'obtenir l'instance de type ValidatorFactory() par défaut. Elle utilise l'implémentation par défaut de la classe ValidationProviderResolver pour déterminer l'ensemble des implémentations présentes dans le classpath.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestBootstrapBuildDefault {

  public static void main(String[] args) {
    ValidatorFactory fabrique = Validation.buildDefaultValidatorFactory();
    Validator validator = fabrique.getValidator();

    ...
  }
}
```

Si plusieurs implémentations sont présentes dans le classpath, il n'y a aucune garantie sur celle qui sera choisie lors de l'utilisation de la méthode buildDefaultValidatorFactory() de la classe Validation.

La méthode byDefaultProvider() permet de configurer la création d'une instance de la classe ValidatorFactory personnalisée. Cette méthode renvoie une instance de l'interface GenericBootstrap.

La méthode providerResolver() de l'interface GenericBootstrap permet éventuellement de préciser l'instance de type ValidationProviderResolver fournie en paramètre qui sera utilisée pour déterminer le ValidationProvider à utiliser.

La méthode configure() de l'interface GenericBootstrap crée une instance générique de l'interface Configuration en utilisant la méthode createGenericConfiguration() du premier ValidationProvider trouvé.

L'interface Configuration propose plusieurs méthodes pour préciser une instance des différents types d'interfaces qui seront utilisés par la fabrique (MessageInterpolator, TraversableResolver et ConstraintValidatorFactory).

Exemple :

```
package fr.jmdoudoux.dej.validation;

import javax.validation.Configuration;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;
import javax.validation.bootstrap.GenericBootstrap;

public class TestBootstratByDefaultProvider {

  public static void main(String[] args) {

    GenericBootstrap bootstrap = Validation.byDefaultProvider();

    Configuration<?> configuration = bootstrap.configure();
    ValidatorFactory factory = configuration.buildValidatorFactory();
    Validator validator = factory.getValidator();
  }
}
```

Il est possible de fournir sa propre instance de ValidationProviderResolver.

La méthode buildDefaultValidatorFactory() est équivalente à une invocation de Validation.byDefaultProvider().configure().buildValidatorFactory().

La méthode byProvider() permet d'obtenir une instance de l'interface ProviderSpecificBootstrap pour une instance de configuration qui soit spécifique à l'implémentation précise fournie en paramètre. Sa méthode configure() permet d'obtenir une instance de l'interface Configuration typée avec l'implémentation en utilisant la méthode createSpecializedConfiguration() de l'instance de ValidationProvider.

Cette méthode est particulièrement utile pour obtenir une instance d'une implémentation particulière alors que plusieurs implémentations sont présentes dans le classpath.

Exemple : obtenir une instance de ValidatorFactory de l'implémentation de référence

Exemple :

```
package fr.jmdoudoux.dej.validation;

import javax.validation.Configuration;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;
import javax.validation.bootstrap.ProviderSpecificBootstrap;

import org.hibernate.validator.HibernateValidator;
import org.hibernate.validator.HibernateValidatorConfiguration;

public class TestBootstrapByProvider {

  public static void main(String[] args) {

    ProviderSpecificBootstrap<HibernateValidatorConfiguration> psb =
      Validation.byProvider(HibernateValidator.class);
    Configuration configuration = psb.configure();
    ValidatorFactory fabrique = configuration.buildValidatorFactory();
    Validator validator = fabrique.getValidator();
  }
}
```

L'instance de l'interface Validator obtenue de la fabrique doit être thread safe et peut donc être mise en cache.

111.2.8.3. Les interfaces ValidationProvider et ValidationProviderResolver

L'interface ValidationProvider définit les fonctionnalités qu'une implémentation doit fournir pour être utilisée par l'API de bootstrap.

L'interface ValidationProviderResolver définit les fonctionnalités pour rechercher les implémentations de l'API Bean Validation.

111.2.8.3.1. L'interface ValidationProviderResolver

Par défaut, les implémentations sont déterminées en utilisant le mécanisme du Java Service Provider. Chaque fournisseur doit fournir un fichier javax.validation.spi.ValidationProvider dans le sous-répertoire META-INF/services du jar qui contient le nom pleinement qualifié de la classe implémentant l'interface javax.validation.spi.ValidationProvider.

L'implémentation par défaut de l'interface ValidationProviderResolver recherche dans le classpath toutes les implémentations qui définissent un service.

L'interface ValidationProviderResolver définit une seule méthode : getValidationProviders() qui renvoie une collection de type List<ValidationProvider<?>> contennant la liste des implémentations utilisables.

Pour des cas spécifiques (utilisation d'un classloader spécifique comme avec OSGi, impossibilité d'utiliser le Java Service Provider, ...), il est possible de développer sa propre implémentation de l'interface ValidationProviderResolver.

111.2.8.3.2. L'interface ValidationProvider

Cette interface a pour rôle de lier l'API de bootstrap et l'implémentation.

La signature de cette interface est typée avec un generic dont le type doit hériter de Configuration :

public interface ValidationProvider<T extends Configuration<T>>

Cette interface définit plusieurs méthodes :

Méthodes	Rôle
T createSpecializedConfiguration(BootstrapState state)	Renvoie une instance spécifique à l'implémentation
Configuration<?> createGenericConfiguration(BootstrapState state)	Renvoie une instance de Configuration générique qui n'est donc pas liée à l'implémentation
ValidatorFactory buildValidatorFactory(ConfigurationState configurationState)	Renvoie une instance initialisée du type ValidatorFactory

Une instance de cette interface permet d'identifier une implémentation particulière de l'API Bean Validation.

Une implémentation de l'API doit fournir une classe implémentant cette interface avec un constructeur sans argument et fournir un fichier javax.validation.spi.ValidationProvider dans le répertoire META-INF/services qui doit avoir le nom pleinement qualifié de cette classe.

111.2.8.4. L'interface MessageInterpolator

Il est possible d'avoir besoin de sa propre implémentation de l'interface MessageInterpolator.

Il faut fournir une instance de cette classe à la méthode messageInterpolator() de l'instance de Configuration utilisée pour obtenir une instance de la ValidationFactory. Ainsi, toutes les instances de Validator créées par la fabrique utiliseront le MessageInterpolator personnalisé.

Il est recommandé qu'une implémentation délègue à la fin de ses traitements une invocation du MessageInterpolator par défaut pour garantir que les règles par défaut soient prises en compte. Pour obtenir une instance du MessageInterpolator par défaut il faut utiliser la méthode Configuration.getDefaultMessageInterpolator().

111.2.8.5. L'interface TraversableResolver

L'interface TraversableResolver a pour but de restreindre l'accès à certaines propriétés lors de la validation d'un bean. Un exemple d'utilisation peut être le besoin de ne pas valider les données d'une propriété d'un bean de type entité dont le chargement est tardif (lazy loading). Au moment de la validation du bean, les données de cette propriété peuvent ne pas être chargée, rendant leur validation erronée.

Cette interface définit plusieurs méthodes :

Méthode	Rôle
boolean isReachable(Object traversableObject, Path.Node traversableProperty, Class<?> rootBeanType, Path pathToTraversableObject, ElementType elementType);	Permet de déterminer si le moteur de validation peut accéder à la valeur de la propriété
boolean isCascadable(Object traversableObject, Path.Node traversableProperty, Class<?> rootBeanType, Path pathToTraversableObject, ElementType elementType);	Permet de déterminer si le moteur de validation peut valider la propriété marquée avec l'annotation @Valid. Cette méthode n'est invoquée que si l'invocation de la méthode isReachable() pour la propriété a renvoyé true

Pour créer la fabrique de type ValidatorFactory, il est possible de définir sa propre implémentation de l'interface TrasersableResolver et de fournir cette instance en paramètre de la méthode traversableResolver() de la Configuration utilisée.

Une implémentation de l'interface TraversableResolver doit être thread safe.

111.2.8.6. L'interface ConstraintValidatorFactory

Une instance d'un valideur de contraintes est créée par une fabrique de type ConstraintValidatorFactory.

L'interface ConstraintValidatorFactory ne définit qu'une seule méthode :

Méthode	Rôle
<T extends ConstraintValidator<?,?>> T getInstance(Class<T> key)	renvoie une instance de l'interface ConstraintValidator

Il est recommandé que la fabrique utilise le constructeur par défaut pour créer l'instance. Elle ne devrait pas mettre en cache les instances créées.

Si une exception est levée dans la méthode getInstance(), celle-ci est propagée sous la forme d'une exception de type ValidationException.

111.2.8.7. L'interface ValidatorFactory

Le but d'une instance de l'interface ValidatorFactory est de proposer une fabrique pour créer et initialiser des objets de type Validator. Chaque instance de type Validator est créée pour un MessageInterpolator, un TraversableResolver et un ConstraintValidatorFactory donnés.

L'interface ValidatorFactory définit plusieurs méthodes.

La méthode getValidator() renvoie l'instance créée par la fabrique : celle-ci peut être stockée dans un pool.

La méthode getMessageInterpolator() renvoie l'instance de type MessageInterpolator utilisée par la fabrique.

La méthode getTraversalResolver() renvoie l'instance de type TraversalResolver utilisée par la fabrique.

La méthode getConstraintValidatorFactory() renvoie l'instance de type ConstraintValidatorFactory utilisée par la fabrique.

La méthode unwrap() permet un accès à un objet spécifique à l'implémentation qui peut encapsuler des données complémentaires. Son utilisation rend le code non portable.

La méthode usingContext() renvoie un objet de type ValidatorContext qui peut encapsuler des informations de configuration. Lors de la création des instances de type Validator notamment une instance de type MessageInterpolator, TraversableResolver ou ConstraintValidatorFactory, ces informations seront utilisées à la place de celles de la fabrique.

Un objet de type ValidatorFactory est créé par un objet de type Configuration.

Une implémentation de ValidatorFactory doit être thread safe.

111.2.8.8. L'interface Configuration

Le but d'une instance de Configuration est de définir les différentes entités utiles à une instance de ValidatorFactory et de permettre de créer une telle instance en ayant sélectionné l'implémentation de l'API à utiliser.

Par défaut, l'implémentation à utiliser est déterminée par :

- l'instance spécifiée par l'utilisation de la méthode byProvider() de la classe Validation
- le contenu du fichier META-INF/validation.xml
- l'instance de type ValidatorProviderResolver fournie à la Configuration ou l'instance par défaut de cette interface

La signature de l'interface est :

public interface Configuration<T extends Configuration<T>>

Cette interface propose plusieurs méthodes notamment :

Méthodes	Rôle
T messageInterpolator(MessageInterpolator interpolator)	Définir l'instance de type MessageInterpolator qui sera utilisée par la fabrique. Si cette méthode n'est pas utilisée ou que la valeur null lui est passée en paramètre alors c'est l'instance par défaut de l'implémentation qui sera utilisée
T constraintValidatorFactory(ConstraintValidatorFactory constraintValidatorFactory)	Définir l'instance de type ConstraintValidatorFactory qui sera utilisée par la fabrique. Si cette méthode n'est pas utilisée ou que la valeur null lui est passée en paramètre alors c'est l'instance par défaut de l'implémentation qui sera utilisée
ValidatorFactory buildValidatorFactory()	Créer une instance de type ValidatorFactory. Le fournisseur à utiliser est déterminé et la méthode buildValidatorFactory de son instance de type ValidationProvider est invoquée
T ignoreXmlConfiguration()	Demander de ne pas tenir compte du contenu du fichier META-INF/validation.xml
T traversableResolver(TraversableResolver resolver)	Définir l'instance de type TraversableResolver qui sera utilisée par la fabrique. Si cette méthode n'est pas utilisée ou que la valeur null lui est passée en paramètre alors c'est l'instance par défaut de l'implémentation qui sera utilisée
T addProperty(String name, String value)	Définir une propriété spécifique à l'implémentation
MessageInterpolator getDefaultMessageInterpolator()	Renvoyer l'implémentation par défaut du type MessageInterpolator
ConstraintValidatorFactory getDefaultConstraintValidatorFactory()	Renvoyer l'implémentation par défaut du type ConstraintValidatorFactory

Les méthodes qui permettent de définir des données renvoient le type en paramètre du generic pour permettre de chaîner leurs invocations.

La détermination de l'implémentation à utiliser suit plusieurs règles ordonnées :

- Utilisation de l'implémentation désignée par l'invocation de la méthode byProvider() de la classe Validation
- Utilisation des informations contenues dans le fichier META-INF/validation.xml
- Utilisation de la première implémentation renvoyée par la méthode getValidationProviders() de la classe ValidationProvider

Une instance de type Configuration est créée grâce à la classe ValidationProvider.

Une instance de Configuration est utilisée par la classe Validation.

111.2.8.9. Le fichier de configuration META-INF/validation.xml

L'utilisation de ce fichier est ignorée si la méthode ignoreXMLConfiguration() de l'instance de type Configuration est invoquée.

L'utilisation du fichier META-INF/validation.xml est optionnelle mais elle permet de facilement définir l'implémentation de l'API Bean Validation qui doit être utilisée et permet de la configurer.

Un seul fichier META-INF/validation.xml doit être présent dans le classpath sinon une exception de type ValidationException est levée.

Ce fichier au format XML possède un tag racine nommé validation-config qui peut avoir plusieurs tags fils.

Tag	Rôle
default-provider	Indiquer le nom pleinement qualifié de l'implémentation du type ValidationProvider du fournisseur à utiliser
message-interpolator	Indiquer le nom pleinement qualifié de l'implémentation du type MessageInterpolator à utiliser (optionnel)
traversable-resolver	Indiquer le nom pleinement qualifié de l'implémentation du type TraversableResolver à utiliser (optionnel)
constraint-validator-factory	Indiquer le nom pleinement qualifié de l'implémentation du type ConstraintValidatorFactory à utiliser (optionnel)
constraint-mapping	Indiquer le chemin d'un fichier XML de mapping (optionnel)
property	Indiquer une propriété spécifique à une implémentation sous la forme d'une paire clé/valeur (optionnel)

Exemple : demander l'utilisation de l'implémentation de référence (Hibernate Validator)

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<validation-config
   xmlns="http://jboss.org/xml/ns/javax/validation/configuration"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation=
     "http://jboss.org/xml/ns/javax/validation/configuration validation-configuration-1.0.xsd">
   <default-provider>org.hibernate.validator.HibernateValidator</default-provider>
</validation-config>
```

111.2.9. La définition de contraintes dans un fichier XML

La suite de cette section sera développée dans une version future de ce document

111.2.10. L'API de recherche des contraintes

Les spécifications de l'API Bean Validation proposent une API qui permet de rechercher les métadonnées relatives aux contraintes définies dans les beans stockés dans un repository.

Cette API est destinée à être utilisée par des outils ou pour l'intégration dans des frameworks ou des bibliothèques.

L'interface Validator propose la méthode getConstraintsForClass() qui attend en paramètre un objet de type Class<?> et renvoie un objet immuable de type BeanDescriptor contenant une description des contraintes de la classe concernée. La méthode getContraintsForProperty() attend en paramètre le nom de la propriété et renvoie un objet immuable de type BeanDescriptor qui contient une description des contraintes de la propriété concernée.

Une instance de type BeanDescriptor encapsule une description des contraintes du bean et propose un accès au métadonnées de ces contraintes.

Si la définition ou la déclaration d'une contrainte contenue dans la classe est invalide alors une exception de type ValidationException ou une de ses sous-classes telles que ConstraintDefinitionException, ConstraintDeclarationException ou UnexpectedTypeException est levée.

111.2.10.1. L'interface ElementDescriptor

L'interface javax.validation.metadata.ElementDescriptor encapsule la description d'un élément de la classe qui possède une ou plusieurs contraintes.

La méthode hasConstraint() renvoie un booléen qui précise si l'élément (classe, champ ou getter) est soumis à au moins une contrainte.

La méthode Set<ConstraintDescriptor<?>> getConstraintDescriptors() renvoie une collection des descriptions des contraintes associées à l'élément.

Un objet de type ConstraintDescriptor encapsule les informations relatives à une contrainte.

La méthode findConstraints() qui renvoie une instance de type ConstraintFinder permet de rechercher des contraintes selon certains critères.

111.2.10.2. L'interface ConstraintFinder

L'interface ConstraintFinder définit plusieurs méthodes qui permettent de préciser les critères de recherche :

Méthode	Rôle
ConstraintFinder unorderedAndMatchingGroup(Class<?> ...)	Filtre sur les groupes déclarés dans la contrainte
ConstraintFinder declaredOn(ElementType ...)	Filtre sur les types d'éléments sur lesquels la contrainte est appliquée (ElementType.FIELD, ElementType.METHOD, ElementType.TYPE)
ConstraintFinder lookingAt(Scope)	Filtre sur la portée de la recherche (Scope.LOCAL_ELEMENT ou Scope.HIERARCHY)

111.2.10.3. L'interface BeanDescriptor

L'interface javax.validation.meta.BeanDescriptor qui hérite de l'interface ElementDescriptor encapsule un bean présentant une ou plusieurs contraintes.

Méthode	Rôle
boolean isBeanConstrained()	renvoie un booléen qui précise si le bean contient une contrainte sur lui-même, sur une de ses propriétés ou si une de ses propriétés est marquée avec l'annotation @valid. Lorsqu'elle renvoie false, le moteur de validation ignore ce bean lors de ses traitements

PropertyDescriptor getConstraintsForProperty(String propertyName)	renvoie un objet qui contient la description de la propriété dont le nom est fourni en paramètre. Renvoie null si la propriété n'existe pas, si elle n'a pas de contrainte ou si elle n'est pas marquée avec @Valid.
Set<PropertyDescriptor> getConstrainedProperties()	renvoie une collection des descripteurs de propriétés qui présentent au moins une contrainte ou qui sont marquées avec l'annotation @Valid.

111.2.10.4. L'interface PropertyDescriptor

L'interface javax.validation.metadata.PropertyDescriptor qui hérite de l'interface ElementDescriptor encapsule une propriété qui présente au moins une contrainte.

Méthode	Rôle
boolean isCascaded()	Renvoie true si la propriété est marquée avec l'annotation @Valid
String getPropertyName()	Renvoie le nom de la propriété

111.2.10.5. L'interface ConstraintDescriptor

L'interface javax.validation.metadata.ConstraintDescriptor<T extends Annotation> décrit une annotation d'une contrainte.

Méthode	Rôle
T getAnnotation()	Renvoie l'annotation de la contrainte
Set<Class< ?>> getGroups	Renvoie une collection des groupes définis dans l'annotation. Si aucun groupe n'est défini alors c'est le groupe par défaut Default qui est retourné
SetClass< ? extends Payload>> getPayload()	Renvoie une collection des données supplémentaires définies dans l'annotation
List<Class< ? extends ConstraintValidator<T, ?>>> getConstraintValidatorClasses()	Renvoie une collection des classes qui implémentent la logique de validation des données
Map<String, Object> getAttributes	Renvoie une collection de type Map qui contient les attributs de l'annotation : la clé contient le nom de l'attribut, la valeur contient sa valeur
Set<ConstraintDescriptor<?>> getComposingConstraints()	Renvoie une collection des contraintes qui sont dans la composition ou un ensemble vide si la contrainte n'est pas composée
boolean isReportAsSingleViolation()	Renvoie true si la contrainte est annotée avec @ReportAsSingleViolation

111.2.10.6. Un exemple de mise en oeuvre

L'exemple de cette section va rechercher les contraintes déclarées sur une propriété d'un bean.

Exemple :

```
package fr.jmdoudoux.dej.validation;

import java.lang.annotation.ElementType;
import java.util.Set;

import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;
import javax.validation.groups.Default;
import javax.validation.metadata.ConstraintDescriptor;
import javax.validation.metadata.PropertyDescriptor;
import javax.validation.metadata.Scope;

public class TestMetaData {

  public static void main(String[] args) {

    Set<ConstraintDescriptor<?>> contraintes = null;

    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();

    // obtenir le descripteur de la propriété
    PropertyDescriptor pd = validator.getConstraintsForClass(PersonneBean.class)
                                     .getConstraintsForProperty("nom");

    // affichage de toutes les contraintes
    contraintes = pd.getConstraintDescriptors();
    System.out.println("Nombre de contraintes=" + contraintes.size());
    afficher(contraintes);

    // recherche des contraintes
    contraintes = pd.findConstraints()
                    .declaredOn(ElementType.METHOD)
                    .unorderedAndMatchingGroups(Default.class)
                    .lookingAt(Scope.LOCAL_ELEMENT)
                    .getConstraintDescriptors();

    System.out.println("Nombre de contraintes trouvees=" + contraintes.size());
    afficher(contraintes);

  }

  private static void afficher(Set<ConstraintDescriptor<?>> constraintes) {
    for (ConstraintDescriptor<?> contrainte : constraintes) {
      System.out.println("  " + contrainte.getAnnotation().toString());
    }
  }
}
```

Résultat :

```
Nombre de contraintes=2
@javax.validation.constraints.NotNull(message={javax.validation.constraints.NotNull.message},
payload=[], groups=[])
@javax.validation.constraints.Size(message={javax.validation.constraints.Size.message},
min=0, max=50, payload=[], groups=[])
Nombre de contraintes trouvees=2
@javax.validation.constraints.NotNull(message={javax.validation.constraints.NotNull.message},
payload=[], groups=[])
@javax.validation.constraints.Size(message={javax.validation.constraints.Size.message},
min=0, max=50, payload=[], groups=[])
```

111.2.11. La validation des paramètres et de la valeur de retour d'une méthode

Les spécifications proposent une extension, dont l'implémentation par un fournisseur est optionnelle, qui permet d'utiliser les mécanismes de définition, de déclaration et de validation des contraintes au niveau des paramètres d'une méthode.

Cette extension peut être exploitée par exemple dans des aspects ou dans un interceptor.

Elle peut être utilisée pour valider les paramètres en entrée ou la valeur de retour d'une méthode lorsque celle-ci est invoquée : la validation doit être appliquée autour de l'invocation de la méthode.

La spécification définit plusieurs méthodes dans l'interface Validator pour permettre la validation des paramètres.

Méthode	Rôle

<T> Set<ConstraintViolation<T>> validateParameters(Class<T> class, Method method, Object[] parameterValues, Class<?> ... groups);	Valider chacune des valeurs passées selon les contraintes des paramètres de la méthode
<T> Set<ConstraintViolation> validateParameter(Class<T> class, Method method, Object parameterValue, int parameterIndex, Class<?>... groups);	Valider la valeur d'un paramètre selon les contraintes de celui dont l'index est fourni
<T> Set<ConstraintViolation> validateReturnedValue(Class<T> class, Method method, Object returnedValue, Class<?>... groups);	Valider la valeur de retour de la méthode
<T> Set<ConstraintViolation> validateParameters(Class<T> class, Constructor constructor, Object[] parameterValues, Class<?> ... groups);	Valider chacune des valeurs passées selon les contraintes des paramètres du constructeur
<T> Set<ConstraintViolation> validateParameter(Class<T> class, Constructor constructor, Object parameterValue, int parameterIndex, Class<?>... groups);	Valider la valeur d'un paramètre selon les contraintes définies sur celui dont l'index est fourni

Les contraintes appliquées sur un paramètre de la méthode ou d'un constructeur seront évaluées. Si l'annotation @Valid est utilisée sur un paramètre alors ce sont les contraintes contenues dans la classe du paramètre qui seront évaluées lors de la validation.

111.2.12. L'implémentation de référence : Hibernate Validator

La version 4.0 du projet Hibernate Validator est l'implémentation de référence de la JSR 303.

Pour mettre en oeuvre Hibernate Validator, il faut :

- télécharger l'archive sur le site du projet
- décompresser l'archive dans un répertoire du système
- ajouter le fichier hibernate-validator-4.0.1.GA.jar et les fichiers *.jar du sous-répertoire lib au classpath du projet

111.2.13. Les avantages et les inconvénients

Les avantages sont :

- standardisation des fonctionnalités de validation des données d'un bean
- déclaration des contraintes simplifiée par des annotations
- uniformisation de la déclaration des contraintes au niveau du bean
- validation des contraintes à la discrétion des développeurs dans n'importe quelle couche Java d'une application

Les inconvénients sont :

- requiert un Java 5 minimum
- seuls les JavaBeans peuvent être validés

Il existe aussi plusieurs manques dans la JSR 303 notamment :

- le support de la Locale du client côté serveur
- le support de la validation des paramètres d'une méthode qui est proposé sous la forme d'une extension dont le support est optionnel
- la possibilité de définir des contraintes en utilisant une expression ou un script : cette fonctionnalité peut être développée sous la forme d'une contrainte personnalisée

111.3. D'autres frameworks pour la validation des données

Il existe plusieurs autres frameworks pour la validation des données.

Framework	Description
Commons-Validator	https://commons.apache.org/validator/ Ce framework du projet Apache Commons propose un moteur de validation et des routines de validation standard
Oval	https://oval.sourceforge.net/ Ce framework open source utilise les annotations pour la déclaration de contraintes sur n'importe quel objet Java.
iScreen	http://iscreen.sourceforge.net/docs/index.html Ce framework open source utilise les annotations pour la déclaration de contraintes sur n'importe quel objet Java.
agimatec-validation	https://code.google.com/p/agimatec-validation/ Ce framework open source implémente la JSR 303

112. L'utilisation des dates

Chapitre 112

Niveau : 🟩🟩⬜⬜⬜⬜ Elémentaire

La manipulation des dates n'est pas toujours facile à mettre en oeuvre :

- il existe plusieurs calendriers dont le plus usité est le calendrier Grégorien
- le calendrier Grégorien comporte de nombreuses particularités : le nombre de jours d'un mois varie selon le mois, le nombre de jours d'une année varie selon l'année (année bissextile), ...
- le format textuel de restitution des dates diffère selon la Locale utilisée
- l'existence des fuseaux horaires qui donnent une date/heure différente d'un point dans le temps selon la localisation géographique
- la possibilité de prendre en compte un décalage horaire lié aux heures d'été et d'hiver
- ...

Pourtant le temps s'écoule de façon linéaire : c'est d'ailleurs de cette façon que les calculs de dates sont réalisés avec Java, en utilisant une représentation de la date qui indique le nombre de millisecondes écoulées depuis un point d'origine défini. Dans le cas de Java, ce point d'origine est le 1er janvier 1970. Ceci permet de définir un point dans le temps de façon unique.

L'utilisation de dates en Java est de surcroît plus compliquée à cause de l'API historique qui permet leur gestion car elle n'est pas toujours intuitive.

Il est intéressant de découpler l'obtention de la date/heure système par exemple en utilisant une fabrique. Cette fabrique renvoie la date/heure système en production mais elle est aussi capable de renvoyer une date/heure déterminée.

Exemple :

```
Date aujourdhui = SystemClockFactory.getDatetime();
```

L'utilisation d'une telle fabrique peut être particulièrement utile lors de tests unitaires ou d'intégration pour faciliter la vérification des résultats par rapport à un type de données dont la valeur par définition évolue constamment.

Ceci évite entre autres d'avoir à modifier la date système sur la ou les machines sur lesquelles les tests sont exécutés.

La bibliothèque jFin propose aussi des fonctionnalités relatives au traitement des dates spécifiquement dédiées à la finance.

Ce chapitre contient plusieurs sections :

♦ Les classes standard du JDK pour manipuler des dates
♦ Des exemples de manipulations de dates
♦ La classe SimpleDateFormat
♦ Joda Time
♦ La classe FastDateFormat du projet Apache commons.lang
♦ L'API Date and Time

112.1. Les classes standard du JDK pour manipuler des dates

En Java 1.0, la classe java.util.Date était seule responsable de l'encapsulation et de la manipulation d'une date.

A partir de Java 1.1, la responsabilité de la gestion et des traitements sur les dates est répartie sur plusieurs classes :

- java.util.Date : elle encapsule un point dans le temps
- java.util.Calendar et java.util.GregorianCalendar : elle permet la manipulation d'une date
- java.util.TimeZone et java.util.SimpleTimeZone : elle encapsule un fuseau horaire à partir du méridien de Greenwich (GMT) et les informations relatives aux décalages concernant les heures d'été et d'hiver
- java. text.DateFormat, java.text.SimpleDateFormat : elle permet de convertir une date en chaîne de caractères et vice versa
- java.text.DateFormatSymbols : elle permet de traduire les différents éléments d'une date (jour, mois, ...)

Les classes permettant la mise en oeuvre des dates sont dans le package java.util exceptées celles relatives à leur conversion de et vers une chaîne de caractères qui sont dans le package java.text.

Le package java.sql contient aussi des classes relatives aux dates et à leur utilisation dans une base de données :

- java.sql.Date : hérite de java.util.Date et n'encapsule que la date sans la partie horaire
- java.sql.Time : hérite de java.util.Date et n'encapsule que la partie horaire sans la partie date
- java.sgl.Timestamp : encapsule un point dans le temps avec une représentation particulière pour une utilisation avec SQL

Les classes abstraites Calendar, TimeZone et DateFormat possèdent toutes une implémentation concrète respectivement GregorianCalendar, SimpleTimeZone et SimpleDateFormat.

La conception des classes qui encapsulent et manipulent des dates ne facilitent pas leur mise en oeuvre. C'est d'autant plus dommageable que l'utilisation de dates est courante notamment dans les applications de gestion.

Par exemple, l'API propose au moins quatre manières pour obtenir un point dans le temps depuis le 1 janvier 1970 :

Exemple :

```
System.out.println(System.currentTimeMillis());
System.out.println(new java.util.Date().getTime());
System.out.println(Calendar.getlnstance().getTimelnMillis() );
System.out.println(Calendar.getlnstance().getTime().getTime ())
```

L'API de gestion des dates en Java est particulièrement propice à la confusion et à l'obtention d'erreurs potentielle :

- nommage de certaines méthodes (Date.getTime(), Date.getDate(), ...)
- gestion des mois de 0 à 11 au lieu de 1 à 12

112.1.1. La classe java.util.Date

Cette classe encapsule, sous la forme d'une variable de type long, un point dans le temps qui est représenté par le nombre de millisecondes écoulées entre le 1 janvier 1970 à 00 heure 00 GMT et l'instant concerné.

Depuis la version 1.1, toutes les méthodes permettant de manipuler la date sont deprecated.

Par défaut, cette classe encapsule le point courant dans le temps obtenu en utilisant la méthode System.currentTimeMillis() ce qui rend sa précision dépendante du système d'exploitation.

112.1.2. La classe java.util.Calendar

La classe Calendar encapsule un point dans le temps (une Date sous la forme d'une variable de type long) et permet une représentation et une manipulation dans un calendrier et un fuseau horaire.

La classe Calendar n'est pas stateless puisqu'elle encapsule un point dans le temps : il est donc nécessaire d'initialiser ce point avant de pouvoir utiliser l'instance de Calendar.

Une nouvelle instance de la classe est toujours initialisée avec le point dans le temps courant. Pour encapsuler un autre point, il faut obligatoirement après l'instanciation utiliser une des méthodes de la classe pour modifier le point encapsulé.

Pour accéder aux différentes propriétés de la date encapsulée dans l'instance de Calendar, il n'existe pas un getter pour chaque propriété mais une seule méthode get() qui attend en paramètre le nom de la propriété souhaitée et qui retourne toujours une valeur de type int.

La classe Calendar définit des constantes de type int pour le nom de ces propriétés.

La classe Calendar définit aussi plusieurs constantes qui contiennent les valeurs possibles pour certaines propriétés. Leur utilisation est fortement recommandée car certaines valeurs sont parfois surprenantes notamment celles qui encapsulent un mois. La valeur d'un mois varie de 0 à 11 correspondant aux constantes Calendar.JANUARY à Calendar.DECEMBER. Calendar définit aussi la constante UNDECIMBER qui représente le treizième mois de l'année requis par certains calendriers.

Attention : toutes ces constantes sont définies pêle-mêle dans la classe et ne sont donc pas groupées par une convention de nommage dans une interface dédiée par rôle. Elles sont toutes de types int, ce qui peut permettre d'utiliser n'importe quelle constante à la place d'une autre.

Exemple :

```
Calendar calendar = Calendar.getInstance();
if ( calendar.get( Calendar.MONTH )==Calendar.JANUARY ) {
  system.out.prinln("la date courante est en janvier"); }
```

La classe Calendar propose trois façons de manipuler la date qu'elle encapsule en agissant sur les éléments qui la composent :

- set() : permet de modifier un élément de la date
- add() : permet de modifier un élément de la date en tenant compte des impacts sur les autres éléments qui composent la date
- roll() : identique à la méthode add() mais sans affecter les autres éléments de la date

La date encapsulée dans Calendar peut être manipulée de deux façons :

- directement par un calcul sur le nombre de millisecondes écoulées depuis le 1er janvier 1970
- en agissant sur les éléments qui composent la date en utilisant les méthodes dédiées

112.1.3. La classe java.util.GregorianCalendar

La classe java.util.GregorianCalendar est la seule implémentation concrète de la classe Calendar fournie en standard. Cette implémentation correspond au calendrier Grégorien.

La méthode isLeapYear() permet de savoir si l'année encapsulée par la classe est bissextile.

112.1.4. Les classes java.util.TimeZone et java.util.SimpleTimeZone

La classe abstraite TimeZone et sa sous-classe SimpleTimeZone encapsulent un fuseau horaire.

Une instance de type TimeZone est utilisée par la classe Calendar pour déterminer la date correspondant au point dans le temps qu'elle encapsule. Un même point dans le temps correspond à des dates/heures différentes pour deux fuseaux horaires différents.

Un fuseau horaire correspond à un certain décalage vis à vis du méridien de référence, le méridien de Greenwich. Le fuseau horaire correspondant à ce méridien est désigné par GMT.

Ce décalage peut en plus être affecté par un second décalage induit par les heures d'été et d'hiver (daylight savings time (DST)) si ceux-ci sont mis en place dans le pays considéré.

La classe TimeZone encapsule un nom long et un nom court qui permet d'identifier le fuseau horaire qu'elle encapsule.

La méthode String[] getAvailableIDs() permet d'obtenir les noms des TimeZones définis en standard : par exemple avec Java 6, il y a 597 TimeZones fournis.

La classe est une fabrique qui permet d'obtenir une instance de TimeZone à partir de son identifiant en utilisant la méthode getTimeZone() ou celle correspondant à la Locale courante en utilisant la méthode getDefault().

112.1.5. La classe java.text.DateFormat

La clase abstraite DateFormat propose les fonctionnalités de base pour interpréter et formater une date sous la forme d'une chaîne de caractères.

Ce formatage doit traduire certains éléments notamment le jour et le mois de la date selon la Locale. De nombreux formats de dates sont aussi utilisés généralement dépendant eux aussi de la Locale.

Quatre styles de formats sont définis par défaut : SHORT, MEDIUM, LONG, et FULL. Avec une Locale et un style, la classe DateFormat peut fournir un formatage standard de la date.

La classe DateFormat propose plusieurs méthodes statiques getXXXlnstance() qui sont des fabriques renvoyant des instances de type DateFormat.

La méthode format() permet de formater une date en chaîne de caractères.

La méthode parse() permet d'extraire une date à partir de sa représentation sous la forme d'une chaîne de caractères.

La Locale et le style de la classe DateFormat ne peuvent pas être modifiés après la création de son instance.

112.1.6. La classe java.util.SimpleDateFormat

La classe SimpleDateFormat permet de formater et d'analyser une date en tenant compte d'une Locale. Elle hérite de la classe abstraite DateFormat.

Pour réaliser ces traitements, cette classe utilise un modèle (pattern) sous la forme d'une chaîne de caractères.

La classe DataFormat propose plusieurs méthodes pour obtenir le modèle par défaut de la Locale courante :

- getTimeInstance(style)
- getDateInstance(style)
- getDateTimeInstance(styleDate, styleHeure)

Ces méthodes utilisent la Locale par défaut mais chacune de ces méthodes possède une surcharge qui permet de préciser une Locale.

Pour chacune de ces méthodes, quatre styles sont utilisables : SHORT, MEDIUM, LONG et FULL. Ils permettent de désigner la richesse des informations contenues dans le modèle pour la date et/ou l'heure.

Exemple :

```
package fr.jmdoudoux.dej.date;

import java.text.DateFormat;
import java.util.Date;
import java.util.Locale;

public class TestFormaterDate2 {

  /**
   * @param args
   */
  public static void main(String[] args) {
    Date aujourdhui = new Date();

    DateFormat shortDateFormat = DateFormat.getDateTimeInstance(
        DateFormat.SHORT,
        DateFormat.SHORT);

    DateFormat shortDateFormatEN = DateFormat.getDateTimeInstance(
        DateFormat.SHORT,
        DateFormat.SHORT, new Locale("EN","en"));

    DateFormat mediumDateFormat = DateFormat.getDateTimeInstance(
        DateFormat.MEDIUM,
        DateFormat.MEDIUM);

    DateFormat mediumDateFormatEN = DateFormat.getDateTimeInstance(
        DateFormat.MEDIUM,
        DateFormat.MEDIUM, new Locale("EN","en"));

    DateFormat longDateFormat = DateFormat.getDateTimeInstance(
        DateFormat.LONG,
        DateFormat.LONG);

    DateFormat longDateFormatEN = DateFormat.getDateTimeInstance(
        DateFormat.LONG,
        DateFormat.LONG, new Locale("EN","en"));

    DateFormat fullDateFormat = DateFormat.getDateTimeInstance(
        DateFormat.FULL,
        DateFormat.FULL);

    DateFormat fullDateFormatEN = DateFormat.getDateTimeInstance(
        DateFormat.FULL,
        DateFormat.FULL, new Locale("EN","en"));

    System.out.println(shortDateFormat.format(aujourdhui));
    System.out.println(mediumDateFormat.format(aujourdhui));
    System.out.println(longDateFormat.format(aujourdhui));
    System.out.println(fullDateFormat.format(aujourdhui));
    System.out.println("");
    System.out.println(shortDateFormatEN.format(aujourdhui));
    System.out.println(mediumDateFormatEN.format(aujourdhui));
    System.out.println(longDateFormatEN.format(aujourdhui));
    System.out.println(fullDateFormatEN.format(aujourdhui));
  }

}
```

Résultat :

```
27/06/06 21:36
27 juin 2006 21:36:30
27 juin 2006 21:36:30 CEST
mardi 27 juin 2006 21 h 36 CEST

6/27/06 9:36 PM
Jun 27, 2006 9:36:30 PM
June 27, 2006 9:36:30 PM CEST
Tuesday, June 27, 2006 9:36:30 PM CEST
```

Il est aussi possible de définir son propre format en utilisant les éléments du tableau ci-dessous. Chaque lettre du tableau est interprétée de façon particulière. Pour utiliser les caractères sans qu'ils soient interprétés dans le modèle il faut les encadrer par de simples quotes. Pour utiliser une quote il faut en mettre deux consécutives dans le modèle.

Lettre	Description	Exemple
G	Era	AD (Anno Domini), BC (Before Christ)
y	Année	06 ; 2006
M	Mois dans l'année	Septembre; Sept.; 07
w	Semaine dans l'année	34
W	Semaine dans le mois	2
D	Jour dans l'année	192
d	jour dans le mois	23
F	Jour de la semaine dans le mois	17
E	Jour de la semaine	Mercredi; Mer.
a	Marqueur AM/PM (Ante/Post Meridiem)	PM, AM
H	Heure (0-23)	23
k	Heure (1-24)	24
K	Heure en AM/PM (0-11)	6
h	Heure en AM/PM (1-12)	7
m	Minutes	59
s	Secondes	59
S	Millisecondes	12564
z	Zone horaire générale	CEST; Heure d'été d'Europe centrale
Z	Zone horaire (RFC 822)	+0200

Ces caractères peuvent être répétés pour préciser le format à utiliser :

- Pour les caractères de type Text : moins de 4 caractères consécutifs représentent la version abrégée sinon c'est la version longue qui est utilisée.
- Pour les caractères de type Number : c'est le nombre de répétitions qui désigne le nombre de chiffres utilisés, complété si nécessaire par des 0 à gauche.
- Pour les caractères de type Year : 2 caractères précisent que l'année est codée sur deux caractères.
- Pour les caractères de type Month : 3 caractères ou plus représentent la forme littérale sinon c'est la forme numérique du mois.

Exemple :

```
package fr.jmdoudoux.dej.date;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Locale;

public class TestFormaterDate {

  public static void main(String[] args) {
    SimpleDateFormat formater = null;

    Date aujourdhui = new Date();

    formater = new SimpleDateFormat("dd-MM-yy");
    System.out.println(formater.format(aujourdhui));
    formater = new SimpleDateFormat("ddMMyy");
    System.out.println(formater.format(aujourdhui));

    formater = new SimpleDateFormat("yyMMdd");
    System.out.println(formater.format(aujourdhui));

    formater = new SimpleDateFormat("h:mm a");
    System.out.println(formater.format(aujourdhui));

    formater = new SimpleDateFormat("K:mm a, z");
    System.out.println(formater.format(aujourdhui));

    formater = new SimpleDateFormat("hh:mm a, zzzz");
    System.out.println(formater.format(aujourdhui));

    formater = new SimpleDateFormat("EEEE, d MMM yyyy");
    System.out.println(formater.format(aujourdhui));

    formater = new SimpleDateFormat("'le' dd/MM/yyyy 'à' hh:mm:ss");
    System.out.println(formater.format(aujourdhui));

    formater = new SimpleDateFormat("'le' dd MMMM yyyy 'à' hh:mm:ss");
    System.out.println(formater.format(aujourdhui));

    formater = new SimpleDateFormat("dd MMMMM yyyy GGG, hh:mm aaa");
    System.out.println(formater.format(aujourdhui));

    formater = new SimpleDateFormat("yyyyMMddHHmmss");
    System.out.println(formater.format(aujourdhui));

  }

}
```

Résultat :

```
27-06-06
270606
060627
9:37 PM
9:37 PM, CEST
09:37 PM, Heure d'été d'Europe centrale
mardi, 27 juin 2006
le 27/06/2006 à 09:37:10
le 27 juin 2006 à 09:37:10
27 juin 2006 ap. J.-C., 09:37 PM
20060627213710
```

Il existe plusieurs constructeurs de la classe SimpleDateFormat :

Constructeur	Rôle
SimpleDateFormat()	Constructeur par défaut utilisant le modèle par défaut et les symboles de formatage de dates de la Locale par défaut
SimpleDateFormat(String)	Constructeur utilisant le modèle fourni et les symboles de formatage de dates de la Locale par défaut
SimpleDateFormat(String, DateFormatSymbols)	Constructeur utilisant le modèle et les symboles de formatage de dates fournis
SimpleDateFormat(String, Locale)	Constructeur utilisant le modèle fourni et les symboles de formatage de dates de la Locale fournie

La classe DateFormatSymbols encapsule les différents éléments textuels qui peuvent entrer dans la composition d'une date pour une Locale donnée (les jours, les libellés courts des mois, les libellés des mois, ...).

Exemple :

```
package fr.jmdoudoux.dej.date;

import java.text.DateFormatSymbols;
import java.util.Locale;

public class TestFormaterDate3 {

  public static void main(String[] args) {
    DateFormatSymbols dfsFR = new DateFormatSymbols(Locale.FRENCH);
    DateFormatSymbols dfsEN = new DateFormatSymbols(Locale.ENGLISH);

    String[] joursSemaineFR = dfsFR.getWeekdays();
    String[] joursSemaineEN = dfsEN.getWeekdays();

    StringBuffer texteFR = new StringBuffer("Jours FR ");
    StringBuffer texteEN = new StringBuffer("Jours EN ");

    for (int i = 1; i < joursSemaineFR.length; i++) {
      texteFR.append(" : ");
      texteFR.append(joursSemaineFR[i]);
      texteEN.append(" : ");
      texteEN.append(joursSemaineEN[i]);
    }
    System.out.println(texteFR);
    System.out.println(texteEN);

    texteFR = new StringBuffer("Mois courts FR ");
    texteEN = new StringBuffer("Mois courts EN ");
    String[] moisCourtsFR = dfsFR.getShortMonths();
    String[] moisCourtsEN = dfsEN.getShortMonths();

    for (int i = 0; i < moisCourtsFR.length - 1; i++) {
      texteFR.append(" : ");
      texteFR.append(moisCourtsFR[i]);
      texteEN.append(" : ");
      texteEN.append(moisCourtsEN[i]);
    }

    System.out.println(texteFR);
    System.out.println(texteEN);

    texteFR = new StringBuffer("Mois FR ");
    texteEN = new StringBuffer("Mois EN ");
    String[] moisFR = dfsFR.getMonths();
    String[] moisEN = dfsEN.getMonths();

    for (int i = 0; i < moisFR.length - 1; i++) {
      texteFR.append(" : ");
      texteFR.append(moisFR[i]);
      texteEN.append(" : ");
      texteEN.append(moisEN[i]);
    }

    System.out.println(texteFR);
    System.out.println(texteEN);

  }

}
```

Résultat :

```
Jours FR  : dimanche : lundi : mardi : mercredi : jeudi : vendredi : samedi
Jours EN  : Sunday : Monday : Tuesday : Wednesday : Thursday : Friday : Saturday
Mois courts FR  : janv. : févr. : mars : avr. : mai : juin : juil. : août : sept. : oct.
 : nov. : déc.
Mois courts EN  : Jan : Feb : Mar : Apr : May : Jun : Jul : Aug : Sep : Oct : Nov : Dec
Mois FR  : janvier : février : mars : avril : mai : juin : juillet : août : septembre :
octobre : novembre : décembre
Mois EN  : January : February : March : April : May : June : July : August : September :
 October : November : December
```

Il est possible de définir son propre objet DateFormatSymbols pour personnaliser les éléments textuels nécessaires au traitement des dates. La classe DateFormatSymbols propose à cet effet des setters sur chacun des éléments.

Exemple :

```
package fr.jmdoudoux.dej.date;

import java.text.DateFormatSymbols;
import java.text.SimpleDateFormat;
import java.util.Date;

public class TestFormaterDate4 {

  public static void main(String[] args) {
    Date aujourdhui = new Date();
    DateFormatSymbols monDFS = new DateFormatSymbols();
    String[] joursCourts = new String[] {
        "",
        "Di",
        "Lu",
        "Ma",
        "Me",
        "Je",
        "Ve",
        "Sa" };
    monDFS.setShortWeekdays(joursCourts);
    SimpleDateFormat dateFormat = new SimpleDateFormat(
        "EEE dd MMM yyyy HH:mm:ss",
        monDFS);
    System.out.println(dateFormat.format(aujourdhui));
  }

}
```

Résultat :

```
Ma 27 juin 2006 21:38:22
```

Attention : il faut consulter la documentation de l'API pour connaître précisément le contenu et l'ordre des éléments fournis sous la forme de tableaux aux setters de la classe. Dans l'exemple, ci-dessus, les jours de la semaine commencent par Dimanche.

La méthode applyPattern() permet de modifier le modèle d'un objet SimpleDateFormat.

La classe SimpleDataFormat permet également d'analyser une date sous la forme d'une chaîne de caractères pour la transformer en objet de type Date en utilisant un modèle. Cette opération est réalisée grâce à la méthode parse(). Si elle échoue, elle lève une exception de type ParseException.

Exemple :

```
package fr.jmdoudoux.dej.date;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class TestParserDate {

  public static void main(String[] args) {
    Date date = null;
    SimpleDateFormat simpleDateFormat = new SimpleDateFormat("dd/MM/yyyy");

    String date1 = "22/06/2006";
    String date2 = "22062006";

    try {
      date = simpleDateFormat.parse(date1);
      System.out.println(date);
      date = simpleDateFormat.parse(date2);
      System.out.println(date);
    } catch (ParseException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
Thu Jun 22 00:00:00 CEST 2006
java.text.ParseException: Unparseable date: "22062006"
        at java.text.DateFormat.parse(Unknown Source)
        at fr.jmdoudoux.dej.date.TestParserDate.main(TestParserDate.java:19)
```

112.1.7. Les classes java.sql.Date, java.sql.Time, java.sql.TimeStamp

Ces trois classes héritent de la classe java.util.Date pour encapsuler des données correspondant aux types DATE, TIME et TIMESTAMP de la norme SQL 92.

La classe java.sql.Date n'encapsule que la partie date en ignorant la partie horaire du point dans le temps qu'elle encapsule.

La classe java.sql.Time, elle, n'encapsule que la partie horaire en ignorant la partie date du point dans le temps qu'elle encapsule.

La classe java.sql.TimeStamp encapsule encapsule un instant exprimé en millisecondes et des informations permettant une expression de cet instant avec une précision à la nanoseconde.

Ces trois méthodes redéfinissent la méthode toString() pour permettre une représentation respectant le standard SQL 92.

Exemple :

```
final java.sql.Date dateSQL = new java.sql.Date(new Date().getTime()) ;
System.out.println(dateSQL);
```

Remarque : ces trois classes ne permettent pas de prendre en compte un TimeZone explicite puisque généralement c'est celui de la base de données qui est toujours utilisé par défaut.

112.2. Des exemples de manipulations de dates

Cette section présente des portions de code pour répondre à des besoins courants de manipulations de dates.

Formater une date :

Exemple :

```
protected static final SimpleDateFormat dateFormat =
  new SimpleDateFormat("dd/MM/yyyy");
protected static final SimpleDateFormat dateHeureFormat =
  new SimpleDateFormat("dd/MM/yyyy hh:mm:ss");

public static String formatterDate(Date date) {
  return dateFormat.format(date);
}
public static String formatterDateHeure(Date date) {
  return dateFormatHeure.format(date);
}
```

Extraire une date d'une chaîne de caractères :

Exemple :

```
DateFormat df = new SimpleDateFormat("dd-MM-yyyy");
Date date=null;
try {
  date= df.parse("25-12-2010");
} catch (ParseException e){
  e.printstacktrace();
}
```

Ajouter/retrancher des jours à une date :

Exemple :

```
public static Date ajouterJour(Date date, int nbJour) {
  Calendar cal = Calendar.getInstance();
  cal.setTime(date.getTime();
  cal.add(Calendar.DATE, nbJour);
  return cal.getTime();
}
```

ou

Exemple :

```
public static Date ajouterJour(Date date, int nbJour) {
  Calendar cal = Calendar.getlnstance();
  cal.setTime(date.getTime();
  cal.add(Calendar.DAY_OF_MONTH, nbJour);
  return cal.getTime();
}
```

Pour retrancher des jours, il faut fournir un paramètre négatif au nombre de jours.

Ajouter/retrancher des mois à une date :

Exemple :

```
public static Date ajouterMois(Date date, int nbMois) {
  Calendar cal = Calendar.getInstance();
  cal.setTime(date.getTime();
  cal.add(Calendar.MONTH, nbMois);
  return cal.getTime();
}
```

Pour retrancher des mois, il faut fournir un paramètre négatif au nombre de mois.

Ajouter/retrancher des années à une date :

Exemple :

```
public static Date ajouterAnnee(Date date, int nbAnnee) {
  Calendar cal = Calendar.getInstance();
  cal.setTime(date.getTime());
  cal.add(Calendar.YEAR, nbAnnee);
  return cal.getTime();
}
```

Pour retrancher des années, il faut fournir un paramètre négatif au nombre d'années.

Ajouter/retrancher des heures à une date :

Exemple :

```
public static Date ajouterHeure(Date date, int nbHeure) {
  Calendar cal = Calendar.getInstance();
  cal.setTime(date.getTime());
  cal.add(Calendar.HOUR, nbHeure);
  return cal.getTime();
}
```

Pour retrancher des heures, il faut fournir un paramètre négatif au nombre d'heures.

Ajouter/retrancher des minutes à une date :

Exemple :

```
public static Date ajouterMinute(Date date, int nbMinute) {
  Calendar cal = Calendar.getInstance();
  cal.setTime(date.getTime());
  cal.add(Calendar.MINUTE, nbMinute);
  return cal.getTime();
}
```

Pour retrancher des minutes, il faut fournir un paramètre négatif au nombre de minutes.

Ajouter/retrancher des secondes à une date :

Exemple :

```
public static Date ajouterSeconde(Date date, int nbSeconde) {
  Calendar cal = Calendar.getlnstance();
  cal.setTime(date.getTime());
  cal.add(Calendar.SECOND, nbSeconde);
  return cal.getTime();
}
```

Pour retrancher des secondes, il faut fournir un paramètre négatif au nombre de secondes.

112.3. La classe SimpleDateFormat

La classe SimpleDateFormat permet de formater une date pour lui donner une représentation textuelle dans un format donné ou de parser une chaîne de caractères pour extraire une date dans un format donné.

112.3.1. L'utilisation de la classe SimpleDateFormat

Le constructeur de la classe SimpleDateFormat attend en paramètre une chaîne de caractères qui précise le format à utiliser durant les traitements de formatage et de parsing.

La méthode format() permet de formater la date fournie en paramètre.

Exemple :

```
SimpleDateFormat simpleDateFormat = new SimpleDateFormat("dd/MM/yyyy");
String dateStr = simpleDateFormat.format(new Date());
System.out.println(dateStr);
```

Le format comporte de nombreuses options et peut même contenir du texte brut qui doit être échappé avec des quotes simples.

Par défaut, la classe SimpleDateFormat travail avec la Locale courante. Il est possible de préciser une autre Locale en tant que paramètre du constructeur.

Exemple :

```
SimpleDateFormat simpleDateFormat = new SimpleDateFormat("dd MMMM yyyy zzzz G", Locale.FRENCH);
String dateStr = simpleDateFormat.format(new Date());
System.out.println(dateStr);
```

La méthode parse() permet de déterminer une date extraite d'une chaîne de caractères en utilisant un format donné.

Exemple :

```
SimpleDateFormat simpleDateFormat = new SimpleDateFormat("dd/MM/yyyy");
Date date = simpleDateFormat.parse("25/12/2010");
System.out.println(date);
```

Par défaut, SimpleDateFormat travaille avec la Locale par défaut qui contient le fuseau horaire (time zone).

Si la chaîne de caractères ne contient pas explicitement le fuseau horaire, il peut être nécessaire de le préciser en utilisant la méthode setTimeZone() :

Exemple :

```
SimpleDateFormat simpleDateFormat = new SimpleDateFormat("dd/MM/yyyy");
simpleDateFormat.setTimeZone(TimeZone.getTimeZone("PST"));
Date date = simpleDateFormat.parse("25/12/2010");
System.out.println(date);
```

Il est possible de préciser le siècle si la date à parser ne contient que deux chiffres : par exemple "01/01/02" peut correspondre à une date de l'année 1902 ou 2002. La méthode set2DigitYearStart() permet de préciser la date de début de la plage de 100 ans dans laquelle l'année sera traitée. Par défaut, cette plage de 100 ans correspond à la date du jour - 80 ans jusqu'à la date du jour + 20 ans.

Exemple :

```
SimpleDateFormat simpleDateFormat = new SimpleDateFormat("dd MMMM yy", Locale.FRENCH);
Date date = simpleDateFormat.parse("25-12-02");
System.out.println(date);
Date debut20emeSiecle = new GregorianCalendar(1901,1,1).getTime();
simpleDateFormat.set2DigitYearStart(debut20emeSiecle);
date = simpleDateFormat.parse("25-12-02");
System.out.println(date);
```

Par défaut, le parsing de la date est très permissif : le format de la date n'a pas à respecter strictement le format fourni à SimpleDateFormat. Dans ce cas, sans générer d'erreur, SimpleDateFormat va tenter d'extraire une date qui potentiellement ne correspond pas du tout.

Exemple :

```
SimpleDateFormat simpleDateFormat = new SimpleDateFormat("dd-MM-yyyy", Locale.FRENCH);
Date date = simpleDateFormat.parse("31-04-10");
System.out.println(date);
```

Dans l'exemple ci-dessus, le mois d'avril ne possède que 30 jours. La classe SimpleDateFormat en déduit que l'on veut le jour suivant le 30 avril soit le 1er mai. Ce comportement est rarement celui souhaité.

Pour demander un respect strict du format, il faut passer la valeur false à la méthode setLenient(). Si le format de la date à traiter ne correspond pas, une exception est levée.

Exemple :

```
SimpleDateFormat simpleDateFormat =
new SimpleDateFormat("dd-MM-yyyy", Locale.FRENCH);
simpleDateFormat.setLenient(false);
Date date = simpleDateFormat.parse("31-04-10");
System.out.println(date);
```

La classe SimpleDateFormat n'est pas thread-safe car elle maintient son état, entre autre, avec deux objets de type Calendar et NumberFormat. Si deux threads utilisent la même instance pour manipuler deux dates en même temps, le résultat des traitements est aléatoire : généralement il est erroné par rapport à la date traitée ce qui conduit à une corruption des données qui n'est pas facilement détectable ou, plus rarement, une exception est levée.

L'utilisation d'une même instance de SimpleDateFormat dans un contexte multithreads implique donc qu'il est nécessaire de prendre des précautions : le résultat peut être aléatoire lors du parsing et du formatage d'une date :

- tout peut bien se passer
- le résultat peut être erroné
- une exception peut être levée

112.3.2. Les points faibles de la classe SimpleDateFormat

La classe SimpleDateFormat présente deux faiblesses lors de sa mise en oeuvre :

- son instanciation est très coûteuse
- ses traitements ne sont pas threadsafe

Exemple :

```
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class DateUtil {

  public static final Date parse(String date) throws ParseException{
    SimpleDateFormat simpleDateFormat = new SimpleDateFormat("dd-MM-yyyy");
    return simpleDateFormat.parse(date);
  }

  public static final String format(Date date) throws ParseException{
    SimpleDateFormat simpleDateFormat = new SimpleDateFormat("dd-MM-yyyy");
    return simpleDateFormat.format(date);
  }
}
```

Cette solution est threadsafe mais son inconvénient est qu'elle peut requérir de nombreuses ressources si le nombre d'invocations est important.

Pour pallier ce premier souci, il est possible de créer une instance de classe statique qui permettra de n'avoir qu'un seul objet.

Exemple :

```
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class DateUtil {
  public static final SimpleDateFormat simpleDateFormat = new SimpleDateFormat("dd-MM-yyyy");

  public static final Date parse(String date) throws ParseException{
    return simpleDateFormat.parse(date);
  }

  public static final String format(Date date) throws ParseException{
    return simpleDateFormat.format(date);
  }
}
```

Cette solution fréquemment utilisée permet de réduire le nombre d'instances créées. Malheureusement, comme indiqué dans la JavaDoc, elle ne fonctionne pas dans un environnement multithread puisque la classe SimpleDateFormat n'est pas threadsafe. L'utilisation de la classe ci-dessus dans un contexte multithread peut donner des résultats aléatoires.

Cependant ces résultats aléatoires ne sont pas faciles à détecter dans une application car il faut que plusieurs threads sollicitent en même temps l'instance de SimpleDateFormat.

Exemple :

```
import java.text.ParseException;

public class TestSimpleDateFormat {

  public static void main(String[] args) {
    final String[] dates = new String[] {"15-01-2000", "28-02-2005", "20-04-2005",
      "31-07-2015" };

    Runnable runnable = new Runnable() { public void run() {
    try {
      for (int j = 0; j < 1000; j++) {
        for (int i = 0; i < 2; i++) {
          String date = DateUtil.format(DateUtil.parse(dates[i]));
          if (!(dates[i].equals(date))) {
            throw new ParseException(dates[i] + " =>"+ date, 0);
          }
        }
      }
    } catch (ParseException e) {
      e.printStackTrace();
    }

    new Thread(runnable).start();

    Runnable runnable2 = new Runnable() {
      public void run() {
        try {
          for (int j = 0; j < 1000; j++) {
            for (int i = 0; i < 2; i++) {
              String date = DateUtil.format(DateUtil.parse(dates[i]));
              if (!(dates[i].equals(date))) {
                throw new ParseException(dates[i] + " =>"+ date, 0);
              }
            }
          }
        } catch (ParseException e) {
          e.printStackTrace();
        }
      }
    };
    new Thread(runnable2).start();
  }
}
```

Dans cet exemple, le nombre d'exceptions et d'anomalies de traitement est important car les threads utilisent en permanence le même objet. Dans la réalité, par exemple dans une application web, les exceptions et les dates erronées sont très rares. L'ennui avec les erreurs de formatage et de parsing c'est qu'elles sont difficiles à détecter.

Il est possible de sécuriser l'utilisation de l'instance de SimpleDateFormat en l'entourant d'un bloc synchronized dont le moniteur est l'instance de la classe SimpleDateFormat ou en définissant les méthodes utilisant l'instance synchronized. Ainsi, un seul thread pourra accéder à l'instance à la fois.

Exemple :

```
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class DateUtil {

  public static final SimpleDateFormat simpleDateFormat = new SimpleDateFormat("dd-MM-yyyy");

  public synchronized static final Date parse(String date) throws ParseException{
    return simpleDateFormat.parse(date);
  }

  public synchronized static final String format(Date date) throws ParseException{
    return simpleDateFormat.format(date);
  }
}
```

Cette solution simple est thread-safe mais elle peut impliquer de la contention liée au verrou posé lors de l'exécution de la méthode qui bloque l'invocation par d'autres threads.

Une autre solution est d'utiliser la classe ThreadLocal qui est capable de fournir une instance pour le thread en cours, ainsi chaque thread peut avoir sa propre instance.

Exemple :

```
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class DateUtil {

  private static ThreadLocal<SimpleDateFormat> format = new ThreadLocal<SimpleDateFormat>() {
    protected synchronized SimpleDateFormat initialValue() {
      return new SimpleDateFormat("dd-MM-yyyy");
    }
  };

  public static final Date parse(String date) throws ParseException{
    return format.get().parse(date);
  }

  public static final String format(Date date) throws ParseException{
    return format.get().format(date); }
  }
```

Remarque : selon l'implémentation fournie de la classe ThreadLocal par le JRE, il peut y avoir des fuites de mémoire lors du redéploiement de l'application dans un conteneur web.

Il peut être intéressant d'utiliser une SoftReference en paramètre du ThreadLocal pour améliorer la gestion de la mémoire par la JVM.

Exemple :

```
import java.lang.ref.SoftReference;
import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class DateUtil {
  private static final ThreadLocal<SoftReference<DateFormat» format =
    new ThreadLocal<SoftReference<DateFormat>>();

  private static DateFormat getDateFormat() {
    SoftReference<DateFormat>      softRef = format.get();
    if (softRef != null) {
      final DateFormat result = softRef.get();
      if (result != null) {
        return result;
      }
    }
    final DateFormat result = new SimpleDateFormat("dd-MM-yyyy");
    softRef = new SoftReference<DateFormat>(result);
    format.set(softRef);
    return result;
  }

  public static final Date parse(final String date) throws ParseException {
    return getDateFormat().parse(date);
  }

  public static final String format(final Date date) throws ParseException {
    return getDateFormat().format(date);
  }
}
```

Cette approche nécessite de recréer l'instance locale de SimpleDateFormat dans le cas où le ramasse-miettes aurait détruit la précédente.

Une autre solution peut être d'utiliser une API tierce telle que :

- Joda Time : en remplaçant java.text.SimpleDateFormat par org.joda.time.format.DateTimeFormatter
- Apache Jakarta Common Lang : utiliser la classe FastDateFormat. Malheureusement cette classe ne permet que de formater mais pas de parser une date.

Lors de la mise en oeuvre de la classe SimpleDateFormat, il faut aussi être vigilent car par défaut, la classe SimpleDateFormat est très permissive : elle tente au mieux de faire correspondre la date selon le format fourni, ce qui peut conduire à un comportement non souhaité et surtout à des résultats indésirables.

Exemple :

```
import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class TestDate {

  public static void main(final String[] args) {
    final DateFormat df = new SimpleDateFormat("yyyyMMddHHmmss");
    Date d;
    try {
      d = df.parse("2010-01-15 07:23:30");
      System.out.println(d);
    } catch (final ParseException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
Mon Nov 30 23:05:07 CET 2009
```

Pour que la classe SimpleDateFormat respecte strictement le format fourni et lève une exception de type java.text.ParseException, il faut invoquer la méthode setLenient() en lui passant la valeur false en paramètre.

Exemple :

```
import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class TestDate {

  public static void main(final String[] args) {
    final DateFormat df = new SimpleDateFormat("yyyyMMddHHmmss");
    df.setLenient(false);
    Date d;
    try {
      d = df.parse("2010-01-15 07:23:30");
      System.out.println(d);
    } catch (final ParseException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
java.text.ParseException: Unparseable date: "2010-01-15 07:23:30"
at java.text.DateFormat.parse(DateFormat.java:337) at TestDate.main(TestDate.java:39)
```

112.4. Joda Time

La plupart des applications ont besoin à un moment ou à un autre de manipuler des données de type date ou heure. Le JDK fournit des classes pour permettre ces manipulations, notamment les classes Date et Calendar, mais leur utilisation n'est pas simple et généralement source d'erreurs.

Joda Time est une bibliothèque open source dont le but est de fournir une solution simple et complète pour manipuler des données de types date/heure.

Joda Time propose au travers de son API :

- Le support de plusieurs systèmes calendaires dont celui par défaut est celui définit par le standard ISO8601 (utilisé par XML) : Grégorien, Julien, Bouddhiste, Islamique, ...
- Le parsing et le formatage de dates
- Le support des fuseaux horaires
- Le support de plusieurs classes temporelles : date/heure locale, durée, période, intervalle, ...

Le but de Joda Time est de proposer une solution de remplacement aux classes de gestion des dates du JDK qui présentent plusieurs inconvénients :

- Il n'est pas facile à utiliser pour manipuler des données courantes
- La conception des classes Date et Calendar
- Les performances de certaines fonctionnalités sont plutôt mauvaises

Par exemple, Joda Time gère les mois de 1 à 12 dans son implémentation du calendrier Grégorien alors que la classe GregorianCalendar du JDK gère les mois de 0 à 11.

Joda Time a été développé pour améliorer la manière d'utiliser des données de type date/heure en mettant l'accent sur :

- la faciliter d'utilisation
- la fourniture d'un ensemble complet de fonctionnalités relatives aux traitements de données de type dates/heures
- l'extensibilité : Joda Time propose le support de plusieurs calendriers qui reposent sur la classe Chronology
- l'interopérabilité avec les classes correspondantes du JDK
- la performance
- la maturité

La version couverte dans cette section est la 2.1. Elle nécessite une version 1.5 ou supérieure du JDK.

La partie publique de l'API est contenue dans les packages org.joda.time et org.joda.time.format.

Joda Time utilise plusieurs concepts :

- instant (un point unique dans le temps)
- temps partiel (points multiples dans le temps)
- intervalle et durée
- système calendaire
- fuseau horaire

La classe JodaTimePermission peut être utilisée dans le mécanisme standard de sécurité de la JVM pour restreindre l'utilisation à certaines fonctionnalités globales de JodaTime.

L'API Joda Time a été utilisée comme une grande source d'inspiration pour la JSR 310.

112.4.1. Les principales classes de JodaTime

La plupart des classes de Joda Time sont immuables : les méthodes qui permettent d'effectuer des modifications les font sur une copie de l'objet qu'elles retournent.

Classe	Rôle
DateTime	Equivalent de la classe Calendar
DateMidnight	Classe immuable qui encapsule une date dont l'heure est forcée à minuit
LocalDate	Classe immuable qui encapsule une date locale (sans fuseau horaire)
LocalTime	Classe immuable qui encapsule une heure locale (sans fuseau horaire)
LocalDateTime	Classe immuable qui encapsule une date/heure locale (sans fuseau horaire)

112.4.2. Le concept d'Instant

Un instant est un point unique dans le temps dont la représentation est le nombre de millisecondes depuis le 1er janvier 1970 00 heure 00. Ceci rend un Instant compatible avec les classes Calendar et Date du JDK.

La représentation d'un instant en une date est dépendante du calendrier et du fuseau horaire utilisés pour représenter cet instant.

Un instant est défini par l'interface ReadableInstant.

112.4.2.1. L'interface ReadableInstant

L'interface ReadableInstant décrit les fonctionnalités d'un objet qui encapsule un instant.

Les implémentations de cette interface peuvent être immuables ou non.

Joda Time propose plusieurs classes qui implémentent l'interface ReadableInstant dont :

- Instant : une implémentation immuable qui encapsule un instant dans le temps sans utiliser de système calendaire ou de fuseau horaire particulier. Elle stocke en interne une valeur de type long qui contient le nombre de millisecondes écoulées depuis le 1er janvier 1970 à 00 heure 00.
- DateTime : encapsule une date/heure selon un système calendaire et un fuseau horaire donnés. C'est l'implémentation la plus fréquemment utilisée.
- DateMidnight : cette classe agit comme la classe DateTime excepté le fait que l'heure encapsulée est toujours minuit.
- MutableDateTime : cette classe agit comme la classe DateTime excepté le fait qu'elle ne soit pas immuable.

Attention : l'interface ReadableInstant n'est qu'un sous-ensemble des fonctionnalités des classes qui l'implémentent. Il est généralement préférable de typer les variables avec leur implémentation plutôt que de les typer avec l'interface ReadableInstant sauf si les fonctionnalités requises de l'instance sont définies dans l'interface.

Il est généralement recommandé d'utiliser dans la mesure du possible des implémentations qui soient immuables. L'objet ne peut ainsi pas être modifié sans créer une nouvelle instance, ce qui lui permet d'être thread safe.

Important : Joda Time considère qu'un instant null correspond à l'instant présent. Ainsi lorsqu'une méthode attend en paramètre un objet de ReadableInstant et que la valeur reçue en paramètre est null, alors cela revient à passer en paramètre un instant qui correspond à l'instant présent.

112.4.2.2. La classe DateTime

La classe DateTime encapsule un instant dans le temps pour un système calendaire et un fuseau horaire donné : ceux-ci lui permettent de restituer l'instant encapsulé sous la forme d'une date et d'une heure.

Par défaut, une instance de type DataTime utilise le système calendaire ISOChronology et le fuseau horaire obtenu du système. De nombreux constructeurs attendent en paramètre un objet de type Chronology et/ou DateTimeZone qui permettent de préciser le système calendaire et /ou le fuseau horaire à utiliser.

Le constructeur sans paramètre crée une instance qui encapsule l'instant courant représenté dans le système calendaire ISO et le fuseau horaire par défaut.

Exemple :

```
    DateTime datetime = new DateTime();
```

Plusieurs constructeurs permettent de préciser les éléments de la date/heure encapsulée : année, mois, jour, heure, minute, seconde.

Exemple :

```
    DateTime datetime = new DateTime(2012,12,25,0,0,0);
```

La classe DateTime propose plusieurs autres constructeurs qui acceptent une instance de type Object comme valeur pour représenter la date/heure. Ces surcharges permettent à Joda Time d'être extensible mais en sacrifiant le typage fort.

Par défaut, la classe ConverterManager permet de gérer les différents types supportés :

- ReadableInstant
- String : une chaîne de caractères qui contient la date au format ISO-8601
- java.util.Calendar : dans ce cas, le système calendaire encapsulé est utilisé
- java.util.Date et javax.sql.Date
- Long : un nombre de millisecondes
- null : est interprété par Joda Time comme l'instant présent

Exemple :

```
    java.util.Date date = new Date();
    long timeEnMs = date.getTime();
    DateTime dateTime = new DateTime(timeEnMs);
```

Exemple :

```
    java.util.Date date = new Date();
    DateTime dateTime = new DateTime(date);
```

Exemple :

```
    java.util.Calendar calendar = Calendar.getInstance();
    calendar.setTime(new Date());
    DateTime dateTime = new DateTime(calendar);
```

Exemple :

```
    String timeString = "2012-12-25";
    DateTime dateTime = new DateTime(timeString);
```

Exemple :

```
    DateTime dt = new DateTime("2012-10-28T16:23:13.324+01:00");
```

Il est ainsi facile de convertir une instance de type java.util.Date ou java.util.Calendar en un objet de type DateTime simplement en passant l'instance au constructeur de la classe DateTime.

A l'exécution de l'exemple ci-dessous une exception de type IllegalArgumentException est levée avec le message No instant converter found for type: java.util.ArrayList car Joda Time ne peut pas convertir l'instance de type Object fournie en paramètre en un instant.

Exemple :

```
    List liste = new ArrayList();
    DateTime dateCourante = new DateTime(liste);
```

Plusieurs méthodes statiques permettent d'obtenir une instance de type DateTime.

Méthode	Rôle
static DateTime now()	Obtenir une instance de type DateTime qui encapsule la date/heure système courante en utilisant le système calendaire ISO et le fuseau horaire par défaut
static DateTime now(Chronology chronology)	Obtenir une instance de type DateTime qui encapsule la date/heure système courante en utilisant le système calendaire fourni en paramètre et le fuseau horaire par défaut
static DateTime now(DateTimeZone zone)	Obtenir une instance de type DateTime qui encapsule la date/heure système courante en utilisant le système calendaire ISO et le fuseau horaire fourni en paramètre
static DateTime parse(String str)	Extraire une date/heure de la chaîne de caractères fournie en paramètre
static DateTime parse(String str, DateTimeFormatter formatter)	Extraire une date/heure de la chaîne de caractères fournie en utilisant le formateur passé en paramètre

Les opérations de manipulations de date/heure encapsulées dans un objet de type DateTime peuvent être réalisées en invoquant des méthodes de DateTime ou en invoquant des méthodes sur les propriétés de l'objet DateTime. Cela rend ces opérations particulièrement pratiques et flexibles.

La classe DateTime encapsule une date/heure de manière immuable. Les méthodes qui permettent de modifier un élément de la date/heure encapsulée renvoie une nouvelle instance de type DateTime encapsulant le résultat de l'opération.

Méthode	Rôle
DateTime minus(long duration)	Renvoyer une nouvelle instance de DateTime dont la durée fournie a été soustraite
DateTime minus(ReadableDuration duration)	Renvoyer une nouvelle instance de DateTime dont la durée fournie a été soustraite

DateTime minus(ReadablePeriod period)	Renvoyer une nouvelle instance de DateTime dont la période fournie a été soustraite
DateTime minusDays(int days)	Renvoyer une nouvelle instance de DateTime dont le nombre de jours fourni a été soustrait
DateTime minusHours(int hours)	Renvoyer une nouvelle instance de DateTime dont le nombre d'heures fourni a été soustrait
DateTime minusMillis(int millis)	Renvoyer une nouvelle instance de DateTime dont le nombre de millisecondes fourni a été soustrait
DateTime minusMinutes(int minutes)	Renvoyer une nouvelle instance de DateTime dont le nombre de minutes fourni a été soustrait
DateTime minusMonths(int months)	Renvoyer une nouvelle instance de DateTime dont le nombre de mois fourni a été soustrait
DateTime minusSeconds(int seconds)	Renvoyer une nouvelle instance de DateTime dont le nombre de secondes fourni a été soustrait
DateTime minusWeeks(int weeks)	Renvoyer une nouvelle instance de DateTime dont le nombre de semaines fourni a été soustrait
DateTime minusYears(int years)	Renvoyer une nouvelle instance de DateTime dont le nombre d'années fourni a été soustrait
DateTime plus(long duration)	Renvoyer une nouvelle instance de DateTime dont la durée fournie a été ajoutée
DateTime plus(ReadableDuration duration)	Renvoyer une nouvelle instance de DateTime dont la durée fournie a été ajoutée
DateTime plus(ReadablePeriod period)	Renvoyer une nouvelle instance de DateTime dont la période fournie a été ajoutée
DateTime plusDays(int days)	Renvoyer une nouvelle instance de DateTime dont le nombre de jours fourni a été ajouté
DateTime plusHours(int hours)	Renvoyer une nouvelle instance de DateTime dont le nombre d'heures fourni a été ajouté
DateTime plusMillis(int millis)	Renvoyer une nouvelle instance de DateTime dont le nombre de millisecondes fourni a été ajouté
DateTime plusMinutes(int minutes)	Renvoyer une nouvelle instance de DateTime dont le nombre de minutes fourni a été ajouté
DateTime plusMonths(int months)	Renvoyer une nouvelle instance de DateTime dont le nombre de mois fourni a été ajouté
DateTime plusSeconds(int seconds)	Renvoyer une nouvelle instance de DateTime dont le nombre de secondes fourni a été ajouté
DateTime plusWeeks(int weeks)	Renvoyer une nouvelle instance de DateTime dont le nombre de semaines fourni a été ajouté
DateTime plusYears(int years)	Renvoyer une nouvelle instance de DateTime dont le nombre d'années fourni a été ajouté
DateMidnight toDateMidnight()	Convertir en une instance de type DateMidnight en utilisant le même système calendaire
LocalDate toLocalDate()	Convertir en une instance de type LocalDate en utilisant le même système calendaire
LocalDateTime toLocalDateTime()	Convertir en une instance de type LocalDateTime en utilisant le même système calendaire
DateTime withCenturyOfEra(int centuryOfEra)	Renvoyer une nouvelle instance de DateTime dont le siècle a été modifié

DateTime withChronology(Chronology newChronology)	Renvoyer une nouvelle instance de DateTime qui utilise le système calendaire fourni en paramètre
DateTime withDate(int year, int monthOfYear, int dayOfMonth)	Renvoyer une nouvelle instance de DateTime dont l'année, le mois et le jour ont été modifié
DateTime withDayOfMonth(int dayOfMonth)	Renvoyer une nouvelle instance de DateTime dont le jour du mois a été modifié
DateTime withDayOfWeek(int dayOfWeek)	Renvoyer une nouvelle instance de DateTime dont le jour de la semaine a été modifié
DateTime withDayOfYear(int dayOfYear)	Renvoyer une nouvelle instance de DateTime dont le jour de l'année a été modifié
DateTime withDurationAdded(long durationToAdd, int scalar)	Renvoyer une nouvelle instance de DateTime à la laquelle la durée a été ajoutée
DateTime withDurationAdded(ReadableDuration durationToAdd, int scalar)	Renvoyer une nouvelle instance de DateTime à la laquelle la durée a été ajoutée
DateTime withEra(int era)	Renvoyer une nouvelle instance de DateTime dont l'ère a été modifiée
DateTime withField(DateTimeFieldType fieldType, int value)	Renvoyer une nouvelle instance de DateTime dont la propriété fournie a été modifiée
DateTime withFieldAdded(DurationFieldType fieldType, int amount)	Renvoyer une nouvelle instance de DateTime dont la propriété fournie a été ajoutée
DateTime withHourOfDay(int hour)	Renvoyer une nouvelle instance de DateTime dont l'heure du jour a été modifiée
DateTime withMillis(long newMillis)	Renvoyer une nouvelle instance de DateTime dont le nombre de millisecondes a été modifié
DateTime withMillisOfDay(int millis)	Renvoyer une nouvelle instance de DateTime dont le nombre de millisecondes du jour a été modifié
DateTime withMillisOfSecond(int millis)	Renvoyer une nouvelle instance de DateTime dont le nombre de millisecondes courant a été modifié
DateTime withMinuteOfHour(int minute)	Renvoyer une nouvelle instance de DateTime dont le nombre de minutes a été modifié
DateTime withMonthOfYear(int monthOfYear)	Renvoyer une nouvelle instance de DateTime dont le mois a été modifié
DateTime withPeriodAdded(ReadablePeriod period, int scalar)	Renvoyer une nouvelle instance de DateTime à laquelle la période a été ajoutée
DateTime withSecondOfMinute(int second)	Renvoyer une nouvelle instance de DateTime dont le nombre de secondes a été modifié
DateTime withTime(int hourOfDay, int minuteOfHour, int secondOfMinute, int millisOfSecond)	Renvoyer une nouvelle instance de DateTime dont les heures, les minutes, les secondes et les millisecondes ont été modifiées
DateTime withYear(int year)	Renvoyer une nouvelle instance de DateTime dont l'année a été modifiée
DateTime withZone(DateTimeZone newZone)	Renvoyer une nouvelle instance de DateTime qui utilise le fuseau horaire fourni en paramètre sans modifier l'instant encapsulé
DateTime withZoneRetainFields(DateTimeZone newZone)	Renvoyer une nouvelle instance de DateTime qui utilise le fuseau horaire fourni en paramètre sans modifier les champs encapsulés

Exemple :

```
    DateTime dateCourante = new DateTime();
    DateTime dateLimite = dateCourante.plusDays(2);
```

La classe DateTime propose plusieurs solutions pour obtenir individuellement chacun des champs de la date/heure encapsulée :

- un getter pour chaque champs
- une méthode qui renvoie un objet de type DateTime.Property pour chaque champs
- la méthode property() qui attend en paramètre et renvoie un objet de type DateTime.Property

Les propriétés contenues dans un DateTime sont :

Propriété	Rôle
centuryOfEra	Le siècle
dayOfMonth	Le jour du mois
dayOfWeek	Le jour de la semaine
dayOfYear	Le jour de l'année
era	L'ère comme défini par le système calendaire
hourOfDay	L'heure
millisOfDay	Le nombre de millisecondes du jour
millisOfSecond	Le nombre de millisecondes de l'heure
minuteOfDay	Le nombre de minutes du jour
minuteOfHour	Le nombre de minutes
monthOfYear	Le mois
secondOfDay	Le nombre de secondes du jour
secondOfMinute	Le nombre de secondes
weekOfWeekyear	La semaine de l'année
weekYear	
Year	L'année
yearOfCentury	L'année du siècle
yearOfEra	

Exemple :

```
    DateTime dateTime = new DateTime();
    System.out.println(dateTime.getYear());
    System.out.println(dateTime.year().get());
    System.out.println(dateTime.property(DateTimeFieldType.year()).get());
```

La classe DateTime.Property encapsule la valeur d'un champ qui est un élément d'une DateTime.La classe DateTime.Property propose quelques getters :

Méthode	Rôle
Chronology getChronology()	Retourne le système calendaire du DateTime correspondant au champ
DateTime getDateTime()	Retourne l'instance de type DateTime correspondant au champ
DateTimeField getField()	Retourne le champ encapsulé

long getMillis()	Retourne le nombre de millisecondes du DateTime correspondant au champ

La classe DateTime propose plusieurs méthodes qui permettent de modifier la valeur du champ et de retourner une nouvelle instance de type DateTime encapsulant le résultat de la mise à jour.

Méthode	Rôle
DateTime addToCopy(int value)	Ajouter une valeur à la valeur de ce champ dans l'instance retournée
DateTime addToCopy(long value)	Ajouter une valeur à la valeur de ce champ dans l'instance retournée
DateTime setCopy(int value)	Modifier la valeur de ce champ dans l'instance retournée
DateTime setCopy(String text)	Modifier la valeur de ce champ dans l'instance retournée
DateTime setCopy(String text, Locale locale)	Modifier la valeur de ce champ dans l'instance retournée
DateTime withMaximumValue()	Forcer la valeur de ce champ à sa valeur maximale dans l'instance retournée
DateTime withMinimumValue()	Forcer la valeur de ce champ à sa valeur minimale dans l'instance retournée

Exemple :

```
    DateTime dateTime = new DateTime(new Date());
    System.out.println(dateTime);
    System.out.println("dayOfMonth     " + dateTime.dayOfMonth().get());
    System.out.println("dayOfWeek      " + dateTime.dayOfWeek().get());
    System.out.println("dayOfYear      " + dateTime.dayOfYear().get());
    System.out.println("ear            " + dateTime.era().get());
    System.out.println("hourOfDay      " + dateTime.hourOfDay().get());
    System.out.println("millisOfDay    " + dateTime.millisOfDay().get());
    System.out.println("millisOfSecond " + dateTime.millisOfSecond().get());
    System.out.println("minuteOfDay    " + dateTime.minuteOfDay().get());
    System.out.println("minuteOfHour   " + dateTime.minuteOfHour().get());
    System.out.println("monthOfYear    " + dateTime.monthOfYear().get());
    System.out.println("secondOfDay    " + dateTime.secondOfDay().get());
    System.out.println("secondOfMinute " + dateTime.secondOfMinute().get());
    System.out.println("weekOfWeekyear " + dateTime.weekOfWeekyear().get());
    System.out.println("weekyear       " + dateTime.weekyear().get());
    System.out.println("year           " + dateTime.year().get());
    System.out.println("yearOfCentury  " + dateTime.yearOfCentury().get());
    System.out.println("yearOfEra      " + dateTime.yearOfEra().get());
```

Résultat :

```
2012-11-08T06:56:46.781+01:00
dayOfMonth     8
dayOfWeek      4
dayOfYear      313
ear            1
hourOfDay      6
millisOfDay    25006781
millisOfSecond 781
minuteOfDay    416
minuteOfHour   56
monthOfYear    11
secondOfDay    25006
secondOfMinute 46
weekOfWeekyear 45
weekyear       2012
year           2012
yearOfCentury  12
yearOfEra      2012
```

Elle possède aussi de nombreuses méthodes héritées de la classe AbstractReadableInstantField.

Méthode	Rôle
int compareTo(ReadableInstant instant)	Comparer ce champ au champ correspondant de l'instant
int compareTo(ReadablePartial partial)	Comparer ce champ au champ correspondant de l'instant partiel
boolean equals(Object object)	Comparer ce champ à un autre
int get()	Obtenir la valeur du champ
String getAsShortText()	Obtenir la valeur textuelle courte du champ dans la locale par défaut.
String getAsShortText(Locale locale)	Obtenir la valeur textuelle du champ dans la locale fournie.
String getAsString()	Obtenir la valeur du champ sous la forme d'une chaîne de caractères
String getAsText()	Obtenir la valeur textuelle du champ dans la locale par défaut.
String getAsText(Locale locale)	Obtenir la valeur textuelle du champ dans la locale fournie.
int getDifference(ReadableInstant instant)	Obtenir la différence entre la valeur de champ et celle correspondante dans l'instant passé en paramètre
long getDifferenceAsLong(ReadableInstant instant)	Obtenir la différence entre la valeur de champ et celle correspondante dans l'instant passé en paramètre
DateTimeFieldType getFieldType()	Obtenir le type du champ
int getMaximumShortTextLength(Locale locale)	Obtenir la taille maximale de valeur textuelle courte pour ce champ
int getMaximumTextLength(Locale locale)	Obtenir la taille maximale de valeur textuelle pour ce champ
int getMaximumValue()	Obtenir la valeur maximale pour ce champ
int getMaximumValueOverall()	Obtenir la valeur maximale pour ce champ sans tenir compte de l'instant
protected abstract long getMillis()	Obtenir le nombre de millisecondes du DateTime
int getMinimumValue()	Obtenir la valeur minimale pour ce champ
int getMinimumValueOverall()	Obtenir la valeur minimale pour ce champ sans tenir compte de l'instant
String getName()	Obtenir le nom du champ
boolean isLeap()	Retourner un booléen qui indique si la valeur du champ est bissextile
String toString()	Obtenir une représentation textuelle orientée debug du champ

Exemple :

```
    DateTime dateTime = new DateTime(new Date());
    System.out.println("date = "+dateTime);
    System.out.println("nom du champ = "+dateTime.year().getName());
    System.out.println("mois EN = "+dateTime.monthOfYear().getAsText(Locale.ENGLISH));
    System.out.println("mois court = "+dateTime.monthOfYear().getAsShortText());
    System.out.println("est bissextile = "+dateTime.year().isLeap());
    System.out.println("jour rounded = "+dateTime.dayOfMonth().roundFloorCopy());
    System.out.println("mois rounded = "+dateTime.monthOfYear().roundFloorCopy());
    System.out.println("dayofWeek = "+dateTime.dayOfWeek().toString());
```

Résultat :

```
date = 2012-11-08T07:04:03.265+01:00
nom du champ = year
mois EN = November
mois court = nov.
est bissextile = true
jour rounded = 2012-11-08T00:00:00.000+01:00
mois rounded = 2012-11-01T00:00:00.000+01:00
dayofWeek = Property[dayOfWeek]
```

112.4.3. Le concept de Partial

Un instant partiel peut représenter plusieurs points dans le temps. Par exemple, le premier janvier existe chaque année dans le calendrier Grégorien. Un instant partiel est aussi pratique pour gérer des dates/heures locales (sans fuseau horaire) ou pour gérer uniquement des dates ou des heures.

L'interface ReadablePartial définit les fonctionnalités d'un objet qui encapsule une date partielle locale (pas de fuseau horaire). Il est possible de définir tout ou partie des champs de la date/heure encapsulée.

Il est parfois nécessaire de manipuler une partie d'une date et/ou d'une heure : par exemple uniquement le jour, le mois ou l'heure. Ce besoin est défini par l'interface ReadablePartial qui représente un instant partiellement défini.

Joda Time propose plusieurs classes qui implémentent l'interface ReadablePartial dont :

- Partial
- LocalDate
- LocalTime
- LocalDateTime

Il est possible de convertir une instance de type ReadablePartial en une instance de type ReadableInstant en utilisant la méthode toDateTime().

La classe LocalDate encapsule une date (année, mois, jour), sans heure et sans fuseau horaire de manière immuable.

La classe LocalDate propose plusieurs constructeurs.

Exemple :

```
    LocalDate localDate = new LocalDate(2012, 12, 25);
```

A partir de la version 1.3 de Joda Time, la classe LocalDate doit être utilisée à la place de la classe YearMonthDay qui est deprecated.

La classe LocalTime encapsule une heure (heure, minutes, secondes, millisecondes) sans fuseau horaire de manière immuable.

La classe LocalTime propose plusieurs constructeurs.

Exemple :

```
    LocalTime localTime = new LocalTime(17, 30, 45);
```

112.4.4. Les concepts d'intervalle, de durée et de période

Joda Time propose un support pour la gestion d'intervalles qui correspondent à une plage entre deux dates et de périodes de temps qui sont une durée grâce à trois classes : Interval, Period et Duration.

Les classes Interval et MutableInterval implémentent l'interface ReadableInterval.

112.4.4.1. La classe Interval

La classe Interval encapsule un intervalle entre deux instants de manière immuable. L'instant de début est inclus et l'instant de fin est exclu de l'intervalle. L'instant de fin doit donc être supérieur ou égal à l'instant de début.

Les deux instants doivent obligatoirement utiliser le même système calendaire et le même fuseau horaire.

La classe Interval propose plusieurs constructeurs.

Exemple :

```
    Interval interval = new Interval(
        new DateTime("2012-12-10"),
        new DateTime("2012-12-15"));
```

La méthode getStart() renvoie l'instant de début de l'intervalle. La méthode getEnd() renvoie l'instant de fin de l'intervalle.

La classe Interval propose plusieurs autres méthodes pour manipuler le contenu de l'intervalle.

Exemple :

```
    DateTime debut = new DateTime("2012-01-01");
    DateTime fin = debut.plus(Months.months(1));
    Interval interval = new Interval(debut, fin);
    System.out.println("Interval = " + interval);
    interval = interval.withEnd(interval.getEnd().plusMonths(1));
    System.out.println("Interval = " + interval);
```

Résultat :

```
Interval =
2012-01-01T00:00:00.000+01:00/2012-02-01T00:00:00.000+01:00
Interval = 2012-01-01T00:00:00.000+01:00/2012-03-01T00:00:00.000+01:00
```

La méthode contains() permet de déterminer si un Instant est inclus dans l'intervalle ou pas.

Exemple :

```
    Interval interval = new Interval(
        new DateTime("2012-12-10"),
        new DateTime("2012-12-15"));
    System.out.println(interval.contains(
        new DateTime(2012, 12, 9, 23, 59, 59, 999)));
    System.out.println(interval.contains(
        new DateTime(2012, 12, 10, 0, 0, 0, 0)));
    System.out.println(interval.contains(
        new DateTime(2012, 12, 14, 23, 59, 59, 999)));
    System.out.println(interval.contains(
        new DateTime(2012, 12, 15, 0, 0, 0, 0)));
```

Résultat :

```
false
true
true
false
```

La méthode toDuration() permet d'obtenir une instance de type Duration à partir de l'instance de type Interval.

Pour comparer deux instances de type Interval, il faut comparer leur durée.

112.4.4.2. La classe Period

Une période ne possède ni système calendaire ni fuseau horaire. Elle ne possède donc pas de représentation en millisecondes : il est nécessaire d'utiliser un Instant qui servira de référence et qui précisera le système calendaire et le fuseau horaire à utiliser pour y associer la période.

Par exemple, une période d'un mois ne correspond pas au même nombre de millisecondes si on l'ajoute au premier janvier ou au premier février. C'est aussi le cas si l'on ajoute une heure : ce ne sont pas forcément 60 minutes qui sont ajoutées selon le fuseau horaire et l'utilisation de l'heure d'été/d'hiver.

La classe Period encapsule une durée dont la valeur est constituée de champs qui expriment ses différentes unités.

Par défaut, les champs utilisables dans une Period (années, mois, semaines, jours, heures, minutes, secondes, millisecondes) sont définis dans une instance de la classe PeriodType. Il est possible de restreindre les champs utilisables en utilisant la classe PeriodType. La classe PeriodType propose plusieurs fabriques qui renvoient des instances de type PeriodType :

- Standard : années, mois, semaine, jours, heures, minutes, secondes, millisecondes (c'est l'instance par défaut)
- YearMonthDayTime : années, mois, jours, heures, minutes, secondes, millisecondes
- YearMonthDay : années, mois, jours
- YearWeekDayTime : années, semaines, jours, heures, minutes, secondes, millisecondes
- YearWeekDay : années, semaines, jours
- YearDayTime : années, jours, heures, minutes, secondes, millisecondes
- YearDay : années, jours, heures
- DayTime : jours, heures, minutes, secondes, millisecondes
- Time : heures, minutes, secondes, millisecondes
- et une fabrique pour chaque champ

JodaTime propose plusieurs classes qui encapsulent une valeur pour un des champs de manière immuable : Years, Weeks, Months, Days, Hours, Minutes, Seconds.

Ces classes implémentent l'interface Comparable et proposent quelques méthodes permettant de réaliser des opérations mathématiques de base sur les valeurs qu'elles encapsulent (plus(), multipliedBy(), dividedBy(), negated(), ...) et des opérations de comparaison (isGreaterThan(), isLesserThan()).

La classe Days encapsule un nombre de jours. Elle ne possède pas de constructeur public : pour obtenir une instance, il faut utiliser une des méthodes statiques qui sont des fabriques.

La méthode days() est une fabrique qui retourne une constante de type Days ou un instance selon le valeur fournie en paramètre.

La méthode daysBetween() permet d'obtenir une instance qui encapsule le nombre de jours entre deux Instant ou deux Partial.

La méthode daysIn() permet d'obtenir une instance qui encapsule le nombre de jours d'un Interval.

La classe Hours encapsule un nombre d'heures. Elle ne possède pas de constructeur public : pour obtenir une instance, il faut utiliser la méthode hours() qui est une fabrique retournant une constante ou une instance de type Hours selon la valeur fournie en paramètre.

La méthode hoursBetween() permet d'obtenir une instance qui encapsule le nombre d'heures entre deux Instant ou deux Partial.

La méthode hoursIn() permet d'obtenir une instance qui encapsule le nombre d'heures d'un Interval.

La classe Minutes encapsule un nombre de minutes. Elle ne possède pas de constructeur public : pour obtenir une instance, il faut utiliser la méthode minutes() qui est une fabrique retournant une constante ou une instance de type Minutes selon la valeur fournie en paramètre.

La méthode minutesBetween() permet d'obtenir une instance qui encapsule le nombre de minutes entre deux Instant ou deux Partial.

La méthode minutesIn() permet d'obtenir une instance qui encapsule le nombre de minutes d'un Interval.

La classe Seconds encapsule un nombre de secondes. Elle ne possède pas de constructeur public : pour obtenir une instance, il faut utiliser la méthode seconds() qui est une fabrique retournant une constante ou une instance de type Seconds selon la valeur fournie en paramètre.

La méthode secondsBetween() permet d'obtenir une instance qui encapsule le nombre de secondes entre deux Instant ou deux Partial.

La méthode secondsIn() permet d'obtenir une instance qui encapsule le nombre de secondes d'un Interval.

La classe Weeks encapsule un nombre de semaines. Elle ne possède pas de constructeur public : pour obtenir une instance, il faut utiliser la méthode weeks() qui est une fabrique retournant une constante ou une instance de type Weeks selon la valeur fournie en paramètre.

La méthode weeksBetween() permet d'obtenir une instance qui encapsule le nombre de semaines entre deux Instant ou deux Partial.

La méthode weeksIn() permet d'obtenir une instance qui encapsule le nombre de semaines d'un Interval.

La classe Years encapsule un nombre d'années. Elle ne possède pas de constructeur public : pour obtenir une instance, il faut utiliser la méthode years() qui est une fabrique retournant une constante ou une instance de type Years selon la valeur fournie en paramètre.

La méthode yearsBetween() permet d'obtenir une instance qui encapsule le nombre d'années entre deux Instant ou deux Partial.

La méthode yearsIn() permet d'obtenir une instance qui encapsule le nombre d'années d'un Interval.

La classe Period propose de nombreux constructeurs.

Une instance de type Period peut s'utiliser avec une instance de type Instant pour obtenir une nouvelle instance de type Instant.

Exemple :

```
    DateTime noel = new DateTime("2012-12-25");
    DateTime nouvelAn = noel.plus(Period.days(7));
    System.out.println(nouvelAn);
```

Les classes Period et MutablePeriod implémentent l'interface ReadablePeriod.

La conversion d'une période peut être complexe : par exemple, une journée ne vaut pas forcément 24 heures : elle peut aussi valoir 23 ou 25 heures en fonction de l'heure d'été/d'hiver. Cependant une journée est généralement considérée comme composée de 24 heures : la classe Days possède la méthode toStandardHours() qui permet de convertir la valeur en heures sur la base d'une journée de 24 heures.

La classe Period propose des méthodes pour obtenir et pour modifier les valeurs des différents champs. Comme la classe Period est immuable, les opérations de modifications renvoient une nouvelle instance.

Il est possible de créer une instance de type Period qui encapsule la durée entre deux instants. Il suffit simplement de passer les deux instants en paramètres du constructeur de la classe Period.

Exemple :

```
    DateTime noel12 = new DateTime("2012-12-25");
    DateTime noel13 = new DateTime("2013-12-25");
    Period period = new Period(noel12, noel13);
    System.out.println(period.getYears() + " an entre les deux dates");
```

Le même calcul peut se faire en utilisant la classe Years :

Exemple :

```
    DateTime noel12 = new DateTime("2012-12-25");
    DateTime noel13 = new DateTime("2013-12-25");
    Years year = Years.yearsBetween(noel12, noel13);
    System.out.println(year.getYears() + " an entre les deux dates");
```

Attention : Joda Time considère une instance de type Period qui est null comme une période dont tous les champs sont à zéro.

112.4.4.3. La classe Duration

La classe Duration encapsule une durée mesurée en millisecondes de manière immuable. Un objet de type Duration ne possède aucun système calendaire ni fuseau horaire.

La classe Duration implémente l'interface ReadableDuration. L'interface ReadableDuration est un sous-ensemble des opérations du type Duration, il est donc généralement préférable de définir une variable de type Duration plutôt que du type ReadableDuration.

La classe Duration possède plusieurs constructeurs qui attendent en paramètres la durée ou deux instants qui seront utilisés pour déterminer la durée encapsulée.

Exemple :

```
    DateTime noel = new DateTime("2012-12-25");
    DateTime nouvelAn = new DateTime("2013-01-01");
    Duration duree = new Duration(noel, nouvelAn);
```

Un objet de type Duration peut être ajouté à un objet de type Instant pour obtenir une nouvelle instance de type Instant.

Exemple :

```
    DateTime noel = new DateTime("2012-12-25");
    DateTime nouvelAn = noel.plus(new Duration(24L * 60L * 60L * 1000L * 7));
    System.out.println(nouvelAn);
```

Une instance de type ReadableDuration à null est considérée par Joda Time comme une instance de type ReadableDurantion ayant pour durée la valeur zéro.

112.4.5. Les calendriers et les fuseaux horaires

Joda Time propose le support de plusieurs systèmes calendaires et la gestion des fuseaux horaires.

La classe abstraite Chronology est la classe de base pour encapsuler un système calendaire. La classe DateTimeZone encapsule un fuseau horaire.

Joda Time utilise par défaut le système calendaire ISO et le fuseau horaire par défaut du système.

En interne, Joda Time utilise des fabriques pour créer des instances de type Chronology et DateTimeZone qui sont des singletons.

112.4.5.1. La classe Chronology

Un système calendaire est une manière particulière de représenter le temps et de permettre de réaliser des calculs temporels. Joda Time propose en standard le support de plusieurs systèmes calendaires.

La classe abstraite Chronology est la classe mère de toutes les classes qui encapsulent un système calendaire. Une instance de type Chronology encapsule un moteur de calcul pour appliquer les règles d'un système calendaire.

Joda Time propose un système extensible pour supporter différents systèmes calendaires. Joda Time propose plusieurs classes filles, chacune encapsulant une implémentation d'un système calendaire :

- ISOChronology : calendrier définit par le standard ISO8601 (calendrier par défaut)
- GJChronology : permet une utilisation du calendrier Julien et du calendrier Grégorien
- GregorianChronology : le calendrier Gregorien
- IslamicChronology : le calendrier Islamique
- JulianChronology : le calendrier Julien
- CopticChronology : le calendrier Copte
- BuddhistChronology : le calendrier Bouddhiste
- EthiopicChronology : le calendrier Ethiopien

Pour obtenir une instance dédiée à un système calendaire, il faut utiliser la fabrique correspondante en invoquant la méthode getInstance() de la classe qui encapsule le système calendaire souhaité.

Exemple :

```
Chronology calendrierCopte = CopticChronology.getInstance()
DateTime dt = new DateTime(calendrierCopte);
```

Le système de calendrier par défaut de Joda Time est le calendrier ISO. Ce calendrier est couramment utilisé mais ne convient pas pour des dates antérieures à 1583.

Il est possible de fournir une instance de type DateTimeZone qui encapsule un fuseau horaire en paramètre de la fabrique pour préciser le fuseau horaire à utiliser.

Exemple :

```
DateTimeZone zone = DateTimeZone.forID("Europe/Paris");
Chronology calendrierCopte = CopticChronology.getInstance(zone)
DateTime dt = new DateTime(calendrierCopte);
```

Attention : une instance de type Chonology à null est toujours considérée par l'API Joda Time comme une instance de type Chronology par défaut (système calendaire ISO8601 et fuseau horaire par défaut).

112.4.5.2. La classe DateTimeZone

Un fuseau horaire correspond à un découpage géographique de la surface de la Terre relatif au méridien de Greenwich : le fuseau horaire de ce méridien est nommé GMT (Greenwich Mean Time). Le concept d'UTC (Universal Coordonated Time) est similaire mais pas tout à fait identique.

Le fuseau horaire permet de préciser un décalage, positif ou négatif, par rapport à l'UTC. La valeur de ce décalage peut varier en fonction de l'utilisation de l'heure d'été/d'hiver (DST en anglais : Daylight Saving Time).

Un fuseau horaire est utilisé pour calculer une heure par rapport à une position géographique.

La classe DateTimeZone encapsule un fuseau horaire de manière immuable.

Lors du calcul de certaines données temporelles, il peut être important de connaître le lieu où un point dans le temps doit être représenté. Cela se fait avec un fuseau horaire car selon celui-ci, la représentation du point dans un calendrier peut être différente.

C'est la raison pour laquelle une instance de type Chronology encapsule une instance de type DateTimeZone. Si aucun fuseau horaire n'est précisé, alors c'est le fuseau horaire par défaut qui est utilisé : c'est celui de la machine hôte.

La méthode forId() de la classe DateTimeZone est une fabrique qui permet de créer une instance en passant en paramètre l'identifiant de la zone concernée.

Exemple :

```
    DateTimeZone zone = DateTimeZone.forID("Europe/Paris");
```

La classe DateTimeZone définit la constante UTC qui correspond à l'instance de DateTimeZone pour l'UTC.

La méthode getDefault() permet d'obtenir une instance de type DateTimeZone encapsulant le fuseau horaire par défaut qui correspond à celui du système hôte.

Exemple :

```
    DateTimeZone zone = DateTimeZone.getDefault();
    System.out.println(zone);
```

C'est ce fuseau horaire qui sera utilisé par défaut par l'API Joda Time si aucun fuseau horaire n'est explicitement précisé.

La méthode statique setDefault() peut être utilisée pour modifier le fuseau horaire qui doit être utilisé par défaut.

Les fuseaux horaires sont des concepts qui évoluent fréquemment en fonction du contexte politique du pays concerné. Le JDK et Joda Time utilise la TZ Database. Comme le JDK peut ne pas être mis à jour, il est possible de mettre à jour la base incluse dans Joda Time et de recompiler la bibliothèque pour tenir compte des mises à jour dans la définition des fuseaux horaires.

112.4.5.3. Le système calendaire ISO8601

Le système calendaire ISO8601 est une normalisation basée sur le calendrier Grégorien afin de faciliter les échanges de date/heures entre applications, systèmes et pays.

Ce système calendaire est implémenté dans la classe ISOChronology qui est immuable.

Ce standard définit :

- 12 mois : de janvier à décembre, numérotés de 1 à 12
- 7 jours : de lundi à dimanche, numérotés de 1 à 7
- plusieurs formats pour restituer la date dont le plus commun est YYYY-MM-DDTHH:MM:SS.SSSZ : il est notamment utilisé dans le standard XML.

La classe ISOChronology est l'implémentation utilisée par défaut par Joda Time : si une instance de type Chronology fournie à l'API est null, alors c'est une instance de type ISOChronology qui sera utilisée.

Pour obtenir une instance de type ISOChronology, il faut invoquer sa méthode getInstance().

Exemple :

```
Chronology chrono = ISOChronology.getInstance();
DateTime dt = new DateTime(2012, 12, 25, 0, 0, 0, 0, chrono);
```

112.4.5.4. La calendrier Bouddhiste

Le calendrier Bouddhiste ne possède qu'une seule ère et ses années possèdent un décalage de 543 ans par rapport au calendrier Grégorien.

La classe BuddhistChronology est l'implémentation du calendrier Bouddhiste. Pour obtenir une instance, il faut invoquer la méthode getInstance() de la classe BuddhistChronology.

Exemple :

```
    DateTime noel12 = new DateTime("2012-12-25");
    DateTime dt = noel12.withChronology(BuddhistChronology.getInstance());
    System.out.println(dt);
```

Résultat :

```
2555-12-25T00:00:00.000+01:00
```

112.4.5.5. Le calendrier Copte

Le calendrier copte est basé sur le calendrier utilisé dans l'Ancien Egypte. Il est utilisé par l'Eglise Orthodoxe Copte.

Le calendrier Copte repose sur 12 mois de 30 jours chacun suivi d'une période de 5 ou 6 jours. L'année contient donc 365 ou 366 jours. Les années bissextiles sont celles qui durent 366 jours : elles surviennent tous les 4 ans.

La classe CopticChronology implémente le calendrier Copte. Dans cette implémentation, les 5 ou 6 jours complémentaires sont stockées dans un treizième mois.

Pour obtenir une instance, il faut invoquer sa méthode getInstance().

Exemple :

```
    DateTime noel12 = new DateTime("2012-12-25");
    DateTime dt = noel12.withChronology(CopticChronology.getInstance());
    System.out.println(dt);
```

Résultat :

```
1729-04-16T00:00:00.000+01:00
```

112.4.5.6. Le calendrier Ethiopien

Le calendrier Ethiopien est similaire au calendrier Copte.

La classe EthiopicChronology implémente le calendrier Ethiopien. Pour obtenir une instance, il faut invoquer sa méthode getInstance().

Exemple :

```
    DateTime noel12 = new DateTime("2012-12-25");
    DateTime dt = noel12.withChronology(EthiopicChronology.getInstance());
    System.out.println(dt);
```

Résultat :

```
2005-04-16T00:00:00.000+01:00
```

112.4.5.7. Le calendrier Grégorien

Le calendrier Grégorien est le calendrier majoritairement utilisé pour les traitements métiers. Ce calendrier a remplacé le calendrier Julien. Le calendrier Grégorien définit une année bissextile tous les quatre ans avec deux exceptions : les années divisibles par 100 ne sont pas bissextiles sauf celles divisibles par 400.

Ce système calendaire est compatible avec le système calendaire ISO même si la gestion du siècle est légèrement différente. Il n'est utilisable que pour des dates postérieures à 1583.

La classe GregorianChronology implémente le calendrier Grégorien. Pour obtenir une instance, il faut invoquer sa méthode getInstance().

Exemple :

```
    DateTime noel12 = new DateTime("2012-12-25");
    DateTime dt = noel12.withChronology(GregorianChronology.getInstance());
    System.out.println(dt);
```

Résultat :

```
2012-12-25T00:00:00.000+01:00
```

112.4.5.8. Le système calendaire Grégorien/Julien

Le système calendaire Grégorien/Julien est la combinaison des systèmes calendaires utilisés par les Chrétiens et les Romains. Ce système calendaire est utilisé pour des traitements de dates historiques puisqu'il permet de gérer les dates du calendrier Julien puis celles du calendrier Grégorien. La date de basculement de calendriers est configurable : elle est par défaut au 15/10/1582 comme l'a défini le pape Grégoire XIII.

La classe GJChronology implémente le calendrier Grégorien/Julien. Cette classe est similaire à la classe java.util.GregorianCalendar du JDK.

Pour obtenir une instance, il faut invoquer sa méthode getInstance().

Exemple :

```
    DateTime noel12 = new DateTime("2012-12-25");
    DateTime dt = noel12.withChronology(GJChronology.getInstance());
    System.out.println(dt);
```

Résultat :

```
2012-12-25T00:00:00.000+01:00
```

Une des surcharge de la méthode getInstance() permet de préciser le point dans le temps où le calendrier Grégorien doit être utilisé.

112.4.5.9. Le calendrier Islamique

Le calendrier Islamique est basé sur les cycles de la Lune : il est utilisé dans de nombreux pays musulmans.

La classe IslamicChronology implémente le système calendaire Islamique. Pour obtenir une instance, il faut invoquer sa méthode getInstance().

Exemple :

```
    DateTime noel12 = new DateTime("2012-12-25");
    DateTime dt = noel12.withChronology(IslamicChronology.getInstance());
    System.out.println(dt);
```

Résultat :

```
1434-02-11T00:00:00.000+01:00
```

La classe IslamicChronology.LeapYearPatternType permet de définir la façon dont les années bissextiles sont définies.

112.4.5.10. Le calendrier Julien

Le calendrier Julien est utilisé jusqu'au 15 octobre 1582 où il a été remplacé par le calendrier Grégorien.

La classe JulianChronology implémente le système calendaire Julien. Pour obtenir une instance, il faut invoquer sa méthode getInstance().

Exemple :

```
    DateTime noel12 = new DateTime("1012-12-25");
    DateTime dt = noel12.withChronology(JulianChronology.getInstance());
    System.out.println(dt);
```

Résultat :

```
1012-12-19T00:00:00.000+00:09:21
```

112.4.6. La manipulation des dates

Les fonctionnalités de manipulation de dates sont le point fort de l'API Joda Time de part leur richesse et leur facilité d'utilisation par rapport aux classes fournies par le JDK.

Exemple :

```
package fr.jmdoudoux.dej.jodatime;

import org.joda.time.DateTime;

public class TestJodaTime {
  public static void main(String[] args) {
    DateTime dateTime = new DateTime(2012, 1, 1, 0, 0, 0, 0);
    System.out.println(dateTime.plusDays(30)
        .toString("dd/MM/yyyy HH:mm:ss.SSS"));
  }
}
```

Le code équivalent en utilisant les classes du JDK est le suivant :

Exemple :

```
package fr.jmdoudoux.dej.jodatime;

import java.text.SimpleDateFormat;
import java.util.Calendar;

public class TestJodaTime {

  public static void main(String[]args) {
    Calendar calendar = Calendar.getInstance();
    calendar.setTimeInMillis(0);
    calendar.set(2012, Calendar.JANUARY, 1, 0, 0, 0);
    SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss.SSS");
    calendar.add(Calendar.DAY_OF_MONTH, 30);
    System.out.println(sdf.format(calendar.getTime()));
  }
}
```

Cette section propose plusieurs exemples pour illustrer certaines fonctionnalités de manipulation de dates proposées par Joda Time.

Exemple : obtenir la date/heure courante plus une heure et demi.

Exemple :

```
    DateTime now = new DateTime();
    DateTime limite = now.plusHours(1).plusMinutes(30);
    System.out.println(limite);
```

Exemple : obtenir le dernier jour du mois précédent.

Exemple :

```
    LocalDate dernierJourduMoisPrecedent = LocalDate.now() // aujourd'hui
        .minusMonths(1) // on retire un mois
        .dayOfMonth() // on récupère le jour du mois
        .withMaximumValue(); // on lui affecte sa valeur maximale
    System.out.println(dernierJourduMoisPrecedent);
```

Exemple : obtenir le lundi de la semaine de Noël.

Exemple :

```
    DateTime dateTime = new DateTime("2012-12-25");
    DateTime result = dateTime.dayOfWeek().setCopy(DateTimeConstants.MONDAY);
    System.out.println(result);
```

Résultat :

```
2012-12-24T00:00:00.000+01:00
```

Exemple : Obtenir la date de paiement à 90 jours fin de mois.

Exemple :

```
    LocalDate datePaiement = LocalDate.now() // aujourd'hui
        .plusDays(90) // on ajoute 90 jours
        .dayOfMonth() // on récupère le jour du mois
        .withMaximumValue(); // on lui affecte sa valeur maximale
    System.out.println(datePaiement);
```

Pour calculer le nombre de jours entre deux dates, il est possible d'utiliser la méthode daysBetween() de la classe Days.

Exemple :

```
    DateTime dateTimeDeb = new DateTime("2012-12-25");
    DateTime dateTimeFin = new DateTime("2012-12-31");
    Days d = Days.daysBetween(dateTimeDeb, dateTimeFin);
    int days = d.getDays();
    System.out.println(days);
```

Résultat :

```
6
```

Pour obtenir le nombre des différents éléments qui composent l'écart entre deux dates, il est possible d'utiliser la classe Period.

Exemple :

```
    DateTime dateTimeDeb = new DateTime("2011-11-25");
    DateTime dateTimeFin = new DateTime("2012-12-31");
    Period p = new Period(dateTimeDeb, dateTimeFin, PeriodType.yearWeekDay());
    System.out.println("annees   " + p.getYears());
    System.out.println("semaines " + p.getWeeks());
    System.out.println("jours    " + p.getDays());
```

Résultat :

```
annees   1
semaines 5
jours    1
```

Exemple : Obtenir le nombre de jours avant la nouvelle année.

Exemple :

```
    LocalDate aujourdhui = LocalDate.now();
    LocalDate nouvelAn = aujourdhui.plusYears(1).withDayOfYear(1);
    Days nbJours = Days.daysBetween(aujourdhui, nouvelAn);
    System.out.println(nbJours.getDays());
```

112.4.7. L'intéropabilité avec les classes du JDK

Joda Time offre une grande facilité pour l'interopérabilité avec les classes du JDK relatives aux traitements des données de type date/heure (notamment les classes Date et Calendar).

Il est possible de convertir des classes du JDK vers leur équivalent et vice versa tout en profitant de la facilité et de la richesse des fonctionnalités de manipulation de dates/heures offertes par Joda Time.

Les classes de Joda Time qui encapsulent une date/heure acceptent comme paramètre dans une surcharge de leur constructeur un objet de type java.util.Date ou java.util.Calendar.

La méthode toCalendar() de la classe AbstractDateTime permet de convertir l'objet Joda Time en une instance de type java.util.Calendar.

Exemple :

```
    DateTime dateTime = new DateTime("2012-12-25");
    Calendar calendar = dateTime.toCalendar(Locale.getDefault());
    System.out.println(calendar);
```

La méthode toGregorianCalendar() de la classe AbstractDateTime permet de convertir l'objet Joda Time en une instance de type java.util.GregorianCalendar.

Exemple :

```
    DateTime dateTime = new DateTime("2012-12-25");
    GregorianCalendar calendar = dateTime.toGregorianCalendar();
    System.out.println(calendar);
```

La méthode toDate() de la classe AbstractInstant permet de convertir l'objet Joda Time en une instance de type java.util.Date.

Exemple :

```
    DateTime dateTime = new DateTime("2012-12-25");
    Date date = dateTime.toDate();
    System.out.println(date);
```

Pour convertir un objet de type LocalDate en un objet de type Date ou Calender, il est nécessaire de le convertir au préalable en une instance de type DateMidnight en invoquant la méthode toDateMidnight().

Exemple :

```
    LocalDate localDate = new LocalDate("2012-12-25");
    Date date = localDate.toDateMidnight().toDate();
    System.out.println(date);
```

112.4.8. Le formattage des dates

L'obtention d'une date à partir d'une ressource externe (fichiers, services web, ...) ou d'une zone de saisie de l'utilisateur, le formattage d'une date sont fréquents dans une application. Le format de ces dates n'est pas toujours le même : Joda Time propose plusieurs solutions pour définir ce format de manière simple ou personnalisée.

Le plus simple pour formater un objet de type DateTime est d'invoquer sa méthode toString().

Il est possible de fournir en paramètre de la méthode toString() une chaîne de caractères qui contient le format désiré. Le format à utiliser est quasiment le même que la classe SimpleDateFormat du JDK.

Exemple :

```
    DateTime dateTime = new DateTime("2012-12-25");
    System.out.println(dateTime.toString("dd-MM-yyyy HH:mm:ss"));
    System.out.println(dateTime.toString("EEEE dd MMMM yyyy HH:mm:ss"));
    System.out.println(dateTime.toString("MM/dd/yyyy HH:mm ZZZZ"));
    System.out.println(dateTime.toString("MM/dd/yyyy HH:mm Z"));
```

Résultat :

```
25-12-2012 00:00:00
mardi 25 décembre 2012 00:00:00
12/25/2012 00:00 Europe/Paris
12/25/2012 00:00 +0100
```

La classe ISODateTimeFormat est une fabrique pour obtenir des instances de type DateTimeFormatter pour différent format de dates respectant la norme ISO8601.

Exemple :

```
    DateTime dateTime = new DateTime("2012-12-25");
    System.out.println(dateTime.toString(ISODateTimeFormat.basicDateTime()));
    System.out.println(dateTime.toString(ISODateTimeFormat
        .basicDateTimeNoMillis()));
    System.out.println(dateTime.toString(ISODateTimeFormat
        .basicOrdinalDateTime()));
    System.out
        .println(dateTime.toString(ISODateTimeFormat.basicWeekDateTime()));
```

Résultat :

```
20121225T000000.000+0100
20121225T000000+0100
2012360T000000.000+0100
2012W522T000000.000+0100
```

La classe DateTimeFormatter est utilisée pour formater et extraire une date d'une chaîne de caractères.

La classe DateTimeFormatter est thread-safe et immuable. Elle contient un cache en interne qui maintient des instances ce qui évite d'avoir à créer une instance à chaque utilisation.

La classe DateTimeFormat propose plusieurs méthodes qui sont des fabriques pour obtenir des instances de type DateTimeFormatter. La plupart de ces méthodes proposent des formats standard. La méthode forPattern() permet de préciser explicitement le format de la date/heure à utiliser.

Exemple :

```
    DateTimeFormatter formatter = DateTimeFormat
        .forPattern("dd-MM-yyyy HH:mm:ss");
    DateTime dateTime = formatter.parseDateTime("25-12-2012 00:00:00");
    System.out.println(dateTime);
```

Résultat :

```
2012-12-25T00:00:00.000+01:00
```

La méthode withLocale() permet de préciser la locale à utiliser et renvoie une nouvelle instance.

Exemple :

```
    DateTime dateTime = new DateTime("2012-12-25");
    DateTimeFormatter formatter = DateTimeFormat
      .forPattern("EEEE dd MMMM yyyy HH:mm:ss");
    DateTimeFormatter frenchFmt = formatter.withLocale(Locale.FRENCH);
    System.out.println(frenchFmt.print(dateTime));
    DateTimeFormatter englishFmt = formatter.withLocale(Locale.ENGLISH);
    System.out.println(englishFmt.print(dateTime));
```

Résultat :

```
mardi 25 décembre 2012 00:00:00
Tuesday 25 December 2012 00:00:00
```

Pour des formats très spécifiques, Joda Time propose la classe DateTimeFormatterBuilder qui implémente le motif de conception builder pour créer une instance de type DateTimeFormatter.

La classe DateTimeFormatterBuilder propose de nombreuses méthodes appendXXX() qui permet de préciser chaque élément qui devra être ajouté pour définir le format de la date.

La méthode toFormatter() permet de demander l'instance de type DateTimeFormatter selon la configuration définie.

L'exemple ci-dessous va demander le formatage de l'année sur deux caractères.

Exemple :

```
    DateTime dateTime = new DateTime("2012-12-25");
    DateTimeFormatter fmt = new DateTimeFormatterBuilder().appendDayOfMonth(2)
        .appendLiteral(' ').appendMonthOfYearShortText().appendLiteral(' ')
        .appendTwoDigitYear(1949).toFormatter();
    System.out.println(fmt.print(dateTime));
```

Résultat :

```
25 déc. 12
```

La méthode clear() permet de réinitialiser la configuration.

112.4.9. D'autres fonctionnalités de Joda Time

Joda Time propose aussi des fonctionnalités pour modifier la date/heure par défaut ou utiliser des objets mutables.

112.4.9.1. La modification de l'heure de la JVM

La classe DateTimeUtils propose plusieurs méthodes qui permettent de modifier la date/heure obtenue par l'API Joda Time.

La méthode setCurrentMillisFixed() permet de modifier la date/heure de Joda Time avec celle correspondant au nombre de millisecondes fourni en paramètre.

La méthode setCurrentMillisOffset() permet de modifier la date/heure de Joda Time en effectuant un décalage avec le nombre de millisecondes fourni en paramètre.

La méthode setCurrentMillisSystem() permet d'accorder la date/heure de Joda Time avec celle du système.

Exemple :

```
    DateTime noel13 = new DateTime("2013-12-25");
    DateTimeUtils.setCurrentMillisFixed(noel13.getMillis());
    System.out.println(new Date());
    System.out.println(LocalDateTime.now());

    // remettre la date de Joda Time à la date systeme
    DateTimeUtils.setCurrentMillisSystem();
    System.out.println(LocalDateTime.now());

    // modifier la date de Joda Time à la veille
    DateTimeUtils.setCurrentMillisOffset(1000 * 60 * 60 * 24);
    System.out.println(LocalDateTime.now());
```

L'utilisation de ces méthodes peut être pratique pour des tests.

Attention : la date/heure de la JVM obtenue avec les API du JDK n'est pas modifiée.

112.4.9.2. Les objets mutables

Comme la plupart des objets Joda Time sont immuables, leur modification implique la création d'une nouvelle instance à chaque méthode invoquée. Si plusieurs champs doivent être modifiés, il peut être intéressant d'utiliser une version mutable afin de limiter le nombre d'instances créées.

Joda Time propose les classes MutableDateTime, MutableInterval et MutablePeriod.

Exemple :

```
    DateTime noel13 = new DateTime("2013-12-25");
    MutableDateTime mdt = noel13.toMutableDateTime();
    mdt.setDayOfMonth(1);
    mdt.setYear(2012);
    mdt.setMonthOfYear(1);
    DateTime result = mdt.toDateTime();
    System.out.println(result);
```

La classe MutableDateTime possède de nombreux constructeurs pour indiquer la date/heure encapsulée.

La classe DateTime propose aussi la méthode toMutableDateTime() qui renvoie une instance de type MutableDateTime encapsulant la date/heure de l'objet.

La classe MutableDateTime propose de nombreuses méthodes qui ne renvoient rien pour modifier un champ de la date/heure encapsulée.

La méthode toDateTime() permet de renvoyer une instance de type DateTime qui encapsule la date/heure de l'objet.

112.5. La classe FastDateFormat du projet Apache commons.lang

La classe org.apache.commons.lang.time.FastDateFormat permet le formatage d'une date comme SimpleDateFormat mais elle est plus performante et surtout thread-safe.

La classe FastDateFormat offre des fonctionnalités de formatage d'une date similaires à celles de la classe SimpleDateFormat : elle propose cependant un support du timezone différent dans le formatage.

FastDateFormat ne permet que le formatage d'une date. Contrairement à la classe SimpleDateFormat, elle ne permet pas d'extraire une date d'une chaîne de caractères.

Exemple :

```
import java.text.SimpleDateFormat; import java.util.Date;

import org.apache.commons.lang.time.FastDateFormat;

public class TestSdf {

  public static void main(final String[] args) {
    final String format = "dd-MM-yyyy HH:mm:ss.SSS";
    SimpleDateFormat sdf = new SimpleDateFormat(format);
    FastDateFormat fdf = FastDateFormat.getlnstance(format);
    final Date d = new Date();
    final int nblteration = 1000000;
    long start = 0;
    long tempsTrt = 0;

    start = System.currentTimeMillis();
    for (int i = 0; i < nblteration; i++) {
      fdf = FastDateFormat.getlnstance(format);
      d.setTime(System.currentTimeMillis());
      fdf.format(d);
    }
    tempsTrt = System.currentTimeMillis() - start;
    System.out.println("FastDateFormat : " + tempsTrt + " ms");

    start = System.currentTimeMillis();
    for (int i = 0; i < nblteration; i++) {
      sdf = new SimpleDateFormat(format);
      d.setTime(System.currentTimeMillis());
      sdf.format(d);
    }
    tempsTrt = System.currentTimeMillis() - start;
    System.out.println("SimpleDateFormat : " + tempsTrt + " ms");
  }

}
```

Résultat :

```
FastDateFormat : 2041 ms
SimpleDateFormat : 5360 ms
```

La classe org.apache.commons.lang.time.DateFormatUtils est une classe utilitaire pour le formatage de dates et d'heures en utilisant la classe FastDateFormat.

Elle propose des constantes pour des instances de FastDateFormat avec des motifs de formatage courants en ISO08601 et SMTP :

ISO_DATETIME_FORMAT	formatage d'une date/heure en ISO860 sans timezone : yyyy-MM-dd'T'HH:mm:ss
ISO_DATETIME_TIMEZONE_FORMAT	formatage d'une date/heure en ISO8601 avec timezone : yyyy-MM-dd'T'HH:mm:ssZZ
ISO_DATE_FORMAT	formatage d'une date enISO8601 sans timezone : yyyy-MM-dd

ISO_TIME_FORMAT	formatage d'une heure en pseudo ISO8601 sans time zone (la spécification ne permet pas d'avoir une heure sans timezone) : 'T'HH:mm:ss
ISO_DATE_TIME_ZONE_FORMAT	formatage d'une date enISO8601 sans timezone (la spécification ne permet pas d'avoir une heure sans timezone) : yyyy-MM-ddZZ
ISO_TIME_TIMEZONE_FORMAT	formatage d'une heure en ISO8601 avec time zone : 'T'HH:mm:ssZZ.
ISO_TIME_NOT_FORMAT	formatage d'une heure en pseudo ISO8601 sans time zone (la spécification requiert que l'heure soit préfixée par le caractère T) : HH:mm:ss.
ISO_TIME_NOTTIMEZONE_FORMAT	formatage d'une heure en pseudo ISO8601 avec time zone (la spécification requiert que l'heure soit préfixée par le caractère T) : HH:mm:ssZZ.
SMTP_DATETIME_FORMAT	formatage d'une date heure au format requis par SMTP : EEE, dd MMM yyyy EH:mm:ss Z in US locale.

Il n'est pas recommandé d'utiliser son constructeur par défaut.

Elle propose de nombreuses méthodes notamment plusieurs surcharges des méthodes format() et formatUTC() acceptant la date à formater en plusieurs formats (long, Date, Calendar) et permettant de préciser le format et la Locale à utiliser.

Exemple :

```
public static void main(final String[] args) {
  final Date today = new Date();

  /*
   *    formatage en ISO8601 sans timezone : yyyy-MM-dd'T'HH:mm:ss.
   */
  String timestamp = DateFormatUtils.ISO_DATETIME_FORMAT.format(today);
  System.out.println("timestamp = " + timestamp);

  /*
   *    formatage en ISO8601 avec timezone : yyyy-MM-dd' T'HH:mm:ssZZ.
   */
  timestamp = DateFormatUtils.ISO_DATETIME_TIME_ZONE_FORMAT.format(today)
  System.out.println("timestamp = " + timestamp);

  /*
   *    formatage au format SMTP : EEE, dd MMM yyyy HH:mm:ss Z (Locale US).
   */
  timestamp = DateFormatUtils.SMTP_DATETIME_FORMAT.format(today);
  System.out.println("timestamp = " + timestamp);
}
```

112.6. L'API Date and Time

Malgré les nombreux intérêts de la plate-forme Java, celle-ci a toujours manqué d'un support correct pour la gestion des données temporelles jusqu'à Java 8. Pourtant un bon support de ce type de fonctionnalités est important car l'utilisation de données temporelles est omniprésente notamment dans les applications de gestion.

Il y a 3 API différentes pour manipuler le temps dans le JDK8 :

- java.util.Date depuis Java 1.0
- java.util.Calendar depuis Java 1.1
- java.time.* depuis Java 1.8

112.6.1. Le besoin d'une nouvelle API

Java 1.0 ne propose que la classe java.util.Date concernant le support de données temporelles. Elle encapsule un point dans le temps avec une précision à la milliseconde.

Elle présente de nombreuses lacunes dont les principales sont :

- elle n'encapsule pas une date mais une date et une heure
- le mois est numéroté de 0 à 11
- l'année commence en 1900 : elle doit être précisée en utilisant un décalage
- elle n'est pas immuable donc pas thread-safe
- elle ne propose aucun support pour l'internationalisation
- elle supporte uniquement du fuseau horaire de la machine
- le formatage de la date grâce à la méthode toString() est particulier

Exemple :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.util.Date;

public class TestDate {
  public static void main(String[] args) {
    Date date = new Date(114, 11, 25);
    System.out.println(date);
  }
}
```

Résultat :

```
Thu Dec 25 00:00:00 CET 2014
```

Pour combler une partie de ces lacunes et conserver la compatibilité, Java 1.1 a rendu deprecated une grande partie des méthodes et constructeurs de la classe Date et a introduit la classe java.util.Calendar. Celle-ci n'a pas résolu tous les problèmes car elle a aussi certains inconvénients de la classe Date mais aussi des inconvénients qui lui sont propres :

- son nom n'est pas très représentatif de ce qu'elle fait
- le mois est numéroté de 0 à 11
- pour obtenir une des valeurs encapsulées, il faut invoquer la méthode get() en lui passant en paramètre un entier : des constantes sont définies pour chacun des éléments
- elle encapsule ses données de deux manières : un entier long qui représente le nombre de millisecondes écoulées depuis un instant donné et leur équivalent sous la forme de champs années, mois, jours, heures, minutes, secondes, ... se qui requiert des conversions pour maintenir leur cohérence
- elle n'est pas immuable donc pas thread-safe
- il n'est pas possible de formater un objet de type Calendar
- elle a introduit une confusion avec son utilisation vis-à-vis de la classe Date

De plus, certaines fonctionnalités requièrent uniquement un objet de type Date ce qui oblige à faire des conversions.

Pour analyser et formater une date, la classe abstraite DateFormat et la classe SimpleDateFormat ont été ajoutées. Elles présentent aussi des inconvénients :

- SimpleDateFormat n'est pas thread-safe
- ses performances sont médiocres

La classe java.util.TimeZone encapsule un fuseau horaire.

Pour pallier ces inconvénients, plusieurs API open source ont été développées : la plus populaire est Joda-Time. Cependant Joda-Time n'est pas exempt de défauts :

- la représentation interne des données temporelles rend les calculs encore plus complexes
- de nombreuses méthodes permettent de recevoir null comme paramètre ce qui peut induire des bugs
- les objets temporels peuvent utiliser différents calendriers

Ces points expliquent en grande partie pourquoi Joda-Time n'est pas et ne peut pas être l'implémentation de référence de la JSR 310.

112.6.2. La JSR 310

L'API Date-Time offre un support pour utiliser et manipuler des données temporelles de manière la plus complète possible : date, heure, intervalle de temps, calendrier, fuseau et décalage horaire, ...

Les spécifications de l'API Date-Time sont définies dans la JSR 310 dont l'implémentation de référence est le projet ThreeTen. Celle-ci s'est largement inspirée de l'API Joda-Time pour proposer une API riche permettant le support de données temporelles dans l'API du JDK. Elle est intégrée à Java SE version 8.

L'API propose une meilleure modélisation des classes et interfaces permettant la gestion de dates/heures qui est riche, puissante et extensible.

L'API possède plusieurs avantages :

- simplicité : les nombreuses classes de l'API permettent une meilleure séparation des rôles. Par exemple, elle définit les classes Date, Time et DateTime. Plusieurs énumérations pour différentes données sont définies pour limiter les risques d'erreurs
- richesse des fonctionnalités
- clarté des opérations notamment celles qui concernent des calculs
- immutabilité donc thread-safe : la plupart des principaux objets sont immuables
- création d'instances grâce à des fabriques
- l'utilisation d'interfaces de type fluent permet de simplifier l'utilisation de l'API en chaînant l'invocation des méthodes
- extensible en utilisant le motif de conception Stratégie

L'API Date-Time utilise :

- par défaut, le calendrier utilisé est celui défini par l'International Organization for Standardization dans les spécifications ISO-8601 : c'est le calendrier utilisé de facto dans le monde et qui repose sur le calendrier Grégorien. Il est possible d'utiliser un autre calendrier. Plusieurs implémentations sont proposées dans le package java.util.chrono
- l'Unicode Common Locale Data Repository (CLRD) pour tenir compte de la Locale lorsque la manipulation d'une donnée temporelle le requiert
- le Time-Zone Database (TZDB) pour obtenir des informations sur chacun des fuseaux horaires

La représentation du temps peut se faire selon deux grands modèles :

- le temps machine : c'est un entier long dont la valeur est une quantité de temps, dans une unité définie, écoulée depuis un point dans le temps. Cette représentation est facilement exploitable par une machine
- le temps humain : le temps est représenté par différentes valeurs dans différents champs qui permettent de préciser différentes unités (siècle, année, mois, jour, heure, minute, seconde, ...). La définition et les règles de gestion de ces champs sont définies dans un calendrier. D'autres concepts peuvent compléter ces règles notamment les fuseaux horaires, l'heure d'été/hiver, ...

L'API propose plusieurs classes qui encapsulent différentes données temporelles sous ces deux formes :

Classe	Rôle
LocalDate	Encapsule une date sans heure ni fuseau horaire ni offset. Cette classe peut par exemple être utile pour stocker une date d'anniversaire
LocalDateTime	Encapsule une date et une heure sans fuseau horaire
Localtime	Encapsule une heure sans date, ni fuseau horaire ni offset
Period	Encapsule une période de temps exprimée dans des unités compréhensibles par un humain, par exemple 1 mois et 12 jours. Les valeurs exprimées dans une unité peuvent être négatives
Duration	

	Encapsule une durée entre deux instants stockée avec une précision de la nanoseconde. La durée peut être négative
Instant	Encapsule un point dans le temps avec une précision de la nanoseconde. Elle permet de représenter un point dans la ligne du temps. La classe Instant est la classe la plus proche de la classe java.util.Date
ZonedDateTime	Encapsule une date et une heure avec un fuseau horaire et son offset correspondant. La classe ZonedDateTime est la classe la plus proche de la classe GregorianCalendar
ZoneId	Encapsule un fuseau horaire précisé par son identifiant, par exemple «Europe/Paris»
ZoneOffset	Encapsule un fuseau horaire précisé par un décalage à partir du méridien de Greenwich/UTC, par exemple +01:00

Il y a plusieurs classes qui permettent de gérer une date mais seulement quelques-unes pour gérer l'heure. Par exemple, aucune classe ne propose d'encapsuler une combinaison heures-minutes ou minutes-secondes. Les heures sont toujours gérées sous la forme des quatre champs (heures, minutes, secondes, nanosecondes) pour simplifier l'API. Si un de ces champs n'est pas utile, il suffit de mettre sa valeur à zéro.

De la même façon pour rester pragmatique, le nombre de classes qui encapsulent une combinaison de champs d'une date est limité : YearMonth et MonthDay.

Les classes de l'API qui permettent d'obtenir ou de manipuler une donnée temporelle telles que Instant, LocalDateTime, LocalDate, LocalTime, ZonedDateTime, ... sont désignées par objets temporels dans les sections suivantes.

Elle propose aussi des classes utilitaires pour réaliser des opérations sur des données temporelles :

Classe	Rôle
ChronoUnit	Une énumération des différentes unités temporelles. Elle permet aussi de réaliser des opérations de calcul en utilisant ces unités
DateTimeFormatter	Analyser et formater une date/heure sous une forme textuelle
Clock	Obtenir la date/heure courante dans le fuseau horaire par défaut. Son but est de permettre de pouvoir être remplacée par une autre implémentation notamment pour faciliter les tests unitaires

Les interfaces et les classes de l'API Date-Time sont contenues dans le package java.time et ses quatre sous-packages :

- java.time : contient les classes de bases, immuables qui encapsulent des données temporelles
- java.time.chrono : contient les implémentations de quelques calendriers (Hijrah, Japanese, Minguo, ThaiBuddhist) et des classes et interfaces pour implémenter son propre calendrier
- java.time.format : contient les classes pour analyser et formater des dates/heures
- java.time.temporal : contient des interfaces et des classes pour permettre l'accès aux informations sur des données temporelles et développer de nouvelles fonctionnalités
- java.time.zone : contient des classes pour le support des fuseaux horaires et leurs règles d'application incluant par exemple la prise en compte de l'heure d'été/hiver.

La conception de cette API repose sur plusieurs principes :

- les principales classes sont immuables
- assurer une bonne séparation des rôles de chaque classe : par exemple, il existe des classes pour encapsuler une date et des classes pour encapsuler une heure
- permettre une extensibilité

L'API Date-Time est composée de nombreuses classes et méthodes. Elle utilise de préférence une convention de nommage pour certaines méthodes ayant le même rôle dans différentes classes afin de renforcer la cohérence et faciliter son utilisation.

Nom ou préfixe	Rôle
format	Formater l'objet pour obtenir une chaîne de caractères
get	Obtenir la valeur d'un élément
is	Obtenir la valeur booléenne d'un élément
with	Renvoyer une copie de l'objet avec un élément modifié
plus	Renvoyer une copie de l'objet avec une portion de temps ajoutée
minus	Renvoyer une copie de l'objet avec une portion de temps soustraite
to	Convertir l'objet dans un autre type
at	Combiner l'objet avec une autre instance

De nombreuses classes de base sont immuables : elles ne proposent donc pas de setter mais des fabriques pour faciliter la création de nouvelles instances. Le nom des fabriques utilise au mieux quelques conventions :

Nom ou préfix	Rôle
of	Créer une nouvelle instance à partir des données passées en paramètres : les valeurs sont validées mais ne sont pas converties
from	Créer une nouvelle instance en opérant une conversion des données fournies en paramètres
now	Créer une nouvelle instance en utilisant les données utiles de la date-heure courante
parse	Analyser la chaîne de caractères fournie en paramètre pour créer une nouvelle instance

112.6.3. Les interfaces et classes de bas niveau de l'API

Le package java.time.temporal contient des interfaces, des classes et des énumérations permettant d'obtenir des informations à partir de données temporelles et de réaliser des opérations basiques sur celles-ci.

Ces interfaces sont de bas niveaux : elles ne devraient pas être utilisées pour déclarer des objets dans le code d'une application. Il est préférable d'utiliser le type d'une classe concrète pour déclarer un objet temporel essentiellement car les interfaces sont indépendantes de tout calendrier.

Interface	Rôle
TemporalAccessor	Accès en lecture seule aux informations d'une donnée temporelle
Temporal	Opérations de base de manipulation de données temporelles
TemporalAmount	Une quantité temporelle
TemporalField	Un champ d'une donnée temporelle
TemporalUnit	Une unité temporelle

Les dates et les heures sont exprimées sous la forme de champs. Ces champs possèdent :

- un type
- une valeur
- une unité qui définit la mesure de la quantité de temps

Un objet temporel qui implémente l'interface TemporalAccessor encapsule ses données sous la forme de champs de type TemporalField.

L'énumération ChronoField qui implémente l'interface TemporalField définit ces différents champs.

La valeur de ces champs est exprimée dans une unité définie par l'interface TemporalUnit.

L'énumération ChronoUnit qui implémente l'interface TemporalUnit définit différentes unités couramment utilisées. Certains calendriers peuvent requérir des unités qui leur sont particulières.

Les quantités passées en paramètre des opérations arithmétiques de l'interface Temporal sont définies par l'interface TemporalAmount. Les classes Period et Duration implémentent cette interface.

Il existe différentes définitions de la notion de semaines en fonction de la Locale : par exemple en Europe elle commence le lundi, aux Etats Unis elle commence le dimanche. La classe WeekFields encapsule ces définitions en proposant des constantes pour les plus courantes d'entre-elles.

L'interface Temporal propose des opérations de base de manipulation de données temporelles telles que l'ajout ou le retrait d'une quantité temporelle.

Il est fréquent d'avoir à ajuster une date pour en obtenir une nouvelle : par exemple, le prochain lundi, le dernier jour du mois, ... Généralement, ces opérations sont plus ou moins complexes et sont définies par l'interface TemporalAdjuster.

L'interface TemporalQuery définit les fonctionnalités qui permettent d'obtenir des informations d'un objet temporel sous la forme d'une requête.

112.6.3.1. L'interface TemporalAccessor

L'interface TemporalAccessor définit des fonctionnalités permettant un accès en lecture seule à certains éléments d'un objet temporel en accédant directement à un champ ou en exécutant une requête.

C'est l'interface de base pour les objets qui encapsulent des dates, des heures et des offsets.

La plupart des champs qui composent un objet temporel peuvent être représentés sous la forme d'un entier. Ces champs implémentent l'interface TemporalField. Deux informations d'un objet temporel ne peuvent pas être représentées sous la forme d'un entier : le calendrier et le fuseau horaire. Ces informations peuvent être accédées en utilisant une requête de type TemporalQuery.

Elle définit plusieurs méthodes :

Méthode	Rôle
default int get(TemporalField field)	Obtenir la valeur du champ précisé en paramètre sous la forme d'un entier
long getLong(TemporalField field)	Obtenir la valeur du champ précisé en paramètre sous la forme d'un entier long
boolean isSupported(TemporalField field)	Vérifier si le champ fourni en paramètre est supporté
default <R> R query(TemporalQuery<R> query)	Exécuter la requête passée en paramètre sur l'instance courante
default ValueRange range(TemporalField field)	Obtenir la plage des valeurs possibles pour le champ fourni en paramètre

112.6.3.2. L'interface Temporal

L'interface Temporal hérite de l'interface TemporalAccessor pour définir des opérations de base de manipulations sur des objets temporels.

Elle définit plusieurs méthodes :

Méthode	Rôle
boolean isSupported(TemporalUnit unit)	Vérifier si l'unité passée en paramètre est supportée
default Temporal minus(long amountToSubtract, TemporalUnit unit) default Temporal minus(TemporalAmount amount)	Renvoyer une instance du même type minorée de la quantité temporelle fournie en paramètre
Temporal plus(long amountToAdd, TemporalUnit unit) default Temporal plus(TemporalAmount amount)	Renvoyer une instance du même type majorée de la quantité temporelle fournie en paramètre
long until(Temporal endExclusive, TemporalUnit unit)	Calculer la quantité de temps jusqu'à l'objet temporel fourni en paramètre. La valeur retournée est exprimée dans l'unité temporelle précisée en paramètre
default Temporal with(TemporalAdjuster adjuster)	Renvoyer une copie dont la valeur est ajustée grâce à l'objet qui encapsule les traitements fourni en paramètre
Temporal with(TemporalField field, long newValue)	Renvoyer un objet du même type dont la valeur du champ fourni en paramètre est modifiée

Plusieurs classes de l'API implémentent l'interface Temporal : HijrahDate, Instant, JapaneseDate, LocalDate, LocalDateTime, LocalTime, MinguoDate, OffsetDateTime, OffsetTime, ThaiBuddhistDate, Year, YearMonth et ZonedDateTime.

112.6.3.3. L'interface TemporalAmount

L'interface TemporalAmount définit les fonctionnalités d'un objet qui encapsule une quantité temporelle telle qu'une heure et 30 minutes, trois jours, un an et un jour, ...

Une quantité temporelle n'est pas liée à un point dans le temps. Cette quantité est exprimée au moyen d'une ou plusieurs paires de valeur-unité.

Elle définit plusieurs méthodes :

Méthode	Rôle
Temporal addTo(Temporal temporal)	Renvoyer une copie de l'objet temporel fourni en paramètre auquel est ajoutée la quantité encapsulée
long get(TemporalUnit unit)	Obtenir la valeur dans l'unité précisée
List<TemporalUnit> getUnits()	Retourner une liste des unités temporelles utilisables pour exprimer la quantité. La collection est triée dans l'ordre décroissant de durée de l'unité (de la plus longue à la plus courte)
Temporal subtractFrom(Temporal temporal)	Renvoyer une copie de l'objet temporel fourni en paramètre duquel est soustrait la quantité encapsulée

La quantité est le cumul des différentes valeurs de chacune des unités : pour chaque unité obtenue en invoquant la méthode getUnits(), cela revient à invoquer la méthode get() en lui passant l'unité en paramètre.

L'API fournit deux classes qui implémentent cette interface :

- Period : encapsule une période en années, mois et jours

- Duration : encapsule une durée en secondes et nanosecondes

112.6.3.4. L'interface TemporalField

L'interface TemporalField définit les fonctionnalités d'un champ d'un objet temporel dont il encapsule une représentation humaine.

Elle définit plusieurs méthodes, notamment :

Méthode	Rôle
<R extends Temporal> R adjustInto(R temporal, long newValue)	Renvoyer une copie de l'objet temporel fourni en paramètre dans lequel la valeur du champ encapsulé est remplacée par celle passée en paramètre
TemporalUnit getBaseUnit()	Renvoyer l'unité temporelle dans laquelle la valeur du champ est exprimée
default String getDisplayName(Locale locale)	Renvoyer le nom du champ dans la Locale fournie en paramètre
long getFrom(TemporalAccessor temporal)	Obtenir la valeur du champ encapsulée dans l'objet temporel fourni en paramètre
boolean isDateBased()	Renvoyer un booléen qui précise si le champ entre dans la composition d'une date
boolean isSupportedBy(TemporalAccessor temporal)	Renvoyer un booléen qui précise si le champ est supporté par l'objet temporel fourni en paramètre
boolean isTimeBased()	Renvoyer un booléen qui précise si le champ entre dans la composition d'une heure
ValueRange range()	Obtenir la plage des valeurs valides pour le champ
ValueRange rangeRefinedBy(TemporalAccessor temporal)	Obtenir la plage des valeurs valides pour le champ dans le contexte de l'objet temporel fourni en paramètre.

Il est possible pour certains champs que les méthodes isDateBased() et isTimeBased() renvoient toutes les deux false.

L'énumération ChronoField implémente l'interface TemporalField. D'autres classes contiennent des objets de type TemporalField : IsoFields, WeekFields et JulienFields.

112.6.3.5. L'énumération ChronoField

L'énumération ChronoField propose des constantes de type TemporalField pour les champs standard utilisés dans la composition de dates et/ou d'heures. Ces champs peuvent être utilisés dans différents calendriers.

Valeur	Description
ALIGNED_DAY_OF_WEEK_IN_MONTH	
ALIGNED_DAY_OF_WEEK_IN_YEAR	
ALIGNED_WEEK_OF_MONTH	La semaine du mois
ALIGNED_WEEK_OF_YEAR	La semaine de l'année
AMPM_OF_DAY	La demi-journée (0 AM, 1 PM)
CLOCK_HOUR_OF_AMPM	L'heure de la demi-journée (1-12)
CLOCK_HOUR_OF_DAY	L'heure de la journée (0-24)

DAY_OF_MONTH	Le jour du mois (1-31)
DAY_OF_WEEK	Le jour de la semaine (1-7)
DAY_OF_YEAR	Le jour de l'année (1-366)
EPOCH_DAY	Le Neme jour depuis l'EPOCH (01/01/1970 dans le calendrier ISO)
ERA	L'ère : dans le calendrier ISO (0 BCE, 1 CE)
HOUR_OF_AMPM	L'heure de la demi-journée (0-11)
HOUR_OF_DAY	L'heure de la journée (0-23)
INSTANT_SECONDS	Représente le nombre de secondes comptées à partir du 1er janvier 1970 à 0000z
MICRO_OF_DAY	La microseconde de la journée
MICRO_OF_SECOND	La microseconde de la seconde
MILLI_OF_DAY	La milliseconde de la journée
MILLI_OF_SECOND	La milliseconde de la seconde
MINUTE_OF_DAY	La minute de la journée (0-1440)
MINUTE_OF_HOUR	La minute de l'heure (0-59)
MONTH_OF_YEAR	Le mois de l'année
NANO_OF_DAY	La nanoseconde de la journée
NANO_OF_SECOND	La nanoseconde de la seconde
OFFSET_SECONDS	Le décalage depuis le méridien de Greenwich
PROLEPTIC_MONTH	Le mois depuis l'année 0
SECOND_OF_DAY	La seconde de la journée (0-86400)
SECOND_OF_MINUTE	La seconde de la minute (0-59)
YEAR	L'année
YEAR_OF_ERA	L'année de l'ère

La méthode isSupported() de la classe TemporalAccessor qui attend en paramètre un objet de type TemporalField permet de déterminer si l'objet temporel supporte le champ passé en paramètre.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.LocalDate;
import java.time.temporal.ChronoField;
import java.time.temporal.IsoFields;

public class TestTemporalField {
  public static void main(String[] args) {
    LocalDate date = LocalDate .now();
    int mois = date.get(ChronoField.MONTH_OF_YEAR);
    System.out.println("Mois="+mois);
    boolean supporte = date.isSupported(ChronoField.HOUR_OF_DAY);
    System.out.println("Supporte="+supporte);
    int trimestre = date.get(IsoFields.QUARTER_OF_YEAR);
    System.out.println("Trimestre="+trimestre);
  }
}
```

Résultat :

```
Mois=2
Supporte=false
Trimestre=1
```

112.6.3.6. L'interface TemporalUnit

L'interface TemporalUnit définit les fonctionnalités d'une unité temporelle. Une unité temporelle permet de mesurer le temps en exprimant une quantité temporelle telle qu'une heure, un jour, une année, ...

Elle définit plusieurs méthodes :

Méthode	Rôle
<R extends Temporal> R addTo(R temporal, long amount)	Renvoyer une copie de l'objet temporel fourni en paramètre dans lequel la quantité précisée est ajoutée. La quantité est exprimée dans l'unité encapsulée
long between(Temporal temporal1Inclusive, Temporal temporal2Exclusive)	Calculer la quantité de temps entre les deux objets temporels fournis en paramètres exprimée dans l'unité précisée
Duration getDuration()	Renvoyer une durée de l'unité qui peut être estimée
boolean isDateBased()	Indiquer si l'unité peut être utilisée sur un champ composant une date
boolean isDurationEstimated()	Indiquer si la durée de l'unité est estimée. Par exemple, à cause de la gestion de l'heure d'été/hiver la durée d'une journée est estimée
default boolean isSupportedBy(Temporal temporal)	Vérifier si l'unité est supportée par l'objet temporel fourni en paramètre
boolean isTimeBased()	Indiquer si l'unité peut être utilisée sur un champ composant une heure

L'énumération ChronoUnit implémente cette interface. Quelques unités sont définies dans la classe IsoFields.

112.6.3.7. L 'énumération ChronoUnit

L'énumération ChronoUnit propose des constantes de type TemporalUnit pour les unités temporelles standard utilisées dans la composition de dates et/ou d'heures. Ces unités pour mesurer le temps peuvent être utilisées dans différents calendriers.

Valeur	Rôle
CENTURIES	Un siècle
DAYS	Un jour
DECADES	Une décade
ERAS	Une ère
FOREVER	Le concept de pour toujours (sans fin)
HALF_DAYS	Une demi-journée
HOURS	Une heure
MICROS	Une microseconde
MILLENNIA	Un millénaire
MILLIS	Une milliseconde
MINUTES	Une minute
MONTH	Un mois

NANOS	Une nanoseconde
SECONDS	Une seconde
WEEKS	Une semaine
YEARS	Une année

La méthode isSupported() de la classe TemporalAccessor qui attend en paramètre un objet de type TemporalUnit permet de vérifier si l'unité passée en paramètre est supportée par l'objet Temporal.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.LocalDate;
import java.time.LocalTime;
import java.time.temporal.ChronoUnit;

public class TestTemporalUnit {
  public static void main(String[] args) {
    LocalDate dateDebut = LocalDate.of(2014, 12, 25);
    LocalDate dateFin = LocalDate.of(2015, 12, 25);
    System.out.println("Duree="+ChronoUnit.DAYS.between(dateDebut, dateFin));
    LocalTime heure = LocalTime.now();
    boolean supporte = heure.isSupported(ChronoUnit.YEARS);
    System.out.println("Supporte="+supporte);
    System.out.println("ToString="+ChronoUnit.DAYS.toString());
  }
}
```

Résultat :

```
Duree=365
Supporte=false
ToString=Days
```

112.6.4. Les classes spécifiques aux dates

L'API Date-Time propose des énumérations spécifiques :

- DayOfWeek : les jours de la semaine
- Month : les mois d'une année

L'API Date-Time propose quatre classes qui encapsulent tout ou partie d'une date sans tenir compte de l'heure et du fuseau horaire :

- LocalDate : encapsule une année, un mois et un jour
- MonthDay : encapsule un jour et un mois
- Year : encapsule une année
- YearMonth : encapsule un mois d'une année

112.6.4.1. L'énumération DayOfWeek

L'énumération java.time.DayOfWeek définit des constantes pour les 7 jours de la semaine tels que définit dans le calendrier ISO-8601 : FRIDAY, MONDAY, SATURDAY, SUNDAY, THURSDAY, TUESDAY et WEDNESDAY.

Chaque élément de l'énumération est associé à une valeur numérique tel que définie dans le calendrier ISO-8601 : 1 pour MONDAY à 7 pour SUNDAY.

Il est recommandé d'utiliser les éléments de l'énumération plutôt que leur valeur numérique.

Elle propose aussi plusieurs méthodes :

Méthode	Rôle
Temporal adjustInto(Temporal temporal)	Renvoyer une copie de l'objet temporel fourni en paramètre dans laquelle le jour est remplacé par celui encapsulé
static DayOfWeek from(TemporalAccessor temporal)	Obtenir une instance à partir des données encapsulées dans l'objet fourni en paramètre
int get(TemporalField field)	Obtenir la valeur du champ fourni en paramètre sous la forme d'un entier
String getDisplayName(TextStyle style, Locale locale)	Renvoyer une représentation textuelle selon le format et la Locale fournis en paramètres
long getLong(TemporalField field)	Obtenir la valeur du champ fourni en paramètre sous la forme d'un entier long
int getValue()	Renvoyer la valeur numérique associée au jour de la semaine
boolean isSupported(TemporalField field)	Vérifier si le champ fourni en paramètre est supporté
DayOfWeek minus(long days)	Renvoyer une copie de l'instance dont le nombre de jours est minoré de la valeur fournie en paramètre
static DayOfWeek of(int dayOfWeek)	Renvoyer le jour de la semaine associé à la valeur fournie en paramètre
DayOfWeek plus(long days)	Renvoyer une copie de l'instance dont le nombre de jours fourni en paramètre est ajouté
<R> R query(TemporalQuery<R> query)	Exécuter la requête passée en paramètre sur l'instance courante
ValueRange range(TemporalField field)	Obtenir la plage des valeurs valides pour le champ fourni en paramètre
static DayOfWeek valueOf(String name)	Renvoyer l'élément de l'énumération dont le nom est fourni paramètre
static DayOfWeek[] values()	Renvoyer un tableau des éléments de l'énumération dans leur ordre de déclaration

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.DayOfWeek;

public class TestDayOfWeek {

  public static void main(String[] args) {
    DayOfWeek dow = DayOfWeek.MONDAY;
    System.out.println(dow.getValue());
    System.out.println(dow.plus(4));
  }
}
```

Résultat :

```
1
FRIDAY
```

La méthode getDisplayName() permet d'obtenir le nom du jour dans le format et la Locale précisés.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.DayOfWeek;
import java.time.format.TextStyle;
import java.util.Locale;
```

```
public class TestDayOfWeek {

  public static void main(String[] args) {
    DayOfWeek dow = DayOfWeek.MONDAY;
    Locale locale = Locale.getDefault();
    System.out.println(dow.getDisplayName(TextStyle.FULL, locale));
    System.out.println(dow.getDisplayName(TextStyle.SHORT, locale));
    System.out.println(dow.getDisplayName(TextStyle.NARROW, locale));
  }
}
```

Résultat :

```
lundi
lun.
L
```

112.6.4.2. L'énumération Month

L'énumération java.time.Month définit des constantes pour les 12 mois de l'année telles que définies dans le calendrier ISO-8601 : JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER et DECEMBER.

Chaque élément de l'énumération est associé à une valeur numérique. Il est recommandé d'utiliser les éléments de l'énumération plutôt que leur valeur numérique.

Elle propose aussi plusieurs méthodes :

Méthode	Rôle
Temporal adjustInto(Temporal temporal)	Renvoyer une copie de l'objet temporel fourni en paramètre dont le mois est ajusté avec celui encapsulé
int firstDayOfYear(boolean leapYear)	Renvoyer le jour de l'année correspondant au premier jour du mois. Le paramètre booléen permet de préciser si cela concerne une année bissextile
Month firstMonthOfQuarter()	Obtenir le premier mois du trimestre auquel appartient le mois encapsulé
static Month from(TemporalAccessor temporal)	Renvoyer une instance de type Month qui correspond au mois encapsulé dans l'objet fourni en paramètre
int get(TemporalField field)	Obtenir la valeur du champ fourni en paramètre sous la forme d'un entier
String getDisplayName(TextStyle style, Locale locale)	Obtenir une représentation textuelle selon le style et la Locale fournis en paramètres
long getLong(TemporalField field)	Obtenir la valeur du champ fourni en paramètre sous la forme d'un entier long
int getValue()	Renvoyer la valeur numérique associée au mois
boolean isSupported(TemporalField field)	Vérifier si le champ fourni en paramètre est supporté
int length(boolean leapYear)	Renvoyer le nombre de jours du mois. Le paramètre booléen permet de préciser si cette valeur est celle d'une année bissextile
int maxLength()	Renvoyer le nombre maximum de jours du mois
int minLength()	Renvoyer le nombre minimum de jours du mois
Month minus(long months)	Renvoyer une copie de l'instance minorée du nombre de mois fourni en paramètre
static Month of(int month)	Renvoyer l'instance de type Month associée à la valeur numérique fournie en paramètre
Month plus(long months)	Renvoyer une copie de l'instance majorée du nombre de mois fourni en paramètre

<R> R query(TemporalQuery<R> query)	Exécuter la requête passée en paramètre sur l'instance courante
ValueRange range(TemporalField field)	Obtenir la plage des valeurs valides pour le champ fourni en paramètre. Seul MONTH_OF_YEAR renvoie une valeur comprise entre 1 et 12. Les autres champs renvoient une exception de type UnsupportedTemporalTypeException
static Month valueOf(String name)	Renvoyer la constante associée au nom fourni en paramètre
static Month[] values()	Renvoyer un tableau des constantes dans l'ordre dans lequel elles sont déclarées

Il ne faut pas utiliser la méthode ordinal() pour obtenir la valeur numérique mais utiliser la méthode getValue().

La méthode getDisplayName() permet d'obtenir le libellé en différente taille pour la Locale fournie en paramètre. La taille est fournie en paramètre grâce à un objet de type java.time.format.TextStyle qui est une énumération (FULL, FULL_STANDALONE, NARROW, NARROW_STANDALONE, SHORT, SHORT_STANDALONE).

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.Month;
import java.time.format.TextStyle;
import java.util.Locale;

public class TestMonth {
  public static void main(String[] args) {
    Month month = Month.NOVEMBER;
    Locale locale = Locale.getDefault();
    System.out.println(month.getDisplayName(TextStyle.FULL, locale));
    System.out.println(month.getDisplayName(TextStyle.SHORT, locale));
    System.out.println(month.getDisplayName(TextStyle.NARROW, locale));
  }
}
```

Résultat :

```
novembre
N
nov.
```

112.6.4.3. La classe YearMonth

La classe YearMonth encapsule un mois d'une année donnée, par exemple octobre 2014. Elle permet d'encapsuler une date partielle pour laquelle le jour n'est pas important. Par exemple, un cas d'utilisation est la date d'expiration d'une carte de crédit. Cette classe est immuable et thread-safe.

Elle possède plusieurs méthodes :

Méthode	Rôle
Temporal adjustInto(Temporal temporal)	Renvoyer une copie de l'objet temporel fourni en paramètre dont le mois et l'année sont ajustés avec ceux encapsulés
LocalDate atDay(int dayOfMonth)	Combiner les données de l'instance avec le jour du mois fourni en paramètre pour créer une instance de type LocalDate.
LocalDate atEndOfMonth()	Renvoyer une instance de type LocalDate qui encapsule la date du dernier jour du mois
int compareTo(YearMonth other)	Comparer l'instance avec celle fournie en paramètre
boolean equals(Object obj)	Tester l'égalité de l'instance avec l'objet fourni en paramètre

String format(DateTimeFormatter formatter)	Formater l'instance en utilisant le Formatter fourni en paramètre
static YearMonth from(TemporalAccessor temporal)	Obtenir une instance de type YearMonth à partir des champs encapsulés dans l'objet fourni en paramètre.
int get(TemporalField field)	Obtenir la valeur du champ demandé en paramètre sous la forme d'un entier
long getLong(TemporalField field)	Obtenir la valeur du champ demandé en paramètre sous la forme d'un entier long
Month getMonth()	Renvoyer le mois encapsulé sous la forme d'une instance de type Month
int getMonthValue()	Obtenir le mois de l'année sous la forme d'un entier compris entre 1 et 12
int getYear()	Renvoyer l'année encapsulée dans l'instance
boolean isAfter(YearMonth other)	Comparer si l'instance est postérieure à celle fournie en paramètre
boolean isBefore(YearMonth other)	Comparer si l'instance est antérieure à celle fournie en paramètre
boolean isLeapYear()	Indiquer si l'année encapsulée est bissextile
boolean isSupported(TemporalField field)	Préciser si le champ fourni en paramètre est supporté
boolean isSupported(TemporalUnit unit)	Préciser si l'unité de temps fournie en paramètre est supportée
boolean isValidDay(int dayOfMonth)	Vérifier si le jour du mois fourni en paramètre est valide pour cette instance
int lengthOfMonth()	Renvoyer le nombre de jours du mois encapsulé
int lengthOfYear()	Renvoyer le nombre de jours de l'année encapsulée
YearMonth minus(long amountToSubtract, TemporalUnit unit)YearMonth minus(TemporalAmount amountToSubtract)	Renvoyer une copie de l'instance pour laquelle la valeur temporelle fournie en paramètre est soustraite
YearMonth minusMonths(long monthsToSubtract)	Renvoyer une copie de l'instance pour laquelle le nombre de mois fourni en paramètre est soustrait
YearMonth minusYears(long yearsToSubtract)	Renvoyer une copie de l'instance pour laquelle le nombre d'années fourni en paramètre est soustrait
static YearMonth now()	Obtenir une instance encapsulant le mois et l'année de l'horloge système dans le fuseau horaire par défaut
static YearMonth now(Clock clock)	Obtenir une instance encapsulant le mois et l'année de l'horloge fournie en paramètre
static YearMonth now(ZoneId zone)	Obtenir une instance encapsulant le mois et l'année de l'horloge système dans le fuseau horaire fourni par défaut
static YearMonth of(int year, int month) static YearMonth of(int year, Month month)	Obtenir une instance encapsulant le mois et l'année fournis en paramètre
static YearMonth parse(CharSequence text)	Obtenir une instance à partir de l'analyse utilisant le DateTimeFormatter par défaut de la représentation textuelle fournie en paramètre (exemple : 2014-12)
static YearMonth parse(CharSequence text, DateTimeFormatter formatter)	Obtenir une instance à partir de l'analyse utilisant le DateTimeFormatter et la représentation textuelle fournis en paramètre
YearMonth plus(long amountToAdd, TemporalUnit	Renvoyer une copie de l'instance à laquelle la valeur

unit) YearMonth plus(TemporalAmount amountToAdd)	temporelle fournie en paramètre est ajoutée
YearMonth plusMonths(long monthsToAdd)	Renvoyer une copie de l'instance à laquelle le nombre de mois fourni en paramètre est ajouté
YearMonth plusYears(long yearsToAdd)	Renvoyer une copie de l'instance à laquelle le nombre d'années fourni en paramètre est ajouté
<R> R query(TemporalQuery<R> query)	Exécuter la requête passée en paramètre sur l'instance courante
ValueRange range(TemporalField field)	Obtenir la plage de valeurs valides pour le champ fourni en paramètre
String toString()	Obtenir une représentation textuelle de l'instance (exemple : 2014-12)
long until(Temporal endExclusive, TemporalUnit unit)	Calculer la quantité de temps entre l'instance et l'objet temporel fourni en paramètre. Le résultat est obtenu dans l'unité précisé en paramètre
YearMonth with(TemporalAdjuster adjuster)	Renvoyer une copie dont la valeur est ajustée grâce à l'objet fourni en paramètre qui encapsule les traitements
YearMonth with(TemporalField field, long newValue)	Obtenir une instance dans laquelle la valeur du champ fournie en paramètre est remplacée
YearMonth withMonth(int month)	Renvoyer une copie de l'instance encapsulant le mois précisé
YearMonth withYear(int year)	Renvoyer une copie de l'instance encapsulant l'année précisée

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.Month;
import java.time.YearMonth;

public class TestYearMonth {
  public static void main(String[] args) {
    YearMonth yearMonth = YearMonth.of(2012, Month.FEBRUARY);
    System.out.println(yearMonth+" : " + yearMonth.lengthOfMonth()+" jours");
    yearMonth = YearMonth.of(2014, Month.FEBRUARY);
    System.out.println(yearMonth+" : " + yearMonth.lengthOfMonth()+" jours");
    yearMonth = YearMonth.parse("2014-12");
    yearMonth = yearMonth.plusYears(2);
    System.out.println(yearMonth);
  }
}
```

Résultat :

```
2012-02 : 29 jours
2014-02 : 28 jours
2016-12
```

Il ne faut pas utiliser d'opérations sur une instance de type YearMonth requérant l'identité de l'objet : opérateur ==, hashCode ou synchronisation. La comparaison de deux instances doit se faire en utilisant la méthode equals().

112.6.4.4. La classe MonthDay

La classe MonthDay encapsule de manière immuable un mois et un jour dans ce mois comme par exemple le jour de l'an, le premier janvier ou un anniversaire. Cette classe est immuable et thread-safe.

Elle possède plusieurs méthodes :

Méthode	Rôle
Temporal adjustInto(Temporal temporal)	Renvoyer une copie de l'objet temporel fourni en paramètre dans laquelle les champs jour et mois sont remplacés par ceux encapsulés
LocalDate atYear(int year)	Combiner le jour et le mois encapsulé avec l'année fournie en paramètre pour obtenir une instance de type LocalDate
int compareTo(MonthDay other)	Comparer l'instance avec celle fournie en paramètre
boolean equals(Object obj)	Tester l'égalité de l'instance avec celle fournie en paramètre
String format(DateTimeFormatter formatter)	Formater l'instance en utilisant le Formatter fourni en paramètre
static MonthDay from(TemporalAccessor temporal)	Obtenir une instance de type MonthDay à partir des champs encapsulés dans l'objet fourni en paramètre
int get(TemporalField field)	Obtenir la valeur du champ fourni en paramètre
int getDayOfMonth()	Obtenir la valeur du champ day-of-month encapsulé
long getLong(TemporalField field)	Obtenir la valeur du champ fourni en paramètre
Month getMonth()	Obtenir le mois encapsulé correspondant au champ month-of-year sous la forme d'un objet de type Month
int getMonthValue()	Obtenir le mois encapsulé correspondant au champ month-of-year sous la forme d'un entier de 1 à 12
boolean isAfter(MonthDay other)	Comparer si l'instance est postérieure à celle fournie en paramètre
boolean isBefore(MonthDay other)	Comparer si l'instance est antérieure à celle fournie en paramètre
boolean isSupported(TemporalField field)	Vérifier si le champ fourni en paramètre est supporté
boolean isValidYear(int year)	Vérifier si le jour et le mois encapsulés sont valides pour l'année fournie en paramètre
static MonthDay now()	Obtenir une instance de type MonthDay à partir de la date courante du système dans le fuseau horaire par défaut
static MonthDay now(Clock clock)	Obtenir une instance de type MonthDay à partir de l'instance de type Clock fournie en paramètre
static MonthDay now(ZoneId zone)	Obtenir une instance de type MonthDay à partir de la date courante du système dans le fuseau horaire fourni en paramètre
static MonthDay of(int month, int dayOfMonth)	Obtenir une instance de type MonthDay encapsulant le jour et le mois fourni en paramètre
static MonthDay of(Month month, int dayOfMonth)	Obtenir une instance de type MonthDay encapsulant le jour et le mois fourni en paramètre
static MonthDay parse(CharSequence text)	Obtenir une instance à partir de l'analyse utilisant le DateTimeFormatter par défaut de la représentation textuelle fournie en paramètre (par exemple --02-29)
static MonthDay parse(CharSequence text, DateTimeFormatter formatter)	Obtenir une instance de type MonthDay encapsulant le jour et le mois extraits du texte en utilisant le Formatter fournis en paramètre
<R> R query(TemporalQuery<R> query)	Exécuter la requête passée en paramètre sur l'instance courante
ValueRange range(TemporalField field)	Obtenir la plage des valeurs valides pour le champ fourni en paramètre
String toString()	Obtenir une représentation textuelle de l'instance en utilisant le DateTimeFormatter par défaut (par exemple --02-29)
MonthDay with(Month month)	Renvoyer une copie de l'instance dont le mois encapsulé est celui fourni en paramètre

MonthDay withDayOfMonth(int dayOfMonth)	Renvoyer une copie de l'instance dont le jour encapsulé est celui fourni en paramètre
MonthDay withMonth(int month)	Renvoyer une copie de l'instance dont le mois encapsulé est celui fourni en paramètre

La méthode isValidYear() permet de vérifier si le jour du mois encapsulé est valide pour l'année fournie en paramètre. Attention, elle n'encapsule pas l'année : elle ne peut donc pas déterminer si l'année est bissextile et considère donc toujours comme valide le 29 février.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.Month;
import java.time.MonthDay;

public class TestMonthDay {
  public static void main(String[] args) {
    MonthDay date = MonthDay.of(Month.FEBRUARY, 29);
    System.out.println(date.isValidYear(2014));
  }
}
```

Résultat :

```
false
```

Elle implémente l'interface TemporalAccessor mais elle ne permet que l'accès aux champs MONTH_OF_YEAR et DAY_OF_MONTH.

Il ne faut pas utiliser d'opérations sur une instance de type YearMonth requérant l'identité de l'objet : opérateur ==, hashCode ou synchronisation. La comparaison de deux instances doit se faire en utilisant la méthode equals().

112.6.4.5. La classe Year

La classe Year encapsule de manière immuable une année du calendrier ISO-8601.

Elle possède plusieurs méthodes :

Méthode	Rôle
Temporal adjustInto(Temporal temporal)	Renvoyer une copie de l'objet temporel fourni en paramètre dans laquelle le champ année est remplacé par celui encapsulé
LocalDate atDay(int dayOfYear)	Combiner le jour de l'année fourni en paramètre avec l'année encapsulée pour obtenir une instance de type LocalDate
YearMonth atMonth(int month)	Combiner le mois fourni en paramètre avec l'année encapsulée pour obtenir une instance de type YearMonth
YearMonth atMonth(Month month)	Combiner le mois fourni en paramètre avec l'année encapsulée pour obtenir une instance de type YearMonth
LocalDate atMonthDay(MonthDay monthDay)	Combiner le jour et le mois fournis en paramètre avec l'année encapsulée pour obtenir une instance de type LocalDate
int compareTo(Year other)	Comparer l'instance avec l'année fournie en paramètre
boolean equals(Object obj)	Vérifier l'égalité avec l'instance fournie en paramètre
String format(DateTimeFormatter formatter)	Obtenir une représentation textuelle en utilisant le Formatter fourni en

	paramètre
static Year from(TemporalAccessor temporal)	Obtenir une instance à partir de l'objet fourni en paramètre
int get(TemporalField field)	Obtenir la valeur du champ fourni en paramètre sous la forme d'un entier
long getLong(TemporalField field)	Obtenir la valeur du champ fourni en paramètre sous la forme d'un entier long
int getValue()	Obtenir la valeur de l'année encapsulée
boolean isAfter(Year other)	Vérifier si l'année encapsulée est postérieure à celle fournie en paramètre
boolean isBefore(Year other)	Vérifier si l'année encapsulée est antérieure à celle fournie en paramètre
boolean isLeap()	Vérifier si l'année encapsulée est bissextile selon le calendrier ISO
static boolean isLeap(long year)	Vérifier si l'année est bissextile selon le calendrier ISO
boolean isSupported(TemporalField field)	Vérifier si le champ fourni en paramètre est supporté
boolean isSupported(TemporalUnit unit)	Vérifier si l'unité de temps est supportée
boolean isValidMonthDay(MonthDay monthDay)	Vérifier si le jour et le mois encapsulés dans le MonthDay fourni en paramètre est valide
int length()	Obtenir le nombre de jours de l'année encapsulée
Year minus(long amountToSubtract, TemporalUnit unit)	Renvoyer une copie de l'instance pour laquelle la quantité de temps fournie en paramètre a été soustraite
Year minus(TemporalAmount amountToSubtract)	Renvoyer une copie de l'instance pour laquelle le nombre d'années fourni en paramètre a été soustrait
Year minusYears(long yearsToSubtract)	Renvoyer une copie de l'instance pour laquelle le nombre d'années fourni en paramètre a été soustrait
static Year now()	Obtenir une instance de type Year encapsulant l'année de la date courante en utilisant le fuseau horaire par défaut
static Year now(Clock clock)	Obtenir une instance de type Year encapsulant l'année obtenue de l'instance de type Clock passée en paramètre
static Year now(ZoneId zone)	Obtenir une instance de type Year encapsulant l'année de la date courante en utilisant le fuseau horaire passé en paramètre
static Year of(int isoYear)	Obtenir une instance de type Year encapsulant l'année passée en paramètre.
static Year parse(CharSequence text)	Obtenir une instance de type Year encapsulant l'année passée en paramètre
static Year parse(CharSequence text, DateTimeFormatter formatter)	Obtenir une instance de type Year encapsulant l'année extraite du texte en utilisant le Formatter fourni en paramètre
Year plus(long amountToAdd, TemporalUnit unit) Year plus(TemporalAmount amountToAdd)	Renvoyer une copie de l'instance encapsulant une année à laquelle la valeur passée en paramètre a été ajoutée
Year plus(TemporalAmount amountToAdd)	Renvoyer une copie de l'instance encapsulant une année à laquelle la valeur passée en paramètre a été ajoutée
Year plusYears(long yearsToAdd)	Renvoyer une copie de l'instance à laquelle le nombre d'années fourni en paramètre est ajouté
<R> R query(TemporalQuery<R> query)	Exécuter la requête passée en paramètre sur l'instance courante
ValueRange range(TemporalField field)	Obtenir la plage des valeurs valides pour le champ fourni en paramètre
String toString()	Obtenir une représentation textuelle qui utilise le Formatter par défaut

long until(Temporal endExclusive, TemporalUnit unit)	Calculer la quantité de temps entre l'instance et l'objet temporel fourni en paramètre. Le résultat est obtenu dans l'unité précisée en paramètre
Year with(TemporalAdjuster adjuster)	Renvoyer une copie dont la valeur est ajustée grâce à l'objet qui encapsule les traitements fourni en paramètre
Year with(TemporalField field, long newValue)	Obtenir une instance dans laquelle la valeur du champ fourni en paramètre est remplacée

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.Year;

public class TestYear {
  public static void main(String[] args) {
    boolean estBissextile = Year.of(2016).isLeap();
    System.out.println(estBissextile);
  }
}
```

Elle implémente l'interface TemporalAccessor mais elle ne permet que l'accès aux champs YEAR_OF_ERA, YEAR et ERA.

Il ne faut pas utiliser d'opérations sur une instance de type Year requérant l'identité de l'objet : opérateur ==, hashCode ou synchronisation. La comparaison de deux instances doit se faire en utilisant la méthode equals().

112.6.5. La gestion du temps machine

Le temps machine représente une quantité de temps depuis un point d'origine : l'API Date-Time le gère sous la forme d'un entier long représentant le nombre de millisecondes écoulées depuis le 1er janvier 1970 à 00:00:00 désigné par le terme epoch.

112.6.5.1. La classe Instant

La classe java.time.Instant est une des principales classes de l'API Date-Time : elle encapsule un point sur la ligne du temps.

La classe Instant définit plusieurs constantes :

- MIN : représente l'instant minimum (dans le passé) qui peut être encapsulé
- MAX : représente l'instant maximum (dans le futur) qui peut être encapsulé

Elle possède de nombreuses méthodes pour obtenir une instance ou réaliser des opérations dessus :

Méthode	Rôle
Temporal adjustInto(Temporal temporal)	Renvoyer une copie du même type de l'objet temporel fourni en paramètre encapsulant une représentation de l'instance courante
OffsetDateTime atOffset(ZoneOffset offset)	Obtenir une instance de type OffsetDateTime qui combine l'instance courante et l'objet fourni en paramètre
ZonedDateTime atZone(ZoneId zone)	Obtenir une instance de type ZonedDateTime qui combine l'instance courante et l'objet fourni en paramètre
int compareTo(Instant otherInstant)	Comparer l'instance courante avec celle fournie en paramètre
boolean equals(Object otherInstant)	Vérifier l'égalité de l'instance courante avec celle fournie en paramètre

static Instant from(TemporalAccessor temporal)	Obtenir une instance à partir des données de l'objet fourni en paramètre
int get(TemporalField field)	Obtenir la valeur du champ fourni en paramètre sous la forme d'un entier
long getEpochSecond()	Obtenir le nombre de secondes écoulées depuis l'EPOCH
long getLong(TemporalField field)	Obtenir la valeur du champ fourni en paramètre sous la forme d'un entier long
int getNano()	Obtenir le nombre de nanosecondes écoulées depuis le début de la seconde
boolean isAfter(Instant otherInstant)	Vérifier si l'instance courante est après l'Instant fourni en paramètre dans la ligne du temps
boolean isBefore(Instant otherInstant)	Vérifier si l'instance courante est avant l'Instant fourni en paramètre dans la ligne du temps
boolean isSupported(TemporalField field)	Vérifier si le champ précisé en paramètre est supporté par l'objet
boolean isSupported(TemporalUnit unit)	Vérifier si l'unité temporelle précisée en paramètre est supportée par l'objet
Instant minus(long amountToSubtract, TemporalUnit unit) Instant minus(TemporalAmount amountToSubtract)	Obtenir une copie de l'instance minorée de la quantité temporelle fournie en paramètre
Instant minusMillis(long millisToSubtract)	Obtenir une copie de l'instance minorée du nombre de millisecondes fourni en paramètre
Instant minusNanos(long nanosToSubtract)	Obtenir une copie de l'instance minorée du nombre de nanosecondes fourni en paramètre
Instant minusSeconds(long secondsToSubtract)	Obtenir une copie de l'instance minorée du nombre de secondes fourni en paramètre
static Instant now()	Obtenir une instance qui encapsule le moment présent de l'horloge système
static Instant now(Clock clock)	Obtenir une instance qui encapsule le moment présent de l'horloge fournie en paramètre
static Instant ofEpochMilli(long epochMilli)	Obtenir une instance qui encapsule le moment correspondant au nombre de millisecondes depuis l'EPOCH
static Instant ofEpochSecond(long epochSecond)	Obtenir une instance qui encapsule le moment correspondant au nombre de secondes depuis l'EPOCH
static Instant parse(CharSequence text)	Analyser, avec le DateTimeFormatter par défaut, la chaîne de caractères fournie en paramètre pour créer une instance (Exemple : 2014-12-25T23:59:59Z)
Instant plus(long amountToAdd, TemporalUnit unit) Instant plus(TemporalAmount amountToAdd)	Obtenir une copie de l'instance majorée de la quantité temporelle fournie en paramètre
Instant plusMillis(long millisToAdd)	Obtenir une copie de l'instance majorée du nombre de millisecondes fourni en paramètre
Instant plusNanos(long nanosToAdd)	Obtenir une copie de l'instance majorée du nombre de nanosecondes fourni en paramètre
Instant plusSeconds(long secondsToAdd)	Obtenir une copie de l'instance majorée du nombre de secondes fourni en paramètre
<R> R query(TemporalQuery<R> query)	Exécuter la requête passée en paramètre sur l'instance courante

ValueRange range(TemporalField field)	Obtenir la plage de valeurs valides pour le champ précisé en paramètre
long toEpochMilli()	Obtenir le nombre de millisecondes écoulées entre l'instant encapsulé et l'EPOCH
String toString()	Obtenir une représentation textuelle au format ISO-8601 de l'instance
Instant truncatedTo(TemporalUnit unit)	Obtenir une copie de l'instance tronquée à l'unité fournie en paramètre
long until(Temporal endExclusive, TemporalUnit unit)	Calculer la quantité de temps entre l'instance et l'objet temporel fourni en paramètre. Le résultat est obtenu dans l'unité précisée en paramètre
Instant with(TemporalAdjuster adjuster)	Renvoyer une copie dont la valeur est ajustée grâce à l'objet qui encapsule les traitements fourni en paramètre
Instant with(TemporalField field, long newValue)	Renvoyer une copie de l'instance dont la valeur du champ précisé est remplacée par la valeur fournie

La classe Instant encapsule le nombre de secondes écoulées depuis le 1er janvier 1970 (1970-01-01T00:00:00Z) désigné par la constante EPOCH sous la forme d'un entier long avec une précision à la nanoseconde sous la forme d'un second entier de type int. Si l'instant encapsulé est antérieure à l'EPOCH alors la valeur est négative sinon elle est positive.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.Instant;

public class TestInstant {
  public static void main(String[] args) {
    System.out.println(Instant.EPOCH);
    Instant instant = Instant.now();
    System.out.println(instant);
  }
}
```

Résultat :

```
1970-01-01T00:00:00Z
2014-12-14T14:21:38.031Z
```

La classe Instant permet de gérer un temps machine mais ne permet pas de gérer un temps humain. Dans ce cas, il faut convertir l'Instant en une classe qui encapsule le temps de manière humaine : LocalDateTime ou ZonedDateTime par exemple.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.Instant;
import java.time.LocalDateTime;

public class TestInstant {
  public static void main(String[] args) {
    Instant instant = Instant.now();
    LocalDateTime localDateTime = LocalDateTime.ofInstant(instant,
      ZoneId.systemDefault());
  }
}
```

Un objet de type ZonedDateTime ou OffsetTimeZone peut être converti en une instance de type Instant mais pas l'inverse sans préciser respectivement le fuseau ou le décalage horaire.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.Instant;
import java.time.ZonedDateTime;

public class TestInstant {
  public static void main(String[] args) {

    ZonedDateTime zonedDateTime = ZonedDateTime.now();
    System.out.println(zonedDateTime);
    Instant instant1 = zonedDateTime.toInstant();
    System.out.println(instant1);
    Instant instant2 = Instant.from(zonedDateTime);
    System.out.println(instant2);
  }
}
```

112.6.5.2. La classe Duration

La classe java.time.Duration encapsule une durée (la distance entre deux points dans le temps) sous la forme d'une quantité temporelle. Elle encapsule une durée temporelle stockée en interne sous la forme de valeurs exprimées en secondes et nanosecondes. Elle est donc utile lors de l'utilisation de temps machine.

Remarque : la classe Period permet aussi d'encapsuler une durée mais exprimée de façon humaine sous la forme de différentes valeurs temporelles (année, mois, jours).

Attention : le JDK contient deux classes Duration respectivement dans les packages javax.xml.datatype et java.time. JavaFx possède aussi une classe Duration dans le package javafx.util.

La valeur encapsulée peut être négative si l'instant de début est plus récent que l'instant de fin.

Elle définit une constante Duration.ZERO qui correspond à une durée nulle d'un point de vue fonctionnelle.

Elle possède de nombreuses méthodes pour créer une nouvelle instance ou obtenir une instance qui soit une copie modifiée de l'instance courante :

Méthode	Rôle
Duration abs()	Obtenir une copie dont la quantité encapsulée est toujours positive
Temporal addTo(Temporal temporal)	Renvoyer une copie de l'objet temporel fourni en paramètre en lui ajoutant la durée encapsulée
static Duration between(Temporal startInclusive, Temporal endExclusive)	Obtenir une instance qui encapsule la durée entre les deux représentations temporelles fournies en paramètres incluses
int compareTo(Duration otherDuration)	Comparer l'instance courante avec celle fournie en paramètre
Duration dividedBy(long divisor)	Obtenir une copie dont la quantité encapsulée est divisée par la quantité fournie en paramètre
boolean equals(Object otherDuration)	Vérifier l'égalité de l'instance courante avec celle fournie en paramètre
static Duration from(TemporalAmount amount)	Obtenir une instance qui encapsule la durée correspondant à la quantité temporelle fournie en paramètre
long get(TemporalUnit unit)	Obtenir la valeur dans l'unité demandée
int getNano()	Obtenir le nombre de nanosecondes de la durée encapsulée
long getSeconds()	Obtenir le nombre de secondes de la durée encapsulée
List<TemporalUnit> getUnits()	Obtenir la liste des unités temporelles supportées
boolean isNegative()	Vérifier si la valeur encapsulée est strictement inférieure à zéro
boolean isZero()	Vérifier si la durée est égale à zéro

Duration minus(Duration duration) Duration minus(long amountToSubtract, TemporalUnit unit)	Obtenir une copie pour laquelle la durée est retirée de celle fournie en paramètre
Duration minusDays(long daysToSubtract)	Obtenir une copie dont la durée est minorée du nombre de jours fourni en paramètre
Duration minusHours(long hoursToSubtract)	Obtenir une copie dont la durée est minorée du nombre d'heures fourni en paramètre
Duration minusMillis(long millisToSubtract)	Obtenir une copie dont la durée est minorée du nombre de millisecondes fourni en paramètre
Duration minusMinutes(long minutesToSubtract)	Obtenir une copie dont la durée est minorée du nombre de minutes fourni en paramètre
Duration minusNanos(long nanosToSubtract)	Obtenir une copie dont la durée est minorée du nombre de nanosecondes fourni en paramètre
Duration minusSeconds(long secondsToSubtract)	Obtenir une copie dont la durée est minorée du nombre de secondes fourni en paramètre
Duration multipliedBy(long multiplicand)	Obtenir une copie dont la durée est multipliée par la valeur fournie en paramètre
Duration negated()	Obtenir une copie dont la valeur est multipliée par -1 (inversion de la valeur positive/négative)
static Duration of(long amount, TemporalUnit unit)	Obtenir une instance qui encapsule la durée correspondant à la quantité temporelle fournie en paramètre sous la forme d'une quantité et de son unité
static Duration ofDays(long days)	Obtenir une durée qui correspond au nombre de jours fournis en paramètre (une journée correspond toujours à 24 heures)
static Duration ofHours(long hours)	Obtenir une durée qui correspond au nombre d'heures fourni en paramètre
static Duration ofMillis(long millis)	Obtenir une durée qui correspond au nombre de millisecondes fourni en paramètre
static Duration ofMinutes(long minutes)	Obtenir une durée qui correspond au nombre de minutes fourni en paramètre
static Duration ofNanos(long nanos)	Obtenir une durée qui correspond au nombre de nanosecondes fourni en paramètre
static Duration ofSeconds(long seconds)	Obtenir une durée qui correspond au nombre de secondes fourni en paramètre
static Duration ofSeconds(long seconds, long nanoAdjustment)	Obtenir une durée qui correspond au nombre de secondes et nanosecondes fournis en paramètre
static Duration parse(CharSequence text)	Obtenir une instance en analysant la chaîne de caractères fournie en paramètre. Le format doit respecter le format PnDTnHnMn.nS.ou n correspond à la valeur d'un champ
Duration plus(Duration duration)	Obtenir une copie qui encapsule la durée majorée de la durée fournie en paramètre
Duration plus(long amountToAdd, TemporalUnit unit)	Obtenir une copie qui encapsule la durée majorée de la quantité temporelle fournie en paramètre sous la forme d'une quantité et de son unité
Duration plusDays(long daysToAdd)	Obtenir une copie qui encapsule la durée majorée du nombre de jours fourni en paramètre
Duration plusHours(long hoursToAdd)	Obtenir une copie qui encapsule la durée majorée du nombre d'heures fourni en paramètre
Duration plusMillis(long millisToAdd)	Obtenir une copie qui encapsule la durée majorée du nombre de millisecondes fourni en paramètre

Duration plusMinutes(long minutesToAdd)	Obtenir une copie qui encapsule la durée majorée du nombre de minutes fourni en paramètre
Duration plusNanos(long nanosToAdd)	Obtenir une copie qui encapsule la durée majorée du nombre de nanosecondes fourni en paramètre
Duration plusSeconds(long secondsToAdd)	Obtenir une copie qui encapsule la durée majorée du nombre secondes fourni en paramètre
Temporal subtractFrom(Temporal temporal)	Renvoyer une copie de l'objet temporel fourni en paramètre dont la durée a été soustraite
long toDays()	Obtenir le nombre de jours de la durée
long toHours()	Obtenir le nombre d'heures de la durée
long toMillis()	Obtenir le nombre de millisecondes de la durée
long toMinutes()	Obtenir le nombre de minutes de la durée
long toNanos()	Obtenir le nombre de nanosecondes de la durée
String toString()	Obtenir une représentation textuelle de la durée respectant le format ISO-8601
Duration withNanos(int nanoOfSecond)	Obtenir une copie de l'instance avec le nombre de nanosecondes fourni en paramètre
Duration withSeconds(long seconds)	Obtenir une copie de l'instance avec le nombre de secondes fourni en paramètre

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.Duration;
import java.time.Instant;
import java.time.temporal.ChronoUnit;
import java.time.temporal.TemporalUnit;

public class TestDuration {
  public static void main(String[] args) {

    Duration uneHeure = Duration.ofHours(1L);
    System.out.println("uneHeure = "+uneHeure);
    uneHeure = Duration.ofMinutes(60L);
    System.out.println("uneHeure = "+uneHeure);
    uneHeure = Duration.ofSeconds(3600L);
    System.out.println("uneHeure = "+uneHeure);

    System.out.println("uneHeure = " +
      uneHeure.get(ChronoUnit.SECONDS) + " secondes");

    System.out.print("TemporalUnit supportees : ");
    for (TemporalUnit tu : uneHeure.getUnits()) {
      System.out.print(tu+" ");
    }
    System.out.println();

    Instant instant1 = Instant.parse("2014-12-25T23:59:59Z");
    Instant instant2 = instant1.plus(1L, ChronoUnit.DAYS);
    Duration duree = Duration.between(instant1, instant2);
    System.out.println("duree = "+duree);
    instant1 = Instant.parse("2014-12-25T23:59:59Z");
    instant2 = Instant.parse("2011-12-25T23:59:59Z");
    duree = Duration.between(instant1, instant2);
    System.out.println("duree = "+duree);

    Duration dureeN = uneHeure.negated();
    System.out.println("uneHeure = "+uneHeure + "
      "+dureeN.isNegative());
    dureeN = dureeN.negated();
    System.out.println("uneHeure = "+uneHeure + "
      "+dureeN.isNegative());

    boolean estEgal = Duration.ofHours(1L).equals(Duration.ofSeconds(3600L));
    System.out.println("est egal = " +estEgal);

    duree = Duration.ofDays(1L);
    Instant instant = instant1.plus(duree);
    System.out.println("instant = "+instant);

    duree = Duration.parse("P2DT8H30M12S");
    System.out.println("duree = "+duree);
  }
}
```

Résultat :

```
uneHeure = PT1H
uneHeure = PT1H
uneHeure = PT1H
uneHeure = 3600 secondes
TemporalUnit supportees : Seconds Nanos
duree = PT24H
duree = PT-26304H
uneHeure = PT1H true
uneHeure = PT1H false
est egal = true
instant = 2014-12-26T23:59:59Z
duree = PT56H30M12S
```

Remarque : une journée correspond à une Duration de 24 heures. L'ajout d'une Duration correspondant à une journée à une instance de type ZonedDateTime va ajouter 24 heures quelque soit la situation : cette opération ne tiendra pas compte d'un décalage horaire comme l'heure d'été/hiver.

Le standard ISO-8601 définit le format textuel d'une durée : PTnHnMn.nS

- il commence par P
- le nombre de jours suivi par D (optionnel)
- la lettre T (Time) permet d'indiquer que la suite correspond à une heure
- le nombre d'heures suivi par H (optionnel)
- le nombre de minutes suivi par M (optionnel)
- le nombre de secondes suivi par S (optionnel)

Si le format ne peut pas être correctement analysé par la méthode parse(), celle-ci lève une exception de type DataTimeParseException avec le message « Text cannot be parsed to a Duration ».

112.6.6. La gestion du temps humain

Le temps humain est représenté sous la forme de champs entiers dans un calendrier pour la date (année, mois, jour) et d'heures, minutes, secondes, millisecondes et nanosecondes

Les classes java.time.LocalDate, java.time.LocalTime et java.time.LocalDateTime représentent des dates et/ou des heures sans fuseau horaire. Ce sont des objets temporels locaux dans le sens où ils ne sont valides que dans un contexte particulier et ne sont pas utilisables en dehors de ce contexte : ce contexte est généralement la machine sur laquelle le code s'exécute. Le préfixe Local fait référence à local par rapport au système dans lequel le code s'exécute : elles ne prennent pas en compte de fuseau ou de décalage horaire.

Les classes LocalDate, LocalTime et LocalDateTime ont différentes méthodes permettant de récupérer une partie de la date, de la tester et d'effectuer des opérations dessus qui renvoient une copie altérée de l'instance.

112.6.6.1. La classe LocalDate

La classe LocalDate encapsule de manière immuable une date sous la forme d'une année, d'un mois et d'un jour dans le calendrier ISO sans fuseau horaire. Elle est utile pour encapsuler une date sans heure, par exemple une date d'anniversaire.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.LocalDate;
import java.time.Month;

public class TestLocalDate {
  public static void main(String[] args) {
    LocalDate date = LocalDate.of(2014, Month.DECEMBER, 25);
    System.out.println(date);
  }
}
```

Résultat :

```
2014-12-25
```

Comme elle ne possède pas de fuseau horaire, elle ne représente pas un point dans le temps.

Elle possède plusieurs méthodes :

Méthode	Rôle
Temporal adjustInto(Temporal temporal)	Renvoyer une copie du même type de l'objet temporel fourni en paramètre encapsulant une représentation de l'instance courante (année, mois, jour)
LocalDateTime atStartOfDay()	Renvoyer une instance de type LocalDatetime qui combine la date encapsulée et minuit comme heure
ZonedDateTime atStartOfDay(ZoneId zone)	Renvoyer une instance de type ZonedDateTime qui combine la date encapsulée et l'heure à minuit en tenant compte du fuseau horaire passé en paramètre
LocalDateTime atTime(int hour, int minute) LocalDateTime atTime(int hour, int minute, int second) LocalDateTime atTime(int hour, int minute, int second, int nanoOfSecond) LocalDateTime atTime(LocalTime time)	Renvoyer une instance de type LocalDatetime qui combine la date encapsulée et l'heure passée en paramètre
OffsetDateTime atTime(OffsetTime time)	Renvoyer une instance de type OffsetDateTime qui combine la date encapsulée et l'OffsetTime passé en paramètre
int compareTo(ChronoLocalDate other)	Comparer la date encapsulée avec celle fournie en paramètre
boolean equals(Object obj)	Vérifier l'égalité entre la date encapsulée et l'objet fourni en paramètre
String format(DateTimeFormatter formatter)	Formater la date encapsulée en utilisant le Formatter passé en paramètre
static LocalDate from(TemporalAccessor temporal)	Obtenir une instance à partir de l'objet fourni en paramètre
int get(TemporalField field)	Obtenir un entier qui est la valeur du champ passé en paramètre de la date encapsulée
IsoChronology getChronology()	Obtenir le système calendaire (calendrier ISO)
int getDayOfMonth()	Obtenir un entier qui représente le jour du mois de la date encapsulée

DayOfWeek getDayOfWeek()	Renvoyer le jour de la semaine de la date encapsulée
int getDayOfYear()	Renvoyer un entier qui représente le numéro du jour dans l'année
Era getEra()	Renvoyer l'ère du système calendaire à laquelle la date encapsulée appartient
long getLong(TemporalField field)	Obtenir un entier long qui représente le jour du mois de la date encapsulée
Month getMonth()	Renvoyer le mois de l'année de la date encapsulée
int getMonthValue()	Renvoyer le mois de l'année de la date encapsulée sous la forme d'un entier (de 1 à 12)
int getYear()	Renvoyer l'année
boolean isAfter(ChronoLocalDate other)	Vérifier si la date encapsulée est postérieure à celle fournie en paramètre
boolean isBefore(ChronoLocalDate other)	Vérifier si la date encapsulée est antérieure à celle fournie en paramètre
boolean isEqual(ChronoLocalDate other)	Vérifier si la date encapsulée est égale à celle fournie en paramètre
boolean isLeapYear()	Vérifier si l'année de la date encapsulée est bissextile selon les règles du calendrier ISO
boolean isSupported(TemporalField field)	Vérifier si le champ temporel passé en paramètre est supporté par cette classe
boolean isSupported(TemporalUnit unit)	Vérifier si l'unité temporelle passée en paramètre est supportée par cette classe
int lengthOfMonth()	Renvoyer le nombre de jours du mois de la date encapsulée
int lengthOfYear()	Renvoyer le nombre de jours de l'année de la date encapsulée
LocalDate minus(long amountToSubtract, TemporalUnit unit) LocalDate minus(TemporalAmount amountToSubtract)	Renvoyer une copie de l'objet avec une portion de temps soustraite
LocalDate minusDays(long daysToSubtract)	Renvoyer une copie de l'objet avec le nombre de jours fourni soustrait
LocalDate minusMonths(long monthsToSubtract)	Renvoyer une copie de l'objet avec le nombre de mois fourni soustrait
LocalDate minusWeeks(long weeksToSubtract)	Renvoyer une copie de l'objet avec le nombre de semaines fourni soustrait
LocalDate minusYears(long yearsToSubtract)	Renvoyer une copie de l'objet avec le nombre d'années fourni soustrait
static LocalDate now()	Renvoyer une instance encapsulant la date courante de l'horloge système en utilisant le fuseau horaire par défaut
static LocalDate now(Clock clock)	Renvoyer une instance encapsulant la date courante de l'horloge fournie en paramètre
static LocalDate now(ZoneId zone)	Renvoyer une instance encapsulant la date courante de l'horloge système en utilisant le fuseau horaire fourni en paramètre
static LocalDate of(int year, int month, int dayOfMonth) static LocalDate of(int year, Month month, int dayOfMonth) static LocalDate ofEpochDay(long epochDay) static LocalDate ofYearDay(int year, int dayOfYear)	Obtenir une instance à partir des éléments temporels fournis en paramètres

static LocalDate parse(CharSequence text)	Obtenir une instance à partir de l'analyse de la date fournie en paramètre sous la forme d'une chaîne de caractères en utilisant le Formatter par défaut (exemple : 2014-12-25)
static LocalDate parse(CharSequence text, DateTimeFormatter formatter)	Obtenir une instance à partir de l'analyse de la date fournie en paramètre sous la forme d'une chaîne de caractères en utilisant le Formatter fourni
LocalDate plus(long amountToAdd, TemporalUnit unit) LocalDate plus(TemporalAmount amountToAdd)	Renvoyer une copie de l'objet avec une portion de temps ajoutée
LocalDate plusDays(long daysToAdd)	Renvoyer une copie de l'objet avec le nombre de jours fourni ajouté
LocalDate plusMonths(long monthsToAdd)	Renvoyer une copie de l'objet avec le nombre de mois fourni ajouté
LocalDate plusWeeks(long weeksToAdd)	Renvoyer une copie de l'objet avec le nombre de semaines fourni ajouté
LocalDate plusYears(long yearsToAdd)	Renvoyer une copie de l'objet avec le nombre d'années fourni ajouté
<R> R query(TemporalQuery<R> query)	Exécuter la requête passée en paramètre sur l'instance courante
ValueRange range(TemporalField field)	Obtenir la plage de valeurs valides pour le champ fourni en paramètre
long toEpochDay()	Renvoyer le nombre de jours entre le 01/01/1970 et la date encapsulée
String toString()	Renvoyer une représentation textuelle de la date encapsulée en utilisant le formateur par défaut (exemple : 2014-12-25)
Period until(ChronoLocalDate endDateExclusive)	Calculer la période entre la date encapsulée et celle fournie en paramètre
long until(Temporal endExclusive, TemporalUnit unit)	Calculer la quantité exprimée dans l'unité demandée entre la date encapsulée et l'objet temporel fourni en paramètre
LocalDate with(TemporalAdjuster adjuster)	Renvoyer une copie dont la valeur est ajustée grâce à l'objet qui encapsule les traitements fourni en paramètre
LocalDate with(TemporalField field, long newValue)	Renvoyer une copie modifiée de la date en prenant en compte le jour de l'année fourni
LocalDate withDayOfMonth(int dayOfMonth)	Renvoyer une copie modifiée de la date en prenant en compte le jour du mois fourni
LocalDate withDayOfYear(int dayOfYear)	Renvoyer une copie modifiée de la date en prenant en compte le jour de l'année fourni
LocalDate withMonth(int month)	Renvoyer une copie modifiée de la date en prenant en compte le mois fourni
LocalDate withYear(int year)	Renvoyer une copie modifiée de la date en prenant en compte l'année fournie

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.LocalDate;
import java.time.Month;
import java.time.ZoneId;
import java.time.temporal.ChronoField;
import java.time.temporal.ChronoUnit;

public class TestLocalDate {
  public static void main(String[] args) {
    LocalDate date = LocalDate.now();
    System.out.println("date = "+date);
    date = LocalDate.now(ZoneId.of("America/Los_Angeles"));
    System.out.println("date = "+date);
    date = LocalDate.ofEpochDay(16428);
    System.out.println("date = "+date);
    date = LocalDate.of(2014, Month.DECEMBER, 25);
    System.out.println("date = "+date);

    System.out.println("Calendrier = "+date.getChronology());
    System.out.println("startOfDay = " + date.atStartOfDay(ZoneId.systemDefault()));
    System.out.println("dayOfMonth = "+date.getDayOfMonth());
    System.out.println("dayOfWeek = "+date.getDayOfWeek());
    System.out.println("dayOfYear = "+date.getDayOfYear());
    System.out.println("ere = "+date.getEra());
    System.out.println("support HOURS = "+date.isSupported(ChronoUnit.HOURS));
    System.out.println("support HOUR_Of_DAY =
      "+date.isSupported(ChronoField.HOUR_OF_DAY));
    System.out.println("lengthOfMonth = "+date.lengthOfMonth());
    System.out.println("lengthOfYear = "+date.lengthOfYear());
    System.out.println("2 semaines avant = "+date.minusWeeks(2));
  }
}
```

Résultat :

```
date = 2015-02-19
date = 2015-02-19
date = 2014-12-24
date = 2014-12-25
Calendrier = ISO
startOfDay = 2014-12-25T00:00+01:00[Europe/Paris]
dayOfMonth = 25
dayOfWeek = THURSDAY
dayOfYear = 359
ere = CE
support HOURS = false
support HOUR_Of_DAY = false
lengthOfMonth = 31
lengthOfYear = 365
2 semaines avant = 2014-12-11
```

La classe LocalDate permet aussi un accès aux champs : day-of-year, day-of-week et week-of-year.

Il ne faut pas utiliser d'opérations sur une instance de type LocalDate requérant l'identité de l'objet : opérateur ==, hashCode ou synchronisation. La comparaison de deux instances doit se faire en utilisant la méthode equals().

Remarque : il n'est pas possible de convertir une instance de type LocalDate en Instant car elle ne contient pas de données relatives à l'heure qui permettrait de déterminer le point dans le temps avec une précision à la seconde.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.DateTimeException;
import java.time.Instant;
import java.time.LocalDate;

public class TestLocalDate {
  public static void main(String[] args) {
    try {
      LocalDate date = LocalDate.of(2014, 12, 25);
      Instant instant = Instant.from(date);
    } catch (DateTimeException dte) {
      dte.printStackTrace();
    }
  }
}
```

Résultat :

```
java.time.DateTimeException: Unable to obtain Instant from
TemporalAccessor: 2014-12-25 of type java.time.LocalDate
```

```
            at java.time.Instant.from(Instant.java:383)
            at fr.jmdoudoux.dej.java8.datetime.TestLocalDate.main(TestLocalDate.java:13)
```

112.6.6.2. La classe LocalDateTime

La classe LocalDateTime encapsule une date (année, mois, jour) et une heure (heure, minute, seconde, nanoseconde) sans fuseau horaire.

Cette classe est immuable et thread-safe.

Elle possède plusieurs méthodes :

Méthode	Rôle
Temporal adjustInto(Temporal temporal)	Renvoyer une copie de l'objet temporel fourni en paramètre dans lequel les champs sont remplacés par ceux encapsulés
OffsetDateTime atOffset(ZoneOffset offset)	Obtenir une instance qui combine l'instance courante avec le décalage horaire passé en paramètre
ZonedDateTime atZone(ZoneId zone)	Obtenir une instance qui combine l'instance courante avec le fuseau horaire passé en paramètre
int compareTo(ChronoLocalDateTime<?> other)	Comparer l'instance avec l'objet temporel passé en paramètre
boolean equals(Object obj)	Vérifier l'égalité de l'instance avec l'objet passé en paramètre
String format(DateTimeFormatter formatter)	Formater l'instance pour obtenir une représentation textuelle dans le format précisé par l'objet passé en paramètre
static LocalDateTime from(TemporalAccessor temporal)	Obtenir une instance à partir de l'objet temporel fourni en paramètre
int get(TemporalField field)	Obtenir la valeur du champ passé en paramètre sous la forme d'un entier
int getDayOfMonth()	Obtenir la valeur du champ jour du mois
DayOfWeek getDayOfWeek()	Obtenir la valeur du champ jour de la semaine encapsulé dans une instance de type DayOfWeek
int getDayOfYear()	Obtenir la valeur du champ jour de l'année
int getHour()	Obtenir la valeur du champ heures
long getLong(TemporalField field)	Obtenir la valeur du champ fourni en paramètre
int getMinute()	Obtenir la valeur du champ minutes
Month getMonth()	Obtenir la valeur du champ mois encapsulé dans une instance de type Month
int getMonthValue()	Obtenir la valeur du champ mois (1-12)
int getNano()	Obtenir la valeur du champ nanosecondes
int getSecond()	Obtenir la valeur du champ secondes
int getYear()	Obtenir la valeur du champ année
boolean isAfter(ChronoLocalDateTime<?> other)	Vérifier si l'instance courante est postérieure à l'objet temporel fourni en paramètre
boolean isBefore(ChronoLocalDateTime<?> other)	Vérifier si l'instance courante est antérieure à l'objet temporel fourni en paramètre

boolean isEqual(ChronoLocalDateTime<?> other)	Vérifier si l'instance est égale à l'objet temporel fourni en paramètre
boolean isSupported(TemporalField field)	Vérifier si le champ passé en paramètre est supporté
boolean isSupported(TemporalUnit unit)	Vérifier si l'unité temporel fournie en paramètre est supportée
LocalDateTime minus(long amountToSubtract, TemporalUnit unit) LocalDateTime minus(TemporalAmount amountToSubtract)	Renvoyer une copie pour laquelle la quantité temporelle passé en paramètre est soustraite
LocalDateTime minusDays(long days)	Renvoyer une copie pour laquelle le nombre de jours passé en paramètre est soustrait
LocalDateTime minusHours(long hours)	Renvoyer une copie pour laquelle le nombre d'heures passé en paramètre est soustrait
LocalDateTime minusMinutes(long minutes)	Renvoyer une copie pour laquelle le nombre de minutes passé en paramètre est soustrait
LocalDateTime minusMonths(long months)	Renvoyer une copie pour laquelle le nombre de mois passé en paramètre est soustrait
LocalDateTime minusNanos(long nanos)	Renvoyer une copie pour laquelle le nombre de nanosecondes passé en paramètre est soustrait
LocalDateTime minusSeconds(long seconds)	Renvoyer une copie pour laquelle le nombre de secondes passé en paramètre est soustrait
LocalDateTime minusWeeks(long weeks)	Renvoyer une copie pour laquelle le nombre de semaines passé en paramètre est soustrait
LocalDateTime minusYears(long years)	Renvoyer une copie pour laquelle le nombre d'années passé en paramètre est soustrait
static LocalDateTime now()	Obtenir une instance à partir de la date/heure système et du fuseau horaire par défaut
static LocalDateTime now(Clock clock)	Obtenir une instance à partir de la date/heure donné par l'objet en paramètre
static LocalDateTime now(ZoneId zone)	Obtenir une instance à partir de la date/heure système et du fuseau horaire passé en paramètre
static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute) static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute, int second) static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute, int second, int nanoOfSecond) static LocalDateTime of(int year, Month month, int dayOfMonth, int hour, int minute) static LocalDateTime of(int year, Month month, int dayOfMonth, int hour, int minute, int second) static LocalDateTime of(int year, Month month, int dayOfMonth, int hour, int minute, int second, int nanoOfSecond)	Obtenir une instance à partir des différentes valeurs des champs passées en paramètres
static LocalDateTime of(LocalDate date, LocalTime time)	Obtenir une instance à partir de la date et de l'heure passées en paramètres
static LocalDateTime ofEpochSecond(long epochSecond, int nanoOfSecond, ZoneOffset offset)	Obtenir une instance qui encapsulera la date-heure issue du calcul de l'ajout du nombre de secondes à l'EPOCH et de la prise en compte du décalage horaire passé en paramètre

static LocalDateTime ofInstant(Instant instant, ZoneId zone)	Obtenir une instance à partir d'un point dans le temps et d'un fuseau horaire fournis en paramètres
static LocalDateTime parse(CharSequence text)	Obtenir une instance à partir de l'analyse de la chaîne de caractères passée en paramètre. Exemple 2014-12-25T23:59:59
static LocalDateTime parse(CharSequence text, DateTimeFormatter formatter)	Obtenir une instance à partir de l'analyse de la date fournie en paramètre sous la forme d'une chaîne de caractères en utilisant le Formatter fourni
LocalDateTime plus(long amountToAdd, TemporalUnit unit) LocalDateTime plus(TemporalAmount amountToAdd)	Renvoyer une copie à laquelle la quantité temporelle passée en paramètre est ajoutée
LocalDateTime plusDays(long days)	Renvoyer une copie à laquelle le nombre de jours passé en paramètre est ajouté
LocalDateTime plusHours(long hours)	Renvoyer une copie à laquelle le nombre d'heures passé en paramètre est ajouté
LocalDateTime plusMinutes(long minutes)	Renvoyer une copie à laquelle le nombre de minutes passé en paramètre est ajouté
LocalDateTime plusMonths(long months)	Renvoyer une copie à laquelle le nombre de mois passé en paramètre est ajouté
LocalDateTime plusNanos(long nanos)	Renvoyer une copie à laquelle le nombre de nanosecondes passé en paramètre est ajouté
LocalDateTime plusSeconds(long seconds)	Renvoyer une copie à laquelle le nombre de secondes passé en paramètre est ajouté
LocalDateTime plusWeeks(long weeks)	Renvoyer une copie à laquelle le nombre de semaine passé en paramètre est ajouté
LocalDateTime plusYears(long years)	Renvoyer une copie à laquelle le nombre d'année passé en paramètre est ajouté
<R> R query(TemporalQuery<R> query)	Exécuter la requête passée en paramètre sur l'instance courante
ValueRange range(TemporalField field)	Obtenir la plage des valeurs acceptables pour le champ fourni en paramètre
LocalDate toLocalDate()	Obtenir une instance de type LocalDate qui contient l'heure encapsulée
LocalTime toLocalTime()	Obtenir une instance de type LocalTime qui contient l'heure encapsulée
String toString()	Obtenir une représentation textuelle de l'instance
LocalDateTime truncatedTo(TemporalUnit unit)	Renvoyer une copie de l'instance dont les valeurs encapsulées sont tronquées à l'unité précisée en paramètre
long until(Temporal endExclusive, TemporalUnit unit)	Calculer la quantité de temps entre l'instance et l'objet temporel fourni en paramètre. Le résultat est obtenu dans l'unité précisée en paramètre
LocalDateTime with(TemporalAdjuster adjuster)	Renvoyer une copie dont la valeur est ajustée grâce à l'objet qui encapsule les traitements fourni en paramètre
LocalDateTime with(TemporalField field, long newValue)	Renvoyer une copie de l'instance dont la valeur du champ précisé en paramètre est celle fournie
LocalDateTime withDayOfMonth(int dayOfMonth)	Renvoyer une copie de l'instance dont le jour du mois est celui fourni en paramètre
LocalDateTime withDayOfYear(int dayOfYear)	

	Renvoyer une copie de l'instance dont le jour de l'année est celui fourni en paramètre
LocalDateTime withHour(int hour)	Renvoyer une copie de l'instance dont l'heure est celle fournie en paramètre
LocalDateTime withMinute(int minute)	Renvoyer une copie de l'instance dont les minutes sont celles fournies en paramètre
LocalDateTime withMonth(int month)	Renvoyer une copie de l'instance dont le mois est celui fourni en paramètre
LocalDateTime withNano(int nanoOfSecond)	Renvoyer une copie de l'instance dont les nanosecondes sont celles fournies en paramètre
LocalDateTime withSecond(int second)	Renvoyer une copie de l'instance dont les secondes de la minute sont celles fournies en paramètre
LocalDateTime withYear(int year)	Renvoyer une copie de l'instance dont l'année est celle fournie en paramètre

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.LocalDateTime;
import java.time.Month;
import java.time.ZoneId;
import java.time.temporal.ChronoField;
import java.time.temporal.ChronoUnit;

public class TestLocalDateTime {
  public static void main(String[] args) {
    LocalDateTime date = LocalDateTime.now();
    System.out.println("dateTime = "+date);
    date = LocalDateTime.now(ZoneId.of("America/Los_Angeles"));
    System.out.println("dateTime = "+date);
    date = LocalDateTime.of(2014, Month.DECEMBER, 25,23,59,59);
    System.out.println("dateTime = "+date);
    System.out.println("date = "+date.toLocalDate());
    System.out.println("time = "+date.toLocalTime());

    System.out.println("Calendrier = "+date.getChronology());
    System.out.println("dayOfMonth = "+date.getDayOfMonth());
    System.out.println("dayOfWeek = "+date.getDayOfWeek());
    System.out.println("dayOfYear = "+date.getDayOfYear());
    System.out.println("support HOURS = "+date.isSupported(ChronoUnit.HOURS));
    System.out.println("support HOUR_Of_DAY = "
      +date.isSupported(ChronoField.HOUR_OF_DAY));
    System.out.println("support INSTANT_SECONDS =
      "+date.isSupported(ChronoField.INSTANT_SECONDS));
    System.out.println("hour = "+date.getHour());
    System.out.println("2 semaines avant = "+date.minusWeeks(2));
    System.out.println("1 heure avant = "+date.minusHours(1));
  }
}
```

Résultat :

```
dateTime = 2015-02-20T21:07:48.406
dateTime = 2015-02-20T12:07:48.437
dateTime = 2014-12-25T23:59:59
date = 2014-12-25
time = 23:59:59
Calendrier = ISO
dayOfMonth = 25
dayOfWeek = THURSDAY
dayOfYear = 359
support HOURS = true
support HOUR_Of_DAY = true
support INSTANT_SECONDS = false
hour = 23
```

```
2 semaines avant = 2014-12-11T23:59:59
1 heure avant = 2014-12-25T22:59:59
```

Il ne faut pas utiliser d'opérations sur une instance de type LocalDateTime requérant l'identité de l'objet : opérateur ==, hashCode ou synchronisation. La comparaison de deux instances doit se faire en utilisant la méthode equals().

112.6.6.3. La classe LocalTime

La classe LocalTime encapsule une heure (heure, minute, seconde, nanoseconde) sans fuseau horaire. Cette classe est immuable et thread-safe.

Elle définit plusieurs constantes :

Valeur	Rôle
static LocalTime MAX	la plus grande valeur encapsulée, '23:59:59.999999999'
static LocalTime MIDNIGHT	Minuit au début de la journée, '00:00'
static LocalTime MIN	la plus petite valeur encapsulée, '00:00'
static LocalTime NOON	Midi au milieu de la journée, '12:00'

Elle possède de nombreuses méthodes :

Méthode	Rôle
Temporal adjustInto(Temporal temporal)	Renvoyer une copie de l'objet temporel fourni en paramètre dans lequel les champs sont remplacés par ceux encapsulés
LocalTime atDate(LocalDate date)	Obtenir une instance qui combine l'instance courante et la date fournie en paramètre
OffsetTime atOffset(ZoneOffset offset)	Obtenir une instance qui combine l'instance courante avec le décalage horaire passé en paramètre
int compareTo(LocalTime other)	Comparer l'instance avec l'instance passée en paramètre
boolean equals(Object obj)	Vérifier l'égalité de l'instance avec l'objet passé en paramètre
String format(DateTimeFormatter formatter)	Formater l'instance pour obtenir une représentation textuelle dans le format précisé par l'objet passé en paramètre
static LocalTime from(TemporalAccessor temporal)	Obtenir une instance à partir de l'objet temporel fourni en paramètre
int get(TemporalField field)	Obtenir la valeur du champ passé en paramètre sous la forme d'un entier
int getDayOfMonth()	Obtenir la valeur du champ jour du mois
long getLong(TemporalField field)	Obtenir la valeur du champ fourni en paramètre sous la forme d'un entier long
int getMinute()	Obtenir la valeur du champ minutes
int getNano()	Obtenir la valeur du champ nanosecondes
int getSecond()	Obtenir la valeur du champ secondes
int getYear()	Obtenir la valeur du champ année
boolean isAfter(LocalTime other)	Vérifier si l'instance courante est postérieure à l'objet temporel fourni en paramètre

boolean isBefore(LocalTime other)	Vérifier si l'instance courante est antérieure à l'objet temporel fourni en paramètre
boolean isSupported(TemporalField field)	Vérifier si le champ passé en paramètre est supporté
boolean isSupported(TemporalUnit unit)	Vérifier si l'unité temporel fournie en paramètre est supportée
LocalTime minus(long amountToSubtract, TemporalUnit unit) LocalTime minus(TemporalAmount amountToSubtract)	Renvoyer une copie pour laquelle la quantité temporelle passée en paramètre est soustraite
LocalTime minusHours(long hours)	Renvoyer une copie pour laquelle le nombre d'heures passé en paramètre est soustrait
LocalTime minusMinutes(long minutes)	Renvoyer une copie pour laquelle le nombre de minutes passé en paramètre est soustrait
LocalTime minusNanos(long nanos)	Renvoyer une copie pour laquelle le nombre de nanosecondes passé en paramètre est soustrait
LocalTime minusSeconds(long seconds)	Renvoyer une copie pour laquelle le nombre de secondes passé en paramètre est soustrait
static LocalTime now()	Obtenir une instance à partir de la date/heure système et du fuseau horaire par défaut
static LocalTime now(Clock clock)	Obtenir une instance à partir de la date/heure donnée par l'objet en paramètre
static LocalTime now(ZoneId zone)	Obtenir une instance à partir de la date/heure système et du fuseau horaire passé en paramètre
static LocalTime of(int hour, int minute) static LocalTime of(int hour, int minute, int second) static LocalTime of(int hour, int minute, int second, int nanoOfSecond)	Obtenir une instance à partir des différentes valeurs des champs passées en paramètres
static LocalTime ofNanoOfDay(long nanoOfDay)	Obtenir une instance à partir des nanosecondes de la journée passées en paramètre
static LocalTime ofSecondOfDay(long secondOfDay)	Obtenir une instance à partir des secondes de la journée passées en paramètre
static LocalTime parse(CharSequence text)	Obtenir une instance à partir de l'analyse de la chaîne de caractères passée en paramètres avec le Formatter par défaut. Exemple 14:39:59
static LocalTime parse(CharSequence text, DateTimeFormatter formatter)	Obtenir une instance à partir de l'analyse de la chaîne de caractères passée en paramètres avec le Formatter fourni en paramètre
LocalTime plus(long amountToAdd, TemporalUnit unit) LocalTime plus(TemporalAmount amountToAdd)	Renvoyer une copie à laquelle la quantité temporelle passée en paramètre est ajoutée
LocalTime plusHours(long hours)	Renvoyer une copie à laquelle le nombre d'heures passé en paramètre est ajouté
LocalTime plusMinutes(long minutes)	Renvoyer une copie à laquelle le nombre de minutes passé en paramètre est ajouté
LocalTime plusNanos(long nanos)	Renvoyer une copie à laquelle le nombre de nanosecondes passé en paramètre est ajouté
LocalTime plusSeconds(long seconds)	Renvoyer une copie à laquelle le nombre de secondes passé en paramètre est ajouté
<R> R query(TemporalQuery<R> query)	Exécuter la requête passée en paramètre sur l'instance courante
ValueRange range(TemporalField field)	

	Obtenir la plage des valeurs acceptables pour le champ fourni en paramètre
long toNanoOfDay()	Obtenir le nombre de nanosecondes de la journée
int toSecondOfDay()	Obtenir le nombre de secondes de la journée
String toString()	Obtenir une représentation textuelle de l'instance
LocalTime truncatedTo(TemporalUnit unit)	Renvoyer une copie de l'instance dont les valeurs encapsulées sont tronquées à l'unité précisée en paramètre
long until(Temporal endExclusive, TemporalUnit unit)	Calculer la quantité de temps entre l'instance et l'objet temporel fourni en paramètre. Le résultat est obtenu dans l'unité précisé en paramètre
LocalTime with(TemporalAdjuster adjuster)	Renvoyer une copie dont la valeur est ajustée grâce à l'objet qui encapsule les traitements fourni en paramètre
LocalTime with(TemporalField field, long newValue)	Renvoyer une copie de l'instance dont la valeur du champ précisé en paramètre est celle fournie
LocalTime withHour(int hour)	Renvoyer une copie de l'instance dont l'heure est celle fournie en paramètre
LocalTime withMinute(int minute)	Renvoyer une copie de l'instance dont les minutes sont celles fournies en paramètre
LocalTime withNano(int nanoOfSecond)	Renvoyer une copie de l'instance dont les nanosecondes sont celles fournies en paramètre
LocalTime withSecond(int second)	Renvoyer une copie de l'instance dont les secondes sont celles fournies en paramètre

Exemple (code Java 8) :

```java
package fr.jmdoudoux.dej.java8.datetime;

import java.time.LocalTime;
import java.time.temporal.ChronoField;
import java.time.temporal.ChronoUnit;

public class TestLocalTime {
  public static void main(String[] args) {
    LocalTime time = LocalTime.now();
    System.out.println("time = "+time);
    time = LocalTime.of(23,59,59,104534);
    System.out.println("time = "+time);

    System.out.println("support HOURS = " +time.isSupported(ChronoUnit.HOURS));
    System.out.println("support HOUR_Of_DAY = "
      +time.isSupported(ChronoField.HOUR_OF_DAY));
    System.out.println("supportINSTANT_SECONDS = "
      +time.isSupported(ChronoField.INSTANT_SECONDS));

    System.out.println("hour = "+time.getHour());
    System.out.println("1 heure avant = " +time.minusHours(1));
  }
}
```

Résultat :

```
time = 06:53:21.625
time = 23:59:59.000104534
support HOURS = true
support HOUR_Of_DAY = true
support INSTANT_SECONDS = false
hour = 23
1 heure avant = 22:59:59.000104534
```

Il ne faut pas utiliser d'opérations sur une instance de type LocalTime requérant l'identité de l'objet : opérateur ==, hashCode ou synchronisation. La comparaison de deux instances doit se faire en utilisant la méthode equals().

112.6.6.4. La classe Period

La classe Period encapsule une période dans le temps : c'est une quantité de temps exprimée sous la forme des différents champs permettant de représenter une période de manière humaine, telle qu'un an, cinq mois et huit jours.

Les unités temporelles supportées sont année, mois et jour : chaque quantité encapsulée est représentée avec une quantité dans ces trois unités, leur valeur pouvant être égale à zéro. Ces valeurs peuvent aussi être négatives. Plusieurs méthodes permettent d'obtenir la quantité dans les différentes unités de temps requises pour représenter cette quantité.

Elle possède de nombreuses méthodes pour obtenir une instance ou renvoyer une copie modifiée :

Méthode	Rôle
Temporal addTo(Temporal temporal)	Renvoyer une copie de l'objet temporel fourni en paramètre auquel la période a été ajoutée
static Period between(LocalDate startDateInclusive, LocalDate endDateExclusive)	Calculer la période entre les deux dates fournies en paramètres
boolean equals(Object obj)	Vérifier si l'instance courante est égale à l'objet fourni en paramètre
static Period from(TemporalAmount amount)	Obtenir une instance qui encapsule la période correspondant à la quantité temporelle fournie en paramètre
long get(TemporalUnit unit)	Obtenir la durée de la période dans l'unité fournie en paramètre
IsoChronology getChronology()	Obtenir le calendrier de l'instance (ISO)
int getDays()	Obtenir le nombre de jours de l'instance
int getMonths()	Obtenir le nombre de mois de l'instance
List<TemporalUnit> getUnits()	Obtenir la liste des unités supportées
int getYears()	Obtenir le nombre d'années de l'instance
boolean isNegative()	Vérifier si la valeur d'au moins un des champs est négative
boolean isZero()	Vérifier que la valeur de tous les champs est zéro
Period minus(TemporalAmount amountToSubtract)	Retourner une copie de l'instance pour laquelle la quantité temporelle fournie en paramètre est soustraite
Period minusDays(long daysToSubtract)	Retourner une copie de l'instance pour laquelle le nombre de jours fourni en paramètre est soustrait
Period minusMonths(long monthsToSubtract)	Retourner une copie de l'instance pour laquelle le nombre de mois fourni en paramètre est soustrait
Period minusYears(long yearsToSubtract)	Retourner une copie de l'instance pour laquelle le nombre d'année fourni en paramètre est soustrait
Period multipliedBy(int scalar)	Retourner une copie de l'instance pour laquelle la valeur de tous les champs est multipliée par la valeur fournie en paramètre
Period negated()	Renvoyer une nouvelle instance dans laquelle toutes les valeurs des champs sont négatives
Period normalized()	Retourner une copie de la période avec les années et les mois normalisés
static Period of(int years, int months, int days)	Obtenir une instance encapsulant le nombre d'années, mois et jours fournis en paramètres

static Period ofDays(int days)	Obtenir une instance encapsulant le nombre de jours fourni en paramètre
static Period ofMonths(int months)	Obtenir une instance encapsulant le nombre de mois fourni en paramètre
static Period ofWeeks(int weeks)	Obtenir une instance encapsulant le nombre de semaines fourni en paramètre
static Period ofYears(int years)	Obtenir une instance encapsulant le nombre d'années fourni en paramètre
static Period parse(CharSequence text)	Renvoyer une instance issue de l'analyse de la chaîne de caractères fournie en paramètre, exemple P1Y5M8D
Period plus(TemporalAmount amountToAdd)	Retourner une copie à laquelle la quantité temporelle fournie en paramètre est ajoutée
Period plusDays(long daysToAdd)	Retourner une copie à laquelle le nombre de jours fourni en paramètre est ajouté
Period plusMonths(long monthsToAdd)	Retourner une copie à laquelle le nombre de mois fourni en paramètre est ajouté
Period plusYears(long yearsToAdd)	Retourner une copie à laquelle le nombre d'années fourni en paramètre est ajouté
Temporal subtractFrom(Temporal temporal)	Renvoyer une copie de l'objet temporel fourni en paramètre pour laquelle la période est soustraite
String toString()	Obtenir une représentation textuelle de la période, exemple P1Y5M8D
long toTotalMonths()	Obtenir le nombre total de mois de la période
Period withDays(int days)	Renvoyer une copie de la période ajustée avec le nombre de jours fourni en paramètre
Period withMonths(int months)	Renvoyer une copie de la période ajustée avec le nombre de mois fourni en paramètre
Period withYears(int years)	Renvoyer une copie de la période ajustée avec le nombre d'années fourni en paramètre

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.LocalDate;
import java.time.Period;
import java.time.temporal.ChronoUnit;

public class TestPeriod {
  public static void main(String[] args) {
    Period period = Period.of(1, 2, 3);
    System.out.println("period=" + period);
    System.out.println("annees=" + period.getYears());
    System.out.println("mois  =" + period.getMonths());
    System.out.println("jours =" + period.getDays());

    System.out.println("annees=" + period.get(ChronoUnit.YEARS));
    System.out.println("mois  =" + period.get(ChronoUnit.MONTHS));
    System.out.println("jours =" + period.get(ChronoUnit.DAYS));

    System.out.println("-1 jour=" + period.minusDays(1));
    System.out.println("nb mois="+period.toTotalMonths());

    LocalDate debut = LocalDate.of(2015, 1, 1);
    LocalDate fin = LocalDate.of(2016, 3, 4);
    System.out.println("ecart deb fin ="+Period.between(debut, fin));
    System.out.println("ecart fin deb ="+Period.between(fin, debut));
  }
}
```

Résultat :

```
period=P1Y2M3D
annees=1
mois  =2
jours =3
annees=1
mois  =2
jours =3
-1 jour=P1Y2M2D
nb mois=14
ecart deb fin =P1Y2M3D
ecart fin deb =P-1Y-2M-3D
```

La période encapsulée représente toujours la somme des valeurs des trois champs.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.LocalDate;
import java.time.Month;
import java.time.Period;

public class TestPeriod {
  public static void main(String[] args) {
    LocalDate dateNaissance = LocalDate.of(1972, Month.JUNE, 5);
    final LocalDate prochainAnniversaire = prochainAnniversaire(dateNaissance);
    System.out.println("prochain anniversaire="+prochainAnniversaire);

    Period period = Period.between(LocalDate.now(), prochainAnniversaire);
    System.out.println("aura lieu dans "+period.getYears()
      +" ans, "+ period.getMonths()+" mois et "
      +period.getDays()+" jours");
  }

  public static LocalDate prochainAnniversaire(LocalDate dateNaissance) {
    LocalDate aujourdhui = LocalDate.now();
    LocalDate resultat = dateNaissance.withYear(aujourdhui.getYear());
    if (resultat.isBefore(aujourdhui) || resultat.isEqual(aujourdhui)) {
      resultat = resultat.plusYears(1);
    }
    return resultat;
  }
}
```

Résultat :

```
prochain anniversaire=2015-06-05
aura lieu dans 0 ans, 3 mois et 2 jours
```

Remarque : pour obtenir une période sous la forme d'une seule unité temporelle, il faut utiliser la méthode between() de la classe ChronoUnit.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.LocalDate;
import java.time.temporal.ChronoUnit;

public class TestChronoUnit {
  public static void main(String[] args) {
    LocalDate debut = LocalDate.of(2015, 1, 1);
    LocalDate fin = LocalDate.of(2015, 1, 2);
    long nbJour = ChronoUnit.DAYS.between(debut, fin);
    System.out.println("nbjour="+nbJour);
  }
}
```

Résultat :

```
nbjour=1
```

Remarque : la classe Duration encapsule aussi une quantité de temps mais sous une forme machine. Les classes Period et Duration gère le décalage horaire différemment lorsqu'elles sont ajoutées à un objet de type ZonedDateTime : Duration ajoute un nombre déterminé de secondes alors que Period tente d'être le plus proche possible de l'heure.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.Duration;
import java.time.LocalDateTime;
import java.time.Period;
import java.time.ZoneId;
import java.time.ZonedDateTime;

public class TestPeriod {
  public static void main(String[] args) {
    Duration unJourD = Duration.ofDays(1);
    Period unJourP = Period.ofDays(1);

    ZonedDateTime dateTime = ZonedDateTime.of(
      LocalDateTime.of(2015, 3, 28, 17, 00), ZoneId.systemDefault());
    System.out.println("datetime         ="+dateTime);
    System.out.println("+ periode 1 jour ="+dateTime.plus(unJourP));
    System.out.println("+ duree 1 jour   ="+dateTime.plus(unJourD));
  }
}
```

Résultat :

```
datetime =2015-03-28T17:00+01:00[Europe/Paris]
+ periode 1 jour =2015-03-29T17:00+02:00[Europe/Paris]
+ duree 1 jour   =2015-03-29T18:00+02:00[Europe/Paris]
```

Period est une classe de type value-based : il ne faut pas utiliser d'opérations sur une instance de type Period requérant l'identité de l'objet : opérateur ==, hashCode ou synchronisation. La comparaison de deux instances doit se faire en utilisant la méthode equals().

112.6.7. Les fuseaux et les décalages horaires

La gestion des fuseaux et des décalages horaires rend beaucoup plus complexe la gestion des objets temporaux.

La surface de la Terre est découpée virtuellement en portions appelées fuseaux horaires : dans chacune de ces portions des règles s'appliquent et l'heure communément utilisée est la même. Il existe une quarantaine de fuseaux horaires.

Chaque fuseau horaire possède :

- un identifiant court (exemple : CET)
- un identifiant long généralement sous la forme région/ville (exemple : «Europe/Paris»)
- des règles qui permettent de déterminer l'heure grâce à un décalage par rapport à l'heure du méridien de Greenwich, aussi désignée par UTC. Ces règles peuvent définir certaines spécificités comme l'utilisation de l'heure d'été/hiver.

L'API Date-time propose deux classes pour encapsuler un fuseau horaire ou un décalage horaire :

- ZoneId : encapsule un fuseau horaire qui possède un décalage horaire et des règles pour déterminer l'heure utilisée dans ce fuseau
- ZoneOffset : encapsule un décalage horaire

Avant Java 8, la classe java.util.TimeZone encapsulait un fuseau horaire. C'est toujours le cas mais l'API Date-Time ne l'utilise pas et propose la nouvelle classe ZoneId car TimeZone est mutable.

L'API Date-Time propose plusieurs classes temporelles qui prennent en charge un fuseau horaire :

- ZonedDateTime : encapsule une date-heure avec un fuseau horaire
- OffsetDateTime : encapsule une date-heure avec un décalage horaire
- OffsetTime : encapsule une heure avec un décalage horaire

Le plus simple est d'utiliser une ZonedDateTime qui gère le décalage horaire sur la base des règles encapsulées dans le ZoneId prenant par exemple en charge l'heure d'été/hiver. Les classes OffsetDatetime et OffsetTime gère un décalage fixe. Généralement les formats d'échanges tels que XML ou de stockage comme une base de données utilisent une OffsetDateTime ou OffsetTime.

112.6.7.1. La classe ZoneId

La classe abstraite ZoneId encapsule un fuseau horaire identifié par son id unique : exemple « Europe/Paris»

Elle contient les règles pour transformer un Instant en LocalDateTime.

Il existe deux types de fuseaux horaires chacun encapsulé dans une classe fille :

- décalage fixe (fixed offset) encapsulé dans le classe ZoneOffset : le décalage horaire par rapport à l'heure de Greenwich est le même pour tous les endroits du fuseau
- régions géographiques encapsulés dans le classe ZoneRegion : des règles particulières s'appliquent sur le décalage horaire

La classe ZoneRules contient pour chaque id les règles à appliquer : celles-ci changent fréquemment car elles sont définies par chaque pays.

Il existe trois types d'id :

- simple : correspond à celui encapsulé dans un ZoneOffset. Sa représentation commence par la lettre Z puis le signe + ou - et la valeur du décalage
- offset-style : commence par un préfixe (UTC, GMT ou UT) puis le signe + ou - et la valeur du décalage
- region-based : commence par au moins deux caractères différents des préfixes de l'offset-style et des caractères + et -,

Plusieurs organismes collectent et diffusent régulièrement les changements sur les fuseaux horaires : IANA Time Zone Database (TZDB), IATA, ... Chaque organisme utilise son propre format : c'est celui du TZDB qui est prioritairement utilisé.

La classe ZoneId contient plusieurs méthodes :

Méthode	Rôle
boolean equals(Object obj)	Vérifier l'égalité avec l'objet fourni en paramètre
static ZoneId from(TemporalAccessor temporal)	Obtenir une instance à partir de l'objet temporel passé en paramètre
static Set<String> getAvailableZoneIds()	Obtenir la liste des identifiants de zones utilisables
String getDisplayName(TextStyle style, Locale locale)	Obtenir une représentation textuelle de la zone sous la forme de son nom ou d'un décalage horaire
abstract String getId()	Obtenir l'identifiant unique de la zone

abstract ZoneRules getRules()	Obtenir les règles de calculs à utiliser pour la zone
ZoneId normalized()	Normaliser l'instance en tentant de renvoyer si possible une instance type ZoneOffset
static ZoneId of(String zoneId)	Obtenir une instance à partir de son identifiant
static ZoneId of(String zoneId, Map<String, String> aliasMap)	Obtenir une instance à partir de son identifiant en utilisant la collections d'alias en remplacement des ID standard
static ZoneId ofOffset(String prefix, ZoneOffset offset)	Obtenir une instance qui encapsule la ZoneOffset passée en paramètre
static ZoneId systemDefault()	Obtenir la ZoneId du système

La classe ZoneId peut être sérialisée : dans ce cas c'est l'ID qui est stocké. Il est possible que l'ID n'existe pas dans la JVM où l'objet est dé-sérialisé. Dans ce cas, l'invocation de la méthode getRules() lève une exception de type ZoneRulesException.

ZoneId est une classe de type value-based : il ne faut pas utiliser d'opérations sur une instance de type Period requérant l'identité de l'objet : opérateur ==, hashCode ou synchronisation. La comparaison de deux instances doit se faire en utilisant la méthode equals().

112.6.7.2. La classe ZoneOffset

La classe ZoneOffset encapsule un décalage horaire.

Elle possède plusieurs méthodes :

Constante	Rôle
static ZoneOffset MAX	Le décalage maximum : +18:00
static ZoneOffset MIN	Le décalage minimum : -18:00
static ZoneOffset UTC	Le décalage pour UTC : 00:00 avec l'ID «Z»

Elle possède plusieurs méthodes :

Méthode	Rôle
Temporal adjustInto(Temporal temporal)	Renvoyer une copie de l'objet temporel fourni en paramètre ajustée avec l'instance de type ZoneOffset
int compareTo(ZoneOffset other)	Comparer l'instance avec celle fournie en paramètre
boolean equals(Object obj)	Vérifier l'égalité avec l'objet fourni en paramètre
static ZoneOffset from(TemporalAccessor temporal)	Obtenir une instance à partir de l'objet temporel fourni en paramètre
int get(TemporalField field)	Obtenir la valeur du champ passé en paramètre sous la forme d'un entier
String getId()	Obtenir l'ID de l'instance
long getLong(TemporalField field)	Obtenir la valeur du champ passé en paramètre sous la forme d'un entier
ZoneRules getRules()	Obtenir les règles de calculs encapsulés dans une instance de type ZoneRules
int getTotalSeconds()	Obtenir le nombre de secondes correspondant au décalage

boolean isSupported(TemporalField field)	Vérifier si le champ passé en paramètre est supporté
static ZoneOffset of(String offsetId)	Obtenir une instance à partir de son ID
static ZoneOffset ofHours(int hours)	Obtenir une instance à partir du nombre d'heures fourni en paramètre
static ZoneOffset ofHoursMinutes(int hours, int minutes)	Obtenir une instance à partir du nombre d'heures et de minutes fournis en paramètres
static ZoneOffset ofHoursMinutesSeconds(int hours, int minutes, int seconds)	Obtenir une instance à partir du nombre d'heures, de minutes et de secondes fournis en paramètres
static ZoneOffset ofTotalSeconds(int totalSeconds)	Obtenir une instance à partir du nombre de secondes fourni en paramètre
<R> R query(TemporalQuery<R> query)	Exécuter la requête passée en paramètre sur l'instance
ValueRange range(TemporalField field)	Obtenir la plage de valeurs utilisables pour le champ passé en paramètre

Le décalage horaire correspond à la différence de temps entre le fuseau horaire et celui de Greenwich : c'est généralement une quantité temporelle exprimée sous la forme d'heures et de minutes. Par exemple : +01:00 pour Paris en hiver et +02:00 en été.

Le fuseau horaire encapsulé dans une instance de type ZoneId possède une ou plusieurs instances de type ZoneOffset. Pour la ZoneId « Europe/Paris », il y a deux ZoneOffset : une pour l'heure d'été et une pour l'heure d'hiver.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.LocalDateTime;
import java.time.Month;
import java.time.ZoneId;
import java.time.zone.ZoneRules;

public class TestZoneOffset {
  public static void main(String[] args) {
    ZoneId zoneId = ZoneId.of("Europe/Paris");

    ZoneRules rules = zoneId.getRules();
    System.out.println(rules.getOffset(LocalDateTime.of(2015,
      Month.FEBRUARY, 10, 0, 0)));
    System.out.println(rules.getOffset(LocalDateTime.of(2015,
      Month.JUNE,10, 0, 0)));
  }
}
```

Résultat :

```
+01:00
+02:00
```

Cette classe est conçue pour être utilisée avec le calendrier ISO.

ZoneOffset est une classe de type value-based : il ne faut pas utiliser d'opérations sur une instance de type Period requérant l'identité de l'objet : opérateur ==, hashCode ou synchronisation. La comparaison de deux instances doit se faire en utilisant la méthode equals().

112.6.7.3. La classe ZonedDateTime

La classe ZonedDateTime encapsule une date, une heure et un fuseau horaire : elle peut être vue comme une combinaison d'une LocalDateTime et d'une ZoneId. Il est nécessaire d'utiliser une instance de type ZonedDateTime pour fournir une donnée temporelle qui soit indépendante de la machine sur laquelle elle est créée.

Elle permet la conversion d'une LocalDateTime en une Instant. Cette conversion requiert le calcul du décalage horaire à appliquer en fonction des règles encapsulées dans la propriété rules de l'instance de type ZoneId.

Le décalage peut être simple en correspondant directement au décalage entre le fuseau horaire et celui de Greenwich. Ce décalage peut aussi être dans certains cas plus complexe notamment lors du passage :

- à l'heure d'été au printemps : dans ce cas, le décalage est incrémenté
- à l'heure d'hiver en automne : dans ce cas, le décalage est décrémenté

Elle possède de nombreuses méthodes :

Méthode	Rôle
String format(DateTimeFormatter formatter)	Formater l'instance avec le Formatter passé en paramètre
static ZonedDateTime from(TemporalAccessor temporal)	Obtenir une instance à partir de l'objet temporel passé en paramètre
int get(TemporalField field)	Obtenir la valeur du champ passé en paramètre sous la forme d'un entier
int getDayOfMonth() DayOfWeek getDayOfWeek()	Obtenir le champ jour du mois
int getDayOfYear()	Obtenir le champ jour de l'année
int getHour()	Obtenir le champ heure
long getLong(TemporalField field)	Obtenir la valeur du champ passé en paramètre sous la forme d'un entier long
int getMinute()	Obtenir les minutes
Month getMonth()	Obtenir le mois de l'année
int getMonthValue()	Obtenir le mois de l'année (1 à 12)
int getNano()	Obtenir les nanosecondes
ZoneOffset getOffset()	Obtenir le décalage horaire. Exemple : «+01:00»
int getSecond()	Obtenir les secondes
int getYear()	Obtenir l'année
ZoneId getZone()	Obtenir le fuseau horaire. Exemple « Europe/Paris»
boolean isSupported(TemporalField field)	Vérifier si le champ passé en paramètre est supporté
boolean isSupported(TemporalUnit unit)	Vérifier si l'unité passée en paramètre est supportée
ZonedDateTime minus(long amountToSubtract, TemporalUnit unit) ZonedDateTime minus(TemporalAmount amountToSubtract)	Renvoyer une copie de l'instance pour laquelle la quantité temporelle a été soustraite
ZonedDateTime minusDays(long days)	Renvoyer une copie de l'instance pour laquelle le nombre de jours a été soustrait
ZonedDateTime minusHours(long hours)	Renvoyer une copie de l'instance pour laquelle le nombre d'heures a été soustrait
ZonedDateTime minusMinutes(long minutes)	Renvoyer une copie de l'instance pour laquelle le nombre de minutes a été soustrait
ZonedDateTime minusMonths(long months)	Renvoyer une copie de l'instance pour laquelle le nombre de mois a été soustrait
ZonedDateTime minusNanos(long nanos)	

	Renvoyer une copie de l'instance pour laquelle le nombre de nanosecondes a été soustrait
ZonedDateTime minusSeconds(long seconds)	Renvoyer une copie de l'instance pour laquelle le nombre de secondes a été soustrait
ZonedDateTime minusWeeks(long weeks)	Renvoyer une copie de l'instance pour laquelle le nombre de semaines a été soustrait
ZonedDateTime minusYears(long years)	Renvoyer une copie de l'instance pour laquelle le nombre d'années a été soustrait
static ZonedDateTime now()	Obtenir une instance à partir de l'heure système et du fuseau horaire par défaut
static ZonedDateTime now(Clock clock)	Obtenir une instance à partir de l'heure obtenue de l'objet passé en paramètre
static ZonedDateTime now(ZoneId zone)	Obtenir une instance à partir de l'heure système et fuseau horaire passé en paramètre
static ZonedDateTime of(int year, int month, int dayOfMonth, int hour, int minute, int second, int nanoOfSecond, ZoneId zone)	Obtenir une instance à partir de différents champs et du fuseau horaire passés en paramètres
static ZonedDateTime of(LocalDate date, LocalTime time, ZoneId zone) static ZonedDateTime of(LocalDateTime localDateTime, ZoneId zone) static ZonedDateTime ofInstant(Instant instant, ZoneId zone)	Obtenir une instance à partir de l'objet temporel et du fuseau horaire passés en paramètres
static ZonedDateTime ofInstant(LocalDateTime localDateTime, ZoneOffset offset, ZoneId zone)	Obtenir une instance à partir de l'objet temporel, du décalage horaire et du fuseau horaire fournis en paramètres
static ZonedDateTime ofLocal(LocalDateTime localDateTime, ZoneId zone, ZoneOffset preferredOffset)	Obtenir une instance à partir de l'objet temporel et du fuseau horaire fournis en paramètres en utilisant le décalage horaire précisé si possible
static ZonedDateTime ofStrict(LocalDateTime localDateTime, ZoneOffset offset, ZoneId zone)	Obtenir une instance respectant strictement la combinaison de paramètres fournie
static ZonedDateTime parse(CharSequence text)	Analyser le texte fourni en paramètre avec le Formatter par défaut pour obtenir une instance. Exemple «2014-12-25T23:59:59+01:00[Europe/Paris]»
static ZonedDateTime parse(CharSequence text, DateTimeFormatter formatter)	Analyser le texte avec le Formatter fournis en paramètre pour obtenir une instance
ZonedDateTime plus(long amountToAdd, TemporalUnit unit) ZonedDateTime plus(TemporalAmount amountToAdd)	Obtenir une copie à laquelle la quantité temporelle a été ajoutée
ZonedDateTime plusDays(long days)	Obtenir une copie à laquelle le nombre de jours a été ajouté
ZonedDateTime plusHours(long hours)	Obtenir une copie à laquelle le nombre d'heures a été ajouté
ZonedDateTime plusMinutes(long minutes)	Obtenir une copie à laquelle le nombre de minutes a été ajouté
ZonedDateTime plusMonths(long months)	Obtenir une copie à laquelle le nombre de mois a été ajouté
ZonedDateTime plusNanos(long nanos)	Obtenir une copie à laquelle le nombre de nanosecondes a été ajouté
ZonedDateTime plusSeconds(long seconds)	Obtenir une copie à laquelle le nombre de secondes a été ajouté
ZonedDateTime plusWeeks(long weeks)	Obtenir une copie à laquelle le nombre de semaines a été ajouté
ZonedDateTime plusYears(long years)	Obtenir une copie à laquelle le nombre d'années a été ajouté

<R> R query(TemporalQuery<R> query)	Renvoyer le résultat de l'exécution de la requête fournie en paramètre sur l'instance
ValueRange range(TemporalField field)	Obtenir la plage de valeurs valides pour le champ fourni en paramètre
LocalDate toLocalDate()	Obtenir une instance de type LocalDate à partir des champs encapsulés
LocalDateTime toLocalDateTime()	Obtenir une instance de type LocalDateTime à partir des champs encapsulés
LocalTime toLocalTime()	Obtenir une instance de type LocalTime à partir des champs encapsulés
OffsetDateTime toOffsetDateTime()	Obtenir une instance de type OffsetDateTime à partir des champs encapsulés
ZonedDateTime truncatedTo(TemporalUnit unit)	Obtenir une copie de l'instance tronquée à l'unité fournie en paramètre
long until(Temporal endExclusive, TemporalUnit unit)	Calculer la quantité de temps exprimée dans l'unité fournie entre l'instance et l'objet temporel fourni en paramètres
ZonedDateTime with(TemporalAdjuster adjuster)	Renvoyer une copie de l'instance ajustée grâce à l'objet fourni en paramètre
ZonedDateTime with(TemporalField field, long newValue)	Renvoyer une copie de l'instance dans la valeur du champ fournie en paramètre est remplacée
ZonedDateTime withDayOfMonth(int dayOfMonth)	Renvoyer une copie de l'instance dont la valeur du champ jour du mois est remplacée avec celle fournie
ZonedDateTime withDayOfYear(int dayOfYear)	Renvoyer une copie de l'instance dont la valeur du champ jour de l'année est remplacée avec celle fournie
ZonedDateTime withEarlierOffsetAtOverlap()	
ZonedDateTime withFixedOffsetZone()	Retourne une copie de l'instance pour laquelle zone ID a la valeur de l'offset
ZonedDateTime withHour(int hour)	Renvoyer une copie de l'instance dont la valeur du champ heure est remplacée avec celle fournie
ZonedDateTime withLaterOffsetAtOverlap()	
ZonedDateTime withMinute(int minute)	Renvoyer une copie de l'instance dont la valeur du champ minute est remplacée avec celle fournie
ZonedDateTime withMonth(int month)	Renvoyer une copie de l'instance dont la valeur du champ mois est remplacée avec celle fournie
ZonedDateTime withNano(int nanoOfSecond)	Renvoyer une copie de l'instance dont la valeur du champ nanosecondes est remplacée avec celle fournie
ZonedDateTime withSecond(int second)	Renvoyer une copie de l'instance dont la valeur du champ seconde est remplacée avec celle fournie
ZonedDateTime withYear(int year)	Renvoyer une copie de l'instance dont la valeur du champ années est remplacée avec celle fournie
ZonedDateTime withZoneSameInstant(ZoneId zone)	Renvoyer une copie de l'instance représentant le même instant mais dans le fuseau horaire fourni en paramètre
ZonedDateTime withZoneSameLocal(ZoneId zone)	Renvoyer une copie de l'instance utilisant le fuseau horaire fourni en paramètre en tentant de conserver la même date/heure locale

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;
```

```
import java.time.LocalDateTime;
import java.time.Month;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.time.zone.ZoneRules;
import java.util.Set;

public class TestZonedDateTime {
  public static void main(String[] args) {
    LocalDateTime dateTime = LocalDateTime.of(2014,12,25,0,0,0);
    ZoneId zoneId = ZoneId.of("Europe/Paris");
    ZonedDateTime zoneDateTime = ZonedDateTime.of(dateTime, zoneId);
    System.out.println("zoneDateTime="+zoneDateTime);
    System.out.println(zoneId.getRules().isDaylightSavings(zoneDateTime.toInstant()));

    zoneId = ZoneId.of("America/Los_Angeles");
    zoneDateTime = ZonedDateTime.of(dateTime, zoneId);
    System.out.println("zoneDateTime L.A. ="+zoneDateTime);
    System.out.println("zoneDateTime Paris="
      +zoneDateTime.withZoneSameInstant(ZoneId.of("Europe/Paris")));
    // Calcul heure locale d'arrivee apres un vol de 12h30
    LocalDateTime dateTimeDepart = LocalDateTime.of(2015,3,11,8,0,0);
    System.out.println("depart Paris "+dateTimeDepart+" heure locale");
    ZoneId zoneIdDepart = ZoneId.of("Europe/Paris");
    ZonedDateTime zoneDateTimeDepart = ZonedDateTime.of(dateTimeDepart, zoneIdDepart);
    ZoneId zoneIdArrivee = ZoneId.of("America/Los_Angeles");
    ZonedDateTime zoneDateT imeArrivee = zoneDateTimeDepart
      .withZoneSameInstant(zoneIdArrivee).plusHours(12).plusMinutes(30);
    LocalDateTime dateTimeArrivee = zoneDateTimeArrivee.toLocalDateTime();
    System.out.println("Arrivee L.A. "+dateTimeArrivee+" heure locale");
  }
}
```

Résultat :

```
zoneDateTime=2014-12-25T00:00+01:00[Europe/Paris]
false
zoneDateTime L.A. =2014-12-25T00:00-08:00[America/Los_Angeles]
zoneDateTime Paris=2014-12-25T09:00+01:00[Europe/Paris]
depart Paris 2015-03-11T08:00
heure locale
Arrivee L.A. 2015-03-11T12:30 heure locale
```

ZonedDateTime est une classe de type value-based : il ne faut pas utiliser d'opérations sur une instance de type ZonedDateTime requérant l'identité de l'objet : opérateur ==, hashCode ou synchronisation. La comparaison de deux instances doit se faire en utilisant la méthode equals().

112.6.7.4. La classe OffsetDateTime

La classe OffsetDateTime encapsule de manière immuable une date, une heure et un décalage horaire à partir de l'heure UTC (méridien de Greenwich) : elle combine une LocalDateTime et une ZoneOffset telles que 2014-12-25T00:00:00+01:00.

Cette classes est utile pour sérialiser une donnée temporelle en dehors de la JVM tel que le stockage dans une base de données, l'échange entre plusieurs serveurs, ...

Les classes ZonedDateTime et OffsetDateTime sont très proches. La différence majeure est que la classe OffsetDateTime applique simplement le décalage : elle ne tient, par exemple, pas compte de l'heure d'été/hiver.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.LocalDateTime;
import java.time.Month;
import java.time.OffsetDateTime;
import java.time.ZoneOffset;

public class TestOffsetDateTime {
  public static void main(String[] args) {
    LocalDateTime localDateTime = LocalDateTime.of(2014,Month.DECEMBER, 25, 4,00,0);
    final ZoneOffset zoneOffset = ZoneOffset.of("+05:00");
    OffsetDateTime offsetDateTime = OffsetDateTime.of(localDateTime, zoneOffset);
    System.out.println(offsetDateTime);

    offsetDateTime = OffsetDateTime.parse("2014-12-25T04:00+05:00");
    offsetDateTime =  offsetDateTime.withOffsetSameInstant(ZoneOffset.of("+01:00"));
    System.out.println(offsetDateTime);
    System.out.println(offsetDateTime.toLocalDateTime());
  }
}
```

Résultat :

```
2014-12-25T04:00+05:00
2014-12-25T00:00+01:00
2014-12-25T00:00
```

OffsetDateTime est une classe de type value-based : il ne faut pas utiliser d'opérations sur une instance de type OffsetDateTime requérant l'identité de l'objet : opérateur ==, hashCode ou synchronisation. La comparaison de deux instances doit se faire en utilisant la méthode equals().

112.6.7.5. La classe OffsetTime

La classe OffsetTime encapsule une heure et un décalage horaire par rapport à l'heure UTC dans le calendrier ISO : elle combine une LocalTime et une ZoneOffset.

La précision de l'heure encapsulée est la nanoseconde. Elle est immuable et thread-safe.

Elle possède de nombreuses méthodes :

Méthode	Rôle
Temporal adjustInto(Temporal temporal)	Renvoyer une copie de l'objet temporel fourni en paramètre dans lequel la valeur du champ encapsulé est remplacée par celle passée en paramètre
OffsetDateTime atDate(LocalDate date)	Combiner l'instance courante avec l'objet temporal fourni en paramètre pour obtenir une instance de type OffsetDateTime
int compareTo(OffsetTime other)	Comparer l'instance courante avec celle fournie en paramètre
String format(DateTimeFormatter formatter)	Formater l'instance avec le formateur passé en paramètre
static OffsetTime from(TemporalAccessor temporal)	Obtenir une instance à partir de l'objet temporel passé en paramètre
int get(TemporalField field)	Obtenir la valeur du champ passé en paramètre sous la forme d'un entier
int getHour()	Obtenir le champ heure
long getLong(TemporalField field)	Obtenir la valeur du champ passé en paramètre sous la forme d'un entier long
int getMinute()	Obtenir les minutes
int getNano()	Obtenir les nanosecondes
ZoneOffset getOffset()	Obtenir le décalage horaire. Exemple : «+01:00»
int getSecond()	Obtenir les secondes

boolean isAfter(OffsetTime other)	Vérifier si l'instance courante est postérieure à celle fournie en paramètre
boolean isBefore(OffsetTime other)	Vérifier si l'instance courante est antéreure à celle fournie en paramètre
boolean isEqual(OffsetTime other)	Vérifier si l'instance est égale à celle fournie en paramètre
boolean isSupported(TemporalField field)	Vérifier si le champ passé en paramètre est supporté
boolean isSupported(TemporalUnit unit)	Vérifier si l'unité passée en paramètre est supportée
OffsetTime minus(long amountToSubtract, TemporalUnit unit) OffsetTime minus(TemporalAmount amountToSubtract)	Renvoyer une copie de l'instance à laquelle la quantité temporelle a été soustraite
OffsetTime minusHours(long hours)	Renvoyer une copie de l'instance pour laquelle le nombre d'heures a été soustrait
OffsetTime minusMinutes(long minutes)	Renvoyer une copie de l'instance pour laquelle le nombre de minutes a été soustrait
OffsetTime minusNanos(long nanos)	Renvoyer une copie de l'instance pour laquelle le nombre de nanosecondes a été soustrait
OffsetTime minusSeconds(long seconds)	Renvoyer une copie de l'instance pour laquelle le nombre de secondes a été soustrait
static OffsetTime now()	Obtenir une instance à partir de l'heure système et fuseau horaire par défaut
static OffsetTime now(Clock clock)	Obtenir une instance à partir de l'heure obtenue de l'objet passé en paramètre
static OffsetTime now(ZoneId zone)	Obtenir une instance à partir de l'heure système et du fuseau horaire passés en paramètre
static OffsetTime of(int hour, int minute, int second, int nanoOfSecond, ZoneOffset offset)	Obtenir une instance à partir de différents champs et du fuseau horaire passés en paramètres
static OffsetTime of(LocalTime time, ZoneOffset offset)	Obtenir une instance à partir de l'objet temporel et du décalage horaire fournis en paramètres
static OffsetTime ofInstant(Instant instant, ZoneId zone)	Obtenir une instance à partir de l'objet temporel, et du fuseau horaire fournis en paramètres
static OffsetTime parse(CharSequence text)	Analyser le texte fourni en paramètre avec le Formatter par défaut pour obtenir une instance. Exemple «23:59:59+01:00»
static OffsetTime parse(CharSequence text, DateTimeFormatter formatter)	Analyser le texte avec le Formatter fourni en paramètre pour obtenir une instance
OffsetTime plus(long amountToAdd, TemporalUnit unit) OffsetTime plus(TemporalAmount amountToAdd)	Obtenir une copie à laquelle la quantité temporelle a été ajoutée
OffsetTime plusHours(long hours)	Obtenir une copie à laquelle le nombre d'heures a été ajouté
OffsetTime plusMinutes(long minutes)	Obtenir une copie à laquelle le nombre de minutes a été ajouté
OffsetTime plusNanos(long nanos)	Obtenir une copie à laquelle le nombre de nanosecondes a été ajouté
OffsetTime plusSeconds(long seconds)	Obtenir une copie à laquelle le nombre de secondes a été ajouté
<R> R query(TemporalQuery<R> query)	Renvoyer le résultat de l'exécution de la requête fournie en paramètre sur l'instance

ValueRange range(TemporalField field)	Obtenir la plage de valeurs valides pour le champ fourni en paramètre
LocalTime toLocalTime()	Obtenir une instance de type LocalTime à partir des champs encapsulés
OffsetTime truncatedTo(TemporalUnit unit)	Obtenir une copie de l'instance tronquée à l'unité fournie en paramètre
long until(Temporal endExclusive, TemporalUnit unit)	Calculer la quantité de temps exprimée dans l'unité fournie entre l'instance et l'objet temporel fournis en paramètres
OffsetTime with(TemporalAdjuster adjuster)	Renvoyer une copie de l'instance ajustée grâce à l'objet fourni en paramètre
OffsetTime with(TemporalField field, long newValue)	Renvoyer une copie de l'instance dont la valeur du champ fournie en paramètre est remplacée
OffsetTime withHour(int hour)	Renvoyer une copie de l'instance dont la valeur du champ heure est remplacée par celle fournie
OffsetTime withMinute(int minute)	Renvoyer une copie de l'instance dont la valeur du champ minute est remplacée par celle fournie
OffsetTime withNano(int nanoOfSecond)	Renvoyer une copie de l'instance dont la valeur du champ nanosecondes est remplacée par celle fournie
OffsetTime withOffsetSameInstant(ZoneOffset offset)	Renvoyer une copie de l'instance représentant le même instant mais dans le décalage horaire fourni en paramètre
OffsetTime withOffsetSameLocal(ZoneOffset offset)	Renvoyer une copie de l'instance représentant la même heure locale avec le décalage horaire fourni en paramètre
OffsetTime withSecond(int second)	Renvoyer une copie de l'instance dont la valeur du champ seconde est remplacée par celle fournie

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.OffsetTime;
import java.time.ZoneOffset;

public class TestOffsetTime {
  public static void main(String[] args) {
    OffsetTime time = OffsetTime.now();
    System.out.println(time);
    // changement du décalage en conservant le même point dans le temps
    ZoneOffset offset = ZoneOffset.ofHours(5);
    OffsetTime sameTimeDifferentOffset = time.withOffsetSameInstant(
        offset);
    System.out.println(sameTimeDifferentOffset);
    // changement du décalage en conservant la date/heure locale
    OffsetTime changeTimeWithNewOffset = time.withOffsetSameLocal(
        offset);
    System.out.println(changeTimeWithNewOffset);
  }
}
```

Résultat :

```
13:38:53.187+01:00
17:38:53.187+05:00
13:38:53.187+05:00
```

OffsetTime est une classe de type value-based : il ne faut pas utiliser d'opérations sur une instance de type OffsetTime requérant l'identité de l'objet : opérateur ==, hashCode ou synchronisation. La comparaison de deux instances doit se faire en utilisant la méthode equals().

112.6.8. L'analyse et le formatage

Les classes qui encapsulent une donnée temporelle proposent :

- la méthode parse() qui permet d'obtenir une instance à partir de l'analyse d'une représentation textuelle de la donnée
- la méthode format() pour obtenir une représentation textuelle

Ces opérations sont fréquemment utilisées pour sérialiser/désérialiser des données temporelles.

112.6.8.1. La classe DateTimeFormatter

L'analyse d'une chaîne de caractères pour obtenir un objet temporel ou le formatage d'un objet temporel pour obtenir une représentation textuelle se font grâce à des objets de type DateTimeFormatter.

Ces opérations se font sur la base d'un motif exprimé sous la forme d'une chaîne de caractères qui permet de préciser le format à utiliser.

La classe propose des constantes de type DateTimeFormatter qui encapsulent des motifs standard pour faciliter leurs utilisations.

Nom	Rôle	Exemple
BASIC_ISO_DATE	Format ISO de base pour date sans décalage horaire	"20141225"
ISO_DATE	Format ISO de date avec ou sans décalage horaire	"2014-12-25", "2014-12-25+01:00"
ISO_DATE_TIME	Format ISO de date-heure avec ou sans fuseau horaire et décalage horaire	"2014-12-25T00:00:00", "2014-12-25T10:00:00+01:00", "2014-12-25T00:00:00+01:00[Europe/Paris]"
ISO_INSTANT	Format ISO de date-heure en UTC	"2014-12-25T00:00:00Z"
ISO_LOCAL_DATE	Format ISO de date sans fuseau horaire ni décalage horaire	"2014-12-25"
ISO_LOCAL_DATE_TIME	Format ISO de date-heure sans fuseau horaire ni décalage horaire	"2014-12-25T00:00:00"
ISO_LOCAL_TIME	Format ISO d'heure sans fuseau horaire ni décalage horaire	"20:15", "20:15:30"
ISO_OFFSET_DATE	Format ISO de date avec décalage horaire	"2014-12-25+01:00".
ISO_OFFSET_DATE_TIME	Format ISO de date-heure avec décalage horaire	"2014-12-25T00:00:00+01:00"
ISO_OFFSET_TIME	Format ISO d'heure avec décalage horaire	"10:15+01:00", "10:15:30+01:00"
ISO_ORDINAL_DATE	Format ISO de date exprimée en jour d'une année sans décalage horaire	"2014-359", "2014-359+01:00"

ISO_TIME	Format ISO d'heure avec ou sans décalage horaire	"10:15", "10:15:30", "10:15:30+01:00"
ISO_WEEK_DATE	Format ISO de date exprimée en semaine d'une année sans décalage horaire	"2014-W52-4", "2014-W52-4+01:00"
ISO_ZONED_DATE_TIME	Format ISO avec fuseau horaire et décalage horaire	"2014-12-25T00:00:00+01:00[Europe/Paris]"
RFC_1123_DATE_TIME	Format RFC-1123 / RFC 822	"Thu, 25 Dec 2014 00:00:00 +0100"

La classe DateTimeFormatter permet d'exprimer son propre motif pour définir le format de la date-heure :

Nom	Rôle	Type de format	Exemple
G	Ere	Texte	AD
u	Année	Année (Numérique)	2014,14
y	Année de l'ère	Année (Numérique)	2014,14
D	Jour de l'année	Numérique	352
M	Mois de l'année	Numérique	7,07
L	Mois de l'année	Texte	J, Jul, July
d	Jour du mois	Numérique	12
Q	Trimestre de l'année	Numérique	3,03
q	Trimestre de l'année	Texte	Q3, 3rd quarter
Y	Année	Année (Numérique)	
w	Semaine de l'année	Numérique	35
W	Semaine du mois	Numérique	4
E	Jour de la semaine	Texte	J, Jeu, Jeudi
e	Jour de la semaine selon la Locale	Numérique	2; 02
c	Jour de la semaine selon la Locale	Texte	
F	Semaine du mois	Numérique	3
a	Matin ou après midi	Texte	AM, PM
h	Heure de la demi-journée	Numérique (1-12)	1 10
K	Heure de la demi-journée	Numérique (0-11)	0 10
k	Heure de la journée	Numérique (1-24)	1 18
H	Heure de la journée	Numérique (0-23)	0 18
m	Minute de l'heure	Numérique	45
s	Seconde de l'heure	Numérique	59
S	Fraction de la seconde	Fraction (Numérique)	875
A	Milliseconde de la journée	Numérique	
n	Nanoseconde de la seconde	Numérique	
N	Nanoseconde de la journée	Numérique	
V	Identifiant du fuseau horaire	Zone-id (Texte)	America/Los_Angeles

z	Nom du fuseau horaire	Nom-Zone (Texte)	Pacific Standard Time; PST
O	Décalage horaire	Offset-O (Texte)	GMT+8; GMT+08:00; UTC-08:00;
X	Décalage horaire	Offset-X (Texte)	Z; -08; -0830; -08:30; -083015; -08:30:15;
x	Décalage horaire	Offset-x (Texte)	+0000; -08; -0830; -08:30; -083015; -08:30:15;
Z	Décalage horaire	Offset-Z (Texte)	+0000; -0800; -08:00;
p	Padding		
'	Délimiteur d'une chaîne de caractères	Texte	
''	Simple quote (doublée)	Texte	
[	Début d'une partie optionnelle		
]	Fin d'une partie optionnelle		
#	Réservé		
{	Réservé		
}	Réservé		

Toutes les lettres minuscules et majuscules sont réservées pour exprimer des données dans le motif.

Le nombre d'occurrences d'une lettre permet de préciser le format utilisé selon le type de format :

Type de format	Nb occurrences	Format
Texte	Inférieure à 4	Short
	4	Full
	5	Narrow
Numérique	1	Minimum de chiffres sans padding Obligatoire pour c (jour de la semaine) et F (semaine du mois)
	Supérieur à 1	Le nombre d'occurrences précise la taille complétée avec des zéros au besoin Maximum 2 pour d, H, h, K, k, m et s Maximun 3 pour D
Numérique/texte	Supérieur ou égal à 3	Format texte
	Inférieur à 3	Format numérique
Fraction	9	Affichage intégral
	Inférieur à 9	Tronquée au nombre d'occurrences
Year	2	Deux chiffres complétés au besoin par un zéro
	Supérieur à 2	Format intégral
Zone-Id	2	Format intégral
	Différent de 2	IllegalArgumentException
Nom-zone	1, 2, 3	Nom court
	4	Nom long
	Supérieur à 4	IllegalArgumentException
	1	

Offset-X et Offset-x (x affiche des zéros, X affiche Z pour un zéro)		Uniquement les heures et éventuellement les minutes si elles ne sont pas égales à zéro. Exemple +01
	2	Heures et minutes sans séparateur. Exemple +0145
	3	Heures et minutes avec un caractère deux points comme séparateur. Exemple +01:45
	4	Heures, minutes et secondes sans séparateur. Exemple +014530
	5	Heures, minutes et secondes avec un caractère deux points comme séparateur. Exemple +01:45:30
	Supérieur à 5	IllegalArgumentException
Offset-O	1	Format court avec les heures non précédées par un zéro. Les minutes et les secondes sont optionnelles. Exemple GMT+1
	4	Format long avec les heures et les minutes précédées par un zéro au besoin. Les secondes sont optionnelles. Elles sont séparées par un caractère deux points. Exemple GMT+01:00
	Autres valeurs	IllegalArgumentException
Offset-Z	1, 2, 3	Uniquement les heures et les minutes. Exemple +0100, +0000
	4	Format long avec les heures et les minutes précédées par un zéro au besoin. Les secondes sont optionnelles. Elles sont séparées par un caractère deux points. Exemple GMT+01:00, GMT+00:00
	5	Idem mais Z si l'offset est zéro
	Supérieur à 5	IllegalArgumentException

La lettre p permet de modifier le mode de padding par espaces de l'élément suivant. Le nombre d'occurrences de la lettre p permet de préciser la taille de l'élément.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.Month;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.time.format.DateTimeFormatter;

public class TestDateTimeFormatter {
  public static void main(String[] args) {
    ZonedDateTime date = ZonedDateTime.of(LocalDateTime.of(2014,
      Month.DECEMBER, 25, 0,0,0), ZoneId.of("GMT"));
    DateTimeFormatter formatter =
      DateTimeFormatter.ofPattern("dd ppppppppppMMM yyyy xxx");
    String dateTimeStr = date.format(formatter);
    System.out.println(dateTimeStr);
  }
}
```

Résultat :

```
25       déc. 2014 +00:00
```

Toutes les lettres non reconnues lèvent une exception. Tous les caractères différents de '[', ']', '{', '}' et '#' sont utilisés tels quels. Il est cependant recommandé d'entourés ces caractères par des quotes simples pour garantir la compatibilité avec de futures versions.

En plus du motif du format à utiliser, il est possible de préciser plusieurs éléments à utiliser dans les traitements :

- la Locale qui peut avoir des impacts sur le formatage et l'analyse
- le calendrier qui sera utilisé pour convertir l'objet temporel avant son formatage et après l'analyse
- le fuseau horaire qui sera utilisé pour convertir l'objet temporel avant son formatage et après l'analyse
- le DecimalStyle

La classe DateTimeFormatter possède de nombreuses méthodes :

Méthode	Rôle
String format(TemporalAccessor temporal)	Formater l'objet temporel fourni en paramètre
void formatTo(TemporalAccessor temporal, Appendable appendable)	Formater l'objet temporel fourni en paramètre et l'ajouter à l'Appendable
Chronology getChronology()	Obtenir le calendrier utilisé lors des traitements
DecimalStyle getDecimalStyle()	Obtenir l'objet de type DecimalStyle utilisé lors des traitements
Locale getLocale()	Obtenir la Locale utilisée lors des traitements
Set<TemporalField> getResolverFields()	Permet de définir les champs qui seront utilisés pour définir l'objet temporel lors de la phase d'analyse. Par défaut, une instance de type DateTimeFormatter ne possède aucun resolver
ResolverStyle getResolverStyle()	Obtenir le style de résolution utilisé par l'instance. Par défaut, une instance de type DataTimeFormatter possède le style SMART
ZoneId getZone()	Obtenir le fuseau horaire utilisé lors des traitements
static DateTimeFormatter ofLocalizedDate(FormatStyle dateStyle)	Obtenir une instance pour le calendrier ISO tenant compte de la Locale pour des dates
static DateTimeFormatter ofLocalizedDateTime(FormatStyle dateTimeStyle) static DateTimeFormatter ofLocalizedDateTime(FormatStyle dateStyle, FormatStyle timeStyle)	Obtenir une instance pour le calendrier ISO tenant compte de la Locale pour des dates heures
static DateTimeFormatter ofLocalizedTime(FormatStyle timeStyle)	Obtenir une instance pour le calendrier ISO tenant compte de la Locale pour des heures
static DateTimeFormatter ofPattern(String pattern)	Obtenir une instance qui utilise le motif passé en paramètre
static DateTimeFormatter ofPattern(String pattern, Locale locale)	Obtenir une instance qui utilise le motif et la Locale passés en paramètres
TemporalAccessor parse(CharSequence text)	Analyser la chaîne de caractères fournie en paramètre pour obtenir une instance d'un objet temporel
TemporalAccessor parse(CharSequence text, ParsePosition position)	Analyser le texte fourni à partir de la position précisée
<T> T parse(CharSequence text, TemporalQuery<T> query)	Analyser le texte pour obtenir une instance du type précisé par la requête
TemporalAccessor parseBest(CharSequence text, TemporalQuery<?>... queries)	Analyser au mieux le texte pour renvoyer une instance d'un des types précisés par les requêtes fournies
static TemporalQuery<Period> parsedExcessDays()	Retourner une requête qui permet d'obtenir une instance de type Period contenant un nombre de jours supplémentaires résultant de l'analyse. Cette période peut par exemple valoir un jour selon le ResolutionStyle et la valeur 24:00 pour l'heure
static TemporalQuery<Boolean> parsedLeapSecond()	Retourner une requête qui permet de déterminer si l'analyse à trouver une leap-second (la valeur de la seconde est 60)
TemporalAccessor parseUnresolved(CharSequence text, ParsePosition position)	Effectuer l'analyse pour extraire les champs sans les résoudre

Format toFormat()	Assurer une certaine compatibilité en retournant une implémentation de type java.text.Format de l'instance
Format toFormat(TemporalQuery<?> parseQuery)	
DateTimeFormatter withChronology(Chronology chrono)	Renvoyer une copie de l'instance qui utilise le calendrier fourni en paramètre
DateTimeFormatter withDecimalStyle(DecimalStyle decimalStyle)	Renvoyer une copie de l'instance qui utilise le DecimalStyle fourni en paramètre
DateTimeFormatter withLocale(Locale locale)	Renvoyer une copie de l'instance qui utilise la Locale fournie en paramètre
DateTimeFormatter withResolverFields(Set<TemporalField> resolverFields) DateTimeFormatter withResolverFields(TemporalField... resolverFields)	Renvoyer une copie de l'instance dans laquelle les champs qui seront utilisés pour définir l'objet temporel lors de la phase d'analyse sont précisés
DateTimeFormatter withResolverStyle(ResolverStyle resolverStyle)	Renvoyer une copie de l'instance dans laquelle le style de résolution est modifié avec celui fourni en paramètre (LENIENT, SMART, STRICT)
DateTimeFormatter withZone(ZoneId zone)	Renvoyer une copie de l'instance qui utilise le fuseau horaire fourni en paramètre

L'analyse d'une chaîne de caractères par rapport au motif se fait en deux phases :

- l'analyse du texte en utilisant le motif crée une collection de type Map qui contient pour chaque champ, la valeur ainsi qu'un calendrier et un fuseau horaire
- le contenu de la Map est validé et combiné dans une étape nommée résolution

La méthode parseUnresolved() ne réalise que la première étape, ce qui réserve son utilisation à des besoins très spécifiques de bas niveau.

L'étape de résolution peut être configurée grâce à deux propriétés :

- ResolverStyle qui est une énumération du mode de résolution : strict, smart (par défaut) et lenient
- resolverFields qui permet de préciser quels champs doivent être utilisés pour la résolution

La résolution est un processus complexe. Il est possible que l'analyse extrait plusieurs champs : année, mois, jour du mois et jour de l'année. Pour composer la date, il est possible d'utiliser :

- soit année, mois, jour du mois
- soit année et jour de l'année

La définition de la propriété resolverFields permet de définir les champs à utiliser. Si la propriété n'est pas définie dans le cas ci-dessus, alors les deux résolutions sont effectuées et doivent renvoyer le même résultat.

Les méthodes d'analyse et de formatage lève une exception en cas de problème durant leur exécution :

- DateTimeParseException durant une analyse
- DateTimeException durant un formatage

Plusieurs méthodes de l'API permettent de faciliter l'utilisation d'un DateTimeFormatter.

La méthode format() d'un objet de type ChronoLocalDate permet d'obtenir une représentation de l'instance sous une forme textuelle en utilisant le DateTimeFormatter fourni en paramètre.

La méthode parse() de la classe LocalDate qui attend en paramètre la chaîne de caractères à analyser utilise le Formatter ISO_LOCAL_DATE. Une surcharge de la méthode parse() attend un second paramètre de type DateTimeFormatter qui permet de préciser le formateur à utiliser.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.LocalDateTime;
import java.time.Month;
import java.time.format.DateTimeFormatter;

public class TestDateTimeFormatter {
  public static void main(String[] args) {

    String dateStr = "";
    LocalDateTime dateTime = LocalDateTime.of(2014, Month.DECEMBER, 25, 0,0,0);
    System.out.println(dateTime.format(DateTimeFormatter.BASIC_ISO_DATE));
  }
}
```

Résultat :

```
20141225
```

Il est aussi possible de définir son propre format.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.Month;
import java.time.format.DateTimeFormatter;

public class TestDateTimeFormatter {
  public static void main(String[] args) {
    LocalDateTime dateTime = LocalDateTime.of(2014, Month.DECEMBER, 25, 0,0,0);
    DateTimeFormatter formatter =
      DateTimeFormatter.ofPattern("dd MMM yyyy");
    String dateTimeStr = dateTime.format(formatter);
    System.out.println(dateTimeStr);

    LocalDate date = LocalDate.parse(dateTimeStr, formatter);
    System.out.println(date);
  }
}
```

Résultat :

```
25 déc. 2014
2014-12-25
```

La méthode parseBest() permet d'obtenir un objet temporel dont les types possibles sont fournis en paramètre sous la forme de requêtes. Ceci est pratique lorsque le motif possède des parties optionnelles : selon le texte fourni, plusieurs types d'objets temporels peuvent être obtenus. Cette méthode tente de renvoyer la plus complète possible en fonction du texte.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.LocalDateTime;
import java.time.Month;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.time.format.DateTimeFormatter;
import java.time.temporal.TemporalAccessor;

public class TestDateTimeFormatter {
  public static void main(String[] args) {
    parseBest("2014-12-25 00:00");
    parseBest("2014-12-25 00:00[Europe/Paris]");
  }

  private static void parseBest(String str) {
    System.out.println("Texte = " + str);
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("uuuu-MM-dd HH:mm['['VV']']");
    TemporalAccessor dt = formatter.parseBest(str, ZonedDateTime::from, LocalDateTime::from);
    if (dt instanceof ZonedDateTime) {
      System.out.println("ZonedDateTime obtenu = " + dt);
    } else {
      System.out.println("LocalDateTime obtenu = " + dt);
    }
  }
}
```

Résultat :

```
Texte = 2014-12-25 00:00
LocalDateTime obtenu = 2014-12-25T00:00
Texte = 2014-12-25 00:00[Europe/Paris]
ZonedDateTime obtenu =
2014-12-25T00:00+01:00[Europe/Paris]
```

La classe DateTimeFormatter est immuable et thread-safe : elle peut donc sans soucis être définie comme variable static.

112.6.8.2. La classe DateTimeFormatterBuilder

La classe DateTimeFormatterBuilder permet de faciliter la composition dynamique d'un motif complexe encapsulé dans une instance de type DateTimeFormatter.

Elle possède de nombreuses méthodes à invoquer successivement pour composer le motif. La méthode toFormatter() permet d'obtenir l'instance une fois la composition réalisée.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.LocalDate;
import java.time.ZonedDateTime;
import java.time.chrono.Chronology;
import java.time.format.DateTimeFormatter;
import java.time.format.DateTimeFormatterBuilder;
import java.time.format.FormatStyle;
import java.time.temporal.ChronoField;
import java.util.Locale;

public class TestDateTimeFormatterBuilder {
  public static void main(String[] args) {
    DateTimeFormatterBuilder builder = new DateTimeFormatterBuilder();
    builder.appendText(ChronoField.DAY_OF_MONTH);
    builder.appendLiteral(" ");
    builder.appendText(ChronoField.MONTH_OF_YEAR);
    builder.appendLiteral(" ");
    builder.appendText(ChronoField.YEAR);
    DateTimeFormatter formatter = builder.toFormatter();
    System.out.println(formatter.toString());
    System.out.println(formatter.format(LocalDate.now()));
    String motif = DateTimeFormatterBuilder.getLocalizedDateTimePattern(
       FormatStyle.FULL, FormatStyle.FULL, Chronology.of("ISO"), Locale.FRENCH);
    System.out.println(motif);
    formatter = DateTimeFormatter.ofPattern(motif);
    System.out.println(formatter.format(ZonedDateTime.now()));
  }
}
```

Résultat :

```
Text(DayOfMonth)' 'Text(MonthOfYear)' 'Text(Year)
25 février 2015
EEEE d MMMM yyyy HH' h 'mm z
mercredi 25 février 2015 23 h 57 CET
```

112.6.9. Les calendriers

Par défaut, l'API Date-Time utilise le calendrier ISO-8601 qui repose sur le calendrier Grégorien.

Le package java.time.chrono propose plusieurs autres implémentations de calendriers : Hijrah, Japanese, Minguo, ThaiBuddhist.

L'interface Chronology définit les fonctionnalités d'un calendrier. Un calendrier permet de représenter de manière humaine un point dans le temps.

L'interface Era définit les fonctionnalités d'une ère. Généralement un calendrier contient deux ères mais certains calendriers sont divisés en plusieurs ères comme par exemple le calendrier japonais.

Plusieurs interfaces permettent de définir les fonctionnalités d'un objet temporel pour un calendrier :

- ChronoLocalDate : définit les fonctionnalités pour manipuler une date sans heure dans un calendrier quelconque sans fuseau horaire
- ChronoLocalDateTime : définit les fonctionnalités pour manipuler une date et une heure dans un calendrier quelconque sans fuseau horaire
- ChronoZonedDateTime : définit les fonctionnalités pour manipuler une date et une heure dans un calendrier quelconque avec un fuseau horaire

L'utilisation de ces interfaces est à réserver pour des besoins calendaires spécifiques et ne devrait pas être utilisée dans un contexte courant.

Chaque calendrier doit proposer une implémentation de Chronology, ChronoLocalDate et Era. Le JDK 8 fournit en standard plusieurs implémentations de calendriers :

Calendrier	Chronology	ChronoLocalDate	Era
ISO	IsoChronology	LocalDate	IsoEra
Hijrah	HijrahChronology	HijrahDate	HijrahEra
Japanese	JapaneseChronology	JapaneseDate	JapaneseEra
Minguo	MinguoChronology	MinguoDate	MinguoEra
ThaiBuddhist	ThaiBuddhistChronology	ThaiBuddhistDate	ThaiBuddhistEra

La méthode from() d'un calendrier permet de convertir l'objet temporel fourni en paramètre pour obtenir sa représentation dans le calendrier. Elle lève une exception de type DateTimeException si cette conversion échoue.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.LocalDate;
import java.time.Month;
import java.time.chrono.HijrahChronology;
import java.time.chrono.HijrahDate;
import java.time.chrono.JapaneseChronology;
import java.time.chrono.JapaneseDate;
import java.time.chrono.MinguoChronology;
import java.time.chrono.MinguoDate;
import java.time.chrono.ThaiBuddhistChronology;
import java.time.chrono.ThaiBuddhistDate;
import java.time.format.DateTimeFormatter;
import java.time.format.FormatStyle;
import java.util.Locale;

public class TestChronology {
  public static void main(String[] args) {
    LocalDate date = LocalDate.of(2014, Month.DECEMBER, 25);
    DateTimeFormatter dtf = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT)
                            .withLocale(Locale.getDefault(Locale.Category.FORMAT));

    JapaneseDate jdate = JapaneseDate.from(date);
    dtf = dtf.withChronology(JapaneseChronology.INSTANCE);
    System.out.println("Japanese      : "+dtf.format(jdate));

    HijrahDate hdate = HijrahDate.from(date);
    dtf = dtf.withChronology(HijrahChronology.INSTANCE);
    System.out.println("Hijrah        : "+dtf.format(hdate));

    MinguoDate mdate = MinguoDate.from(date);
    dtf = dtf.withChronology(MinguoChronology.INSTANCE);
    System.out.println("Minguo        : "+dtf.format(mdate));

    ThaiBuddhistDate tdate = ThaiBuddhistDate.from(date);
    dtf = dtf.withChronology(ThaiBuddhistChronology.INSTANCE);
    System.out.println("Thai Buddhist : "+dtf.format(tdate));
  }
}
```

Résultat :

```
Japanese      : 25/12/26 H
Hijrah        : 3/3/1436 AH
Minguo        : 25/12/103 1
Thai Buddhist : 25/12/2557
```

Inversement, il est possible de convertir une date d'un calendrier différent d'ISO en la passant en paramètre de la méthode from() de la classe LocalDate.

112.6.10. Les autres classes de l'API

L'API Date-Time propose quelques classes et interfaces pour des fonctionnalités spécifiques.

112.6.10.1. La classe Clock

Une instance de type Clock permet d'obtenir une date-heure «courante» dans un fuseau horaire.

L'implémentation par défaut est équivalente à une utilisation de la méthode currentTimeMillis() de la classe System et de la méthode getDefault() de la classe TimeZone.

L'utilisation d'une instance de type Clock est facultative car la plupart des classes qui encapsulent des données temporelles possèdent une méthode now() qui permet de créer une instance encapsulant tout ou partie de la date-heure système dans le fuseau horaire par défaut. Cette méthode utilise une instance de type Clock pour obtenir ces informations du système d'exploitation.

Une surcharge de la méthode now() attend en paramètre un objet de type Clock qui permet de fournir sa propre implémentation.

Cependant, le but de la classe Clock est de fournir facilement une implémentation alternative : ceci peut être particulièrement intéressant notamment pour les tests automatisés. Une bonne pratique est alors de permettre l'injection d'une instance de type Clock et d'utiliser cette dernière pour obtenir l'instant courant. Cela permet par exemple d'utiliser l'implémentation par défaut en production et d'utiliser une instance obtenue en invoquant la fabrique fixed() ou offset() lors des tests automatisés. Ceci pour, par exemple, tester le code dans différents fuseaux horaires ou renvoyer systématiquement le même instant à chaque invocation.

La classe Clock est abstraite : il n'est pas possible d'en créer une instance en utilisant l'opérateur new. Il est obligatoire d'utiliser une des méthodes statiques qui sont des fabriques : systemDefaultZone(), system(ZoneId), offset(Clock, Duration), systemUTC(), fixed(Instant, ZoneId), tick(Clock, Duration), tickMinutes(ZoneId) et tickSeconds(ZoneId).

Elle possède plusieurs méthodes :

Méthode	Rôle
boolean equals(Object obj)	Vérifier l'égalité avec l'objet fourni en paramètre
static Clock fixed(Instant fixedInstant, ZoneId zone)	Obtenir une instance qui renvoie toujours le même instant
abstract ZoneId getZone()	Obtenir le fuseau horaire utilisé lors de la détermination des dates et heures
abstract Instant instant()	Obtenir l'instant courant de l'horloge
long millis()	Obtenir le nombre de millisecondes courantes de l'horloge depuis le 1er janvier 1970
static Clock offset(Clock baseClock, Duration offsetDuration)	Obtenir une instance qui va utiliser l'instance de type Clock passée en paramètre pour obtenir la date-heure de base et appliquée la durée fournie comme décalage
static Clock system(ZoneId zone)	Obtenir une instance qui utilise l'horloge système
static Clock systemDefaultZone()	Obtenir une instance qui utilise l'horloge système dans le fuseau horaire par défaut
static Clock systemUTC()	Obtenir une instance qui utilise l'horloge système dans le fuseau horaire UTC
static Clock tick(Clock baseClock, Duration tickDuration)	Obtenir une instance qui renvoie l'instant de la Clock fournie en paramètre tronquée avec la durée fournie
static Clock tickMinutes(ZoneId zone)	Obtenir l'instant courant de l'horloge avec les nanosecondes et les secondes toujours à zéro
static Clock tickSeconds(ZoneId zone)	Obtenir l'instant courant de l'horloge avec les nanosecondes toujours à zéro
abstract Clock withZone(ZoneId zone)	Renvoyer une copie de l'instance courante qui utilise le fuseau horaire passé en paramètre

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.Clock;
import java.time.Duration;
import java.time.Instant;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.time.format.DateTimeFormatter;
import java.util.Locale;

public class TestClock {
  public static void main(String[] args) {
    Clock clock = Clock.systemDefaultZone();
    Instant instant = clock.instant();
    System.out.println("instant=    "+instant);
    DateTimeFormatter df =
      DateTimeFormatter.ofPattern("uuuu-MM-dd hh:mm:ss",Locale.FRENCH);
    System.out.println("            "
      +df.format(ZonedDateTime.ofInstant(instant, ZoneId.systemDefault())));
    clock = Clock.systemUTC();
    System.out.println("systemUTC=  "+clock.instant());
    clock = Clock.system(ZoneId.of("America/Los_Angeles"));
    System.out.println("system=     "+clock.instant());
    clock = Clock.fixed(Instant.parse("2014-12-25T23:59:59Z"), ZoneId.systemDefault());
    System.out.println("fixed=      "+clock.instant());
    clock = Clock.tick(Clock.systemDefaultZone(), Duration.ofDays(400L));
    System.out.println("tick=       "+clock.instant());
    clock = Clock.tick(Clock.systemDefaultZone(), Duration.ofHours(2L));
    System.out.println("tick=       "+clock.instant());
    clock = Clock.tickMinutes(ZoneId.systemDefault());
    System.out.println("tickMinutes="+clock.instant());
  }
}
```

Résultat :

```
instant=    2015-02-10T06:18:52.156Z
            2015-02-10 07:18:52
systemUTC=  2015-02-10T06:18:52.203Z
system=     2015-02-10T06:18:52.203Z
fixed=      2014-12-25T23:59:59Z
tick=       2014-11-26T00:00:00Z
tick=       2015-02-10T06:00:00Z
tickMinutes=2015-02-10T06:18:00Z
```

Par défaut, les instances obtenues avec l'implémentation fournie par le JDK tentent d'utiliser la meilleure horloge disponible sur le système : par défaut c'est la méthode currentTimeMillis() de la classe System ou une autre solution si une meilleure est disponible sur le système pour obtenir l'instant courant. La méthode currentTimeMillis() de la classe System possède cependant des limites en termes de précision et de véracité. Il est possible de définir ses propres implémentations de la classe Clock pour par exemple utiliser un serveur NTP.

Une implémentation personnelle de classe Clock doit respecter plusieurs recommandations :

- être final, immuable et thread-safe
- certaines méthodes peuvent lever une exception mais cela devrait être dans des cas particuliers
- l'implémentation peut être Serializable

112.6.10.2. L'interface TemporalAdjuster et la classe TemporalAdjusters

L'interface java.time.temporal.TemporalAdjuster définit le contrat d'un objet qui contient des traitements pour ajuster une date encapsulée sous la forme d'un type Temporal tel que le premier ou le dernier jour du mois, le prochain mardi, le troisième vendredi du mois, ...

Elle ne définit qu'une seule méthode :

Méthode	Rôle
Temporal adjustInto(Temporal temporal)	Renvoyer une copie ajustée de l'objet temporel fourni en paramètre

La classe TemporalAdjusters propose plusieurs méthodes statiques qui renvoient des implémentations de type TemporalAdjuster pour réaliser plusieurs ajustements courants :

Méthode	Rôle
static TemporalAdjuster dayOfWeekInMonth(int ordinal, DayOfWeek dayOfWeek)	Renvoyer une instance qui encapsule la date courante ajustée au N-ième (précisé par le paramètre ordinal) jour de la semaine (précisé par le paramètre dayOfWeek) du même mois
static TemporalAdjuster firstDayOfMonth()	Renvoyer une instance qui encapsule la date courante ajustée au premier jour du mois
static TemporalAdjuster firstDayOfNextMonth()	Renvoyer une instance qui encapsule la date courante ajustée au premier jour du mois suivant
static TemporalAdjuster firstDayOfNextYear()	Renvoyer une instance qui encapsule la date courante ajustée au premier jour de l'année suivante

static TemporalAdjuster firstDayOfYear()	Renvoyer une instance qui encapsule la date courante ajustée au premier jour de l'année
static TemporalAdjuster firstInMonth(DayOfWeek dayOfWeek)	Renvoyer une instance qui encapsule la date courante ajustée sur le premier dayOfWeek du mois
static TemporalAdjuster lastDayOfMonth()	Renvoyer une instance qui encapsule la date courante ajustée au dernier jour du mois
static TemporalAdjuster lastDayOfYear()	Renvoyer une instance qui encapsule la date courante ajustée au dernier jour de l'année
static TemporalAdjuster lastInMonth(DayOfWeek dayOfWeek)	Renvoyer une instance qui encapsule la date courante ajustée sur le dernier dayOfWeek du même mois
static TemporalAdjuster next(DayOfWeek dayOfWeek)	Renvoyer une instance qui encapsule la date courante ajustée au prochain dayOfWeek
static TemporalAdjuster nextOrSame(DayOfWeek dayOfWeek)	Renvoyer une instance qui ajuste la date à la première occurrence du dayOfWeek qui suit la date ajustée sauf si ce jour est déjà atteint auquel cas la même instance est retournée
static TemporalAdjuster ofDateAdjuster(UnaryOperator<LocalDate> dateBasedAdjuster)	Obtenir une instance qui encapsule l'objet de type UnaryOperator<LocalDate> réalisant l'ajustement
static TemporalAdjuster previous(DayOfWeek dayOfWeek)	Renvoyer une instance qui ajuste la date sur la première occurrence du dayOfWeek fourni qui précède la date à ajuster
static TemporalAdjuster previousOrSame(DayOfWeek dayOfWeek)	Renvoyer une instance qui ajuste la date sur la première occurrence du dayOfWeek fourni qui précède la date ajustée sauf si ce jour est déjà atteint auquel cas la même instance est retournée

L'ajustement de ces méthodes ne concerne que la date : par exemple, si l'instance fournie en paramètre est de type ZonedDateTime alors la valeur retournée aura l'heure et le fuseau horaire originaux.

Toutes les instances retournées par les méthodes statiques sont immuables.

Il existe deux manières d'utiliser un TemporalAdjuster :

- invoquer la méthode adjustInto() de l'instance de type TemporalAdjuster en lui passant l'objet temporel en paramètre
- invoquer la méthode with() de l'objet temporel en lui passant l'instance de type TemporalAdjuster en paramètre

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.DayOfWeek;
import java.time.LocalDate;
import java.time.Month;
import static java.time.temporal.TemporalAdjusters.*;

public class TestTemporalAdjuster {

  public static void main(String[] args) {
    LocalDate date = LocalDate.of(2015, Month.JANUARY, 31);
    System.out.println("premier jour du mois : "
      + date.with(firstDayOfMonth()));
    System.out.println("premier lundi du mois : "
      + date.with(firstInMonth(DayOfWeek.MONDAY)));
    System.out.println("dernier jour du mois : "
      + date.with(lastDayOfMonth()));
    System.out.println("premier jour de l'année : "
      + date.with(firstDayOfYear()));
    System.out.println("troisieme mardi du mois : "
      + date.with(dayOfWeekInMonth(3, DayOfWeek.FRIDAY)));
    System.out.println("prochain vendredi du mois : "
      + date.with(next(DayOfWeek.FRIDAY)));
    System.out.println("prochain samedi du mois ou lui meme : "
      + date.with(nextOrSame(DayOfWeek.SATURDAY)));
    System.out.println("premier jour du mois suivant : "
      + date.with(firstDayOfNextMonth()));
    System.out.println("premier jour de l'année suivante : "
      + date.with(firstDayOfNextYear()));
    System.out.println("premier jour du mois : "
      + firstDayOfMonth().adjustInto(date));
    System.out.println("premier lundi du mois : "
      + firstInMonth(DayOfWeek.MONDAY).adjustInto(date));
    System.out.println("dernier jour du mois : "
      + lastDayOfMonth().adjustInto(date));
    System.out.println("troisieme mardi du mois : "
      + dayOfWeekInMonth(3, DayOfWeek.MONDAY).adjustInto(date));
    System.out.println("prochain vendredi du mois : "
      + next(DayOfWeek.FRIDAY).adjustInto(date));
    System.out.println("prochain samedi du mois ou lui-meme : "
      + nextOrSame(DayOfWeek.SATURDAY).adjustInto(date));
    System.out.println("premier jour de l'année : "
      + firstDayOfYear().adjustInto(date));
    System.out.println("premier jour du mois suivant : "
      + firstDayOfNextMonth().adjustInto(date));
    System.out.println("premier jour de l'année suivante : "
      + firstDayOfNextYear().adjustInto(date));
  }
}
```

Résultat :

```
premier jour du mois : 2015-01-01
premier lundi du mois : 2015-01-05
dernier jour du mois : 2015-01-31
premier jour de l'année : 2015-01-01
troisieme mardi du mois : 2015-01-16
prochain vendredi du mois : 2015-02-06
prochain samedi du mois ou lui meme : 2015-01-31
premier jour du mois suivant : 2015-02-01
premier jour de l'année suivante : 2016-01-01
premier jour du mois : 2015-01-01
premier lundi du mois : 2015-01-05
dernier jour du mois : 2015-01-31
troisieme mardi du mois : 2015-01-19
prochain vendredi du mois : 2015-02-06
prochain samedi du mois ou lui-meme : 2015-01-31
premier jour de l'année : 2015-01-01
premier jour du mois suivant : 2015-02-01
premier jour de l'année suivante : 2016-01-01
```

Il est possible de développer ses propres TemporalAdjuster pour des besoins spécifiques. Pour cela, il faut créer une classe qui implémente l'interface TemporalAdjuster.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.DayOfWeek;
import java.time.LocalDate;
import java.time.Month;
import java.time.temporal.Temporal;
import java.time.temporal.TemporalAdjuster;
import static java.time.temporal.TemporalAdjusters.*;

public class JourDePaieAdjuster implements TemporalAdjuster {

  @Override
  public Temporal adjustInto(Temporal date) {
    LocalDate resultat = LocalDate.from(date);
    if (resultat.getDayOfMonth() >= 20) {
        resultat = resultat.with(firstDayOfNextMonth());
    }
    resultat = resultat.withDayOfMonth(20);

    if (resultat.getDayOfWeek() == DayOfWeek.SATURDAY
        || resultat.getDayOfWeek() == DayOfWeek.SUNDAY) {
      resultat = resultat.with(previous(DayOfWeek.FRIDAY));
    }
    return date.with(resultat);
  }
}
```

Dans l'exemple ci-dessus, la classe JourDePaieAdjuster permet de déterminer la date de prochaine paie par rapport à la date fournie en paramètre selon les règles suivantes :

- si le jour du mois de la date est inférieur à 20 alors on prend le 20 du mois courant
- sinon on prend le 20 du mois suivant
- si ce 20ième jour est un samedi ou un dimanche alors on détermine le vendredi précédent

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.LocalDate;
import java.time.Month;
import java.time.temporal.Temporal;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class JourDePaieAdjusterTest {

  private JourDePaieAdjuster sut;

  public JourDePaieAdjusterTest() {
  }

  @Before
  public void setUp() {
    sut = new JourDePaieAdjuster();
  }

  /**
   * Test la date est avant le 20 du mois
   */
  @Test
  public void testAdjustIntoAvantLe20() {
    // given
    Temporal date = LocalDate.of(2015, Month.JANUARY, 15);
    Temporal expResult = LocalDate.of(2015, Month.JANUARY, 20);
    // when
    Temporal result = sut.adjustInto(date);
    // then
    assertEquals("Mauvaise date ajustee", expResult, result);
  }

  /**
   * Test la date est le 20 du mois
   */
  @Test
  public void testAdjustIntoLe20() {
    // given
    Temporal date = LocalDate.of(2015, Month.JANUARY, 20);
    Temporal expResult = LocalDate.of(2015, Month.FEBRUARY, 20);
    // when
    Temporal result = sut.adjustInto(date);
    // then
    assertEquals("Mauvaise date ajustee", expResult, result);
  }

  /**
   * Test la date est le 21 du mois
   */
  @Test
  public void testAdjustIntoLe21() {
    // given
    Temporal date = LocalDate.of(2015, Month.JANUARY, 21);
    Temporal expResult = LocalDate.of(2015, Month.FEBRUARY, 20);
    // when
    Temporal result = sut.adjustInto(date);
    // then
    assertEquals("Mauvaise date ajustee", expResult, result);
  }

  /**
   * Test la date est le 15 juin (le 20 juin est un week end)
   */
  @Test
  public void testAdjustIntoLe15Juin() {
    // given
    Temporal date = LocalDate.of(2015, Month.JUNE, 15);
    Temporal expResult = LocalDate.of(2015, Month.JUNE, 19);
    // when
    Temporal result = sut.adjustInto(date);
    // then
    assertEquals("Mauvaise date ajustee", expResult, result);
  }
}
```

Il est recommandé que l'implémentation soit immuable.

112.6.10.3. L'interface TemporalQuery et la classe TemporalQueries

L'interface java.time.temporal.TemporalQuery définit les fonctionnalités qui permettent d'obtenir des informations d'un objet temporel sous la forme d'une requête. C'est une interface fonctionnelle.

La requête peut retourner n'importe quel type.

Remarque : l'interface TemporalField définit aussi les fonctionnalités pour obtenir la valeur d'un champ d'une date mais sa valeur de retour se limite à un entier long.

Elle ne définit qu'une seule méthode

Méthode	Rôle
R queryFrom(TemporalAccessor temporal)	Traiter la requête sur l'instance fournie en paramètre et renvoyer le résultat sous la forme d'une instance du type generic R

La classe java.time.temporal.TemporalQueries propose plusieurs méthodes statiques qui renvoient des implémentations de type TemporalQuery pour réaliser des requêtes courantes :

Méthode	Rôle
static TemporalQuery<Chronology> chronology()	Appliquer une requête qui renvoie le calendrier ou null si aucun n'est trouvé
static TemporalQuery<LocalDate> localDate()	Appliquer une requête qui renvoie la LocalDate ou null si aucune n'est trouvée
static TemporalQuery<LocalTime> localTime()	Appliquer une requête qui renvoie la LocalTime ou null si aucune n'est trouvée
static TemporalQuery<ZoneOffset> offset()	Appliquer une requête qui renvoie la ZoneOffset ou null si aucune n'est trouvée

static TemporalQuery<TemporalUnit> precision()	Appliquer une requête qui renvoie la plus petite unité temporelle supportée
static TemporalQuery<ZoneId> zone()	Appliquer une requête qui renvoie la ZoneId ou la ZoneOffset si aucune n'est trouvée
static TemporalQuery<ZoneId> zoneId()	Appliquer une requête qui renvoie la ZoneId ou null si aucune n'est trouvée

Ces méthodes sont utiles notamment lorsque l'on ne connaît pas précisément le type de l'objet temporel.

Il existe deux manières d'utiliser un TemporalQuery :

- invoquer la méthode queryFrom() de l'instance de type TemporalQuery en lui passant l'objet temporel en paramètre
- invoquer la méthode query() de l'objet temporel de type TemporalAccessor en lui passant l'instance de type TemporalQuery en paramètre

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.Instant;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.OffsetDateTime;
import java.time.Year;
import java.time.YearMonth;
import java.time.ZoneId;
import java.time.ZoneOffset;
import java.time.ZonedDateTime;
import java.time.chrono.Chronology;
import static java.time.temporal.TemporalQueries.*;
import java.time.temporal.TemporalQuery;
import java.time.temporal.TemporalUnit;

public class TestTemporalQuery {
  public static void main(String[] args) {
    TemporalQuery<TemporalUnit> queryPrecision = precision();
    System.out.println("Precision : ");
    System.out.println("LocalDate : " +  LocalDate.now().query(queryPrecision));
    System.out.println("LocalDateTime : " + LocalDateTime.now().query(queryPrecision));
    System.out.println("  Year : " + Year.now().query(queryPrecision));
    System.out.println("  YearMonth : " + YearMonth.now().query(queryPrecision));
    System.out.println("  Instant : " +  Instant.now().query(queryPrecision));
    TemporalQuery<ZoneId> queryZoneId = zoneId();
    System.out.println("ZoneId : ");
    System.out.println("  ZonedDateTime : " + ZonedDateTime.now().query(queryZoneId));
    System.out.println("  LocalDate : " + LocalDate.now().query(queryZoneId));

    TemporalQuery<Chronology> queryChronology = chronology();
    System.out.println("Chronology : ");
    System.out.println("  ZonedDateTime : " + ZonedDateTime.now().query(queryChronology));
    System.out.println("  Instant : " + Instant.now().query(queryChronology));

    TemporalQuery<ZoneId> queryZone = zone();
    System.out.println("Zone : ");
    System.out.println("  ZonedDateTime : " + ZonedDateTime.now().query(queryZone));
    System.out.println("  OffsetDateTime : " + OffsetDateTime.now().query(queryZone));
    System.out.println("  Instant : " + Instant.now().query(queryZone));

    TemporalQuery<ZoneOffset> queryOffset = offset();
    System.out.println("Offset : ");
    System.out.println("  ZonedDateTime : " + ZonedDateTime.now().query(queryOffset));
    System.out.println("  OffsetDateTime : " + OffsetDateTime.now().query(queryOffset));
    System.out.println("  Instant : " + Instant.now().query(queryOffset));

    TemporalQuery<LocalDate> queryLocalDate = localDate();
    System.out.println("LocalDateTime : ");
    System.out.println("  LocalDateTime : " + ZonedDateTime.now().query(queryLocalDate));
    System.out.println("  Instant : " + Instant.now().query(queryLocalDate));

    TemporalQuery<LocalTime> queryLocalTime = localTime();
    System.out.println("LocalTime : ");
    System.out.println("  LocalTime : " + ZonedDateTime.now().query(queryLocalTime));
    System.out.println("  Instant : " + Instant.now().query(queryLocalTime));
  }
}
```

Résultat :

```
Precision :
  LocalDate : Days
  LocalDateTime : Nanos
  Year : Years
  YearMonth : Months
  Instant : Nanos
ZoneId :
ZonedDateTime : Europe/Paris
  LocalDate : null
Chronology :
  ZonedDateTime : ISO
  Instant : null
Zone :
  ZonedDateTime :
Europe/Paris
  OffsetDateTime : +01:00
  Instant : null
Offset :
  ZonedDateTime : +01:00
  OffsetDateTime : +01:00
  Instant : null
LocalDateTime :
  LocalDateTime : 2015-02-11
  Instant : null
LocalTime :
  LocalTime : 18:43:20.250
  Instant : null
```

Il est possible de définir ses propres requêtes en créant une classe qui implémente l'interface TemporalQuery. Cette implémentation contient les traitements réalisés par la requête : elle doit être thread-safe. L'objet fourni en paramètre ne doit pas être modifié. Celui-ci peut ne pas utiliser le calendrier ISO : l'implémentation doit en tenir compte et éventuellement ne pas exécuter la requête si ce n'est pas le cas.

La valeur de retour peut être null pour indiquer qu'il n'y a pas de résultat à la requête.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.Month;
import java.time.MonthDay;
import java.time.temporal.ChronoField;
import java.time.temporal.TemporalAccessor;
import java.time.temporal.TemporalQuery;

public class EstPrintempsQuery implements TemporalQuery<Boolean> {
  @Override
  public Boolean queryFrom(TemporalAccessor date) {
    Boolean resultat = null;
    int jour = date.get(ChronoField.DAY_OF_MONTH);
    int mois = date.get(ChronoField.MONTH_OF_YEAR);
    MonthDay courant = MonthDay.of(mois, jour);
    MonthDay debut = MonthDay.of(Month.MARCH, 20);
    MonthDay fin = MonthDay.of(Month.JUNE, 21);

    return (courant.isAfter(debut) && courant.isBefore(fin));
  }
}
```

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.LocalDate;
import java.time.Month;
import java.time.temporal.Temporal;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class EstPrintempsQueryTest {
  private EstPrintempsQuery sut;
  @Before
  public void setUp() {
    sut = new EstPrintempsQuery();
  }

  /**
   * Test date avant le printemps
   */
  @Test
  public void testQueryFromAvantPrintemps() {
    // given
    Temporal date = LocalDate.of(2015, Month.JANUARY, 15);
    // when
    Boolean result = sut.queryFrom(date);
    // then
    assertEquals("Mauvaise reponse", Boolean.FALSE, result);
  }

  /**
   * Test date apres le printemps
   */
  @Test
  public void testQueryFromApresPrintemps() {
    // given
    Temporal date = LocalDate.of(2015, Month.OCTOBER, 15);
    // when
    Boolean result = sut.queryFrom(date);
    // then
    assertEquals("Mauvaise reponse", Boolean.FALSE, result);
  }

  /**
   * Test date le premier jour du printemps
   */
  @Test
  public void testQueryFromPremierJourDuPrintemps() {
    // given
    Temporal date = LocalDate.of(2015, Month.MARCH, 21);
    // when
    Boolean result = sut.queryFrom(date);
    // then
    assertEquals("Mauvaise reponse", Boolean.TRUE, result);
  }

  /**
   * Test date le dernier jour du printemps
   */
  @Test
  public void testQueryFromDernierJourDuPrintemps() {
    // given
    Temporal date = LocalDate.of(2015, Month.JUNE, 20);
    // when
    Boolean result = sut.queryFrom(date);
    // then
    assertEquals("Mauvaise reponse", Boolean.TRUE, result);
  }

  /**
   * Test date le dernier jour de l'hiver
   */
  @Test
  public void testQueryFromDernierJourHiver() {
    // given
    Temporal date = LocalDate.of(2015, Month.MARCH, 20);
    // when
    Boolean result = sut.queryFrom(date);
    // then
    assertEquals("Mauvaise reponse", Boolean.FALSE, result);
  }

  /**
   * Test date le premier jour de l'été
   */
  @Test
  public void testQueryFromPermierJourEte() {
    // given
    Temporal date = LocalDate.of(2015, Month.JUNE, 21);
    // when
    Boolean result = sut.queryFrom(date);
    // then
    assertEquals("Mauvaise reponse", Boolean.FALSE, result);
  }
}
```

Certaines méthodes de classes temporelles respectent la signature de l'interface TemporalQuery et peuvent donc être utilisées sous la forme d'une référence de méthode dans une expression lambda de type TemporalQuery. C'est par exemple le cas des méthodes from() des classes LocalDate et ZoneId.

112.6.11. L'utilisation de l'API Date-Time

La définition d'une API aussi riche que l'API Date-Time pose nécessairement plusieurs questions relatives à son utilisation :

- le choix du type d'objet temporel à utiliser
- les exceptions de l'API
- l'intégration avec du code avant Java 8

112.6.11.1. Le choix du type d'objet temporel à utiliser

L'API Date-Time contient de nombreuses classes répondant à la plupart des besoins en matière de gestion de données temporelles utilisant le calendrier ISO :

- Instant : représentation machine d'un point dans la ligne du temps par rapport à l'EPOCH
- LocalDate, LocalTime, LocalDateTime : représentation humaine d'une date et/ou d'une heure sans fuseau horaire
- ZonedDateTime : représentation humaine d'une date et d'une heure avec fuseau horaire
- OffsetTime, OffsetDateTime : représentation humaine d'une date et/ou d'une heure avec décalage horaire
- Duration : une durée mesurée en secondes et nanosecondes
- Period : une période est une quantité temporelle exprimée en années, mois et jours

Cette richesse oblige à choisir la classe la plus adaptée à son besoin selon plusieurs critères :

- représentation humaine ou machine de la donnée
- utilisation d'un fuseau horaire
- date et/ou heure
- pour une date uniquement, quelle combinaison de champs est utile : année, mois, jour

Le tableau ci-dessous synthétise les principaux champs supportés par chaque objet temporel :

Class	Année	Mois	Jour	Heure	Min	Sec	Nano	ZoneOffset	ZoneID

Instant 2014-12-25T17:46:39.567Z						X	X		
LocalDate 2014-12-25	X	X	X						
LocalDateTime 2014-12-25T17:46:39.567	X	X	X	X	X	X	X		
ZonedDateTime 2014-12-25T17:46:39.567+01:00[Europe/Paris]	X	X	X	X	X	X	X	X	X
LocalTime 17:46:39.567				X	X	X	X		
MonthDay --12-25		X	X						
Year 2014	X								
YearMonth 2014-12	X	X							
Month DECEMBER		X							
OffsetDateTime 2014-12-25T17:46:39.567+01:00	X	X	X	X	X	X	X	X	
OffsetTime 17:66:39.567+01:00				X	X	X	X	X	
Duration PT10H				1	1	1	X		
Period P15D	X	X	X						

(1) : la classe Duration ne stocke pas ces données mais propose des méthodes pour convertir la valeur dans ces unités

Il est préférable d'utiliser autant que possible des instances de type LocalDate, LocalTime, LocalDateTime et Instant. L'utilisation d'objets temporels qui gèrent un fuseau horaire rend les calculs beaucoup plus complexes. Ainsi, il est préférable de faire les calculs sur des objets sans fuseau puis de les convertir pour par exemple les afficher dans l'interface graphique.

L'utilisation principale des classes OffsetTime et OffsetDateTime est d'échanger une donnée temporelle sur le réseau ou de la stocker dans une base de données.

112.6.11.2. Les exceptions de l'API

Généralement, l'API tente de respecter plusieurs règles dans l'utilisation des exceptions :

- le passage de null comme argument à une méthode d'un objet temporel lève une exception de type NullPointerException
- les méthodes qui permettent de réaliser des calculs sur des objets temporels peuvent lever une exception unchecked de type java.time.DateTimeException ou ArithmeticException

L'exception de type java.time.temporal.UnsupportedTemporalTypeException est levée par une méthode d'un objet temporel si ce dernier ne supporte pas un champ (ChronoField) ou une unité (ChronoUnit). Pour éviter cette exception, il faut s'assurer que les champs ou les unités utilisées directement ou indirectement sont supportés par l'objet temporel : ce n'est pas toujours évident au premier abord.

112.6.11.3. L'intégration avec le code avant Java 8

Avant Java 8, la gestion des données temporelles était assurée par les classes java.util.Date, java.util.Calendar, java.util.TimeZone et leurs sous-classes.

La conception et l'implémentation de l'API Date-Time est complètement différente des classes historiques de gestion des données temporelles contenues dans le package java.util. Il n'est donc pas toujours facile de trouver une correspondance direct entre les fonctionnalités des deux API. Le tableau ci-dessous tente de fournir une approche assez globale de cette correspondance :

java.util	java.time	Commentaires
Date	Instant	Elles encapsulent toutes les deux un point dans le temps de manière indépendante de tout fuseau horaire
GregorianCalendar	ZonedDateTime	Les méthodes from() et to() de la classe GregorianCalendar permettent de faire les conversions
TimeZone	ZoneId ou ZoneOffset	ZoneId pour un fuseau horaire, ZoneOffset pour un décalage horaire
GregorianCalendar avec l'heure à 00:00	LocalDate	

Pour permettre de faciliter l'intégration avec du code antérieur à Java 8, plusieurs méthodes ont été ajoutées :

- la méthode toInstant() de la classe Calendar permet d'obtenir un Instant à partir d'un Calendar
- la méthode from() de la classe Date permet de créer un objet de type Date à partir d'une instance de type Instant
- la méthode toInstant() de la classe Date permet de créer un objet de type Instant à partir de l'instance de type Date
- la méthode toZonedDateTime() de la classe GregorianCalendar permet d'obtenir une instance de type ZonedDateTime à partir de l'instance de GregorianCalendar
- la méthode from() de la classe GregorianCalendar permet d'obtenir une instance de type GregorianCalendar à partir du ZonedDateTime fourni en paramètre
- la méthode toZoneId() de la classe TimeZone permet d'obtenir une instance de type ZoneId à partir de l'instance de type TimeZone

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.java8.datetime;

import java.time.Instant;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;
import java.util.TimeZone;

public class TestAvecCodeLegacy {
  public static void main(String[] args) {
    Calendar maintenant = Calendar.getInstance();
    ZonedDateTime zonedDateTime =
      ZonedDateTime.ofInstant(maintenant.toInstant(), ZoneId.systemDefault());
    System.out.println("zonedDateTime="+zonedDateTime);

    Date date = new Date();
    Instant instant = date.toInstant();
    System.out.println("instant="+instant);

    date = Date.from(instant);
    System.out.println("date="+date);

    GregorianCalendar calendar = (GregorianCalendar) GregorianCalendar.getInstance();
    TimeZone timezone = calendar.getTimeZone();
    System.out.println("timezone="+timezone);
    int offset = calendar.get(Calendar.ZONE_OFFSET);
    System.out.println("offset="+offset);
    zonedDateTime = calendar.toZonedDateTime();
    System.out.println("zonedDateTime="+zonedDateTime);
    LocalDateTime localDateTime = zonedDateTime.toLocalDateTime();
    System.out.println("localDateTime="+localDateTime);

    calendar = GregorianCalendar.from(zonedDateTime);
    System.out.println("date="+calendar.getTime());
  }
}
```

Résultat :

```
zonedDateTime=2015-03-09T23:32:17.328+01:00[Europe/Paris]
instant=2015-03-09T22:32:17.500Z
date=Mon Mar 09 23:32:17 CET 2015
timezone=sun.util.calendar.ZoneInfo[id="Europe/Paris",offset=3600000,
dstSavings=3600000,useDaylight=true,transitions=184,
lastRule=java.util.SimpleTimeZone[id=Europe/Paris,offset=3600000,
dstSavings=3600000,useDaylight=true,startYear=0,startMode=2,startMonth=2,
startDay=-1,startDayOfWeek=1,startTime=3600000,startTimeMode=2,endMode=2,
endMonth=9,endDay=-1,endDayOfWeek=1,endTime=3600000,endTimeMode=2]]
offset=3600000
zonedDateTime=2015-03-09T23:32:17.515+01:00[Europe/Paris]
localDateTime=2015-03-09T23:32:17.515
date=Mon Mar 09 23:32:17 CET 2015
```

L'API JDBC n'a pas été modifiée spécifiquement pour le support de l'API Date-Time : il suffit simplement d'utiliser les méthodes setObject() et getObject().

Le tableau ci-dessous montre la correspondance entre les types ANSI SQL et Java 8 :

ANSI SQL	Java SE 8
DATE	LocalDate
TIME	LocalTime
TIMESTAMP	LocalDateTime
TIME WITH TIMEZONE	OffsetTime
TIMESTAMP WITH TIMEZONE	OffsetDateTime

Attention : les types "TIME WITH TIME ZONE" et "TIMESTAMP WITH TIME ZONE" sont relativement mal nommés car ils ne contiennent pas de fuseau horaire mais uniquement un offset. Il n'est donc pas possible à partir d'une donnée d'un de ces types de déterminer le fuseau horaire. Si celui-ci est requis, il faut le stocker dans une colonne dédiée.

113. La planification de tâches

Chapitre 113

Niveau : 🟩🟩🟨⬜⬜⬜ Intermédiaire

Il est fréquent de devoir exécuter des tâches planifiées :

- pour une seule exécution
- pour plusieurs exécutions
- pour des exécutions récurrentes
- avec éventuellement un délai d'attente avant la première exécution

La planification de tâches peut requérir différentes fonctionnalités :

- configuration avancée des déclenchements
- persistance des tâches

Le JDK propose des solutions basiques pour planifier l'exécution de tâches. Pour des besoins plus évolués, il est nécessaire d'utiliser une solution tierce : une des plus utilisée est l'API open source Quartz.

Ce chapitre contient plusieurs sections :

♦ La planification de tâches avec l'API du JDK
♦ Quartz

113.1. La planification de tâches avec l'API du JDK

Le JDK propose deux solutions pour la planification basique de l'exécution de tâches :

- Timer et TimerTask : à partir de Java 1.3
- ScheduledExecutorService : à partir de Java 5

La planification peut être pour une exécution unique ou pour une exécution répétée périodiquement avec ou sans délai d'attente avant la première exécution.

113.1.1. Les classes Timer et TimerTask

Depuis Java 1.3, la classe java.util.Timer permet d'exécuter des traitements de manière périodique : elle joue le rôle d'un scheduler simple.

Elle offre une solution simple, fournie en standard dans le JDK, pour exécuter des tâches de manière répétée.

La tâche à exécuter doit être encapsulée dans une classe qui hérite de la classe TimerTask.

113.1.1.1. La classe java.util.TimerTask

La classe abstraite java.util.TimerTask encapsule les traitements d'une tâche qui sera exécutée par un Timer. Elle implémente l'interface Runnable.

Son exécution pourra alors être planifiée en utilisant une instance de type Timer.

La classe java.util.TimerTask permet la planification d'une seule exécution ou de plusieurs. La classe TimerTask est ajoutée à partir de Java 1.3.

Elle ne possède qu'un seul constructeur qui n'attend aucun paramètre. Elle possède plusieurs méthodes :

Méthode	Rôle
boolean cancel()	Annuler la planification des prochaines exécutions. Cette méthode n'interrompt pas les exécutions en cours. Elle renvoie un booléen qui précise si au moins une exécution a été annulée
abstract void run()	Contenir les traitements à exécuter
long scheduledExecutionTime()	Renvoyer l'heure planifiée de la prochaine exécution ou de l'exécution courante si elle est en cours

Pour créer une tâche, il faut créer une classe fille qui hérite de la classe TimerTask.

Exemple (code Java 1.3) :

```
package fr.jmdoudoux.dej.timer;

import java.util.Date;
import java.util.TimerTask;

public class MaTask extends TimerTask {
  @Override
  public void run() {
    System.out.println(new Date() + " Execution de ma tache");
  }
}
```

113.1.1.2. La classe java.util.Timer

La classe Timer est un scheduler basique qui permet la planification de l'exécution d'une tâche pour une seule exécution ou pour une exécution répétée à intervalles réguliers. Elle utilise pour cela un thread dédié.

La classe Timer permet de planifier l'exécution d'une tâche pour :

- une seule exécution à un moment donné
- plusieurs exécutions périodiques espacées d'un même intervalle

Les possibilités de planification du Timer sont donc très limitées : les répétitions ne peuvent se faire que sur l'attente d'un délai exprimé en millisecondes. Il n'est par exemple pas possible de définir des planifications quotidiennes à heures fixes, mensuelles, ... Il faut dans ce cas utiliser une autre solution proposée par un tiers.

Elle possède plusieurs constructeurs :

Constructeur	Rôle
Timer()	Construire une instance, le thread associé n'est pas un démon
Timer(boolean isDaemon)	Construire une instance en précisant si le thread associé au timer est un démon
Timer(String name)	Construire une instance en précisant le nom du thread associé au timer (depuis Java 5)

Timer(String name, boolean isDaemon)	Construire une instance en précisant le nom du thread et si celui-ci est un démon (depuis Java 5)

Tous ces constructeurs lancent le thread qui leur est associé.

Elle possède plusieurs méthodes :

Méthode	Rôle
void cancel()	Terminer le timer : toutes les planifications sont supprimées
int purge()	Retirer de la file toutes les tâches annulées ce qui les rend éligibles pour le ramasse-miettes
void schedule(TimerTask task, Date time)	Planifier l'exécution de la tâche à la date/heure fournies en paramètres. Si celle-ci est dans le passé alors la tâche est exécutée immédiatement
void schedule(TimerTask task, Date firstTime, long period)	Planifier l'exécution répétée de la tâche : la première exécution est prévue à firstTime et le délai entre chaque exécution est précisé par le paramètre period en millisecondes
void schedule(TimerTask task, long delay)	Planifier l'exécution de la tâche après avoir attendu delay millisecondes
void schedule(TimerTask task, long delay, long period)	Planifier l'exécution répétée de la tâche : la première exécution intervient après avoir attendu delay millisecondes et le délai entre chaque exécution est précisé par le paramètre period en millisecondes
void scheduleAtFixedRate(TimerTask task, Date firstTime, long period)	Planifier l'exécution répétée de la tâche : la première exécution est prévue à firstTime et le délai entre chaque exécution est précisé par le paramètre period en millisecondes
void scheduleAtFixedRate(TimerTask task, long delay, long period)	Planifier l'exécution répétée de la tâche : la première exécution intervient après avoir attendu delay millisecondes et le délai entre chaque exécution est précisé par le paramètre period en millisecondes

La planification peut se faire de deux manières en utilisant une des surcharges des méthodes :

- schedule() : l'exécution suivante survient dès que le délai précisé est atteint après la fin de l'exécution précédente. Les déclenchements de l'exécution de la tâche ne sont donc pas fixes puisqu'ils dépendent du temps de l'exécution de la tâche précédente
- scheduleAtFixedRate() : l'exécution est déterminée par rapport au début de la première exécution en utilisant la période de manière fixe. Si une tâche est plus longue que prévue c'est sans influence sur les exécutions suivantes à moins que le temps d'exécution dépasse le délai auquel cas l'exécution suivante attend l'achèvement de la tâche précédente

Exemple (code Java 1.3) :

```
package fr.jmdoudoux.dej.timer;

import java.util.Timer;

public class TestTimer {
  public static void main(final String[] args) {
    Timer timer;
    timer = new Timer();
    timer.schedule(new MaTask(), 1000, 5000);
  }
}
```

Résultat :

```
Sun Jun 15 14:00:11 CEST 2014 Execution de ma tache
Sun Jun 15 14:00:16 CEST 2014 Execution de ma tache
Sun Jun 15 14:00:21 CEST 2014 Execution de ma tache
Sun Jun 15 14:00:26 CEST 2014 Execution de ma tache
Sun Jun 15 14:00:31 CEST 2014 Execution de ma tache
Sun Jun 15 14:00:36 CEST 2014 Execution de ma tache
Sun Jun 15 14:00:41 CEST 2014 Execution de ma tache
Sun Jun 15 14:00:46 CEST 2014 Execution de ma tache
```

Si une tâche est en cours d'exécution par le thread, le Timer attend la fin de son exécution pour exécuter la planification suivante même si le délai est largement dépassé.

Exemple (code Java 1.3) :

```java
package fr.jmdoudoux.dej.timer;

import java.util.Date;
import java.util.TimerTask;

public class MaTache extends TimerTask {
  @Override
  public void run() {
    System.out.println("Debut execution tache " + new Date());
    try {
      Thread.sleep(5000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.println("Fin execution tache " + new Date());
  }
}
```

Exemple (code Java 1.3) :

```java
package fr.jmdoudoux.dej.timer;

import java.util.Timer;
import java.util.TimerTask;

public class TestTimer {
  public static void main(final String[] args) {
    TimerTask timerTask = new MaTache();
    Timer timer = new Timer(true);
    timer.scheduleAtFixedRate(timerTask, 0, 2000);
    System.out.println("Lancement execution");

    try {
      Thread.sleep(20000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    timer.cancel();

    System.out.println("Abandon execution");
    try {
      Thread.sleep(20000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
Lancement execution
Debut execution tache Sun Jun 15 14:25:12 CET 2014
Fin execution tache Sun Jun 15 14:25:17 CET 2014
Debut execution tache Sun Jun 15 14:25:17 CET 2014
Fin execution tache Sun Jun 15 14:25:22 CET 2014
Debut execution tache Sun Jun 15 14:25:22 CET 2014
Fin execution tache Sun Jun 15 14:25:27 CET 2014
Debut execution tache Sun Jun 15 14:25:27 CET 2014
Abandon execution
Fin execution tache Sun Jun 15 14:25:32 CET 2014
```

Dans l'exemple ci-dessus, l'exécution de la tâche est prévue toutes les deux secondes mais son temps d'exécution est de 5 secondes. Il faut s'assurer que le délai d'exécution de la tâche est inférieur au délai d'attente entre deux exécutions, sinon les exécutions vont s'empiler dans la queue.

Une surcharge de la méthode schedule() qui attend en paramètres un objet de type TimerTask et un de type Date permet de demander l'exécution de la tâche à la date/heure fournie en paramètre. Si celle-ci est dans le passé alors elle est exécutée immédiatement.

Exemple (code Java 1.3) :

```
package fr.jmdoudoux.dej.timer;

import java.util.Calendar;
import java.util.Date;
import java.util.Timer;

public class TestTimer {

  public static void main(final String[] args) {

    Calendar calendar = Calendar.getInstance();
    calendar.set(Calendar.HOUR_OF_DAY, 18);
    calendar.set(Calendar.MINUTE, 0);
    calendar.set(Calendar.SECOND, 0);
    Date time = calendar.getTime();

    Timer timer = new Timer();
    timer.schedule(new MaTask(), time);
  }
}
```

Une surcharge de la méthode schedule() qui attend en paramètres un objet de type TimerTask, un de type Date et un de type long permet de demander l'exécution répétée de la tâche à la date/heure fournie en paramètre. Le délai d'attente entre deux exécutions est précisé en millisecondes.

Exemple (code Java 1.3) :

```
package fr.jmdoudoux.dej.timer;

import java.util.Calendar;
import java.util.Date;
import java.util.Timer;

public class TestTimer {

  public static final long VINGT_QUATRE_HEURES = 1000 * 60 * 60 * 24;

  public static void main(final String[] args) {
    Calendar calendar = Calendar.getInstance();
    calendar.set(Calendar.HOUR_OF_DAY, 18);
    calendar.set(Calendar.MINUTE, 00);
    calendar.set(Calendar.SECOND, 0);
    Date time = calendar.getTime();

    Timer timer = new Timer();
    timer.schedule(new MaTask(), time, VINGT_QUATRE_HEURES);
  }
}
```

La méthode cancel() permet d'arrêter le timer.

La classe Timer est thread-safe. Elle ne possède qu'un seul thread pour exécuter ses tâches planifiées. Il est donc important que le temps d'exécution des tâches soit le plus rapide possible pour ne pas éventuellement bloquer l'exécution d'autres tâches.

Le type de thread utilisé par le Timer influe sur la capacité de la JVM à s'arrêter :

- par défaut, le thread est bloquant (il empêche l'arrêt de la JVM tant qu'il est en cours d'exécution) et pour l'arrêter il est nécessaire d'invoquer la méthode cancel()
- démon : le thread n'empêche pas l'arrêt de la JVM

Si le thread du Timer n'est pas un démon, alors il est nécessaire pour arrêter l'application d'invoquer la méthode cancel() : celle-ci va supprimer toutes les planifications et permettre au thread de s'arrêter. Une fois la méthode cancel() invoquée, il n'est plus possible de replanifier l'exécution de tâches par le Timer : par exemple, l'invocation de la méthode schedule() lèvera une exception de type IllegalStateException.

L'utilisation d'un Timer est simple mais présente quelques limitations :

- pas de planification complexe
- exécution d'une seule tâche à la fois
- pas de persistance des planifications et des tâches

113.1.2. Le ScheduledExecutorService

Java 5.0 propose, dans l'API Executor, le ScheduledExecutorService qui permet l'exécution planifiée et éventuellement répétée de tâches en utilisant un pool de threads.

Cette solution dispose de plusieurs avantages par rapport à l'utilisation d'un Timer :

- utilise un pool de threads, ce qui améliore les performances notamment dans le cas de l'exécution en parallèle de plusieurs tâches
- les délais peuvent être précisés avec plusieurs unités
- les tâches à exécuter n'ont plus besoin d'hériter de la classe TimerTask

113.1.2.1. L'interface java.util.concurrent.ScheduledExecutorService

L'interface java.util.concurrent.ScheduledExecutorService hérite de l'interface ExecutorService. Elle définit les fonctionnalités pour planifier l'exécution d'une tâche après un certain délai ou son exécution répétée avec un intervalle de temps fixe.

L'exécution d'une tâche se fait de manière asynchrone dans un thread dédié.

La classe Executors est une fabrique qui permet de créer des instances de type Executor. Elle propose notamment plusieurs méthodes pour créer une instance de type ScheduledExecutorService.

Méthode	Rôle
static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)	Renvoyer une instance de type ScheduledExecutorService qui utilise un pool de threads dont la taille est fournie en paramètre
static ScheduledExecutorService newScheduledThreadPool(int corePoolSize, ThreadFactory threadFactory)	Renvoyer une instance de type ScheduledExecutorService qui utilise un pool de threads dont la taille et la fabrique sont fournies en paramètres
static ScheduledExecutorService newSingleThreadScheduledExecutor()	Renvoyer une instance de type ScheduledExecutorService qui utilise un seul thread pour l'exécution des tâches
static ScheduledExecutorService newSingleThreadScheduledExecutor(ThreadFactory threadFactory)	Renvoyer une instance de type ScheduledExecutorService qui utilise un seul thread pour l'exécution des tâches

static ScheduledExecutorService unconfigurableScheduledExecutorService(ScheduledExecutorService executor)	Renvoyer une instance de type ScheduledExecutorService qui délègue les invocations de ses méthodes vers celles de l'instance fournie en paramètre

Il est possible d'utiliser la méthode execute() héritée de l'interface Executor ou des surcharges de la méthode submit() héritée de l'interface ExecutorService pour demander l'exécution immédiate d'une tâche.

Elle définit aussi plusieurs méthodes qui peuvent avoir en paramètres un délai et ou une période relative mais pas de valeur absolue comme une date ou un timestamp :

Méthode	Rôle
<V> ScheduledFuture<V> schedule(Callable<V> callable, long delay, TimeUnit unit)	Planifier l'exécution du Callable fourni en paramètre après le délai précisé
ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit)	Planifier l'exécution du Runnable fourni en paramètre après le délai précisé
ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)	Planifier l'exécution du Runnable fourni en paramètre après le délai précisé. La prochaine exécution est déterminée en ajoutant l'initialDelay et la période multipliée par le nombre d'exécutions
ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)	Planifier l'exécution du Runnable fourni en paramètre après le délai précisé. La prochaine exécution est déterminée en ajoutant le delay au timestamp de la fin de la dernière exécution

Les méthodes qui permettent de demander une planification renvoient une instance de type ScheduledFuture. Cette interface hérite des interfaces Delayed et Future : elle permet de demander l'annulation de l'exécution, de déterminer si elle est terminée et d'obtenir le résultat de l'exécution.

Lors de l'exécution d'un Runnable, comme il ne peut pas renvoyer de valeur, l'invocation de la méthode get() du ScheduledFuture renvoie toujours null lorsque l'exécution de la tâche est terminée.

113.1.2.2. L'interface java.util.concurrent.ScheduledFuture

Cette interface hérite des interfaces Comparable<Delayed>, Delayed et Future<V>.

Une instance de type ScheduledFuture permet de gérer le statut de l'exécution d'une tâche planifiée : déterminer si elle est terminée, obtenir le résultat, annuler l'exécution, ... C'est notamment une instance de ce type qui est retournée par les méthodes de l'interface ScheduledExecutorService.

113.1.2.3. L'utilisation d'un ScheduledExecutorService

Java 5.0 propose la classe java.util.concurrent.ScheduledThreadPoolExecutor qui implémente l'interface ScheduledExecutorService et hérite de la classe ThreadPoolExecutor.

Elle permet d'exécuter des tâches après un délai d'attente ou de manière périodique en utilisant un pool de threads.

La classe ScheduledExecutorService est un ExecutorService qui peut planifier l'exécution de tâches après un délai d'attente ou l'exécution répétée de tâches avec un intervalle de temps fixe entre chaque exécution. Les tâches sont exécutées de manière asynchrone par un des threads du pool.

Pour obtenir une instance de type ScheduledExecutorService, il est possible :

- d'invoquer un des constructeurs de la classe ScheduledThreadPoolExecutor
- d'utiliser des méthodes de la classe Executors qui sont des fabriques pour obtenir une instance de type ScheduledThreadPoolExecutor (newSingleThreadExecutor() qui permet de créer un seul thread pour l'exécution ou newFixedThreadPool() qui permet de créer un pool de threads dont la taille est fournie en paramètre)

Exemple :

```
package fr.jmdoudoux.dej.executor;

import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledThreadPoolExecutor;

public class TestScheduledExecutorService {

  public static void main(String[] args) {
    ScheduledExecutorService ses1 = new ScheduledThreadPoolExecutor(10);
    ScheduledExecutorService ses2 = Executors.newScheduledThreadPool(10);
  }
}
```

La méthode schedule() permet de demander l'exécution d'une tâche éventuellement différée d'un certain délai initial.

Exemple (code Java 5.0) :

```
package fr.jmdoudoux.dej.executor;

import static java.util.concurrent.TimeUnit.SECONDS;

import java.util.Date;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;

public class TestScheduledExecutorService {

  public static void main(String[] args) {
    final ScheduledExecutorService executor = Executors
        .newScheduledThreadPool(2);

    final Runnable maTache = new Runnable() {
      public void run() {
        System.out.println("Ma tache (" + Thread.currentThread().getName()
            + ") " + new Date());
      }
    };

    ScheduledFuture<String> maTacheFuture = executor.schedule(
        new Callable<String>() {
          @Override
          public String call() throws Exception {
            System.out.println("Execution de la tache ("
                + Thread.currentThread().getName() + ") " + new Date());
            return "Resultat de la tache";
          }
        }, 10, SECONDS);

    System.out.println("Autre traitement (" + Thread.currentThread().getName()
        + ") " + new Date());

    try {
      System.out.println("Resultat = " + maTacheFuture.get());
    } catch (InterruptedException e) {
      e.printStackTrace();
    } catch (ExecutionException e) {
      e.printStackTrace();
    }

    executor.shutdown();
  }
}
```

Résultat :

```
Autre traitement (main) Sun Jun 15 17:42:45 CET 2014
Execution de la tache (pool-1-thread-1) Sun Jun 15 17:42:55 CET 2014
Resultat = Resultat de la tache
```

La méthode scheduleAtFixedRate() permet de demander l'exécution périodique d'une tâche. La première exécution peut être différée d'un certain délai. La période est utilisée à partir de la première exécution pour déterminer l'exécution suivante en ajoutant la période à l'heure de début de l'exécution. Si le temps d'exécution de la tâche est supérieur à la période lors l'exécution suivante est effectuée immédiatement. La tâche ne sera exécutée que par un seul thread.

La tâche est exécutée périodiquement jusqu'à ce qu'une des exécutions lève une exception ou que la méthode shutdown() ou shutdownNow() du ScheduleExecutorService soit invoquée.

Exemple (code Java 5.0) :

```java
package fr.jmdoudoux.dej.executor;

import static java.util.concurrent.TimeUnit.SECONDS;

import java.util.Date;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;

public class TestScheduledExecutorService {

  public static void main(String[] args) {
    final ScheduledExecutorService executor = Executors
        .newScheduledThreadPool(2);

    final Runnable maTache = new Runnable() {
      public void run() {
        System.out.println("Ma tache (" + Thread.currentThread().getName()
            + ") " + new Date());
      }
    };

    final ScheduledFuture<?> maTacheFuture = executor.scheduleAtFixedRate(
        maTache, 10, 10, SECONDS);
    final ScheduledFuture<?> maTacheFuture2 = executor.scheduleAtFixedRate(
        maTache, 5, 10, SECONDS);

    Runtime.getRuntime().addShutdownHook(new Thread() {
      public void run() {
        executor.shutdown();
      }
    });
  }
}
```

Pour planifier l'exécution d'une tâche de manière répétée durant une certaine période, il faut planifier l'exécution de la tâche et l'exécution d'une autre tâche déclenchée à la fin du délai dont le but est d'arrêter l'exécution de la tâche planifiée.

Exemple (code Java 5.0) :

```java
package fr.jmdoudoux.dej.executor;

import static java.util.concurrent.TimeUnit.SECONDS;

import java.util.Date;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;
```

```
public class TestScheduledExecutorService {

  public static void main(String[] args) {
    final ScheduledExecutorService executor = Executors
        .newScheduledThreadPool(2);

    final Runnable maTache = new Runnable() {
      public void run() {
        System.out.println("Ma tache " + new Date());
      }
    };

    final ScheduledFuture<?> maTacheFuture = executor.scheduleAtFixedRate(
        maTache, 10, 10, SECONDS);

    executor.schedule(new Runnable() {
      public void run() {
        maTacheFuture.cancel(true);
        executor.shutdown();
        System.out.println("Arret de l'executor");
      }
    }, 60, SECONDS);
  }
}
```

La méthode scheduleAtFixedDelay() permet de demander l'exécution périodique d'une tâche. La première exécution peut être différée d'un certain délai initial. La prochaine exécution est déterminée en ajoutant le délai à l'heure de fin d'exécution de la tâche.

Les exécutions successives d'une tâche planifiée en utilisant les méthodes scheduleAtFixedRate() ou scheduleWithFixedDelay() ne se chevauchent pas.

Comme tout ExecutorService, il est important d'invoquer la méthode shutdown() ou shutdownNow() lorsqu'il n'est plus utilisé pour arrêter le thread qui lui est associé.

Par défaut, si une tâche est annulée avant son invocation alors son exécution est supprimée mais elle reste dans la pile des tâches. Pour supprimer les tâches annulées de la pile, il faut au préalable avoir invoqué la méthode setRemoveOnCancelPolicy() en lui passant la valeur true en paramètre.

113.2. Quartz

Quartz est une bibliothèque open source largement utilisée pour planifier l'exécution de tâches. C'est une des bibliothèques de planification open source les plus populaires.

Quartz utilise plusieurs concepts

- scheduler : c'est le moteur de gestion de la planification
- job : c'est une tâche à exécuter
- trigger : c'est un déclencheur qui permet de définir la planification d'exécution d'un job
- calendar : définir des moments dans le temps où la planification est omise
- listener : permettre de s'abonner à des événements

Un job peut être associé à plusieurs triggers mais un trigger n'est associé qu'à un seul job.

Quartz peut être téléchargé à l'url : http://www.quartz-scheduler.org/downloads/

Quartz possède une dépendance sur la bibliothèque slf4j.

Plusieurs versions de Quartz ont été diffusées :

- 1.7.2, 1.7.3 (février 2010) :

- 1.8.0, 1.8.1, 1.8.2, 1.8.3, 1.8.4, 1.8.5, 1.8.6 (novembre 2012) :
- 2.0.0, 2.0.1, 2.0.2 (mai 2011) :
- 2.1.0, 2.1.1, 2.1.2, 2.1.3, 2.1.4, 2.1.5, 2.1.6, 2.1.7 (août 2013) :
- 2.2.0, 2.2.1 (septembre 2013) :

Attention : il existe de grosses différences entre les API de la version 1.x et 2.x de Quartz notamment des classes qui sont devenues des interfaces.

113.2.1. Un exemple simple

Cet exemple va simplement afficher un message sur la console toutes les 5 secondes.

Il faut ajouter les dépendances requises (quartz, log4j et slf4j-log4j) au classpath, par exemple avec Maven.

Exemple :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.jmdoudoux.dej.quartz</groupId>
  <artifactId>TestQuartz</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <dependencies>
    <dependency>
      <groupId>org.quartz-scheduler</groupId>
      <artifactId>quartz</artifactId>
      <version>2.2.1</version>
    </dependency>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.17</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
      <version>1.6.6</version>
    </dependency>
 </dependencies>
</project>
```

Il faut définir les traitements à exécuter dans une classe qui implémente l'interface org.quartz.Job.

Exemple :

```
package fr.jmdoudoux.dej.quartz;

import org.quartz.Job;
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;

public class MonJob implements Job {

  @Override
  public void execute(final JobExecutionContext context) throws JobExecutionException {
    System.out.println("Execution de mon job");
  }
}
```

La mise en oeuvre de Quartz pour exécuter une tâche de manière répétitive suit plusieurs étapes :

- invoquer la fabrique pour obtenir une instance de type Scheduler
- créer une instance de type JobDetail
- créer une instance de type Trigger
- démarrer le scheduler
- enregistrer l'association du job et du trigger dans le scheduler

Exemple :

```
package fr.jmdoudoux.dej.quartz;

import java.io.IOException;
import java.util.Date;

import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.SchedulerFactory;
import org.quartz.SimpleTrigger;
import org.quartz.impl.JobDetailImpl;
import org.quartz.impl.StdSchedulerFactory;
import org.quartz.impl.triggers.SimpleTriggerImpl;

public class TestQuartz {

  public static void main(final String[] args) {
    final SchedulerFactory factory = new StdSchedulerFactory();
    Scheduler scheduler = null;
    try {
      scheduler = factory.getScheduler();

      final JobDetailImpl jobDetail = new JobDetailImpl();
      jobDetail.setName("Mon job");
      jobDetail.setJobClass(MonJob.class);

      final SimpleTriggerImpl simpleTrigger = new SimpleTriggerImpl();
      simpleTrigger.setStartTime(new Date(System.currentTimeMillis() + 1000));
      simpleTrigger.setRepeatCount(SimpleTrigger.REPEAT_INDEFINITELY);
      simpleTrigger.setRepeatInterval(5000);
      simpleTrigger.setName("Trigger execution toutes les 5 secondes");

      scheduler.start();
      scheduler.scheduleJob(jobDetail, simpleTrigger);

      System.in.read();
      if (scheduler != null) {
        scheduler.shutdown();
      }
    } catch (final SchedulerException e) {
      e.printStackTrace();
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

113.2.2. Les principales classes et interfaces

Quartz propose une API dont les principales classes et interfaces sont :

- Scheduler : permet d'interagir avec le moteur de planification
- Job : interface qui doit être implémentée par tous les jobs
- JobDetail : permet de définir des instances de type Job
- Trigger : permet de définir un ou plusieurs déclenchements
- JobExecutionContext : interface qui permet un accès aux différentes entités du context d'exécution d'un job
- SchedulerFactory : interface qui décrit les fonctionnalités d'une fabrique d'instances de Scheduler (par défaut, c'est l'implémentation StdSchedulerFactory qui est utilisée)

Depuis la version 2.0, Quartz propose plusieurs builders qui facilitent la création d'instances de différents types grâce à l'utilisation de fluent API :

- JobBuilder : builder pour créer des instances de type JobDetail
- TriggerBuilder : builder pour créer des instances de type Trigger
- DateBuilder : builder pour faciliter la création d'instances de type Date
- ScheduleBuilder (SimpleScheduleBuilder, CronScheduleBuilder, CalendarIntervalScheduleBuilder, DailyTimeIntervalScheduleBuilder) : builder pour créer une planification

Chaque builder propose une ou plusieurs méthodes statiques. Il est possible de faire un import static de ces méthodes pour faciliter leur utilisation :

Exemple :

```
import static org.quartz.JobBuilder.*;
import static org.quartz.SimpleScheduleBuilder.*;
import static org.quartz.CronScheduleBuilder.*;
import static org.quartz.CalendarIntervalScheduleBuilder.*;
import static org.quartz.TriggerBuilder.*;
import static org.quartz.DateBuilder.*;
```

La classe DateBuilder propose des méthodes pour faciliter la création d'instances de type java.util.Date : elle est particulièrement utile pour créer une instance de type Date en fonction de différents critères et/ou de données fournies ce qui est pratique lors de la définition de triggers notamment pour le calcul de dates de début et de fin.

Méthode	Rôle
static Date dateOf(int hour, int minute, int second)	Obtenir une instance de type Date à partir de l'heure précisée en paramètre
static Date dateOf(int hour, int minute, int second, int dayOfMonth, int month)	Obtenir une instance de type Date à partir des éléments précisés en paramètres et l'année courante
static Date dateOf(int hour, int minute, int second, int dayOfMonth, int month, int year)	Obtenir une instance de type Date à partir des différents éléments passés en paramètre
static Date evenHourDate(Date date)	Renvoyer la prochaine heure à partir de celle fournie en paramètre
static Date evenHourDateAfterNow()	Renvoyer la prochaine heure à partir de maintenant
static Date evenHourDateBefore(Date date)	Renvoyer la précédente heure à partir de celle fournie en paramètre
static Date evenMinuteDate(Date date)	Renvoyer la prochaine minute à partir de celle fournie en paramètre
static Date evenMinuteDateAfterNow()	Renvoyer la prochaine minute à partir de celle fournie en paramètre
static Date evenMinuteDateBefore(Date date)	Renvoyer la précédente minute à partir de celle fournie en paramètre
static Date evenSecondDate(Date date)	Renvoyer la prochaine seconde à partir de celle fournie en paramètre
static Date evenSecondDateAfterNow()	Renvoyer la prochaine seconde à partir de maintenant
static Date evenSecondDateBefore(Date date)	Renvoyer la précédente seconde à partir de celle fournie en paramètre
static Date futureDate(int interval, DateBuilder.IntervalUnit unit)	Obtenir la date correspondant à maintenant plus l'intervalle de temps fourni en paramètre (durée et unité)
static Date translateTime(Date date, TimeZone src, TimeZone dest)	Obtenir une instance de type Date qui encapsule la date fournie en paramètre relative au fuseau horaire passé en paramètre

Généralement, les builders proposent une valeur par défaut pour les propriétés qui ne sont pas explicitement valorisées.

113.2.3. Le scheduler

Le scheduler est le coeur de Quartz. Il stocke un ensemble de JobDetail et de Trigger : il a la charge de déclencher les Trigger selon leur configuration et d'exécuter les Job qui lui sont associés.

Ces fonctionnalités sont décrites dans l'interface org.quartz.Scheduler. Elle propose plusieurs méthodes pour :

- gérer le cycle de vie du scheduler
- gérer les JobDetail et les Trigger qu'elle contient
- accéder au ListenerManager qui permet d'enregistrer des listeners sur des événements

Pour obtenir une instance de type Scheduler, il faut utiliser une fabrique de type SchedulerFactory.

Quartz fournit en standard la classe org.quartz.impl.StdSchedulerFactory qui implémente l'interface SchedulerFactory pour créer des instances à partir d'une configuration sous la forme d'un fichier properties. Cette fabrique recherche un fichier de configuration pour paramétrer l'instance créée. Si ce fichier facultatif n'est pas trouvé alors c'est une instance avec la configuration par défaut qui est créée.

La configuration peut aussi être fournie à la fabrique en invoquant une des surcharges de la méthode initialize() qui attend en paramètre :

- un objet de type Properties qui contient la configuration
- une chaîne de caractères qui précise le fichier properties à utiliser
- un InputStream qui permet de lire le flux d'un fichier properties
- sans paramètre : la configuration est chargée du fichier précisé par la variable d'environnement org.quartz.properties de la JVM. Si elle n'est pas définie, la configuration est recherchée dans un fichier quartz.properties qui doit être dans le répertoire de travail ou dans le classpath. S'il n'est pas trouvé, alors c'est celui contenu dans la bibliothèque quartz.jar qui est utilisé pour configurer la nouvelle instance

A sa création, une instance de type Scheduler est en mode stand-by : elle doit être démarrée en invoquant la méthode start().

Exemple :

```
SchedulerFactory schedFact = new org.quartz.impl.StdSchedulerFactory();
Scheduler sched = schedFact.getScheduler();
sched.start();
```

Tant que le scheduler n'est pas démarré aucun trigger n'est déclenché.

Pour arrêter un scheduler, il faut invoquer une des surcharges de la méthode shutdown() :

- en passant en paramètre la valeur true, le scheduler va attendre que tous les jobs en cours d'exécution soient terminés avant que la méthode shutdown() ne se termine
- en passant la valeur false ou aucun paramètre, le scheduler s'arrête mais les jobs en cours d'exécution se poursuivent

Dès qu'un scheduler est éteint en invoquant sa méthode shutdown(), il ne peut pas être redémarré : il est nécessaire de créer une nouvelle instance.

Il est possible de mettre le scheduler en pause en utilisant la méthode standby(). Pour le réactiver, il faut de nouveau invoquer la méthode start().

Pour démarrer Quartz dans une webapp, il y a deux solutions :

- utiliser un ServletContextListener de type QuartzInitializerListener
- utiliser une servlet de type QuartzInitializerServlet

La première solution consiste à déclarer un ServletContextListener de type QuartzInitializerListener. Plusieurs paramètres de configuration peuvent être définis dans les paramètres du contexte.

Exemple :

```
<context-param>
    <param-name>quartz:config-file</param-name>
    <param-value>/mawebapp/config/webapp_quartz.properties</param-value>
</context-param>
<context-param>
    <param-name>quartz:shutdown-on-unload</param-name>
    <param-value>true</param-value>
</context-param>
<context-param>
    <param-name>quartz:wait-on-shutdown</param-name>
    <param-value>false</param-value>
</context-param>
<context-param>
    <param-name>quartz:start-scheduler-on-load</param-name>
    <param-value>true</param-value>
</context-param>

<listener>
    <listener-class>
        org.quartz.ee.servlet.QuartzInitializerListener
    </listener-class>
</listener>
```

Plusieurs paramètres d'initialisation peuvent être définis dans le contexte :

- quartz:config-file permet de préciser le chemin du fichier properties de configuration de Quartz. Par défaut, quartz.properties
- quartz:shutdown-on-unload : booléen qui précise si la méthode shutdown() du scheduler doit être invoquée lorsque la webapp est déchargée. La valeur par défaut est true
- quartz:wait-on-shutdown : lorsque quartz:shutdown-on-unload vaut true, permet de préciser un booléen qui sera passé en paramètre de la méthode shutdown(). La valeur par défaut est false
- quartz:start-on-load : booléen qui permet de préciser si la méthode start() du scheduler sera invoquée lorsque la webapp est chargée. La valeur par défaut est true
- quartz:servlet-context-factory-key : permet de préciser le nom de la clé du contexte qui va contenir l'instance de la fabrique
- quartz:scheduler-context-servlet-context-key : permet de préciser le nom de la clé dans le SchedulerContext qui sera associé à l'instance de type ServletContext
- quartz:start-delay-seconds : permet de préciser un délai entre l'initialisation du scheduler et l'invocation de sa méthode start()

La seconde solution est de déclarer une servlet de type org.quartz.ee.servlet.QuartzInitializerServlet en demandant son chargement au démarrage de la webapp.

Exemple :

```
<servlet>
  <servlet-name>QuartzInitializer</servlet-name>
  <servlet-class>org.quartz.ee.servlet.QuartzInitializerServlet</servlet-class>
  <init-param>
    <param-name>shutdown-on-unload</param-name>
    <param-value>true</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

La servlet va stocker une instance de type StdSchedulerFactory dans le contexte associé par défaut à la clé QuartzFactoryServlet.QUARTZ_FACTORY_KEY.

Exemple :

```
StdSchedulerFactory factory = (StdSchedulerFactory) ctx
```

```
                .getAttribute(QuartzFactoryServlet.QUARTZ_FACTORY_KEY);
```

La servlet possède plusieurs paramètres :

- config-file qui permet de préciser le chemin du fichier de configuration. La valeur par défaut est quartz.properties
- shutdown-on-unload : valeur booléenne qui permet de préciser si la méthode shutdown() du scheduler doit être invoquée lorsque la servlet est déchargée. La valeur par défaut est true
- wait-on-shutdown : lorsque shutdown-on-unload vaut true, permet de préciser un booléen qui sera passé en paramètre de la méthode shutdown(). La valeur par défaut est false
- start-scheduler-on-load : booléen qui permet de préciser si la méthode start() du scheduler sera invoquée lorsque la servlet est chargée. La valeur par défaut est true
- servlet-context-factory-key permet de préciser le nom de la clé du contexte qui va contenir l'instance de la fabrique
- scheduler-context-servlet-context-key permet de préciser le nom de la clé dans le SchedulerContext qui sera associée à l'instance de type ServletContext
- start-delay-seconds permet de préciser un délai entre l'initialisation du scheduler et l'invocation de sa méthode start()

113.2.4. Les Jobs et les Triggers

Les Triggers permettent de définir la condition de déclenchement de l'exécution d'un job. Un trigger peut être associé à un objet de type JobDataMap qui permet de passer à un job des paramètres spécifiques au déclenchement.

Quartz propose plusieurs implémentations de l'interface Trigger notamment SimpleTrigger et CronTrigger.

La classe SimpleTrigger permet de définir la condition :

- d'une seule exécution à un moment donné
- de plusieurs exécutions espacées chacune d'une même durée

La classe CronTrigger permet de définir des conditions d'exécution basées sur le calendrier. Par exemple : exécution tous les jours à 08:00, exécution tous les lundis à 23:00, ...

Pour respecter le principe SOC (Separation of Concern), l'interface Job encapsule les traitements et l'interface Trigger encapsule les conditions de déclenchement d'une ou plusieurs exécutions.

Un Job peut être créé et stocké dans le scheduler sans être associé à un Trigger. Cela permet par exemple de conserver un job dans le scheduler après que tous les triggers qui lui étaient associés soient expirés : il est possible de les replanifier en les associant à de nouveaux triggers. Plusieurs triggers peuvent être associés à un même job.

Les jobs et les triggers sont identifiés par une clé (respectivement JobKey et TriggerKey) lorsqu'ils sont enregistrés dans le scheduler.

Ils peuvent être regroupés dans des catégories par exemple « reporting », « mise à jour », « maintenance », ...

Le nom de la clé doit être unique dans un même groupe.

113.2.5. Les jobs

Les traitements à exécuter sont encapsulés dans une classe qui implémente l'interface org.quartz.Job.

L'interface Job ne définit qu'une seule méthode execute().

Exemple :

```
package fr.jmdoudoux.dej.quartz;

import org.quartz.Job;
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;

public class MonJob implements Job {
  @Override
  public void execute(final JobExecutionContext context) throws JobExecutionException {
    System.out.println("Execution de mon job");
  }
}
```

Il ne faut pas créer une instance de type Job directement mais créer une instance de type JobDetail qui va permettre à Quartz de créer et gérer une instance.

La classe JobDetailImpl implémente l'interface JobDetail.

Exemple :

```
      final JobDetailImpl jobDetail = new JobDetailImpl();
      jobDetail.setName("Mon job");
      jobDetail.setJobClass(MonJob.class);
```

Il faut au minimum définir le nom qui est l'identifiant et la classe de type Job qui encapsule les traitements.

Il est possible d'associer des paramètres au job en utilisant une instance de type org.quartz.JobDataMap.

Exemple :

```
      JobDataMap map = new JobDataMap();
      map.put("monParametre","12345");
      jobDetail.setJobDataMap(map);
```

Il est alors possible de récupérer ces paramètres dans la méthode execute() du job en invoquant les méthodes getJobDetail() et getJobDataMap() du paramètre de type JobExecutionContext.

Exemple :

```
package fr.jmdoudoux.dej.quartz;

import org.quartz.Job;
import org.quartz.JobDataMap;
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;

public class MonJob implements Job {
  @Override
  public void execute(final JobExecutionContext context) throws JobExecutionException {
    final JobDataMap map = context.getJobDetail().getJobDataMap();
    final String monParametre = map.getString("monParametre");
    System.out.println("Execution de mon job pour " + monParametre);
  }
}
```

La classe JobDetailImpl possède plusieurs constructeurs qui sont tous deprecated sauf le constructeur par défaut. Plutôt que d'utiliser ces constructeurs, il est préférable d'utiliser la classe JobBuilder qui implémente le motif de conception monteur.

Elle définit plusieurs méthodes pour configurer et obtenir l'instance de type JobDetail.

Méthode	Rôle

JobDetail build()	Obtenir l'instance de type JobDetail
static JobBuilder newJob()	Fabrique pour obtenir l'instance de type JobBuilder
static JobBuilder newJob(Class<? extends Job> jobClass)	Fabrique pour obtenir l'instance de type JobBuilder en passant en paramètre la classe qui implémente l'interface Job
JobBuilder ofType(Class<? extends Job> jobClazz)	Préciser la classe qui implémente l'interface Job
JobBuilder requestRecovery()	Préciser au scheduler de réexécuter le job si une situation de recovery ou de fail-over survient
JobBuilder requestRecovery(boolean jobShouldRecover)	Préciser au scheduler de réexécuter ou non le job si une situation de recovery ou de fail-over survient
JobBuilder setJobData(JobDataMap newJobDataMap)	Préciser les paramètres associés au job
JobBuilder storeDurably()	Préciser au scheduler de conserver le job si celui-ci n'est plus associé à un trigger
JobBuilder storeDurably(boolean jobDurability)	Préciser au scheduler de conserver ou non le job si celui-ci n'est plus associé à un trigger
JobBuilder usingJobData(JobDataMap newJobDataMap)	Ajouter les paramètres fournis à ceux associés au job
JobBuilder usingJobData(String dataKey, Boolean value) JobBuilder usingJobData(String dataKey, Double value) JobBuilder usingJobData(String dataKey, Float value) JobBuilder usingJobData(String dataKey, Integer value) JobBuilder usingJobData(String dataKey, Long value) JobBuilder usingJobData(String dataKey, String value)	Ajouter un paramètre au JobDataMap
JobBuilder withDescription(String jobDescription)	Préciser une description
JobBuilder withIdentity(JobKey jobKey)	Préciser l'identifiant du job
JobBuilder withIdentity(String name)	Préciser le nom du job qui sera utilisé dans son identifiant
JobBuilder withIdentity(String name, String group)	Préciser le nom du job et le nom de son groupe qui seront utilisés dans son identifiant

La méthode newJob() est une fabrique pour créer une instance de type JobBuilder. Il suffit alors d'invoquer les différentes méthodes pour configurer le builder et invoquer sa méthode build() pour obtenir l'instance correspondante.

Exemple :

```
      final JobDetail jobDetail = JobBuilder
          .newJob(MonJob.class)
          .withIdentity("monJob", "groupe_1")
          .usingJobData("monParametre", "12345")
          .build();
```

Les données contenues dans le JobDataMap sont partagées entre les différentes exécutions d'un même Trigger. Il est donc nécessaire de s'assurer de la concurrence d'accès pour mettre à jour ces données.

Depuis la version 2.0, il faut annoter la classe du job avec l'annotation @PersistJobDataAfterExecution pour préciser que le job met à jour les données partagées dans son contexte.

L'interface StatefulJob doit être implémentée par la classe d'un job pour préciser qu'il n'est pas possible d'exécuter le job en parallèle. Si plusieurs Triggers déclenchent l'exécution du job en même temps, ceux-ci seront exécutés les uns après les autres.

Depuis la version 2.0, cette interface est deprecated : il est préférable d'annoter la classe du Job avec l'annotation @DisallowConcurrentExecution. Pour maintenir la compatibilité, l'interface StatefulJob est annotée avec @DisallowConcurrentExecution et @PersistJobDataAfterExecution.

La classe JobExecutionContext encapsule des informations liées à l'exécution du job notamment en permettant un accès aux instances du Scheduler, du Trigger et du JobDetail.

La classe JobDetail encapsule la configuration du job à fournir au scheduler.

A chaque fois que l'exécution d'un job est déclenchée, le moteur de Quartz va créer une nouvelle instance à partir du type fourni en paramètre dans le JobDetail en utilisant par défaut la fabrique JobFactory.

C'est la raison pour laquelle, la classe qui implémente :

- doit avoir un constructeur par défaut
- ne doit pas avoir de membres pour conserver son état

L'échange de données entre plusieurs exécutions peut se faire en utilisant une instance de type JobDataMap.

Toutes les données stockées dans un JobDataMap doivent être sérialisables.

Exemple :

```
package fr.jmdoudoux.dej.quartz;

import org.quartz.Job;
import org.quartz.JobDataMap;
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;
import org.quartz.JobKey;

public class MonJob implements Job {

  @Override
  public void execute(final JobExecutionContext context) throws JobExecutionException {
    final JobDataMap map = context.getJobDetail().getJobDataMap();
    final String monParametre = map.getString("monParametre");

    final JobKey key = context.getJobDetail().getKey();

    System.out.println("Execution de '" + key + "' pour " + monParametre);
  }
}
```

Dans la classe qui encapsule le job, il est possible de définir des setters pour des propriétés correspondants aux noms des paramètres passés. L'implémentation de JobFactory va alors par introspection invoquer ses setters en leur passant en paramètres les valeurs correspondantes au moment de la création de l'instance. Ceci évite d'avoir à obtenir l'instance de type JobDataMap du context et d'extraire les valeurs.

Exemple :

```
package fr.jmdoudoux.dej.quartz;

import org.quartz.Job;
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;
import org.quartz.JobKey;

public class MonJob implements Job {
  private String monParametre;

  @Override
  public void execute(final JobExecutionContext context) throws JobExecutionException {

    final JobKey key = context.getJobDetail().getKey();

    System.out.println("Execution de '" + key + "' pour " + this.monParametre);
  }

  public void setMonParametre(final String monParametre) {
    this.monParametre = monParametre;
  }
}
```

Les données contenues dans une instance de type JobDataMap obtenue à partir du contexte contiennent les paramètres du JobDataMaps, du JobDetail et du Trigger. Si deux paramètres ont le même nom c'est celui du trigger qui est prioritaire.

L'annotation @DisallowConcurrentExecution utilisée sur une classe qui encapsule un job permet de préciser à Quartz de ne pas exécuter de manière concurrente plusieurs instances d'un même JobDetail utilisant la classe. La discrimination ne se fait donc pas sur une instance de la classe mais sur une instance de type JobDetail.

L'annotation @PersistJobDataAfterExecution utilisée sur une classe qui encapsule un job permet de préciser à Quartz de mettre à jour les données de la copie de JobDataMap après que la méthode execute() a été correctement exécutée (aucune exception n'est levée durant ses traitements). La prochaine exécution du JobDetail recevra les données mises à jour dans son instance de type JobDataMap. Cette annotation s'utilise sur la classe du job mais s'applique par instance de type JobDetail.

Lorsque l'annotation @PersistJobDataAfterExecution est utilisée, il est fortement recommandé d'utiliser aussi l'annotation @DisallowConcurrentExecution pour éviter des problèmes liés aux valeurs sauvegardées dans le JobDataMap à la fin de l'exécution concurrente de JobDetail.

Un JobDetail contient plusieurs propriétés :

- Durability : un job qui n'est pas durable est automatiquement retiré du scheduler lorsque plus aucun trigger actif ne lui est associé
- RequestsRecovery : permet de demander la réexécution d'un job qui était en cours d'exécution alors que le scheduler ou la JVM ont été arrêté inopinément. Lors de la réexécution, la méthode JobExecutionContext.isRecovering() renvoie true

Par défaut, le scheduler supprime les jobs qui ne sont plus associés à aucun trigger : pour demander au scheduler de les conserver, il faut invoquer la méthode setDurability(true) du JobDetail correspondant.

En cas de soucis dans les traitements de la méthode execute(), il faut uniquement lever une exception de type JobExecutionException. Généralement le code de la méthode est contenu dans un try/catch global à la méthode pour catcher les exceptions et les chainer dans une nouvelle exception de type JobExecutionException.

L'exception JobExecutionException possède trois propriétés qui seront utilisées par le scheduler soit pour réexécuter le JobDetail soit pour annuler la planification:

- refireImmediately : réexécuter le JobDetail avec le même JobExecutionContext
- unscheduleAllTriggers :
- unscheduleFiringTrigger :

Si la propriété refireImmediately est à true alors les deux autres propriétés sont ignorées.

113.2.5.1. Les jobs fournis par Quartz

Quartz fournit en standard plusieurs implémentations de type Job permettant de répondre à quelques besoins basiques.

Pour utiliser ces jobs, il faut ajouter la bibliothèque quartz-jobs-x.y.z.jar au classpath.

La classe org.quartz.jobs.ee.mail.SendMailJob est une implémentation d'un job qui permet l'envoi d'un mail.

Exemple :

```
JobDetail jobDetail = new JobDetail("emailJob", Scheduler.DEFAULT_GROUP, SendMailJob.class);
JobDataMap map = new JobDataMap();
map.put(SendMailJob.PROP_SMTP_HOST,"smtp.test.fr");
map.put(SendMailJob.PROP_SENDER,"emetteur@test.fr");
map.put(SendMailJob.PROP_RECIPIENT,"destinataire@test.fr");
map.put(SendMailJob.PROP_SUBJECT,"Sujet du mail");
map.put(SendMailJob.PROP_MESSAGE,"Corps du mail");
jobDetail.setJobDataMap(map);
```

L'utilisation de ce job requiert que les bibliothèques javamail et activation soient incluses dans le classpath.

La classe org.quartz.jobs.NoOpJob est un Job qui ne fait rien. Il peut être utile pour permettre de déclencher des traitements exécutés par des listeners sur le trigger ou le job.

La classe org.quartz.jobs.NativeJob permet d'exécuter une commande sur le système d'exploitation.

Exemple :

```
      final JobDataMap map = new JobDataMap();
      map.put(org.quartz.jobs.NativeJob.PROP_COMMAND, "c:\\windows\\notepad.exe");
      map.put("PROP_WAIT_FOR_PROCESS", "true");

      final JobDetail jobDetail = JobBuilder.newJob(NativeJob.class)
        .withIdentity("monJob").usingJobData(map).build();
```

La classe org.quartz.jobs.ee.ejb.EJBInvokerJob permet d'invoquer un EJB.

Les informations nécessaires pour connaitre l'EJB à invoquer doivent être fournies dans le JobDataMap associé au job en utilisant les clés :

- EJB_JNDI_NAME_KEY: le nom JNDI de l'interface home de l'EJB
- EJB_METHOD_KEY: le nom de la méthode de l'EJB à invoquer
- EJB_ARGS_KEY: un tableau contenant les arguments à fournir lors de l'invocation de la méthode de l'EJB
- EJB_ARG_TYPES_KEY: un tableau contenant les types des arguments fournis à la méthode de l'EJB
- INITIAL_CONTEXT_FACTORY (optionnel) : le nom pleinement qualifié de la classe de type ContextFactory à utiliser
- PROVIDER_URL (optionnel) : URL d'accès au conteneur d'EJB

La classe org.quartz.jobs.ee.jmx.JMXInvokerJob permet d'invoquer un MBean.

Les informations nécessaires pour connaitre le MBean à invoquer doivent être fournies dans le JobDataMap associé au job en utilisant les clés : JMX_OBJECTNAME, JMX_METHOD et JMX_PARAMDEFS.

Attention : ce job ne peut invoquer qu'un MBean qui soit enregistré sur un serveur de MBean de la même JVM que le scheduler.

Les classes org.quartz.jobs.ee.jms.SendDestinationMessageJob, org.quartz.jobs.ee.jms.SendQueueMessageJob et org.quartz.jobs.ee.jms.SendTopicMessageJob permettent d'envoyer un message JMS respectivement vers une destination, une queue ou un topic.

Les informations nécessaires à l'émission du message doivent être fournies dans le JobDataMap associé au job.

Attention : cette classe n'est utilisable qu'à partir de JMS 1.1.

La classe org.quartz.jobs.FileScanJob permet de vérifier si la date de dernière modification d'un fichier a été modifiée par rapport à la précédente invocation. Si le fichier a été modifié, un FileScanListener est invoqué.

Les informations nécessaires à l'exécution de ce job doivent être fournies dans le JobDataMap associé au job en utilisant les clés :

- FILE_NAME : chemin et nom du fichier à surveiller
- FILE_SCAN_LISTENER_NAME : nom du FileScanListener à invoquer
- MINIMUM_UPDATE_AGE : durée en millisecondes qui doit s'être écoulée après la date/heure de dernière modification pour considérer le fichier comme modifié

113.2.6. Les triggers

Quartz propose plusieurs implémentations de type Trigger pour répondre à différents besoins de planifications. Les triggers possèdent un certain nombre de propriétés qui permettent de définir les critères de déclenchement.

Chaque trigger est identifié de manière unique par une instance de type TriggerKey.

Plusieurs propriétés sont communes à tous les types de Trigger :

Propriété	Rôle
jobKey	Identité du job qui sera exécuté lors du déclenchement
startTime	Date/heure à partir de laquelle le déclenchement d'événements peut débuter. Ceci permet de définir des triggers par avance
endTime	Date/heure de fin de déclenchement du trigger
priority	Priorité du trigger. Au cas où Quartz ne possède plus de threads pour exécuter des jobs déclenchés au même moment. Dans ce cas, Quartz prend d'abord les triggers ayant la priorité la plus élevée. La valeur peut être négative ou positive. L'heure de déclenchement est toujours prioritaire
misfireInstruction	Une misfire instruction permet de préciser au scheduler comment il doit se comporter si le déclenchement d'un trigger est raté. Le déclenchement raté d'un trigger peut avoir plusieurs origines notamment : • le scheduler est arrêté • aucun thread du pool n'est disponible pour l'exécution du job Chaque trigger possède aussi des misfire instructions qui lui sont propres. L'instruction MISFIRE_INSTRUCTION_SMART_POLICY définit celle par défaut pour un type de trigger.

A partir de la version 2.0 de Quartz, il faut créer des instances de type Trigger en utilisant la classe TriggerBuilder. La classe TriggerBuilder est un builder qui facilite la création d'une instance de type Trigger.

Elle possède plusieurs méthodes pour configurer le trigger et obtenir une instance à partir de cette configuration :

Méthode	Rôle
T build()	Obtenir l'instance de type Trigger
TriggerBuilder<T> endAt(Date endTime)	Définir la date/heure de fin de déclenchement du trigger
TriggerBuilder<T> forJob(JobDetail jobDetail)	Définir le job associé au trigger qui sera obtenu à partir du JobDetail fourni en paramètre
TriggerBuilder<T> forJob(JobKey jobKey)	Définir le job associé au trigger qui sera celui correspondant au JobKey fourni en paramètre
TriggerBuilder<T> forJob(String jobName)	Définir le job associé au trigger qui sera celui correspondant au JobKey dont le nom est fourni en paramètre avec le nom du groupe par défaut

TriggerBuilder<T> forJob(String jobName, String jobGroup)	Définir le job associé au trigger qui sera celui correspondant au JobKey dont le nom et le nom du groupe sont fournis en paramètres
TriggerBuilder<T> modifiedByCalendar(String calendarName)	Définir le nom du Calendar qui sera appliqué pour déterminer les déclenchements
static TriggerBuilder<Trigger> newTrigger()	Obtenir une nouvelle instance de type TriggerBuilder
TriggerBuilder<T> startAt(Date startTime)	Définir la date/heure de début de déclenchement du trigger avec celle fournie en paramètre
TriggerBuilder<T> startNow()	Définir la date/heure de début de déclenchement du trigger à maintenant
TriggerBuilder<T> usingJobData(JobDataMap newJobDataMap)	Définir l'instance de type JobDataMap associée au trigger
TriggerBuilder<T> usingJobData(String key, Boolean value) TriggerBuilder<T> usingJobData(String key, Double value) TriggerBuilder<T> usingJobData(String key, Float value) TriggerBuilder<T> usingJobData(String key, Integer value) TriggerBuilder<T> usingJobData(String key, Long value) TriggerBuilder<T> usingJobData(String key, String value)	Ajouter une paire clé/valeur fournie en paramètres dans l'instance de type JobDataMap
TriggerBuilder<T> withDescription(String description)	Définir la description du trigger
TriggerBuilder<T> withIdentity(String name)	Définir l'identité du trigger : l'instance de type TriggerKey sera composée du nom fourni en paramètre et du nom du groupe par défaut
TriggerBuilder<T> withIdentity(String name, String group)	Définir l'identité du trigger : l'instance de type TriggerKey sera composée du nom et du nom du groupe fournis en paramètres
TriggerBuilder<T> withIdentity(TriggerKey key)	Définir l'identité du trigger : l'instance de type TriggerKey est celle fournie en paramètre
TriggerBuilder<T> withPriority(int priority)	Définir la priorité du trigger
<SBT extends T> TriggerBuilder<SBT> withSchedule(ScheduleBuilder<SBT> scheduleBuilder)	Définir l'instance de type ScheduleBuilder qui permettra de définir la planification des déclenchements

Exemple :

```
package fr.jmdoudoux.dej.quartz;

import java.io.IOException;
import org.quartz.JobBuilder;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.SchedulerFactory;
import org.quartz.SimpleScheduleBuilder;
import org.quartz.Trigger;
import org.quartz.TriggerBuilder;
import org.quartz.impl.StdSchedulerFactory;

public class TestQuartz {
  public static void main(final String[] args) {
    final SchedulerFactory factory = new StdSchedulerFactory();
    Scheduler scheduler = null;
    try {
      scheduler = factory.getScheduler();
      final JobDetail jobDetail = JobBuilder.newJob(MonJob.class)
        .withIdentity("monJob").build();
      final Trigger trigger = TriggerBuilder
          .newTrigger()
          .withIdentity("monTrigger", "monGroup")
          .startNow()
          .withSchedule(SimpleScheduleBuilder.simpleSchedule()
            .withIntervalInHours(1).repeatForever())
          .build();
      scheduler.start();
      scheduler.scheduleJob(jobDetail, trigger);
      System.in.read();
      if (scheduler != null) {
        scheduler.shutdown();
      }
    } catch (final SchedulerException e) {
      e.printStackTrace();
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

Une instance de type JobDataMaps peut être associée à un trigger pour passer des données sous la forme de paires clé/valeur disponibles à chaque déclenchement.

Lorsqu'un déclenchement est raté (le job associé ne peut pas être exécuté au moment prévu), le trigger passe à l'état misfired. Ceci peut par exemple survenir lorsque tous les threads du pool sont déjà occupés à exécuter d'autres jobs déclenchés.

Il est possible de préciser au scheduler le comportement qu'il doit avoir dans une telle circonstance à l'aide d'une misfire instruction. Chaque type de trigger possède un ensemble défini de misfire instructions.

Lorsque le scheduler démarre, il recherche si le déclenchement de triggers persistants a été raté et pour ceux concernés il applique le comportement défini par la misfire instruction configurée dans le trigger.

Quartz propose trois implémentations de l'interface Trigger :

- SimpleTrigger
- CronTrigger
- CalendarIntervalTrigger

113.2.6.1. SimpleTrigger

Un SimpleTrigger permet de planifier :

- une exécution unique immédiate
- une exécution unique à un moment donné
- des exécutions récurrentes en utilisant une période fixe avec un nombre d'exécutions fixe ou illimité

L'implémentation de SimpleTrigger possède plusieurs propriétés qui permettent de définir la ou les conditions de déclenchement :

Propriété	Rôle
repeatCount	Définir le nombre de déclenchements souhaité avant la suppression du trigger. Les valeurs possibles sont zéro, une valeur positive ou la constante SimpleTrigger.REPEAT_INDEFINITELY
endTime	Date/heure de fin des déclenchements et de la suppression du trigger. Si elle est définie, elle prend le pas sur la propriété repeatCount. Ceci permet d'éviter d'avoir à calculer le nombre de répétitions voulues durant la plage de dates. Dans ce cas, la propriété repeatCount doit être supérieure au nombre de déclenchement qui doivent survenir avant la date/heure de fin
repeatInterval	Définir l'intervalle en millisecondes entre chaque répétition. Les valeurs possibles sont zéro ou une valeur positive. La valeur zéro provoque tous les déclenchements de manière concurrente ou quasi concurrente selon les capacités du Scheduler
startTime	Date/heure de début du premier déclenchement

Avant la version 2.0 de Quartz, org.quartz.SimpleTrigger est une classe que l'on peut instancier en invoquant un de ses constructeurs.

Exemple :

```
SimpleTrigger trigger = new SimpleTrigger("monTrigger",
  Scheduler.DEFAULT_GROUP, SimpleTrigger.REPEAT_INDEFINITELY, 30000);
```

Il est possible d'invoquer le constructeur par défaut et d'utiliser ses setters pour configurer l'instance.

Exemple :

```
SimpleTrigger trigger = new SimpleTrigger();
      trigger.setName("monTrigger");
      trigger.setRepeatCount(SimpleTrigger.REPEAT_INDEFINITELY);
      trigger.setRepeatInterval(30000);
```

A partir de la version 2.0 de Quartz, org.quartz.SimpleTrigger est une interface. La classe SimpleTriggerImpl implémente cette interface. Elle possède plusieurs constructeurs qui sont tous deprecated sauf le constructeur par défaut.

Plutôt que d'utiliser ces constructeurs, il faut utiliser la classe TriggerBuilder qui implémente le motif de conception monteur. Pour obtenir une instance de type SimpleTrigger, il faut lui associer un SimpleScheduleBuilder.

La classe SimpleScheduleBuilder implémente le motif de conception monteur pour faciliter la création d'une instance configurée de type SimpleTrigger.

Elle possède plusieurs méthodes pour configurer et obtenir une instance :

Méthode	Rôle
MutableTrigger build()	Obtenir l'instance de type Trigger selon la configuration courante : ne devrait pas être invoquée par le développeur mais uniquement par un TriggerBuilder
SimpleScheduleBuilder repeatForever()	Préciser que la répétition des déclenchements est infinie
static SimpleScheduleBuilder repeatHourlyForever()	Créer une instance configurée pour une exécution infinie à une heure d'intervalle
static SimpleScheduleBuilder repeatHourlyForever(int hours)	Créer une instance configurée pour une exécution infinie au nombre d'heures passé en paramètre d'intervalle
static SimpleScheduleBuilder repeatHourlyForTotalCount(int count)	Créer une instance configurée pour une exécution du nombre de fois passé en paramètre à une heure d'intervalle
static SimpleScheduleBuilder repeatHourlyForTotalCount(int count, int hours)	Créer une instance configurée pour une exécution du nombre de fois passé en paramètre au nombre d'heures passé en paramètre d'intervalle
static SimpleScheduleBuilder repeatMinutelyForever()	Créer une instance configurée pour une exécution infinie à une minute d'intervalle
static SimpleScheduleBuilder repeatMinutelyForever(int minutes)	Créer une instance configurée pour une exécution infinie au nombre de minutes passé en paramètre d'intervalle
static SimpleScheduleBuilder repeatMinutelyForTotalCount(int count)	Créer une instance configurée pour une exécution du nombre de fois passé en paramètre à une minute d'intervalle
static SimpleScheduleBuilder repeatMinutelyForTotalCount(int count, int minutes)	Créer une instance configurée pour une exécution du nombre de fois passé en paramètre au nombre de minutes passé en paramètre d'intervalle
static SimpleScheduleBuilder repeatSecondlyForever()	

	Créer une instance configurée pour une exécution infinie à une seconde d'intervalle
static SimpleScheduleBuilder repeatSecondlyForever(int seconds)	Créer une instance configurée pour une exécution infinie au nombre de secondes passé en paramètre d'intervalle
static SimpleScheduleBuilder repeatSecondlyForTotalCount(int count)	Créer une instance configurée pour une exécution du nombre de fois passé en paramètre à une seconde d'intervalle
static SimpleScheduleBuilder repeatSecondlyForTotalCount(int count, int seconds)	Créer une instance configurée pour une exécution du nombre de fois passé en paramètre au nombre de secondes passé en paramètre d'intervalle
static SimpleScheduleBuilder simpleSchedule()	Créer une instance
SimpleScheduleBuilder withIntervalInHours(int intervalInHours)	Préciser un intervalle de répétitions en heures
SimpleScheduleBuilder withIntervalInMilliseconds(long intervalInMillis)	Préciser un intervalle de répétitions en millisecondes
SimpleScheduleBuilder withIntervalInMinutes(int intervalInMinutes)	Préciser un intervalle de répétitions en minutes
SimpleScheduleBuilder withIntervalInSeconds(int intervalInSeconds)	Préciser un intervalle de répétitions en secondes
SimpleScheduleBuilder withMisfireHandlingInstructionFireNow()	Définir l'instruction de type Trigger.MISFIRE_INSTRUCTION_FIRE_NOW si un déclenchement est raté
SimpleScheduleBuilder withMisfireHandlingInstructionIgnoreMisfires()	Définir l'instruction de type Trigger. MISFIRE_INSTRUCTION_IGNORE_MISFIRE_POLICY
SimpleScheduleBuilder withMisfireHandlingInstructionNextWithExistingCount()	Définir l'instruction de type Trigger.MISFIRE_INSTRUCTION_RESCHEDULE _NEXT_WITH_EXISTING_COUNT si un déclenchement est raté
SimpleScheduleBuilder withMisfireHandlingInstructionNextWithRemainingCount()	Définir l'instruction de type Trigger.MISFIRE_INSTRUCTION_RESCHEDULE _NEXT_WITH_REMAINING_COUNT si un déclenchement est raté
SimpleScheduleBuilder withMisfireHandlingInstructionNowWithExistingCount()	Définir l'instruction de type Trigger.MISFIRE_INSTRUCTION_RESCHEDULE _NOW_WITH_EXISTING_REPEAT_COUNT si un déclenchement est raté
SimpleScheduleBuilder withMisfireHandlingInstructionNowWithRemainingCount()	Définir l'instruction de type Trigger.MISFIRE_INSTRUCTION_RESCHEDULE _NOW_WITH_REMAINING_REPEAT_COUNT si un déclenchement est raté
SimpleScheduleBuilder withRepeatCount(int repeatCount)	Préciser le nombre de fois que le trigger sera déclenché ; le nombre total d'exécutions est égal à la valeur en paramètre + 1

Pour obtenir une instance de type SimpleTrigger, il faut utiliser un TriggerBuilder pour les propriétés principales du trigger et un SimpleScheduleBuilder pour les propriétés spécifiques au SimpleTrigger.

L'exemple ci-dessous planifie une exécution à une heure d'intervalle débutant dans 30 minutes.

Exemple :

```
package fr.jmdoudoux.dej.quartz;
```

```
import java.io.IOException;
import org.quartz.DateBuilder;
import org.quartz.DateBuilder.IntervalUnit;
import org.quartz.JobBuilder;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.SchedulerFactory;
import org.quartz.SimpleScheduleBuilder;
import org.quartz.Trigger;
import org.quartz.TriggerBuilder;
import org.quartz.TriggerKey;
import org.quartz.impl.StdSchedulerFactory;

public class TestQuartz {

  public static void main(final String[] args) {

    final SchedulerFactory factory = new StdSchedulerFactory();
    Scheduler scheduler = null;
    try {
      scheduler = factory.getScheduler();
      final JobDetail jobDetail = JobBuilder.newJob(MonJob.class)
        .withIdentity("monJob").build();
      final Trigger trigger = TriggerBuilder.newTrigger()
        .withIdentity(new TriggerKey("monTrigger", "monGroup"))
        .withSchedule(SimpleScheduleBuilder.simpleSchedule()
          .withIntervalInHours(1).repeatForever())
        .startAt(DateBuilder.futureDate(30, IntervalUnit.MINUTE))
        .build();
      scheduler.start();
      scheduler.scheduleJob(jobDetail, trigger);
      System.in.read();
      if (scheduler != null) {
        scheduler.shutdown();
      }
    } catch (final SchedulerException e) {
      e.printStackTrace();
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

L'exemple ci-dessous planifie une exécution immédiate.

Exemple :

```
package fr.jmdoudoux.dej.quartz;

import java.io.IOException;
import org.quartz.JobBuilder;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.SchedulerFactory;
import org.quartz.SimpleTrigger;
import org.quartz.TriggerBuilder;
import org.quartz.impl.StdSchedulerFactory;

public class TestQuartz {

  public static void main(final String[] args) {
    final SchedulerFactory factory = new StdSchedulerFactory();
    Scheduler scheduler = null;
      try {
        scheduler = factory.getScheduler();
        final JobDetail jobDetail =
JobBuilder.newJob(MonJob.class).withIdentity("monJob").build();
        final SimpleTrigger trigger = (SimpleTrigger) TriggerBuilder
          .newTrigger()
          .withIdentity("monTrigger", "monGroup")
          .build();
      scheduler.start();
      scheduler.scheduleJob(jobDetail, trigger);
      System.in.read();
      if (scheduler != null) {
        scheduler.shutdown();
      }
    } catch (final SchedulerException e) {
      e.printStackTrace();
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

Dans l'exemple ci-dessous, la propriété repeatCount est initialisée à 10 ce qui générera 11 déclenchements.

Exemple :

```
package fr.jmdoudoux.dej.quartz;

import java.io.IOException;
import org.quartz.JobBuilder;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.SchedulerFactory;
import org.quartz.SimpleScheduleBuilder;
import org.quartz.SimpleTrigger;
import org.quartz.TriggerBuilder;
import org.quartz.impl.StdSchedulerFactory;

public class TestQuartz {
  public static void main(final String[] args) {
    final SchedulerFactory factory = new StdSchedulerFactory();
    Scheduler scheduler = null;
    try {
      scheduler = factory.getScheduler();
      final JobDetail jobDetail = JobBuilder.newJob(MonJob.class)
        .withIdentity("monJob").build();
      final SimpleTrigger trigger = TriggerBuilder
          .newTrigger()
          .withIdentity("monTrigger", "monGroup")
          .withSchedule(SimpleScheduleBuilder.simpleSchedule()
            .withIntervalInSeconds(10).withRepeatCount(10))
          .build();
      scheduler.start();
      scheduler.scheduleJob(jobDetail, trigger);
      System.in.read();
      if (scheduler != null) {
        scheduler.shutdown();
      }
    } catch (final SchedulerException e) {
      e.printStackTrace();
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

La classe SimpleTrigger définit plusieurs instructions pour préciser le comportement du scheduler au cas où le déclenchement du trigger est raté.

- MISFIRE_INSTRUCTION_IGNORE_MISFIRE_POLICY : indique au scheduler d'ignorer les déclenchements ratés
- MISFIRE_INSTRUCTION_FIRE_NOW : indique au scheduler de déclencher le trigger immédiatement (à utiliser pour des triggers à déclenchement unique de préférence)

- MISFIRE_INSTRUCTION_RESCHEDULE_NOW_WITH_EXISTING_REPEAT_COUNT : indique au scheduler de déclencher le trigger immédiatement avec le nombre de répétitions inchangé
- MISFIRE_INSTRUCTION_RESCHEDULE_NOW_WITH_REMAINING_REPEAT_COUNT : indique au scheduler de déclencher le trigger immédiatement (à utiliser pour des triggers à déclenchements répétés). Le trigger peut passer directement au statut COMPLETE si le nombre de déclenchement est atteint même si certains d'entre-eux sont ratés
- MISFIRE_INSTRUCTION_RESCHEDULE_NEXT_WITH_REMAINING_COUNT : indique au scheduler de replanifier la prochaine l'exécution avec le nombre de répétitions égal à ce qu'il serait si le déclenchement n'avait pas été raté
- MISFIRE_INSTRUCTION_RESCHEDULE_NEXT_WITH_EXISTING_COUNT : indique au scheduler de replanifier la prochaine l'exécution avec le nombre de répétitions inchangé

113.2.6.2. CronTrigger

La classe CronTrigger permet de planifier des déclenchements en utilisant la syntaxe des expressions de l'outil cron des systèmes Unix. Le CronTrigger est particulièrement utile pour définir des répétitions basées sur des éléments du calendrier.

La configuration se fait en utilisant une expression de type cron. L'expression est une chaîne de caractères contenant des champs séparés par un caractère espace. Celle-ci est analysée et évaluée par une instance de type org.quartz.CronExpression.

La syntaxe d'une expression cron utilise 6 champs obligatoires et un champ facultatif. Chaque champ correspond à un élément permettant de définir la planification. Les champs doivent respecter un certain ordre :

Secondes Minutes Heures Jour_du_mois mois Jour_de_la_semaine Année (optionnel)

Chaque champ possède des valeurs acceptables et utilise éventuellement un caractère ou une combinaison de caractères spéciaux.

Champs	Obligatoire	Valeurs	Caractères spéciaux
Seconde	Oui	0-59	, - * /
Minute	Oui	0-59	, - * /
Heure	Oui	0-23	, - * /
Jour du mois	Oui	1-31	, - * ? L W
Mois	Oui	1-12 ou JAN-DEC ou jan-dec	, - * /
Jour de la semaine	Oui	1-7 ou SUN-SAT ou sun-sat	, - * ? L #
Année	Non	1970-2099	, - * /

Le jour du mois peut avoir une valeur entre 1 et 31 mais cette valeur doit exister pour le mois considéré : c'est particulièrement vrai pour les mois de février (28 ou 29 jours selon que l'année soit bissextile ou non) et avril, juin, septembre et novembre qui ne possède pas de 31.

Le mois peut être précisé en utilisant son numéro (1-12) ou les trois premières lettres de son nom en anglais (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV et DEC)

Le jour de la semaine peut avoir une valeur entre 1 et 7 où 1 est dimanche ou être désigné par les trois premières lettres de son nom en anglais (SUN, MON, TUE, WED, THU, FRI et SAT).

Les valeurs alphabétiques ne sont pas sensibles à la casse.

Les caractères spéciaux permettent de définir des situations particulières :

Caractère	Description

*	Désigner toutes les valeurs possibles pour le champ. Par exemple : dans le champ heure, cela signifie toutes les heures
?	Ne désigner aucune valeur particulière. Permet de préciser dans les champs jour du mois ou jour de la semaine que le champ doit être ignoré au profit de l'autre
-	Désigner une plage de valeurs dont les bornes sont incluses. Par exemple, 1-5 correspond aux valeurs 1, 2, 3, 4 et 5
,	Désigner plusieurs valeurs. Par exemple, MON, WED, FRI dans le champ jour de la semaine correspond à lundi, mercredi et vendredi
/	Désigner une valeur d'incrémentation à partir d'une valeur initiale limitée selon la plage de valeurs acceptables par le champ concerné. Par exemple, 0/15 dans le champ seconde correspond aux valeurs 0, 15, 30 et 45 et 10/10 correspond aux valeurs 10, 20, 30, 40 et 50. Attention : /35 ne signifie pas tous les 35 minutes mais toutes les 35 minutes à partir de la minute 0, ce qui est identique à 0,35 et déclenchera donc deux événements dans une même heure
L	Désigner la dernière valeur acceptable par le champ concerné. Il ne peut être utilisé que sur les champs jour de la semaine et jour du mois. Dans le champ jour du mois, il désigne le dernier jour du mois en tenant compte des années bissextiles pour le mois de février. Par exemple, L dans le champ jour du mois correspond au 31 janvier, 28 février pour les années non bissextiles, le 29 février pour les années bissextiles, le 31 mars, le 30 avril, ... Dans le champ jour de la semaine, cela correspond à la valeur 7 soit samedi. Il est possible de préciser un jour avant le L pour désigner le dernier jour correspondant du mois. Par exemple, 6L correspond au dernier vendredi du mois. Il est aussi possible de préciser un offset après le caractère L. Par exemple, L-3 correspond au troisième avant le dernier jour du mois. Il ne faut surtout pas utiliser plusieurs valeurs avec le caractère L
W	Désigner un jour de la semaine hors week end le plus proche de celui indiqué avant le caractère W. Par exemple, 15W. Si le 15 du mois est un samedi alors l'exécution aura lieu le vendredi 14. Si le 15 du mois est un dimanche alors l'exécution aura lieu le lundi 16. Si le 15 n'est ni un samedi ni un dimanche alors l'exécution a lieu le jour précisé. Il ne faut surtout pas utiliser plusieurs valeurs avec le caractère W.
LW	Désigner le dernier jour du mois hors week end dans le champ jour du mois
#	Désigner le n-ieme jour de la semaine du mois. Par exemple, LUN#1 correspond au premier lundi du mois, 6#3 correspond au troisième vendredi du mois

Les listes de valeurs et les plages peuvent être combinées dans un même champ.

Exemple : MON-WED,SAT qui correspond à lundi, mardi, mercredi et samedi.

Les champs jour du mois et jour de la semaine sont généralement mutuellement exclusifs : si l'un à une ou plusieurs valeurs définies, l'autre peut avoir le caractère ? pour préciser de ne pas tenir compte du champ.

Exemple

Expression cron	Description du déclenchement
0 0 12 * * ?	Tous les jours à midi
0/30 * * * * ?	Toutes les 30 secondes
0 0/5 * * * ?	Toutes les 5 minutes
10 0/5 * * * ?	10 secondes après toutes les 5 minutes

0 30 14 ? * * 0 30 14 * * ? 0 30 14 * * ? *	Tous les jours à 14h30
0 30 14 * * ? 2014	Tous les jours de 2014 à 14h30
0 30 14 * * ? 2014-2016	Tous les jours de 2014, 2015 et 2016 à 14h30
0 * 15 * * ?	Toutes les minutes entre 15h et 15h59
0 0/5 15 * * ?	Toutes les 5 minutes entre 15h et 15h59
0 0/5 10,15 * * ?	Toutes les 5 minutes entre 10h et 10h59 et entre 15h et 15h59
0 0/30 8-9 * * ?	Toutes les demi-heures entre 8h00 et 10h00 (exclu) : 8:00, 8:30, 9:00 et 9:30
0 0-5 15 * * ?	Toutes les minutes entre 10h et 10h05
0 15,45 15 * * ?	Tous les jours à 15h15 et 15h45
0 0 12 ? * WED	Tous les mercredis à midi
0 0 12 ? 1 WED	Tous les mercredis du mois de janvier à midi
0 0 12 ? * MON-FRI	Du lundi au vendredi à midi
0 30 11-13 ? * WED,FRI	Tous les mercredis, jeudis et vendredi à 11H30, 12H30 et 13H30
0 0 12 15 * ?	Le 15 de chaque mois à 12h00
0 0 12 L * ?	Le dernier jour de chaque mois à midi
0 0 12 L-1 * ?	L'avant dernier jour de chaque mois à midi
0 0 12 ? * 6L	Le dernier vendredi de chaque mois à midi
0 0 12 ? * 6#3	Le troisième vendredi de chaque mois à midi
0 0 12 1/5 * ?	Tous les 5 jours de chaque mois à partir du premier jour du mois à midi
59 59 23 25 12 ?	La visite du père Noël chaque année

Parfois, il n'est pas possible de créer certaines conditions dans un seul Trigger notamment si celle-ci est une composition de plusieurs conditions. Dans ce cas, il faut définir plusieurs triggers qui devront tous exécuter le même job.

Attention lors de la définition de triggers qui pourraient être déclenchés durant le passage à l'heure d'été ou d'hiver (daylight savings) car cela pourrait faire rater un déclenchement ou le faire exécuter en double selon le cas.

Avant la version 2.0, CronTrigger était une classe qui possédait de nombreux constructeurs.

Exemple :

```
CronTrigger trigger = new CronTrigger("monTrigger",
Scheduler.DEFAULT_GROUP, "0/30 * * * * ?");
```

Il est aussi possible de créer une instance et d'utiliser les différents setters pour configurer l'instance.

Exemple :

```
      CronTrigger trigger = new CronTrigger();
      trigger.setName("monTrigger");
      trigger.setCronExpression("0/30 * * * * ?");
```

A partir de la version 2.0, CronTrigger est une interface dont la classe CronTriggerImpl est une implémentation.

Pour créer une instance de type CronTrigger, il faut utiliser un TriggerBuilder pour les propriétés générales du trigger et utiliser un CronSchedulerBuilder pour les propriétés spécifiques au CronTrigger.

La classe CronSchedulerBuilder met en oeuvre le motif de conception monteur et propose donc plusieurs méthodes pour configurer et obtenir l'instance.

Méthode	Rôle
static CronScheduleBuilder atHourAndMinuteOnGivenDaysOfWeek(int hour, int minute, Integer... daysOfWeek)	Obtenir une instance de type CronScheduleBuilder correspondant à une expression pour l'heure, la minute et les jours de la semaine fournis en paramètre
MutableTrigger build()	Obtenir l'instance du Trigger. Cette méthode ne devrait être appelée que par un TriggerBuilder
static CronScheduleBuilder cronSchedule(CronExpression cronExpression)	Obtenir une instance de type CronScheduleBuilder correspondant à la CronExpression fournie en paramètre
static CronScheduleBuilder cronSchedule(String cronExpression)	Obtenir une instance de type CronScheduleBuilder correspondant à l'expression fournie en paramètre qui doit être valide
static CronScheduleBuilder cronScheduleNonvalidatedExpression(String cronExpression)	Obtenir une instance de type CronScheduleBuilder correspondant à l'expression fournie en paramètre. Lève une exception de type ParseException si l'expression n'est pas valide
static CronScheduleBuilder dailyAtHourAndMinute(int hour, int minute)	Obtenir une instance de type CronScheduleBuilder correspondant à une expression pour l'heure et la minute fournies en paramètre pour tous les jours de la semaine
CronScheduleBuilder inTimeZone(TimeZone timezone)	Préciser le fuseau horaire à utiliser
static CronScheduleBuilder monthlyOnDayAndHourAndMinute(int dayOfMonth, int hour, int minute)	Obtenir une instance de type CronScheduleBuilder correspondant à une expression pour l'heure, la minute et le jour du mois fournis en paramètre
static CronScheduleBuilder weeklyOnDayAndHourAndMinute(int dayOfWeek, int hour, int minute)	Obtenir une instance de type CronScheduleBuilder correspondant à une expression pour l'heure, la minute et le jour de la semaine fournis en paramètre
CronScheduleBuilder withMisfireHandlingInstructionDoNothing()	Préciser d'utiliser l'instruction de type: CronTrigger.MISFIRE_INSTRUCTION_DO_NOTHING en cas de déclenchement raté
CronScheduleBuilder withMisfireHandlingInstructionFireAndProceed()	Préciser d'utiliser l'instruction de type : CronTrigger.MISFIRE_INSTRUCTION_FIRE_ONCE_NOW en cas de déclenchement raté
CronScheduleBuilder withMisfireHandlingInstructionIgnoreMisfires()	Préciser d'utiliser l'instruction de type: Trigger.MISFIRE_INSTRUCTION_IGNORE_MISFIRE_POLICY en cas de déclenchement raté

Exemple :

```
package fr.jmdoudoux.dej.quartz;

import java.io.IOException;
import java.util.TimeZone;

import org.quartz.CronScheduleBuilder;
import org.quartz.JobBuilder;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.SchedulerFactory;
import org.quartz.Trigger;
import org.quartz.TriggerBuilder;
import org.quartz.impl.StdSchedulerFactory;

public class TestQuartz {

  public static void main(final String[] args) {
    final SchedulerFactory factory = new StdSchedulerFactory();
    Scheduler scheduler = null;
    try {
      scheduler = factory.getScheduler();

      final JobDetail jobDetail = JobBuilder
          .newJob(MonJob.class)
          .withIdentity("monJob", "groupe_1")
          .usingJobData("monParametre", "12345")
          .build();

      final Trigger cronTrigger = TriggerBuilder
          .newTrigger()
          .withIdentity("monTrigger", "groupe_1")
          .withSchedule(
              CronScheduleBuilder.cronSchedule("0/5 * * * * ?")
                .inTimeZone(TimeZone.getTimeZone("Europe/Paris")))
          .build();

      scheduler.start();
      scheduler.scheduleJob(jobDetail, cronTrigger);

      System.in.read();
      if (scheduler != null) {
        scheduler.shutdown();
      }
    } catch (final SchedulerException e) {
      e.printStackTrace();
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

La classe CronTrigger définit plusieurs constantes pour désigner une misfire instruction :

- MISFIRE_INSTRUCTION_DO_NOTHING : déterminer le prochain déclenchement sans effectuer un déclenchement immédiat
- MISFIRE_INSTRUCTION_FIRE_NOW : demander au scheduler d'effectuer un déclenchement immédiat

113.2.6.3. CalendarIntervalTrigger

Depuis la version 2.0 de Quartz, un CalendarIntervalTrigger permet de planifier des déclenchements répétés selon un intervalle de temps dans le calendrier. Ce type de trigger ne peut pas être défini avec SimpleTrigger parce que par exemple tous les mois ne possèdent pas le même nombre de secondes ni avec un CronTrigger par ce que par exemple 5 mois n'est pas divisible de manière entière par 12.

L'interface CalendarIntervalTrigger définit plusieurs méthodes :

Méthode	Rôle
int getRepeatInterval()	Obtenir la valeur de l'interface de répétitions
DateBuilder.IntervalUnit getRepeatIntervalUnit()	Obtenir l'unité de l'intervalle
int getTimesTriggered()	Obtenir le nombre de fois que le trigger a déjà été déclenché
TriggerBuilder<CalendarIntervalTrigger> getTriggerBuilder()	Obtenir un TriggerBuilder configuré pour produire un nouveau trigger identique

L'unité de l'intervalle est définie par l'énumération DateBuilder.IntervalUnit qui définit les valeurs suivantes : DAY, HOUR, MILLISECOND, MINUTE, MONTH, SECOND, WEEK et YEAR

La classe org.quartz.CalendarIntervalScheduleBuilder met en oeuvre le motif de conception monteur et propose donc plusieurs méthodes pour configurer et obtenir l'instance.

Méthode	Rôle
MutableTrigger build()	Obtenir l'instance du Trigger. Cette méthode ne devrait être appelée que par un TriggerBuilder
static CalendarIntervalScheduleBuilder calendarIntervalSchedule()	Obtenir une instance de type CalendarIntervalScheduleBuilder
CalendarIntervalScheduleBuilder withInterval(int interval, DateBuilder.IntervalUnit unit)	Définir l'unité de temps et l'intervalle entre chaque déclenchement
CalendarIntervalScheduleBuilder withIntervalInDays(int intervalInDays)	Définir l'unité de temps et l'intervalle entre chaque déclenchement avec l'unité de temps en jour (IntervalUnit.DAY)
CalendarIntervalScheduleBuilder withIntervalInHours(int intervalInHours)	Définir l'unité de temps et l'intervalle entre chaque déclenchement avec l'unité de temps en heure (IntervalUnit.HOUR)
CalendarIntervalScheduleBuilder withIntervalInMinutes(int intervalInMinutes)	Définir l'unité de temps et l'intervalle entre chaque déclenchement avec l'unité de temps en minute (IntervalUnit.MINUTE)
CalendarIntervalScheduleBuilder withIntervalInMonths(int intervalInMonths)	Définir l'unité de temps et l'intervalle entre chaque déclenchement avec l'unité de temps en mois (IntervalUnit.MONTH)
CalendarIntervalScheduleBuilder withIntervalInSeconds(int intervalInSeconds)	Définir l'unité de temps et l'intervalle entre chaque déclenchement avec l'unité de temps en seconde (IntervalUnit.SECOND)
CalendarIntervalScheduleBuilder withIntervalInWeeks(int intervalInWeeks)	Définir l'unité de temps et l'intervalle entre chaque déclenchement avec l'unité de temps en semaine (IntervalUnit.WEEK)
CalendarIntervalScheduleBuilder withIntervalInYears(int intervalInYears)	Définir l'unité de temps et l'intervalle entre chaque déclenchement avec l'unité de temps en année (IntervalUnit.YEAR)
CalendarIntervalScheduleBuilder withMisfireHandlingInstructionDoNothing()	Utiliser l'instruction CalendarIntervalTrigger.MISFIRE_ INSTRUCTION_DO_NOTHING en cas d'échec du déclenchement du trigger
CalendarIntervalScheduleBuilder withMisfireHandlingInstructionFireAndProceed()	Utiliser l'instruction CalendarIntervalTrigger.MISFIRE_ INSTRUCTION_FIRE_ONCE_NOW en cas d'échec du déclenchement du trigger
CalendarIntervalScheduleBuilder withMisfireHandlingInstructionIgnoreMisfires()	Utiliser l'instruction CalendarIntervalTrigger.MISFIRE_ INSTRUCTION_IGNORE_MISFIRE_POLICY en cas d'échec du déclenchement du trigger

Exemple :

```
package fr.jmdoudoux.dej.quartz;

import java.io.IOException;
import org.quartz.CalendarIntervalScheduleBuilder;
import org.quartz.DateBuilder;
import org.quartz.JobBuilder;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.SchedulerFactory;
import org.quartz.Trigger;
import org.quartz.TriggerBuilder;
import org.quartz.impl.StdSchedulerFactory;

public class TestQuartz {
  public static void main(final String[] args) {
    final SchedulerFactory factory = new StdSchedulerFactory();
    Scheduler scheduler = null;
    try {
      scheduler = factory.getScheduler();
      final JobDetail jobDetail = JobBuilder.newJob(MonJob.class)
        .withIdentity("monJob").build();
      final Trigger trigger = TriggerBuilder
          .newTrigger()
          .withIdentity("monTrigger", "monGroupe")
          .startAt(DateBuilder.tomorrowAt(12, 0, 0))
          .withSchedule(CalendarIntervalScheduleBuilder
            .calendarIntervalSchedule().withIntervalInDays(2))
          .build();
      scheduler.start();
      scheduler.scheduleJob(jobDetail, trigger);
      System.in.read();
      if (scheduler != null) {
        scheduler.shutdown();
      }
    } catch (final SchedulerException e) {
      e.printStackTrace();
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

Si l'intervalle de temps utilisé est le mois, il faut faire attention quand la propriété startTime est proche de la fin du mois. Par exemple, si le trigger est configuré avec un intervalle de 1, une unité de temps égale à MONTH et la propriété startTime initialisée avec 31 janvier alors les déclenchements auront lieu le 28 février, le 28 mars et ainsi de suite même si le mois comporte 30 ou 31 jours. Dans ce cas, il est préférable d'utiliser un CronTrigger configuré pour se déclencher le dernier jour du mois.

113.2.6.4. DailyTimeIntervalTrigger

Depuis la version 2.1 de Quartz, un DailyTimeIntervalTrigger permet de planifier des déclenchements répétés selon un intervalle de temps journalier.

Il permet de planifier un intervalle de répétition exprimé en secondes, minutes ou heures qui peut survenir durant une plage horaire tous les jours ou pour certains jours de la semaine.

L'interface DailyTimeIntervalTrigger définit plusieurs méthodes :

Méthode	Rôle
Set<Integer> getDaysOfWeek()	Définir les jours de la semaine où le déclenchement peut s'appliquer
TimeOfDay getEndTimeOfDay()	Définir l'heure de fin de la plage horaire
int getRepeatInterval()	Définir la valeur de l'intervalle selon l'unité précisée par la propriété repeatIntervalUnit
DateBuilder.IntervalUnit getRepeatIntervalUnit()	Définir l'unité de temps de l'intervalle de répétitions
TimeOfDay getStartTimeOfDay()	Définir l'heure de début de la plage horaire
int getTimesTriggered()	Obtenir le nombre de fois que le trigger a été déclenché
TriggerBuilder<DailyTimeIntervalTrigger> getTriggerBuilder()	Obtenir une instance de type TriggerBuilder qui soit configuré pour créer une instance identique à celle courante

Par défaut, un DailyTimeIntervalScheduleBuilder utilise des valeurs par défaut pour ses propriétés :

- startTimeOfDay : 00:00:00

- endTimeOfDay : 23:59:59
- daysOfWeek : tous les jours (ALL_DAYS_OF_THE_WEEK)
- startTime : maintenant

Tous les jours, la propriété startTime est initialisée avec la valeur de startTimeOfDay. A chaque déclenchement la valeur de repeatinterval est ajoutée jusqu'à atteindre la valeur de endTimeOfDay.

La classe org.quartz.DailyTimeIntervalScheduleBuilder met en oeuvre le motif de conception monteur et propose donc plusieurs méthodes pour configurer et obtenir l'instance.

Méthode	Rôle
MutableTrigger build()	Obtenir l'instance du Trigger. Cette méthode ne devrait être appelée que par un TriggerBuilder
static DailyTimeIntervalScheduleBuilder dailyTimeIntervalSchedule()	Obtenir une instance de DailyTimeIntervalScheduleBuilder
DailyTimeIntervalScheduleBuilder endingDailyAfterCount(int count)	Déterminer l'heure de fin de la plage horaire selon le nombre de répétitions fournis en paramètre, la durée de l'intervalle et l'heure de début de la plage horaire. Ces deux dernières informations doivent être définies avant d'invoquer cette méthode
DailyTimeIntervalScheduleBuilder endingDailyAt(TimeOfDay timeOfDay)	Préciser l'heure de fin de la plage horaire de planification
DailyTimeIntervalScheduleBuilder onDaysOfTheWeek(Integer... onDaysOfWeek)	Définir les jours de la semaine où les déclenchements peuvent survenir
DailyTimeIntervalScheduleBuilder onDaysOfTheWeek(Set<Integer> onDaysOfWeek)	Définir les jours de la semaine où les déclenchements peuvent survenir
DailyTimeIntervalScheduleBuilder onEveryDay()	Définir les déclenchements tous les jours de la semaine
DailyTimeIntervalScheduleBuilder onMondayThroughFriday()	Définir les déclenchements uniquement du lundi au vendredi inclus
DailyTimeIntervalScheduleBuilder onSaturdayAndSunday()	Définir les déclenchements uniquement les samedis et les dimanches
DailyTimeIntervalScheduleBuilder startingDailyAt(TimeOfDay timeOfDay)	Préciser l'heure de début de la plage horaire de planification
DailyTimeIntervalScheduleBuilder withInterval(int timeInterval, DateBuilder.IntervalUnit unit)	Préciser l'intervalle de temps entre chaque répétition (durée et unité de temps)
DailyTimeIntervalScheduleBuilder withIntervalInHours(int intervalInHours)	Préciser un intervalle de temps en heures entre chaque répétition
DailyTimeIntervalScheduleBuilder withIntervalInMinutes(int intervalInMinutes)	Préciser un intervalle de temps en minutes entre chaque répétition
DailyTimeIntervalScheduleBuilder withIntervalInSeconds(int intervalInSeconds)	Préciser un intervalle de temps en secondes entre chaque répétition
DailyTimeIntervalScheduleBuilder withMisfireHandlingInstructionDoNothing()	Utiliser l'instruction DailyTimeIntervalTrigger.MISFIRE_INSTRUCTION _FIRE_DO_NOTHING en cas d'échec du déclenchement du trigger
DailyTimeIntervalScheduleBuilder withMisfireHandlingInstructionFireAndProceed()	Utiliser l'instruction DailyTimeIntervalTrigger.MISFIRE_INSTRUCTION _FIRE_ONCE_NOW en cas d'échec du déclenchement du trigger
DailyTimeIntervalScheduleBuilder withMisfireHandlingInstructionIgnoreMisfires()	Utiliser l'instruction DailyTimeIntervalTrigger.MISFIRE_INSTRUCTION _IGNORE_MISFIRE_POLICY en cas d'échec du

	déclenchement du trigger
DailyTimeIntervalScheduleBuilder withRepeatCount(int repeatCount)	Définir le nombre de répétitions de l'intervalle

L'exemple ci-dessous planifie l'exécution du job du lundi au vendredi à 9h00, 12h00, 15h00 et 18h00.

Exemple :

```
package fr.jmdoudoux.dej.quartz;

import java.io.IOException;

import org.quartz.DailyTimeIntervalScheduleBuilder;
import org.quartz.DailyTimeIntervalTrigger;
import org.quartz.JobBuilder;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.SchedulerFactory;
import org.quartz.TimeOfDay;
import org.quartz.TriggerBuilder;
import org.quartz.impl.StdSchedulerFactory;

public class TestQuartz {

  public static void main(final String[] args) {
    final SchedulerFactory factory = new StdSchedulerFactory();
    Scheduler scheduler = null;
    try {
      scheduler = factory.getScheduler();

      final JobDetail jobDetail = JobBuilder
          .newJob(MonJob.class)
          .withIdentity("monJob")
          .storeDurably()
          .build();

      final DailyTimeIntervalTrigger trigger = TriggerBuilder
          .newTrigger()
          .withIdentity("monTrigger")
          .withSchedule(
              DailyTimeIntervalScheduleBuilder
                  .dailyTimeIntervalSchedule()
                  .withIntervalInHours(3)
                  .onDaysOfTheWeek(
                      DailyTimeIntervalScheduleBuilder.MONDAY_THROUGH_FRIDAY)
                  .startingDailyAt(TimeOfDay.hourAndMinuteOfDay(9, 00))
                  .endingDailyAt(TimeOfDay.hourAndMinuteOfDay(19, 00)))
          .build();

      scheduler.scheduleJob(jobDetail, trigger);
      scheduler.start();

      System.in.read();
      if (scheduler != null) {
        scheduler.shutdown();
      }
    } catch (final SchedulerException e) {
      e.printStackTrace();
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

La classe DailyTimeIntervalTrigger définit plusieurs constantes pour désigner une misfire instruction :

- MISFIRE_INSTRUCTION_DO_NOTHING : déterminer le prochain déclenchement sans effectuer un déclenchement immédiat
- MISFIRE_INSTRUCTION_FIRE_NOW : demander au scheduler d'effectuer un déclenchement immédiat

113.2.6.5. L'exclusion de périodes dans la planification

L'interface org.quartz.Calendar permet de définir des périodes de temps durant lesquelles un trigger ne pourra pas être déclenché. L'utilisation d'un Calendar ne permet que d'exclure des périodes dans le temps.

Par exemple, il est possible de définir un trigger qui s'exécute tous les jours à midi et utiliser un Calendar pour exclure les jours fériés.

Un Calendar Quartz peut être associé à un trigger et est stocké dans le scheduler. Il permet de définir des périodes de temps durant lesquelles un trigger ne pourra pas être déclenché.

L'interface Calendar définit plusieurs méthodes :

Méthode	Rôle
Calendar getBaseCalendar()	Obtenir le calendrier de base
String getDescription()	Obtenir la description
long getNextIncludedTime(long timeStamp)	Obtenir le prochain moment (en millisecondes) qui est inclus dans le calendrier par rapport au moment fourni en paramètre
boolean isTimeIncluded(long timeStamp)	Déterminer si le moment fourni en paramètre (en millisecondes) est inclus dans le calendrier ou pas
void setBaseCalendar(Calendar baseCalendar)	Définir le calendrier de base ou retirer le calendrier de base actuel en passant null en paramètre
void setDescription(String description)	Définir la description du calendrier. C'est une donnée informative qui n'est pas utilisée par Quartz

Ceci permet d'exclure par exemple des jours particuliers : jours fériés, fériés bancaires, jours de fermeture dominicale, ...

La granularité utilisée par un Calendar est la milliseconde, ce qui est beaucoup plus fin que la plupart des besoins qui sont généralement de l'ordre de l'heure ou de la journée. Pour faciliter cette gestion, Quartz propose, dans le package org.quartz.impl.calendar, plusieurs implémentations de l'interface Calendar qui héritent de la classe BaseCalendar.

Classe	Rôle
AnnualCalendar	Exclure un ou plusieurs jours dans l'année
CronCalendar	Exclure les périodes définies par une expression de type cron
DailyCalendar	Exclure une période continue dans une même journée
HolidayCalendar	Exclure les jours de congés
MonthlyCalendar	Exclure un ou plusieurs jours dans le mois
WeeklyCalendar	Exclure un ou plusieurs jours de la semaine. Par exemple : les samedis et les dimanches

Pour utiliser un Calendar, il faut :

- en créer une instance
- la configurer
- l'enregistrer dans le scheduler pour le stocker dans le jobstore courant
- l'associer avec chacun des triggers concernés

L'enregistrement d'un Calendar dans un Scheduler se fait en utilisant sa méthode addCalendar() qui attend en paramètres : le nom du Calendar, son instance, un booléen qui précise si l'instance doit remplacer celle précédemment enregistrée et un autre booléen qui précise si les triggers utilisant déjà le Calendar doivent être mis à jour.

Le nom du Calendar sera utilisé pour associer le Calendar à un trigger. Il est possible d'associer le même Calendar à plusieurs triggers.

Attention : il est nécessaire de bien vérifier le nom du Calendar fournit en paramètre

Exemple :

```
scheduler.addCalendar("jours_feries", cal, true, true);
```

Avant la version 2.0 de Quartz, il faut invoquer la méthode setCalendar() de la classe Trigger pour lui associer un Calendar.

Exemple :

```
trigger.setCalendarName("jours_feries");
```

A partir de la version 2.0, il faut utiliser la méthode modifiedByCalendar() de la classe TriggerBuilder.

Exemple :

```
      final Trigger trigger = TriggerBuilder
          .newTrigger()
          .withIdentity("monTrigger", "monGroupe")
          .modifiedByCalendar("jours_feries")
          .withSchedule(simpleSchedule()
            .withIntervalInSeconds(30)
            .repeatForever())
          .build();
```

La classe org.quartz.impl.calendar.WeeklyCalendar permet de définir des jours de la semaine durant lesquels le déclenchement d'une planification peut avoir lieu. Par défaut, elle exclue le samedi et le dimanche.

La méthode setDayExcluded() permet de préciser pour un jour s'il doit être exclu ou non. Elle attend en paramètre un entier qui précise le jour (le plus simple est d'utiliser une des constantes de la classe java.util.Calendar) et un booléen (true pour exclusion et false pour inclusion).

Exemple :

```
package fr.jmdoudoux.dej.quartz;

import java.io.IOException;
import java.util.Date;
import java.util.List;

import org.quartz.CalendarIntervalScheduleBuilder;
import org.quartz.DateBuilder;
import org.quartz.JobBuilder;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.SchedulerFactory;
import org.quartz.Trigger;
import org.quartz.TriggerBuilder;
import org.quartz.impl.StdSchedulerFactory;
import org.quartz.impl.calendar.WeeklyCalendar;
import org.quartz.spi.OperableTrigger;

public class TestQuartz {

  public static void main(final String[] args) {
    final SchedulerFactory factory = new StdSchedulerFactory();
    Scheduler scheduler = null;
    try {
      scheduler = factory.getScheduler();

      final WeeklyCalendar cal = new WeeklyCalendar();
      cal.setDayExcluded(java.util.Calendar.MONDAY, true);
      cal.setDayExcluded(java.util.Calendar.SUNDAY, true);
      cal.setDayExcluded(java.util.Calendar.SATURDAY, false);

      scheduler.addCalendar("jours_d_ouverture", cal, true, true);

      final JobDetail jobDetail = JobBuilder.newJob(MonJob.class)
        .withIdentity("monJob").storeDurably().build();

      final Trigger trigger = TriggerBuilder
          .newTrigger()
          .withIdentity("monTrigger", "monGroupe")
          .modifiedByCalendar("jours_d_ouverture")
          .startAt(DateBuilder.todayAt(14, 0, 0))
          .withSchedule(CalendarIntervalScheduleBuilder
            .calendarIntervalSchedule().withIntervalInDays(2))
          .build();

      scheduler.scheduleJob(jobDetail, trigger);
      scheduler.start();

      System.in.read();
      if (scheduler != null) {
        scheduler.shutdown();
      }
    } catch (final SchedulerException e) {
      e.printStackTrace();
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

La classe org.quartz.impl.calendar.AnnualCalendar permet de définir des jours dans une année pour lesquels la planification d'un trigger ne se fera pas. Ce type de Calendar ne tient pas compte de l'année : les jours exclus le sont pour toutes les années. Il est donc pratique pour définir les jours fériés fixes d'une année à l'autre.

La méthode setDayExcluded() permet de définir un ou plusieurs jours à exclure. Elle possède deux surcharges :

- la première attend en paramètre une instance de type java.util.Calendar qui encapsule le jour et un booléen qui précise si le jour est exclu (true) ou non (false)
- la seconde attend en paramètre une collection de type ArrayList contenant une ou plusieurs instances de type java.util.Calendar

La méthode removeExcludedDay() permet d'enlever le jour fourni en paramètre de la liste des jours exclus.

Exemple :

```
package fr.jmdoudoux.dej.quartz;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.Date;
import java.util.List;

import org.quartz.CalendarIntervalScheduleBuilder;
import org.quartz.DateBuilder;
import org.quartz.JobBuilder;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.SchedulerFactory;
import org.quartz.Trigger;
import org.quartz.TriggerBuilder;
import org.quartz.impl.StdSchedulerFactory;
import org.quartz.impl.calendar.AnnualCalendar;
import org.quartz.spi.OperableTrigger;

public class TestQuartz {

  public static void main(final String[] args) {
    final SchedulerFactory factory = new StdSchedulerFactory();
    Scheduler scheduler = null;
    try {
      scheduler = factory.getScheduler();

      final ArrayList<Calendar> joursFeriesFixes =
        new ArrayList<Calendar>(7);
      final AnnualCalendar cal = new AnnualCalendar();

      Calendar jourFerie = Calendar.getInstance();
      jourFerie.set(Calendar.MONTH, Calendar.JANUARY);
      jourFerie.set(Calendar.DATE, 1);
      joursFeriesFixes.add(jourFerie);

      jourFerie = Calendar.getInstance();
      jourFerie.set(Calendar.MONTH, Calendar.MAY);
      jourFerie.set(Calendar.DATE, 1);
      joursFeriesFixes.add(jourFerie);

      jourFerie = Calendar.getInstance();
      jourFerie.set(Calendar.MONTH, Calendar.MAY);
      jourFerie.set(Calendar.DATE, 8);
      joursFeriesFixes.add(jourFerie);

      jourFerie = Calendar.getInstance();
      jourFerie.set(Calendar.MONTH, Calendar.JULY);
      jourFerie.set(Calendar.DATE, 14);
      joursFeriesFixes.add(jourFerie);

      jourFerie = Calendar.getInstance();
      jourFerie.set(Calendar.MONTH, Calendar.AUGUST);
      jourFerie.set(Calendar.DATE, 15);
      joursFeriesFixes.add(jourFerie);

      jourFerie = Calendar.getInstance();
      jourFerie.set(Calendar.MONTH, Calendar.NOVEMBER);
      jourFerie.set(Calendar.DATE, 1);
      joursFeriesFixes.add(jourFerie);

      jourFerie = Calendar.getInstance();
      jourFerie.set(Calendar.MONTH, Calendar.NOVEMBER);
      jourFerie.set(Calendar.DATE, 11);
      joursFeriesFixes.add(jourFerie);

      jourFerie = Calendar.getInstance();
      jourFerie.set(Calendar.MONTH, Calendar.DECEMBER);
      jourFerie.set(Calendar.DATE, 25);
      joursFeriesFixes.add(jourFerie);

      cal.setDaysExcluded(joursFeriesFixes);

      scheduler.addCalendar("Jours feries fixes", cal, true, true);

      final JobDetail jobDetail = JobBuilder.newJob(MonJob.class)
        .withIdentity("monJob").storeDurably().build();

      final Trigger trigger = TriggerBuilder
          .newTrigger()
          .withIdentity("monTrigger", "monGroupe")
          .modifiedByCalendar("Jours feries fixes")
          .startAt(DateBuilder.todayAt(14, 0, 0))
          .withSchedule(CalendarIntervalScheduleBuilder
            .calendarIntervalSchedule().withIntervalInDays(2))
          .build();

      scheduler.scheduleJob(jobDetail, trigger);
      scheduler.start();

      System.in.read();
      if (scheduler != null) {
        scheduler.shutdown();
      }
    } catch (final SchedulerException e) {
      e.printStackTrace();
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

La classe org.quartz.impl.calendar.HolidayCalendar permet de définir des jours à exclure en tenant compte de l'année.

La méthode addExcludedDate() permet d'exclure un jour : elle attend en paramètre une instance de type Date.

La méthode removeExcludedDate() permet de retirer un jour de la liste de ceux à exclure.

Exemple :

```
package fr.jmdoudoux.dej.quartz;

import java.io.IOException;
import java.util.Calendar;
import java.util.Date;
import java.util.List;

import org.quartz.CalendarIntervalScheduleBuilder;
import org.quartz.DateBuilder;
import org.quartz.JobBuilder;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.SchedulerFactory;
import org.quartz.Trigger;
import org.quartz.TriggerBuilder;
import org.quartz.impl.StdSchedulerFactory;
import org.quartz.impl.calendar.HolidayCalendar;
import org.quartz.spi.OperableTrigger;

public class TestQuartz {

  public static void main(final String[] args) {
    final SchedulerFactory factory = new StdSchedulerFactory();
    Scheduler scheduler = null;
    try {
      scheduler = factory.getScheduler();

      final HolidayCalendar cal = new HolidayCalendar();

      final Calendar gCal = Calendar.getInstance();

      gCal.set(2014, Calendar.JANUARY, 1);
      cal.addExcludedDate(gCal.getTime());
      gCal.set(2014, Calendar.APRIL, 21);
      cal.addExcludedDate(gCal.getTime());
      gCal.set(2014, Calendar.MAY, 1);
      cal.addExcludedDate(gCal.getTime());
      gCal.set(2014, Calendar.MAY, 8);
      cal.addExcludedDate(gCal.getTime());
      gCal.set(2014, Calendar.MAY, 29);
      cal.addExcludedDate(gCal.getTime());
      gCal.set(2014, Calendar.JUNE, 9);
      cal.addExcludedDate(gCal.getTime());
      gCal.set(2014, Calendar.JULY, 14);
      cal.addExcludedDate(gCal.getTime());
      gCal.set(2014, Calendar.AUGUST, 15);
      cal.addExcludedDate(gCal.getTime());
      gCal.set(2014, Calendar.NOVEMBER, 1);
      cal.addExcludedDate(gCal.getTime());
      gCal.set(2014, Calendar.NOVEMBER, 11);
      cal.addExcludedDate(gCal.getTime());
      gCal.set(2014, Calendar.DECEMBER, 25);
      cal.addExcludedDate(gCal.getTime());

      scheduler.addCalendar("Jour feries 2014", cal, true, true);

      final JobDetail jobDetail = JobBuilder.newJob(MonJob.class)
        .withIdentity("monJob").storeDurably().build();

      final Trigger trigger = TriggerBuilder
          .newTrigger()
          .withIdentity("monTrigger", "monGroupe")
          .modifiedByCalendar("Jour feries 2014")
          .startAt(DateBuilder.todayAt(14, 0, 0))
          .withSchedule(CalendarIntervalScheduleBuilder
            .calendarIntervalSchedule().withIntervalInDays(2))
          .build();

      scheduler.scheduleJob(jobDetail, trigger);
      scheduler.start();

      System.in.read();
      if (scheduler != null) {
        scheduler.shutdown();
      }
    } catch (final SchedulerException e) {
      e.printStackTrace();
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

Il est possible de définir sa propre implémentation de type org.quartz.Calendar qui doit être sérialisable.

113.2.6.6. La classe TriggerUtils

La classe TriggerUtils propose des utilitaires qui peuvent être pratiques.

Avant la version 2.0, elle propose :

- plusieurs surcharges de la méthode getDateOf() pour obtenir une instance de type Date
- de nombreuses méthodes getXXXDate() et getXXXDateBefore() pour obtenir une instance de type Date
- de nombreuses méthodes et leurs surcharges makeXXXTrigger() pour obtenir des instances de type Trigger
- deux surcharges de la méthode setTriggerIdentity()

A partir de la version 2.0, elle ne propose plus que trois méthodes qui permettent de faire des calculs relatifs au nombre de déclenchements d'un trigger.

- static Date computeEndTimeToAllowParticularNumberOfFirings(OperableTrigger trigg, Calendar cal, int numTimes) : renvoyer une instance de type Date encapsulant le déclenchement suivant les numTimes premiers déclenchements pour le trigger et le Calendar fournis
- static List<Date> computeFireTimes(OperableTrigger trigg, Calendar cal, int numTimes) : renvoyer une collection de type Date contenant les numTimes premiers déclenchements pour le trigger et le Calendar fournis
- static List<Date> computeFireTimesBetween(OperableTrigger trigg, Calendar cal, Date from, Date to) : renvoyer une collection de type Date contenant les déclenchements entre les deux dates fournies en paramètres pour le trigger et le Calendar fournis

Exemple :

```
package fr.jmdoudoux.dej.quartz;

import java.io.IOException;
import java.util.Date;
import java.util.List;

import org.quartz.CalendarIntervalScheduleBuilder;
import org.quartz.DateBuilder;
import org.quartz.JobBuilder;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.SchedulerFactory;
import org.quartz.Trigger;
import org.quartz.TriggerBuilder;
import org.quartz.TriggerUtils;
import org.quartz.impl.StdSchedulerFactory;
import org.quartz.spi.OperableTrigger;

public class TestQuartz {

  public static void main(final String[] args) {
    final SchedulerFactory factory = new StdSchedulerFactory();
    Scheduler scheduler = null;
    try {
      scheduler = factory.getScheduler();

      final JobDetail jobDetail = JobBuilder.newJob(MonJob.class)
        .withIdentity("monJob").storeDurably().build();

      final Trigger trigger = TriggerBuilder
          .newTrigger()
          .withIdentity("monTrigger", "monGroupe")
          .startAt(DateBuilder.tomorrowAt(12, 0, 0))
          .withSchedule(CalendarIntervalScheduleBuilder
            .calendarIntervalSchedule().withIntervalInDays(2))
          .build();

      scheduler.scheduleJob(jobDetail, trigger);
      scheduler.start();

      final List<Date> dates = TriggerUtils
        .computeFireTimes((OperableTrigger) trigger, null, 20);

      for (final Date date : dates) {
        System.out.println(date);
      }

      System.in.read();
      if (scheduler != null) {
        scheduler.shutdown();
      }
    } catch (final SchedulerException e) {
      e.printStackTrace();
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

113.2.7. La gestion des jobs et des triggers

L'interface Scheduler propose de nombreuses méthodes qui permettent de gérer les triggers et les jobs.

Méthodes	Rôle
void addCalendar(String calName, Calendar calendar, boolean replace, boolean updateTriggers)	Enregistrer le Calendar avec le nom fourni en paramètres
void addJob(JobDetail jobDetail, boolean replace)	Enregistrer le job dans le scheduler sans l'associer avec un trigger
boolean checkExists(JobKey jobKey)	Vérifier si un job dont l'identifiant est fourni en paramètre

	est déjà enregistré dans le scheduler
boolean checkExists(TriggerKey triggerKey)	Vérifier si un trigger dont l'identifiant est fourni en paramètre est déjà enregistré dans le scheduler
void clear()	Supprimer tous les éléments enregistrés dans le scheduler (Triggers, Jobs et Calendars)
boolean deleteCalendar(String calName)	Supprimer le Calendar dont le nom est fourni en paramètre
boolean deleteJob(JobKey jobKey)	Supprimer le job dont l'identifiant est fourni en paramètre. Les triggers associés sont aussi retirés
boolean deleteJobs(List<JobKey> jobKeys)	Supprimer les jobs dont les identifiants sont fournis en paramètre. Les triggers associés sont aussi retirés
Calendar getCalendar(String calName)	Obtenir l'instance de Calendar enregistré avec le nom fourni en paramètre
List<String> getCalendarNames()	Obtenir la liste des noms de Calendar enregistrés
SchedulerContext getContext()	Obtenir l'instance de type SchedulerContext
List<JobExecutionContext> getCurrentlyExecutingJobs()	Obtenir la liste des JobExecutionContext des jobs en cours d'exécution dans le scheduler
JobDetail getJobDetail(JobKey jobKey)	Obtenir le JobDetail du job dont l'identifiant est fourni en paramètre
List<String> getJobGroupNames()	Obtenir la liste de tous les groupes associés à au moins un job
Set<JobKey> getJobKeys(GroupMatcher<JobKey> matcher)	Obtenir les identifiants des jobs qui sont associés à des groupes répondant au GroupMatcher fourni en paramètre
ListenerManager getListenerManager()	Obtenir l'instance de type ListenerManager permettant de gérer les listeners
SchedulerMetaData getMetaData()	Obtenir l'instance de type SchedulerMetaData
Set<String> getPausedTriggerGroups()	Obtenir la liste des noms de groupes de triggers qui sont en pause
String getSchedulerInstanceId()	Renvoyer l'identifiant de l'instance
String getSchedulerName()	Renvoyer le nom du scheduler
Trigger getTrigger(TriggerKey triggerKey)	Obtenir le trigger dont l'identifiant est fourni en paramètre
List<String> getTriggerGroupNames()	Obtenir la liste de tous les groupes associés à au moins un trigger
Set<TriggerKey> getTriggerKeys(GroupMatcher<TriggerKey> matcher)	Obtenir les identifiants des triggers qui sont associés à des groupes répondant au GroupMatcher fourni en paramètre
List<? extends Trigger> getTriggersOfJob(JobKey jobKey)	Obtenir une liste des triggers qui sont associés au job dont l'identifiant est passé en paramètre
Trigger.TriggerState getTriggerState(TriggerKey triggerKey)	Obtenir l'état d'un trigger dont l'identifiant est fourni en paramètre
boolean interrupt(JobKey jobKey)	Demander l'interruption de l'exécution dans l'instance du scheduler du ou des jobs dont l'identifiant est fourni en paramètre. Les jobs concernés doivent implémenter l'interface InterruptableJob interface
boolean isInStandbyMode()	Déterminer si le scheduler est en mode standby
boolean isShutdown()	Déterminer si le scheduler a été arrêté
boolean isStarted()	Déterminer si le scheduler est démarré
void pauseAll()	Mettre en pause tous les triggers

void pauseJob(JobKey jobKey)	Mettre en pause le job dont l'identifiant est fourni en paramètre
void pauseJobs(GroupMatcher<JobKey> matcher)	Mettre en pause les jobs dont l'identifiant répond au GroupMatcher fourni en paramètre
void pauseTrigger(TriggerKey triggerKey)	Mettre en pause le trigger dont l'identifiant est fourni en paramètre
void pauseTriggers(GroupMatcher<TriggerKey> matcher)	Mettre en pause les triggers dont l'identifiant répond au GroupMatcher fourni en paramètre
Date rescheduleJob(TriggerKey triggerKey, Trigger newTrigger)	Supprimer le trigger dont l'identifiant est fourni en paramètre et enregistrer celui fourni en paramètre à sa place. Ils n'ont pas l'obligation d'avoir le même nom mais ils doivent être associé au même identifiant de job
void resumeAll()	Réactiver tous les triggers
void resumeJob(JobKey jobKey)	Réactiver le job dont l'identifiant est fourni en paramètre
void resumeJobs(GroupMatcher<JobKey> matcher)	Réactiver les jobs dont l'identifiant répond au GroupMatcher fourni en paramètre
void resumeTrigger(TriggerKey triggerKey)	Réactiver le trigger dont l'identifiant est fourni en paramètre
void resumeTriggers(GroupMatcher<TriggerKey> matcher)	Mettre en pause les triggers dont l'identifiant répond au GroupMatcher fourni en paramètre
Date scheduleJob(JobDetail jobDetail, Trigger trigger)	Enregistrer la planification du job avec le trigger fournis en paramètres
Date scheduleJob(Trigger trigger)	Enregistrer la planification du job associé au trigger fourni en paramètre
void scheduleJobs(Map<JobDetail,List<Trigger>> triggersAndJobs, boolean replace)	Planifier les jobs associés aux triggers fournis en paramètres
void setJobFactory(JobFactory factory)	Définir l'instance de la fabrique pour créer des instances de type Job
void shutdown()	Arrêter le scheduler
void shutdown(boolean waitForJobsToComplete)	Arrêter le scheduler avec une attente de la fin de l'exécution des jobs en cours
void standby()	Mettre le scheduler en mode standby : le déclenchement des triggers est suspendu
void start()	Démarrer le scheduler
void startDelayed(int seconds)	Démarrer le scheduler après avoir attendu le nombre de secondes fourni en paramètre
void triggerJob(JobKey jobKey)	Exécuter immédiatement le Job dont l'identifiant est fourni en paramètre
void triggerJob(JobKey jobKey, JobDataMap data)	Exécuter immédiatement le Job dont l'identifiant est fourni en paramètre avec les métadonnées fournies
boolean unscheduleJob(TriggerKey triggerKey)	Retirer le trigger dont l'identifiant est fourni en paramètre
boolean unscheduleJobs(List<TriggerKey> triggerKeys)	Retirer la liste de triggers fournie en paramètre

La planification d'un job

Il est nécessaire d'enregistrer la planification d'un job dans le scheduler pour permettre son déclenchement. Il faut utiliser une des surcharges de la méthode scheduleJob() ou la méthode scheduleJobs() du scheduler.

Exemple :

```
scheduler.scheduleJob(jobDetail, trigger);
```

L'enregistrement d'un job pour une utilisation ultérieure

Il faut invoquer la méthode addJob() du scheduler en lui passant en paramètre l'instance de type JobDetail et le booléen true.

Exemple :

```
final JobDetail jobDetail = JobBuilder
  .newJob(MonJob.class).withIdentity("monJob").storeDurably().build();
scheduler.addJob(jobDetail, true);
```

La modification d'un trigger ou d'un job enregistré

Il faut invoquer la méthode rescheduleJob() du scheduler en lui passant en paramètre l'identifiant du trigger à remplacer et l'instance du Trigger. Le nouveau trigger sera associé au même job même si un autre job lui est déjà associé.

Exemple :

```
final JobDetail jobDetail = JobBuilder
    .newJob(MonJob.class)
    .withIdentity("monJob")
    .storeDurably()
    .build();

final Trigger trigger = TriggerBuilder
    .newTrigger()
    .withIdentity(new TriggerKey("monTrigger", "monGroup"))
    .withSchedule(
        SimpleScheduleBuilder
            .simpleSchedule()
            .withIntervalInHours(1)
            .repeatForever())
    .startAt(DateBuilder.evenHourDate(new Date()))
    .build();

scheduler.scheduleJob(jobDetail, trigger);

final Trigger triggerNouveau = TriggerBuilder
    .newTrigger()
    .withIdentity(new TriggerKey("monTriggerNouveau", "monGroup"))
    .withSchedule(
        SimpleScheduleBuilder
            .simpleSchedule()
            .withIntervalInHours(2)
            .repeatForever())
    .startAt(DateBuilder.evenMinuteDate(new Date()))
    .build();

scheduler.rescheduleJob(trigger.getKey(), triggerNouveau);
```

La suppression d'un job ou de sa planification

Pour retirer la planification d'un job, il faut invoquer la méthode unscheduleJob() en lui passant en paramètre l'identifiant du trigger concerné. Pour retirer plusieurs jobs, il faut invoquer la méthode unscheduleJobs().

Exemple :

```
scheduler.unscheduleJob(trigger.getKey());
```

Pour retirer un job de la ou les planifications qui lui sont associées, il faut invoquer la méthode deleteJob() en lui passant en paramètre l'identifiant du job concerné.

Exemple :

```
scheduler.deleteJob(jobDetail.getKey());
```

La mise en pause d'un job ou d'un trigger

La méthode pauseJob() permet de suspendre la planification de l'exécution d'un job.

Exemple :

```
scheduler.pauseJob(jobDetail.getKey());
```

La méthode resumeJob() permet de rependre la planification de l'exécution d'un job.

Exemple :

```
scheduler.resumeJob(jobDetail.getKey());
```

L'obtention de la liste des jobs et des triggers

La méthode getJobGroupNames() permet d'obtenir la liste des groupes. Avant Quartz 2.0, la méthode getJobNames() qui attend en paramètre le nom d'un groupe permet d'obtenir les jobs qui appartiennent à ce groupe.

Exemple :

```
for (String groupName : scheduler.getJobGroupNames()) {
    for (String jobName : scheduler.getJobNames(groupName)) {
      System.out.println("jobName : " + jobName + ", group : " + groupName);
    }
}
```

A partir de Quartz 2.0, la méthode getJobKeys() qui attend en paramètre une instance de type GroupMatcher renvoie les jobs dont le nom du groupe répond au matcher.

Exemple :

```
for(String group: scheduler.getJobGroupNames()) {
    for(JobKey jobKey : scheduler.getJobKeys(GroupMatcher.groupEquals(group))) {
        System.out.println("job : " + jobKey);
    }
}
```

Pour obtenir la liste des prochains déclenchements de chaque job, il faut :

- itérer sur chaque job
- obtenir la liste des triggers associés au job
- invoquer la méthode getNextFireTime() du premier trigger s'il existe

Avant Quartz 2.0 :

Exemple :

```
for (String nomGroupe : scheduler.getJobGroupNames()) {

    for (String nomJob : scheduler.getJobNames(nomGroupe)) {
      Trigger[] triggers = scheduler.getTriggersOfJob(nomJob, nomGroupe);
      Date nextFireTime = null;
      if (triggers.length > 0) {
          nextFireTime = triggers[0].getNextFireTime();
      }
      System.out.println("Job : " + nomJob + " groupe : " + nomGroupe
          + ", prochain declenchement " + nextFireTime);
    }
}
```

A partir de Quartz 2.0 :

Exemple :

```
for (final String nomGroupe : scheduler.getJobGroupNames()) {

  for (final JobKey jobKey : scheduler.getJobKeys(GroupMatcher
      .jobGroupEquals(nomGroupe))) {
    Date nextFireTime = null;
    final List<Trigger> triggers = (List<Trigger>) scheduler
        .getTriggersOfJob(jobKey);
    if (!triggers.isEmpty()) {
      nextFireTime = triggers.get(0).getNextFireTime();
    }
    System.out.println("Job : " + jobKey.getName() + " groupe : " + jobKey.getGroup()
        + ", prochain declenchement " + nextFireTime);
  }
}
```

A partir de Quartz 2.0, il faut utiliser la méthode getTriggerKeys() qui attend en paramètre une instance de type GroupMatcher et renvoie les triggers dont le nom du groupe répond au matcher.

Exemple :

```
for (final String group : scheduler.getTriggerGroupNames()) {
  for (final TriggerKey triggerKey : scheduler
      .getTriggerKeys(GroupMatcher.triggerGroupEquals(group))) {
    System.out.println("Trigger : " + triggerKey.getName() + ", groupe "
        + triggerKey.getGroup());
  }
}
```

L'obtention des jobs associés à un trigger

La méthode getTriggerOfJob() permet d'obtenir la liste des triggers qui sont associés au job dont l'identifiant est fourni en paramètre.

Exemple :

```
System.out.println("Trigger for job " + jobDetail.getKey());
final List<? extends Trigger> jobTriggers = scheduler
     .getTriggersOfJob(jobDetail.getKey());
for (final Trigger jobTrigger : jobTriggers) {
  System.out.println("Trigger : " + jobTrigger.getKey());
}
```

L'arrêt d'un job en cours d'exécution

Pour permettre l'arrêt d'un job, sa classe doit implémenter l'interface InterruptableJob. Elle hérite de l'interface Job et ne définit qu'une seule méthode : void interrupt().

Cette méthode sera invoquée par le scheduler pour tenter d'arrêter l'exécution du job. En aucun cas elle ne permet d'arrêter directement l'exécution mais elle permet de demander son arrêt. Généralement son implémentation consiste à changer la valeur d'un boolean qui doit être périodiquement testé dans les traitements du job pour les arrêter. La logique d'interruption doit donc être codée dans les traitements du job eux-mêmes.

Pour interrompre un job en cours d'exécution, il faut :

- Avant Quartz 2.0 : invoquer la méthode boolean interrupt(JobKey) qui attend en paramètre l'identifiant du job à arrêter
- A partir de Quartz 2.0 : invoquer la méthode boolean interrupt(String, String) qui attend en paramètre le nom et le groupe du job à arrêter

113.2.8. Les listeners

Quartz propose la possibilité de réagir à des événements en utilisant des Listeners.

Quartz propose trois types de Listeners :

- JobListener : pour être informé d'événements liés à l'exécution d'un job
- TriggerListener : pour être informé d'événements liés au déclenchement d'un trigger
- SchedulerListener : pour être informé d'événements liés au moteur de planification

L'implémentation d'un listener peut se faire de deux manières :

- implémenter l'interface
- hériter de la classe xxxSupport où xxx est le nom du listener et redéfinir la ou les méthodes utiles

Un listener doit avoir un nom retourné par la méthode getName()

Pour utiliser un listener, il faut :

- créer une instance du listener
- l'enregistrer dans le scheduler en utilisant son ListenerManager

Avant la version 2.0 de Quartz, l'enregistrement d'un listener dans le scheduler se fait en invoquant une de ses méthodes addXXXListener(). A partir de la version 2.0, l'enregistrement d'un listener dans le scheduler se fait en utilisant le ListenerManager qui lui est associé.

L'interface ListenerManager définit des méthodes pour gérer les listeners associés à un scheduler :

Méthode	Rôle
void addJobListener(JobListener jobListener, List<Matcher<JobKey>> matchers) void addJobListener(JobListener jobListener, Matcher<JobKey>... matchers)	Ajouter un JobListener au Scheduler et l'enregistrer pour être invoqué sur les événements des Job dont la clé correspond à au moins un des Matcher fournis
boolean addJobListenerMatcher(String listenerName, Matcher<JobKey> matcher)	Ajouter un Matcher au listener dont le nom est fourni en paramètre
void addSchedulerListener(SchedulerListener schedulerListener)	Ajouter un SchedulerListener
void addTriggerListener(TriggerListener triggerListener, List<Matcher<TriggerKey>> matchers) void addTriggerListener(TriggerListener triggerListener, Matcher<TriggerKey>... matchers)	Ajouter un TriggerListener et l'enregistrer dans le scheduler pour être invoqué sur les événements des triggers dont la clé correspond à au moins un des Matcher fournis
boolean addTriggerListenerMatcher(String listenerName, Matcher<TriggerKey> matcher)	Ajouter un Matcher au listener dont le nom est fourni en paramètre
JobListener getJobListener(String name)	Obtenir un JobListener à partir de son nom
List<Matcher<JobKey>> getJobListenerMatchers(String listenerName)	Obtenir les Matcher associés au listener dont le nom est fourni en paramètre

List<JobListener> getJobListeners()	Obtenir tous les JobListener enregistrés
List<SchedulerListener> getSchedulerListeners()	Obtenir tous les SchedulerListener
TriggerListener getTriggerListener(String name)	Obtenir un TriggerListener à partir de son nom
List<Matcher<TriggerKey>> getTriggerListenerMatchers(String listenerName)	Obtenir les Matcher associés au listener dont le nom est fourni en paramètre
List<TriggerListener> getTriggerListeners()	Obtenir tous les TriggerListener enregistrés
boolean removeJobListener(String name)	Retirer le JobListener dont le nom est fourni en paramètre
boolean removeJobListenerMatcher(String listenerName, Matcher<JobKey> matcher)	Retirer le Matcher associé au listener dont le nom est fourni en paramètre
boolean removeSchedulerListener(SchedulerListener schedulerListener)	Retirer le schedulerListener fourni en paramètre du scheduler
boolean removeTriggerListener(String name)	Retirer le TriggerListener dont le nom est fourni en paramètre
boolean removeTriggerListenerMatcher(String listenerName, Matcher<TriggerKey> matcher)	Retirer le Matcher associé au listener dont le nom est fourni en paramètre
boolean setJobListenerMatchers(String listenerName, List<Matcher<JobKey>> matchers)	Définir les Matchers associés au listener dont le nom est fourni en paramètre
boolean setTriggerListenerMatchers(String listenerName, List<Matcher<TriggerKey>> matchers)	Définir les Matchers associés au listener dont le nom est fourni en paramètre

Lors de l'enregistrement d'un listener, il est possible de filtrer les éléments (Job ou Trigger) qui seront concernés par les événements en utilisant un objet de type org.quartz.Matcher.

Quartz fournit en standard plusieurs implémentations de type Matcher pour définir le filtre : AndMatcher, EverythingMatcher, GroupMatcher, KeyMatcher, NameMatcher, NotMatcher, OrMatcher et StringMatcher.

Les listeners ne sont pas enregistrés dans le JobStore : il est donc nécessaire d'enregistrer les listeners à chaque démarrage du scheduler. Il est cependant possible de configurer des listeners globaux qui seront instanciés et enregistrés par la fabrique au démarrage du scheduler.

113.2.8.1. JobListener

Un JobListener permet de réagir à des événements liés à l'exécution d'un job.

L'interface JobListener définit plusieurs méthodes :

Méthode	Rôle
String getName()	Obtenir le nom du listener
void jobExecutionVetoed(JobExecutionContext context)	Invoquée par le scheduler lorsqu'un JobDetail est sur le point d'être exécuté (suite au déclenchement d'un trigger lié au JobDetail) mais qu'un TriggerListener a mis un véto sur son exécution
void jobToBeExecuted(JobExecutionContext context)	Invoquée par le scheduler lorsqu'un JobDetail est sur le point d'être exécuté (suite au déclenchement d'un trigger lié au JobDetail)
void jobWasExecuted(JobExecutionContext context, JobExecutionException jobException)	Invoquée par le scheduler lorsqu'un JobDetail a été exécuté

Exemple :

```
package fr.jmdoudoux.dej.quartz;

import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;
import org.quartz.listeners.JobListenerSupport;

public class MonJobListener extends JobListenerSupport {

  @Override
  public String getName() {
    return "MonJobListener";
  }

  @Override
  public void jobToBeExecuted(final JobExecutionContext context) {
    final String jobName = context.getJobDetail().getKey().toString();
    System.out.println("Le job : " + jobName + " va demarrer");
  }

  @Override
  public void jobWasExecuted(final JobExecutionContext context,
      final JobExecutionException jobException) {

    final String jobName = context.getJobDetail().getKey().toString();
    System.out.println("Le job : " + jobName + " est termine");

    if (jobException != null) {
      System.out.println("Exception levee par le job " + jobName + " : "
          + jobException.getMessage());
    }
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.quartz;

import java.io.IOException;

import org.quartz.JobBuilder;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.SchedulerFactory;
import org.quartz.SimpleScheduleBuilder;
import org.quartz.Trigger;
import org.quartz.TriggerBuilder;
import org.quartz.TriggerKey;
import org.quartz.impl.StdSchedulerFactory;
import org.quartz.impl.matchers.KeyMatcher;

public class TestQuartz {

  public static void main(final String[] args) {
    final SchedulerFactory factory = new StdSchedulerFactory();
    Scheduler scheduler = null;
    try {
      scheduler = factory.getScheduler();

      final JobDetail jobDetail = JobBuilder
          .newJob(MonJob.class)
          .withIdentity("monJob")
          .storeDurably()
          .build();

      final Trigger trigger = TriggerBuilder
          .newTrigger()
          .withIdentity(new TriggerKey("monTrigger", "monGroup"))
          .withSchedule(
              SimpleScheduleBuilder
                  .simpleSchedule()
                  .withIntervalInHours(1)
                  .repeatForever())
          .startNow()
          .build();

      scheduler.getListenerManager().addJobListener(new MonJobListener(),
          KeyMatcher.keyEquals(jobDetail.getKey()));

      scheduler.scheduleJob(jobDetail, trigger);
      scheduler.start();

      System.in.read();
      if (scheduler != null) {
        scheduler.shutdown();
      }
    } catch (final SchedulerException e) {
      e.printStackTrace();
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

113.2.8.2. TriggerListener

Un TriggerListener permet de réagir à des événements lies à l'exécution d'un trigger.

L'interface TriggerListener définit plusieurs méthodes :

Méthode	Rôle
String getName()	Obtenir le nom du listener
void triggerComplete(Trigger trigger, JobExecutionContext context, Trigger.CompletedExecutionInstruction triggerInstructionCode)	Invoquée par le scheduler lorsque les traitements du déclenchement d'un trigger sont terminés (le job associé a été exécuté)
void triggerFired(Trigger trigger, JobExecutionContext context)	Invoquée par le scheduler lorsqu'un trigger est déclenché et que son Job associé est prêt à être exécuté
void triggerMisfired(Trigger trigger)	Invoquée par le scheduler lorsque le déclenchement d'un trigger est raté
boolean vetoJobExecution(Trigger trigger, JobExecutionContext context)	Invoquée par le scheduler lorsqu'un trigger est déclenché et que son Job associé est prêt à être exécuté

La méthode vetoJobExecution() permet de mettre un véto sur l'exécution du job lié au trigger simplement en renvoyant true. Elle est invoquée par le scheduler après la méthode triggerFired().

Exemple :

```
package fr.jmdoudoux.dej.quartz;

import org.quartz.JobExecutionContext;
import org.quartz.Trigger;
import org.quartz.listeners.TriggerListenerSupport;

public class MonTriggerListener extends TriggerListenerSupport {

  private final String name;

  public MonTriggerListener(final String name) {
    this.name = name;
  }

  @Override
  public String getName() {
    return this.name;
  }

  @Override
  public void triggerFired(final Trigger trigger,
      final JobExecutionContext context) {
    final String triggerName = context.getTrigger().getKey().toString();
    System.out.println("Le trigger : " + triggerName + " est declenche");
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.quartz;

import java.io.IOException;

import org.quartz.JobBuilder;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.SchedulerFactory;
import org.quartz.SimpleScheduleBuilder;
import org.quartz.Trigger;
import org.quartz.TriggerBuilder;
import org.quartz.TriggerKey;
import org.quartz.impl.StdSchedulerFactory;
import org.quartz.impl.matchers.KeyMatcher;

public class TestQuartz {

  public static void main(final String[] args) {
    final SchedulerFactory factory = new StdSchedulerFactory();
    Scheduler scheduler = null;
    try {
      scheduler = factory.getScheduler();

      final JobDetail jobDetail = JobBuilder
          .newJob(MonJob.class)
          .withIdentity("monJob")
          .storeDurably()
          .build();

      final Trigger trigger = TriggerBuilder
          .newTrigger()
          .withIdentity(new TriggerKey("monTrigger", "monGroup"))
          .withSchedule(
              SimpleScheduleBuilder
                  .simpleSchedule()
                  .withIntervalInHours(1)
                  .repeatForever())
          .startNow()
          .build();

      scheduler.getListenerManager().addTriggerListener(
          new MonTriggerListener("monTriggerListener"),
          KeyMatcher.keyEquals(trigger.getKey()));

      scheduler.scheduleJob(jobDetail, trigger);
      scheduler.start();

      System.in.read();
      if (scheduler != null) {
        scheduler.shutdown();
      }
    } catch (final SchedulerException e) {
      e.printStackTrace();
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

113.2.8.3. SchedulerListener

Un SchedulerListener permet de réagir à des événements liés au scheduler lui-même notamment ceux concernant son cycle de vie : l'ajout/suppression de jobs/triggers, la mise en pause ou la reprise de jobs/triggers et la levée d'erreurs.

L'interface SchedulerListener définit plusieurs méthodes :

Méthode	Rôle
void jobAdded(JobDetail jobDetail)	Invoquée par le scheduler lorsqu'un JobDetail est ajouté
void jobDeleted(JobKey jobKey)	Invoquée par le scheduler lorsqu'un job est supprimé
void jobPaused(JobKey jobKey)	Invoquée par le scheduler lorsqu'un job est mis en pause
void jobResumed(JobKey jobKey)	Invoquée par le scheduler lorsqu'un job est redémarré
void jobScheduled(Trigger trigger)	Invoquée par le scheduler lorsqu'un job est planifié
void jobsPaused(String jobGroup)	Invoquée par le scheduler lorsqu'un groupe de jobs est mis en pause
void jobsResumed(String jobGroup)	Invoquée par le scheduler lorsqu'un group de jobs est redémarré
void jobUnscheduled(TriggerKey triggerKey)	Invoquée par le scheduler lorsque la planification d'un job est supprimée
void schedulerError(String msg, SchedulerException cause)	Invoquée par le scheduler lorsqu'une erreur grave survient dans ses propres traitements (échec de mises à jour du JobStore, impossible d'instancier un job lors du déclenchement d'un trigger, ...)
void schedulerInStandbyMode()	Invoquée par le scheduler lorsque son état passe à standby
void schedulerShutdown()	Invoquée par le scheduler lorsqu'il s'arrête
void schedulerShuttingdown()	Invoquée par le scheduler lorsqu'il débute ses traitements pour son arrêt
void schedulerStarted()	Invoquée par le scheduler lorsqu'il démarre
void schedulerStarting()	Invoquée par le scheduler lorsqu'il débute ses traitements pour son démarrage
void schedulingDataCleared()	Invoquée par le scheduler lorsque tous les jobs, triggers et calendars ont été supprimés
void triggerFinalized(Trigger trigger)	Invoquée par le scheduler lorsque tous les déclenchements du trigger ont été réalisés
void triggerPaused(TriggerKey triggerKey)	Invoquée par le scheduler lorsqu'un trigger est mis en pause
void triggerResumed(TriggerKey triggerKey)	Invoquée par le scheduler lorsqu'un trigger est redémarré
void triggersPaused(String triggerGroup)	Invoquée par le scheduler lorsqu'un groupe de triggers est mis en pause
void triggersResumed(String triggerGroup)	Invoquée par le scheduler lorsqu'un groupe de triggers est redémarré

Exemple :

```
package fr.jmdoudoux.dej.quartz;

import org.quartz.Trigger;
import org.quartz.listeners.SchedulerListenerSupport;

public class MonSchedulerListener extends SchedulerListenerSupport {

  @Override
  public void schedulerStarted() {
    System.out.println("Demarrage scheduler");
  }

  @Override
  public void schedulerShutdown() {
    System.out.println("Arret scheduler");
  }

  @Override
  public void jobScheduled(final Trigger trigger) {
    System.out.println("Planification du job " + trigger.getJobKey()
        + " avec le trigger " + trigger.getKey());
  }
}
```

Exemple :

```
package fr.jmdoudoux.dej.quartz;

import java.io.IOException;
import java.util.Date;

import org.quartz.DateBuilder;
import org.quartz.JobBuilder;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.SchedulerFactory;
import org.quartz.SimpleScheduleBuilder;
import org.quartz.Trigger;
import org.quartz.TriggerBuilder;
import org.quartz.TriggerKey;
import org.quartz.impl.StdSchedulerFactory;

public class TestQuartz {

  public static void main(final String[] args) {
    final SchedulerFactory factory = new StdSchedulerFactory();
    Scheduler scheduler = null;
    try {
      scheduler = factory.getScheduler();

      final JobDetail jobDetail = JobBuilder
          .newJob(MonJob.class)
          .withIdentity("monJob")
          .storeDurably()
          .build();

      final Trigger trigger = TriggerBuilder
          .newTrigger()
          .withIdentity(new TriggerKey("monTrigger", "monGroup"))
          .withSchedule(
              SimpleScheduleBuilder
                  .simpleSchedule()
                  .withIntervalInHours(1)
                  .repeatForever())
          .startAt(DateBuilder.evenMinuteDate(new Date()))
          .build();

      scheduler.getListenerManager().addSchedulerListener(
          new MonSchedulerListener());

      scheduler.scheduleJob(jobDetail, trigger);
      scheduler.start();

      System.in.read();
      if (scheduler != null) {
        scheduler.shutdown();
      }
    } catch (final SchedulerException e) {
      e.printStackTrace();
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

113.2.9. Les JobStores

Un JobStore a la responsabilité de stocker les éléments utilisés par le scheduler (Jobs, Triggers, Calendars, ...).

Le JobStore utilisé par le Scheduler est défini et configuré dans les propriétés fournies à la fabrique de type SchedulerFactory.

Quartz fournit plusieurs JobStore en standard :

- RAMJobStore : il stocke les informations en mémoire
- JDBCJobStore : il stocke les informations dans une base de données relationnelle
- TerracottaJobStore : il stocke les informations dans un serveur Terracotta

113.2.9.1. Le RAMJobStore

C'est le plus simple et le plus performant des JobStore mais il n'est pas persistant. Toutes les informations sont donc perdues lors de l'arrêt normal ou non de la JVM dans laquelle le scheduler s'exécute. C'est le JobStore utilisé par défaut.

Pour configurer Quartz à utiliser le RAMJobStore, il faut le préciser dans le fichier de configuration :

Exemple :

```
org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore
```

113.2.9.2. Le JDBCJobStore

Un JDBCJobStore stocke les informations dans une base de données relationnelles. Il est nécessaire de créer les différentes tables requises en utilisant les scripts SQL contenus dans le sous-répertoire docs/dbTables de la distribution Quartz.

Le préfixe du nom des tables est configurable : cela peut être pratique pour stocker la configuration de plusieurs schedulers dans la même base de données.

Deux classes d'implémentation sont proposées par Quartz selon le type de transactions à utiliser :

- JobStoreTX pour laisser Quartz gérer lui-même ses transactions
- JobStoreCMT pour s'intégrer dans une transaction distribuée

Le paramétrage du JDBCJobStore se fait dans le fichier de configuration de Quartz notamment la définition de la Datasource qui peut être créée par Quartz ou obtenue d'un serveur d'applications par JNDI.

113.2.10. La configuration

Quartz propose un ensemble de fonctionnalités configurables dans un fichier de type properties :

- identification du scheduler
- pool de threads
- persistance des jobs et triggers
- utilisation de Quartz en mode client/serveur
- les listeners globaux
- les plugins

Exemple :

```
org.quartz.scheduler.instanceName = MonScheduler
org.quartz.threadPool.threadCount = 3
org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore
```

La configuration de Quartz se définit dans un fichier properties qui sera utilisé par la classe StdSchedulerFactory pour créer une instance de type Scheduler selon la configuration.

Par défaut, elle recherche un fichier nommé quart.properties dans le répertoire courant. Si le fichier n'est pas trouvé, alors c'est le fichier quartz.properties contenu dans le package org.quartz qui est chargé.

Pour préciser un autre fichier de configuration à utiliser, il est possible de :

- le passer comme valeur à la propriété org.quartz.properties de la JVM.
- le passer en paramètre d'une des surcharges de la méthode initialize() de la classe StdSchedulerFactory. La méthode initialize() doit être invoquée en la méthode getScheduler()

La configuration de certaines instances se fait en les identifiant grâce à un nom unique contenu dans leur clé.

Une propriété peut faire référence à la valeur d'une autre propriété en préfixant son nom avec $@. Par exemple, "$@org.quartz.scheduler.instanceName" sera remplacé par le nom de l'instance du Scheduler.

113.2.10.1. La configuration globale

Plusieurs propriétés concernent la configuration globale du scheduler, notamment :

Nom de la propriété	Type	Description
org.quartz.scheduler.instanceName	String	Définir le nom logique du scheduler. Dans un cluster, tous les schedulers doivent avoir le même nom logique. Valeur par défaut : «QuartzScheduler»
org.quartz.scheduler.instanceId	String	Identifiant de l'instance du scheduler. Doit être unique pour chaque scheduler d'un même cluster. La valeur doit être AUTO pour laisser Quartz donner un identifiant ou SYS_PROP pour que Quartz récupère la valeur de la propriété org.quartz.scheduler.instanceId de la JVM Valeur par défaut : «NON_CLUSTERED»
org.quartz.scheduler .instanceIdGenerator.class	String	N'est utilisé que si la propriété org.quartz.scheduler.instanceId vaut AUTO. Plusieurs classes sont proposées en standard par Quartz : • org.quartz.simpl.SimpleInstanceIdGenerator qui génère une valeur à partir du hostname et du timestamp courant • org.quartz.simpl.SystemPropertyInstanceIdGenerator qui obtient l'identifiant à partir de la propriété org.quartz.scheduler.instanceId • org.quartz.simpl.HostnameInstanceIdGenerator qui obtient l'identifiant à partir du hostname Valeur par défaut : org.quartz.simpl.SimpleInstanceIdGenerator
org.quartz.scheduler.threadName	String	Définir le nom des threads Valeur par défaut : instanceName + '_QuartzSchedulerThread'
org.quartz.scheduler .makeSchedulerThreadDaemon	boolean	Préciser si le thread principal du scheduler doit être de type démon (true) ou non (false)

		Valeur par défaut : false
org.quartz.scheduler .dbFailureRetryInterval	long	Définir le nombre de millisecondes que le scheduler va attendre pour se reconnecter au système sous-jacent du JobStore (par exemple une base de données) après une perte de connexion. Valeur par défaut : 15000
org.quartz.scheduler.jobFactory.class	String	Nom pleinement qualifié de la classe de type JobFactory utilisée par Quartz pour créer des instances de type Job. Quartz fournit plusieurs classes d'implémentation possibles : • org.quartz.simpl.PropertySettingJobFactory qui crée une nouvelle instance en invoquant newInstance() • org.quartz.simpl.PropertySettingJobFactory qui crée une instance par introspection en récupérant les informations dans le contexte du scheduler et les JobDataMaps du Job et du Trigger Valeur par défaut : org.quartz.simpl.PropertySettingJobFactory
org.quartz.context.key.SOME_KEY	String	Définir une paire clé/valeur qui sera ajoutée dans le contexte du scheduler. Il faut remplacer SOME_KEY par le nom de la clé Exemple : org.quartz.context.key.MaCle = maValeur est équivalent à scheduler.getContext().put("MaCle", "maValeur")
org.quartz.scheduler .userTransactionURL	String	Préciser l'URL JNDI du gestionnaire de transactions distribuées du serveur d'applications. Valeur par défaut : 'java:comp/UserTransaction'
Org.quartz.scheduler .wrapJobExecutionInUserTransaction	boolean	Préciser si Quartz doit démarrer une nouvelle UserTransaction avant d'exécuter un job. La transaction est validée lorsque le job est terminé. Valeur par défaut : false
org.quartz.scheduler.skipUpdateCheck	boolean	Préciser si Quartz doit vérifier sur Internet s'il existe une version plus récente. La propriété org.terracotta.quartz.skipUpdateCheck de la JVM permet de préciser si la vérification doit être activée. Il est préférable de désactiver cette fonctionnalité en production. Valeur par défaut : false

113.2.10.2. La configuration de listeners globaux

La fabrique StdChedulerFactory peut lire, créer et enregistrer des listeners dans le scheduler au moment de sa création. Ces listeners seront globaux donc concerneront tous les événements de l'instance de leur type (job et trigger).

Chaque listener doit être défini et identifié avec un nom unique contenu dans les clés le concernant.

Pour définir un listener, il faut :

- utiliser une clé de la forme org.quartz.xxxListener.NAME.class ayant pour valeur le nom pleinement qualifié du Listener
- une clé de la forme org.quartz.xxxListener.NAME.yyyyyy pour assigner une valeur à la propriété yyyyy du listener

Dans la forme des clés, il faut remplacer :

- xxx par le type de listener : job ou trigger
- NAME par le nom unique de l'instance
- yyyyyy par le nom de la propriété correspondante

Exemple de configuration d'un JobListener :

```
org.quartz.jobListener.MonJobListener.class = fr.jmdoudoux.dej.quartz.MonJobListener
org.quartz.jobListener.MonJobListener.min = 100
org.quartz.jobListener.MonJobListener.max = 1000
```

113.2.10.3. La configuration des exécutions concurrentes des jobs

Quartz utilise un pool de threads pour permettre d'exécuter des jobs en parallèle.

Lorsqu'un trigger déclenche l'exécution d'un job, le scheduler obtient un thread libre de son pool et lui fait exécuter les traitements du job.

La taille du pool de threads doit être configurée pour permettre l'exécution en simultanée de tous les jobs requis.

Cette configuration se fait grâce à plusieurs propriétés :

Nom	Description
org.quartz.threadPool.class	Nom pleinement qualifié de l'implémentation du pool de threads à utiliser. Valeur par défaut : null
org.quartz.threadPool.threadCount	Valeur entière positive qui représente le nombre de threads dans le pool Valeur par défaut : -1 qui signifie pas de pool de threads
org.quartz.threadPool.threadPriority	Entier compris entre Thread.MIN_PRIORITY(1) et Thread.MAX_PRIORITY(10) qui précise la priorité des threads (optionnel) Valeur par défaut : Thread.NORM_PRIORITY (5)

Plusieurs propriétés sont spécifiques à la classe SimpleThreadPool qui est l'implémentation du pool de threads fournie par défaut par Quartz.

Nom	Description
org.quartz.threadPool.makeThreadsDaemons	Les threads du pool sont des démons (optionnel) Valeur par défaut : false
org.quartz.threadPool.threadsInheritGroupOfInitializingThread	(optionnel) Valeur par défaut : true
org.quartz.threadPool.threadsInheritContextClassLoaderOfInitializingThread	(optionnel) Valeur par défaut : false
org.quartz.threadPool.threadNamePrefix	Préfixe utilisé dans le nom des threads du pool (optionnel) Valeur par défaut : [Scheduler Name]_Worker

113.2.10.4. Le stockage des entités de planification

Les entités utilisées par le scheduler pour réaliser la planification sont stockées dans un JobStore. Quartz propose plusieurs implémentations d'un JobStore, chacune possédant ses propres propriétés pour sa configuration.

Un RAMJobStore stocke les informations de planification du scheduler en mémoire. Pour utiliser un JobStore de type RAMJobStore, il faut préciser le nom pleinement qualifié de la classe comme valeur pour la propriété org.quartz.jobstore.class.

Exemple :

```
org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore
```

Un RAMJobStore peut être configuré grâce à une propriété :

Nom de la propriété	Type	Description
org.quartz.jobStore.misfireThreshold	int	Définir un nombre de millisecondes de tolérance avant que le déclenchement d'un trigger ne soit considéré comme raté Valeur par défaut : 60000

Un JDBCStore permet de stocker les informations des entités de planification (jobs, triggers, calendars, ...) dans une base de données relationnelle.

Remarque : les scripts de création des tables Quartz se trouvent dans le répertoire docs/dbTables de l'archive de la distribution.

Deux implémentations sont utilisables selon le comportement transactionnel souhaité :

- JobStoreTX : le JobStore assure lui-même la gestion des transactions
- JobStoreCMT : le JobStore utilise un gestionnaire de transactions distribuées

Un JobStoreTX est à mettre en oeuvre si aucune transaction distribuée n'est utilisée.

Pour utiliser un JobStore de type JobStoreTX, il faut préciser le nom pleinement qualifié de la classe comme valeur pour la propriété org.quartz.jobstore.class

Exemple :

```
org.quartz.jobStore.class = org.quartz.impl.jdbcjobstore.JobStoreTX
```

Un JobStoreTX peut être configuré grâce à plusieurs propriétés, notamment :

Nom de la propriété	Type	Description
org.quartz.jobStore.driverDelegateClass	String	Définir le nom pleinement qualifié de la classe qui va assurer la communication avec une base de données précise. (Obligatoire) Quartz propose une implémentation générique et plusieurs implémentations pour le support des principales bases de données lorsque celles-ci requièrent des opérations spécifiques : org.quartz.impl.jdbcjobstore.StdJDBCDelegate : support générique pour un pilote JDBC org.quartz.impl.jdbcjobstore.MSSQLDelegate pour SQL Server

		org.quartz.impl.jdbcjobstore.PostgreSQLDelegate pour PostgreSQL org.quartz.impl.jdbcjobstore.oracle.OracleDelegate pour Oracle org.quartz.impl.jdbcjobstore.DB2v6Delegate, org.quartz.impl.jdbcjobstore.DB2v7Delegate, org.quartz.impl.jdbcjobstore.DB2v8Delegate pour DB2 org.quartz.impl.jdbcjobstore.HSQLDBDelegate pour HSQLDB org.quartz.impl.jdbcjobstore.SybaseDelegate pour Sybase org.quartz.impl.jdbcjobstore.CloudscapeDelegate pour Cloudscape et Derby Valeur par défaut : null
org.quartz.jobStore.dataSource	String	Préciser le nom de la datasource : ce nom doit être défini dans la configuration (Obligatoire) Valeur par défaut : null
org.quartz.jobStore.tablePrefix	String	Définir le préfixe des tables dans la base de données Valeur par défaut : "QRTZ_"
org.quartz.jobStore.useProperties	boolean	Indiquer au JDBCJobStore que toutes les valeurs des JobDataMaps sont des String ce qui évite d'avoir à les sérialiser et stocker ces valeurs dans un champ de type blob Valeur par défaut : false
org.quartz.jobStore.misfireThreshold	int	Définir un nombre de millisecondes de tolérance avant que le déclenchement d'un trigger ne soit considéré comme raté Valeur par défaut : 60000
org.quartz.jobStore.isClustered	boolean	Préciser si la base de données est utilisée par plusieurs instances de Quartz. Valeur par défaut : false
org.quartz.jobStore.dontSetAutoCommitFalse	boolean	Préciser de ne pas invoquer la méthode setAutoCommit(false) sur les connexions à la DataDource. Valeur par défaut : false
org.quartz.jobStore.selectWithLockSQL	String	Définir la requête SQL qui est utilisée pour poser un verrou dans la table xxxLOCKS {0} est remplacé à l'exécution par le préfixe des tables {1} est remplacé à l'exécution par le nom du scheduler Valeur par défaut : "SELECT * FROM {0}LOCKS WHERE SCHED_NAME = {1} AND LOCK_NAME = ? FOR UPDATE"
org.quartz.jobStore.txIsolationLevelSerializable	boolean	Demander d'utiliser le niveau d'isolation Connection.TRANSACTION_SERIALIZABLE Valeur par défaut : false
org.quartz.jobStore.driverDelegateInitString	String	Chaîne de caractères permettant de préciser des paires clé=valeur qui seront utilisées par le DriverDelegate. Les paires son séparées par un caractère |

		Exemple : "propriete1=valeur1|propriete2=valeur2" Valeur par défaut : null

Exemple :

```
org.quartz.jobStore.class = org.quartz.impl.jdbcjobstore.JobStoreTX
org.quartz.jobStore.driverDelegateClass =  org.quartz.impl.jdbcjobstore.StdJDBCDelegate
org.quartz.jobStore.dataSource = QUARTZ_DS
org.quartz.jobStore.tablePrefix = QUARTZ_
```

Un JobStoreCMT est à mettre en oeuvre si les opérations du JobStore doivent être incluses dans une transaction distribuée. Le gestionnaire de transaction doit être démarré avant le scheduler.

Un JobStoreCMT a besoin de deux datasources :

- une qui soit incluse dans la transaction JTA
- une qui ne le soit pas

Pour utiliser un JobStore de type JobStoreCMT, il faut préciser le nom pleinement qualifié de la classe comme valeur pour la propriété org.quartz.jobstore.class

Exemple :

```
org.quartz.jobStore.class = org.quartz.impl.jdbcjobstore.JobStoreCMT
```

Un JobStoreCMT peut être configuré grâce à plusieurs propriétés :

Nom de la propriété	Type	Description
org.quartz.jobStore.driverDelegateClass	String	Identique à la propriété correspondante du JobStoreTX (Obligatoire) Valeur par défaut : null
org.quartz.jobStore.dataSource	String	Préciser le nom de la datasource dont les connexions doivent être compatibles JTA : ce nom doit être défini dans la configuration (Obligatoire) Valeur par défaut : null
org.quartz.jobStore.nonManagedTXDataSource	String	Préciser le nom de la datasource dont les connexions ne doivent pas être compatibles JTA : ce nom doit être défini dans la configuration (Obligatoire) Valeur par défaut :
org.quartz.jobStore.tablePrefix	String	Identique à la propriété correspondante du JobStoreTX Valeur par défaut : "QRTZ_"
org.quartz.jobStore.useProperties	boolean	Identique à la propriété correspondante du JobStoreTX Valeur par défaut : false
org.quartz.jobStore.misfireThreshold	int	Identique à la propriété correspondante du JobStoreTX Valeur par défaut : 60000
org.quartz.jobStore.isClustered	boolean	Identique à la propriété correspondante du JobStoreTX Valeur par défaut : false
org.quartz.jobStore.dontSetAutoCommitFalse	boolean	Identique à la propriété correspondante du JobStoreTX

		Valeur par défaut : false
org.quartz.jobStore.selectWithLockSQL	String	Identique à la propriété correspondante du JobStoreTX Valeur par défaut : "SELECT * FROM {0}LOCKS WHERE SCHED_NAME = {1} AND LOCK_NAME = ? FOR UPDATE"
org.quartz.jobStore.txIsolationLevelSerializable	boolean	Identique à la propriété correspondante du JobStoreTX Valeur par défaut : false
org.quartz.jobStore.txIsolationLevelReadCommitted	boolean	Identique à la propriété correspondante du JobStoreTX Valeur par défaut : false
org.quartz.jobStore.driverDelegateInitString	String	Identique à la propriété correspondante du JobStoreTX Valeur par défaut : null

La configuration d'une datasource utilisée par le JDBCJobStore peut se faire de trois manières :

- définir chacune des propriétés dans le fichier de configuration pour permettre à Quartz de créer sa propre instance
- préciser le nom JNDI de la Datasource définie dans le serveur d'applications
- créer et utiliser une implémentation de l'interface org.quartz.utils.ConnectionProvider

Il est préférable que la taille du pool de connexions lié à la Datasource soit plus grande que la taille du pool de threads. Lors de l'utilisation du JobStoreCMT, la taille du pool de la Datasource non JTA doit être au moins de 4.

Une Datasource créée par Quartz peut être configurée grâce à plusieurs propriétés :

Nom de la propriété	Type	Description
org.quartz.dataSource.NAME.driver	String	Nom pleinement qualifié de la classe Java qui correspond au driver JDBC à utiliser (Obligatoire) Valeur par défaut : null
org.quartz.dataSource.NAME.URL	String	L'url de connexion à la base de données Valeur par défaut : null
org.quartz.dataSource.NAME.user	String	Le nom de l'utilisateur utilisé pour se connecter à la base de données Valeur par défaut : ""
org.quartz.dataSource.NAME.password	String	Le mot de passe de l'utilisateur utilisé pour se connecter à la base de données Valeur par défaut : ""
org.quartz.dataSource.NAME.maxConnections	int	Le nombre maximum de connexions que pourra contenir le pool de connexions Valeur par défaut : 10
org.quartz.dataSource.NAME.validationQuery	String	Requête SQL utilisée pour tester les connexions invalides Valeur par défaut : null
org.quartz.dataSource.NAME.validateOnCheckout	boolean	

		Demander l'exécution d'une requête SQL chaque fois d'une connexion est obtenue du pool pour vérifier qu'elle est toujours valide Valeur par défaut : false
org.quartz.dataSource.NAME.discardIdleConnectionsSeconds	int	Timeout en secondes pour retirer une connexion inutilisée du pool de connexions Valeur par défaut : 0

Exemple :

```
org.quartz.dataSource.QUARTZ_DS.driver = oracle.jdbc.driver.OracleDriver
org.quartz.dataSource.QUARTZ_DS.URL = jdbc:oracle:thin:@localhost:1521:demodb
org.quartz.dataSource.QUARTZ_DS.user = mon_user
org.quartz.dataSource.QUARTZ_DS.password = user_mdp
org.quartz.dataSource.QUARTZ_DS.maxConnections = 10
```

Chaque Datasource définie doit avoir un nom unique qui correspond à NAME dans le tableau ci-dessous. L'utilisation de Datasource configurée dans le serveur d'applications se fait grâce à plusieurs propriétés :

Nom de la propriété	Type	Description
org.quartz.dataSource.NAME.jndiURL	String	L'URL JNDI de la datasource configurée dans le serveur d'applications. (Obligatoire) Valeur par défaut : null
org.quartz.dataSource.NAME.java.naming.factory.initial	String	Nom pleinement qualifié de la classe de type InitialContextFactor Valeur par défaut : null
org.quartz.dataSource.NAME.java.naming.provider.url	String	URL pour se connecter au contexte JNDI Valeur par défaut : null
org.quartz.dataSource.NAME.java.naming.security.principal	String	Le nom de l'utilisateur pour se connecter au contexte JNDI Valeur par défaut : null
org.quartz.dataSource.NAME.java.naming.security.credentials	String	Le mot de passe de l'utilisateur pour se connecter au contexte JNDI Valeur par défaut : null

113.2.10.5. L'utilisation du moteur en mode client/serveur

Il est possible d'utiliser Quartz en mode client/serveur : dans ce mode, le scheduler s'exécute dans une JVM distincte. Le client peut interagir avec le scheduler en utilisant de manière transparente RMI.

Cela peut être utile pour mutualiser le scheduler sur plusieurs applications et/ou pour réduire l'activité sur le client puisque l'exécution du scheduler et des tâches qu'il va déclencher se fait sur la JVM distante.

La configuration de l'utilisation en mode client/serveur utilise plusieurs propriétés de la configuration.

Nom de la propriété	Type	Description
org.quartz.scheduler.rmi.export	String	Demander au scheduler d'exporter ses fonctionnalités grâce à RMI

		Valeur par défaut : false
org.quartz.scheduler.rmi.registryHost	String	Préciser le nom du host sur lequel s'exécute le registre RMI : c'est généralement la machine locale Valeur par défaut : localhost
org.quartz.scheduler.rmi.registryPort	String	Préciser le port sur lequel le registre RMI écoute Valeur par défaut : 1099
org.quartz.scheduler.rmi.createRegistry	String	Définir si oui et comment le registre RMI est créé. Plusieurs valeurs sont utilisables : • false ou never : ne jamais créé le registre • true ou as_needed : tentative d'utilisation d'un registre existant avant d'en créer un • always : toujours tenter de créer un registre et utiliser un existant si cela échoue Valeur par défaut : never
org.quartz.scheduler.rmi.serverPort	String	Préciser le numéro du port sur lequel le Scheduler attend des connexions Valeur par défaut : random
org.quartz.scheduler.rmi.proxy	String	Préciser que l'on souhaite se connecter à un scheduler distant. Dans ce cas, l'instance du scheduler sera un proxy permettant d'invoquer des méthodes du scheduler distant Valeur par défaut : false

Remarque : il n'est pas cohérent de mettre les valeurs true en même temps aux propriétés org.quartz.scheduler.rmi.export et org.quartz.scheduler.rmi.proxy.

Pour lancer le scheduler, il est nécessaire de mettre en place un RMISecurityManager.

Exemple :

```java
package fr.jmdoudoux.dej.quartz;

import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.SchedulerFactory;
import org.quartz.impl.StdSchedulerFactory;

public class QuartzServeur {

  public static void main(final String[] args) {
    if (System.getSecurityManager() != null) {
      System.setSecurityManager(new java.rmi.RMISecurityManager());
    }
    try {
      final SchedulerFactory schedulerFactory = new StdSchedulerFactory();
      final Scheduler scheduler = schedulerFactory.getScheduler();
      scheduler.start();
    } catch (final SchedulerException se) {
      se.printStackTrace();
    }
  }
}
```

Il faut configurer le scheduler pour permettre une invocation de ces fonctionnalités en utilisant RMI.

Exemple : le fichier Quartz_serveur.properties

```
org.quartz.scheduler.instanceName = scheduler_serveur
org.quartz.scheduler.instanceId = scheduler1

org.quartz.scheduler.rmi.export = true
org.quartz.scheduler.rmi.registryHost = localhost
org.quartz.scheduler.rmi.registryPort = 1099
org.quartz.scheduler.rmi.createRegistry = true

org.quartz.threadPool.class = org.quartz.simpl.SimpleThreadPool
org.quartz.threadPool.threadCount = 4

org.quartz.plugin.shutdownhook.class = org.quartz.plugins.management.ShutdownHookPlugin
org.quartz.plugin.shutdownhook.cleanShutdown = true
```

Il faut préciser le fichier de configuration en tant que paramètre de la JVM :

-Dorg.quartz.properties=src/main/resources/quartz_serveur.properties

Résultat :

```
INFO StdSchedulerFactory - Using default implementation for ThreadExecutor
INFO SchedulerSignalerImpl - Initialized Scheduler Signaller of type:
class org.quartz.core.SchedulerSignalerImpl
INFO QuartzScheduler - Quartz Scheduler v.2.2.1 created.
INFO ShutdownHookPlugin - Registering Quartz shutdown hook.
INFO RAMJobStore - RAMJobStore initialized.
INFO QuartzScheduler - Scheduler bound to RMI registry under name
'scheduler_serveur_$_scheduler1'
INFO QuartzScheduler - Scheduler meta-data: Quartz Scheduler (v2.2.1)
'scheduler_serveur' with instanceId 'scheduler1'
  Scheduler class: 'org.quartz.core.QuartzScheduler' - access via RMI.
  NOT STARTED.
  Currently in standby mode.
  Number of jobs executed: 0
  Using thread pool 'org.quartz.simpl.SimpleThreadPool' - with 4 threads.
  Using job-store 'org.quartz.simpl.RAMJobStore' - which does not support
  persistence. and is not clustered.

INFO StdSchedulerFactory - Quartz scheduler 'scheduler_serveur' initialized
 from specified file: 'src/main/resources/quartz_serveur.properties'
INFO StdSchedulerFactory - Quartz scheduler version: 2.2.1
INFO QuartzScheduler - Scheduler scheduler_serveur_$_scheduler1 started.
```

Dans le mode client/serveur, l'instance du scheduler côté client est uniquement un proxy qui permet d'invoquer les fonctionnalités sur l'instance distante du scheduler.

Exemple :

```
package fr.jmdoudoux.dej.quartz;

import java.util.Date;

import org.quartz.DateBuilder;
import org.quartz.JobBuilder;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerFactory;
import org.quartz.SimpleScheduleBuilder;
import org.quartz.Trigger;
import org.quartz.TriggerBuilder;
import org.quartz.TriggerKey;
import org.quartz.impl.StdSchedulerFactory;

public class QuartzClient {

  public static void main(final String[] args) {
    try {
```

```
   final SchedulerFactory factory = new StdSchedulerFactory();
   final Scheduler scheduler = factory.getScheduler();

   final JobDetail jobDetail = JobBuilder
     .newJob(MonJob.class)
     .withIdentity("monJob")
     .storeDurably()
     .build();

   JobBuilder
     .newJob(MonJob.class)
     .withIdentity("monJobNouveau")
     .storeDurably()
     .build();

   final Trigger trigger = TriggerBuilder
     .newTrigger()
     .withIdentity(new TriggerKey("monTrigger", "monGroup"))
     .withSchedule(
       SimpleScheduleBuilder
         .simpleSchedule()
         .withIntervalInMinutes(1)
         .repeatForever())
     .startAt(DateBuilder.evenMinuteDate(new Date()))
     .build();

   scheduler.scheduleJob(jobDetail, trigger);

   System.in.read();
  } catch (final Exception e) {
   e.printStackTrace();
  }
 }
}
```

La configuration du scheduler précise uniquement les informations utiles pour invoquer le scheduler distant grâce à RMI.

Exemple : Quartz_client.properties

```
org.quartz.scheduler.instanceName = scheduler_serveur
org.quartz.scheduler.instanceId = scheduler1
org.quartz.scheduler.rmi.proxy = true
org.quartz.scheduler.rmi.registryHost = localhost
org.quartz.scheduler.rmi.registryPort = 1099
```

Il faut préciser le fichier de configuration en tant que paramètre de la JVM

-Dorg.quartz.properties=src/main/resources/quartz_client.properties

113.2.10.6. La configuration des plugins

Chaque plugin doit être défini avec un nom unique qui doit suivre org.quarts.plugin dans la définition de la configuration.

Pour utiliser un plugin, il faut préciser le nom pleinement qualifié de sa classe comme valeur de la propriété org.quartz.plugin.XXX.class où XXX est le nom unique du plugin dans l'instance du scheduler.

Il est possible de fournir une valeur à un paramètre du plugin en utilisant la propriété org.quartz.plugin.XXX.nom_propriete où XXX est le nom unique du plugin et nom_propriete est le nom de la propriété qui sera valorisée par introspection.

Quartz propose en standard plusieurs plugins contenus dans le package org.quartz.plugins et ses sous-packages :

- org.quartz.plugins.SchedulerPluginWithUserTransactionSupport
- org.quartz.plugins.history.LoggingJobHistoryPlugin

- org.quartz.plugins.history.LoggingTriggerHistoryPlugin
- org.quartz.plugins.management.ShutdownHookPlugin
- org.quartz.plugins.xml.XMLSchedulingDataProcessorPlugin

La classe abstraite SchedulerPluginWithUserTransactionSupport offre des fonctionnalités de base pour développer des plugins dont les méthodes start() et shutdown() ont en paramètre un UserTransaction. C'est utile lorsqu'un JobStoreCMT est utilisé et que le plugin interagit avec des jobs ou des triggers. Si le JobStore utilisé est de type JobStoreCMT, il faut obligatoirement que la propriété wrapInUserTransaction soit positionnée à la valeur true. Le paramètre de type UserTransaction ne doit pas être null si la propriété wrapInUserTransaction est à true. La méthode initialize() est invoquée par le Scheduler lors de l'initialisation du plugin.

Le plugin LoggingJobHistoryPlugin permet de journaliser l'exécution des jobs dans un log en utilisant la bibliothèque Apache Commons Logging.

De manière similaire, le plugin LoggingTriggerHistoryPlugin permet de journaliser l'exécution des triggers dans un log.

Il est possible de configurer le contenu journalisé selon le type de messages en utilisant la syntaxe de la classe java.util.MessageFormat. En fonction du type de message, plusieurs placeholders ({n} où n varie de 0 à 8) sont utilisables pour configurer le contenu du message. Chaque message associe un numéro à une donnée : chacune de ces valeurs est détaillée dans la Javadoc.

Résultat :

```
org.quartz.plugin.jobsLogging.class = org.quartz.plugins.history.LoggingTriggerHistoryPlugin
org.quartz.plugin.jobsLogging.jobSuccessMessage=Le job [{1}.{0}]
a terminé son execution avec le code retour : {8}
org.quartz.plugin.jobsLogging.jobFailedMessage=Le job [{1}.{0}]
a terminé son execution avec l'exception: {8}
org.quartz.plugin.jobsLogging.jobWasVetoedMessage=L'execution du job [{1}.{0}]
a été bloquée. Il a été déclenché
par le trigger [{4}.{3}] à {2, date, dd-MM-yyyy HH:mm:ss.SSS}

org.quartz.plugin.triggersLogging.class=org.quartz.plugins.history.LoggingTriggerHistoryPlugin
org.quartz.plugin.triggersLogging.triggerFiredMessage=Le trigger [{1}.{0}]
déclenche le job [{6}.{5}] prévu à :
{2, date, dd-MM-yyyy HH:mm:ss.SSS}, prochaine exécution à :
{3, date, dd-MM-yyyy HH:mm:ss.SSS}
org.quartz.plugin.triggersLogging.triggerCompleteMessage=Le trigger [{1}.{0}]
a terminé le déclenchement du job [{6}.{5}] avec le résultat : {9}.
org.quartz.plugin.triggersLogging.triggerMisfiredMessage=Le trigger [{1}.{0}]
a rate le déclenchement du job [{6}.{5}] qui aurait du survenir à
{3, date, dd-MM-yyyy HH:mm:ss.SSS}
```

Le plugin ShutdownHookPlugin permet de capturer l'événement de terminaison de la JVM pour permettre un arrêt propre du scheduler.

Exemple :

```
org.quartz.plugin.shutdownhook.class = org.quartz.plugins.management.ShutdownHookPlugin
org.quartz.plugin.shutdownhook.cleanShutdown = true
```

Le plugin XMLSchedulingDataProcessorPlugin permet de lire un fichier XML contenant la définition de jobs et triggers qui seront lus et enregistrés dans le scheduler lors de son démarrage. Il permet aussi de gérer les jobs et les triggers.

Exemple :

```
org.quartz.plugin.initialisation.class=org.quartz.plugins.xml.XMLSchedulingDataProcessorPlugin
org.quartz.plugin.initialisation.fileNames=ma_configuration.xml
org.quartz.plugin.initialisation.failOnFileNotFound=true
```

Pour la version 1.8.5 de Quartz, le schéma du fichier de configuration XML peut être obtenu à l'url :

https://www.quartz-scheduler.org/xml/job_scheduling_data_1_8.xsd

A partir de la version 2.0 de Quartz, le schéma du fichier de configuration XML peut être obtenu à l'url :

https://www.quartz-scheduler.org/xml/job_scheduling_data_2_0.xsd

113.2.11. L'utilisation en cluster

Plusieurs instances de Scheduler peuvent être rassemblées dans un cluster qui permet :

- une meilleure répartition de la charge d'exécution des jobs grâce au load balancing
- une haute disponibilité grâce au fail over

Le partage des informations entre les instances du cluster se fait en utilisant un JDBCJobStore.

Chaque instance du cluster doit avoir dans sa configuration la valeur true dans sa propriété org.quartz.jobStore.isClustered. Chacun d'eux doit avoir une valeur unique pour un même cluster dans sa propriété org.quartz.scheduler.instanceId.

Toutes les horloges des machines exécutant des instances du cluster doivent être synchronisées.

114. Des bibliothèques open source

Chapitre 114

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

L'écosystème Java est très riche en bibliothèques open source qui couvrent de très nombreux sujets. Leur utilisation permet d'être productif avec des solutions fiables et stables.

Ce chapitre contient plusieurs sections :

- ♦ JFreeChart
- ♦ Beanshell
- ♦ Apache Commons
- ♦ JGoodies
- ♦ Apache Lucene

114.1. JFreeChart

JFreeChart est une bibliothèque open source qui permet d'afficher des données statistiques sous la forme de graphiques. Elle possède plusieurs formats dont le camembert, les barres ou les lignes et propose de nombreuses options de configuration pour personnaliser le rendu des graphiques. Elle peut s'utiliser dans des applications standalones ou des applications web et permet également d'exporter le graphique sous la forme d'une image.

https://www.jfree.org/jfreechart/

La version utilisée dans cette section est la 0.9.18.

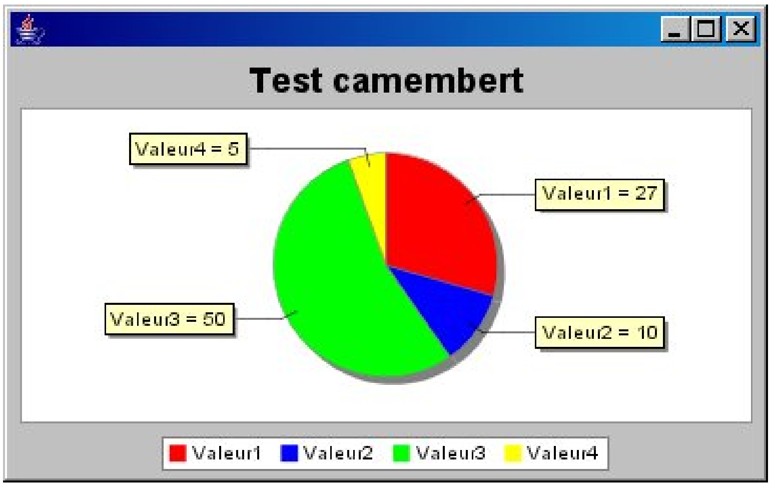
Pour l'utiliser, il faut télécharger le fichier jfreechart-0.9.18.zip et le décompresser. Son utilisation nécessite l'ajout dans le classpath des fichiers jfreechart-0.9.18.zip et des fichiers .jar présents dans le répertoire lib décompressé.

Les données utilisées dans le graphique sont encapsulées dans un objet de type Dataset. Il existe plusieurs sous-types de cette classe en fonction du type de graphique souhaité.

Un objet de type JFreechart encapsule le graphique. Une instance d'un tel objet est obtenue en utilisant une des méthodes de la classe ChartFactory.

Exemple : Un exemple avec un graphique en forme de camembert

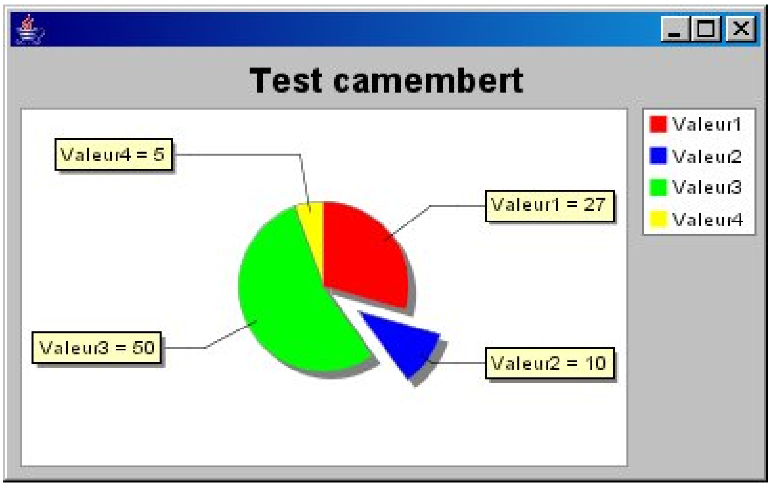
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import org.jfree.chart.*;
import org.jfree.chart.plot.*;
import org.jfree.data.*;

public class TestPieChart extends JFrame {
  private JPanel pnl;

  public TestPieChart() {
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        dispose();
        System.exit(0);
      }
    });
    pnl = new JPanel(new BorderLayout());
    setContentPane(pnl);
    setSize(400, 250);

    DefaultPieDataset pieDataset = new DefaultPieDataset();
    pieDataset.setValue("Valeur1", new Integer(27));
    pieDataset.setValue("Valeur2", new Integer(10));
    pieDataset.setValue("Valeur3", new Integer(50));
    pieDataset.setValue("Valeur4", new Integer(5));

    JFreeChart pieChart = ChartFactory.createPieChart("Test camembert",
      pieDataset, true, true, true);
    ChartPanel cPanel = new ChartPanel(pieChart);
    pnl.add(cPanel);
  }

  public static void main(String args[]) {
    TestPieChart tpc = new TestPieChart();
    tpc.setVisible(true);
  }
}
```



Pour chaque graphique, il existe de nombreuses possibilités de configuration.

Exemple :

```
...
    JFreeChart pieChart = ChartFactory.createPieChart("Test camembert",
      pieDataset, true, true, true);
    PiePlot piePlot = (PiePlot) pieChart.getPlot();
    piePlot.setExplodePercent(1, 0.5);
    Legend legend = pieChart.getLegend();
    legend.setAnchor(Legend.EAST_NORTHEAST);
    ChartPanel cPanel = new ChartPanel(pieChart);
...
```



Il est très facile d'exporter le graphique dans un flux.

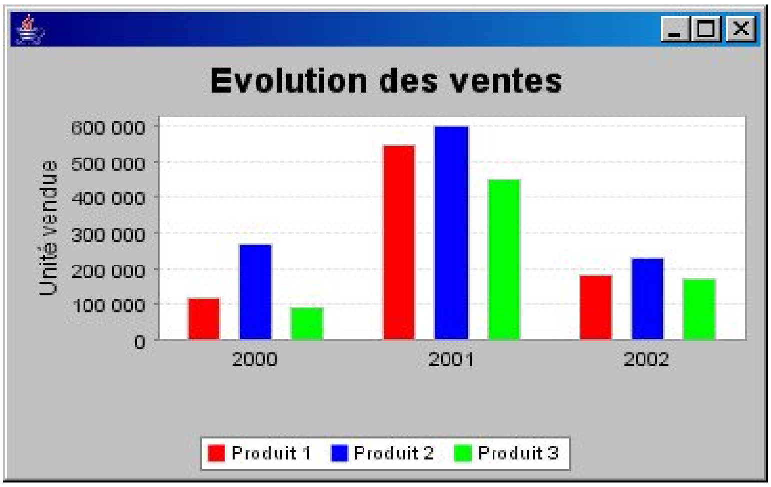
Exemple : enregistrement du graphique dans un fichier

```
...
    File fichier = new File("image.png");
    try {
      ChartUtilities.saveChartAsPNG(fichier, pieChart, 400, 250);
    } catch (IOException e) {
      e.printStackTrace();
    }
...
```

JFreeChart propose aussi plusieurs autres types de graphiques dont les histogrammes.

Exemple : un graphique sous formes de barres

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import org.jfree.chart.*;
import org.jfree.chart.plot.*;
import org.jfree.data.*;

public class TestBarChart extends JFrame {
  private JPanel pnl;

  public TestBarChart() {
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        dispose();
        System.exit(0);
      }
    });
    pnl = new JPanel(new BorderLayout());
    setContentPane(pnl);
    setSize(400, 250);

    DefaultCategoryDataset dataset = new DefaultCategoryDataset();
    dataset.addValue(120000.0, "Produit 1", "2000");
    dataset.addValue(550000.0, "Produit 1", "2001");
    dataset.addValue(180000.0, "Produit 1", "2002");
    dataset.addValue(270000.0, "Produit 2", "2000");
    dataset.addValue(600000.0, "Produit 2", "2001");
    dataset.addValue(230000.0, "Produit 2", "2002");
    dataset.addValue(90000.0, "Produit 3", "2000");
    dataset.addValue(450000.0, "Produit 3", "2001");
    dataset.addValue(170000.0, "Produit 3", "2002");

    JFreeChart barChart = ChartFactory.createBarChart("Evolution des ventes", "",
      "Unité vendue", dataset, PlotOrientation.VERTICAL, true, true, false);
    ChartPanel cPanel = new ChartPanel(barChart);
    pnl.add(cPanel);
  }

  public static void main(String[] args) {
    TestBarChart tbc = new TestBarChart();
    tbc.setVisible(true);
  }
}
```



JFreechart peut aussi être mis en oeuvre dans une application web, le plus pratique étant d'utiliser une servlet qui renvoie dans la réponse une image générée par JfreeChart.

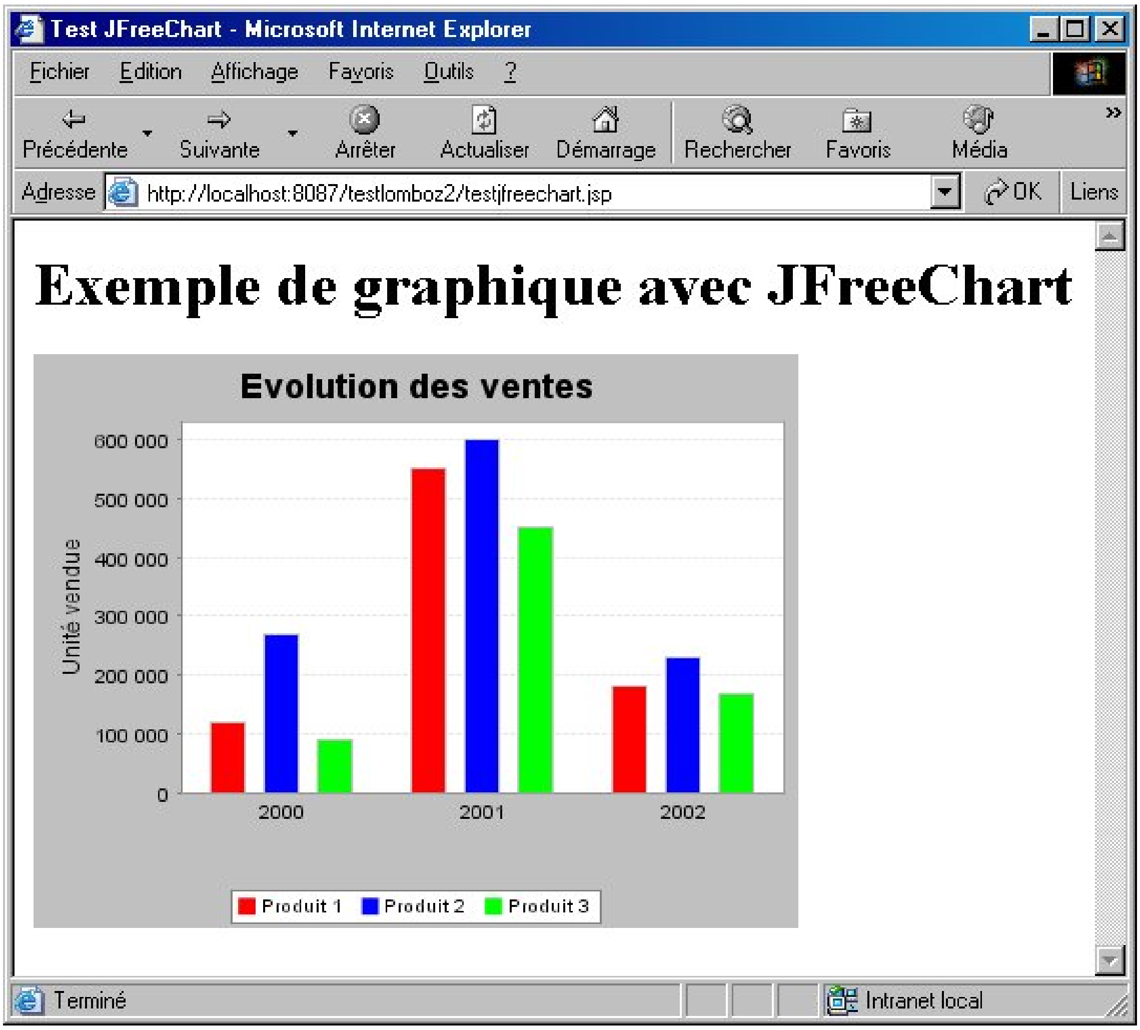
Exemple : JSP qui affiche le graphique

```
<%@ page language="java" %>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
<title>Test JFreeChart</title>
</head>
<body bgcolor="#FFFFFF">
<H1>Exemple de graphique avec JFreeChart</h1>
<img src="/testlomboz2/ServletBarChart">
</body>
</html>
```

Dans l'exemple précédent, l'image contenant le graphique est générée par une servlet.

Exemple : servlet qui générère l'image

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import org.jfree.chart.*;
import org.jfree.chart.plot.*;
import org.jfree.data.*;

public class ServletBarChart extends HttpServlet {

  protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    DefaultCategoryDataset dataset = new DefaultCategoryDataset();
    dataset.addValue(120000.0, "Produit 1", "2000");
    dataset.addValue(550000.0, "Produit 1", "2001");
    dataset.addValue(180000.0, "Produit 1", "2002");
    dataset.addValue(270000.0, "Produit 2", "2000");
    dataset.addValue(600000.0, "Produit 2", "2001");
    dataset.addValue(230000.0, "Produit 2", "2002");
    dataset.addValue(90000.0, "Produit 3", "2000");
    dataset.addValue(450000.0, "Produit 3", "2001");
    dataset.addValue(170000.0, "Produit 3", "2002");

    JFreeChart barChart = ChartFactory.createBarChart("Evolution des ventes", "",
      "Unité vendue", dataset, PlotOrientation.VERTICAL, true, true, false);
    OutputStream out = response.getOutputStream();
    response.setContentType("image/png");
    ChartUtilities.writeChartAsPNG(out, barChart, 400, 300);
  }
}
```



Cette section n'a proposé qu'une introduction à JFreeChart en proposant quelques exemples très simples sur les nombreuses possibilités de cette puissante bibliothèque.

114.2. Beanshell

Beanshell est un interpréteur de scripts qu'il est possible d'intégrer dans une application.

Le site officiel du projet est à l'url http://www.beanshell.org/

114.3. Apache Commons

Le projet Apache Commons a pour but de fournir des utilitaires.

Le site officiel du projet est à l'url https://commons.apache.org/.

114.4. JGoodies

Cette bibliothèque propose de faciliter le développement d'applications graphiques utilisant Swing.

Le site officiel du projet est à l'url : http://www.jgoodies.com/.

114.5. Apache Lucene

Apache Lucene est un moteur d'indexation et de recherche de texte.

Le site officiel du projet est à l'url : https://lucene.apache.org/.

115. Apache Commons

Chapitre 115

Le projet Apache Commons est un ensemble de bibliothèques regroupant des utilitaires par thèmes.

Le projet est composé de trois parties :

- Commons Proper : les bibliothèques actives
- Commons Sandbox : les bibliothèques en cours de développement
- Commons Dormant : les bibliothèques inactives

La partie Commons Proper contient de nombreuses bibliothèques :

Bibliothèque	Rôle
BCEL	Byte Code Engineering Library : manipuler le bytecode et créer des fichiers .class
BeanUtils	Faciliter l'utilisation de l'API Reflection
Betwixt	Fonctionnalité de type OXM (mapping objet/XML)
BSF	Bean Scripting Framework : interface pour utiliser des langages de scripting (JSR-223)
Chain	Implémentation du motif de concept chaîne de responsabilités
CLI	Analyser les arguments fournis par la ligne de commandes
Codec	Proposer des algorithmes d'encodage/décodage (Base64, ...)
Collections	Fournir différentes collections
Compress	Utiliser des algorithmes de compression de fichiers comme zip, tar, ...
Configuration	Gérer des configurations provenant de différents formats
CSV	Gérer des données au format CSV (chaque valeur est séparée par une virgule)
Daemon	Mettre en oeuvre des Services sous Windows ou des daemons sous Unix
DBCP	Fournir une implémentation de pool de connexions vers une base de données
DbUtils	Fournir des utilitaires concernant JDBC
Digester	Faciliter l'extraction de données d'un document XML
Discovery	Proposer un service de localisation de ressources
EL	Interpréteur pour l'Expression Language défini dans les spécifications de JSP 2.0
Email	Faciliter l'utilisation d'emails
Exec	Faciliter la gestion de l'exécution de processus externes
FileUpload	Faciliter la mise en oeuvre de fonctionnalités de type upload de fichiers dans une webapp
Functor	Manipuler des fonctions en tant qu'objets
Imaging	Fournir des fonctionnalités de manipulation d'images
IO	Fournir des utilitaires pour les opérations de type entrée/sortie

JCI	Java Compiler Interface
JCS	Fournir une solution de cache
Jelly	Moteur de traitements orienté XML
Jexl	Java Expression Language
JXPath	Permettre la manipulation de Java Beans en utilisant la syntaxe XPath
Lang	Fournir des utilitaires de base
Launcher	Lanceur d'applications Java multi-plateforme
Logging	Wrapper qui permet d'utiliser plusieurs implémentations d'API de logging
Math	Librairie de fonctions mathématiques et statistiques
Modeler	Faciliter la création de Model MBeans respectant les spécifications JMX
Net	Fournir des utilitaires pour les opérations de type entrée/sortie
OGNL	Langage d'expression permettant, entre autres, de manipuler les propriétés d'objets Java
Pool	Fournir une solution de type pool d'objets
Primitives	Librairie de collections et d'outils concue spécialement pour l'utilisation des types primitifs
Proxy	Faciliter la création de proxys dynamiques
SCXML	
Transaction	
Validator	
VFS	

Le site du projet est à l'url : https://commons.apache.org/

Ce chapitre contient plusieurs sections :

- Apache Commons Configuration
- Apache Commons CLI

115.1. Apache Commons Configuration

L'API Apache Commons Configuration propose une solution pour faciliter la gestion de la configuration d'une application en gérant ses paramètres.

Apache Commons Configuration propose de nombreuses fonctionnalités dont les principales sont :

- le chargement de la configuration à partir de différentes sources (Fichiers Properties, Fichier XML, Fichier INI de Windows, Fichiers Property list (plist), JNDI, JDBC, Propriétés système, ...)
- l'agrégation possible de différentes sources avec une priorisation de leur importance dans une hiérarchie
- un accès typé à la valeur d'une clé qui peut être simple ou multivaleur
- la modification et la persistance de la configuration
- la gestion d'événements lors du changement de la configuration

Indépendamment de la solution proposée par Apache Commons Configuration pour accéder et gérer une configuration, cette bibliothèque permet aussi d'agréger plusieurs sources différentes pour composer le contenu de la configuration. Cette agrégation est réalisée en utilisant les classes ConfigurationFactory et CompositeConfiguration.

115.1.1. L'interface Configuration

L'interface Configuration permet de manipuler ou modifier les éléments de la configuration de manière générique. Chaque classe qui encapsule une source de configuration implémente l'interface Configuration.

Un élément d'une configuration est une paire clé/valeur.

L'interface définit de nombreuses méthodes pour obtenir la valeur typée d'une clé. Les types primitifs ou objets supportés sont : boolean/Boolean, byte/Byte, double/Double, float/Float, int/Integer, long/Long, short, BigDecimal, BigInteger, String, String[] et List<Object>.

Le nom de ces méthodes commence par get suivi du type de retour de la donnée et elles attendent toute au moins un paramètre de type String qui contient le nom de la clé dont on veut obtenir la valeur. La méthode tente alors de retrouver la valeur de la clé dans la configuration et de la convertir dans le type retourné par la méthode. La plupart de ces méthodes possèdent une surcharge avec un paramètre supplémentaire permettant de préciser une valeur par défaut si la clé n'est pas trouvée dans la configuration.

Par défaut, si la clé n'est pas trouvée et que la valeur de retour est un objet, alors la valeur retournée sera null. Il est possible de changer ce comportement par défaut pour lever une exception de type NoSuchElementException en invoquant la méthode setThrowExceptionOnMissing() et en lui passant la valeur true. Une exception de type ConversionException est levée si la valeur ne peut pas être convertie dans le type retourné par la méthode.

Pour les types primitifs, une exception de type NoSuchElementException est levée.

Exemple :

```
package fr.jmdoudoux.dej.commons.configuration;

import java.util.NoSuchElementException;

import org.apache.commons.configuration.BaseConfiguration;
import org.apache.commons.configuration.Configuration;

public class TestBasicConfiguration {

  public static void main(final String[] args) {
    final Configuration config = new BaseConfiguration();
    try {
      System.out.println(config.getBoolean("maValeur"));

    } catch (final NoSuchElementException nsee) {
      System.out.println("La propriété maValeur n'est pas trouvée");
    }
    System.out.println(config.getBoolean("maValeur", true));
    System.out.println(config.getBoolean("maValeur", Boolean.TRUE));
    config.setProperty("maValeur", false);
    System.out.println(config.getBoolean("maValeur", true));
  }
}
```

Résultat :

```
La propriété maValeur n'est pas trouvée
true
true
false
```

La méthode getProperty(), qui attend en paramètre le nom de la clé, renvoie la valeur sous la forme d'un Object.

Les méthodes getList() et getStringArray() de l'interface Configuration, qui attendent en paramètre le nom d'une clé, permettent d'obtenir les valeurs associées à une propriété multivaleur. Une telle propriété est de type String et contient au moins un caractère qui est défini comme étant le délimiteur. Par défaut, ce caractère de délimitation de chaque occurrence est la virgule. La méthode setListDelimiter() permet de préciser un autre caractère.

Il est possible d'utiliser la méthode statique setDefaultListDelimiter() de la classe AbstractConfiguration pour modifier le caractère de délimitation pour toutes les configurations.

Passer true en paramètre de la méthode setDelimiterParsingDisabled() permet de désactiver le découpage des valeurs des propriétés. Dans ce cas, la configuration ne propose plus le support des multivaleurs.

Les méthodes getList() et getStringArray() renvoie toujours null si la clé n'est pas trouvée.

La valeur d'une clé peut faire référence à une autre clé : lors de la récupération de la valeur la référence est remplacée par la valeur de la clé correspondante.

Plusieurs méthodes permettent de gérer le contenu de la configuration :

Méthode	Rôle
clearProperty(String key)	Supprimer la clé de la configuration
addProperty(String key, Object value)	Ajouter une clé dans la configuration en lui associant la valeur fournie. Si la propriété existe déjà, la nouvelle valeur est ajoutée à la valeur existante. La clé possède alors plusieurs valeurs
setProperty(String key, Object value)	Modifier la valeur d'une clé avec celle fournie en paramètre. La clé est créée dans la configuration si elle n'existe pas
clear()	Supprimer tous les éléments de la configuration

115.1.2. Les différents types de configuration

Toutes les classes qui encapsulent une source de configuration implémentent l'interface org.apache.commons.configuration.Configuration.

L'API propose une classe pour gérer chaque source de configuration :

- BaseConfiguration : la configuration est stockée en mémoire
- PropertiesConfiguration : la configuration est stockée dans un fichier de type .properties
- PropertyListConfiguration : la configuration est stockée dans un fichier de type .plist
- XMLPropertiesConfiguration
- XMLPropertyListConfiguration (Mac OS X)
- XMLConfiguration : la configuration est stockée dans un fichier XML
- INIConfiguration : la configuration est stockée dans un fichier de type .ini (Windows)
- SystemConfiguration : la configuration contient les variables d'environnement système
- DataBaseConfiguration
- XMLConfiguration
- JNDIConfiguration : les éléments de la configuration sont retrouvés en utilisant JNDI
- HierarchicalConfiguration : la configuration est stockée en mémoire
- ConfigurationDynaBean : permet d'utiliser la configuration comme un bean dynamique
- ConfigurationFactory : permet de gérer une configuration composite qui va pouvoir contenir plusieurs sources potentiellement de types différents.

En fonction des sources de configuration, l'API Common Configuration va requérir diverses dépendances.

115.1.2.1. Les configurations reposant sur des fichiers

Les classes qui encapsulent des configurations stockées dans des fichiers implémentent l'interface org.apache.commons.configuration.FileConfiguration.

L'interface FileConfiguration propose plusieurs méthodes pour préciser le fichier contenant la configuration.

- setFile() : attend en paramètre un objet de type java.io.File
- setURL() : attend en paramètre un objet de type URL

- setFileName() : attend un objet de type String qui précise le nom du fichier
- setBasePath() : attend un objet de type String qui précise le chemin du fichier

Une instance de type File ou une URL permet d'identifier de manière unique un fichier. Si le fichier est précisé en utilisant un chemin et/ou un nom de fichier alors le framework tente de déterminer le fichier dans un ordre précis :

- si la combinaison du chemin et du nom du fichier est un URL, alors cette URL est utilisée pour charger le fichier
- si la combinaison du chemin et du nom du fichier est un chemin absolu vers le fichier, alors ce chemin est utilisé pour charger le fichier
- si la combinaison du chemin et du nom du fichier est un chemin relatif vers le fichier, alors ce chemin est utilisé pour charger le fichier
- si le nom du fichier existe dans le répertoire HOME de l'utilisateur, alors ce fichier est chargé
- le fichier est considéré comme une ressource et sera chargé par un classloader à partir de son nom (le fichier doit dans ce cas être dans le classpath)

Si toutes ces tentatives échouent alors une exception de type ConfigurationException est levée.

La méthode load() permet de charger la configuration ou de lever une exception de type ConfigurationException si le chargement échoue. La méthode load() possède plusieurs surcharges qui permettent de préciser une source à charger différente de celle configurée dans l'objet (la source fournie en paramètre ne remplace pas la source configurée dans l'objet).

Il est possible d'invoquer plusieurs fois la méthode load() avec des sources différentes : dans ce cas, la configuration existante n'est pas réinitialisée mais elle est ajoutée.

Généralement les classes qui implémentent l'interface FileConfiguration proposent des constructeurs attendant en paramètre la source sous différents types et invoquent la méthode load() sur la source.

Commons Configuration propose le concept de stratégie de rechargement (reloading strategy). Il est ainsi possible d'utiliser un mécanisme nommé ReloadingStrategy qui permet de déterminer si la configuration doit être rechargée automatiquement si celle-ci a été modifiée. Cette stratégie est invoquée à chaque fois que l'une des méthodes getXXX() est appelée.

Par défaut, un FileConfiguration est associé à une stratégie InvariantReloadingStrategy qui précise que la configuration n'est jamais rechargée.

La méthode setReloadingStrategy() permet de préciser une instance de stratégie différente.

Le rechargement automatique d'une configuration dont la source a été modifiée est particulièrement intéressant notamment pour des applications qui doivent avoir un fort taux de disponibilité. Ce rechargement permet d'éviter d'avoir à redémarrer l'application pour obtenir les nouvelles valeurs de la configuration.

Si la source est un fichier, la stratégie de base est implémentée dans la classe FileChangedReloadingStrategy qui recharge le contenu du fichier si celui-ci a été modifié.

Exemple :

```
package fr.jmdoudoux.dej.commons.configuration;

import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.XMLConfiguration;
import org.apache.commons.configuration.reloading.FileChangedReloadingStrategy;

public class TestXMLConfiguration {

  public static void main(final String[] args) {
    try {
      final XMLConfiguration config = new XMLConfiguration("maConfig.xml");
      final FileChangedReloadingStrategy strategy = new FileChangedReloadingStrategy();
      strategy.setRefreshDelay(5000);
      config.setReloadingStrategy(strategy);
      System.out.println(config.getFile());
      System.out.println("ma.valeur=" + config.getLong("ma.valeur"));
    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

La stratégie FileChangedReloadingStrategy implique que la classe est invoquée à chaque accès à une propriété de la configuration pour vérifier si la date de dernière modification du fichier a changée depuis la précédente modification. Si la date a changée alors la configuration est rechargée. Pour éviter un accès disque lors de l'obtention d'une valeur de la configuration, il est possible de préciser un délai d'attente en millisecondes entre chaque vérification (refresh delay).

La classe ManagedReloadingStrategy est une alternative au rechargement automatique de la configuration : elle permet de forcer le rafraîchissement à la demande en invoquant sa méthode refresh().

Exemple en utilisant Spring et JMX

```
<bean id="configuration" class=" org.apache.commons.configuration.PropertiesConfiguration">
  <constructor-arg type="java.net.URL" value="file:${user.home}/monAppConfig.properties"/>
  <property name="reloadingStrategy" ref="reloadingStrategy"/>
</bean>

<bean id="reloadingStrategy"
  class=" org.apache.commons.configuration.reloading.ManagedReloadingStrategy"/>

<bean id="mbeanMetadataExporter" class="org.springframework.jmx.export.MBeanExporter">
  <property name="server" ref="mbeanServer"/>
  <property name="beans">
    <map>
      <entry key="monApp:bean=configuration" value-ref="reloadingStrategy"/>
    </map>
  </property>
</bean>
```

La méthode save() permet d'enregistrer la configuration en remplaçant la source encapsulée dans l'objet. Il est possible de préciser un autre fichier cible en utilisant une des surcharges de la méthode save().

La méthode setAutoSave() attend en paramètre un booléen qui s'il vaut true permet de demander l'enregistrement de la configuration à chaque fois qu'elle est modifiée. Dans ce cas, chaque modification faite dans la configuration en utilisant l'API va sauvegarder la configuration. Attention toutefois si le nombre de mises à jour est important cela peut engendrer beaucoup d'accès au système de fichiers.

Les changements vont être vérifiés périodiquement par la stratégie de rechargement (reloading strategy) associée à la configuration.

La méthode clear() permet de réinitialiser le contenu de la configuration.

115.1.2.1.1. Les fichiers properties

Ce format de fichier est supporté nativement par la plate-forme Java. Il permet de définir des paires clé/valeur dans un fichier texte, une sur chaque ligne, la clé étant séparée de sa valeur par un caractère '='.

La valeur peut contenir plusieurs valeurs qui doivent dans ce cas être séparées par un caractère ','. Il est aussi possible de définir plusieurs fois la clé en lui affectant à chaque fois une des valeurs.

Si la valeur doit tenir sur plusieurs lignes, chaque ligne sauf la dernière doit être terminée par un caractère '\' (antislash).

Les lignes commençant un caractère '# ' sont des commentaires et sont ignorées.

La classe org.apache.commons.configuration.PropertiesConfiguration encapsule une configuration dont la source est un fichier .properties.

Il est possible d'inclure le contenu d'autres fichiers .properties en utilisant comme clé 'include' et comme valeur le chemin absolu ou relatif du fichier .properties à inclure.

Les commentaires contenus dans le fichier .properties source sont perdus lors de l'enregistrement de la configuration. La méthode setHeader() permet cependant de préciser un commentaire qui sera écrit dans le fichier .properties lors de son enregistrement.

Plusieurs implémentations de l'interface Configuration héritent de la classe AbstractConfiguration.

Il faut créer un fichier .properties qui va contenir les éléments de la configuration.

Résultat :

```
data.database.url=127.0.0.1
data.database.port=3306
data.database.login=admin
data.database.password=password
```

Le plus simple est de créer une instance de la classe PropertiesConfiguration en lui passant en paramètre le nom du fichier .properties.

Exemple :

```
package fr.jmdoudoux.dej.commons.configuration;

import org.apache.commons.configuration.Configuration;
import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.PropertiesConfiguration;

public class TestPropertiesConfiguration {

  public static void main(final String[] args) {
    try {
      final Configuration config = new PropertiesConfiguration("maConfig.properties");
      final String url = config.getString("data.database.url");
      System.out.println("url = " + url);
    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

Si le chemin fourni en paramètre n'est pas un chemin absolu, alors le fichier est successivement recherché dans le répertoire courant, le répertoire HOME de l'utilisateur et dans le classpath.

Le fichier peut aussi être chargé en utilisant une des surcharges de la méthode load().

Il est possible de demander l'inclusion d'autres fichiers .properties en utilisant une paire clé/valeur particulière dans le fichier .properties : le nom de la clé doit être obligatoirement include et la valeur est le nom du fichier .properties à inclure.

Résultat :

```
include = db.properties
```

Il est possible d'associer plusieurs valeurs à une même clé en utilisant deux syntaxes :

- séparer les différentes valeurs associées à une clé en utilisant le caractère virgule
- définir une ligne pour chaque valeur avec la même clé

Résultat :

```
valeurs.paires = 0, 2, 4, 6, 8
valeurs.impaires = 1
valeurs.impaires = 3
valeurs.impaires = 5
valeurs.impaires = 7
valeurs.impaires = 9
```

Il est possible d'obtenir directement un tableau ou une collection des valeurs associées à la clé en utilisant les méthodes getStringArray() ou getList().

Exemple :

```
package fr.jmdoudoux.dej.commons.configuration;

import java.util.Arrays;
import java.util.List;

import org.apache.commons.configuration.Configuration;
import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.PropertiesConfiguration;

public class TestPropertiesConfiguration {

  public static void main(final String[] args) {
    try {
      final Configuration config = new PropertiesConfiguration("maConfig.properties");
      final String[] valeursPaires = config.getStringArray("valeurs.paires");
      final List<Object> valeursImpaires = config.getList("valeurs.impaires");
      System.out.println("Valeurs paires=" + Arrays.deepToString(valeursPaires));
      System.out.println("Valeurs impaires=" + valeursImpaires);
    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
Valeurs paires=[0, 2, 4, 6, 8]
Valeurs impaires=[1, 3, 5, 7, 9]
```

Pour sauvegarder une configuration modifiée, il suffit d'invoquer la méthode save().

Exemple :

```
package fr.jmdoudoux.dej.commons.configuration;

import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.PropertiesConfiguration;

public class TestPropertiesConfiguration {

  public static void main(final String[] args) {
    try {
      final PropertiesConfiguration config =
        new PropertiesConfiguration("maConfig.properties");
      System.out.println(config.getFile());
      System.out.println("ma.valeur=" + config.getLong("ma.valeur"));
      config.setProperty("ma.valeur", 100);
      config.save();
    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

Attention : il est possible que le fichier utilisé lors de la sauvegarde ne soit pas celui utilisé pour lire la configuration si c'est uniquement le nom du fichier qui est fourni en paramètre du constructeur. La méthode getFile() permet de connaitre le chemin du fichier utilisé.

Il est possible de fournir en paramètre de la méthode save() le nom du fichier pour par exemple écrire dans un autre fichier.

Exemple :

```
package fr.jmdoudoux.dej.commons.configuration;

import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.PropertiesConfiguration;

public class TestPropertiesConfiguration {

  public static void main(final String[] args) {
    try {
      final PropertiesConfiguration config =
        new PropertiesConfiguration("maConfig.properties");
      System.out.println("ma.valeur=" + config.getLong("ma.valeur"));
      config.setProperty("ma.valeur", 100);
      config.save("maConfig.properties.bck");
    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

Certains caractères doivent être échappés dans le fichier properties :

- le retour chariot : \n
- la tabulation : \t
- le caractère antislash : \\
- les caractères Unicode : \u20ac
- le séparateur des éléments de listes de valeurs (virgule par défaut) : \,

L'échappement des caractères peut devenir délicat dans certaines circonstances.

Exemple :

```
config.sources = c:\\config\\\\,c:\\data\\
config.urls = \\\\\\\\host1,\\\\\\\\host2
```

Dans ce cas, il est plus lisible de définir chacune des valeurs sur une ligne dédiée.

Résultat :

```
config.sources = c:\\config\\
config.sources = c:\\data\\
config.urls = \\\\host1
config.urls = \\\\host2
```

Une instance de type PropertiesConfigurationLayout est associée aux objets de type PropertiesConfiguration. Le rôle de la classe PropertiesConfigurationLayout est de préserver autant que possible la structure originale du fichier de properties qui a été chargé lorsque la configuration est sauvegardée. Par exemple, elle tente de conserver les commentaires et les lignes blanches.

La configuration par défaut de cet objet tente de préserver au mieux la structure du fichier .properties : plusieurs propriétés peuvent être modifiées pour des besoins spécifiques.

115.1.2.1.2. Les fichiers properties au format XML

Depuis la version 5.0 de Java, il est possible de stocker les paires clé/valeur encapsulées dans une instance de type properties dans un fichier XML en utilisant la méthode loadFromXML().

La classe XMLPropertiesConfiguration propose un support d'un fichier XML contenant les données d'un Properties comme source de données.

Dans ce cas, le format du fichier XML est imposé par les spécifications de l'API Java dans une DTD.

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- DTD for properties -->
<!ELEMENT properties ( comment?, entry* ) >
<!ATTLIST properties version CDATA #FIXED "1.0">
<!ELEMENT comment (#PCDATA) >
<!ELEMENT entry (#PCDATA) >
<!ATTLIST entry key CDATA #REQUIRED>
```

Le fichier XML minimal est donc :

Exemple :

```
<?xml version="1.0"?>
<!DOCTYPE properties SYSTEM «http://java.sun.com/dtd/properties.dtd»>
<properties>
</properties>
```

Chaque paire de clé/valeur est définie dans un tag <entry>. Le nom de la clé est fourni dans l'attribut key et la valeur est fournie dans le corps du tag.

Exemple :

```
<?xml version="1.0"?>
<!DOCTYPE properties SYSTEM «http://java.sun.com/dtd/properties.dtd»>
<properties>
  <entry key="cle">Valeur</entry>
</properties>
```

Les commentaires sont définis en utilisant le tag <comment>.

L'utilisation de la classe XMLPropertiesConfiguration requiert les bibliothèques Commons Digester, Commons BeanUtils et un parser XML comme dépendances.

115.1.2.1.3. Les fichiers XML

La classe org.apache.commons.configuration.XMLConfiguration encapsule une configuration dont la source est un fichier XML.

La structure du fichier XML est libre : elle n'est pas vérifiée à partir d'une DTD ou d'un schéma XML. Le nom des clés est déterminé en utilisant le nom de chaque tag père (sauf le tag racine) et le nom du tag lui-même séparés par un caractère point.

L'utilisation de la classe XMLConfiguration requiert deux dépendances du projet apache Commons : BeanUtils et JxPath.

Exemple avec Maven :

```
<dependencies>
```

```
    <dependency>
        <groupId>commons-configuration</groupId>
        <artifactId>commons-configuration</artifactId>
        <version>1.8</version>
    </dependency>
    <dependency>
        <groupId>commons-beanutils</groupId>
        <artifactId>commons-beanutils</artifactId>
        <version>1.8.0</version>
    </dependency>
    <dependency>
        <groupId>commons-jxpath</groupId>
        <artifactId>commons-jxpath</artifactId>
        <version>1.3</version>
    </dependency>
</dependencies>
```

La classe XMLConfiguration lit la configuration à partir d'un fichier XML et mappe la hiérarchie des tags sur la notation basée sur les points.

Par exemple : *<config><database><url> est mappé sur database.url.*

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
  <database>
    <url>127.0.0.1</url>
    <port>3306</port>
    <login>admin</login>
    <password>password</password>
  </database>
</config>
```

L'utilisation de ce fichier de configuration requiert la création d'une instance de type XMLConfiguration et l'utilisation de la notation par points pour obtenir les valeurs.

Exemple :

```
XMLConfiguration config = new XMLConfiguration("dbConfig.xml");
String url = config.getString("database.url");
String port = config.getString("database.port");
```

Il est possible d'accéder à un élément fils possédant plusieurs occurrences en utilisant son index.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
  <databases>
    <database>
      <env>dev</env>
      <url>127.0.0.1</url>
      <port>3306</port>
      <login>admin</login>
      <password>password</password>
    </database>
    <database>
      <env>test</env>
      <url>192.13.24.200</url>
      <port>3306</port>
      <login>admin</login>
      <password>password</password>
    </database>
  </database>
</config>
```

L'index du premier élément vaut 0. L'index à utiliser doit être indiqué entre parenthèses.

Exemple :

```
XMLConfiguration config = new XMLConfiguration("dbConfig.xml");
String url = config.getString("databases.database(1).url");

String port = config.getString("databases.database(1).port");
```

La méthode getMaxIndex(), héritée de la classe org.apache.commons.HierarchicalConfiguration, qui attend en paramètre le nom d'une clé renvoie le nombre d'éléments présents dans le fichier de configuration.

Il est possible de récupérer la valeur à partir d'un attribut d'un tag en utilisant une syntaxe particulière dans laquelle le nom de l'attribut précédé d'un arobase est indiqué entre crochets.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
    <database url="127.0.0.1" port="3306" login="admin" password="password"/>
</config>
```

Exemple :

```
XMLConfiguration config = new XMLConfiguration("dbConfig.xml");
String url = config.getString("databases.database.[@url]");
String port = config.getString("databases.database.[@port]");
```

La notation par point pour accéder à un élément de la configuration fonctionne bien pour des cas simples mais elle s'avère limitative pour des cas plus complexes. Pour ces cas, il est possible d'utiliser XPath en associant à la configuration une instance de type XPathExpressionEngine.

Exemple :

```
XMLConfiguration config = new XMLConfiguration("dbConfig.xml");
config.setExpressionEngine(new XPathExpressionEngine());
config.getString("databases/database[env = 'test']/url");
config.getString("databases/database[env = 'test']/port");
```

Les commentaires du fichier XML d'origine qui a été utilisé pour charger la configuration sont préservés lors de la sauvegarde d'une configuration modifiée.

115.1.2.2. Les configurations dans la base de données

La classe org.apache.commons.configuration.DatabaseConfiguration encapsule une configuration stockée dans une base de données.

La table qui va stocker des données de la configuration doit contenir deux colonnes : une pour la clé et une autre pour la valeur. Une troisième colonne optionnelle peut être utilisée pour contenir le nom de la configuration : ceci permet de stocker plusieurs configurations dans la table.

Plusieurs constructeurs de la classe DatabaseConfiguration permettent de fournir les informations utiles pour accéder aux informations dans la table :

- DataSource datasource : connexion à la base de données
- String table : nom de la table
- String nameColumn : nom de la colonne qui contient le nom de la configuration

- String keyColumn : nom de la colonne qui contient la clé
- String valueColumn : nom de la colonne qui contient la valeur
- String name : le nom de la configuration

Un accès à la base de données est réalisé à chaque demande de la valeur d'une clé, ce qui peut engendrer de nombreux accès à la base

115.1.2.3. Les configurations dans une instance de type Map

La classe org.apache.commons.configuration.MapConfiguration encapsule une configuration dont la source est une collection existante de type Map.

Les données contenues dans la Map doivent respecter certaines conventions pour être utilisables comme source de la configuration, notamment que la clé soit de type String et que les valeurs ne soient pas de type List.

La classe MapConfiguration possède deux constructeurs : un premier qui attend en paramètre un objet de type Map<String, Object> et un second qui attend en paramètre un objet de type Properties.

Les clés de la configuration sont directement mappées sur les clés de la map : par exemple, la méthode getProperty() invoque directement la méthode get() sur l'instance de la Map.

Exemple :

```
package fr.jmdoudoux.dej.commons.configuration;

import java.util.HashMap;
import java.util.Map;

import org.apache.commons.configuration.MapConfiguration;

public class TestMapConfiguration {

  public static void main(final String[] args) {
    final Map<String, Object> donnees = new HashMap<String, Object>();
    donnees.put("valeurs", new Object[] { "valeur1", "valeur2", "valeur3" });
    donnees.put("maCle", "maValeur");

    final MapConfiguration config = new MapConfiguration(donnees);
    System.out.println("valeurs=" + config.getList("valeurs"));
    System.out.println("maCle=" + config.getString("maCle"));
  }
}
```

Comme l'instance de type Map fournie au constructeur est utilisée pour stocker les valeurs, c'est le type de cette instance qui détermine si les opérations sont thread-safe ou non.

115.1.2.4. JNDI

La classe org.apache.commons.configuration.JNDIConfiguration utilise les données d'un service de nommage accessible grâce à JNDI comme source pour la configuration.

La propriété context permet de préciser le contexte JNDI pour accéder à la ressource.

La propriété prefix permet de préciser le préfixe à utiliser dans le Context lors des recherches.

La classe JNDIConfiguration possède quatre constructeurs : le constructeur par défaut et trois surcharges qui permettent de préciser le Context et/ou le préfixe.

Les valeurs sont recherchées dans la Context JNDI en utilisant le préfixe et la valeur de la clé.

La configuration encapsulée dans un JNDIConfiguration est en lecture seule. Les méthodes addPropertyDirect() et setProperty() lèvent une exception de type UnsupportedOperationException.

La méthode getBaseContext() renvoie le Context JNDI pour lequel le préfixe est appliqué.

115.1.2.5. Les variables systèmes

La classe org.apache.commons.configuration.SystemConfiguration utilise comme source de la configuration les propriétés système et les propriétés de la JVM fournies avec l'option -Dcle=valeur.

Exemple :

```
package fr.jmdoudoux.dej.commons.configuration;

import org.apache.commons.configuration.EnvironmentConfiguration;

public class TestEnvironmentConfiguration {
  public static void main(final String[] args) {
    final EnvironmentConfiguration config = new EnvironmentConfiguration();
    System.out.println("java home=" + config.getString("JAVA_HOME"));
    System.out.println("os=" + config.getString("OS"));
  }
}
```

115.1.3. Les beans dynamiques

La classe org.apache.commons.configuration.beanutils.ConfigurationDynaBean permet d'encapsuler une configuration et de l'utiliser comme un DynaBean. Un DynaBean permet d'obtenir et de modifier les propriétés d'un bean de manière dynamique.

Pour créer une instance de type ConfigurationDynaBean, il suffit d'invoquer son constructeur en lui passant en paramètre la configuration.

Exemple :

```
package fr.jmdoudoux.dej.commons.configuration;

import java.util.HashMap;
import java.util.Map;

import org.apache.commons.configuration.MapConfiguration;
import org.apache.commons.configuration.beanutils.ConfigurationDynaBean;

public class TestConfigurationDynaBean {

  public static void main(final String[] args) {
    final Map<String, Object> donnees = new HashMap<String, Object>();
    donnees.put("valeurs", new Object[] { "valeur1", "valeur2", "valeur3" });

    final MapConfiguration config = new MapConfiguration(donnees);

    final ConfigurationDynaBean bean = new ConfigurationDynaBean(config);
    final Object value = bean.get("valeurs", 1);
    System.out.println(value);
  }
}
```

115.1.4. Les configurations composites

Il est fréquent que la configuration provienne de plusieurs sources : la classe org.apache.commons.configuration.CompositeConfiguration permet d'agréger plusieurs sources pour obtenir une configuration avec un seul point d'entrée.

La contrepartie est qu'une fois la configuration chargée, il n'est plus possible d'utiliser les spécificités de chacune des configurations qui sont agrégées : seules les fonctionnalités communes définies par l'interface Configuration sont utilisables.

La méthode addConfiguration() permet d'ajouter une configuration à la composition.

Exemple :

```
CompositeConfiguration config = new CompositeConfiguration();
config.addConfiguration(new SystemConfiguration());
config.addConfiguration(new PropertiesConfiguration("monApp.properties"));
```

Une propriété demandée à une composition est recherchée successivement dans les configurations dans leur ordre d'ajout. Dès que la propriété est trouvée dans une configuration, sa valeur est retournée. Ce mode de fonctionnement permet de mettre en place un mécanisme de redéfinition de valeurs pour une même propriété.

Exemple : le fichier maConfig.properties

```
ma.valeur=100
ma.valeur2=200
```

Exemple : le fichier maConfigDefaut.properties

```
ma.valeur=10
ma.valeur2=20
```

Exemple :

```
package fr.jmdoudoux.dej.commons.configuration;

import org.apache.commons.configuration.CompositeConfiguration;
import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.PropertiesConfiguration;

public class TestCompositeConfiguration {

  public static void main(final String[] args) {
    final CompositeConfiguration config = new CompositeConfiguration();
    try {
      config.addConfiguration(new PropertiesConfiguration("maConfig.properties"));
      config.addConfiguration(new PropertiesConfiguration("maConfigDefaut.properties"));

      System.out.println("ma.valeur=" + config.getString("ma.valeur"));
    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
ma.valeur=100
```

Une CompositeConfiguration est donc particulièrement utile pour gérer une hiérarchie de configuration. Comme une propriété est recherchée dans l'ordre dans lequel les configurations ont été ajoutées à la composition, il suffit que la dernière composition contienne les valeurs par défaut. Si la propriété n'est pas redéfinie dans les autres configurations, c'est celle obtenue de la configuration par défaut qui sera retournée.

Exemple : le fichier maConfig.properties

```
ma.valeur2=200
```

Exemple : le fichier maConfigDefaut.properties

```
ma.valeur=10
ma.valeur2=20
```

Résultat :

```
ma.valeur=10
```

Si une propriété de la composition doit pouvoir être modifiée et sauvegardée, il est nécessaire de fournir une configuration en paramètre du constructeur utilisé pour créer l'instance de type CompositionConfiguration.

Toutes les modifications faites dans la configuration sont réalisées dans la source précisée en paramètre du constructeur (in memory configuration). La méthode getInMemoryConfiguration() retourne l'instance qui encapsule cette source.

115.1.5. Les configurations hiérarchiques

Plusieurs sources de configuration utilisent une structure hiérarchique ou arborescente pour organiser leurs données, par exemple celles utilisant XML. Ce type de configuration est encapsulé dans une classe qui hérite de la classe HierarchicalConfiguration.

Plusieurs configurations sont de type hiérarchique :

- AbstractHierarchicalFileConfiguration : HierarchicalINIConfiguration, MultiFileHierarchicalConfiguration, PatternSubtreeConfigurationWrapper, PropertyListConfiguration, XMLConfiguration, XMLPropertyListConfiguration
- CombinedConfiguration
- SubnodeConfiguration

Les éléments imbriqués sont désignés en utilisant la notation par point et en ignorant l'élément racine.

Pour accéder à un attribut d'un tag XML, il faut utiliser une syntaxe proche de XPath :

- cle(index) : permet d'accéder à l'élément précisé par index. Le premier élément possède l'index 0
- [@nom_attribut] : permet d'accéder à la valeur de l'attribut dont le nom est nom_attribut

Exemple :

```
package fr.jmdoudoux.dej.commons.configuration;

import java.util.List;

import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.HierarchicalConfiguration;
import org.apache.commons.configuration.XMLConfiguration;
import org.apache.commons.configuration.reloading.FileChangedReloadingStrategy;

public class TestXMLConfiguration {

  public static void main(final String[] args) {
    try {
      final XMLConfiguration config = new XMLConfiguration("maConfig.xml");
     System.out.println(config.getList("environnements.environnement(0).champs.champ"));
      System.out.println(config.getList("paire"));
      System.out.println(config.getString("environnements.environnement(1)[@nom]"));

    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
[local1, local2, local3]
[2, 4, 6, 8]
test
```

Parfois le nom d'une clé peut être long si l'arborescence de la configuration est complexe. Pour faciliter l'utilisation d'une portion de la configuration, il est possible d'utiliser la méthode configurationAt() . Elle renvoie une instance de type HierarchicalConfiguration qui encapsule la portion de la configuration sous la clé fournie en paramètre.

Exemple :

```
package fr.jmdoudoux.dej.commons.configuration;

import java.util.List;

import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.HierarchicalConfiguration;
import org.apache.commons.configuration.XMLConfiguration;
import org.apache.commons.configuration.reloading.FileChangedReloadingStrategy;

public class TestXMLConfiguration {

  public static void main(final String[] args) {
    try {
      final XMLConfiguration config = new XMLConfiguration("maConfig.xml");

      final HierarchicalConfiguration sub =
        config.configurationAt("environnements.environnement(1).champs");
      final List<Object> champs = sub.getList("champ");
      System.out.println(champs);

    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
[test1, test2, test3]
```

Cette syntaxe est utilisable aussi pour modifier la configuration. Pour ajouter un nouvel élément dans une liste en première position, il faut utiliser un index avec la valeur -1.

Il est nécessaire d'échapper certains caractères dans le nom des clés ou dans les valeurs.

Par défaut, le caractère point est utilisé comme séparateur entre les différents éléments qui composent la clé. Cependant le caractère point peut être utilisé dans le nom d'un tag XML. Dans ce cas, une exception de type NoSuchElementException est levée.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
  <ma.valeur>100</ma.valeur>
</config>
```

Exemple :

```
package fr.jmdoudoux.dej.commons.configuration;

import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.XMLConfiguration;
import org.apache.commons.configuration.reloading.FileChangedReloadingStrategy;

public class TestXMLConfiguration {

  public static void main(final String[] args) {
    try {
      final XMLConfiguration config = new XMLConfiguration("maConfig.xml");
      System.out.println("ma.valeur=" + config.getLong("ma.valeur"));
    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
Exception in thread "main" java.util.NoSuchElementException: 'ma.valeur' doesn't map
to an existing object
        at org.apache.commons.configuration.AbstractConfiguration.getLong(Abstract
Configuration.java:862)
        at fr.jmdoudoux.dej.commons.configuration.TestXMLConfiguration.main(Test
XMLConfiguration.java:16)
```

Pour changer ce comportement, il faut doubler le caractère point dans le nom de la clé pour déspécialiser le caractère point.

Exemple :

```
package fr.jmdoudoux.dej.commons.configuration;

import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.XMLConfiguration;
import org.apache.commons.configuration.reloading.FileChangedReloadingStrategy;

public class TestXMLConfiguration {

  public static void main(final String[] args) {
    try {
      final XMLConfiguration config = new XMLConfiguration("maConfig.xml");
      System.out.println("ma.valeur=" + config.getLong("ma..valeur"));
    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

Par défaut, la virgule est utilisée comme séparateur pour une liste de valeurs dans le corps d'un tag.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
  <mavaleur>100,200</mavaleur>
</config>
```

Exemple :

```
package fr.jmdoudoux.dej.commons.configuration;

import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.XMLConfiguration;
import org.apache.commons.configuration.reloading.FileChangedReloadingStrategy;

public class TestXMLConfiguration {

  public static void main(final String[] args) {
    try {
      final XMLConfiguration config = new XMLConfiguration("maConfig.xml");
      System.out.println("mavaleur=" + config.getLong("mavaleur"));
    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
Exception in thread "main" java.util.NoSuchElementException: 'ma.valeur' doesn't map
to an existing object
        at org.apache.commons.configuration.AbstractConfiguration.getLong(Abstract
Configuration.java:862)
        at fr.jmdoudoux.dej.commons.configuration.TestXMLConfiguration.main(Test
XMLConfiguration.java:16)
```

Par défaut, le caractère virgule est utilisé comme séparateur de plusieurs éléments d'une valeur.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
  <mavaleur>100,123456789</mavaleur>
</config>
```

Exemple :

```
package fr.jmdoudoux.dej.commons.configuration;

import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.XMLConfiguration;
import org.apache.commons.configuration.reloading.FileChangedReloadingStrategy;

public class TestXMLConfiguration {

  public static void main(final String[] args) {
    try {
      final XMLConfiguration config = new XMLConfiguration("maConfig.xml");
      System.out.println("mavaleur=" + config.getString("mavaleur"));
    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
mavaleur=100
```

Si les clés ne doivent pas être considérées comme pouvant avoir plusieurs valeurs, il faut désactiver la fonctionnalité en invoquant les méthodes setDelimiterParsingDisabled() et setAttributeSplittingDisabled() avec la valeur true en paramètre.

La désactivation de cette fonctionnalité doit impérativement se faire avant le chargement du fichier XML.

Exemple :

```
package fr.jmdoudoux.dej.commons.configuration;

import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.XMLConfiguration;
import org.apache.commons.configuration.reloading.FileChangedReloadingStrategy;

public class TestXMLConfiguration {

  public static void main(final String[] args) {
    try {
      final XMLConfiguration config = new XMLConfiguration();

      config.setDelimiterParsingDisabled(true);
      config.setAttributeSplittingDisabled(true);
      config.load("maConfig.xml");

      System.out.println("mavaleur=" + config.getString("mavaleur"));

    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
mavaleur=100,123456789
```

Pour les clés qui doivent avoir plusieurs valeurs, il faut alors utiliser un tag pour chaque valeur dans le fichier de configuration XML.

Pour des besoins plus complexes, il est possible d'utiliser une instance de type ExpressionEngine. Un objet de type ExpressionEngine est responsable de l'interprétation des clés dans une source de la configuration.

La méthode setExpressionEngine() permet de préciser une instance de type ExpressionEngine à utiliser. Les clés utilisées doivent alors respecter la syntaxe exploitable par l'ExpressionEngine.

La syntaxe utilisée par défaut est définie dans une instance de type DefaultExpressionEngine.

La classe DefaultExpressionEngine possède plusieurs attributs :

- AttributStart : le marqueur de début d'attribut
- AttributEnd : le marqueur de fin d'attribut
- IndexStart : le marqueur de début d'index
- IndexEnd : le marqueur de fin d'index
- PropertyDelimiter : le délimiteur de propriétés

La méthode setDefaultExpressionEngine() de la classe HierarchicalConfiguration permet de modifier l'ExpressionEngine utilisé par défaut.

Exemple :

```
package fr.jmdoudoux.dej.commons.configuration;

import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.HierarchicalConfiguration;
import org.apache.commons.configuration.XMLConfiguration;
import org.apache.commons.configuration.tree.DefaultExpressionEngine;

public class TestDefaultExpressionEngine {

  public static void main(final String[] args) {
    final DefaultExpressionEngine engine = new DefaultExpressionEngine();

    engine.setPropertyDelimiter("/");
    engine.setIndexStart("{");
    engine.setIndexEnd("}");
    engine.setAttributeStart("@");
    engine.setAttributeEnd(null);

    HierarchicalConfiguration.setDefaultExpressionEngine(engine);

    final XMLConfiguration config = new XMLConfiguration();
    try {
      config.load("maConfig.xml");
      System.out.println(config.getString("environnements/environnement{0}/champs/champ"));
      System.out.println(config.getString("environnements/environnement{0}@name"));
    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
local1
null
```

La classe XPathExpressionEngine encapsule un ExpressionEngine qui exploite la syntaxe XPath.

Son utilisation requiert que la bibliothèque Commons JXPath soit ajoutée au classpath.

Exemple :

```
package fr.jmdoudoux.dej.commons.configuration;

import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.XMLConfiguration;
import org.apache.commons.configuration.tree.xpath.XPathExpressionEngine;

public class TestXPathExpressionEngine {

  public static void main(final String[] args) {
    try {
      final XMLConfiguration config = new XMLConfiguration();
      config.setExpressionEngine(new XPathExpressionEngine());
      config.load("maConfig.xml");

      System.out.println(config.getList(
        "environnements/environnement[@nom='test']/champs/champ"));

    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
[test1, test2, test3]
```

Il est possible de valider le fichier XML grâce à sa DTD en passant la valeur true à la méthode setValidating().

Exemple :

```
package fr.jmdoudoux.dej.commons.configuration;

import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.XMLConfiguration;
import org.apache.commons.configuration.reloading.FileChangedReloadingStrategy;

public class TestXMLConfiguration {

  public static void main(final String[] args) {
    try {
      final XMLConfiguration config = new XMLConfiguration();
      config.setValidating(true);
      config.load("maConfig.xml");
    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

Il est possible de valider le fichier XML grâce à son schéma en passant la valeur true à la méthode setSchemaValidating().

Exemple :

```
package fr.jmdoudoux.dej.commons.configuration;

import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.XMLConfiguration;
import org.apache.commons.configuration.reloading.FileChangedReloadingStrategy;

public class TestXMLConfiguration {

  public static void main(final String[] args) {
    try {
      final XMLConfiguration config = new XMLConfiguration();
      config.setSchemaValidating(true);
      config.load("maConfig.xml");
    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

115.1.6. La classe CombinedConfiguration

La classe CombinedConfiguration permet de combiner plusieurs sources de configuration. Elle est similaire à la classe CompositeConfiguration avec plusieurs différences :

- c'est une configuration hiérarchique (HierarchicalConfiguration)
- la façon dont la combinaison est faite est configurable en utilisant une instance de type NodeCombiner
- un nom peut être assigné à chaque source pour permettre leur accès
- chaque source peut avoir un préfixe qui sera ajouté à chacune de ses propriétés

La classe CombinedConfiguration propose une vue logique des propriétés des sources de configuration qui lui sont associées. Cette vue est réalisée grâce à l'utilisation d'une instance de type NodeCombiner.

Si une des configurations sources est modifiée alors la vue est invalidée et reconstruite par le NodeCombiner si nécessaire.

Plusieurs classes filles de type NodeCombiner sont proposées en standard :

- OverrideCombiner
- MergeCombiner
- UnionCombiner

La méthode addConfiguration() permet d'ajouter une Configuration à la combinaison. Une surcharge attend aussi en paramètre un nom. Ce nom peut être utilisé en paramètre de la méthode getConfiguration() pour obtenir la configuration correspondante.

La méthode addListNode() permet de préciser qu'un élément doit être considéré comme une liste.

Les exemples de cette section vont utiliser deux fichiers de configurations qui seront combinés.

Exemple : Le fichier configUsers.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <database>
    <tables>
      <table>
        <name>users</name>
        <fields>
          <field>
```

```
            <name>user_id</name>
          </field>
          <field>
            <name>user_name</name>
          </field>
          </fields>
      </table>
    </tables>
  </database>
</configuration>
```

Exemple : le fichier configGroups.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <database>
    <tables>
      <table>
        <name>groups</name>
        <fields>
          <field>
            <name>group_id</name>
          </field>
          <field>
            <name>user_id</name>
          </field>
        </fields>
      </table>
    </tables>
  </database>
</configuration>
```

La classe MergeCombiner implémente une combinaison par fusion.

Exemple :

```
package fr.jmdoudoux.dej.commons.configuration;

import org.apache.commons.configuration.CombinedConfiguration;
import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.XMLConfiguration;
import org.apache.commons.configuration.tree.MergeCombiner;
import org.apache.commons.configuration.tree.NodeCombiner;

public class TestCombinedConfiguration {

  public static void main(final String[] args) {
    try {
      final XMLConfiguration confUsers = new XMLConfiguration("configUsers.xml");
      final XMLConfiguration confGroups = new XMLConfiguration("configGroups.xml");

      final NodeCombiner combiner = new MergeCombiner();

      final CombinedConfiguration cc = new CombinedConfiguration(combiner);
      cc.addConfiguration(confUsers, "users");
      cc.addConfiguration(confGroups);

      System.out.println(cc.getList("database.tables.table.name"));
      System.out.println(cc.getList("database.tables.table.fields.field.name"));

    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
[users]
[user_id, user_name]
```

La classe UnionCombiner implémente une combinaison par union. Les éléments des différentes sources sont ajoutés dans la vue.

Exemple :

```
package fr.jmdoudoux.dej.commons.configuration;

import org.apache.commons.configuration.CombinedConfiguration;
import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.XMLConfiguration;
import org.apache.commons.configuration.tree.NodeCombiner;
import org.apache.commons.configuration.tree.UnionCombiner;

public class TestCombinedConfiguration {

  public static void main(final String[] args) {
    try {
      final XMLConfiguration confUsers = new XMLConfiguration("configUsers.xml");
      final XMLConfiguration confGroups = new XMLConfiguration("configGroups.xml");

      final NodeCombiner combiner = new UnionCombiner();

      final CombinedConfiguration cc = new CombinedConfiguration(combiner);
      cc.addConfiguration(confUsers, "users");
      cc.addConfiguration(confGroups);

      System.out.println(cc.getList("database.tables.table.name"));
      System.out.println(cc.getList("database.tables.table.fields.field.name"));

    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
[users, groups]
[user_id, user_name, group_id, user_id]
```

La classe OverrideCombiner implémente une combinaison par surcharge : les éléments de la première configuration sont prépondérants sur ceux des configurations suivantes. Un élément de la seconde configuration ne peut être inclus dans la combinaison que s'il n'existe pas dans la première configuration.

Il n'est pas recommandé de faire des modifications directement dans une instance de type CombinedConfiguration même si rien ne l'empêche. Du fait de l'utilisation du NodeCombiner le résultat peut être inattendu : il est donc préférable de faire les modifications dans les configurations qui sont contenues dans la CombinedConfiguration.

115.1.7. Les configurations dynamiques

Les configurations dynamiques permettent de créer une instance de type CompositeConfiguration dont les différentes sources sont précisées dans un fichier XML.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- config.xml -->
<configuration>

  <env />
  <xml fileName="config.xml" />
  <properties fileName="config.properties" />
</configuration>
```

Le tag racine est le tag <configuration>.

Chacune des sources qui doivent composer la configuration est précisée grâce à un tag dédié :

- <env> :
- <xml> : La propriété fileName permet de préciser le fichier XML
- <properties> : La propriété fileName permet de préciser le fichier .properties

Ces tags peuvent être directement des fils du tag racine ou d'un tag <override>.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- config.xml -->
<configuration>
  <override>
    <env />
    <xml fileName="config.xml" />
    <properties fileName="config.properties"/>
  </override>
</configuration>
```

Pour utiliser cette fonctionnalité, il est nécessaire d'ajouter la bibliothèque Commons Digester dans le classpath.

Le format général du fichier de définition de la configuration est de la forme :

Exemple :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<configuration>
  <header> ...</header>
  <override> ... </override>
  <additional> ... </additional>
</configuration>
```

Tous les tags fils <header>, <override> et <additional> sont optionnels.

La déclaration des différences sources de configuration à utiliser se fait en utilisant des tags dédiés par type de source :

- properties : ajouter un fichier .properties dont le nom est précisé grâce au tag fileName. Si le nom du fichier se termine par .xml alors c'est une instance de type XMLPropertiesConfiguration qui est utilisée sinon c'est une instance de type PropertiesConfiguration
- xml : ajouter un fichier .xml dont le nom est précisé grâce au tag fileName. C'est une instance de type XMLConfiguration qui est utilisée
- jndi : ajouter une ressource JNDI. L'attribut prefix permet de préciser la racine dans l'arborescence JNDI. C'est une instance de type JNDICOnfiguration qui est utilisée
- plist : ajouter un fichier de type .plist dont le nom est précisé grâce au tag fileName. Si le fichier est au format xml alors c'est une instance de type XMLPropertyListConfiguration qui est utilisée sinon c'est une instance de type PropertyListConfiguration
- system : ajouter un accès aux propriétés systèmes. C'est une instance de type SystemConfiguration qui est utilisée
- ini : ajouter un fichier de type .ini. C'est une instance de type HierarchicalINIConfiguration qui est utilisée
- env : ajouter un accès aux propriétés d'environnement

Pour chaque source, il est possible de modifier des propriétés :

Exemple :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<configuration>
  <system />
  <properties fileName="maConfig.properties" throwExceptionOnMissing="true">
    <reloadingStrategy refreshDelay="60000" config-class=
      "org.apache.commons.configuration.reloading.FileChangedReloadingStrategy" />
  </properties>
  <properties fileName="maConfigDefaut.properties" />
</configuration>
```

L'attribut config-name permet de définir un nom pour la configuration.

Il est possible d'effectuer un remplacement dynamique de valeurs avec celles de variables système en précisant le nom de la variable entre ${ et }. Dans ce cas, il est nécessaire que ces variables soient chargées en utilisant une source <system>.

Exemple :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<configuration>
  <system/>
  <properties fileName="${CONFIG_FILE}"/>
</configuration>
```

La recherche d'une propriété se fait dans l'ordre de déclaration des sources dans le fichier XML.

Par défaut, si une source définie dans le fichier de configuration n'est pas trouvée lors du chargement, alors une exception de type ConfigurationException est levée. Il est possible de définir une source optionnelle en utilisant l'attribut config-optional avec la valeur true.

Exemple :

```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration>
  <properties fileName="user.properties" config-optional="true"/>
  <properties fileName="default.properties"/>
</configuration>
```

Si la source est configurée comme optionnelle et qu'elle n'est pas trouvée à son chargement alors aucune exception n'est levée est un message de type warning est inséré dans le log.

Il est possible d'unir plusieurs sources de configuration comme si ce n'était qu'une seule source. Les différentes sources doivent être définies en tant que tag fils du tag <additional>. L'attribut config-at permet de préciser avec quelle source elles doivent être fusionnées.

Par défaut, le tag <override> peut être omis sauf si un tag <additional> est présent.

115.1.7.1. La classe ConfigurationFactory

La classe org.apache.commons.configuration.ConfigurationFactory permet de construire des configurations dynamiques.

Exemple : le fichier config.xml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<configuration>
  <system/>
  <properties fileName="maConfig.properties"/>
  <properties fileName="maConfigDefaut.properties"/>
</configuration>
```

Exemple :

```
package fr.jmdoudoux.dej.commons.configuration;
```

```
import org.apache.commons.configuration.Configuration;
import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.ConfigurationFactory;

public class TestConfigurationFactory {

  public static void main(final String[] args) {

    final ConfigurationFactory factory = new ConfigurationFactory("config.xml");
    try {
      final Configuration config = factory.getConfiguration();

      System.out.println("ma.valeur=" + config.getString("ma.valeur"));
    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

A partir de la version 1.3, la classe ConfigurationFactory est deprecated : il faut utiliser la classe DefaultConfigurationBuilder.

115.1.7.2. La classe DefaultConfigurationBuilder

La classe DefaultConfigurationBuilder permet de créer une configuration à partir des informations fournies dans un fichier XML.

Le plus simple est de créer une instance de type DefaultConfigurationBuilder. Cette classe propose plusieurs constructeurs dont certains permettent de préciser le fichier de configuration.

La méthode setFile() permet de préciser le fichier de configuration si celui-ci n'a pas été fourni en paramètre du constructeur.

La méthode getConfiguration() permet de charger la configuration en créant une instance de type Configuration.

Une surcharge de la méthode getConfiguration(), qui attend en paramètre un booléen précisant le fichier XML devant être lu, renvoie une instance de type CombinedConfiguration.

Exemple :

```
DefaultConfigurationBuilder builder = new DefaultConfigurationBuilder("config.xml");
CombinedConfiguration config = builder.getConfiguration(true);
```

115.1.8. Les sous-ensembles d'une configuration

Il est possible d'utiliser un sous-ensemble d'une configuration qui correspond aux éléments commençant par un même préfixe.

Il y a deux façons d'obtenir un sous-ensemble :

- en créant une instance de type SubsetConfiguration dont les constructeurs attendent en paramètre la configuration source et le préfixe
- en invoquant la méthode subset() qui attend en paramètre le préfixe et renvoie une instance de type SubsetConfiguration

Une instance de type SubsetConfiguration implémente l'interface Configuration. Il faut retirer le préfixe qui est contenu dans la configuration originale pour accéder aux éléments correspondants dans le sous-ensemble.

Exemple :

```
package fr.jmdoudoux.dej.commons.configuration;

import org.apache.commons.configuration.Configuration;
import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.XMLConfiguration;

public class TestSubsetConfiguration {

  public static void main(final String[] args) {
    try {
      final XMLConfiguration config = new XMLConfiguration("maConfig.xml");
      String url = config.getString("data.database.url");
      System.out.println("url = " + url);

      final Configuration dbConfig = config.subset("data.database");
      url = dbConfig.getString("url");
      System.out.println("url = " + url);

    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
url = 127.0.0.1
url = 127.0.0.1
```

115.1.9. Les événements sur la configuration

La classe AbstractConfiguration propose des fonctionnalités pour enregistrer des listeners sur des événements qui permettent d'être notifié lors d'un changement sur des données de la configuration.

L'interface ConfigurationListener définit la méthode configurationChanged() qui sera invoquée si le listener est enregistré et qu'un événement de modification de la configuration est émis. La méthode configurationChanged() possède un paramètre de type ConfigurationEvent qui encapsule des informations sur la modification :

- source object : généralement la configuration qui a été modifiée
- event type : constante numérique qui précise le type d'événement (EVENT_ADD_PROPERTY, EVENT_SET_PROPERTY, EVENT_CLEAR_PROPERTY, EVENT_CLEAR définies dans la classe AbstractConfiguration, EVENT_RELOAD définie dans la classe AbstractFileConfiguration)
- le nom de la propriété concernée si elle est connue
- la valeur de la propriété si elle est connue

La méthode isBeforeUpdate() renvoie un booléen qui précise si l'événement est émis avant une modification (valeur true) ou après (valeur false).

Exemple :

```
package fr.jmdoudoux.dej.commons.configuration;

import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.PropertiesConfiguration;
import org.apache.commons.configuration.event.ConfigurationEvent;
import org.apache.commons.configuration.event.ConfigurationListener;

public class TestConfigurationListener {

  public static void main(final String[] args) {
    try {
      final PropertiesConfiguration config =
         new PropertiesConfiguration("maConfig.properties");
      config.addConfigurationListener(new ConfigurationListener() {

        @Override
        public void configurationChanged(final ConfigurationEvent event) {
          if (!event.isBeforeUpdate()) {
            System.out.println("Event: ");
            System.out.println("  Type = " + event.getType());
            System.out.println("  Source = " + event.getSource());
            if (event.getPropertyName() != null) {
              System.out.println("  Property name = " + event.getPropertyName());
            }
            if (event.getPropertyValue() != null) {
              System.out.println("  Property value = " + event.getPropertyValue());
            }
            System.out.println("");
          }
        }
      });

      System.out.println("ma.valeur=" + config.getLong("ma.valeur"));
      config.setProperty("ma.valeur", 100);
      System.out.println("ma.valeur=" + config.getLong("ma.valeur"));
      config.addProperty("nouvelle.valeur", "200");
    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }

}
```

Résultat :

```
ma.valeur=10
Event:
  Type = 3
  Source = org.apache.commons.configuration.PropertiesConfiguration@ca2dce
  Property name = ma.valeur
  Property value = 100

ma.valeur=100
Event:
  Type = 1
  Source = org.apache.commons.configuration.PropertiesConfiguration@ca2dce
  Property name = nouvelle.valeur
  Property value = 200
```

L'interface ConfigurationErrorListener définit la méthode configurationError() qui sera invoquée si le listener est enregistré et qu'une exception est levée par un sous-système utilisé par la configuration. La méthode configurationError() possède un paramètre de type ConfigurationErrorEvent qui encapsule l'exception. Celle-ci peut être obtenue en invoquant la méthode getCause() qui renvoie une instance de Throwable.

115.1.10. La classe ConfigurationUtils

La classe org.apache.commons.configuration.ConfigurationUtils est une classe proposant des utilitaires dont toutes les méthodes sont statiques.

Plusieurs méthodes concernent les configurations.

Méthode	Rôle
void append(Configuration source, Configuration cible)	Ajouter la configuration source dans la configuration cible. Les valeurs des clés existantes dans la configuration cible sont remplacées par les valeurs correspondantes dans la configuration source
Configuration cloneConfiguration(Configuration config)	Retourner un objet clone de la configuration

HierarchicalConfiguration convertToHierarchical(Configuration conf)	Convertir une configuration en configuration hiérarchique
HierarchicalConfiguration convertToHierarchical(Configuration conf, ExpressionEngine engine)	Convertir une configuration en configuration hiérarchique en utilisant l'ExpressionEngine fournie en paramètre
copy(Configuration source, Configuration cible)	Copier la configuration source dans la configuration cible

dump(Configuration conf, PrintStream out)
dump(Configuration conf, PrintWriter out)
	Envoyer un dump de la configuration, sous la forme clé=valeur dans le flux fourni en paramètre

void enableRuntimeExceptions(Configuration src)	Activer les exceptions de type Runtime pour la configuration
toString(Configuration conf)	Renvoyer un dump de la configuration, sous la forme clé=valeur

Plusieurs méthodes concernent les fichiers : getFile(), getURL(), locate(), fileFromURL().

115.2. Apache Commons CLI

La bibliothèque Commons CLI permet de facilement interpréter les différents arguments qui sont passés par la ligne de commande à une application standalone.

Ces arguments peuvent permettre de définir des options, fournir des valeurs, ... Leur exploitation n'est pas toujours facile car elles peuvent avoir un nom court et/ou long, une ou plusieurs valeurs, être obligatoires ou non, ...

La bibliothèque Commons CLI (Command Line Interface) permet d'analyser et d'obtenir les options passées au programme par la ligne de commande

Le site du projet est à l'url https://commons.apache.org/cli/

Commons CLI propose le support de plusieurs formats pour les options :

- style POSIX : une option est identifiée par une lettre. Chaque option peut être précédée d'un caractère tiret ou les options peuvent être concaténées devant un seul caractère tiret

(exemple : tar -xvf monarchive.tar)

- style GNU : le format long est précédé d'un double tiret

(exemple : tar --verbose --extract --file=monarchive.tar)

- Propriété Java

(exemple : -Dpropriete=valeur)

Les arguments passés à une ligne de commande peuvent être de deux natures :

- options ou flags : généralement préfixé par un caractère tiret ou deux caractères tirets (sous Unix) ou un caractère slash sous Windows
- valeurs d'un argument : les différentes valeurs sont concaténées avec un caractère de séparation, généralement un caractère virgule

Une option et sa valeur peuvent être concaténées avec un caractère de séparation, généralement un caractère égal.

Une option peut être associée à une ou plusieurs valeurs.

Les traitements relatifs aux options de la ligne de commandes avec CLI se font en trois étapes :

- définition des options
- analyse des options fournies en paramètres du programme
- obtention des options et de leurs valeurs

Commons CLI permet aussi d'afficher un résumé des options, ce qui est particulièrement utile notamment notamment pour fournir une aide sur les options utilisables quand l'analyse échoue.

CLI ne permet pas de gérer certains cas complexes ou particuliers comme par exemple :

- le fait que la valeur soit concaténée à l'option (exemple : java -Xmx128m)
- s'assurer de la cohérence de la présence d'autres options lorsqu'une option particulière est présente
- vérifier et convertir au besoin la ou les valeurs associées à une option

La version utilisée dans cette section est la 1.2 : elle peut être téléchargée à l'url :
https://commons.apache.org/cli/download_cli.cgi

115.2.1. La définition des options

La classe Options est un conteneur qui va encapsuler les différentes instances de type org.apache.commons.cli.Option.

Pour créer une nouvelle instance de type org.apache.commons.cli.Options, il suffit d'invoquer son constructeurs sans arguments.

Une option est encapsulée dans une instance type Option qui possède plusieurs propriétés :

Propriété	Rôle
String opt	Nom court de l'option
String longOpt	Nom long de l'option
String description	Description de l'option
Boolean required	Définir si l'option est obligatoire
Boolean arg	Définir si l'option possède un argument obligatoire
Boolean args	Définir si l'option peut avoir plusieurs arguments
Boolean optionalArg	Définir si l'option possède un argument optionnel
String argName	Nom de l'argument
Char valueSeparatorChar	Définir le caractère de séparation des différentes valeurs de l'argument
Object type	Le type de la valeur de l'argument
String value	La valeur de l'option
String[] values	Les valeurs de l'option

La méthode addOption() de la classe Options permet d'ajouter la définition d'une nouvelle option. Elle possède plusieurs surcharges.

Une option est encapsulée dans la classe Option.

Une instance de type Option peut être créée de trois manières pour être ajoutée dans une instance de type Options :

- invoquer un de ses constructeurs
- utiliser une des surcharges de la méthodes addOption() de la classe Options qui attend en paramètres les principales caractéristiques de l'option
- utiliser la classe OptionBuilder qui implémente le motif de conception Monteur

La classe Option possède plusieurs constructeurs :

Constructeur	Rôle
Option(String opt, boolean hasArg, String description)	Créer une instance de type Option avec les paramètres fournis : • opt : le nom de l'option tel qu'il sera fourni par l'utilisateur • hasArg : l'option possède un argument ou non • description : la description de l'option qui sera utilisée pour afficher une aide à l'utilisateur • longOpt :
Option(String opt, String description)	
Option(String opt, String longOpt, boolean hasArg, String description)	

La méthode addOption() possède une surcharge attendant en paramètre une instance de type Option.

Exemple :

```
package fr.jmdoudoux.dej.cli;

import org.apache.commons.cli.Option;
import org.apache.commons.cli.Options;

public class App {

  public static void main(final String[] args) {
    final Options options = new Options();
    final Option optionNom = new Option("n", true, "[nom] votre nom");
    options.addOption(optionNom);
  }
}
```

La méthode addOption() de la classe Options possède deux surcharges attendant plusieurs paramètres qui sont les principales caractéristiques de l'option.

Méthode	Rôle
Options addOption(String opt, boolean hasArg, String description)	Ajouter une option qui ne possède qu'un nom court
Options addOption(String opt, String longOpt, boolean hasArg, String description)	Ajouter une option qui possède un nom court et un nom long

Exemple :

```
package fr.jmdoudoux.dej.cli;

import org.apache.commons.cli.Options;

public class App {

  public static void main(final String[] args) {
    final Options options = new Options();
    options.addOption("n", true, "[nom] votre nom");
  }
}
```

La classe org.apache.commons.cli.OptionBuilder implémente le motif de conception monteur pour faciliter la configuration et l'obtention d'une instance de type Option. L'utilisation d'un OptionBuilder permet d'avoir un contrôle très précis sur la configuration de l'Option qui sera instanciée.

Méthode	Rôle

static Option create()	Obtenir l'instance à partir de la configuration courante
static Option create(char opt)	Obtenir l'instance à partir de la configuration courante et le nom court correspond au caractère fourni en paramètre
static Option create(String opt)	Obtenir l'instance à partir de la configuration courante et le nom court fourni en paramètre
static OptionBuilder hasArg()	Préciser que l'option possède un argument obligatoire
static OptionBuilder hasArg(boolean hasArg)	
static OptionBuilder hasArgs()	Préciser que l'option peut avoir plusieurs arguments dont le nombre n'est pas limité
static OptionBuilder hasArgs(int num)	Préciser que l'option peut avoir plusieurs arguments dont le nombre est fourni en paramètre
static OptionBuilder hasOptionalArg()	Préciser que l'option possède un argument facultatif
static OptionBuilder hasOptionalArgs()	Préciser que l'option peut avoir plusieurs arguments facultatifs
static OptionBuilder hasOptionalArgs(int numArgs)	Préciser que l'option peut avoir plusieurs arguments facultatifs dont le nombre est fourni en paramètre
static OptionBuilder isRequired()	Préciser que l'option est obligatoire
static OptionBuilder isRequired(boolean newRequired)	
static OptionBuilder withArgName(String name)	Préciser le nom de l'argument de l'option
static OptionBuilder withDescription(String newDescription)	Préciser la description de l'option
static OptionBuilder withLongOpt(String newLongopt)	Préciser le nom long de l'option
static OptionBuilder withType(Object newType)	Préciser le type de la valeur de l'option
static OptionBuilder withValueSeparator()	Préciser que les valeurs de l'argument sont séparés par un caractère =
static OptionBuilder withValueSeparator(char sep)	Préciser le caractère de séparation des valeurs de l'argument

Exemple :

```
package fr.jmdoudoux.dej.cli;

import org.apache.commons.cli.OptionBuilder;
import org.apache.commons.cli.Options;

public class App {

  @SuppressWarnings("static-access")
  public static void main(final String[] args) {
    final Options options = new Options();
    options.addOption(OptionBuilder.withArgName("nom").hasArg()
      .withDescription("[nom] votre nom").create("n"));
  }
}
```

Il est possible de définir un groupe d'options qui contient un ensemble d'options parmi lesquelles une doit être utilisée.

Un groupe d'options est encapsulé dans un classe de type org.apache.commons.cli.OptionGroup.

Exemple :

```
package fr.jmdoudoux.dej.cli;

import org.apache.commons.cli.OptionBuilder;
import org.apache.commons.cli.OptionGroup;
import org.apache.commons.cli.Options;

public class App {

  @SuppressWarnings("static-access")
  public static void main(final String[] args) {
    final Options options = new Options();
    final OptionGroup groupe = new OptionGroup();
    groupe.setRequired(true);
    groupe.addOption(OptionBuilder.withDescription("Create").create("c"));
    groupe.addOption(OptionBuilder.withDescription("Read").create("r"));
    groupe.addOption(OptionBuilder.withDescription("Update").create("u"));
    groupe.addOption(OptionBuilder.withDescription("Delete").create("d"));
    options.addOptionGroup(groupe);
  }
}
```

Il est possible de définir des options utilisant le format des propriétés fournies à une JVM.

Exemple :

```
package fr.jmdoudoux.dej.cli;

import org.apache.commons.cli.Option;
import org.apache.commons.cli.OptionBuilder;
import org.apache.commons.cli.Options;

public class App {

  @SuppressWarnings("static-access")
  public static void main(final String[] args) {
    final Options options = new Options();
    final Option property = OptionBuilder
        .withArgName("propriete=valeur")
        .hasArgs(2)
        .isRequired()
        .withValueSeparator()
        .withDescription("Assigner une valeur à la propriete")
        .create("D");
    options.addOption(property);
  }
}
```

Résultat :

```
java app -DmaPropriete=maValeur
```

La définition des options est la même quel que soit le parseur qui sera utilisé ultérieurement pour analyser les arguments passés en paramètres du programme.

115.2.2. L'analyse des options fournies en paramètres

Un parseur va utiliser la définition des options encapsulées dans une instance de type Options et les arguments passés au programme pour les analyser.

CLI propose plusieurs parseurs qui implémentent l'interface CommandLineParser en héritant de la classe Parser :

- BasicParser : analyseur basique qui impose un espace entre les options et leurs arguments
- GnuParser : analyseur qui supporte le style d'options Gnu (support des options avec noms longs, l'argument peut être accolé à l'option en utilisant un caractère égal, ...)

- PosixParser : analyseur qui supporte le style d'options Posix (noms courts des options, possibilité de concaténer plusieurs options, ...)

L'interface CommandLineParser définit deux méthodes :

Méthode	Rôle
CommandLine parse(Options options, String[] arguments)	Analyser les arguments par rapport aux options définies
CommandLine parse(Options options, String[] arguments, boolean stopAtNonOption)	Analyser les arguments par rapport aux options définies : le booléen permet de préciser si l'analyse se poursuit à la rencontre d'une option non définie

Les surcharges de la méthode parse() renvoient une instance de type CommandeLine qui encapsule les options et leurs valeurs extraites suite à l'analyse des arguments.

Exemple :

```
package fr.jmdoudoux.dej.cli;
import org.apache.commons.cli.CommandLineParser;
import org.apache.commons.cli.GnuParser;
import org.apache.commons.cli.Option;
import org.apache.commons.cli.OptionBuilder;
import org.apache.commons.cli.Options;
import org.apache.commons.cli.ParseException;

public class App {

  @SuppressWarnings("static-access")
  public static void main(final String[] args) {
    final Options options = new Options();
    final Option property = OptionBuilder
        .withArgName("propriete=valeur")
        .hasArgs(2)
        .isRequired()
        .withValueSeparator()
        .withDescription("Assigner une valeur à la propriete")
        .create("D");
    options.addOption(property);
    final CommandLineParser parser = new GnuParser();
    try {
      parser.parse(options, args, false);
    } catch (final ParseException exp) {
      System.err.println("Echec de l'analyse des options: " + exp.getMessage());
    }
  }
}
```

Résultat :

```
java App -DmaPropriete=maValeur -X
Echec de l'analyse des options: Unrecognized option: -X
```

Si une erreur est détectée lors de l'analyse par la méthode parse() alors elle lève une exception de type ParseException dont le message précise l'origine de l'anomalie.

L'exception ParseException possède plusieurs exceptions filles qui permettent d'obtenir des informations spécifiques :

- AlreadySelectedException : levée si plusieurs options d'un même groupe sont fournies en argument
- MissingArgumentException : levée si un argument obligatoire est manquant
- MissingOptionException : levée si une ou plusieurs options obligatoires sont manquantes
- UnrecognizedOptionException : levée si une option passée en argument n'est pas définie les options

115.2.3. L'obtention des options et de leurs valeurs

La classe CommandLine encapsule les options et les valeurs extraites suite à l'analyse des arguments passés à l'application.

Elle possède plusieurs méthodes qui permettent d'obtenir les informations qu'elle encapsule :

Méthode	Rôle
List getArgList()	
String[] getArgs()	
Object getOptionObject(char opt)	Retourner le type de l'option dont le nom court est passé en paramètre
Object getOptionObject(String opt)	Deprecated. Il est préférable d'utiliser la méthode getParsedOptionValue(String)
Properties getOptionProperties(String opt)	
Option[] getOptions()	Retourner un tableau des options
String getOptionValue(char opt)	Retourner l'argument de l'option dont le nom court est passé en paramètre
String getOptionValue(char opt, String defaultValue)	Retourner l'argument de l'option dont le nom court est passé en paramètre ou la valeur par défaut fournie en paramètre si l'argument n'est pas défini
String getOptionValue(String opt)	Retourner l'argument de l'option dont le nom court est passé en paramètre
String getOptionValue(String opt, String defaultValue)	Retourner l'argument de l'option dont le nom court est passé en paramètre ou la valeur par défaut fournie en paramètre si l'argument n'est pas défini
String[] getOptionValues(char opt)	Retourner un tableau des valeurs de l'option dont le nom court est passé en paramètre
String[] getOptionValues(String opt)	Retourner un tableau des valeurs de l'option dont le nom court est passé en paramètre
Object getParsedOptionValue(String opt)	Retourner l'argument de l'option dont le nom court est passé en paramètre
boolean hasOption(char opt)	Déterminer si l'option est définie ou non
boolean hasOption(String opt)	Déterminer si l'option est définie ou non
Iterator iterator()	Retourner un Iterator sur les options

Exemple :

```
package fr.jmdoudoux.dej.cli;

import java.util.Arrays;
import org.apache.commons.cli.CommandLine;
import org.apache.commons.cli.CommandLineParser;
import org.apache.commons.cli.GnuParser;
import org.apache.commons.cli.Option;
import org.apache.commons.cli.OptionBuilder;
import org.apache.commons.cli.Options;
import org.apache.commons.cli.ParseException;

public class App {

  @SuppressWarnings("static-access")
  public static void main(final String[] args) {
    final Options options = new Options();
    final Option property = OptionBuilder
        .withArgName("propriete=valeur")
        .hasArgs(2)
        .isRequired()
        .withValueSeparator()
        .withDescription("Assigner une valeur à la propriete")
        .create("D");
    options.addOption(property);
    final CommandLineParser parser = new GnuParser();
    try {
      final CommandLine line = parser.parse(options, args);
      if (line.hasOption("D")) {
        System.out.println(line.getOptionValue("D"));
        System.out.println(Arrays.deepToString(line.getOptionValues("D")));
      }
    } catch (final ParseException exp) {
      System.err.println("Echec de l'analyse des options: " + exp.getMessage());
    }
  }
}
```

Résultat :

```
java App -DmaPropriete=maValeur
maPropriete
[maPropriete, maValeur]
```

115.2.4. L'affichage d'une aide sur les options

La classe HelpFormatter permet de gérer un résumé des options qui pourra être affiché à l'utilisateur.

Elle possède plusieurs méthodes pour configurer les options de formatage et obtenir le résultat.

Plusieurs surcharges de la méthode printHelp() permettent d'afficher sur la sortie standard ou sur un Writer un résumé des options utilisables.

Plusieurs surcharges de la méthode printUsage() permettent d'afficher sur la sortie standard ou sur un Writer un résumé de l'utilisation des options.

Exemple :

```
package fr.jmdoudoux.dej.cli;

import org.apache.commons.cli.HelpFormatter;
import org.apache.commons.cli.Option;
import org.apache.commons.cli.OptionBuilder;
import org.apache.commons.cli.Options;

public class App {

  @SuppressWarnings("static-access")
  public static void main(final String[] args) {
    final Options options = new Options();
    final Option property = OptionBuilder
        .withArgName("propriete=valeur")
        .hasArgs(2)
        .isRequired()
        .withValueSeparator()
        .withDescription("Assigner une valeur à la propriete")
        .create("D");
    options.addOption(property);
    final HelpFormatter formatter = new HelpFormatter();
    formatter.printHelp("App", options, true);
  }
}
```

Résultat :

```
usage: App -D <propriete=valeur>
 -D <propriete=valeur>   Assigner une valeur à la propriete
```

Partie 16 : Les tests automatisés

Partie 16 :
Les tests
automatisés

Les tests automatisés sont une composante très importante de la qualité logicielle.

Cette partie contient les chapitres suivants :

- Les frameworks de tests : propose une présentation de frameworks et d'outils pour faciliter les tests du code
- JUnit : présente en détail le framework de tests unitaires le plus utilisé
- JUnit 5 : la version 5 de JUnit est une réécriture complète qui supporte Java 8 et de nombreuses nouvelles fonctionnalités
- Les objets de type mock : ce chapitre détaille la mise en oeuvre des objets de type mocks et les doublures d'objets

116. Les frameworks de tests

Chapitre 116

Niveau : 🟩🟩⬜⬜⬜⬜ Elémentaire

Le but d'un test est de vérifier qu'une fonctionnalité fait ce que l'on attend d'elle.

Les tests d'une application sont une phase très importante dans les cycles de développement et de maintenance d'une application. Ils permettent de détecter des bugs et de s'assurer que l'application réponde au cahier des charges et aux spécifications.

Ces tests peuvent prendre différentes formes :

- tests unitaires : les tests unitaires automatisés sont un des mécanismes les plus importants pour améliorer la qualité et tenter de garantir la fiabilité du code d'une application.
- tests d'intégration
- tests de recette : le but est de vérifier que l'application réponde aux spécifications fonctionnelles. Ces tests sont faits par les utilisateurs qui devraient fournir un PV de recette
- tests de charge (robustesse, performance, montée en charge, ...)
- tests de stress
- tests d'acceptabilité
- tests de sécurité
- ...

La mise en oeuvre des tests peut être facilitée par l'utilisation d'outils :

- frameworks d'automatisation des tests unitaires : xUnit, TestNG, ...
- outils de couverture de code (code coverage) : EMMA, Cobertura, ...
- outils pour automatiser les tests des IHM : Selenium pour les applications web, ...
- outils pour les tests fonctionnels : FitNesse, ...
- ...

Ce chapitre va essentiellement se concentrer sur la mise en oeuvre de quelques-uns de ces tests avec certains outils.

Ce chapitre contient plusieurs sections :

- ♦ Les tests unitaires
- ♦ Les frameworks et outils de tests

116.1. Les tests unitaires

Les tests unitaires peuvent être réalisés de différentes manières :

- manuelle : par exemple en utilisant les capacités de l'IDE notamment celles du débogueur
- manuelle et reproductible : par exemple en créant pour chaque classe une méthode main qui permet d'exécuter des tests. Ce type de tests nécessite un lancement à la main et une analyse humaine des résultats
- automatisée avec un framework de tests

L'utilisation d'un débogueur peut être pratique pour tester du code fraîchement écrit et comprendre son fonctionnement mais il ne permet pas d'automatiser ces tests. En effet, cette technique requiert une intervention manuelle et une interprétation humaine des résultats.

C'est la même problématique avec l'utilisation de traces avec des System.out ou l'écriture dans un fichier : les données de ces traces doivent être analysées par une personne.

L'utilisation de frameworks dédiés à l'automatisation des tests unitaires permet d'assurer une meilleure qualité et fiabilité du code. Cette automatisation facilite aussi le passage de tests de non-régression notamment lors des mises à jour du code. De plus, l'utilisation de ces frameworks ne nécessite aucune modification dans le code à tester ce qui sépare clairement les traitements représentés dans le code de leurs tests. Enfin, l'analyse des résultats peut être automatisée puisque chaque résultat de tests possède un statut généralement ok ou en erreur.

Ces frameworks ne sont que des outils qui permettent la mise en oeuvre de tests unitaires mais ils ne dispensent pas d'utiliser une méthodologie pour mettre en oeuvre ces tests.

Un test unitaire se déroule en quatre étapes :

- setup : initialiser des objets ou des ressources
- call : exécuter le code à vérifer
- verify : vérifier des données issues des traitements
- teardown : permettre de faire le ménage ou de libérer des ressources

Les tests unitaires automatisés sont très importants et ce durant tout le cycle de vie d'une application :

- conception : rédaction de la liste des cas de tests
- développement : tests du code, détection précoce de bugs,
- maintenance : tests de non-régression, encouragent et facilitent le refactoring

116.1.1. L'utilité des tests unitaires automatisés

L'utilité des tests unitaires automatisés n'est plus à démontrer : ils sont même primordiaux dans certaines méthodologies notamment XP (eXtreme Programming) et TDD (Test Driven Development). Ils servent à promouvoir et vérifier la qualité et la fiabilité du code.

Les tests unitaires automatisés sont un des outils les plus puissants pour améliorer la qualité d'une application. De plus, l'utilisation de tests unitaires améliore l'organisation et la stabilité du code.

Les tests unitaires n'ont pas qu'un effet de test immédiat du code mais surtout ils permettent d'effectuer des tests de non-régression lors de modifications qui interviennent inévitablement durant la vie d'une application. Les tests unitaires automatisés sont donc particulièrement intéressants pour les tests de non-régression qui seront automatisés. Il est courant d'avoir des portions de code fréquemment perçues comme mystiques car personne ne comprend plus comment elles fonctionnent malgré le fait que ce code soit primordial. Il est alors toujours délicat de faire évoluer ce code lors de maintenances correctrices ou évolutives.

La présence de tests unitaires automatisés va rassurer le développeur car il pourra réexécuter ces tests avant et après les modifications pour s'assurer qu'il n'y a pas de regression. Bien sûr, le dégré d'assurance augmente avec la croissance du nombre de tests et leur qualité.

L'écriture de cas de tests permet de prouver que le code à tester fonctionne. Les cas de tests permettent ensuite de s'assurer de la non-régression lors des maintenances dans le code. Les tests unitaires permettent de capitaliser sur les tests à effectuer et ainsi de limiter les effets de bord liés aux inévitables modifications correctrices ou évolutives du code.

La POO implique naturellement des dépendances entre les classes. Une modification dans une de ces classes peut facilement induire des effets de bord dans les classes appelantes. Si les tests sont complets et corrects, une modification ayant un effet de bord fera échouer les tests existants. Dans ce cas, soit la modification nécessite une adaptation du cas de tests soit un bug a introduit un effet de bord dans le comportement du code.

L'existence de tests unitaires couvrant une majorité des cas de tests permet d'être plus confiant lors de la modification de code : cela peut améliorer la garantie qu'une modification n'a pas d'effet de bord.

Si une classe possède un ensemble complet de tests unitaires, il y a moins de réticences à faire des modifications dans son code lors des maintenances correctives ou évolutives, pour améliorer les performances ou pour faire du refactoring. Les tests permettent de s'assurer de la non-régression des fonctionnalités proposées.

Au fur et à mesure que des modifications sont faites dans le temps, les risques augmentent dans une application qui ne possède pas ou peu de tests unitaires. L'absence de tests automatisés implique des tests manuels qui peuvent être oubliés ou mal interprétés augmentant ainsi le risque de ne pas détecter d'effets de bord.

Le coût d'écriture des tests est largement compensé par celui gagné par la réutilisation des tests à chaque itération corrective.

Il est plus facile d'effectuer des opérations de refactoring si les classes disposent d'un ensemble des tests unitaires complets.

Les tests unitaires sont les premiers tests réalisés parmi l'ensemble des tests qui seront réalisés sur l'application. Il ne faut surtout pas les sous-estimer en se disant que les tests suivants permettront de détecter les bugs car leur grand avantage est qu'avec un framework dédié ils peuvent être automatisés.

La rédaction de tests unitaires implique nécessairement une amélioration de la conception du code. Il est très facile d'écrire du code lorsque celui-ci ne doit pas être testé. Cependant pour écrire du code qui doit être testé, il faut que la conception du code soit adaptée pour faciliter la mise en oeuvre des tests unitaires :

- améliorer la granularité des méthodes : il est plus facile de tester des méthodes courtes que de longues méthodes
- réduire la dépendance entre les objets : il est intéressant de mettre en oeuvre certains design patterns afin de réduire le couplage entre les objets
- une classe avec un couplage fort vers d'autres classes est difficile à tester.

Les tests unitaires peuvent facilement servir d'exemples d'utilisation du code testé puisque le code est nécessairement invoqué durant les tests.

Il est encore fréquent de voir des scénarios de tests écrits dans un document et exécutés manuellement par un humain. Cette approche est obsolète dans la mesure où des outils existent pour automatiser une bonne partie de ces tests évitant ainsi les erreurs humaines (aucune exécution des tests, oublie de l'exécution de cas, mauvaise interprétations des résultats, ...). De plus, les fonctionnalités d'une application ont tendance à augmenter avec le temps ce qui rend ce processus encore plus long et fastidieux.

116.1.2. Quelques principes pour mettre en oeuvre des tests unitaires

Il existe plusieurs approches pour mettre en oeuvre des tests unitaires automatisés : chacune a des avantages et des inconvénients dont il faut tenir compte selon le contexte.

Il est important de définir quand les tests unitaires sont écrits. Plusieurs mises en oeuvre sont possibles :

- écrire les tests juste après avoir écrit une méthode
- idéalement, écrire les tests avant le code à tester
- écrire les tests, écrire le code pour faire échouer les tests, vérifier que les tests échouent, corriger le code, vérifier que les tests sont OK

Il faut développer de préférences les tests unitaires le plus tôt possible. Dans une approche traditionnelle, juste après l'écriture de la méthode. Dans une approche TDD (Test Drive Development), avant l'écriture de la méthode.
Ceci présente plusieurs avantages par rapport à une écriture ultérieure de tests :

- permettre de détecter des bugs le plus rapidement possible,
- coder des cas oubliés dans la méthode,
- s'assurer que les tests unitaires sont écrits,
- écrire du code testable évitant ainsi un refactoring parfois conséquent
- ...

Il est préférable d'appliquer trois règles avec les tests unitaires :

- tester le plus possible : afin d'augmenter les chances de découvrir des bugs
- tester le plus tôt possible : plus les tests sont faits tôt plus les bugs sont rapidement détectés
- tester le plus souvent possible : en les automatisant et si possible en les intégrant dans un processus d'intégration continue

Un test unitaire devrait respecter certains principes :

- le test doit être le plus petit et le plus simple possible
- chaque test doit être isolé : un test ne doit pas dépendre d'un autre. Ceci permet aussi de garantir qu'une modification d'un test n'aura pas d'impact sur un autre
- pour pouvoir être facilement exécutés régulièrement, les tests unitaires doivent être automatisés

Le rôle des tests unitaires est d'automatiser des tests sur des unités de code les plus petites possibles, généralement une méthode. Cependant le code d'une méthode peut avoir besoin d'autres objets ou de ressources externes.

Plus le code à tester va avoir de dépendances plus il sera difficile à tester. Il faut donc minimiser ces dépendances en utilisant plusieurs solutions :

- utilisation de design patterns
- utilisation d'objets de type mock
- éviter de faire appels à la base de données dans les cas de tests
- ...

Un test unitaire doit impérativement se faire de façon isolée, donc sans dépendre d'autres tests ni requérir les dépendances utilisées par la fonctionnalité en cours de test. Le but d'un test unitaire est de tester les traitements de la fonctionnalité et non de tester les interactions qu'elle peut avoir avec ces dépendances.

Un test unitaire doit obligatoirement être répétable pour obtenir toutes ses lettres de noblesse. Cela permet de capitaliser les tests unitaires non seulement pour les tests unitaires mais aussi pour les tests de non-régression.

Chaque cas de tests doit être autonome et ne doit donc pas dépendre d'un ou plusieurs autres cas de tests.

Il est pratique de définir et d'utiliser des conventions de nommages pour les classes de tests. Certaines sont imposées par le framework de tests utilisé : dans ce cas leurs mises en oeuvre est obligatoire. Dans les autres cas, il est préférable de définir ses propres conventions et de les mettre en oeuvre, par exemple :

- préfixer les classes de tests par Test suivi du nom de la classe testée et les mettre dans un package dédié dont le nom correspond au nom du package des classes testés préfixé par test
- mettre les classes de test dans le même package que les classes à tester en les suffixant par Test. Cela permet entre autres de facilement identifier les classes sans classes de tests. Ant peut être utilisé pour filtrer les classes à inclure dans la génération des livrables
- écrire une classe de tests par classe testée

116.1.3. Les difficultés lors de la mise en oeuvre de tests unitaires

Plusieurs difficultés sont rencontrées lors de la mise en oeuvre de tests unitaires :

- réticences à la mise en oeuvre
- difficultés de rédaction et de codage
- couverture du code testé
- temps nécessaire à la rédaction des cas tests
- véracité des cas de tests
- temps nécessaire à la maintenance des cas de tests
- les cas de tests doivent être répétables
- il n'y a pas que le code qui doit être testé, il est aussi nécessaire de tester les valeurs de certaines ressources (base de données, fichiers, ...)
- ...

Lorsque l'on parle aux développeurs de rédiger des tests unitaires, il est fréquent d'obtenir des réticences avec des justifications futiles :

- "Je n'ai pas le temps",
- "Je ne sais pas les écrire",
- "Ce n'est pas mon job",
- "Je ne fais jamais de bugs",
- ...

Dans la plupart des cas, il est plus difficile d'écrire les tests que d'écriture le code à tester. Ainsi, l'écriture du code d'une application est un art mais l'écriture de tests pour ce code est un art encore plus complexe. De ce fait, la rédaction des cas de tests est fréquemment confiée à des développeurs expérimentés ou dédiés à cette activité.

Les tests unitaires doivent évoluer avec le code de l'application. Il est donc très important que le code des tests unitaires soit simple, compréhensible et maintenable.

La mise en oeuvre de tests unitaires automatisés augmente la fiabilité du code mais elle ne peut pas offrir une garantie à 100% pour plusieurs raisons :

- la couverture du code testé ne peut généralement pas être totale
- il est impossible de couvrir tous les cas de tests
- les tests unitaires peuvent contenir, eux-mêmes, des bugs

Il n'est pas possible de couvrir tous les cas possibles avec des cas de tests unitaires. Il est donc nécessaire de déterminer quelles classes posséderont des tests unitaires, de maximiser le nombre de ces classes testées, de définir les cas de tests de chaque classes et de maximiser le nombre de ces cas.

Une des grandes difficultés lors de la rédaction de cas de tests est de s'assurer qu'un maximum de cas de tests est implémenté. Il ne faut surtout pas se contenter de ne tester que les cas de fonctionnement standard mais aussi couvrir un maximum de cas de fonctionnement anormal (données invalides, levée d'exceptions, tests aux limites, ...).

Généralement, les tests unitaires possèdent des dépendances vers des ressources externes (fichiers, bases de données, bibliothèques tierces, connexions réseau, ...). L'utilisation de ces ressources dans les tests unitaires doit être évitée car généralement elle limite la répétitabilité des tests et entraîne un surcoût dans le temps d'exécution des tests unitaires.

Malgré ces difficultés, les tests unitaires automatisés ne doivent pas être occultés car ils peuvent améliorer de façon significative la qualité et la fiabilité du code lors de son écriture et surtout de sa maintenance.

116.1.4. Des best practices

La rédaction des tests unitaires devrait suivre quelques recommandations :

- le nom des tests devrait permettre de facilement fournir une indication sur le but du test
- il est préférable de n'avoir qu'un seul assert par test car un test ne devrait avoir qu'une seule raison d'échouer
- le code des tests unitaires doit être maintenu au même titre que le code qu'il teste : la même attention doit être portée dans leur écriture (respect des normes, commentaires, refactoring, ...)
- stocker les tests unitaires dans un package dédié dont le nom est celui du package de la classe à tester avec le prefixe test

Chaque test unitaire doit s'exécuter le plus rapidement possible : le nombre de tests unitaires va croître au fur et à mesure des développements donc le temps d'exécution des tests va croître lui aussi.

Il est préférable d'inclure l'exécution des tests unitaires dans un processus d'intégration continue.

Il faut conserver les cas de tests les plus simples possibles. Par exemple, pour le test d'une méthode qui additionne deux nombres, il est préférable pour tester le cas standard qui utilise de petits nombres plutôt que d'utiliser de grands nombres. La véracité du test est la même mais le test est plus facile à comprendre et à vérifier.

Chaque test doit correspondre à un cas de test unique. Il est préférable de n'avoir qu'un seul test dans un cas de test, soit une seule instruction de type assert. Ceci rendra le code du test plus simple et facilitera le calcul de métriques lors de l'exécution de tests.

Le test d'un constructeur nécessite généralement l'invocation de getters et setters pour vérifier les valeurs des paramètres fournis au constructeur et généralement utilisées pour initialiser directement ou indirectement des champs de l'objet.

Il n'est pas toujours facile de rendre les tests d'une méthode indépendants de l'utilisation d'autres méthodes. Par exemple, il est difficile de tester un setter sans faire appel au getter de la propriété correspondante.

Pour tester des méthodes privées, il faut tester les méthodes qui font appels à ces méthodes privées.

Il est aussi généralement non trivial, de tester une méthode qui n'a pas de paramètre de retour. Ces méthodes effectuent généralement des modifications sur des éléments internes ou externes à la classe. Il faut alors capturer le résultat de ces modifications pour pouvoir réaliser les tests.

Il ne faut pas hésiter à remonter dans le gestionnaire de source du code dont un ou plusieurs tests unitaires échouent.

Il ne faut pas hésiter à enrichir les tests avec de nouveaux cas ou créer des cas de tests pour des classes qui n'en ont pas. Le code d'une application et le code des tests unitaires doivent évoluer dans une optique d'améliorations continues.

A chaque maintenance dans le code, les tests unitaires doivent être exécutés et maintenus eux aussi au besoin. Il ne faut surtout pas livrer du code dont au moins un test unitaire échoue quelques soient les raisons.

116.2. Les frameworks et outils de tests

De nombreux frameworks et outils open source sont proposés pour faciliter la mise en oeuvre des tests

- frameworks de tests unitaires et leurs extensions
- frameworks pour le mocking
- outils de tests de charge
- outils d'analyse de couverture du test
- ...

116.2.1. Les frameworks pour les tests unitaires

Plusieurs frameworks open source sont utilisables dans le monde Java notamment :

- JUnit : C'est le plus ancien et le plus répandu ce qui en fait un standard de facto
- TestNG :

JUnit est à l'origine de plusieurs frameworks similaires pour différentes plates-formes ou langages notamment nUnit (.Net), dUnit (Delphi), cppUnit (C++), ... Tous ces frameworks sont regroupés dans une famille nommée xUnit.

116.2.2. Les frameworks pour le mocking

Généralement les tests unitaires de code d'une application, notamment celles développées en couches, nécessitent l'utilisation d'objets de type mock pour permettre de se concentrer sur le test du code de la méthode en minimisant les effets de bord liés aux autres objets utilisés dans le code.

L'utilisation de ces frameworks est détaillé dans le chapitre «Les objets de type mock» .

116.2.3. Les extensions de JUnit

JUnit est utilisé dans un certain nombre de projets qui proposent d'étendre ses fonctionnalités :

- JunitReport : une tache Ant pour générer un rapport des tests effectués avec JUnit sous Ant

- JWebUnit : un framework open source de tests pour des applications web
- StrutsTestCase : extension de JUnit pour les tests d'applications utilisant Struts 1.0.2 et 1.1
- XMLUnit : extension de JUnit pour les tests sur des documents XML
- Cactus : un framework open source de tests pour des composants serveur J2EE

116.2.4. Les outils de tests de charge

Apache JMeter est l'outil de tests de charge le plus répandu pour tout ce qui repose sur le protocole http.

SoapUI est particulièrement adapté pour les tests unitaires et les tests de charges de services web.

116.2.5. Les outils d'analyse de couverture de tests

Des outils sont proposés pour vérifier le taux de couverture des cas de tests vis-à-vis du code (test coverage analyser).

Le but de ces outils est de faciliter la détermination des fonctionnalités qui possèdent des tests et par conséquent permettre de déterminer quelles portions du code ne sont pas testées du tout ou insuffisamment testées.

Plusieurs outils open source existent notamment :

- Cobertura
- JCoverage
- Jester
- CodeCover

117. JUnit

Chapitre 117

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

JUnit est un framework open source pour le développement et l'exécution de tests unitaires automatisables. Le principal intérêt est de s'assurer que le code répond toujours aux besoins même après d'éventuelles modifications. Plus généralement, ce type de tests est appelé tests unitaires de non-régression.

JUnit a été initialement développé par Erich Gamma et Kent Beck.

JUnit propose :

- Un framework pour le développement des tests unitaires reposant sur des assertions qui testent les résultats attendus
- Des applications pour permettre l'exécution des tests et afficher les résultats

Le but est d'automatiser les tests. Ceux-ci sont exprimés dans des classes sous la forme de cas de tests avec leurs résultats attendus. JUnit exécute ces tests et les comparent avec ces résultats.

Cela permet de séparer le code de la classe, du code qui permet de la tester. Souvent pour tester une classe, il est facile de créer une méthode main() qui va contenir les traitements de tests. L'inconvénient est que ce code "superflu" est inclus dans la classe. De plus, son exécution doit se faire manuellement.

La rédaction de cas de tests peut avoir un effet immédiat pour détecter des bugs mais surtout elle a un effet à long terme qui facilite la détection d'effets de bords lors de modifications.

Les cas de tests sont regroupés dans des classes Java qui contiennent une ou plusieurs méthodes de tests. Les cas de tests peuvent être exécutés individuellement ou sous la forme de suites de tests.

JUnit permet le développement incrémental d'une suite de tests.

Avec JUnit, l'unité de test est une classe dédiée qui regroupe des cas de tests. Ces cas de tests exécutent les tâches suivantes :

- création d'une instance de la classe et de tout autre objet nécessaire aux tests
- appel de la méthode à tester avec les paramètres du cas de tests
- comparaison du résultat attendu avec le résultat obtenu : en cas d'échec, une exception est levée

JUnit est particulièrement adapté pour être utilisé avec la méthode eXtreme Programming puisque cette méthode préconise, entre autres, l'automatisation des tâches de tests unitaires qui ont été définies avant l'écriture du code.

La version utilisée dans ce chapitre est la 3.8.1 sauf dans la section dédiée à la version 4 de JUnit.

La page officielle est à l'url : https://junit.org/.

La dernière version de JUnit peut être téléchargée sur le site junit.org/. Pour l'installer, il suffit de décompresser l'archive dans un répertoire du système.

Pour pouvoir utiliser JUnit, il faut ajouter le fichier junit.jar au classpath.

Ce chapitre contient plusieurs sections :

- Un exemple très simple
- L'écriture des cas de tests
- L'exécution des tests
- Les suites de tests
- L'automatisation des tests avec Ant
- JUnit 4

117.1. Un exemple très simple

L'exemple utilisé dans cette section est la classe suivante :

Exemple :

```
public class MaClasse{

  public static int calculer(int a, int b) {
    int res = a + b;

    if (a == 0){
      res = b * 2;
    }

    if (b == 0) {
      res = a * a;
    }
    return res;
  }
}
```

Il faut suivre plusieurs étapes :

1) compiler cette classe : javac MaClasse.java

2) écrire une classe qui va contenir les différents tests à réaliser par JUnit. L'exemple est volontairement simpliste en ne définissant qu'un seul cas de tests.

Exemple :

```
import junit.framework.*;

public class MaClasseTest extends TestCase{

  public void testCalculer() throws Exception {

    assertEquals(2,MaClasse.calculer(1,1));
  }
}
```

3) compiler cette classe avec le fichier junit.jar qui doit être dans le classpath.

4) enfin, appeler JUnit pour qu'il exécute la séquence de tests.

Exemple :

```
java -cp junit.jar;. junit.textui.TestRunner MaClasseTest
C:\java\testjunit>java -cp junit.jar;. junit.textui.TestRunner
  MaClasseTest
  .
  Time: 0,01
OK (1 test)
```

Attention : le respect de la casse dans le nommage des méthodes de tests est très important. Les méthodes de tests doivent obligatoirement commencer par test en minuscule car JUnit utilise l'introspection pour déterminer les méthodes à exécuter.

Exemple :

```
import junit.framework.*;

public class MaClasseTest extends TestCase{

  public void TestCalculer() throws Exception {

    assertEquals(2,MaClasse.calculer(1,1));
  }
}
```

L'utilisation de cette classe avec JUnit produit le résultat suivant :

Résultat :

```
C:\java\testjunit>java -cp junit.jar;. junit.textui.TestRunner
MaClasseTest
.F
Time: 0,01
There was 1 failure:
1) warning(junit.framework.TestSuite$1)junit.framework.AssertionFailedError: No
tests found in MaClasseTest

FAILURES!!!
Tests run: 1,  Failures: 1,  Errors: 0
```

117.2. L'écriture des cas de tests

JUnit propose un framework pour écrire les classes de tests.

Un test est une classe qui hérite de la classe TestCase. Par convention le nom de la classe de test est composé du nom de la classe suivi de Test.

Chaque cas de tests fait l'objet d'une méthode dans la classe de tests. Le nom de ces méthodes doit obligatoirement commencer par le préfixe test.

Chacune de ces méthodes contient généralement des traitements en trois étapes :

- Instanciation des objets requis
- Invocation des traitements sur les objets
- Vérification des résultats des traitements

Il est important de se souvenir lors de l'écriture de cas de tests que ceux-ci doivent être indépendants les uns des autres. JUnit ne garantit pas l'ordre d'exécution des cas de tests puisque ceux-ci sont obtenus par introspection.

Toutes les classes de tests avec JUnit héritent de la classe Assert.

117.2.1. La définition de la classe de tests

Pour écrire les cas de tests, il faut écrire une classe qui étende la classe junit.framework.TestCase. Le nom de cette classe est le nom de la classe à tester suivi par "Test".

Remarque : dans la version 3.7 de JUnit, une classe de tests doit obligatoirement posséder un constructeur qui attend un objet de type String en paramètre.

Exemple :

```
import junit.framework.*;

public class MaClasseTest extends TestCase{

  public MaClasseTest(String testMethodName) {
    super(testMethodName);
  }

  public void testCalculer() throws Exception {
    fail("Cas de tests a ecrire");
  }
}
```

Dans cette classe, il faut écrire une méthode dont le nom commence par "test" en minuscule suivi du nom du cas de tests (généralement le nom de la méthode à tester). Chacune de ces méthodes doit avoir les caractéristiques suivantes :

- elle doit être déclarée public
- elle ne doit renvoyer aucune valeur
- elle ne doit pas posséder de paramètres.

Par introspection, JUnit va automatiquement rechercher toutes les méthodes qui respectent cette convention. Le respect de ces règles est donc important pour une bonne exécution des tests par JUnit.

117.2.2. La définition des cas de tests

Chaque classe de tests doit avoir obligatoirement au moins une méthode de test sinon une erreur est remontée par JUnit.

JUnit recherche, par introspection, les méthodes qui débutent par test, n'ont aucun paramètre et ne retourne aucune valeur. Ces méthodes peuvent lever des exceptions qui sont automatiquement capturées par JUnit qui remonte alors une erreur et donc un échec du cas de tests.

Dès qu'un test échoue, l'exécution de la méthode correspondante est interrompue et JUnit passe à la méthode suivante.

La classe suivante sera utilisée dans les exemples de cette section :

Exemple :

```
public class MaClasse2{
  private int a;
  private int b;

  public MaClasse2(int a, int b) {
    this.a = a;
    this.b = b;
  }

  public int getA() {
    return a;
  }

  public int getB() {
    return b;
  }

  public void setA(int unA) {
    this.a = unA;
  }

  public void setB(int unB) {
    this.b = unB;
  }

  public int calculer() {
    int res = a + b;

    if (a == 0){
      res = b * 2;
    }

    if (b == 0) {
      res = a * a;
    }
    return res;
  }

  public int sommer() throws IllegalStateException {
    if ((a == 0) && (b==0)) {
      throw new IllegalStateException("Les deux valeurs sont nulles");
    }
    return a+b;
  }
}
```

Avec JUnit, la plus petite unité de tests est l'assertion dont le résultat de l'expression booléenne indique un succès ou une erreur.

Les cas de tests utilisent des affirmations (assertion en anglais) sous la forme de méthodes nommées assertXXX() proposées par le framework. Il existe de nombreuses méthodes de ce type qui sont héritées de la classe junit.framework.Assert :

Méthode	Rôle
assertEquals()	Vérifier l'égalité de deux valeurs de type primitif ou objet (en utilisant la méthode equals()). Il existe de nombreuses surcharges de cette méthode pour chaque type primitif, pour un objet de type Object et pour un objet de type String
assertFalse()	Vérifier que la valeur fournie en paramètre est fausse
assertNull()	Vérifier que l'objet fourni en paramètre soit null
assertNotNull()	Vérifier que l'objet fourni en paramètre ne soit pas null
assertSame()	Vérifier que les deux objets fournis en paramètre font référence à la même entité Exemples identiques : assertSame("Les deux objets sont identiques", obj1, obj2); assertTrue("Les deux objets sont identiques ", obj1 == obj2);
assertNotSame()	Vérifier que les deux objets fournis en paramètre ne font pas référence à la même entité
assertTrue()	Vérifier que la valeur fournie en paramètre est vraie

Bien qu'il soit possible de n'utiliser que la méthode assertTrue(), les autres méthodes assertXXX() facilitent l'expression des conditions de tests.

Chacune de ces méthodes possède une version surchargée qui accepte un paramètre supplémentaire sous la forme d'une chaîne de caractères indiquant un message qui sera affiché en cas d'échec du cas de tests. Le message devrait décrire le cas de tests évalué à true.

L'utilisation de cette version surchargée est recommandée car elle facilite l'exploitation des résultats des cas de tests.

Exemple :

```
import junit.framework.*;

public class MaClasse2Test extends TestCase{

  public void testCalculer() throws Exception {
    MaClasse2 mc = new MaClasse2(1,1);
    assertEquals(2,mc.calculer());
  }
}
```

L'ordre des paramètres contenant la valeur attendue et la valeur obtenue est important pour avoir un message d'erreur fiable en cas d'échec du cas de tests. Quelle que soit la surcharge utilisée l'ordre des deux valeurs à tester est toujours le même : c'est toujours la valeur attendue qui précède la valeur courante.

La méthode fail() permet de forcer le cas de tests à échouer. Une version surchargée permet de préciser un message qui sera affiché.

Il est aussi souvent utile lors de la définition des cas de tests de tester si une exception est levée lors de l'exécution des traitements.

Exemple :

```
import junit.framework.*;

public class MaClasse2Test extends TestCase{

  public void testSommer() throws Exception {
    MaClasse2 mc = new MaClasse2(0,0);
    mc.sommer();
  }
}
```

Résultat :

```
C:\>java -cp junit.jar;. junit.textui.TestRunner MaClasse2Test
.E
Time: 0,01
There was 1 error:
1) testSommer(MaClasse2Test)java.lang.IllegalStateException: Les deux valeurs so
nt nulles
        at MaClasse2.sommer(MaClasse2.java:42)
        at MaClasse2Test.testSommer(MaClasse2Test.java:31)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.
java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAcces
sorImpl.java:25)

FAILURES!!!
Tests run: 2,  Failures: 0,  Errors: 1
```

Avec JUnit, pour réaliser de tels cas de tests, il suffit d'appeler la méthode avec les conditions qui doivent lever une exception, d'encapsuler cet appel dans un bloc try/catch et d'appeler la méthode fail() si l'exception désirée n'est pas levée.

Exemple :

```
import junit.framework.*;

public class MaClasse2Test extends TestCase{

  public void testSommer() throws Exception {
    MaClasse2 mc = new MaClasse2(1,1);

    // cas de test 1
    assertEquals(2,mc.sommer());
    // cas de test 2
    try {
      mc.setA(0);
      mc.setB(0);
      mc.sommer();
      fail("Une exception de type IllegalStateException aurait du etre levee");
    } catch (IllegalStateException ise) {
    }
  }
}
```

117.2.3. L'initialisation des cas de tests

Il est fréquent que les cas de tests utilisent une instance d'un même objet ou nécessitent l'usage de ressources particulières telles qu'une instance d'une classe pour l'accès à une base de données par exemple.

Pour réaliser ces opérations de création et de destruction d'objets, la classe TestCase propose les méthodes setUp() et tearDown() qui sont respectivement appelées avant et après l'appel de chaque méthode contenant un cas de tests.

Il suffit simplement de redéfinir ces deux méthodes en fonction de ses besoins.

Cette section va tester le bean ci-dessous :

Exemple :

```
package fr.jmdoudoux.dej.junit;

public class Personne {

  private String nom;

  private String prenom;

  public Personne() {
    super();
  }

  public Personne(String nom, String prenom) {
    super();
    this.nom = nom;
    this.prenom = prenom;
  }

  public String getNom() {
    return nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

  public String getPrenom() {
    return prenom;
  }

  public void setPrenom(String prenom) {
    this.prenom = prenom;
  }
}
```

Le plus simple est de définir un membre privé du type dont on a besoin et de créer une instance de ce type dans la méthode setUp().

Il est important de se souvenir que la méthode setUp() est invoquée systématiquement avant l'appel de chaque méthode de tests. Sa mise en oeuvre n'est donc requise que si toutes les méthodes de tests ont besoin de créer une instance d'un même type ou d'exécuter un même traitement.

Exemple :

```
package fr.jmdoudoux.dej.junit;

import junit.framework.TestCase;

public class PersonneTest extends TestCase {

  private Personne personne;

  public PersonneTest(String name) {
    super(name);
  }

  protected void setUp() throws Exception {
    super.setUp();
    personne = new Personne("nom1","prenom1");
  }

  protected void tearDown() throws Exception {
    super.tearDown();
    personne = null;
  }

  public void testPersonne() {
    assertNotNull("L'instance n'est pas créée", personne);
  }

  public void testGetNom() {
    assertEquals("Le nom est incorrect", "nom1", personne.getNom());
  }

  public void testSetNom() {
    personne.setNom("nom2");
    assertEquals("Le nom est incorrect", "nom2", personne.getNom());
  }

  public void testGetPrenom() {
    assertEquals("Le prenom est incorrect", "prenom1", personne.getPrenom());
  }

  public void testSetPrenom() {
    personne.setPrenom("prenom2");
    assertEquals("Le prenom est incorrect", "prenom2", personne.getPrenom());
  }
}
```

Ceci évite de créer l'instance dans chaque méthode de tests et simplifie donc l'écriture des cas de tests.

Dans l'exemple la méthode tearDown() remet à null l'instance créée : ceci n'est pas une obligation d'autant que le temps des traitements réalisés durant les tests est normalement négligeable. La méthode tearDown() peut cependant avoir un grand intérêt pour, par exemple, libérer des ressources comme une connexion à une base de données initialisée dans la méthode setUp().

Pour des besoins particuliers, il peut être nécessaire d'exécuter du code une seule fois avant l'exécution des cas de tests et/ou d'exécuter du code une fois tous les cas de tests exécutés.

JUnit propose pour cela la classe junit.Extensions.TestSetup qui propose la mise en oeuvre du design pattern décorateur.

Exemple :

```
package fr.jmdoudoux.dej.junit;

import junit.extensions.TestSetup;
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class PersonneTest extends TestCase {
  private Personne personne;

  public PersonneTest(String name) {
    super(name);
  }

  protected void setUp() throws Exception {
    super.setUp();
    personne = new Personne("nom1", "prenom1");
  }

  protected void tearDown() throws Exception {
    super.tearDown();
    personne = null;
  }

...

  public static Test suite() {
    TestSetup setup = new TestSetup(new TestSuite(PersonneTest.class)) {
      protected void setUp() throws Exception {
        // code execute une seule fois avant l'exécution des cas de tests
        System.out
            .println("Appel de la methode setUp() de la classe de tests");
      }

      protected void tearDown() throws Exception {
        // code execute une seule fois après l'exécution de tous les cas de tests
        System.out
            .println("Appel de la methode tearDown() de la classe de tests");
      }
    };
    return setup;
  }

  public static void main(String[] args) {
      junit.textui.TestRunner.run(suite());
  }
}
```

Dans l'exemple ci-dessus, les méthodes setUp() et tearDown() de la classe PersonneTest seront toujours invoquées respectivement avant et après chaque exécution d'un cas de tests.

117.2.4. Le test de la levée d'exceptions

Il est fréquent qu'une méthode puisse lever une ou plusieurs exceptions durant son exécution. Il faut prévoir des cas de tests pour vérifier que dans les conditions adéquates une exception attendue est bien levée.

Exemple :

```
package fr.jmdoudoux.dej.junit;

public class Personne {

  private String nom;

  private String prenom;

  public Personne() {
    super();
  }

  public Personne(String nom, String prenom) {
    super();
    this.nom = nom;
    this.prenom = prenom;
  }
  public String getNom() {
    return nom;
  }

  public void setNom(String nom) {
    if (nom == null) {
      throw new IllegalArgumentException("la propriété nom ne peut pas être null");
    }
    this.nom = nom;
  }

  public String getPrenom() {
    return prenom;
  }

  public void setPrenom(String prenom) {
    this.prenom = prenom;
  }
}
```

Pour effectuer la vérification de la levée d'une exception, il faut inclure l'invocation de la méthode dans un bloc try/catch et faire appel à la méthode fail() si l'exception n'est pas levée.

Exemple :

```
package fr.jmdoudoux.dej.junit;

import junit.framework.TestCase;

public class PersonneTest extends TestCase {

  private Personne personne;

  public PersonneTest(String name) {
    super(name);
  }

  protected void setUp() throws Exception {
    super.setUp();
    personne = new Personne("nom1","prenom1");
  }

  protected void tearDown() throws Exception {
    super.tearDown();
    personne = null;
  }

  public void testPersonne() {
    assertNotNull("L'instance n'est pas créée", personne);
  }

  public void testGetNom() {
    assertEquals("Le nom est incorrect", "nom1", personne.getNom());
  }

  public void testSetNom() {
    personne.setNom("nom2");
    assertEquals("Le nom est incorrect", "nom2", personne.getNom());
    try {
      personne.setNom(null);
      fail("IllegalArgumentException non levée avec la propriété nom à null");
    } catch (IllegalArgumentException iae) {
      // ignorer l'exception puisque le test est OK (l'exception est levée)
    }
  }

  public void testGetPrenom() {
    assertEquals("Le prenom est incorrect", "prenom1", personne.getPrenom());
  }

  public void testSetPrenom() {
    personne.setPrenom("prenom2");
    assertEquals("Le prenom est incorrect", "prenom2", personne.getPrenom());
  }
}
```

Attention : une erreur courante lorsque l'on code ses premiers tests unitaires est d'inclure les invocations de méthodes dans des blocs try/catch. Leur utilisation doit être uniquement réservée aux situations telles que celle de l'exemple précédant. Dans tous les autres cas, il faut laisser l'exception se propager : dans ce cas, JUnit va automatiquement reporter un échec du test. Il est en particulier inutile d'utiliser un bloc try/catch et de faire appel à la méthode fail() dans le catch puisque JUnit le fait déjà.

117.2.5. L'héritage d'une classe de base

Il est possible de définir une classe de base qui servira de classe mère à d'autres classes de tests notamment en leur fournissant des fonctionnalités communes.

JUnit n'impose pas qu'une classe de tests dérive directement de la classe TestCase. Ceci est particulièrement pratique lorsque l'on souhaite que certaines initialisations ou certains traitements soit systématiquement exécutés (exemple chargement d'un fichier de configuration, ...).

Il est par exemple possible de faire des initialisations dans le constructeur de la classe mère et d'invoquer ce constructeur dans les constructeurs des classes filles.

117.3. L'exécution des tests

JUnit propose trois applications différentes nommées TestRunner pour exécuter les tests en mode ligne de commande ou application graphique :

- une application console : junit.textui.TestRunner qui est très rapide et adaptée à une intégration dans un processus de générations automatiques.
- une application graphique avec une interface Swing : junit.swingui.TestRunner
- une application graphique avec une interface AWT : junit.awtui.TestRunner

Quelle que soit l'application utilisée, les entités suivantes doivent être incluses dans le classpath :

- le fichier junit.jar
- les classes à tester et les classes des cas de tests
- les classes et bibliothèques dont toutes ces classes dépendent

Suite à l'exécution d'un cas de tests, celui-ci peut avoir un des trois états suivants :

- échoué : une exception de type AssertionFailedError est levée
- en erreur : une exception non émise par le framework et non capturée a été levée dans les traitements
- passé avec succès

L'échec d'un seul cas de tests entraîne l'échec du test complet.

L'échec d'un cas de tests peut avoir plusieurs origines :

- le cas de tests contient un ou plusieurs bugs
- le code à tester contient un ou plusieurs bugs
- le cas de tests est mal défini
- une combinaison des cas précédents

117.3.1. L'exécution des tests dans la console

L'utilisation de l'application console nécessite quelques paramètres :

Résultat :

```
C:\>java -cp junit.jar;. junit.textui.TestRunner MaClasseTest
```

Le seul paramètre obligatoire est le nom de la classe de tests. Celle-ci doit obligatoirement être sous la forme pleinement qualifiée si elle appartient à un package.

Résultat :

```
C:\>java -cp junit.jar;. junit.textui.TestRunner fr.jmdoudoux.dej.junit.MaClasseTest
```

Il est possible de faire appel au TestRunner dans une application en utilisant sa méthode run() à laquelle on passe en paramètre un objet de type Class qui encapsule la classe de tests à exécuter.

Résultat :

```
public class TestJUnit1 {
    public static void main(String[] args) {
        junit.textui.TestRunner.run(MaClasseTest.class);
    }
}
```

Le TestRunner affiche le résultat de l'exécution des tests dans la console.

La première ligne contient un caractère point pour chaque test exécuté. Lorsque de nombreux tests sont exécutés cela permet de suivre la progression.

Le temps total d'exécution en secondes est ensuite affiché sur la ligne "Time:"

Enfin, un résumé des résultats de l'exécution est affiché.

Résultat :

```
.
Time: 0,078

OK (1 test)
```

En cas d'erreur, la première ligne contient un F à la suite du caractère point correspondant au cas de tests en échec.

Le résumé de l'exécution affiche le détail de chaque cas de tests qui a échoué.

Résultat :

```
.F
Time: 0,063
There was 1 failure:
1) testCalculer(fr.jmdoudoux.dej.junit.MaClasseTest)junit.framework.AssertionFailedError:
 expected:<3> but was:<2>
        at fr.jmdoudoux.dej.junit.MaClasseTest.testCalculer(MaClasseTest.java:9)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
        at fr.jmdoudoux.dej.junit.MaClasseTest.main(MaClasseTest.java:13)

FAILURES!!!
Tests run: 1,  Failures: 1,  Errors: 0
```

Les cas en échec (failures) correspondent à une vérification faite par une méthode assertXXX() qui a échoué.

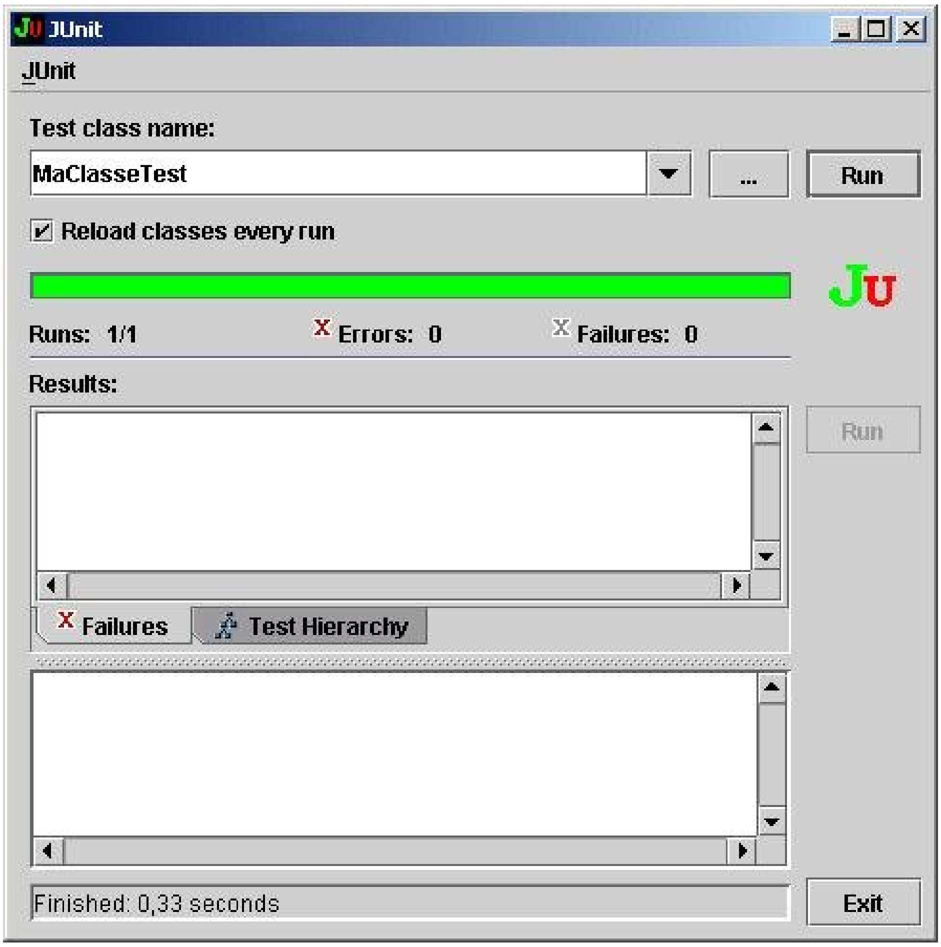
Les cas en erreur (errors) correspondent à la levée inattendue d'une exception lors de l'exécution du cas de tests.

117.3.2. L'exécution des tests dans une application graphique

Pour utiliser des classes de tests avec ces applications graphiques, il faut obligatoirement que les classes de tests et toutes celles dont elles dépendent soient incluses dans le CLASSPATH. Elles doivent obligatoirement être sous la forme de fichier .class non inclus dans un fichier jar.

Exemple :

```
C:\java\testjunit>java -cp junit.jar;. junit.swingui.TestRunner MaClasseTest
```



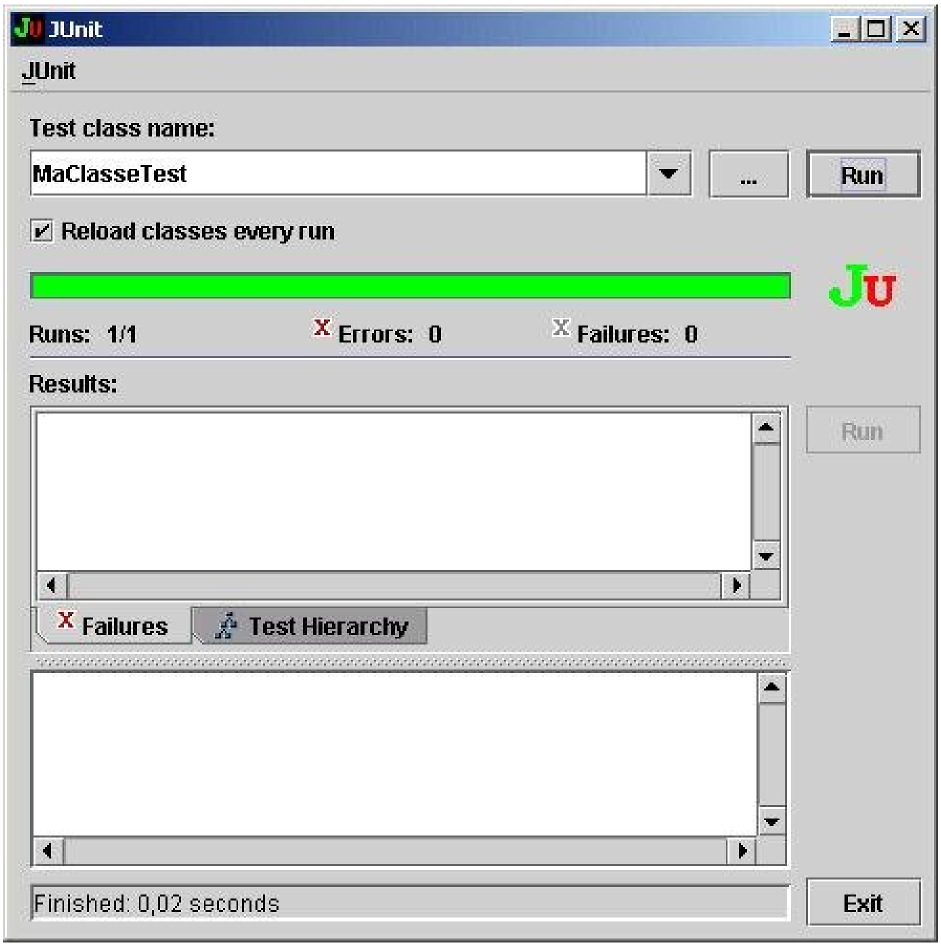
Test class name: MaClasseTest

☑ Reload classes every run

Runs: 1/1 Errors: 0 Failures: 0

Finished: 0,33 seconds

Il suffit de cliquer sur le bouton "Run" pour lancer l'exécution des tests.



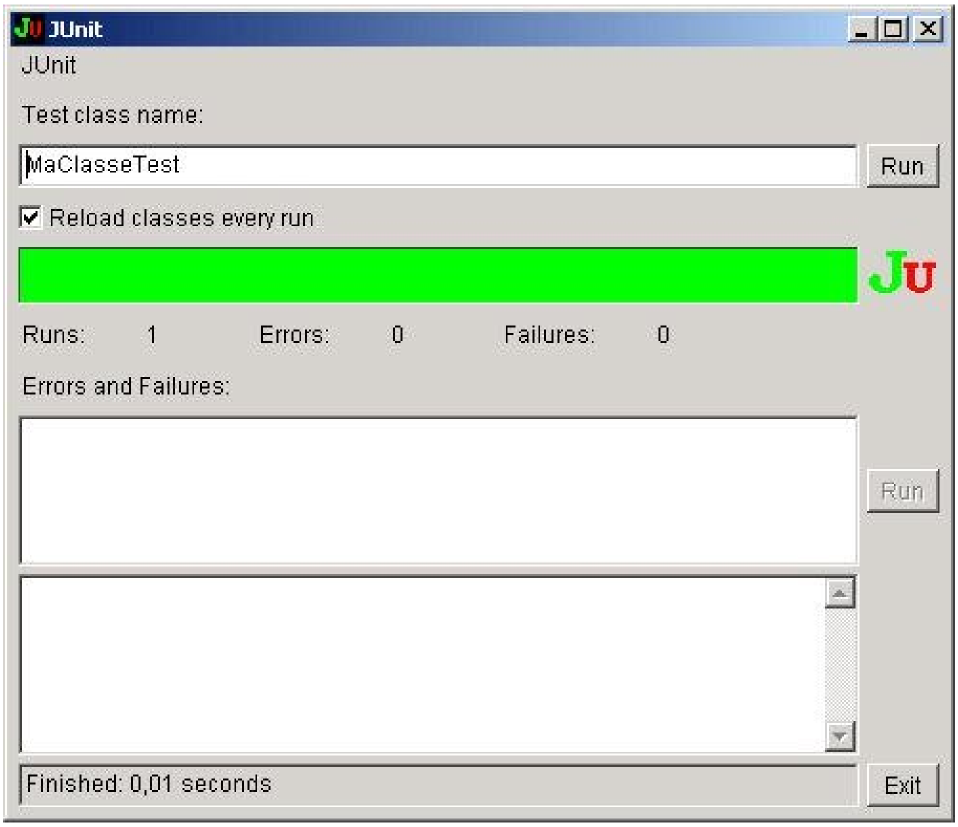
Test class name: MaClasseTest

☑ Reload classes every run

Runs: 1/1 Errors: 0 Failures: 0

Finished: 0,02 seconds

C:\java\testjunit>java -cp junit.jar;. junit.awtui.TestRunner MaClasseTest



Test class name: MaClasseTest

☑ Reload classes every run

Runs: 1 Errors: 0 Failures: 0

Finished: 0,01 seconds

La case à cocher "Reload classes every run" indique à JUnit de recharger les classes à chaque exécution. Ceci est très pratique car cela permet de modifier les classes tout en laissant l'application de tests ouverte.

Si un ou plusieurs tests échouent la barre de résultats n'est plus verte mais rouge. Dans ce cas, le nombre d'erreurs et d'échecs est affiché ainsi que leur liste complète. Il suffit d'en sélectionner un pour obtenir le détail de la raison du problème.

Il est aussi possible de ne réexécuter que le cas sélectionné.

117.3.3. L'exécution d'une classe de tests

Il est possible de définir une classe main() dans une classe de tests qui va se charger d'exécuter les tests.

Exemple :

```
public class MaClasseTest extends TestCase {
  ...
  public static void main(String[] args) {
    junit.textui.TestRunner.run(new TestSuite(MaClasseTest.class));
  }
}
```

117.3.4. L'exécution répétée d'un cas de tests

JUnit propose la classe junit.extensions.RepeatedTest qui permet d'exécuter plusieurs fois la même suite de tests.

Le constructeur de cette classe attend en paramètre une instance de la suite de tests et le nombre de répétitions de l'exécution.

Exemple :

```
package fr.jmdoudoux.dej.junit;

import junit.extensions.RepeatedTest;
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class PersonneTest extends TestCase {

  private Personne personne;

  public PersonneTest(String name) {
    super(name);
  }

...

  public static Test suite() {
    return new RepeatedTest(new TestSuite(PersonneTest.class), 5);
  }

  public static void main(String[] args) {
      junit.textui.TestRunner.run(suite());
  }
}
```

117.3.5. L'exécution concurrente de tests

JUnit propose la classe junit.extensions.ActiveTestSuite qui permet d'exécuter plusieurs suites de tests chacune dans un thread dédié. Ainsi l'exécution des suites de tests se fait de façon concurrente.

Exemple :

```
package fr.jmdoudoux.dej.junit;
```

```
import junit.extensions.ActiveTestSuite;
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class PersonneTest extends TestCase {

  private Personne personne;

  public PersonneTest(String name) {
    super(name);
  }

...

  public static Test suite() {
    TestSuite suite = new ActiveTestSuite();
    suite.addTest(new TestSuite(PersonneTest.class));
    suite.addTest(new TestSuite(PersonneTest.class));
    suite.addTest(new TestSuite(PersonneTest.class));
    return suite;

  }

  public static void main(String[] args) {
    junit.textui.TestRunner.run(suite());
  }
}
```

L'ensemble de la suite de tests ne se termine que lorsque tous les threads sont terminés.

Même si cela n'est pas recommandé, la classe ActiveTestSuite peut être utilisée comme un outil de charge rudimentaire. Il est ainsi possible de combiner l'utilisation des classes ActiveTestsuite et RepeatedTest.

Exemple :

```
package fr.jmdoudoux.dej.junit;

import junit.extensions.ActiveTestSuite;
import junit.extensions.RepeatedTest;
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class PersonneTest extends TestCase {

  private Personne personne;

  public PersonneTest(String name) {
    super(name);
  }

...

  public static Test suite() {
    TestSuite suite = new ActiveTestSuite();
    suite.addTest(new RepeatedTest(new TestSuite(PersonneTest.class), 10));
    suite.addTest(new RepeatedTest(new TestSuite(PersonneTest.class), 20));
    return suite;

  }

  public static void main(String[] args) {
    junit.textui.TestRunner.run(suite());
  }
}
```

117.4. Les suites de tests

Les suites de tests permettent de regrouper plusieurs tests dans une même classe. Ceci permet l'automatisation de l'ensemble des tests inclus dans la suite et de préciser leur ordre d'exécution.

Pour créer une suite, il suffit de créer une classe de type TestSuite et d'appeler la méthode addTest() pour chaque classe de tests à ajouter. Celle-ci attend en paramètre une instance de la classe de tests qui sera ajoutée à la suite. L'objet de type TestSuite ainsi créé doit être renvoyé par une méthode dont la signature est obligatoirement public static Test suite(). Celle-ci sera appelée par introspection par le TestRunner.

Il peut être pratique de définir une méthode main() dans la classe qui encapsule la suite de tests pour pouvoir exécuter le TestRunner de la console en exécutant directement la méthode statique Run(). Ceci évite de lancer JUnit sur la ligne de commandes.

Exemple :

```
import junit.framework.*;

public class ExecuterLesTests {

  public static Test suite() {
    TestSuite suite = new TestSuite("Tous les tests");
    suite.addTestSuite(MaClasseTest.class);
    suite.addTestSuite(MaClasse2Test.class);

    return suite;
  }

  public static void main(String args[]) {
    junit.textui.TestRunner.run(suite());
  }
}
```

Deux versions surchargées des constructeurs permettent de donner un nom à la suite de tests.

Un constructeur de la classe TestSuite permet de créer automatiquement par introspection une suite de tests contenant tous les tests de la classe fournie en paramètre.

Exemple :

```
import junit.framework.*;

public class ExecuterLesTests2 {

  public static Test suiteDeTests() {
    TestSuite suite = new TestSuite(MaClasseTest.class,"Tous les tests");
    return suite;
  }

  public static void main(String args[]) {
    junit.textui.TestRunner.run(suiteDeTests());
  }
}
```

Pour éviter d'avoir à gérer une suite de tests, il est possible d'utiliser la tâche Ant optionnelle junit qui exécutera un ensemble de cas de tests en fonction d'un filtre sur le nom des classes.

Exemple :

```
...
<junit printsummary="yes" haltonfailure="yes">
  ...
  <batchtest fork="yes">
    <fileset dir="${src.dir}">
       <include name="**/Test*.java" />
    </fileset>
  </batchtest>
</junit>
...
```

Le détail de la mise en oeuvre de JUnit avec Ant est couvert dans la section suivante.

La méthode addTestSuite() permet d'ajouter à une suite une autre suite.

117.5. L'automatisation des tests avec Ant

L'automatisation des tests fait par JUnit au moment de la génération de l'application est particulièrement pratique. Ainsi Ant propose une tâche optionnelle dédiée nommée junit pour exécuter un TestRunner dans la console.

Pour pouvoir utiliser cette tâche, les fichiers junit.jar (fourni avec JUnit) et optional.jar (fourni avec Ant) doivent être accessibles dans le CLASSSPATH.

Cette tâche possède plusieurs attributs dont aucun n'est obligatoire. Les principaux sont :

Attribut	Rôle	Valeur par défaut
printsummary	affiche un résumé statistique de l'exécution de chaque test	off
fork	exécution du TestRunner dans une JVM séparée	off
haltonerror	arrêt de la génération en cas d'erreur	off
haltonfailure	arrêt de la génération en cas d'échec d'un test	off
outfile	base du nom du fichier qui va contenir les résultats de l'exécution	

La tâche <junit> peut avoir les éléments fils suivants : <jvmarg>, <sysproperty>, <env>, <formatter>, <test>, <batchtest>.

L'élément <formatter> permet de préciser le format de sortie des résultats de l'exécution des tests. Il possède l'attribut type qui précise le format (les valeurs possibles sont : xml, plain ou brief) et l'attribut usefile qui précise si les résultats doivent être envoyés dans un fichier (les valeurs possibles sont : true ou false).

L'élément <test> permet de préciser un cas de tests simple ou une suite de tests selon le contenu de la classe précisée par l'attribut name. Cet élément possède de nombreux attributs et il est possible d'utiliser un élément fils de type <formatter> pour définir le format de sortie du test.

L'élément <batchtest> permet de réaliser toute une série de tests. Cet élément possède de nombreux attributs et il est possible d'utiliser un élément fils de type <formatter> pour définir le format de sortie des tests. Les différentes classes dont les tests sont à exécuter sont précisées par un élément fils <fileset>.

La tâche <junit> doit être exécutée après la compilation des classes à tester.

Exemple : extrait d'un fichier build.xml pour Ant

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<project name="TestAnt1" default="all">
   <description>Génération de l'application</description>
     <property name="bin" location="bin"/>
        <property name="src" location="src"/>
        <property name="build" location="build"/>
        <property name="doc" location="${build}/doc"/>
        <property name="lib" location="${build}/lib"/>

        <property name="junit_path" value="junit.jar"/>

        ...
        <target name="test" depends="compil" description="Executer les tests avec JUnit">
           <junit fork="yes" haltonerror="true" haltonfailure="on" printsummary="on">
              <formatter type="plain" usefile="false" />
              <test name="ExecuterLesTests"/>
              <classpath>
                 <pathelement location="${bin}"/>
                 <pathelement location="${junit_path}"/>
              </classpath>
           </junit>
        </target>
        ...

</project>
```

Cet exemple exécute les tests de la suite de tests encapsulée dans la classe ExecuterLesTests

117.6. JUnit 4

JUnit version 4 est une évolution majeure depuis les quelques années d'utilisation de la version 3.8.

Un des grands bénéfices de cette version est l'utilisation des annotations introduites dans Java 5. La définition des cas de tests et des tests ne se fait donc plus sur des conventions de nommage et sur l'introspection mais sur l'utilisation d'annotations ce qui facilite la rédaction des cas de tests.

Une compatibilité descendante est assurée avec les suites de tests de JUnit 3.8.

JUnit 4 requiert une version 5 ou ultérieure de Java.

Le nom du package des classes de JUnit est différent entre la version 3 et 4 :

- les classes de Junit 3 sont dans le package junit.framework.
- les classes de Junit 4 sont dans le package org.junit.

117.6.1. La définition d'une classe de tests

Une classe de tests n'a plus l'obligation d'étendre la classe TestCase sous réserve d'utiliser les annotations définies par JUnit et d'utiliser des imports static sur les méthodes de la classe org.junit.Assert.

Exemple :

```
package fr.jmdoudoux.dej.junit4;

import org.junit.*;
import static org.junit.Assert.*;

public class MaClasse {

}
```

117.6.2. La définition des cas de tests

Les méthodes contenant les cas de tests n'ont plus l'obligation d'utiliser la convention de nommage qui imposait de préfixer le nom des méthodes avec test.

Avec JUnit 4, il suffit d'annoter la méthode avec l'annotation @Test.

Il est ainsi possible d'utiliser n'importe quelle méthode comme cas de tests simplement en utilisant l'annotation @Test.

Exemple :

```
  @Test
  public void getNom() {
    assertEquals("Le nom est incorrect", "nom1", personne.getNom());
  }
```

Ceci permet d'utiliser le nom de méthode que l'on souhaite. Il est cependant conseillé de définir et d'utiliser une convention de nommage qui facilitera l'identification des classes de tests et des cas de tests. Il est par exemple possible de maintenir les conventions de nommage de JUnit 3.

L'annotation @Ignore permet de demander au framework d'ignorer un cas de tests. Les cas de tests dans ce cas sont marqués avec la lettre I lors de leur exécution en mode console.

Attention : l'utilisation de l'annotation @Ignore devrait être temporaire et justifiée. Son utilisation ne doit pas devenir une solution à certains problèmes.

JUnit 4 inclut deux nouvelles surcharges de la méthode assertEquals() qui permettent de comparer deux tableaux d'objets. La comparaison se fait sur le nombre d'occurrences dans les tableaux et sur l'égalité de chaque objet d'un tableau dans l'autre tableau.

117.6.3. L'initialisation des cas de tests

JUnit 3 imposait une redéfinition des méthodes setUp() et TearDown() pour définir des traitements exécutés systématiquement avant et après chaque cas de tests.

JUnit 4 propose simplement d'annoter la méthode exécutée avant avec l'annotation @Before et la méthode exécutée après avec l'annotation @After.

Exemple :

```
  @Before
  public void initialiser() throws Exception {
    personne = new Personne("nom1","prenom1");
  }

  @After
  public void nettoyer() throws Exception {
    personne = null;
  }
```

Il est possible d'annoter une ou plusieurs méthodes avec @Before ou @After. Dans ce cas, toutes les méthodes seront invoquées au moment correspondant à leur annotation.

Il n'est pas nécessaire d'invoquer explicitement les méthodes annotées avec @Before et @After d'une classe mère. Tant que ces méthodes ne sont pas redéfinies, elles seront automatiquement invoquées lors de l'exécution des tests :

- les méthodes annotées avec @Before de la classe mère seront invoquées avant celles de la classe fille
- les méthodes annotées avec @After de la classe fille seront invoquées avant celles de la classe mère

JUnit 4 propose simplement d'annoter une ou plusieurs méthodes exécutées avant l'exécution du premier cas de tests avec l'annotation @BeforeClass et une ou plusieurs méthodes exécutées après l'exécution de tous les cas de tests de la classe avec l'annotation @AfterClass.

Ces initialisations peuvent être très utiles notamment pour des connexions coûteuses à des ressources qu'il est préférable de ne réaliser qu'une seule fois plutôt qu'à chaque cas de tests. Ceci peut contribuer à améliorer les performances lors de l'exécution des tests.

117.6.4. Le test de la levée d'exceptions

Avec JUnit 3, pour vérifier la levée d'une exception dans un cas de tests, il faut entourer l'appel du traitement dans un bloc try/catch et invoquer la méthode fail() à la fin du bloc try.

JUnit 4 propose une annotation pour faciliter la vérification de la lever d'une exception.

L'attribut expected de l'annotation @Test attend comme valeur la classe de l'exception qui devrait être levée.

Exemple :

```
  @Test(expected=IllegalArgumentException.class)
  public void setNom() {
    personne.setNom("nom2");
    assertEquals("Le nom est incorrect", "nom2", personne.getNom());
    personne.setNom(null);
  }
```

Si lors de l'exécution du test l'exception du type précisée n'est pas levée (aucune exception levée ou une autre exception est levée) alors le test échoue.

Attention : l'utilisation de l'annotation ne permet que de vérifier que l'exception est levée. Pour vérifier des propriétés de l'exception, il est nécessaire d'utiliser le mécanisme utilisé avec JUnit 3 pour capturer l'exception et ainsi avoir accès aux membres de son instance.

117.6.5. L'exécution des tests

Les applications graphiques AWT et Swing permettant l'exécution et l'affichage des résultats des cas de tests ne sont plus fournies avec JUnit 4.

JUnit laisse le soin de cette restitution aux IDE qui intègrent JUnit 4 comme par exemple Eclipse.

Une autre grande différence dans la façon d'exécuter les cas de tests avec JUnit 4 concerne le fait qu'il n'y a plus de différence entre un test échoué (échec d'une méthode assert() ou appel à la méthode fail()) et un test en erreur (une exception inattendue est levée).

Lors de l'exécution, si un avertissement de type "AssertionFailedError: No tests found in XXX" est fourni par JUnit c'est qu'aucun cas de tests n'est fourni dans la classe (aucune méthode n'est annotée avec l'annotation @Test).

Dans une classe de tests, il est toujours possible de définir une méthode main() qui permette de demander l'exécution des cas de tests de la classe. Il faut invoquer la méthode main() de la classe org.junit.runner.JUnitCore.

Exemple :

```
package fr.jmdoudoux.dej.junit4;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class PersonneTest {

  private Personne personne;

  @Before
  public void initialiser() throws Exception {
    personne = new Personne("nom1","prenom1");
  }

  @After
  public void nettoyer() throws Exception {
    personne = null;
  }

  @Test
  public void personne() {
    assertNotNull("L'instance n'est pas créée", personne);
  }

...

  public static void main(String[] args) {
    org.junit.runner.JUnitCore.main("fr.jmdoudoux.dej.junit4.PersonneTest");
  }
}
```

117.6.6. Un exemple de migration de JUnit 3 vers JUnit 4

La section ci-dessous propose une classe qui encapsule des tests avec JUnit 3 et une classe qui propose des fonctionnalités équivalentes en JUnit 4.

Exemple avec JUnit 3 :

```
package fr.jmdoudoux.dej.junit;

import junit.framework.TestCase;

public class PersonneTest extends TestCase {

  private Personne personne;

  public PersonneTest(String name) {
    super(name);
  }

  protected void setUp() throws Exception {
    super.setUp();
    personne = new Personne("nom1","prenom1");
  }

  protected void tearDown() throws Exception {
    super.tearDown();
    personne = null;
  }

  public void testPersonne() {
    assertNotNull("L'instance n'est pas créée", personne);
  }

  public void testGetNom() {
    assertEquals("Le nom est incorrect", "nom1", personne.getNom());
  }

  public void testSetNom() {
    personne.setNom("nom2");
    assertEquals("Le nom est incorrect", "nom2", personne.getNom());
  }

  public void testGetPrenom() {
    assertEquals("Le prenom est incorrect", "prenom1", personne.getPrenom());
  }

  public void testSetPrenom() {
    personne.setPrenom("prenom2");
    assertEquals("Le prenom est incorrect", "prenom2", personne.getPrenom());
  }
}
```

Exemple avec JUnit 4 :

```
package fr.jmdoudoux.dej.junit4;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class PersonneTest {

  private Personne personne;

  @Before
  public void initialiser() throws Exception {
    personne = new Personne("nom1","prenom1");
  }

  @After
  public void nettoyer() throws Exception {
    personne = null;
  }

  @Test
  public void personne() {
    assertNotNull("L'instance n'est pas créée", personne);
  }

  @Test
  public void getNom() {
    assertEquals("Le nom est incorrect", "nom1", personne.getNom());
  }

  @Test(expected=IllegalArgumentException.class)
  public void setNom() {
    personne.setNom("nom2");
    assertEquals("Le nom est incorrect", "nom2", personne.getNom());
    personne.setNom(null);
  }

  @Test
  public void getPrenom() {
    assertEquals("Le prenom est incorrect", "prenom1", personne.getPrenom());
  }

  @Test
  public void setPrenom() {
    personne.setPrenom("prenom2");
    assertEquals("Le prenom est incorrect", "prenom2", personne.getPrenom());
  }
}
```

117.6.7. La limitation du temps d'exécution d'un cas de tests

JUnit 4 propose une fonctionnalité rudimentaire pour vérifier qu'un cas de tests s'exécute dans un temps maximum donné.

L'attribut timeout de l'annotation @Test attend comme valeur un délai maximum d'exécution exprimé en millisecondes.

Exemple :

```
package fr.jmdoudoux.dej.junit4;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class PersonneTest {
```

```
...

  @Test(timeout=100)
  public void compteur() {
    for(long i = 0 ; i < 999999999; i++) { long a = i + 1; }
  }

  public static void main(String[] args) {
    org.junit.runner.JUnitCore.main("fr.jmdoudoux.dej.junit4.PersonneTest");
  }
}
```

Si le temps d'exécution du cas de tests est supérieur au temps fourni, alors le cas de tests échoue.

Résultat :

```
JUnit version 4.3.1
......E
Time: 0,141
There was 1 failure:
1) compteur(fr.jmdoudoux.dej.junit4.PersonneTest)
java.lang.Exception: test timed out after 100 milliseconds
        at org.junit.internal.runners.TestMethodRunner.runWithTimeout
                (TestMethodRunner.java:68)
        at org.junit.internal.runners.TestMethodRunner.run
                (TestMethodRunner.java:43)
        at org.junit.internal.runners.TestClassMethodsRunner.invokeTestMethod
                (TestClassMethodsRunner.java:66)
        at org.junit.internal.runners.TestClassMethodsRunner.run
                (TestClassMethodsRunner.java:35)
        at org.junit.internal.runners.TestClassRunner$1.runUnprotected
                (TestClassRunner.java:42)
        at org.junit.internal.runners.BeforeAndAfterRunner.runProtected
                (BeforeAndAfterRunner.java:34)
        at org.junit.internal.runners.TestClassRunner.run(TestClassRunner.java:52)
        at org.junit.internal.runners.CompositeRunner.run(CompositeRunner.java:29)
        at org.junit.runner.JUnitCore.run(JUnitCore.java:130)
        at org.junit.runner.JUnitCore.run(JUnitCore.java:109)
        at org.junit.runner.JUnitCore.run(JUnitCore.java:100)
        at org.junit.runner.JUnitCore.runMain(JUnitCore.java:81)
        at org.junit.runner.JUnitCore.main(JUnitCore.java:44)
        at fr.jmdoudoux.dej.junit4.PersonneTest.main(PersonneTest.java:58)

FAILURES!!!
Tests run: 6,  Failures: 1
```

117.6.8. Les tests paramétrés

Cette section sera développée dans une version future de ce document

117.6.9. La rétro compatibilité

Il est possible d'exécuter des tests JUnit 4 dans une application d'exécution de Tests JUnit 3.

Pour cela, il faut dans la classe de tests, ajouter une méthode suite() qui retourne un objet de type junit.framework.Test. Cette méthode instancie un objet de type JUnit4TestAdapter qui attend comme paramètre de son constructeur l'objet class de la classe de tests.

Exemple :

```
package fr.jmdoudoux.dej.junit4;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class PersonneTest {

...

  public static junit.framework.Test suite() {
    return new JUnit4testAdapter(PersonneTest.class);
  }
}
```

L'exécution nécessite tout de même une version 5 ou supérieure de Java.

117.6.10. L'organisation des tests

Il est généralement préférable de n'avoir qu'un seul assert par test car un test ne devrait avoir qu'une seule raison d'échouer.

Exemple :

```
package fr.jmdoudoux.dej;

public class SecuriteHelper {

  public static boolean isMotDePasseValide(String mdp) {
    boolean resultat = true;

    if (mdp == null) {
      resultat = false;
      throw new IllegalArgumentException("le mot de passe n'est pas renseigne");
    } else {

      if (mdp.length() < 6 || mdp.length() > 15) {
        resultat = false;
      }

      if (!mdp.matches(".*[a-zA-Z]*[0-9]*[a-zA-Z]")) {
        resultat = false;
      }
    }
    return resultat;
  }
}
```

La méthode en exemple permet de valider un mot de passe en contrôlant quelques règles simples :

- lever une exception si l'argument est null (pour tester la levée de l'exception dans le cas de tests)
- la taille doit être comprise entre 6 et 15 caractères
- il faut au moins un chiffre entre deux caractères
- le dernier caractère doit être une lettre

Il est possible d'écrire une classe de tests ne possédant qu'un seul cas de tests avec plusieurs asserts.

Exemple :

```
package fr.jmdoudoux.dej;

import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.fail;

import org.junit.Test;

public class SecurirteHelperTest {

  @Test
  public void testIsMotDePasseValide() {
    try {
      SecuriteHelper.isMotDePasseValide(null);
      fail("Absence de la levee de l'exception IllegaleArgumentException");
    } catch (IllegalArgumentException iae) {
      // l'exception est levée
    }

    assertFalse("Le mot de passe est vide",
      SecuriteHelper.isMotDePasseValide(""));
    assertFalse("Le mot de passe est trop court",
      SecuriteHelper.isMotDePasseValide("aaa"));
    assertFalse("Le mot de passe est trop long",
      SecuriteHelper.isMotDePasseValide("aaaaaaaaaaaaaaaaaaaaaaaaaaaa"));
    assertFalse("Le mot de passe ne contient pas de chiffre",
      SecuriteHelper.isMotDePasseValide("aaaaa"));
    assertFalse("Le mot de passe contient un chiffre en derniere position",
      SecuriteHelper.isMotDePasseValide("aaaaa6"));

    assertTrue("Le mot de passe est valide",
      SecuriteHelper.isMotDePasseValide("aAa6Aa"));
    assertTrue("Le mot de passe est valide",
      SecuriteHelper.isMotDePasseValide("a@aA6aa"));
    assertTrue("Le mot de passe est valide",
      SecuriteHelper.isMotDePasseValide("abc456def"));
  }
```

Il est préférable d'écrire une méthode par cas de tests même si cela nécessite l'écriture de plus de code.

Exemple :

```
package fr.jmdoudoux.dej;

import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.fail;

import org.junit.Test;

public class SecurirteHelperTest {

  @Test(expected=IllegalArgumentException.class)
  public void testIsMotDePasseValideNull() {
      SecuriteHelper.isMotDePasseValide(null);
  }

  @Test
  public void testIsMotDePasseValideVide() {
    assertFalse("Le mot de passe est vide",
      SecuriteHelper.isMotDePasseValide(""));
  }

  @Test
  public void testIsMotDePasseValideTropCourt() {
    assertFalse("Le mot de passe est trop court",
      SecuriteHelper.isMotDePasseValide("aaa"));
  }

  @Test
  public void testIsMotDePasseValideTropLong() {
    assertFalse("Le mot de passe est trop long",
      SecuriteHelper.isMotDePasseValide("aaaaaaaaaaaaaaaaaaaaaaaaaaaa"));
  }

  @Test
  public void testIsMotDePasseValideSansChiffre() {
    assertFalse("Le mot de passe ne contient pas de chiffre",
      SecuriteHelper.isMotDePasseValide("aaaaa"));
  }

  @Test
  public void testIsMotDePasseValideChiffreEnDernier() {
    assertFalse("Le mot de passe contient un chiffre en derniere position",
      SecuriteHelper.isMotDePasseValide("aaaaa6"));
  }

  @Test
  public void testIsMotDePasseValideAvecMinMaj() {
    assertTrue("Le mot de passe est valide",
      SecuriteHelper.isMotDePasseValide("aAa6Aa"));
  }

  @Test
  public void testIsMotDePasseValideAvecArobase() {
    assertTrue("Le mot de passe est valide",
      SecuriteHelper.isMotDePasseValide("a@aA6aa"));
  }

  @Test
  public void testIsMotDePasseValideStandard() {
    assertTrue("Le mot de passe est valide",
      SecuriteHelper.isMotDePasseValide("abc456def"));
  }
}
```

En plus de s'assurer que tous les cas de tests sont exécutés même s'il y en a un qui échoue cela permet aussi de connaître plus précisément le nombre de cas de tests exécutés (10 au lieu de 1 dans l'exemple). Les cas de tests sont aussi plus simples donc plus maintenables.

Il est cependant possible d'utiliser plusieurs asserts dans un cas de tests si ceux-ci concernent un même cas fonctionnel.

Pour des cas plus concrets, il peut être nécessaire d'utiliser des méthodes de type setUp() ou TearDown() au besoin pour réduire la quantité de code nécessaire à la mise en place du contexte d'exécution de chaque cas de tests.

118. JUnit 5

Chapitre 118

Niveau :	🟩🟩🟨🟨⬜⬜ Supérieur
Version utilisée :	5.0

JUnit est un framework mature pour permettre l'écriture et l'exécution de tests automatisés.

JUnit 4 a été publié en 2005 pour permettre la prise en compte des annotations de Java 5.

JUnit 5, publié en 2017, utilise des fonctionnalités de Java 8 notamment les lambdas, les annotations répétées, ...

JUnit 5 est une réécriture intégrale du framework ayant plusieurs objectifs :

- le support et l'utilisation des nouvelles fonctionnalités de Java 8 : par exemple, les lambdas peuvent être utilisés dans les assertions
- une nouvelle architecture reposant sur plusieurs modules
- le support de différents types de tests
- un mécanisme d'extension qui permet l'ouverture vers des outils tiers ou des API

Les classes de tests JUnit 5 sont similaires à celles de JUnit 4 : basiquement, il suffit d'écrire une classe contenant des méthodes annotées avec @Test. Cependant, JUnit 5 est une réécriture complète de l'API contenue dans des packages différents de ceux de JUnit 4.

JUnit 5 apporte cependant aussi son lot de nouvelles fonctionnalités :

- les tests imbriqués
- les tests dynamiques
- les tests paramétrés qui offrent différentes sources de données
- un nouveau modèle d'extension
- l'injection d'instances en paramètres des méthodes de tests

Contrairement aux versions précédentes livrées en un seul jar, JUnit 5 est livré sous la forme de différents modules notamment pour répondre à la nouvelle architecture qui sépare :

- l'API
- le moteur d'exécution
- l'exécution et intégration

JUnit 5 ne peut être utilisée qu'avec une version supérieure ou égale à 8 de Java : il n'est pas possible d'utiliser une version antérieure.

Ce chapitre contient plusieurs sections :

♦ L'architecture
♦ Les dépendances
♦ L'écriture de tests
♦ L'écriture de tests standard
♦ Les assertions
♦ Les suppositions
♦ La désactivation de tests

- ♦ Les tags
- ♦ Le cycle de vie des instances de test
- ♦ Les tests imbriqués
- ♦ L'injection d'instances dans les constructeurs et les méthodes de tests
- ♦ Les tests répétés
- ♦ Les tests paramétrés
- ♦ Les tests dynamiques
- ♦ Les tests dans une interface
- ♦ Les suites de tests
- ♦ La compatibilité
- ♦ La comparaison entre JUnit 4 et JUnit 5

118.1. L'architecture

La version 5 de JUnit est composée de trois sous-projets :

- JUnit Platform : propose une API permettant aux outils de découvrir et exécuter des tests. Il définit une interface entre JUnit et les clients qui souhaitent exécuter les tests (IDE ou outils de build par exemple)
- JUnit Jupiter : propose une API reposant sur des annotations pour écrire des tests unitaires JUnit 5 et un TestEngine pour les exécuter
- JUnit Vintage : propose un TestEngine pour exécuter des tests JUnit 3 et 4 et ainsi assurer une compatibilité ascendante

L'objectif de cette architecture est de séparer les responsabilités des tests, d'exécution et d'extensions. Elle doit aussi permettre de faciliter l'intégration d'autres frameworks de tests dans JUnit.

118.2. Les dépendances

JUnit 5 utilise des fonctionnalités de Java 8 donc pour l'utiliser, il est nécessaire d'avoir une version 8 ou ultérieure de Java. La version 1.0 de JUnit 5 est diffusée en septembre 2017.

JUnit 5 est livré sous la forme de plusieurs jars qu'il faut ajouter au classpath en fonction des besoins. Le plus simple est d'utiliser Maven.

Group ID	Version	Artefact ID
org.unit.jupiter	5.0.0	junit-jupiter-api API pour l'écriture des tests avec JUnit Jupiter junit-jupiter-engine Implémentation du moteur d'exécution des tests JUnit Jupiter junit-jupiter-params Support des tests paramétrés avec JUnit Jupiter.
org.junit.platform	1.0.0	junit-platform-commons Utilitaires à usage interne de JUnit junit-platform-console Support pour la découverte et l'exécution des tests JUnit dans la console junit-platform-console-standalone Jar exécutable qui contient toutes les dépendances pour exécuter les tests dans une console junit-platform-engine API publique pour les moteurs d'exécution des tests

		junit-platform-gradle-plugin Support pour la découverte et l'exécution des tests JUnit avec Gradle junit-platform-launcher Support pour la découverte et l'exécution des tests JUnit avec des IDE et des outils de build junit-platform-runner Implémentation d'un Runner pour exécuter des tests JUnit 5 dans un environnement JUnit 4 junit-platform-suite-api Support pour l'exécution des suites de tests junit-platform-surefire-provider Support pour la découverte et l'exécution des tests JUnit avec le plugin Surefire de Maven
org.junit.vintage	4.12.0	junit-vintage-engine Implémentation d'un moteur d'exécution des tests écrits avec JUnit 3 et 4 dans la plateforme JUnit 5

Les dépendances peuvent être définies dans un projet Maven selon les besoins.

Exemple :

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <java.version>1.8</java.version>
  <junit.version>4.12</junit.version>
  <junit.jupiter.version>5.0.0</junit.jupiter.version>
  <junit.vintage.version>${junit.version}.0</junit.vintage.version>
  <junit.platform.version>1.0.0</junit.platform.version>
</properties>
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>${junit.jupiter.version}</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>${junit.jupiter.version}</version>
    <scope>test</scope>
  </dependency>
  <!-- Pour executer des tests ecrits avec un IDE
          qui ne supporte que les versions precedentes de JUnit -->
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>${junit.version}</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.platform</groupId>
    <artifactId>junit-platform-runner</artifactId>
    <version>${junit.platform.version}</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>${junit.jupiter.version}</version>
  </dependency>
  <dependency>
    <groupId>org.junit.vintage</groupId>
    <artifactId>junit-vintage-engine</artifactId>
```

```
      <version>${junit.vintage.version}</version>
    </dependency>
  </dependencies>
```

118.3. L'écriture de tests

L'écriture de classes de tests avec JUnit 5 est similaire à celle avec JUnit 4 : il faut définir des méthodes annotées avec des annotations de JUnit5. Certaines de ces annotations ont été renommées, notamment celles relatives au cycle de vie des instances de tests et d'autres ont été ajoutées. Parmi celles-ci, JUnit 5 propose une annotation permet de définir un nom d'affichage pour un cas de test, une autre permet de définir un test imbriqué sous la classe d'une classe interne.

Contrairement à JUnit 4, les classes et les méthodes de tests n'ont plus l'obligation d'être public. Avec JUnit 5, elles peuvent aussi être package friend (visibilité par défaut si aucune visibilité n'est précisée).

Les méthodes qui implémentent un cas de test utilisent des assertions pour effectuer des vérifications des résultats de l'exécution du test. Ces assertions ont été réécrites : certaines surcharges attendent en paramètres des interfaces fonctionnelles qui peuvent donc être définies avec des expressions Lambda. Quelques nouvelles assertions ont été ajoutées notamment une qui permet de définir un groupe d'assertions qui seront toutes évaluées ensemble.

118.3.1. Les annotations

JUnit Jupiter propose plusieurs annotations pour la définition et la configuration des tests. Ces annotations sont dans le package org.junit.jupiter.api.

Annotation	Rôle
@Test	La méthode annotée est un cas de test. Contrairement à l'annotation @Test de JUnit, celle-ci ne possède aucun attribut
@ParameterizedTest	La méthode annotée est un cas de test paramétré
@RepeatedTest	La méthode annotée est un cas de test répété
@TestFactory	La méthode annotée est une fabrique pour des tests dynamiques
@TestInstance	Configurer le cycle de vie des instances de tests
@TestTemplate	La méthode est un modèle pour des cas de tests à exécution multiple
@DisplayName	Définir un libellé pour la classe ou la méthode de test annotée
@BeforeEach	La méthode annotée sera invoquée avant l'exécution de chaque méthode de la classe annotée avec @Test, @RepeatedTest, @ParameterizedTest ou @Testfactory. Cette annotation est équivalente à @Before de JUnit 4
@AfterEach	La méthode annotée sera invoquée après l'exécution de chaque méthode de la classe annotée avec @Test, @RepeatedTest, @ParameterizedTest ou @Testfactory. Cette annotation est équivalente à @After de JUnit 4
@BeforeAll	La méthode annotée sera invoquée avant l'exécution de la première méthode de la classe annotée avec @Test, @RepeatedTest, @ParameterizedTest ou @Testfactory. Cette annotation est équivalente à @BeforeClass de JUnit 4. La méthode annotée doit être static sauf si le cycle de vie de l'instance est per-class
@AfterAll	La méthode annotée sera invoquée après l'exécution de toutes les méthodes de la classe annotées avec @Test, @RepeatedTest, @ParameterizedTest et @Testfactory. Cette annotation est équivalente à @AfterClass de JUnit 4.

	La méthode annotée doit être static sauf si le cycle de vie de l'instance est per-class.
@Nested	Indiquer que la classe annotée correspond à un test imbriqué
@Tag	Définir une balise sur une classe ou une méthode qui permettra de filtrer les tests exécutés. Cette annotation est équivalente aux Categories de JUnit 4 ou aux groups de TestNG
@Disabled	Désactiver les tests de la classe ou la méthode annotée. Cette annotation est similaire à @Ignore de JUnit 4
@ExtendWith	Enregistrer une extension

Les méthodes annotées avec @Test, @TestTemplate, @RepeatedTest, @BeforeAll, @AfterAll, @BeforeEach ou @AfterEach ne doivent pas retourner de valeur.

Les annotations de JUnit Jupiter peuvent être utilisées comme méta-annotation : il est possible de définir des annotations, elles-mêmes annotées avec ces annotations pour qu'elles héritent de leurs caractéristiques.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5.TestJUnit5;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

@Target({ ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Tag("Mon_Tag")
@Test
public @interface TestAvecMonTag {
}
```

118.4. L'écriture de tests standard

L'écriture de tests standard avec JUnit 5 est très similaire à celle de de JUnit 4. Basiquement, il faut écrire une classe contenant des méthodes annotées pour implémenter les cas de tests ou le cycle de vie des tests. Les méthodes qui implémentent des cas de tests utilisent des assertions pour faire les vérifications requises.

118.4.1. La définition d'une méthode de test

La définition d'un cas de test se fait avec l'annotation @org.junit.jupiter.api.Test utilisée sur une méthode. Cette annotation est similaire à celle de JUnit avec quelques différences :

- le nom du package est différent
- elle ne possède plus aucun attribut

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

public class MonTest {

  @Test
  public void simpleTest() {
    System.out.println("simpleTest");
    Assertions.assertTrue(true);
  }
}
```

Maintenant avec JUnit 5, ni les classes ni les méthodes de tests n'ont l'obligation d'être public : ils peuvent avoir la visibilité package friend (sans marqueur de visibilité).

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

public class MonTest {

  @Test
  void simpleTest() {
    System.out.println("simpleTest");
    Assertions.assertTrue(true);
  }
}
```

Attention : les méthodes private sont ignorées et ne sont pas exécutées.

118.4.2. La définition d'un libellé

Les classes et les méthodes de tests peuvent avoir un libellé qui sera affiché par les tests runners ou dans le rapport d'exécution des tests. Ce libellé est défini en utilisant l'annotation @org.junit.jupiter.api.DisplayName. Elle n'attend qu'un seul attribut obligatoire qui précise le libellé.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5.TestJUnit5;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

@DisplayName("Ma classe de test JUnit5")
public class MonTest {

  @Test
  @DisplayName("Mon cas de test")
  void premierTest() {
    // ...
  }
}
```

118.4.3. Le cycle de vie des tests

Comme avec JUnit 4, par défaut JUnit 5 créé une nouvelle instance pour exécuter chaque méthode de tests.

Une classe de test JUnit peut avoir des méthodes annotées pour définir des actions exécutées durant le cycle de vie des tests. Le cycle de vie d'un test peut être enrichi grâce à quatre annotations utilisées sur des méthodes pour réaliser des initialisations ou du ménage :

- @BeforeAll : exécutée une seule fois avant l'exécution du premier test de la classe
- @BeforeEach : exécutée avant chaque méthode de tests
- @AfterEach : exécutée après chaque méthode de tests
- @AfterAll : exécutée une seule fois après l'exécution de tous les tests de la classe

A part leur nom, ces annotations fonctionnent de manière similaire à leurs équivalents JUnit 4.

Par défaut, une nouvelle instance est créée pour exécuter chaque méthode de tests : il n'y a alors pas d'instance à utiliser pour invoquer les méthodes @BeforeAll et @AfterAll. Celles-ci doivent donc être statique.

Exemple (code Java 8) :

```java
package fr.jmdoudoux.dej.junit5;

import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class MonTest {

  @BeforeAll
  static void initAll() {
    System.out.println("beforeAll");
  }

  @BeforeEach
  void init() {
    System.out.println("beforeEach");
  }

  @AfterEach
  void tearDown() {
    System.out.println("afterEach");
  }

  @AfterAll
  static void tearDownAll() {
    System.out.println("afterAll");
  }

  @Test
  void simpleTest() {
    System.out.println("simpleTest");
    Assertions.assertTrue(true);
  }

  @Test
  void secondTest() {
    System.out.println("secondTest");
    Assertions.assertTrue(true);
  }
}
```

Résultat :

```
beforeAll
beforeEach
simpleTest
afterEach
beforeEach
secondTest
afterEach
afterAll
```

L'ordre d'exécution de méthodes annotées avec les mêmes annotations (par exemple @BeforeAll) est indéfini.

118.4.3.1. La définition de méthodes exécutées avant/après tous les tests

Une méthode annotée avec @BeforeAll sera exécutée avant l'exécution de la première méthode de tests. Avec le cycle de vie par défaut des instances de tests, il est obligatoire qu'une méthode annotée avec @BeforeAll soit statique.

L'annotation @BeforeAll de JUnit 5 est équivalente à l'annotation @BeforeClass de JUnit 4.

Une méthode annotée avec @AfterAll est exécutée après l'exécution de toutes les méthodes de tests de la classe.

Avec le cycle de vie par défaut des instances de tests, il est obligatoire qu'une méthode annotée avec @AfterAll soit statique.

L'annotation @AfterAll de JUnit 5 est équivalente à l'annotation @AfterClass de JUnit 4.

118.4.3.2. La définition de méthodes exécutées avant/après chaque tests

Une méthode annotée avec @BeforeEach sera exécutée avant chaque exécution d'une méthode de tests. Elle ne peut pas être statique sinon une exception de type JUnitException est levée à l'exécution.

L'annotation @BeforeEach de JUnit 5 est équivalente à l'annotation @Before de JUnit 4.

Une méthode annotée avec @AfterEach sera exécutée après chaque exécution d'une méthode de tests. Elle ne peut pas être statique sinon une exception de type JUnitException est levée à l'exécution.

L'annotation @AfterEach de JUnit 5 est équivalente à l'annotation @After de JUnit 4.

118.5. Les assertions

Les assertions ont pour rôle de faire des vérifications pour le test en cours. Si ces vérifications échouent, alors l'assertion lève une exception qui fait échouer le test.

JUnit Jupiter contient la plupart des assertions de JUnit 4 mais propose aussi ses propres annotations dont certaines surcharges peuvent utiliser les lambdas. Ces assertions sont des méthodes statiques de la classe org.junit.jupiter.Assertions.

Les assertions classiques permettent de faire des vérifications sur une instance ou une valeur ou effectuer des comparaisons. La classe org.junit.jupiter.Assertions contient de nombreuses méthodes statiques qui permettent d'effectuer différentes vérifications de données. Ces assertions permettent de comparer les données obtenues avec celles attendues dans un cas de test.

Egalité	Nullité	Exceptions
assertEquals()	assertNull()	assertThrows()
assertNotEquals()	assertNotNull()	
assertTrue()		
assertFalse()		
assertSame()		
assertNotSame()		

Les assertions classiques de JUnit 5 sont similaires à celles correspondantes de JUnit 4 :

- Les noms sont les mêmes
- Les assertions qui font des comparaisons attendent au moins deux paramètres qui sont dans l'ordre : la valeur attendue et la valeur actuelle. Il est important de respecter cet ordre pour que le message en cas d'échec soit

fiable

Les assertions JUnit 5 possèdent cependant des différences par rapport à leur équivalent JUnit 4 :

- elles ont été réécrites
- elles sont dans un package différent : org.junit.jupiter
- elles possèdent une ou plusieurs surcharges qui attendent en paramètre une interface fonctionnelle : il est ainsi possible d'utiliser des expressions lambdas
- lorsqu'il est présent dans les paramètres, le message affiché si l'assertion échoue est le dernier paramètre et peut être fourni sous la forme d'une chaîne de caractères ou d'une interface fonctionnelle de type Supplier<String> si la construction est coûteuse. Les paramètres les plus importants sont donc en premier.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotSame;
import static org.junit.jupiter.api.Assertions.assertTrue;

import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;
import org.junit.jupiter.api.Test;

public class PremiereClasseTest {

  @Test
  void monPremierTest() {
    assertTrue(true);
    assertTrue(this::isValide);
    assertTrue(true, () -> "Description " + "du cas " + "de test");
    List<String> attendu = Arrays.asList("e1", "e2", "e2");
    List<String> actual = new LinkedList<>(attendu);
    assertEquals(attendu, actual);
    assertEquals(attendu, actual, "Les listes ne sont pas égales");
    assertEquals(attendu, actual, () -> "Les listes " + "ne sont " + "pas égales");
    assertNotSame(attendu, actual, "Les instances sont les memes");
  }

  boolean isValide() {
    return true;
  }
}
```

En plus des assertions classiques, de nouvelles assertions sont aussi ajoutées :

- assertAll() pour regrouper différentes assertions qui seront toutes exécutées pour au final fournir la liste de celles en échec si au moins une échoue
- assertThrows() pour vérifier la lever d'une exception durant le test

La manière de vérifier une exception attendue change en JUnit 5. Avec JUnit 4, il fallait utiliser un attribut de l'annotation @Test ou protéger le code dans un bloc try/catch. Avec JUnit 5, il suffit d'utiliser l'assertion assertThrows().

118.5.1. L'assertion assertAll

L'assertion assertAll permet de regrouper plusieurs assertions qui seront toutes exécutés. L'assertion assertAll échoue si au moins une des assertions qu'elle regroupe échoue. Même si une assertion du groupe échoue, toutes les assertions du groupe seront évaluées.

Cette fonctionnalité très pratique, notamment pour vérifier l'état d'un POJO, peut être mise en oeuvre facilement grâce à l'utilisation d'une ou plusieurs expressions Lambda.

La méthode assertAll() de la classe Assertions possède plusieurs surcharges :

- static void assertAll(Executable... executables)
- static void assertAll(Stream<Executable> executables)
- static void assertAll(String heading, Executable... executables)
- static void assertAll(String heading, Stream<Executable> executables)

La méthode assertAll() vérifie que l'exécution de tous les Executable fournis ne lève aucune exception.

Si la vérification d'au moins une des assertions définies dans le groupe échoue alors la méthode assertAll() lève une exception de type org.opentest4j.MultipleFailuresError. L'affichage de l'erreur est de la responsabilité du moteur d'exécution des tests.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5;

import java.awt.Dimension;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

public class DimensionTest {
  @Test
  void verifierAttributs() {
    Dimension sut = new Dimension(800, 600);
    Assertions.assertAll("Dimensions non conformes",
      () -> Assertions.assertTrue(sut.getWidth() == 801, "Valeur de width erronee"),
      () -> Assertions.assertTrue(sut.getHeight() == 601, "Valeur de height erronee"));
  }
}
```

Résultat :

```
org.opentest4j.MultipleFailuresError: Dimensions non conformes (2 failures)
             Valeur de width erronee
             Valeur de height erronee
```

118.5.2. L'assertion assertArrayEquals

L'assertion assertArrayEquals vérifie que deux tableaux sont égaux.

La méthode assertArrayEquals() possède de nombreuses surcharges pour des tableaux de types boolean[], byte[], char[], double[], float[], int[], long[] et short[] et éventuellement un message sous la forme d'une chaîne de caractères ou d'un Supplier<String> :

- public static void assertArrayEquals(int[] expected, int[] actual)
- public static void assertArrayEquals(int[] expected, int[] actual, String message)
- public static void assertArrayEquals(int[] expected, int[] actual, Supplier<String> messageSupplier)

Plusieurs autres surcharges pour les tableaux de type float[] et double[] permettent de préciser une valeur qui sera utilisée comme marge lors des comparaisons des valeurs.

Les surcharges qui attendent des tableaux d'Object vérifie l'égalité de manière profonde.

Exemple (code Java 8) :

```
  @Test
  void verifierEgaliteTableaux() {
    Assertions.assertArrayEquals(new int[] { 1, 2, 3 }, new int[] { 1, 2, 3 },
      "Egalite des tableaux");
  }
```

Le test ci-dessous échoue car l'ordre des éléments des deux tableaux est différent.

Exemple (code Java 8) :

```
  @Test
  void verifierEgaliteTableaux() {
    Assertions.assertArrayEquals(new int[] { 1, 2, 3 }, new int[] { 3, 2, 1 },
      "Egalite des tableaux");
  }
```

Résultat :

```
  org.opentest4j.AssertionFailedError: Egalite
des tableaux ==> array contents differ at index [0], expected: <1> but was: <3>
```

Le test ci-dessous échoue car le nombre d'éléments des deux tableaux n'est pas identique.

Exemple (code Java 8) :

```
  @Test
  void verifierEgaliteTableaux() {
    Assertions.assertArrayEquals(new int[] { 1, 2, 3 }, new int[] { 1, 2, 3, 4 },
      "Egalite des tableaux");
  }
```

Résultat :

```
org.opentest4j.AssertionFailedError: Egalite des tableaux ==> array
lengths differ, expected: <3> but was: <4>
```

118.5.3. Les assertions assertEquals et assertNotEquals

L'assertion assertEquals permet de vérifier que la valeur actuelle et la valeur attendue soient égales.

La méthode assertEquals() de la classe Assertions possède de nombreuses surcharges pour différents types de données et éventuellement un message sous la forme d'une chaîne de caractères ou d'un Supplier<String>.

- public static void assertEquals(xxx expected, xxx actual)
- public static void assertEquals(xxx expected, xxx actual, String message)
- public static void assertEquals(xxx expected, xxx actual, Supplier<String> messageSupplier)

Le type de données des surcharges supportés sont : byte, char, double, float, int long, Object et short.

Pour les types double et float les surcharges sont différentes : elles acceptent en plus une valeur de type double nommée delta qui permet préciser une valeur qui servira de marge lors de la comparaison.

Exemple (code Java 8) :

```
  @Test
  void verifierEgalite() {
    Dimension sut = new Dimension(801, 601);
    Assertions.assertEquals(new Dimension(800, 600), sut, "Dimensions non egales");
  }
```

Résultat :

```
org.opentest4j.AssertionFailedError: Dimensions non egales ==> expected:
<java.awt.Dimension[width=800,height=600]> but was: <java.awt.Dimension[width=801,height=601]>
```

L'assertion assertNotEquals permet de vérifier que la valeur actuelle et la valeur attendue ne soient pas égales.

Contrairement à la méthode assertEquals(), la méthode assertNotEquals() ne possède des surcharges que pour le type Object.

- public static void assertNotEquals(Object expected, Object actual)

- public static void assertNotEquals(Object expected, Object actual, String message)
- public static void assertNotEquals(Object expected, Object actual, Supplier<String> messageSupplier)

Exemple (code Java 8) :

```
  @Test
  void verifierInegalite() {
    Dimension sut = new Dimension(800, 600);
    Assertions.assertNotEquals(new Dimension(800, 600), sut, "Dimensions egales");
  }
```

Résultat :

```
org.opentest4j.AssertionFailedError: Dimensions egales ==> expected: not equal but was:
<java.awt.Dimension[width=800,height=600]>
```

118.5.4. Les assertions assertTrue et assertFalse

L'assertion assertTrue permet de vérifier que la condition fournie est vraie.

La méthode assertTrue() de la classe Assertions possède plusieurs surcharges pour fournir la condition sous la forme d'un booléen ou d'un BooleanSupplier et éventuellement un message :

- public static void assertTrue(boolean condition)
- public static void assertTrue(boolean condition, String message)
- public static void assertTrue(boolean condition, Supplier<String> messageSupplier)
- public static void assertTrue(BooleanSupplier booleanSupplier)
- public static void assertTrue(BooleanSupplier booleanSupplier, String message)
- public static void assertTrue(BooleanSupplier booleanSupplier, Supplier<String> messageSupplier)

Exemple (code Java 8) :

```
  @Test
  void verifierTrue() {
    boolean bool = true;
    Assertions.assertTrue(bool);
    Assertions.assertTrue(MonTest::getBooleen, "Booleen different de true");
  }

  static boolean getBooleen() {
    return false;
  }
```

Résultat :

```
org.opentest4j.AssertionFailedError: Booleen different de true
```

L'assertion assertFalse permet de vérifier que la condition fournie est fausse.

La méthode assertFalse() de la classe Assertions possède plusieurs surcharges pour fournir la condition sous la forme d'un booléen ou d'un BooleanSupplier et éventuellement un message :

- public static void assertFalse(boolean condition)
- public static void assertFalse(boolean condition, String message)
- public static void assertFalse(boolean condition, Supplier<String> messageSupplier)
- public static void assertFalse(BooleanSupplier booleanSupplier)
- public static void assertFalse(BooleanSupplier booleanSupplier, String message)
- public static void assertFalse(BooleanSupplier booleanSupplier, Supplier<String> messageSupplier)

Exemple (code Java 8) :

```
  @Test
  void verifierFalse() {
    boolean bool = false;
    Assertions.assertFalse(bool);
    Assertions.assertFalse(MonTest::getBooleen, "Booleen different de false");
  }

  static boolean getBooleen() {
    return true;
  }
}
```

Résultat :

```
org.opentest4j.AssertionFailedError: Booleen different de false
```

118.5.5. L'assertion assertIterableEquals

L'assertionIterableEquals permet de vérifier que deux Iterables sont égaux de manière profonde, ce qui implique plusieurs vérifications :

- le nombre des éléments
- l'ordre des éléments
- lors de l'itération sur les éléments, chacun des éléments doit être égal à celui correspondant dans l'autre liste

La méthode assertIterableEquals() possède plusieurs surcharges qui permettent de préciser l'Iterable attendu, l'Iterable à vérifier et éventuellement un message sous la forme d'une chaîne de caractères ou d'un Supplier<String> :

- public static void assertIterableEquals(Iterable<?> expected, Iterable> actual)
- public static void assertIterableEquals(Iterable<?> expected, Iterable> actual, String message)
- public static void assertIterableEquals(Iterable<?> expected, Iterable> actual, Supplier<String> messageSupplier)

Exemple (code Java 8) :

```
  @Test
  void verifierIterableEquals() {
    Iterable<Integer> attendu = new ArrayList<>(Arrays.asList(1, 2, 3));
    Iterable<Integer> actuel = new ArrayList<>(Arrays.asList(1, 2, 3));
    Assertions.assertIterableEquals(attendu, actuel);
  }
```

L'exemple ci-dessous échoue car le nombre d'éléments des deux collections est différent.

Exemple (code Java 8) :

```
  @Test
  void verifierIterableEquals() {
    Iterable<Integer> attendu = new ArrayList<>(Arrays.asList(1, 2, 3));
    Iterable<Integer> actuel = new ArrayList<>(Arrays.asList(1, 2));
    Assertions.assertIterableEquals(attendu, actuel);
  }
```

Résultat :

```
org.opentest4j.AssertionFailedError: iterable lengths differ, expected: <3> but was: <2>
```

L'exemple ci-dessous échoue car l'ordre des éléments des deux collections est différent.

Exemple (code Java 8) :

```
  @Test
  void verifierIterableEquals() {
    Iterable<Integer> attendu = new ArrayList<>(Arrays.asList(1, 2, 3));
    Iterable<Integer> actuel = new ArrayList<>(Arrays.asList(3, 2, 1));
    Assertions.assertIterableEquals(attendu, actuel);
  }
```

Résultat :

```
org.opentest4j.AssertionFailedError: iterable
contents differ at index [0], expected: <1> but was: <3>
```

118.5.6. L'assertion assertLinesMatch

L'assertion assertLinesMatch vérifie que les éléments d'une List<String> sont en correspondance avec une autre List<String>. Cette assertion est un cas spécifique de comparaison de collections.

La méthode assertLinesMatch() attend donc en paramètres deux List<String>.

static void assertLinesMatch(List<String> expectedLines, List<String> actualLines)

La correspondance est vérifiée en utilisant plusieurs règles pour chaque élément des listes :

- vérifier que les deux éléments sont égaux (avec la méthode equals()) : si c'est le cas, passage à l'élément suivant
- sinon l'élément expected va être utilisé comme une expression régulière pour vérifier s'il y a correspondance avec l'élément actual : si c'est le cas, passage à l'élément suivant
- sinon vérifie si l'élément expected est un marqueur de type avance rapide : Si c'est la cas, passage à l'élément suivant

Dans sa forme la plus simple, elle compare simplement les éléments des deux listes.

Exemple (code Java 8) :

```
  @Test
  void verifierLinesMatch() {
    List<String> expectedLines = Arrays.asList("A1", "A2", "A3", "A4");
    List<String> emails = Arrays.asList("A1", "A2", "A3", "A4");
    Assertions.assertLinesMatch(expectedLines, emails);
  }
```

Mais il est aussi possible d'utiliser des expressions régulières pour vérifier la valeur d'un élément.

Exemple (code Java 8) :

```
  @Test
  void verifierLinesMatch() {
    List<String> expectedLines = Arrays.asList("(.*)@(.*)", "(.*)@(.*)");
    List<String> emails = Arrays.asList("test@gmail.com", "jm@test.fr");
    Assertions.assertLinesMatch(expectedLines, emails);
  }
```

Il est aussi possible d'ignorer un ou plusieurs éléments durant la comparaison grâce à un marqueur d'avance rapide : ils peuvent par exemple permettre d'ignorer des éléments dont la valeur change à chaque exécution.

Un marqueur d'avance rapide commence et termine par «>>» et doit posséder au moins quatre caractères.

Exemple (code Java 8) :

```
  @Test
  void verifierLinesMatch() {
    List<String> expectedLines = Arrays.asList("(.*)@(.*)", ">>>>", "(.*)@(.*)");
    List<String> emails = Arrays.asList("test@gmail.com", "test", "email", "jm@test.fr");
    Assertions.assertLinesMatch(expectedLines, emails);
  }
```

Il est possible de mettre une description entre les doubles chevrons : cette description sera ignorée.

Exemple (code Java 8) :

```
  @Test
  void verifierLinesMatch() {
    List<String> expectedLines =
      Arrays.asList("(.*)@(.*)", ">> aller au dernier >>", "(.*)@(.*)");
    List<String> emails = Arrays.asList("test@gmail.com", "test", "email","jm@test.fr");
    Assertions.assertLinesMatch(expectedLines, emails);
  }
```

Il est possible de préciser un nombre exact d'éléments à ignorer.

Exemple (code Java 8) :

```
  @Test
  void verifierLinesMatch() {
    List<String> expectedLines = Arrays.asList("A1", ">> 2 >>", "A4");
    List<String> emails = Arrays.asList("A1", "A2", "A3", "A4");
    Assertions.assertLinesMatch(expectedLines, emails);
  }
```

Si le nombre d'éléments à ignorer ne peut être atteint ou est insuffisant alors la méthode lève une exception.

Exemple (code Java 8) :

```
  @Test
  void verifierLinesMatch() {

    List<String> expectedLines = Arrays.asList("A1", ">> 1 >>", "A4");
    List<String> emails = Arrays.asList("A1", "A2", "A3", "A4");
    Assertions.assertLinesMatch(expectedLines, emails);
  }
```

Résultat :

```
org.opentest4j.AssertionFailedError:
expected line #3:`A4` doesn't match ==> expected: <A1 >> 1 >> A4> but was: <A1
A2
A3
A4>
```

118.5.7. Les assertions assertNull et assertNotNull

L'assertion assertNull permet de vérifier que l'objet fourni en paramètre est null.

La méthode assertNull() de la classe Assertions possède plusieurs surcharges pour fournir l'objet et éventuellement un message sous la forme d'une chaîne de caractères ou d'un Supplier<String> :

- public static void assertNull(Object actual)
- public static void assertNull(Object actual, String message)
- public static void assertNull(Object actual, Supplier<String> messageSupplier)

Exemple (code Java 8) :

```
  @Test
  void verifierNull() {
    Object sut = new Dimension(800, 600);
    Assertions.assertNull(sut);
  }
```

Résultat :

```
org.opentest4j.AssertionFailedError:
          expected: <null> but was: <java.awt.Dimension[width=800,height=600]>
```

L'assertion assertNotNull permet de vérifier que l'objet fourni en paramètre n'est pas null.

La méthode assertNotNull() de la classe Assertions possède plusieurs surcharges pour fournir l'objet et éventuellement un message sous la forme d'une chaîne de caractères ou d'un Supplier<String> :

- public static void assertNotNull(Object actual)
- public static void assertNotNull(Object actual, String message)
- public static void assertNotNull(Object actual, Supplier<String> messageSupplier)

Exemple (code Java 8) :

```
  @Test
  void verifierNotNull() {
    Object sut = null;
    Assertions.assertNotNull(sut);
  }
```

Résultat :

```
org.opentest4j.AssertionFailedError: expected: not <null>
```

118.5.8. Les assertions assertSame et assertNotSame

L'assertion assertSame permet de vérifier que les objets fournis en paramètre sont le même objet.

La méthode assertSame() de la classe Assertions possède plusieurs surcharges pour fournir les deux objets et éventuellement un message sous la forme d'une chaîne de caractères ou d'un Supplier<String> :

- public static void assertSame(Object expected, Object actual)
- public static void assertSame(Object expected, Object actual, String message)
- public static void assertSame(Object expected, Object actual, Supplier<String> messageSupplier)

Exemple (code Java 8) :

```
  @Test
  void verifierSame() {
    Object sut = new Dimension(800, 600);
    Object expected = new Dimension(800, 600);
    Assertions.assertSame(sut, expected);
  }
```

Résultat :

```
org.opentest4j.AssertionFailedError:
expected: java.awt.Dimension@10bdf5e5<java.awt.Dimension[width=800,height=600]>
but was: java.awt.Dimension@6e1ec318<java.awt.Dimension[width=800,height=600]>
```

L'assertion assertNotSame permet de vérifier que les objets fournis en paramètre ne sont pas le même objet.

La méthode assertNotSame() de la classe Assertions possède plusieurs surcharges pour fournir les deux objets et éventuellement un message sous la forme d'une chaîne de caractères ou d'un Supplier<String> :

- public static void assertNotSame(Object expected, Object actual)
- public static void assertNotSame(Object expected, Object actual, String message)
- public static void assertNotSame(Object expected, Object actual, Supplier<> messageSupplier)

Exemple (code Java 8) :

```
  @Test
  void verifierNotSame() {
    Object sut = new Dimension(800, 600);
    Object expected = sut;
    Assertions.assertNotSame(sut, expected);
  }
```

Résultat :

```
org.opentest4j.AssertionFailedError:
expected: not same but was: <java.awt.Dimension[width=800,height=600]>
```

118.5.9. L'assertion assertThrows

Contrairement à JUnit 4 qui utilisait des attributs de l'annotation @Test, la vérification de la levée d'une exception avec JUnit 5 se fait avec une assertion, ce qui rend plus homogène cette fonctionnalité.

L'assertion assertThrows vérifie que l'exécution de la méthode passée en paramètre lève l'exception précisée : si ce n'est pas le cas, elle lève une exception pour faire échouer le test.

La méthode assertThrows() possède plusieurs surcharges qui attendent en paramètres le type de l'exception qui doit être levée, une interface fonctionnelle de type Executable qui est le code à exécuter et éventuellement un message sous la forme d'une chaîne de caractères ou d'un Supplier<String>

- static <T extends Throwable> T assertThrows(Class<T> expectedType, Executable executable)
- static <T extends Throwable> T assertThrows(Class<T> expectedType, Executable executable, String message)
- static <T extends Throwable> T assertThrows(Class<T> expectedType, Executable executable, Supplier<String> messageSupplier)

Exemple (code Java 8) :

```
  @Test
  void verifierException() {
    String valeur = null;
    assertThrows(NumberFormatException.class, () -> {
      Integer.valueOf(valeur);
    });
  }
```

Il est aussi possible de préciser une super-classe de l'exception attendue.

L'exemple de test ci-dessous réussi car l'exception NumberFormatException hérite de l'exception IllegalArgumentException.

Exemple (code Java 8) :

```
  @Test
  void verifierException() {
    String valeur = null;
    assertThrows(IllegalArgumentException.class, () -> {
      Integer.valueOf(valeur);
    });
  }
```

Si aucune exception n'est levée par le par les traitements fournis ou si une exception différente est levée alors la méthode assertThrows() lève une exception qui fait échouer le test.

L'exemple de test ci-dessous échoue car aucune exception n'est levée.

Exemple (code Java 8) :

```
  @Test
  void verifierException() {
    String valeur = "1";
    assertThrows(NumberFormatException.class, () -> {
      Integer.valueOf(valeur);
    });
  }
```

Résultat :

```
org.opentest4j.AssertionFailedError:
Expected java.lang.NumberFormatException to be thrown, but nothing was thrown.
```

Si l'exception est levée par le code passé en paramètre, alors celle-ci est fournie en retour de l'exécution de la méthode assertThrows(). Il est alors aussi possible de faire des vérifications sur l'exception obtenue.

Exemple :

```
package fr.jmdoudoux.dej.junit5;

public class MaClasse {

  public void maMethode() {
    throw new RuntimeException("mon message d'erreur");
  }
}
```

Exemple (code Java 8) :

```
  @Test
  void verifierException() {
    MaClasse sut = new MaClasse();
    RuntimeException excep = assertThrows(RuntimeException.class, sut::maMethode);
    assertAll(() -> assertEquals("message erreur", excep.getMessage()),
              () -> assertNull(excep.getCause()));
  }
```

Résultat :

```
org.opentest4j.MultipleFailuresError: Multiple Failures (1 failure)
       expected: <message erreur> but was: <mon message d'erreur>
```

118.5.10. Les assertions assertTimeout et assertTimeoutPreemptively

Les assertions assertTimeout et assertTimeoutPreemptively vérifie que les traitements fournis en paramètre s'exécutent avant le délai précisé. La différence entre les deux est que assertTimeoutPreemptively interrompt l'exécution des traitements si le délai est dépassé.

La méthode assertTimeout() possède plusieurs surcharges qui permettent de préciser la durée maximale d'exécution (timeout), les traitements à exécuter sous la forme d'un Executable ou d'un ThrowingSupplier et éventuellement un message sous la forme d'une chaîne de caractères ou d'un Supplier<String> :

- public static void assertTimeout(Duration timeout, Executable executable)
- public static void assertTimeout(Duration timeout, Executable executable, String message)
- public static void assertTimeout(Duration timeout, Executable executable, Supplier<String> messageSupplier)
- public static void assertTimeout(Duration timeout, ThrowingSupplier<T> supplier, String message)
- public static void assertTimeout(Duration timeout, ThrowingSupplier<T> supplier, Supplier<String> messageSupplier)

Exemple (code Java 8) :

```
  @Test
  void verifierTimeout() {
    Assertions.assertTimeout(Duration.ofMillis(200), () -> {
      return "";
    });
    Assertions.assertTimeout(Duration.ofSeconds(1), DimensionTest::traiter);
  }

  private static String traiter() throws InterruptedException {
    Thread.sleep(2000);
    return "";
  }
```

Résultat :

```
org.opentest4j.AssertionFailedError: execution exceeded timeout of 1000 ms by 1001 ms
```

La méthode assertTimeoutPreemptively() possède plusieurs surcharges qui permettent de préciser la durée maximale d'exécution (timeout), les traitements à exécuter sous la forme d'un Executable ou d'unThrowingSupplier et éventuellement un message sous la forme d'une chaîne de caractères ou d'un Supplier<String> :

- static void assertTimeoutPreemptively(Duration timeout, Executable executable)
- static void assertTimeoutPreemptively(Duration timeout, Executable executable, String message)
- static void assertTimeoutPreemptively(Duration timeout, Executable executable, Supplier<String> messageSupplier)
- static <T> T assertTimeoutPreemptively(Duration timeout, ThrowingSupplier<T> supplier)
- static <T> T assertTimeoutPreemptively(Duration timeout, ThrowingSupplier<T> supplier, String message)
- static <T> T assertTimeoutPreemptively(Duration timeout, ThrowingSupplier<T> supplier, Supplier<String> messageSupplier)

Exemple (code Java 8) :

```
  @Test
  void verifierTimeoutPreemptively() {
    Assertions.assertTimeoutPreemptively(Duration.ofMillis(200), () -> {
      return "";
    });

    Assertions.assertTimeoutPreemptively(Duration.ofSeconds(1), MonTest::traiter);
  }
```

Résultat :

```
org.opentest4j.AssertionFailedError: execution timed out after 1000 ms
```

118.5.11. L'assertion fail

Les surcharges de la méthode fail() permettent de faire échouer le test en levant une exception de type AssertionFailedError.

Méthode	Rôle
static <V> V fail(String message)	Faire échouer le test avec le message indiquant la raison de l'échec
static <V> V fail(String message, Throwable cause)	Faire échouer le test avec le message indiquant la raison et la cause de l'échec
static <V> V fail(Supplier<String> messageSupplier)	Faire échouer le test avec le message fourni par le Supplier indiquant la raison de l'échec
static <V> V fail(Throwable cause)	Faire échouer le test avec la cause de l'échec

Exemple (code Java 8) :

```
  @Test
  void monTest() {
    fail("la raison de l'échec du test");
  }
```

118.5.12. L'utilisation d'assertions de bibliothèques tiers

JUnit Jupiter propose un ensemble d'assertions qui peuvent suffire pour des tests simples mais il est aussi possible d'utiliser d'autres bibliothèques d'assertions telles que :

- AssertJ
- Hamcrest
- Truth

Ces bibliothèques sont compatibles avec JUnit 5. C'est d'autant plus nécessaire que JUnit 5 a fait le choix de ne pas fournir d'implémentation de l'assertion assertThat() qui attendait en paramètre un objet de type Matcher de la bibliothèque Hamcrest. JUnit 5 préfère laisser les développeurs utiliser ces bibliothèques tierces.

Remarque : contrairement à JUnit 4, la bibliothèque Hamcrest n'est donc plus fournie en standard avec JUnit 5. Elle peut cependant être utilisée avec JUnit 5 si elle est ajoutée au classpath.

L'exemple ci-dessous utilise l'assertion assertThat de la bibliothèque Hamcrest.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5;

import static org.hamcrest.CoreMatchers.equalTo;
import static org.hamcrest.CoreMatchers.is;
import static org.hamcrest.MatcherAssert.assertThat;

import org.junit.jupiter.api.Test;

public class MaClasseTest {

  @Test
  void testAvecHamcrest() {
    assertThat(1 + 2, is(equalTo(3)));
  }
}
```

118.6. Les suppositions

Les suppositions permettent de conditionner l'exécution de tout ou partie d'un cas de test. Elles peuvent interrompre un test (sans le faire échouer) si une condition est remplie ou peuvent conditionner l'exécution de certains traitements d'un test selon une condition.

Si l'évaluation d'une supposition échoue alors l'exécution du test est interrompue car elle lève une exception de type org.opentest4j.TestAbortedException. Dans ce cas, le test interrompu est considéré comme désactivé.

Un cas de test vide (qui n'a réalisé aucune assertion) est évalué comme réussi (vert).

Dans les deux cas, le test est réussi.

JUnit Jupiter propose des suppositions sous la forme de méthodes statiques de la classe org.junit.jupiter.Assumptions. La classe Assumptions possède plusieurs méthodes statiques : assumeTrue(), assumeFalse() et assumingThat() dont certaines surcharges attendent en paramètre des interfaces fonctionnelles qui permettent donc d'utiliser des expressions Lambdas.

La supposition assumeTrue permet d'exécuter la suite des traitements du test uniquement si la valeur booléenne fournie est true.

La méthode assumeTrue() possède plusieurs surcharges qui attendent en paramètre la valeur booléenne sous la forme d'un boolean ou d'un BooleanSupplier et éventuellement un message affiché si le booléen vaut false sous la forme d'un String ou d'un Supplier<String>.

- static void assumeTrue(boolean assumption)
- static void assumeTrue(boolean assumption, String message)
- static void assumeTrue(BooleanSupplier assumptionSupplier)
- static void assumeTrue(boolean assumption, Supplier<String> messageSupplier)
- static void assumeTrue(BooleanSupplier assumptionSupplier, String message)
- static void assumeTrue(BooleanSupplier assumptionSupplier, Supplier<String> messageSupplier)

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5;

import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.junit.jupiter.api.Assumptions.assumeTrue;

import org.junit.jupiter.api.Test;

public class MaClasseTest {

  @Test
  void testSousWindows() {
    System.out.println(System.getenv("OS"));
    assumeTrue(System.getenv("OS").startsWith("Windows"));
    assertTrue(false);
  }
}
```

La supposition assumeFalse permet d'exécuter la suite des traitements du test uniquement si la valeur booléenne fournie est false. C'est l'inverse de la supposition assumeTrue().

La méthode assumeFalse() possède plusieurs surcharges qui attendent en paramètre la valeur booléenne sous la forme d'un boolean ou d'un BooleanSupplier et éventuellement un message affiché si le booléen vaut false sous la forme d'un String ou d'un Supplier<String>.

- static void assumeFalse(boolean assumption)
- static void assumeFalse(boolean assumption, String message)
- static void assumeFalse(BooleanSupplier assumptionSupplier)
- static void assumeFalse(boolean assumption, Supplier<String> messageSupplier)
- static void assumeFalse(BooleanSupplier assumptionSupplier, String message)
- static void assumeFalse(BooleanSupplier assumptionSupplier, Supplier<String> messageSupplier)

La supposition assumingThat permet d'exécuter le traitement fourni uniquement si la valeur booléenne fournie est true.

La méthode assumingThat() possède deux surcharges qui attendent en paramètre la valeur booléenne sous la forme d'un boolean ou d'un BooleanSupplier et un objet de type Executable qui contient les traitements à exécuter.

- static void assumingThat(boolean assumption, Executable executable)
- static void assumingThat(BooleanSupplier assumptionSupplier, Executable executable)

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5;

import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.junit.jupiter.api.Assumptions.assumingThat;

import java.io.File;
import org.junit.jupiter.api.Test;

public class MaClasseTest {

  @Test
  void testAvecSupposition() {
    assumingThat(System.getenv("OS").startsWith("Windows"), () -> {
      assertTrue(new File("C:/Windows").exists(), "Repertoire Windows inexistant");
    });
    assertTrue(true);
  }
}
```

118.7. La désactivation de tests

L'annotation @org.junit.jupiter.api.Disabled permet de désactiver un test.

Il est possible de fournir une description optionnelle de la raison de la désactivation

L'annotation @Disabled peut être utilisée sur une méthode ou sur une classe. L'utilisation sur une méthode désactive uniquement la méthode concernée.

Exemple (code Java 8) :

```
  @Test
  @Disabled("A écrire plus tard")
  void monTest() {
    fail("Non implémenté");
  }
```

L'utilisation de l'annotation sur une classe désactive toutes les méthodes de tests de la classe.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5.testJUnit5;

import static org.junit.jupiter.api.Assertions.fail;

import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

@DisplayName("Ma classe de test JUnit5")
@Disabled
public class MonTest {
  @Test
  @DisplayName("Cas de test")
  void monTest() {
    fail("un test en echec");
  }
}
```

L'annotation @Disabled de JUnit 5 est équivalente à l'annotation @Ignore de JUnit 4.

118.8. Les tags

Les classes et les méthodes de tests peuvent être tagguées pour permettre d'utiliser ces tags ultérieurement pour déterminer les tests à exécuter. Ils peuvent par exemple être utilisés pour créer différents scénarios de tests ou pour exécuter les tests uniquement sur des environnements dédiés.

Le libellé d'un tag ne doit pas :

- être null ou une chaîne vide
- contenir d'espace
- contenir les caractères réservés : , ( ) & | !

Les tags sont définis grâce à l'annotation @org.junit.jupiter.api.Tag. Elle peut être utilisée sur une classe de tests et/ou sur des méthodes d'une telle classe.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5;

import static org.junit.Assert.assertTrue;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

@Tag("principal")
public class MaClasseTest {

  @Test
  @Tag("general")
  void testCas1() {
    assertTrue(true);
  }

  @Test
  @Tag("specifique")
  void testCas2() {
    assertTrue(true);
  }
}
```

Il est possible d'utiliser plusieurs annotations @Tag sur un même élément pour définir plusieurs étiquettes.

Les Tags sont équivalents aux Categories de JUnit 4.

118.9. Le cycle de vie des instances de test

Dans le cycle de vie des instances de tests, les méthodes de tests sont des méthodes annotées avec @Test, @ParameterizedTest, @RepeatedTest, @TestFactory ou @TestTemplate.

Par défaut, JUnit créé une nouvelle instance pour chaque test avant d'exécuter la méthode concernée. Le but de ce comportement est d'exécuter le test de manière isolée et ainsi d'éviter les effets de bord liés à l'exécution des autres tests.

Il est possible de modifier ce comportement par défaut en utilisant l'annotation @org.junit.jupiter.api.TestInstance.

Elle ne possède qu'un seul attribut de type TestInstance.Lifecycle qui est une énumération possédant deux valeurs :

Valeur	Rôle
PER_CLASS	Une seule instance est créée pour tous les tests d'une même classe
PER_METHOD	Une nouvelle instance est créée pour exécuter chaque méthode de test. C'est la valeur par défaut de l'attribut de l'annotation @TestInstance

Donc pour modifier le comportement par défaut et demander à JUnit 5 d'exécuter toutes les méthodes de tests de la même instance, il faut annoter la classe avec @TestInstance(Lifecycle.PER_CLASS). Si les tests utilisent des variables d'instances, il est possible de réinitialiser leur état dans des méthodes annotées avec @BeforeEach et/ou @AfterEach.

Le mode PER_CLASS permet aussi :

- que les méthodes annotées avec @BeforeAll et @AfterAll n'ait pas l'obligation d'être statique
- d'utiliser les annotations @BeforeAll et @AfterAll sur des méthodes de classes annotées avec @Nested

118.10. Les tests imbriqués

Les tests imbriqués permettent de grouper des cas de test pour renforcer le lien qui existent entre-eux.

JUnit 5 permet de créer des tests imbriquées (nested tests) en utilisant une annotation @Nested sur une classe interne. Seules les classes internes non statiques peuvent être annotées avec @Nested.

Il permet d'utiliser des classes internes pour structurer le code de test de manière à conserver les tests en relation avec la classe de tests englobante. Il est par exemple possible de tester différents cas ou d'utiliser la même méthode de test dans la classe englobante et la classe interne.

Les tests imbriqués vont être exécutées en même temps que ceux de leur classe englobante.

Comme la classe de test imbriquée est une classe interne, elle a accès aux propriétés finales ou effectivement finales de la classe englobante.

Pour que la classe de tests imbriquées puisse avoir accès aux champs de la classe de tests englobante alors la classe imbriquée ne doit pas être statique. Si la classe imbriquée n'est pas static, alors il n'est pas possible d'utiliser de méthode statique dans la classe et donc d'avoir des méthodes annotées avec @BeforeAll et @AfterAll.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5;

import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Nested;
import org.junit.jupiter.api.Test;

public class MonTest {

  private int valeur = 0;

  @BeforeAll
  static void initAll() {
    System.out.println("BeforeAll");
  }

  @BeforeEach
  void init() {
    System.out.println("BeforeEach");
    valeur = 1;
  }

  @AfterEach
  void tearDown() {
    System.out.println("AfterEach");
    valeur = 0;
  }

  @AfterAll
  static void tearDownAll() {
    System.out.println("AfterAll");
  }

  @Test
  void simpleTest() {
    System.out.println("SimpleTest valeur=" + valeur);
    Assertions.assertEquals(1, valeur);
  }

  @Nested
  class MonTestImbrique {
    @BeforeEach
    void init() {
      System.out.println("BeforeEach imbrique");
      valeur = 2;
    }

    @Test
    void simpleTestImbrique() {
      System.out.println("SimpleTest imbrique valeur=" + valeur);
      Assertions.assertEquals(2, valeur);
    }
  }
}
```

Résultat :

```
BeforeAll
BeforeEach
SimpleTest valeur=1
AfterEach
BeforeEach
BeforeEach imbrique
SimpleTest imbrique
valeur=2
AfterEach
AfterAll
```

Une classe de test imbriquée ne peut pas par défaut avoir de méthode static annotée : les annotations @BeforeAll et @AfterAll ne peuvent donc pas être utilisées. Pour pouvoir le faire, il faut annoter la classe imbriquée avec @TestInstance(Lifecycle.PER_CLASS) pour pouvoir utiliser les annotations @BeforeAll ou @AfterAll sur des méthodes qui ne sont pas static.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5;

import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Nested;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestInstance;
import org.junit.jupiter.api.TestInstance.Lifecycle;

public class MonTest {

  private int valeur = 0;

  @BeforeAll
  static void initAll() {
    System.out.println("BeforeAll");
  }

  @BeforeEach
    void init() {
    System.out.println("BeforeEach");
    valeur = 1;
  }

  @AfterEach
    void tearDown() {
    System.out.println("AfterEach");
    valeur = 0;
  }

  @AfterAll
    static void tearDownAll() {
    System.out.println("AfterAll");
  }

  @Test
    void simpleTest() {
    System.out.println("SimpleTest valeur=" + valeur);
    Assertions.assertEquals(1, valeur);
  }

  @Nested
  @TestInstance(Lifecycle.PER_CLASS)
  class MonTestImbrique {
    @BeforeAll
    void initAllImbrique() {
      System.out.println("BeforAll imbrique");
    }

    @BeforeEach
      void init() {
      System.out.println("BeforeEach imbrique");
      valeur = 2;
    }

    @Test
    void simpleTestImbrique() {
      System.out.println("SimpleTest imbrique valeur=" + valeur);
      Assertions.assertEquals(2, valeur);
    }
  }
}
```

Résultat :

```
BeforeAll
BeforeEach
SimpleTest valeur=1
AfterEach
BeforAll imbrique
BeforeEach
BeforeEach imbrique
SimpleTest imbrique valeur=2
AfterEach
AfterAll
```

Il est possible d'inclure un test imbriqué dans un autre test imbriqué.

118.11. L'injection d'instances dans les constructeurs et les méthodes de tests

Dans les versions antérieures à la version 5 de JUnit, les constructeurs et les méthodes de test des classes de tests ne pouvaient pas avoir de paramètres pour être exécutées par le Runner standard.

JUnit 5 permet de mettre en oeuvre l'injection de dépendances via des paramètres pour les constructeurs et les méthodes des classes de tests. L'interface org.junit.jupiter.api.extension.ParameterResolver permet de définir des fonctionnalités pour résoudre dynamiquement des paramètres au runtime.

Les constructeurs et les méthodes annotées @Test, @TestFactory, @BeforeEach, @AfterEach, @BeforeAll ou @AfterAll d'une classe de tests peuvent avoir un ou plusieurs paramètres. Ces paramètres doivent être résolus à l'exécution par une instance de type ParameterResolver préalablement enregistrée.

Par défaut, JUnit 5 enregistre automatiquement 3 ParameterResolver :

- TestInfoParameterResolver : injecte une instance de type TestInfo
- RepetitionInfoParameterResolver : injecte une instance de type RepetitionInfo uniquement pour les tests répétés
- TestReporterParameterResolver : injecte une instance de type TestReporter

Un TestInfoParameterResolver permet de résoudre et d'injecter une instance de type TestInfo. Une instance de type TestInfo permet d'obtenir des informations sur le test en cours d'exécution.

L'interface TestInfo définit plusieurs méthodes :

Méthode	Rôle
String getDisplayName()	Obtenir le nom du test courant

Set<String> getTags()	Obtenir une collection des tags associés au test courant
Optional<Class<?>> getTestClass()	Obtenir la classe utilisée pour le test courant si disponible
Optional<Method> getTestMethod()	Obtenir la méthode utilisée pour le test courant si disponible

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5;

import static org.junit.Assert.assertTrue;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestInfo;

@DisplayName("Ma classe de test JUnit5")
class MonTestSimple {
  @Test
  @Tag("monTag")
  @DisplayName("mon test")
  void monTest(TestInfo testInfo) {
    System.out.println("Display name : " + testInfo.getDisplayName());
    System.out.println("Classe de test : " + testInfo.getTestClass().get().getName());
    System.out.println("Methode de test : " + testInfo.getTestMethod().get().getName());
    System.out.println("Tag : " + testInfo.getTags().toArray()[0]);
    System.out.println();
    assertTrue(true);
  }
}
```

Résultat :

```
Display name    : mon test
Classe de test  : fr.jmdoudoux.dej.junit5.MonTestSimple
Methode de test :
monTest
Tag             : monTag
```

Un RepetitionInfoParameterResolver permet de résoudre et d'injecter une instance de type RepetitionInfo dans une méthode annotée avec @RepeatedTest, @BeforeEach ou @AfterEach. Une instance de type RepetitionInfo permet d'obtenir des informations sur l'itération du test répété en cours d'exécution.

L'interface RepetitionInfo définit deux méthodes :

Méthode	Rôle
int getCurrentRepetition()	Obtenir l'itération courante d'un test répété défini par une méthode annotée avec @RepeatedTest
int getTotalRepetitions()	Obtenir le nombre d'itération d'un test répété défini par une méthode annotée avec @RepeatedTest

Un TestReporterParameterResolver permet de résoudre et d'injecter une instance de type TestReporter dans une méthode annotée avec @Test, @BeforeEach ou @AfterEach. Une instance de type TestReporter permet d'ajouter des informations sur le test courant qui pourront être exploitées par la méthode reportingEntryPublished() de la classe TestExecutionListener pour générer le rapport d'exécution des tests.

L'interface TestReport est une interface fonctionnelle qui définit deux méthodes :

Méthode	Rôle
void publishEntry(Map<String, String> values)	Publier les clés/valeurs fournies dans la Map

default void publishEntry(String key, String value)	Publier la paire clé/valeur fournie

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5;

import static org.junit.Assert.assertTrue;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestReporter;

class MonTestSimple {
  @Test
  void monTest(TestReporter testReporter) {
    testReporter.publishEntry("maCle", "maValeur");
    assertTrue(true);
  }
}
```

D'autres ParameterResolver peuvent être enregistrés avec l'annotation @ExtendWith pour pouvoir être utilisés.

118.12. Les tests répétés

JUnit Jupiter permet une exécution répétée un certain nombre de fois d'une méthode de test en l'annotant avec @RepeatedTest.

L'annotation @RepeatedTest possède deux attributs :

Attribut	Rôle
int value	Le nombre de répétitions à opérer. Obligatoire
String name	Le libellé de chaque test exécuté

Une méthode annotée avec @RepeatedTest doit respecter plusieurs contraintes :

- ne pas être private
- ne pas être static
- doit obligatoirement retourner void

Si tel n'est pas le cas, la méthode ne sera tout simplement pas exécutée par le moteur d'exécution des tests.

Chaque exécution du test se comporte comme celle d'un test standard : ainsi les méthodes annotées avec @BeforeEach et @AfterEach seront invoquée avant chaque exécution répétée.

Exemple (code Java 8) :

```
  @DisplayName("test addition repété")
  @RepeatedTest(3)
  void testRepete() {
    Assertions.assertEquals(2, 1 + 1, "Valeur obtenue erronée");
  }
```

Il est possible de personnaliser le libellé de chaque exécution du test répété en utilisant l'attribut name de l'annotation @RepeatedTest. Le libellé fourni comme valeur peut contenir trois placeholders qui seront remplacés par leurs valeurs courantes au moment de l'exécution

Placeholder	Rôle
{displayName}	Le libellé du test défini avec @DisplayName

{currentRepetition}	L'itération courante
{totalRepetitions}	Le nombre total de répétitions

Il existe aussi deux libellés prédéfinis :

- RepeatedTest.LONG_DISPLAY_NAME : correspond au format "{displayName} :: repetition {currentRepetition} of {totalRepetitions}"
- RepeatedTest.SHORT_DISPLAY_NAME. SHORT_DISPLAY_NAME : correspond au format "repetition {currentRepetition} of {totalRepetitions}". C'est le format par défaut si aucun n'est précisé

Exemple (code Java 8) :

```
@DisplayName("test addition repété")
@RepeatedTest(value = 3, name = RepeatedTest.LONG_DISPLAY_NAME)
void testRepete() {
  Assertions.assertEquals(2, 1 + 1, "Valeur obtenue erronée");
}
```

Pour obtenir des informations sur l'itération courante, il est possible d'ajouter un paramètre de type RepetitionInfo à la méthode annotée avec @RepeatedTest. Une instance de ce type sera alors injectée par le moteur d'exécution au moment de l'invocation de la méthode.

Un objet de type RepeatedInfo peut être utilisé comme paramètre dans une méthode annotée avec @RepeatedTest, @BeforeEach et @AfterEach.

L'interface RepetitionInfo définit deux méthodes :

Méthode	Rôle
int getCurrentRepetition()	Obtenir l'itération courante du test répété
int getTotalRepetitions()	Obtenir le nombre total de répétitions du test répété

Important : l'invocation de méthode annotée avec @BeforeEach et @AfterEach ayant un paramètre de type RepetitionInfo relative à des tests non répétés lèvent une exception de type ParameterResolutionException.

Exemple (code Java 8) :

```
@DisplayName("test addition repété")
@RepeatedTest(value = 3)
void testRepete(RepetitionInfo repInfo) {
  System.out.println("iteration courante : " + repInfo.getCurrentRepetition());
  System.out.println("nombre de repetition :" + repInfo.getTotalRepetitions());
  Assertions.assertEquals(2, 1 + 1, "Valeur obtenue erronée");
}
```

118.13. Les tests paramétrés

Les tests paramétrés permettent d'exécuter un même test plusieurs fois avec différentes valeurs qui lui sont passés en paramètres.

La méthode de test doit être annotée avec @org.junit.jupiter.params.ParameterizedTest. Il est aussi nécessaire de déclarer une source de données permettant d'obtenir les différentes valeurs à l'exécution des tests.

Chaque exécution du test avec les différentes valeurs est signalée de manière séparée.

Pour utiliser les tests paramétrés, il faut ajouter la dépendance junit-jupiter-params.

118.13.1. Les sources des arguments

Les paramètres des tests sont fournis grâce à une source. JUnit Jupiter propose en standard plusieurs annotations pour différents types de source dans la package org.junit.jupiter.params.provider.

Annotation	Rôle
@ValueSource	Une source de données simple sous la forme d'un tableau de chaînes de caractères ou de primitifs (int, long ou double)
@EnumSource	Une source de données simple sous la forme d'une énumération
@MethodSource	Une source de données dont les valeurs sont fournies par une méthode
@CsvSource	Une source de données dont les valeurs sont fournies sous la forme de chaînes de caractères dans laquelle chaque argument est séparé par une virgule
@CsvSourceFile	Une source de données dont les valeurs sont fournies sous la forme d'un ou plusieurs fichiers CSV
@ArgumentsSource	Une source de données qui est une méthode d'une instance de type ArgumentProvider

118.13.1.1. L'annotation @ValueSource

L'annotation @ValueSource est une source de données simple sous la forme d'un tableau de chaînes de caractères ou de primitifs (int, long ou double). Une seule valeur ne pourra être fournie pour chaque test.

L'annotation @ValueSource possède plusieurs attributs pour permettre de fournir le tableau de valeurs :

Attributs	Rôle
doubles	Fournir un tableau de valeurs de type double
longs	Fournir un tableau de valeurs de type long
ints	Fournir un tableau de valeurs de type int
strings	Fournir un tableau de valeurs de chaînes de caractères

Attention : il ne faut utiliser qu'un seul de ses attributs à la fois. Dans le cas contraire, le test échoue en levant une exception de type JUnitException.

Exemple (code Java 8) :

```
  @ParameterizedTest
  @ValueSource(ints = { 1, 2, 3 })
  void testParametreAvecValueSource(int valeur) {
    assertEquals(valeur + valeur, valeur * 2);
  }
```

118.13.1.2. L'annotation @EnumSource

L'annotation @EnumSource est une source de données simple sous la forme d'une énumération. Les valeurs définies dans l'énumération seront fournies à la méthode, une pour chaque exécution.

L'annotation @EnumSource possède plusieurs attributs :

Attributs	Rôle
Class<? extends Enum<?>> value	Préciser le type de l'énumération dont les valeurs seront utilisées pour les tests. Obligatoire
EnumSource.Mode mode	Préciser le mode de sélection des valeurs de l'énumération.

	EnumSource.Mode.INCLUDE par défaut
String[] names	Le nom des valeurs à utiliser ou des expressions régulières selon le mode utilisé. Un tableau vide par défaut. Si le tableau est vide, alors toutes les valeurs de l'énumération sont utilisées.

Exemple (code Java 8) :

```
  @ParameterizedTest
  @EnumSource(Month.class)
  void testParametreAvecEnumSource(Month mois) {
    System.out.println(mois);
    Assertions.assertNotNull(mois);
  }
```

Résultat :

```
JANUARY
FEBRUARY
MARCH
APRIL
MAY
JUNE
JULY
AUGUST
SEPTEMBER
OCTOBER
NOVEMBER
DECEMBER
```

L'attribut names permet de préciser les différents éléments de l'énumération qui devront être fournis en paramètre.

Exemple (code Java 8) :

```
  @ParameterizedTest
  @EnumSource(value = Month.class, names = { "JANUARY", "FEBRUARY", "MARCH" })
  void testParametreAvecEnumSource(Month mois) {
    System.out.println(mois);
    Assertions.assertNotNull(mois);
  }
```

Résultat :

```
JANUARY
FEBRUARY
MARCH
```

L'attribut mode permet d'avoir un contrôle sur les valeurs de l'énumération fournies en paramètre.

L'énumération EnumSource.Mode possède plusieurs valeurs :

Valeur	Rôle
EXCLUDE	Fournir tous les éléments de l'énumération sauf ceux dont le nom est précisé dans l'attribut names
INCLUDE	Ne fournir que les éléments de l'énumération dont le nom est précisé dans l'attribut names
MATCH_ALL	Ne fournir que les éléments de l'énumération dont le nom correspond aux motifs fournis dans l'attribut names
MATCH_ANY	Ne fournir que les éléments de l'énumération dont le nom correspond à un des motifs fournis dans l'attribut names

Exemple (code Java 8) :

```
  @ParameterizedTest
  @EnumSource(value = Month.class, mode = Mode.MATCH_ALL, names = { "^J.+$" })
  void testParametreAvecEnumSource(Month mois) {
    System.out.println(mois);
    Assertions.assertNotNull(mois);
  }
```

Résultat :

```
JANUARY
JUNE
JULY
```

118.13.1.3. L'annotation @MethodSource

L'annotation @MethodSource est une source de données dont les valeurs sont fournies par une méthode sous la forme d'un Stream, d'un Stream pour type primitif, d'un Iterable, d'un Iterator ou d'un tableau.

La méthode utilisée ne doit pas avoir de paramètre et doit être static sauf si la classe est annotée avec @TestInstance(Lifecycle.PER_CLASS).

Si le cas de tests n'a besoin que d'un seul paramètre, il suffit de retourner un ensemble de données de ce type.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5;

import static org.junit.jupiter.api.Assertions.assertTrue;
import java.util.stream.Stream;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.MethodSource;

public class MaClasseTest {

  @ParameterizedTest
  @MethodSource("fournirDonnees")
  void testExecuter(String element) {
    assertTrue(element.startsWith("elem"));
  }

  static Stream<String> fournirDonnees() {
    return Stream.of("elem1", "elem2");
  }
}
```

Si la méthode de test requière plusieurs arguments, la source de données retourne des instances qui sont un tableau avec les différentes valeurs.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5;

import static org.junit.jupiter.api.Assertions.assertTrue;

import java.util.Arrays;
import java.util.List;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.MethodSource;

public class MaClasseTest {

  @ParameterizedTest
  @MethodSource("fournirDonnees")
  void testTraiter(int index, String element) {
    assertTrue(index > 0);
    assertTrue(element.startsWith("elem"));
  }

  static List<Object[]> fournirDonnees() {
    return Arrays.asList(new Object[][] { { 1, "elem1" }, { 2, "elem2" } });
  }
}
```

Il est aussi possible que la source de données retourne des instances de type org.junit.jupiter.params.provider.Arguments

L'interface Arguments propose une fabrique statique of() qui attend en paramètre un varargs d'Object pour créer des instances.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5;

import static org.junit.jupiter.api.Assertions.assertTrue;
import java.util.stream.Stream;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.MethodSource;

public class MaClasseTest {

  @ParameterizedTest
  @MethodSource("fournirDonnees")
  void testTraiter(int index, String element) {
    assertTrue(index > 0);
    assertTrue(element.startsWith("elem"));
  }

  static Stream<Arguments> fournirDonnees() {
    return Stream.of(Arguments.of(1, "elem1"), Arguments.of(2, "elem2"));
  }
}
```

118.13.1.4. L'annotation @CsvSource

L'annotation @CsvSource est une source de données dont les valeurs sont fournies sous la forme de chaînes de caractères dans laquelle chaque arguments est séparés par une virgule.

La valeur par défaut de l'annotation @CsvSource est un tableau de chaînes de caractères. Chaque chaîne de caractères sera utilisée comme source de données pour une exécution du cas de test : chacune des valeurs requises doit être séparées par une virgule.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5;

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;

public class MaClasseTest {

  @DisplayName("Addition")
  @ParameterizedTest()
  @CsvSource({ "1, 1", "1, 2", "2, 3" })
  void testAdditioner(int a, int b) {
    int attendu = a + b;
    assertEquals(attendu, a + b);
  }
}
```

118.13.1.5. L'annotation @CsvFileSource

L'annotation @CsvSourceFile est une source de données dont les valeurs sont fournies sous la forme d'un ou plusieurs fichiers CSV. Chaque ligne du fichier CSV sera utilisé comme source de données pour une exécution du cas de test : chacune des valeurs requises doit être séparée par une virgule.

Exemple : le fichier additionner_source.csv

Résultat :

```
1,1
1,2
2,3
```

Le fichier CSV doit être accessible dans le classpath pour permettre son chargement.

L'attribut resources de l'annotation @ParameterizedTest permet de préciser un ou plusieurs fichiers CSV sous la forme d'un tableau de chaînes de caractères.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5;

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvFileSource;

public class MaClasseTest {

  @DisplayName("Addition")
  @ParameterizedTest()
  @CsvFileSource(resources = "additionner_source.csv")
  void testAdditionner(int a, int b) {
    int attendu = a + b;
    assertEquals(attendu, a + b);
  }
}
```

118.13.1.6. L'annotation @ArgumentsSource

L'annotation @org.junit.jupiter.params.provider.ArgumentsSource permet de préciser la classe de type org.junit.jupiter.params.provider.ArgumentProvider dont une instance sera utilisée comme source de données.

L'interface ArgumentProvider ne définit qu'une seule méthode

Méthode	Rôle
Stream<? extends Arguments> provideArguments(ExtensionContext context)	Renvoyer un Stream qui fournit les arguments passés à une méthode de test annotée avec @ParameterizedTest

Une implémentation de l'interface ArgumentsProvider doit proposer un constructeur par défaut.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5;

import static org.junit.jupiter.api.Assertions.assertTrue;

import java.util.stream.Stream;
import org.junit.jupiter.api.extension.ExtensionContext;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.ArgumentsProvider;
import org.junit.jupiter.params.provider.ArgumentsSource;

public class MaClasseTest {

  @ParameterizedTest
  @ArgumentsSource(MonArgumentsProvider.class)
  void testAvecArgumentsSource(String valeur) {
    assertTrue(valeur.startsWith("elem"));
  }

  static class MonArgumentsProvider implements ArgumentsProvider {

    @Override
    public Stream<? extends Arguments> provideArguments(ExtensionContext context) {
      return Stream.of("elem1", "elem2", "elem3").map(Arguments::of);
    }
  }
}
```

118.13.1.7. La conversion implicite des arguments

Pour permettre le support des valeurs de certaines sources de données fournies sous la forme de chaînes de caractères, JUnit Jupiter effectue des conversions au besoin vers des types selon ceux utilisés dans les paramètres de la méthode de test.

JUnit Jupiter supporte la conversion implicite pour plusieurs types :

Type cible	Exemple	
	Valeur fournie	Valeur convertie
Boolean, boolean	"true"	true
Byte, byte	"1"	1 (byte)
Character, char	"a"	'a'
Short, short	"1"	1 (short)
Integer, int	"1"	1
Long, long	"1"	1L
Float, float	"1.0"	1.0f
Double, double	"1.0"	1.0d
Enum	"APRIL"	Month.APRIL
java.time.Instant	"1970-01-01T00:00:00Z"	Instant.ofEpochMilli(0)
java.time.LocalDate	"2017-12-25"	LocalDate.of(2017, 12, 25)
java.time.LocalDateTime	"2017-12-25T13:30:59.524"	LocalDateTime.of(2017, 12, 25, 13, 30, 59, 524_000_000)
java.time.LocalTime	"13:30:59.524"	LocalTime.of(13, 30, 59, 524_000_000)
java.time.OffsetDateTime	"2017-12-25T13:30:59.524Z"	OffsetDateTime.of(2017, 12, 25, 13, 30, 59, 524_000_000, ZoneOffset.UTC)
java.time.OffsetTime	"13:30:59.524Z"	OffsetTime.of(13, 30, 59, 524_000_000, ZoneOffset.UTC)
java.time.Year	"2017"	Year.of(2017)
java.time.YearMonth	"2017-12"	YearMonth.of(2017, 12)
java.time.ZonedDateTime	"2017-12-25T13:30:59.524Z"	ZonedDateTime.of(2017, 12, 25, 13, 30, 59, 524_000_000, ZoneOffset.UTC)

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5;

import static org.junit.Assert.assertNotNull;
import java.time.Month;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

class MonTestSimple {

  @ParameterizedTest
  @ValueSource(strings = { "JANUARY", "FEBRUARY", "MARCH" })
  void testWithImplicitArgumentConversion(Month mois) {
    assertNotNull(mois.name());
  }
}
```

118.13.1.8. La conversion implicite des arguments

La conversion implicite des arguments est réalisée par une classe qui implémente l'interface ArgumentConverter.

L'interface ArgumentConverter ne définit qu'une seule méthode :

Méthode	Rôle
Object convert(Object source, ParameterContext context)	Convertir l'objet source selon le contexte fourni

La classe abstraite SimpleArgumentConverter qui implémente l'interface ArgumentConverter peut être utilisée comme classe de base pour une implémentation d'un Converter.

La classe DefaultArgumentConverter hérite de la classe SimpleArgumentConverter et est le Converter par défaut permettant de convertir une chaîne de caractères vers les wrappers des principaux types primitifs.

Pour demander l'application de la conversion sur un paramètre, il faut utiliser l'annotation @ConvertWith en lui passant en paramètre la classe de l'implémentation de type ArgumentConverter.

Le module junit-jupiter-params ne propose qu'une seule implémentation de type ArgumentConverter : JavaTimeArgumentConverter. L'utilisation de cet ArgumentConverter se fait en grâce à l'annotation @JavaTimeConversionPattern qui attend comme valeur le format de la donnée temporelle.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5;

import static org.junit.jupiter.api.Assertions.assertEquals;

import java.time.LocalDate;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.converter.JavaTimeConversionPattern;
import org.junit.jupiter.params.provider.ValueSource;

class MonTestSimple {

  @ParameterizedTest
  @ValueSource(strings = "25/12/2017")
  void testWithExplicitJavaTimeConverter(@JavaTimeConversionPattern("dd/MM/yyyy")
    LocalDate date) {
    assertEquals(2017, date.getYear());
  }
}
```

118.13.2. La personnalisation du libellé des tests

Par défaut, le nom de chaque cas de test est composé de l'index du cas suivi d'une représentation des arguments utilisés lors de son exécution.

Il est possible de personnaliser le libellé du test en utilisant l'attribut name de l'annotation @ParameterizedTest.

Plusieurs placeholders peuvent être utilisés dans la chaîne définissant l'attribut name : leur valeur correspondante sera utilisée dans le libellé affiché :

Placeholder	Rôle
{index}	L'indice du test courant, le premier ayant l'indice 1
{arguments}	La liste des valeurs de tous les arguments séparées par une virgule
{0}, {1}, ...	La valeur de l'argument dont l'index est fourni entre les accolades

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5;

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;

public class MaClasseTest {

  @DisplayName("Addition")
  @ParameterizedTest(name = "{index} : l''addition de {0} et {1}")
  @CsvSource({ "1, 1", "1, 2", "2, 3" })
  void testAdditioner(int a, int b) {
    int attendu = a + b;
    assertEquals(attendu, a + b);
  }
}
```

118.14. Les tests dynamiques

Les tests dynamiques sont une nouvelle fonctionnalité de JUnit 5 qui permet la création dynamique de tests à l'exécution. Les tests standard définis avec l'annotation @Test sont statiques : l'intégralité de leur définition doit être fournie à la compilation.

Les tests dynamiques sont des tests qui sont générés à l'exécution par une méthode de type fabrique qui est annotée avec @TestFactory. Les méthodes annotées avec @TestFactory ne sont donc pas des cas de tests mais des fabriques pour fournir un ensemble de cas de tests. Les tests dynamiques permettent par exemple d'obtenir les données requises par les cas de tests d'une source externe.

Le code du test doit être encapsulé dans une instance de type DynamicTest qui est créé dynamiquement à l'exécution.

Une méthode annotée avec @TestFactory ne peut pas être static ou private et peut retourner un objet de type :

- Stream<DynamicTest>
- Collection<DynamicTest>
- Iterable<DynamicTest>
- Iterator<DynamicTest>

Si la méthode renvoie un objet d'un autre type alors une exception de type JUnitException est levée.

Une instance de DynamicTest peut être créée en utilisant la méthode statique dynamicTest(String, Executable) de la classe DynamicTest. Cette méthode est une fabrique qui attend en paramètre le nom du test et le code à exécuter. Le test est fourni sous la forme d'une interface fonctionnelle Executable ce qui permet de fournir l'implémentation sous la forme d'une expression Lambda.

La fabrique peut renvoyer une Collection de DynamicTest

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5;

import static org.junit.jupiter.api.Assertions.assertEquals;

import java.util.ArrayList;
import java.util.Collection;
import org.junit.jupiter.api.DynamicTest;
import org.junit.jupiter.api.TestFactory;

class MonTestSimple {

  @TestFactory
  Collection<DynamicTest> dynamicTestsAvecCollection() {
    Collection<DynamicTest> resultat = new ArrayList<>();
    for (int i = 1; i <= 5; i++) {
      int val = i;
      resultat.add(DynamicTest.dynamicTest("Ajout " + val + "+" + val,
          () -> assertEquals(val * 2, val + val)));
    }
    return resultat;
  }
}
```

La fabrique peut aussi renvoyer un Iterable de DynamicTest

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5;

import static org.junit.jupiter.api.Assertions.assertEquals;

import java.util.ArrayList;
import java.util.List;
import org.junit.jupiter.api.DynamicTest;
import org.junit.jupiter.api.TestFactory;

class MonTestSimple {

  @TestFactory
  Iterable<DynamicTest> dynamicTestAvecIterable() {
    List<DynamicTest> resultat = new ArrayList<>();
    for (int i = 1; i <= 5; i++) {
      int val = i;
      resultat.add(DynamicTest.dynamicTest("Ajout " + val + "+" + val,
          () -> assertEquals(val * 2, val + val)));
    }
    return resultat;
  }
}
```

La fabrique peut aussi renvoyer un Iterator de DynamicTest

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5;

import static org.junit.jupiter.api.Assertions.assertEquals;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import org.junit.jupiter.api.DynamicTest;
import org.junit.jupiter.api.TestFactory;
```

```
class MonTestSimple {

  @TestFactory
  Iterator<DynamicTest> dynamicTestsAvecIterator() {
    List<DynamicTest> resultat = new ArrayList<>();
    for (int i = 1; i <= 5; i++) {
      int val = i;
      resultat.add(DynamicTest.dynamicTest("Ajout " + val + "+" + val,
         () -> assertEquals(val * 2, val + val)));
    }
    return resultat.iterator();
  }
}
```

La fabrique peut enfin renvoyer un Stream de DynamicTest

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5;

import static org.junit.jupiter.api.Assertions.assertEquals;

import java.util.stream.IntStream;
import java.util.stream.Stream;
import org.junit.jupiter.api.DynamicTest;
import org.junit.jupiter.api.TestFactory;

class MonTestSimple {

  @TestFactory
  Stream<DynamicTest> dynamicTestsAvecStream() {
    return IntStream.rangeClosed(1,5)
        .mapToObj(val -> DynamicTest.dynamicTest("Ajout " + val + "+" + val,
           () -> assertEquals(val * 2, val + val)));
  }
}
```

Si la fabrique renvoie un Stream, sa méthode close() sera invoquée à la fin de l'exécution des tests dynamiques qu'il contient, ce qui est important si la source du Stream requiert sa fermeture.

Le moteur d'exécution de JUnit va invoquer les méthodes annotées avec @TestFactory, ajouter le DynamicTest obtenu dans les tests et l'exécuter.

Les méthodes annotées avec @TestFactory peuvent avoir si besoin des paramètres dont les valeurs devront être injectées grâce à des ParameterResolvers.

L'exécution des DynamicTest est différente de celle des tests standards : les cas de tests dynamiques ne peuvent pas avoir de méthodes du cycle de vie invoquées. Les méthodes annotées avec @BeforeEach et @AfterEach sont invoquées pour la méthode annotée avec @TestFactory mais elles ne sont pas invoquées lors de l'exécution des cas de tests créés dynamiquement.

118.15. Les tests dans une interface

JUnit Jupiter permet d'utiliser les annotations @Test, @RepeatedTest, @ParameterizedTest, @TestFactory, @TestTemplate, @BeforeEach et @AfterEach sur des méthodes par défaut d'une interface.

Il est aussi possible d'utiliser les annotations @BeforeAll et @AfterAll sur des méthodes static ou default d'une interface qui doit être annotée dans ce cas avec @TestInstance(Lifecycle.PER_CLASS).

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5;

import static org.junit.Assert.assertTrue;
import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.DynamicTest.dynamicTest;

import java.util.Arrays;
import java.util.Collection;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DynamicTest;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestFactory;
import org.junit.jupiter.api.TestInfo;
import org.junit.jupiter.api.TestInstance;
import org.junit.jupiter.api.TestInstance.Lifecycle;

@TestInstance(Lifecycle.PER_CLASS)
interface MonInterface {

  @BeforeAll
  default void beforeAll() {
    System.out.println("Before all");
  }

  @AfterAll
  default void afterAll() {
    System.out.println("After all");
  }

  @BeforeEach
  default void beforeEach(TestInfo testInfo) {
    System.out.println("Before test " + testInfo.getDisplayName());
  }

  @AfterEach
  default void afterEach(TestInfo testInfo) {
    System.out.println("After test " + testInfo.getDisplayName());
  }

  @TestFactory
  default Collection<DynamicTest> dynamicTests() {
    return Arrays.asList(dynamicTest("true", () ->assertTrue(true)),
          dynamicTest("false", () -> assertFalse(false)));
  }

  @Test
  default void monPremierTest() {
    assertTrue(true);
  }
}
```

Il est aussi possible d'utiliser les annotations @ExtendWith et @Tag sur une interface, ce qui permettra aux classes qui l'implémente d'hériter de ces annotations.

Pour exécuter les tests, il est nécessaire d'écrire une classe qui implémente l'interface pour permettre au moteur d'exécution la création d'une instance.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5;

import static org.junit.jupiter.api.Assertions.assertTrue;
import org.junit.jupiter.api.Test;

class TestAvecInterface implements MonInterface {

  @Test
  void monSecondTest() {
    assertTrue(true);
  }
}
```

Résultat :

```
Before all
Before test dynamicTests()
After test dynamicTests()
Before test monPremierTest()
After test monPremierTest()
Before test monSecondTest()
After test monSecondTest()
After all
```

118.16. Les suites de tests

JUnit 5 permet la création de suites de tests (tests suite) qui sont une agrégation de multiples classes de tests qui pourront être exécutées ensembles.

Une suite de tests peut être composée de classes provenant de différents packages. Attention dans ce cas, les règles de visibilité doivent être respectées. Dans d'autres packages, seuls les tests de classes et méthodes de test publiques seront exécutés.

JUnit 5 propose plusieurs annotations pour définir et configurer une suite de tests :

Annotation Type	Description
@ExcludeClassNamePatterns	Préciser une ou plusieurs expressions régulières que le nom pleinement qualifié des classes à exclure dans la suite doivent respecter
@ExcludePackages	Préciser des packages dont les tests doivent être ignorés lors de l'exécution de la suite
@ExcludeTags	Préciser des tags dont les tests doivent être ignorés lors de l'exécution de la suite
@IncludeClassNamePatterns	Préciser une ou plusieurs expressions régulières que le nom pleinement qualifié des classes à inclure dans la suite doivent respecter
@IncludePackages	Préciser des packages et leur sous-packages dont les tests doivent être utilisés lors de l'exécution de la suite
@IncludeTags	Préciser des tags dont les tests doivent être utilisés lors de l'exécution de la suite
@SelectClasses	Préciser un ensemble de classes à sélectionner lors de l'exécution de la suite de tests
@SelectPackages	Préciser des packages dont les tests doivent être utilisés lors de l'exécution de la suite
@UseTechnicalNames	Demander d'utiliser le nom technique (nom pleinement qualifié de la classe) plutôt que le nom par défaut

Ces annotations peuvent être utilisées pour sélectionner les packages, les classes et les méthodes à inclure dans la suite de tests en utilisant des filtres pour inclure ou exclure ces éléments.

118.16.1. La création d'une suite de tests en précisant les packages

Il est possible d'utiliser l'annotation @SelectPackages sur une classe annotée avec @RunWith(JUnitPlatform.class) pour sélectionner les classes à inclure dans la suite.

Il est possible de ne préciser qu'un seul package comme valeur de l'attribut de l'annotation @SelectPackages : la suite sera alors composée des classes du package et de ses sous-packages. Par défaut, les classes de tests exécutées doivent avoir un nom qui se terminent par Test ou Tests.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5suite;

import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.suite.api.SelectPackages;
import org.junit.runner.RunWith;

@RunWith(JUnitPlatform.class)
@SelectPackages("fr.jmdoudoux.dej.junit5")
public class MaSuiteDeTests {
}
```

Il est aussi possible de passer plusieurs classes comme attribut de l'annotation @SelectPackages.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5suite;

import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.suite.api.SelectPackages;
import org.junit.runner.RunWith;

@RunWith(JUnitPlatform.class)
@SelectPackages({"fr.jmdoudoux.dej.junit5.modele","fr.jmdoudoux.dej.junit5.service"})
public class MaSuiteDeTests {
}
```

Les classes de tests exécutées sont celles contenues dans les packages précisés et leurs sous-packages.

118.16.2. Créer une suite de tests en précisant les classes de tests

Il est possible d'utiliser l'annotation @SelectClasses sur une classe annotée avec @RunWith(JUnitPlatform.class) pour sélectionner les classes à inclure dans la suite.

Il est possible de ne préciser qu'une seule classe comme valeur de l'attribut de l'annotation @SelectClasses : la suite sera alors uniquement composée de la classe précisée.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5;

import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.suite.api.SelectClasses;
import org.junit.runner.RunWith;

@RunWith(JUnitPlatform.class)
@SelectClasses(MonTest.class)
public class MaSuiteDeTests {
}
```

Il est aussi possible de passer plusieurs classes comme attribut de l'annotation @SelectClasses

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5;

import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.suite.api.SelectClasses;
import org.junit.runner.RunWith;

@RunWith(JUnitPlatform.class)
@SelectClasses({ MonTest.class, MonTestSimple.class, MaClasseTest.class })
```

```
public class MaSuiteDeTests {
}
```

118.16.3. Les annotations @IncludePackages et @ExcludePackages

Par défaut, l'annotation @SelectPackages recherche les classes de tests à inclure dans les packages précisés et leurs sous-packages. Les annotations @IncludePackages et @ExcludePackages permettent respectivement d'inclure uniquement ou d'exclure un ou plusieurs sous-packages particuliers.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5suite;

import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.suite.api.SelectPackages;
import org.junit.platform.suite.api.IncludePackages;
import org.junit.runner.RunWith;

@RunWith(JUnitPlatform.class)
@SelectPackages("fr.jmdoudoux.dej.junit5")
@IncludePackages("fr.jmdoudoux.dej.junit5.service")
public class MaSuiteDeTests {
}
```

Dans l'exemple ci-dessus, seules les classes du sous-packages service seront incluses dans la suite.

L'annotation @ExcludePackages s'utilise de manière similaire pour exclure des classes de tests d'un ou plusieurs packages de la suite.

118.16.4. Les annotations @IncludeClassNamePatterns et @ExcludeClassNamePatterns

Il n'est pas toujours possible de fournir explicitement le nom de toutes les classes à inclure/exclure dans la suite notamment si ce nombre est important.

Les annotations @IncludeClassNamePatterns et @ExcludeClassNamePatterns permettent respectivement d'inclure ou d'exclure des classes selon que leur nom respecte un ou plusieurs motifs.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5suite;

import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.suite.api.IncludeClassNamePatterns;
import org.junit.platform.suite.api.SelectPackages;
import org.junit.runner.RunWith;

@RunWith(JUnitPlatform.class)
@SelectPackages("fr.jmdoudoux.dej.junit5")
@IncludeClassNamePatterns({ "^.*Simple$" })
public class MaSuiteDeTests {
}
```

L'exemple ci-dessus n'inclut que les classes de tests dont le nom se termine par Simple.

L'annotation @ExcludeClassNamePattern s'utilise de manière similaire pour exclure des classes de tests de la suite si leur nom respecte le ou les motifs.

Plusieurs motifs peuvent être fournis : dans ce cas, les différents motifs sont combinés avec un opérateur OR. Si le nom pleinement qualifié de la classe respecte au moins un motif, la classe est incluse/exclue de la suite de test.

118.16.5. Les annotations @IncludeTags et @ExcludeTags

Il est possible de n'exécuter que les tests qui sont taggués ou au contraire d'exclure des tests taggués.

Cela peut par exemple permettre de définir des suites de tests qui ne seront exécutées que dans certaines circonstances ou dans un environnement d'exécution particulier.

Les annotations @IncludeTags et @ExcludeTags permettent de créer des plans de tests dans lesquels seront inclus ou exclus les tests selon leurs tags.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5suite;

import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.suite.api.IncludeTags;
import org.junit.platform.suite.api.SelectPackages;
import org.junit.runner.RunWith;

@RunWith(JUnitPlatform.class)
@SelectPackages("fr.jmdoudoux.dej.junit5")
@IncludeTags("MonTag")
public class MaSuiteDeTests {
}
```

Pour préciser plusieurs tags, il suffit d'utiliser un tableau de chaînes de caractères comme valeur.

Exemple (code Java 8) :

```
package fr.jmdoudoux.dej.junit5suite;

import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.suite.api.IncludeTags;
import org.junit.platform.suite.api.SelectPackages;
import org.junit.runner.RunWith;

@RunWith(JUnitPlatform.class)
@SelectPackages("fr.jmdoudoux.dej.junit5")
@IncludeTags({"MonTag","MonAutreTag"})
public class MaSuiteDeTests {
}
```

L'annotation @ExcludeTags s'utilise de manière similaire pour exclure des classes de tests de la suite.

Important : il n'est pas possible d'utiliser @IncludeTags et @ExcludeTags dans un même plan de test.

118.17. La compatibilité

Junit 5 ne propose pas de support pour certaines fonctionnalités de JUnit 4 telles que les Rules ou les Runner car elles proposent d'autres fonctionnalités de remplacement. Cependant JUnit 5 propose un moteur d'exécution des tests dans le module JUnit Vintage qui permet d'exécuter des tests JUnit 3 ou JUnit 4.

Le module JUnit Vintage propose une implémentation de l'interface TestEngine pour exécuter des tests JUnit 3 et 4. Ceci est d'autant plus important qu'il existe un nombre gigantesque de tests automatisés écrits en utilisant JUnit 3 et 4. Il faut ajouter la dépendance junit-vintage-engine dans le classpath pour permettre l'exécution de tests JUnit 3 et 4 par la plateforme JUnit 5.

Comme les classes et interfaces de JUnit 4 et 5 sont dans des packages différents, il est possible d'avoir les jars de deux versions dans la classpath et ainsi d'avoir des tests dans les deux versions dans un même projet. Cela peut permettre de commencer à utiliser JUnit 5 dans un projet et de migrer les tests existant au fur et à mesure.

L'équipe de JUnit prévoit de livrer des versions de maintenance et de bug fixes pour JUnit 4, ce qui devrait laisser le temps de migrer les tests vers JUnit Jupiter.

Les bibliothèques existantes comme Hamcrest ou AssertJ sont toujours compatibles avec JUnit 5.

Attention JUnit 4 avait Hamcrest en dépendance et proposait l'assertion assertThat() pour utiliser ses matcher. Avec JUnit 5, il faut utiliser directement Hamcrest après avoir ajouté la dépendance dans le classpath.

118.17.1. La migration de JUnit 4 vers JUnit 5

Plusieurs points sont à prendre en compte pour migrer des tests JUnit 4 vers JUnit 5 :

- le nom des packages de JUnit 4 est org.junit, le nom des packages de JUnit 5 est org.junit.jupiter.api
- les assertions sont dans la classe org.junit.jupiter.Assertions et les suppositions dans la classe org.junit.jupiter.Assumption
- le message des assertions est en premier paramètre en JUnit 4 et en dernier paramètre en JUnit 5
- les annotations @BeforeAll et @AfterAll remplacent les annotations @BeforeClass et @AfterClass
- les annotations @BeforeEach et @AfterEach remplacent les annotations @Before et @After
- l'annotation @Disabled remplace l'annotation @Ignore
- l'annotation @Tag remplace l'annotation @Category
- l'annotation @ExtendWith remplace les annotations @RunWith et @Rule
- l'API Hamcrest n'est pas fournie en tant que dépendance de JUnit 5 : pour continuer à utiliser les assertions d'Hamcrest comme assertThat, il faut explicitement ajouter la dépendance dans le classpath
- les tests relatifs aux exceptions levées durant les tests se font avec l'assertion assertThrows

118.18. La comparaison entre JUnit 4 et JUnit 5

JUnit 4 et 5 ont des annotations similaires mais pour la plupart avec des noms différents :

ROLE	JUNIT 4	JUNIT 5
Définir une méthode comme un cas de test	@Test	@Test
Exécuter la méthode annotée avant l'exécution de la première méthode de tests de la classe courante	@BeforeClass	@BeforeAll
Exécuter la méthode annotée après l'exécution de toutes les méthodes de tests de la classe courante	@AfterClass	@AfterAll
Exécuter la méthode annotée avant l'exécution de chaque méthode de test	@Before	@BeforeEach
Exécuter la méthode annotée après l'exécution de chaque méthode de test	@After	@AfterEach
Désactiver une classe ou une méthode de test	@Ignore	@Disabled
Définir une fabrique pour des tests dynamiques		@TestFactory
Définir un test imbriqué		@Nested
Associer un tag à la classe ou la méthode de tests	@Category	@Tag
Enregistrer une extension		@ExtendWith

Il existe de nombreuses différences entre JUnit 4 et JUnit 5.

	JUnit 4	JUNIT 5
Architecture	Un seul jar	

		Composée de trois sous-projets : JUnit Platform, JUnit Jupiter et JUnit Vintage
Version du JDK	Java 5 ou supérieur	Java 8 ou supérieur
Les assertions	La classe org.junit.Assert contient les assertions. Les surcharges qui acceptent en paramètre un message l'attendent en premier paramètre Hamcrest est fournie en dépendance de JUnit utilisable dans l'assertion assertThat()	La classe org.junit.jupiter.Assertions contient les assertions. Elle contient notamment la méthode assertAll() pour permettre de réaliser des assertions groupées. Les surcharges qui acceptent en paramètre un message l'attendent en dernier paramètre Hamcrest n'est plus fournie en dépendance

Les assumptions	La classe org.junit.Assume contient les suppositions.
	
	Elle propose plusieurs méthodes :
	assumeFalse()
	assumeNoException()
	assumeNotNull()
	assumeThat()
	assumeTrue()
	assumeFalse()
		La classe org.junit.jupiter.Assumptions contient les suppositions.
		
		Elle propose uniquement trois méthodes qui possèdent de nombreuses surcharges :
		assumeFalse()
		assumingThat()
		assumeTrue()

Les tags	@category	@tag
Les suites de tests	@RunWith et @Suite	@RunWith, @SelectPackages et @SelectClasses

119. Les objets de type mock

Chapitre 119

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Les doublures d'objets ou les objets de type mock permettent de simuler le comportement d'autres objets. Ils peuvent trouver de nombreuses utilités notamment dans les tests unitaires où ils permettent de tester le code en maitrisant le comportement des dépendances.

Ce chapitre contient plusieurs sections :

- ♦ Les doublures d'objets et les objets de type mock
- ♦ L'utilité des objets de type mock
- ♦ Les tests unitaires et les dépendances
- ♦ L'obligation d'avoir une bonne organisation du code
- ♦ Les frameworks
- ♦ Les inconvénients des objets de type mock

119.1. Les doublures d'objets et les objets de type mock

En POO, il existe plusieurs types d'objets, généralement appelés doublures, permettant de simuler le comportement d'un autre objet :

- dummy (fantôme, bouffon) : objets "vides" qui n'ont pas de fonctionnalités implémentées.
- stub (bouchon) : classes qui renvoient une valeur codée en dur à l'invocation d'une méthode
- fake (substitut, simulateur) : classes qui sont une implémentation partielle et qui, par exemple, renvoient toujours les mêmes réponses quels que soient les paramètres fournis
- spy (espion) : classe qui vérifie l'utilisation qui en est faite après l'exécution
- mock (simulacre) : classes qui agissent comme un stub et un spy

Le vocabulaire lié à ces types d'objets est assez confus dans la langue anglaise donc il l'est d'autant plus dans la langue française où l'on tente de le traduire. Ce chapitre va se concentrer essentiellement sur les objets de type mock.

Un objet de type doublure permet donc de simuler le comportement d'un autre objet concret de façon maitrisée.

L'emploi de doublures est largement utilisé pour les tests unitaires mais il peut aussi être mis en oeuvre lors des développements pour par exemple remplacer un objet qui n'est pas encore écrit.

L'utilisation des doublures permet aux tests unitaires de se concentrer sur les tests du code de la méthode qui correspond au System Under Test (SUT) sans avoir à se préoccuper des dépendances.

Les doublures ont pour rôle de simuler le comportement d'un objet permettant ainsi de réaliser les tests de l'objet de façon isolée et répétable.

Un objet de type mock permet de simuler le comportement d'un autre objet concret de façon maitrisée et de vérifier les invocations qui sont faites de cet objet.

Cette double fonctionnalité permet dans un test unitaire de faire des tests sur l'état (state test) et des tests sur le comportement (behavior test).

119.1.1. Les types d'objets mock

Il existe deux grands types d'objets mock :

- statique : ce sont des classes Java écrites ou générées avec un outil par le développeur
- dynamique : ils sont mis en oeuvre par un framework

Les objets mock peuvent être codés manuellement ou utiliser un framework qui va permettre de les générer dynamiquement. L'avantage des mocks dynamiques c'est qu'aucune classe implicite n'a besoin d'être écrite.

Les frameworks de mocking peuvent utiliser plusieurs solutions pour mettre en oeuvre des mocks dynamiques :

- proxy : un proxy est un objet qui est utilisé à la place d'un autre objet. Il est alors nécessaire de fournir ce proxy à l'objet qui l'utilise en utilisant un constructeur ou un setter. Ceci nécessite donc qu'un mécanisme d'injection de dépendance soit mis en oeuvre dans la classe à tester (EasyMock, ...)
- instrumentation : un classloader spécifique est utilisé pour dynamiquement charger une classe à la place d'une autre notamment en utilisant la classe java.lang.Instrument de Java 1.5 (jmockit)
- AOP : permet d'invoquer la méthode d'un mock à la place de celle d'une implémentation concrète sans avoir à mettre en oeuvre une interface ni à requérir un mécanisme d'injection de dépendances. L'invocation de la méthode est interceptée et remplacée par l'invocation de la méthode du mock. Ceci ne doit cependant pas être une excuse pour ne pas écrire du code qui mette en oeuvre une bonne conception car en plus d'être testable cela rend le code plus compréhensible, plus maintenable et plus évolutif. (jeasytest, amock, ...)

Avec l'utilisation de proxies, il est indispensable d'avoir un mécanisme d'injection de dépendances permettant de fournir l'implémentation à utiliser. Ceci permet dans le cas des tests unitaires de fournir un objet de type mock qui sera utilisé lors de l'exécution des tests à la place d'une vraie instance de classe dépendante.

Ce mécanisme d'injection de dépendances peut être fourni par un framework (exemple : Spring) ou implémenté manuellement mais dans tous les cas le code à tester doit fournir une solution pour le réaliser.

Il existe plusieurs frameworks de mocking en Java qui permettent de créer dynamiquement des objets de type mock.

119.1.2. Exemple d'utilisation dans les tests unitaires

Dans une application, les classes ont généralement des dépendances entre elles. Ceci est particulièrement vrai dans les applications développées en couches (présentation, service, métier, accès aux données (DAO), ...).

L'idée lors de l'exécution d'un test unitaire est de tester la plus petite unité de code possible, soit la méthode et uniquement le code de la méthode. Cependant les classes utilisées dans le code de cette méthode font généralement appel à un ou plusieurs autres objets. Le but n'est pas de tester ces objets qui feront eux-mêmes l'objet de tests unitaires mais de tester le code de la méthode : le test unitaire doit concerner uniquement la méthode et ne pas tester les dépendances.

Il faut donc une solution pour s'assurer que les objets dépendants fournissent les réponses désirées à leur invocation. Cette solution repose sur les objets de type simulacre.

Cela suppose que si le code de la méthode fonctionne comme voulu (validé par des tests unitaires) et que les dépendances fonctionnent de même (validées par leurs tests unitaires) alors ils fonctionneront normalement ensembles.

Les classes dépendantes ne doivent pas être testées dans les tests unitaires de la classe. Elles doivent être considérées comme testées, sachant que des tests unitaires qui leur sont dédiés doivent exister. Certaines classes doivent aussi être considérées comme testées : c'est notamment le cas des classes du JRE.

Il est très important que les tests unitaires ne concernent que le code de la méthode en cours de test. Autrement, il est difficile de trouver un bug qui peut être dans un objet dépendant de niveau -N.

Il est alors nécessaire de simuler le fonctionnement des classes dépendantes.

Le but d'un objet Mock est de remplacer un autre objet en proposant de forcer les valeurs de retour de ses méthodes selon certains paramètres.

Ainsi l'invocation d'un objet de type mock garantit d'avoir les valeurs attendues selon les paramètres fournis.

119.1.3. La mise en oeuvre des objets de type mock

Un des avantages à utiliser des objets mock, notamment dans les tests unitaires, est qu'ils forcent le code à être écrit ou adapté par des refactoring pour qu'il respecte une conception permettant de le rendre testable.

Généralement, un objet de type mock est une implémentation d'une interface qui se limite le plus souvent à renvoyer des valeurs déterminées en fonction des paramètres reçus. L'interface est parfaitement adaptée puisque l'objet simulé et l'objet mock doivent avoir le même contrat.

Un objet de type mock possède donc la même interface que l'objet qu'il doit simuler, ce qui permet d'utiliser le mock ou une implémentation concrète de façon transparente pour l'objet qui l'invoque.

Les objets mock simulent le comportement d'autres objets mais ils sont aussi capables de vérifier les invocations qui sont faites sur le mock : nombres d'invocations, paramètres fournis, ordre d'invocations, ...

La mise en oeuvre d'un objet de type mock dans les tests unitaires suit généralement plusieurs étapes :

- définir le comportement du mock : méthodes invoquées, paramètres fournis, valeurs de retour ou exception ...
- exécuter le test en invoquant la méthode à tester
- vérifier des résultats du test
- vérifier les invocations du ou des objets de type mock : nombre d'invocations, ordre d'invocations, ...

119.2. L'utilité des objets de type mock

Les objets de type mock peuvent être utilisés dans différentes circonstances :

- renvoyer des résultats déterminés notamment dans des tests unitaires automatisés
- obtenir un état difficilement reproductible (erreur d'accès réseau, ...)
- éviter d'invoquer des ressources longues à répondre (accès à une base de données, ...)
- invoquer un composant qui n'existe encore pas
- ...

Les objets de type mock sont donc très intéressants pour simuler le comportement de composants invoqués de façon distante (exemple : EJB, services web, RMI, ...) et particulièrement pour tester les cas d'erreurs (problème de communication, défaillance du composant ou du serveur qui gère leur cycle de vie, ...).

119.2.1. L'utilisation dans les tests unitaires

Les tests unitaires automatisés sont une composante très importante du processus de développement et de maintenance d'une application, malgré le fait qu'ils soient fréquemment négligés.

Pour permettre de facilement détecter et corriger d'éventuels bugs dans le code testé, il est nécessaire d'isoler ce code en simulant le comportement de ses dépendances.

L'utilisation des objets mock est une technique particulièrement puissante pour permettre des tests unitaires sur des classes.

Les objets de type mock permettent réellement des tests qui soient unitaires puisque leur résultat est prévisible. Si le test échoue, il y a une forte probabilité que l'origine du problème soit dans la méthode en cours de test. Ceci facilite la résolution du problème puisque celui-ci est isolé à l'intérieur de cette méthode.

Les objets de type mock permettent de s'assurer que l'échec d'un test n'est pas lié à une de ses dépendances sauf si les données retournées par le ou les objets mock sont erronées vis-à-vis du cas de test en échec.

119.2.2. L'utilisation dans les tests d'intégration

Les objets mock sont particulièrement utiles dans les tests unitaires mais ils sont à éviter dans les tests d'intégration. Le but des tests d'intégration étant de tester les interactions entre les modules et les composants, il n'est pas forcément souhaitable de simuler le comportement de certains d'entre-eux.

Pour les tests d'intégration, les objets mock peuvent cependant être utiles dans certaines circonstances :

- pour simuler un module dont le temps de traitement est particulièrement long
- pour simuler un module qui est complexe à initialiser
- pour simuler des cas d'erreur

119.2.3. La simulation de l'appel à des ressources

Les tests unitaires doivent toujours s'exécuter le plus rapidement possible notamment si ceux-ci sont intégrés dans un processus de build automatique. Un test unitaire ne doit donc pas utiliser de ressources externes comme une base de données, des fichiers, des services, ... Les tests avec ces ressources doivent être faits dans les tests d'intégration puisque ce sont des dépendances.

119.2.4. La simulation du comportement de composants ayant des résultats variables

Les tests utilisant une fonctionnalité dont le résultat est aléatoire ou fluctuant selon les appels avec le même contexte ne sont pas répétables.

Exemple : une méthode qui convertit le montant d'une monnaie dans une autre. La méthode utilise un service web pour obtenir le cours de la monnaie cible. A chaque exécution du cas de test, le résultat peut varier puisque le cours d'une monnaie fluctue.

Pour permettre d'exécuter correctement les tests d'une méthode qui utilise une telle fonctionnalité, il faut simuler le comportement du service dépendant pour qu'il retourne des valeurs prédéfinies selon le contexte fourni en paramètre. Ainsi pour chaque cas de tests, le service retournera la même valeur rendant ainsi les résultats prédictibles et donc les tests répétables.

Bien sûr ce type de tests pose comme pré-requis que le service fonctionne correctement mais cela est du ressort des développeurs du service qui doivent eux aussi garantir le bon fonctionnement de leur service en utilisant des tests unitaires.

119.2.5. La simulation des cas d'erreurs

Les objets de type mock peuvent aussi permettre de facilement tester des cas d'erreurs. Certaines erreurs sont difficiles à reproduire donc à tester, par exemple un problème de communication avec le réseau, d'accès à une ressource, de connexion à un serveur (Base de données, Broker de messages, système de fichiers partagés,...).

Il est possible d'effectuer des opérations manuelles pour réaliser ces tests (débrancher le câble réseau, arrêter un serveur, ...) mais ces opérations sont fastidieuses et peu automatisables.

Il est par contre très facile pour un objet mock de retourner une exception qui va permettre de simuler et de tester le cas d'erreur correspondant.

119.3. Les tests unitaires et les dépendances

Le principe de limiter les responsabilités d'un objet pour faciliter la réutilisation implique qu'un objet a souvent besoin d'autres objets pour réaliser ses tâches. Ces objets dépendants ont eux aussi des dépendances vers d'autres objets. Ces dépendances forment rapidement un graphe d'objets complexe qui pose rapidement des problèmes pour les tests et surtout qui empêche l'isolation du test de l'objet à tester. Cela devient particulièrement vrai si une ou plusieurs de ces dépendances utilisent des ressources distantes, longues à répondre ou dont les résultats ne sont pas constants.

Le test unitaire d'un objet peut utiliser des objets de type mock pour simuler le comportement de leurs dépendances immédiates.

Seules les dépendances de premier niveau ont besoin d'être remplacées par des objets mock pour tester l'objet. Pour tester l'objet, il est inutile de créer des mocks pour les dépendances de niveaux 2 et supérieurs.

119.4. L'obligation d'avoir une bonne organisation du code

Un code mal conçu pour être testable est un frein à la rédaction des tests unitaires automatisés car ceux-ci seront trop compliqués voire impossibles à écrire.

Les tests unitaires, et plus encore, l'utilisation d'objets de type mock encourage voire impose une conception adaptée du code qui, au final, le rend non seulement testable mais aussi plus compréhensible et plus maintenable.

L'écriture de tests unitaires impose que le code écrit soit testable, ce qui implique notamment que :

- le code d'une méthode ne doit pas être trop important (amélioration du découpage des fonctionnalités)
- le code ne doit pas être complexe
- le code ne doit pas avoir trop de dépendances
- les dépendances doivent pouvoir être injectées
- ...

119.4.1. Quelques recommandations

Si une classe dépendante est simplement instanciée dans le code de la méthode à tester, il ne sera pas possible de la simuler en la remplaçant par un autre objet. Pour faciliter les tests unitaires qui utilisent des objets mocks, il faut mettre en oeuvre un mécanisme d'injection de dépendances et définir une interface pour chaque objet dépendant.

Les dépendances doivent être décrites par une interface pour permettre facilement de créer des objets de type mocks. Il est possible de créer une instance d'une interface sans avoir à fournir d'implémentations des méthodes. Les objets mocks vont utiliser cette fonctionnalité pour fournir une implémentation qui va simuler le comportement du véritable objet.

L'utilisation de certaines fonctionnalités ou motifs de conception peut entraver, voire rendre compliqué et même impossible le test du code avec des tests unitaires, par exemple :

- il faut éviter le motif de conception singleton
- il ne faut pas utiliser l'initialisation statique des propriétés
- ...

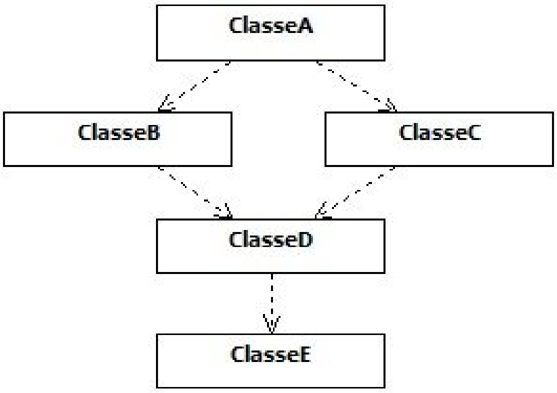
Si l'accès à une ressource est codé de façon statique, cette ressource devra être disponible lors de l'exécution des tests quand elle sera sollicitée.

Il est généralement préférable d'effectuer un refactoring pour rendre le code testable plutôt que d'écrire des tests unitaires compliqués ou de ne pas les écrire du tout. Au final, le code est amélioré.

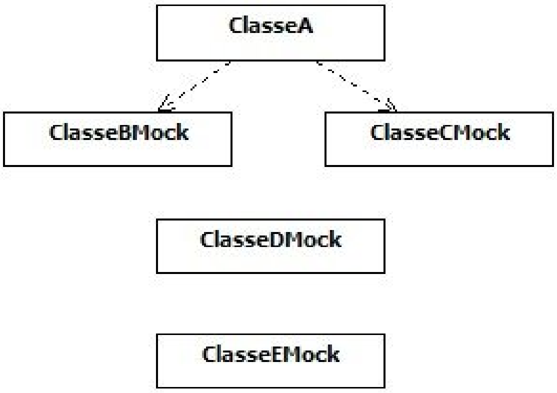
119.4.2. Les dépendances et les tests unitaires

Les objets de type mock permettent de réaliser des tests unitaires dans le contexte d'un système reposant sur des développements orientés objets. Dans un tel contexte, il existe des dépendances plus ou moins nombreuses entre les objets.

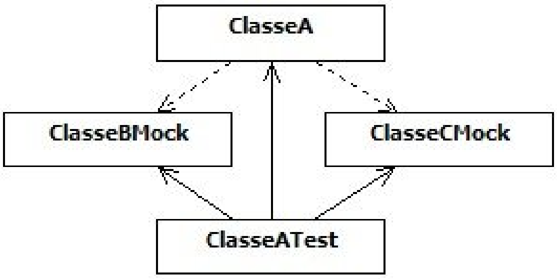
Ainsi la méthode d'une classe, qui est l'unité à tester, dépends généralement d'un ou plusieurs objets dépendant eux aussi d'autres objets. L'invocation de la méthode lors de son test va inévitablement faire appel à ces dépendances : ceci n'est alors plus un test unitaire mais un test d'intégration. De plus, cela complexifie généralement la détermination de l'origine d'un problème et induit des difficultés de répétatibilité de l'exécution des tests.



Les objets de type mock permettent de maitriser le fonctionnement des dépendances en simulant de façon prédéterminée leur comportement lors de l'exécution des cas de tests.

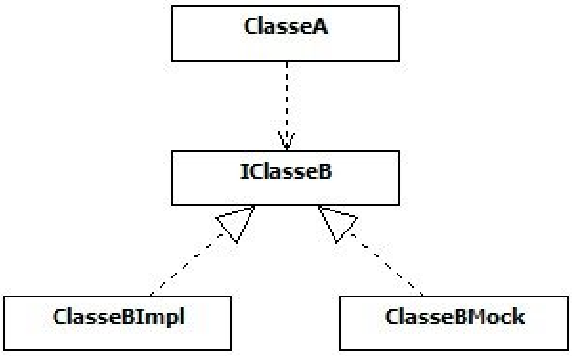


Les objets de type mock sont donc conçus pour répondre au besoin des tests unitaires.



L'entité testée ne doit pas savoir si l'objet utilisé durant les tests est un mock. Pour cela, les dépendances d'une entité testée doivent être décrites avec une interface et l'entité doit utiliser un mécanisme d'injection pour permettre d'utiliser une implémentation de la dépendance en production et d'utiliser un mock pour les tests.

Le fait de décrire les fonctionnalités d'une dépendance avec une interface facilite la mise oeuvre d'un objet de type mock.



L'injection de dépendances doit permettre de substituer l'implémentation de la dépendance à un objet de type mock. Si le code à tester instancie en dur la dépendance en utilisant l'opérateur new, il est extrêmement difficile de réaliser la substitution sauf en mettant en oeuvre des fonctionnalités avancées et complexes (avec un classloader par exemple).

L'injection peut se faire de différentes façons, par exemple :

- elle peut se faire directement sur le type de l'interface
- sur une fabrique qui se charge de créer l'instance voire un mélange de ces deux solutions
- définir une méthode protected qui renvoie une instance de l'interface (par défaut celle de l'implémentation). Pour les tests, il suffit alors d'hériter de la classe à tester et de redéfinir ce getter pour renvoyer une instance du mock
- utiliser un framework qui propose une solution d'injection de dépendance (Exemple : Spring)
- ...

119.4.3. Exemple de mise en oeuvre de l'injection de dépendances

Cette section va faire évoluer une classe dont une méthode à tester utilise une dépendance directement instanciée.

L'important pour pouvoir utiliser les mocks c'est que la classe ait été conçue et développée pour être testable. Ceci implique notamment de proposer un mécanisme permettant de pouvoir remplacer une implémentation d'une dépendance par un objet de type mock.

Exemple :

```
public class ClasseA {
  public String maMethode(){
     // début des traitements
    ClasseB classeB = new ClasseB();
    // suite des traitements utilisant classeB
  }
}
```

Dans l'exemple ci-dessus, le test de la méthode maMethode() va être difficile car l'instanciation de la dépendance se fait en dur dans le code. Il ne va pas être facile de remplacer cette instance par celle d'un objet mock. Une solution consiste à proposer une méthode qui se charge de retourner une instance de la classe ClasseB. Il est important que cette méthode puisse être redéfinie dans une classe fille.

Exemple :

```
public class ClasseA {
  public String maMethode(){
    // début des traitements
    ClasseB classeB = creerClasseB();
    // suite des traitements utilisant classeB
  }
  protected ClasseB creerClasseB() {
    return new ClasseB();
  }
}
```

Pour faciliter la création d'un objet mock, il est préférable que chaque objet dépendant implémente une interface qui décrive ses fonctionnalités.

Exemple :

```
public class ClasseB implement InterfaceB {

...
}
```

L'objet est alors défini comme une implémentation de son interface, il est ainsi plus facile de créer un objet mock car cela évite de dériver la classe.

Exemple :

```
public class ClasseA {
  public String maMethode(){
    // début des traitements
    InterfaceB classeB = creerClasseB();
    // suite des traitements utilisant classeB
  }

  protected InterfaceB creerClasseB() {
    return new ClasseB();
  }
}
```

Lors de l'écriture du test, il faut dériver la classe à tester et réécrire la méthode qui instancie la dépendance pour qu'elle renvoie une instance de l'objet mock.

Exemple :

```
public class TestClasseA extends TestCase {

  public void testMaMethode() {
    ClasseA classeA = new ClasseA(){
      // reecriture de la méthode pour qu'elle renvoie un mock
      protected InterfaceB creerClasseB() {
        // renvoie une instance du mock de la classe B
        return new MockB();
      }
    }

    String resultat = classeA.maMethode();
    // évaluation des résultats du cas de test
  }
}
```

Cet exemple permet de facilement mettre en oeuvre le principe d'injections de dépendances dans du code qui n'a rien pour mettre en oeuvre ce type de fonctionnalité. Certains frameworks, comme Spring, offrent également cette possibilité.

119.4.4. Limiter l'usage des singletons

Les singletons ne permettent pas de remplacer facilement leur unique instance par un objet de type mock.

119.4.5. Encapsuler les ressources externes

Il faut encapsuler les dépendances vers des ressources externes dans des entités dédiées pour permettre de les injecter et ainsi utiliser une implémentation à l'exécution et un objet mock lors des tests.

119.5. Les frameworks

L'écriture d'objets de type mock à la main peut être longue et fastidieuse, de plus, des objets peuvent contenir des bugs comme toute portion de code. Des frameworks ont donc été développés pour créer ces objets dynamiquement et de façon fiable.

La plupart des frameworks de mocking permettent de spécifier le comportement que doit avoir l'objet mock :

- les méthodes invoquées : paramètres d'appels et valeur de retour
- l'ordre d'invocations de ces méthodes
- le nombre d'invocations de ces méthodes

Les frameworks de mocking permettent de créer dynamiquement des objets mocks, généralement à partir d'interfaces. Ils proposent fréquemment des fonctionnalités qui vont bien au-delà de la simple simulation d'une valeur de retour :

- simulation de cas d'erreurs en levant des exceptions
- validation des appels de méthodes
- validation de l'ordre de ces appels
- ...

Plusieurs frameworks relatifs aux objets de type mock existent dans le monde Java, notamment :

- EasyMock
- JMockIt
- Mockito
- JMock
- MockRunner

Les différents frameworks vont être utilisés pour mocker les dépendances dans les tests unitaires de la classe ci-dessous qui n'a qu'un rôle purement éducatif.

Exemple :

```
package fr.jmdoudoux.dej;

public class MonService {

  protected Calculatrice creerCalculatrice() {
    return new CalculatriceImpl();
  }

  /**
   * Calculer la somme de deux entiers positifs
   *
   * @param val1
   *          la premiere valeur
   * @param val2
   *          la seconde valeur
   * @return la somme des deux arguments ou -1 si un des deux arguments est
   *         negatif
   */
  public long additionner(int val1, int val2) {
    long retour = 0l;
    Calculatrice calculatrice = creerCalculatrice();

    try {
      retour = calculatrice.additionner(val1, val2);
    } catch (IllegalArgumentException iae) {
      retour = -1l;
    }

    return retour;
  }

  /**
   * Calculer la somme des deux premiers parametres et soustraire la valeur du troisième
   * @param val1
   * @param val2
   * @param val3
   * @return le resultat du calcul
   */
  public long calculer(int val1, int val2, int val3) {
    long retour = 0l;
    Calculatrice calculatrice = creerCalculatrice();

    try {
      long somme = calculatrice.additionner(val1, val2);
      retour = calculatrice.soustraire(somme, val3);
    } catch (IllegalArgumentException iae) {
      retour = -1l;
    }

    return retour;
  }
}
```

Cette classe utilise une dépendance dont les fonctionnalités sont décrites dans une interface.

Exemple :

```
package fr.jmdoudoux.dej;

public interface Calculatrice {

  public long additionner(int val1, int val2);

  public long soustraire(long val1, int val2);

}
```

Exemple :

```
package fr.jmdoudoux.dej;

public class CalculatriceImpl implements Calculatrice {

  @Override
  public long additionner(int val1, int val2) {
    if (val1 <  0 || val2 < 0) {
      throw new IllegalArgumentException("La valeur ne peut pas etre negative");
    }
    return val1+val2;
  }

  @Override
  public long soustraire(long val1, int val2) {
    return val1-val2;
  }
}
```

119.5.1. EasyMock

Easy Mock est un framework de mocking open source qui permet de créer et d'utiliser des objets de type mock.

EasyMock est un framework simple (composé uniquement d'une douzaine de classes et interfaces) et très puissant qui permet de créer et d' utiliser des objets de type mock.

EasyMock travaille à partir d'interfaces pour créer des objets de type mock mais il propose une extension qui permet de créer des objets de type mock pour des classes.

Les mocks créés par EasyMock peuvent avoir plusieurs utilités allant du simple dummy qui renvoie une valeur au spy qui permet de vérifier le comportement des invocations de méthodes selon les paramètres fournis.

La version utilisée dans cette section est la 2.4. Elle requiert un Java 5 minimum.

Pour l'utiliser, il faut télécharger l'archive sur le site https://easymock.org/ et la décompresser dans un répertoire du système. Il suffit alors d'ajouter le fichier easymock.jar dans le classpath.

La classe principale d'EasyMock est la classe EasyMock : toutes ses méthodes sont statiques. Il est donc possible de faire un import static de cette classe pour pouvoir invoquer ses méthodes sans avoir à les préfixer par le nom de la classe.

Voici un exemple de tests unitaires de la classe MonService qui utilise EasyMock pour simuler le comportement des dépendances.

Exemple :

```
package fr.jmdoudoux.dej;

import org.easymock.EasyMock;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

public class MonServiceTest {

  private Calculatrice mock = null;

  private MonService monService = null;

  @Before
  public void setUp() throws Exception {
    mock = EasyMock.createMock(Calculatrice.class);

    monService = new MonService() {
      @Override
      protected Calculatrice creerCalculatrice() {
        return mock;
      }
    };
  }

  @Test
  public void testAdditionner() {
    long retour = 0l;

    EasyMock.expect(mock.additionner(1, 2)).andReturn(Long.valueOf(3l));
    EasyMock.replay(mock);

    retour = monService.additionner(1, 2);
    Assert.assertEquals("La valeur retournee est invalide", 3l, retour);
  }

  @Test
  public void testAdditionnerParametreInvalide() {
    long retour = 0l;

    EasyMock.expect(mock.additionner(-1, 2)).andThrow(new IllegalArgumentException());
    EasyMock.replay(mock);

    retour = monService.additionner(-1, 2);
    Assert.assertEquals("La valeur retournee est invalide", -1l, retour);
  }

  @Test
  public void testCalculer() {
    long retour = 0l;

    EasyMock.expect(mock.additionner(20, 30)).andReturn(Long.valueOf(50l));
    EasyMock.expect(mock.soustraire(50, 10)).andReturn(Long.valueOf(40l));
    EasyMock.replay(mock);

    retour = monService.calculer(20, 30, 10);
    Assert.assertEquals("La valeur retournee est invalide", 40l, retour);
  }
}
```

119.5.1.1. La création d'objets mock

Par défaut, EasyMock ne permet que de créer des objets de type mock pour des interfaces. EasyMock propose une extension pour créer des objets mock à partir d'une classe. Le code à utiliser est similaire hormis qu'il faut importer le package org.easymock.classextension.EasyMock à la place du package org.easymock.EasyMock.

La classe org.easymock.EasyMock permet de créer et utiliser des objets de type mock à partir d'une interface qui précise les fonctionnalités de l'objet à simuler.

La classe org.easymock.classestension.EasyMock permet de créer et utiliser des objets de type mock à partir d'une classe.

La création d'une instance d'un objet de type mock se fait en invoquant la méthode statique createMock() de la classe EasyMock qui attend en paramètre la classe de l'interface.

Exemple :

```
    mock = EasyMock.createMock(Calculatrice.class);
```

EasyMock propose trois types de mocks :

- mock : mock classique qui vérifie l'invocation des méthodes et s'obtient en invoquant la méthode EasyMock.createMock()
- strict mock : mock qui vérifie l'invocation des méthodes ainsi que l'ordre de ces invocations. Il est créé en utilisant la méthode EasyMock.createStrictMock()
- nice mock : mock qui renvoie une valeur par défaut lors de l'invocation d'une méthode du mock dont le comportement n'a pas été précisé. Sa création se fait en utilisant la méthode EasyMock.createNiceMock()

119.5.1.2. La définition du comportement des objets mock

La définition du comportement des mocks se fait sous la forme enregistrer/rejouer.

Pour préciser le comportement d'une méthode d'un mock qui renvoie une valeur, il faut :

- utiliser la méthode statique EasyMock.expect() pour l'invoquer en lui précisant ses paramètres
- utiliser la méthode andReturn() de l'objet de type IExpectationSetters retourné par l'appel précédent pour préciser la valeur de retour

La méthode expect() permet de préciser le comportement attendu en retour de l'invocation d'une méthode.

L'avantage de cette approche qui nécessite l'invocation de la méthode dont le comportement est à simuler est qu'elle permet d'utiliser le code completion et le refactoring de l'IDE utilisé.

Exemple :

```
    EasyMock.expect(mock.additionner(1, 2)).andReturn(Long.valueOf(3l));
```

EasyMock propose de simuler la levée d'une exception dans la définition du comportement d'une méthode en utilisant la méthode andThrow() qui attend en paramètre l'instance de l'exception à lever.

Exemple :

```
    EasyMock.expect(mock.additionner(-1, 2)).andThrow(new IllegalArgumentException());
```

Ceci est particulièrement utile pour tester des cas d'erreurs difficiles à automatiser comme une coupure réseau par exemple. Tous les types d'exceptions peuvent être simulés (checked, runtime ou error).

Pour préciser le comportement d'une méthode qui ne retourne aucune valeur (void), il faut simplement invoquer la méthode sur l'instance du mock sans utiliser la méthode expect(). EasyMock va simplement enregistrer le comportement.

Ainsi, si le résultat de l'invocation de la méthode ne renvoie aucune valeur ni ne lève aucune exception, il ne faut pas utiliser la méthode expect() mais simplement invoquer la méthode sur l'objet de type mock.

Certaines méthodes permettent de préciser le nombre d'invocations d'une méthode :

- times(n,m) : la méthode devra être invoquée entre n et m fois
- atLeastOnce() : la méthode devra être invoquée au moins une fois
- anyTimes() : la méthode peut être invoquée un nombre indéfini de fois

EasyMock permet aussi facilement de définir des comportements différents lors de plusieurs invocations de la même méthode car les appels aux méthodes andReturn(), andThrow() et times() peuvent être chaînés.

Exemple :

```
    EasyMock.expect(mock.additionner(-1, 2))
      .andReturn(Long.valueOf(3l))
      .andThrow(new IllegalArgumentException());
```

Le comportement attendu est ainsi enregistré par l'objet mock jusqu'à l'invocation de la méthode replay().

119.5.1.3. L'initialisation des objets mock

Une fois définis tous les comportements attendus pour le ou les mocks, il faut invoquer la méthode replay() sur l'objet de type Control.

Exemple :

```
    EasyMock.replay(mock);
```

Si la méthode statique replay() de la classe EasyMock n'est pas invoquée et que des comportements de l'objet mock sont définis alors une exception de type IllegalStateException est levée lors de l'utilisation de ce mock.

Exemple :

```
    java.lang.IllegalStateException: missing behavior definition
      for the preceeding method call additionner(20, 30)
        at org.easymock.internal.MockInvocationHandler.invoke(MockInvocationHandler.java:30)
        at org.easymock.internal.ObjectMethodsFilter.invoke(ObjectMethodsFilter.java:61)
        at $Proxy5.soustraire(Unknown Source)
        at fr.jmdoudoux.dej.MonService.calculer(MonService.java:45)
        at fr.jmdoudoux.dej.MonServiceTest.testCalculer(MonServiceTest.java:56)
...
```

Le nom de la méthode replay() peut être source de confusions : en fait, elle ne rejoue pas le comportement de l'objet mock mais elle réinitialise son état interne pour lui permettre d'avoir le comportement attendu lors des futures invocations des méthodes.

L'appel à la méthode replay() permet de placer le mock en situation de reproduire le comportement défini à l'invocation d'une méthode du mock et d'enregistrer ces invocations.

119.5.1.4. La vérification des invocations des objets mock

EasyMock n'est pas utile que pour fournir des réponses déterminées à des invocations données : il permet aussi de vérifier les paramètres fournis et l'ordre d'invocations des méthodes.

Une exception est levée si les paramètres utilisés lors de l'invocation ne correspondent pas à ceux définis dans l'objet de type mock.

Résultat :

```
java.lang.AssertionError:
  Unexpected method call additionner(2, 2):
    additionner(1, 2): expected: 1, actual: 0
        at org.easymock.internal.MockInvocationHandler.invoke(MockInvocationHandler.java:32)
        at org.easymock.internal.ObjectMethodsFilter.invoke(ObjectMethodsFilter.java:61)
        at $Proxy5.additionner(Unknown Source)
        at fr.jmdoudoux.dej.MonService.additionner(MonService.java:24)
        at fr.jmdoudoux.dej.MonServiceTest.testAdditionner(MonServiceTest.java:33)
```

Cette exception est levée directement par EasyMock et ne nécessite donc aucune assertion explicite.

C'est le mode de fonctionnement pas défaut d'EasyMock. Cependant, parfois ce mode de fonctionnement est trop strict.

Pour définir qu'un comportement ne requiert pas une valeur précise comme paramètre, il faut utiliser la méthode de la classe EasyMock correspondant au type attendu pour préciser la valeur d'un paramètre : anyObject(), anyInt(), anyShort(), anyByte(), anyLong(), anyFloat(), anyDouble() et anyBoolean().

La méthode EasyMock.notNull() permet de préciser qu'une valeur d'un paramètre doit être non null sans fournir plus de précision sur la valeur.

La méthode EasyMock.matches() permet de préciser qu'une valeur d'un paramètre correspond à un certain motif sous la forme d'une expression régulière.

La méthode EasyMock.find() permet de préciser qu'une valeur d'un paramètre doit contenir un sous-ensemble correspondant à un certain motif sous la forme d'une expression régulière.

La méthode EasyMock.lt() permet de préciser qu'une valeur d'un paramètre numérique doit être inférieure à la valeur fournie en paramètre. La méthode EasyMock.gt() permet de préciser qu'une valeur d'un paramètre numérique doit être supérieure à celle fournie en paramètre.

119.5.1.5. La vérification de l'ordre d'invocations des mocks

Il peut être important de vouloir vérifier l'ordre d'invocations des méthodes d'un objet mock. Attention cependant, ceci induit un couplage entre l'objet et son mock.

EasyMock ne se contente pas de vérifier les valeurs des paramètres des méthodes invoquées par le mock et les valeurs retournées. Elle vérifie aussi l'ordre d'invocation de ces méthodes et s'assure que seules les méthodes définies dans le comportement son invoquées. Cette vérification n'est pas faite pas défaut : pour l'activer, il faut utiliser la méthode EasyMock.verify() une fois toutes les invocations des méthodes du mock réalisées.

Exemple :

```
  @Test
  public void testCalculer() {
    long retour = 0l;
    EasyMock.expect(mock.additionner(20, 30)).andReturn(Long.valueOf(50l));
    EasyMock.expect(mock.soustraire(50, 10)).andReturn(Long.valueOf(40l));
    EasyMock.expect(mock.additionner(40, 60)).andReturn(Long.valueOf(100l));
    EasyMock.replay(mock);

    retour = monService.calculer(20, 30, 10);
    Assert.assertEquals("La valeur retournee est invalide", 40l, retour);
  }
```

Ce test s'exécute sans problème.

Exemple :

```
  @Test
  public void testCalculer() {
    long retour = 0l;

    EasyMock.expect(mock.additionner(20, 30)).andReturn(Long.valueOf(50l));
    EasyMock.expect(mock.soustraire(50, 10)).andReturn(Long.valueOf(40l));
    EasyMock.expect(mock.additionner(40, 60)).andReturn(Long.valueOf(100l));
    EasyMock.replay(mock);

    retour = monService.calculer(20, 30, 10);
    Assert.assertEquals("La valeur retournee est invalide", 40l, retour);
    EasyMock.verify(mock);
  }
```

Ce test échoue car lors de la vérification, le comportement précise une seconde invocation de la méthode additionner() qui n'est pas réalisée dans les traitements.

Une exception est alors levée par EasyMock pour signaler la différence entre le comportement défini et les traitements invoqués.

Exemple :

```
java.lang.AssertionError:
  Expectation failure on verify:
    additionner(40, 60): expected: 1, actual: 0
        at org.easymock.internal.MocksControl.verify(MocksControl.java:101)
        at org.easymock.EasyMock.verify(EasyMock.java:1570)
        at fr.jmdoudoux.dej.MonServiceTest.testCalculer(MonServiceTest.java:59)
```

Ceci peut être particulièrement utile dans certaines circonstances.

Le fonctionnement de la méthode verify() dépend du mode de création de l'objet de type mock. Pour rappel, EasyMock propose trois modes de création :

- normal : toutes les méthodes doivent être invoquées avec les arguments fournis sans tenir compte de leur ordre d'invocation. L'appel à une méthode non définie dans le comportement ou avec des paramètres différents fait échouer le test. La création d'un objet mock dans ce mode se fait avec la méthode EasyMock.createMock()
- strict : toutes les méthodes doivent être invoquées avec les arguments fournis en tenant compte de leur ordre d'invocation. L'appel à une méthode non définie dans le comportement avec des paramètres différents ou dans un ordre différent fait échouer le test. La création d'un objet mock dans ce mode se fait avec la méthode EasyMock.createStrictMock()
- nice : toutes les méthodes doivent être invoquées avec les arguments fournis sans tenir compte de leur ordre d'invocation. L'appel à une méthode non définie dans le comportement ou avec des paramètres différents renvoie une valeur par défaut selon le type de donnée retourné. La création d'un objet mock dans ce mode se fait avec la méthode EasyMock.createNiceMock()

119.5.1.6. La gestion de plusieurs objets de type mock

Un objet de type IMocksControl permet de coupler plusieurs objets de type mock. Ce couplage permet de maintenir des relations sur l'ordre du comportement des différents objets mock.

La mise en oeuvre d'EasyMock avec un control nécessite plusieurs instances :

- un objet de type IMocksControl
- plusieurs instances d'objets de type mock

L'interface IMocksControl propose des méthodes similaires pour mettre en oeuvre les mocks, notamment : createMock(), replay() et verify().

La méthode checkOrder() permet de préciser si l'ordre d'invocations des méthodes des mocks doit être vérifié ou non.

La méthode reset() permet de supprimer tous les comportements définis dans les mocks du control.

Les trois méthodes resetToNice(), resetToStrict() et resetToDefault() permettent de supprimer tous les comportements définis dans les mocks du control et de basculer les mocks, respectivement, en mode nice, strict et défaut.

Toutes ces méthodes agissent sur les mocks définis dans le control.

119.5.1.7. Un exemple complexe

L'exemple de cette section va mocker le comportement d'une classe de type DAO en proposant notamment d'utiliser des objets de type mock pour les classes Connection, PreparedStatement et ResultSet.

Ainsi, il sera possible de tester unitairement la méthode du DAO sans avoir à faire appel à la base de données.

La suite de cette section sera développée dans une version future de ce document

119.5.2. Mockito

Le site officiel du projet est à l'url https://site.mockito.org/

119.5.3. JMock

Le site officiel du projet est à l'url http://jmock.org/

La suite de cette section sera développée dans une version future de ce document

119.5.4. MockRunner

Certaines classes de l'API standard sont particulièrement complexes à mocker. MockRunner propose un ensemble de mocks pour quelques-unes de ces classes.

Le site officiel du projet est à l'url http://mockrunner.sourceforge.net/

La suite de cette section sera développée dans une version future de ce document

119.6. Les inconvénients des objets de type mock

La génération d'objets mock n'est pas toujours pratique car elle permet de créer des objets mais ceux-ci doivent être maintenus au fur et à mesure des évolutions du code à tester.

Il est préférable d'utiliser un framework qui va créer dynamiquement les objets mock. L'inconvénient c'est que le code du test unitaire devient plus important, donc plus complexe et ainsi, plus difficile à maintenir.

L'utilisation d'objets de type mock peut coupler les tests unitaires avec l'implémentation des dépendances utilisées. La plupart des frameworks permettent de préciser et de vérifier l'ordre et le nombre d'appels des méthodes mockées qui sont invoquées lors des tests. Si un refactoring est appliqué sur ces méthodes, changeant leur ordre d'invocation, le test devra être adapté en conséquence.

La mise en oeuvre d'objets de type mock doit tenir compte des limites de leur utilisation. Par exemple, elle masque complètement les problèmes d'interactions entre les dépendances. C'est pour cette raison que les tests unitaires sont nécessaires mais pas suffisants. Il peut aussi être intéressant de ne pas mocker systématiquement toutes les dépendances.

Partie 17 : Les outils de profiling et monitoring

Partie 17 :

Les outils
de profiling
et de monitoring

Cette partie contient les chapitres suivants :

- Arthas : Cette section détaille la mise en oeuvre de l'outil de profiling et de monitoring Arthas
- VisualVM : VisualVM est un outil historiquement fourni dans la JDK est maintenant en open source qui propose des fonctionnalités de monitoring et de profiling basic extensible par plug-in

120. Arthas

Chapitre 120

Niveau :	🟩🟩🟨🟨⬜⬜ Supérieur
Version utilisée :	3.1.7

Il n'est pas envisageable de déboguer une application en production car cela dégraderait les performances voir interromprait l'exécution pour une durée plus ou moins longue.

Généralement, on tente de reproduire la situation dans un autre environnement mais cela est parfois compliqué notamment lorsque les données et l'environnement ne sont pas similaires, les scénarios ou le timing sont différents.

Il est aussi possible d'ajouter des logs pour obtenir des informations mais cela implique de modifier le code et de redéployer le code en production avec tout ce que cela peut engendrer dans le respect du cycle de vie d'une nouvelle version et surtout l'interruption de service lors de redéploiement. Tout cela pour un résultat qui n'est toujours probant du premier coup et peut éventuellement nécessiter plusieurs itérations.

Arthas est un outil de diagnostic Java open source créé par Alibaba, qui propose de nombreuses fonctionnalités qui peuvent aider à identifier de nombreux problèmes dans des applications exécutées dans une JVM sans avoir à la redémarrer. C'est un outil en mode console qui permet de se connecter à une application Java en cours d'exécution.

Le grand intérêt d'Arthas est de permettre d'obtenir de nombreuses informations pour identifier des problèmes dans une application exécutée dans une JVM sans avoir à rejouter du code ni même de redémarrer la JVM. Arthas peut ainsi être utilisable dans un environnement de production puisqu'il peut être utilisé en ligne de commande locale, sans avoir à redémarrer la JVM.

Arthas offre de nombreuses fonctionnalités qui peuvent être très utiles dans de nombreuses situations notamment :

- Connexion à une JVM sans avoir à la redémarrer : utilisation sur une JVM locale ou distante en utilisant le CLI ou la Web console via telnet ou WebSocket
- Surveillance en temps réel de l'état de la JVM
- Obtenir des informations sur les ClassLoaders
- Rechercher des classes et des méthodes
- Obtention dynamique d'informations sur le code exécuté
- Obtenir des informations sur l'invocation de méthodes (paramètres, valeur de retour, exceptions)
- Obtenir la stacktrace d'invocation de méthodes
- Tracer le temps d'invocation de méthodes
- Obtenir des statistiques d'invocations de méthodes
- La possibilité de décompiler, recompiler et remplacer des classes chargées dans la JVM
- ...

Arthas est utilisable sur les systèmes d'exploitation Linux, Mac et Windows disposant d'un JDK 6 minimum.

Le site officiel d'Arthas est à l'url : https://alibaba.github.io/arthas/index.html

Le projet est sur GitHub : https://github.com/alibaba/arthas

Ce chapitre contient plusieurs sections :

♦ Installation

- Le démarrage
- Les fonctionnalités
- L'exécution de commandes en asynchrone
- Les scripts batch
- La console web

120.1. Installation

Arthas peut être utilisé sur tout système qui possède une JVM Java version 6 minimum.

L'installation peut être réalisée de différentes manières selon le système d'exploitation utilisé.

120.1.1. L'installation avec le fichier arthas-boot.jar

Le plus simple pour utiliser Arthas est de télécharger le fichier jar auto-exécutable nommé arthas-boot.jar à l'url https://alibaba.github.io/arthas/arthas-boot.jar

Par exemple en utilisant curl

Résultat :

```
jm@L-X1:/mnt/c/temp/arthas$ curl https://alibaba.github.io/arthas/arthas-boot.jar --output
arthas-boot.jar

  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  108k  100  108k    0     0  38441      0  0:00:02  0:00:02 --:--:-- 38439
jm@L-X1:/mnt/c/temp/arthas$
```

ou en utilisant wget

Résultat :

```
jm@L-X1:/mnt/c/temp/arthas$ wget https://alibaba.github.io/arthas/arthas-boot.jar
--2020-03-13 19:48:29--  https://alibaba.github.io/arthas/arthas-boot.jar
Résolution de alibaba.github.io (alibaba.github.io)... 185.199.109.153, 185.199.111.153,
 185.199.110.153, ...
Connexion à alibaba.github.io (alibaba.github.io)|185.199.109.153|:443... connecté.
requête HTTP transmise, en attente de la réponse... 200 OK
Taille : 111090 (108K) [application/java-archive]
Enregistre : «arthas-boot.jar»

arthas-boot.jar              100%[================================================>] 108,49K
   333KB/s    in 0,3s

2020-03-13 19:48:31 (333 KB/s) - «arthas-boot.jar» enregistré [111090/111090]

jm@L-X1:/mnt/c/temp/arthas$
```

Il faut exécuter le fichier arthas-boot.jar et avoir une JVM en cours d'exécution pour que l'application puisse s'y connecter. Comme c'est la première exécution, l'application Arthas est téléchargée et installée dans le sous-répertoire .arthas du répertoire home de l'utilisateur.

Résultat :

```
C:\java> java -jar arthas-boot.jar
[INFO] arthas-boot version: 3.1.7
[INFO] Found existing java process, please choose one and hit RETURN.
* [1]: 4192 arthas-demo.jar
```

```
[INFO] Start download arthas from remote server: https://repo1.maven.org/maven2/com/taobao/arth
as/arthas-packaging/3.1.7/arthas-packaging-3.1.7-bin.zip
[INFO] File size: 10,33 MB, downloaded size: 108,13 KB, downloading ...
[INFO] File size: 10,33 MB, downloaded size: 270,50 KB, downloading ...
...
[INFO] File size: 10,33 MB, downloaded size: 10,17 MB, downloading ...
[INFO] File size: 10,33 MB, downloaded size: 10,33 MB, downloading ...
[INFO] Download arthas success.
[INFO] arthas home: C:\Users\jm\.arthas\lib\3.1.7\arthas
[INFO] Try to attach process 4192
[INFO] Attach process 4192 success.
[INFO] arthas-client connect 127.0.0.1 3658
  ,---.  ,------. ,--------.,--.  ,--.  ,---.   ,---.
 /  O  \ |  .--. ''--.  .--'|  '--'  | /  O  \ '   .-'
|  .-.  ||  '--'.'   |  |   |  .--.  ||  .-.  |`.  `-.
|  | |  ||  |\  \    |  |   |  |  |  ||  | |  |.-'    |
`--' `--'`--' '--'   `--'   `--'  `--'`--' `--'`-----'


wiki      https://alibaba.github.io/arthas
tutorials https://alibaba.github.io/arthas/arthas-tutorials
version   3.1.7
pid       4192
time      2020-03-15 14:24:57
```

120.1.2. L'installation via les binaires sous Windows

Pour cela, il faut télécharger une archive zip sur Maven Central dont le groupId est com.taobao.arthas et l'artefactId est arthas-packaging :

L'archive peut aussi être téléchargée sur la page des releases d'Arthas : https://github.com/alibaba/arthas/releases

Il faut ensuite décompresser l'archive.

Pour lancer Arthas, il faut exécuter le script as.bat dans le sous-répertoire bin.

120.1.3. L'installation via un script sous Linux et Mac

Il est possible de télécharger et d'exécuter le script install.sh proposé par Arthas. Les exemples de cette section sont dans un Ubuntu On Windows (WSL).

Résultat :

```
jm@L-X1:/mnt/c/temp/arthas$ curl -L https://alibaba.github.io/arthas/install.sh | sh
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   961  100   961    0     0    752      0  0:00:01  0:00:01 --:--:--   752
downloading... ./as.sh.23
Arthas install successed.
jm@L-X1:/mnt/c/temp/arthas$ ls -al
total 28
drwxrwxrwx 1 jm jm   512 mars  14 18:27 .
drwxrwxrwx 1 jm jm   512 mars  14 18:25 ..
-rwxrwxrwx 1 jm jm 28075 mars  14 18:27 as.sh
jm@L-X1:/mnt/c/temp/arthas$
```

Ce script peut être placé dans un répertoire quelconque et même déplacé par la suite mais il est pratique dans le rajouter dans la variable système PATH.

Le script requière l'outil unzip.

Résultat :

```
jm@L-X1:/mnt/c/temp/arthas$ ./as.sh
Error: unzip is not installed. Try to use java -jar arthas-boot.jar
```

Sur une distribution Ubuntu, il suffit d'exécuter la commande ci-dessous :

Résultat :

```
jm@L-X1:/mnt/c/temp/arthas$ sudo apt-get install zip
```

Lors de la première exécution, il faut préciser la version d'Arthas à utiliser et le pid de la JVM à laquelle Arthas va s'attacher.

Résultat :

```
jm@L-X1:/mnt/c/temp/arthas$ ./as.sh --use-version 3.1.7 829
Arthas script version: 3.1.7
[INFO] JAVA_HOME: /usr/lib/jvm/java-9-openjdk-amd64
updating version 3.1.7 ...
Download arthas from: https://repo1.maven.org/maven2/com/taobao/arthas/arthas-packaging/3.1.7/a
rthas-packaging-3.1.7-bin.zip

curl: (28) Connection timed out after 5001 milliseconds
update fail, ignore this update.
Arthas home: /home/jm/.arthas/lib/3.1.7/arthas
Arthas home is not a directory, please delete it and retry.
jm@L-X1:/mnt/c/temp/arthas$ ./as.sh --use-version 3.1.7 829
Arthas script version: 3.1.7
[INFO] JAVA_HOME: /usr/lib/jvm/java-9-openjdk-amd64
updating version 3.1.7 ...
Download arthas from: https://repo1.maven.org/maven2/com/taobao/arthas/arthas-packaging/3.1.7/a
rthas-packaging-3.1.7-bin.zip
########################################################################100,0%
Archive:  /tmp/temp_3.1.7_1377/arthas-3.1.7-bin.zip
   creating: /tmp/temp_3.1.7_1377/async-profiler/
  inflating: /tmp/temp_3.1.7_1377/arthas-spy.jar
  inflating: /tmp/temp_3.1.7_1377/arthas-agent.jar
  inflating: /tmp/temp_3.1.7_1377/arthas-client.jar
  inflating: /tmp/temp_3.1.7_1377/arthas-boot.jar
  inflating: /tmp/temp_3.1.7_1377/arthas-demo.jar
  inflating: /tmp/temp_3.1.7_1377/install-local.sh
  inflating: /tmp/temp_3.1.7_1377/as.sh
  inflating: /tmp/temp_3.1.7_1377/as.bat
  inflating: /tmp/temp_3.1.7_1377/as-service.bat
  inflating: /tmp/temp_3.1.7_1377/async-profiler/libasyncProfiler-linux-x64.so
  inflating: /tmp/temp_3.1.7_1377/async-profiler/libasyncProfiler-mac-x64.so
  inflating: /tmp/temp_3.1.7_1377/arthas-core.jar
update completed.
Arthas home: /home/jm/.arthas/lib/3.1.7/arthas
Calculating attach execution time...
Attaching to 829 using version /home/jm/.arthas/lib/3.1.7/arthas...
*** java.lang.instrument ASSERTION FAILED ***: "!errorOutstanding" with message trans
form method call failed at JPLISAgent.c line: 884
*** java.lang.instrument ASSERTION FAILED ***: "!errorOutstanding" with message trans
form method call failed at JPLISAgent.c line: 884


real    0m3.406s
user    0m0.734s
sys     0m0.234s
Attach success.
telnet connecting to arthas server... current timestamp is 1584124377
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
  ,---.  ,------. ,--------.,--.  ,--.  ,---.   ,---.
 /  O  \ |  .--. ''--.  .--'|  '--'  | /  O  \ '   .-'
|  .-.  ||  '--'.'   |  |   |  .--.  ||  .-.  |`.  `-.
|  | |  ||  |\  \    |  |   |  |  |  ||  | |  |.-'    |
`--' `--'`--' '--'   `--'   `--'  `--'`--' `--'`-----'


wiki      https://alibaba.github.io/arthas
tutorials https://alibaba.github.io/arthas/arthas-tutorials
version   3.1.7
pid       829
time      2020-03-13 19:32:56

[arthas@829]$
```

Le script télécharge l'archive zip sur le dépôt Maven Central et la décompresse dans le sous-répertoire .arthas/lib/{version}/arthas du répertoire home de l'utilisateur.

120.1.4. L'installation via des packages pour Debian et Fedora

Arthas propose des packages pour Debian et Fedora qu'il est possible de télécharger sur la page des releases : https://github.com/alibaba/arthas/releases.

Pour lancer l'installation sur Debian, il faut utiliser la commande ci-dessous :

Résultat :

```
$ sudo dpkg -i arthas*.deb
```

La commande à utiliser pour Debian est :

Résultat :

```
$ sudo rpm -i arthas*.rpm
```

120.1.5. La mise à jour

Lors du lancement d'Arthas via le jar arthas-boot.jar, une vérification est faite sur l'existence d'une nouvelle version et si c'est le cas, elle est automatiquement téléchargée et utilisée.

Résultat :

```
C:\java>java -jar arthas-boot.jar
[INFO] arthas-boot version: 3.1.7
[INFO] Found existing java process, please choose one and hit RETURN.
* [1]: 20032 arthas-demo.jar

[INFO] local lastest version: 3.1.7, remote lastest version: 3.2.0, try to download from remote
[INFO] Start download arthas from remote server: https://repo1.maven.org/maven2/com/taobao/arth
as/arthas-packaging/3.2.0/arthas-packaging-3.2.0-bin.zip
[INFO] File size: 10,82 MB, downloaded size: 649,35 KB, downloading ...
[INFO] File size: 10,82 MB, downloaded size: 1,14 MB, downloading ...
[INFO] File size: 10,82 MB, downloaded size: 1,48 MB, downloading ...
...
[INFO] File size: 10,82 MB, downloaded size: 10,16 MB, downloading ...
[INFO] File size: 10,82 MB, downloaded size: 10,52 MB, downloading ...
[INFO] Download arthas success.
[INFO] arthas home: C:\Users\jm\.arthas\lib\3.2.0\arthas
[INFO] Try to attach process 20032
```

120.1.6. La désinstallation

Il faut supprimer les sous-répertoires .arthas, logs/arthas et logs/arthas-cache dans le répertoire home de l'utilisateur.

Exemple sous Linux :

Résultat :

```
$ rm -rf ~/.arthas/

$ rm -rf ~/logs/arthas/
```

120.2. Le démarrage

Pour s'éviter des ennuis, il faut utiliser le même utilisateur pour lancer l'application dans une JVM et Arthas ou utiliser un utilisateur qui a les mêmes privilèges.

Au cas où Arthas n'arrive pas à s'attacher à la JVM, il faut consulter les logs d'Arthas qui sont dans le sous-répertoire logs/arthas du répertoire home de l'utilisateur.

Le démarrage d'Arthas peut se faire de plusieurs manières.

Lorsque le client Arthas se connecte à la JVM, des messages sont affichés dans la console :

Résultat :

```
Sat Mar 14 21:52:07 CET 2020 com.taobao.arthas.agent.ArthasClassloader@27254f36 JM.Log:INFO Log
 root path: C:\Users\jm\logs\
Sat Mar 14 21:52:07 CET 2020 com.taobao.arthas.agent.ArthasClassloader@27254f36 JM.Log:INFO Set
 arthas log path: C:\Users\jm\logs\arthas
```

120.2.1. Le démarrage avec le jar arthas-boot.jar

Pour lancer Arthas, il suffit d'exécuter le fichier jar arthas-boot.jar téléchargé

Résultat :

```
java -jar arthas-boot.jar
[INFO] arthas-boot version: 3.1.7
[INFO] Found existing java process, please choose one and hit RETURN.
* [1]: 6852 arthas-demo.jar
```

Arthas affiche les JVM locales en cours d'exécution et il est possible de sélectionner celle à laquelle le client Arthas doit se connecter. Celle précédée d'une étoile est celle sélectionnée par défaut en appuyer simplement sur la touche « Entrée ».

Pour sélectionner une JVM en particulier sur suffit de saisir le numéro entre crochets et d'appuyer sur la touche « Entrée ».

Résultat :

```
[INFO] arthas home: C:\Users\jm\.arthas\lib\3.1.7\arthas
[INFO] Try to attach process 6852
[INFO] Attach process 6852 success.
[INFO] arthas-client connect 127.0.0.1 3658
  ,---.  ,------. ,--------.,--.  ,--.  ,---.   ,---.
 /  O  \ |  .--. ''--.  .--'|  '--'  | /  O  \ '   .-'
|  .-.  ||  '--'.'   |  |   |  .--.  ||  .-.  |`.  `-.
|  | |  ||  |\  \    |  |   |  |  |  ||  | |  |.-'    |
`--' `--'`--' '--'   `--'   `--'  `--'`--' `--'`-----'
```

```
wiki      https://alibaba.github.io/arthas
tutorials https://alibaba.github.io/arthas/arthas-tutorials
version   3.1.7
pid       6852
time      2020-02-23 18:31:28
```

Si Arthas n'arrive pas à détecter de JVM, alors il s'arrête avec un message d'erreur.

Résultat :

```
C:\java>java -jar arthas-boot.jar
[INFO] arthas-boot version: 3.1.7
[INFO] Can not find java process. Try to pass <pid> in command line.
Please select an available pid.
```

Il est possible de passer l'option -h ou --help à Arthas pour obtenir le détail des options utilisables.

Résultat :

```
C:\java> java -jar arthas-boot.jar -h
[INFO] arthas-boot version: 3.1.7
Usage: arthas-boot [--repo-mirror <value>] [--session-timeout <value>]
       [--use-version <value>] [--http-port <value>] [--arthas-home <value>]
       [-h] [--target-ip <value>] [--telnet-port <value>] [-c <value>]
       [--tunnel-server <value>] [--agent-id <value>] [--width <value>]
       [--versions] [--height <value>] [--use-http] [--stat-url <value>]
       [--attach-only] [-f <value>] [-v] [pid]

Bootstrap Arthas

EXAMPLES:
  java -jar arthas-boot.jar <pid>
  java -jar arthas-boot.jar --target-ip 0.0.0.0
  java -jar arthas-boot.jar --telnet-port 9999 --http-port -1
  java -jar arthas-boot.jar --tunnel-server 'ws://192.168.10.11:7777/ws'
  java -jar arthas-boot.jar --tunnel-server 'ws://192.168.10.11:7777/ws'
--agent-id bvDOe8XbTM2pQWjF4cfw
  java -jar arthas-boot.jar --stat-url 'http://192.168.10.11:8080/api/stat'
  java -jar arthas-boot.jar -c 'sysprop; thread' <pid>
  java -jar arthas-boot.jar -f batch.as <pid>
  java -jar arthas-boot.jar --use-version 3.1.7
  java -jar arthas-boot.jar --versions
  java -jar arthas-boot.jar --session-timeout 3600
  java -jar arthas-boot.jar --attach-only
  java -jar arthas-boot.jar --repo-mirror aliyun --use-http
WIKI:
  https://alibaba.github.io/arthas
```

Arthas propose de nombreuses options utilisables à son démarrage :

Option	Rôle
--target-ip	Adresse IP de la JVM cible : la valeur par défaut est 127.0.0.1 (localhost)
--telnet-port	Le port telnet utilisé par la JVM pour communiquer avec le client : la valeur par défaut est 3658
--http-port	Le port http utilisé par la JVM pour accéder à la console web : la valeur par défaut est 8563
--use-version	Utiliser la version précisée d'Arthas
--use-http	Utiliser HTTP au lieu d'HTTPS utilisé par défaut pour les téléchargements

-v, --verbose	Afficher des informations de debug
-f, --batch-file <value>	Exécuter le script précisé comme valeur
--repo-mirror <value>	Utiliser le dépôt Maven précisé comme valeur
--session-timeout <value>	Définir le timeout de la session : par défaut 1800 (30 minutes)
--arthas-home <value>	Définir le répertoire qui contient Arthas
-h, --help	Afficher l'aide en ligne
-c, --command <value>	Exécuter la commande précisée comme valeur. Plusieurs commandes doivent être séparées avec un point-virgule
--tunnel-server <value>	
--agent-id <value>	
--width <value>	Définir la longueur du terminal Arthas
--versions	Afficher la liste des versions d'Arthas locales et téléchargeables
--height <value>	Définir la hauteur du terminal Arthas
--stat-url <value>	
--attach-only	S'attacher à la JVM sans s'y connecter
<pid>	Préciser le pid de la JVM

L'option --versions affiche les versions locales et celles qu'il est possible de télécharger.

Résultat :

```
java -jar arthas-boot.jar --versions
[INFO] arthas-boot version: 3.1.7
Local versions:
 3.1.7
Remote versions:
 3.1.7
 3.1.6
 3.1.5
 3.1.4
 3.1.3
 3.1.2
 3.1.1
 3.1.0
 3.0.5
 3.0.4
 3.0.3
 3.0.0-RC
```

120.2.2. Le démarrage avec le script as.sh/as.bat

Il est possible de démarrer Arthas en utilisant le script as.sh ou as.bat contenu dans l'archive d'arthas dans le sous-répertoire .arthas dans le répertoire de l'utilisateur.

Le script attend en paramètre le pid de la JVM : si celui-ci n'est pas fourni un message d'erreur s'affiche

Résultat :

```
C:\Users\jm\.arthas\lib\3.1.7\arthas> as.bat
Example:
  as.bat 452
  as.bat 452 --ignore-tools # for jdk 9/10/11
```

```
Need the pid argument, you can run jps to list all java process ids.
exit était inattendu.

C:\Users\jm\.arthas\lib\3.1.7\arthas>
```

Comme indiqué dans le message, il est possible d'utiliser la commande jps fournie dans le JDK pour trouver le pid de la JVM concernée.

Résultat :

```
C:\Users\jm\.arthas\lib\3.1.7\arthas> jps
8224 Jps
4856 jar

C:\Users\jm\.arthas\lib\3.1.7\arthas> as.bat 4856
JAVA_HOME: C:\Program Files\Java\jdk1.8.0_202
telnet port: 3658
http port: 8563

  ,---.  ,------. ,--------.,--.  ,--.  ,---.   ,---.
 /  O  \ |  .--. ''--.  .--'|  '--'  | /  O  \ '   .-'
|  .-.  ||  '--'.'   |  |   |  .--.  ||  .-.  |`.  `-.
|  | |  ||  |\  \    |  |   |  |  |  ||  | |  |.-'    |
`--' `--'`--' '--'   `--'   `--'  `--'`--' `--'`-----'


wiki      https://alibaba.github.io/arthas
tutorials https://alibaba.github.io/arthas/arthas-tutorials
version   3.1.7
pid       4856
time      2020-03-13 16:15:28
```

120.2.3. L'utilisation de la console CLI

Après le démarrage d'Arthas et la connexion à la JVM choisie, un prompt est affiché et attend la saisie d'une commande.

Résultat :

```
C:\java> java -jar arthas-boot.jar
[INFO] arthas-boot version: 3.1.7
[INFO] Process 14128 already using port 3658
[INFO] Process 14128 already using port 8563
[INFO] Found existing java process, please choose one and hit RETURN.
* [1]: 14128 arthas-demo.jar

[INFO] arthas home: C:\Users\jm\.arthas\lib\3.1.7\arthas
[INFO] The target process already listen port 3658, skip attach.
[INFO] arthas-client connect 127.0.0.1 3658
  ,---.  ,------. ,--------.,--.  ,--.  ,---.   ,---.
 /  O  \ |  .--. ''--.  .--'|  '--'  | /  O  \ '   .-'
|  .-.  ||  '--'.'   |  |   |  .--.  ||  .-.  |`.  `-.
|  | |  ||  |\  \    |  |   |  |  |  ||  | |  |.-'    |
`--' `--'`--' '--'   `--'   `--'  `--'`--' `--'`-----'


wiki      https://alibaba.github.io/arthas
tutorials https://alibaba.github.io/arthas/arthas-tutorials
version   3.1.7
pid       14128
time      2020-02-14 00:06:47


[arthas@14128]$
```

Il est alors possible de saisir des commandes qui seront exécutées en fonction des besoins.

La touche tabulation peut être utilisée après avoir saisie un ou plusieurs caractères pour aider à la complétion de différents éléments :

- les commandes
- le nom pleinement qualifié de classes
- le nom de méthodes

120.3. Les fonctionnalités

La console propose une interface en ligne de commandes (CLI) qui permet d'envoyer des commandes au serveur Arthas exécuté dans la JVM cible.

Lors de l'exécution de certaines commandes, Arthas affiche un message qui indique le nombre d'éléments impactés (classes, méthodes, lignes, ...) et le temps utilisé.

Résultat :

```
Affect(class-cnt:7 , method-cnt:13) cost in 94 ms.
```

120.3.1. Les fonctionnalités de base

Le CLI d'Arthas propose plusieurs fonctionnalités de base dont certaines sont inspirées par des commandes standard sous Unix.

Commande	Rôle
help	Afficher l'aide
cls	Effacer l'écran
cat	Concaténer / afficher un fichier
grep	Filtrer sur un motif
pwd	Renvoyer le répertoire courant
session	Afficher l'identifiant de la session
reset	Réinitialiser les classes qui ont été enrichies pour permettre leur instrumentation pour qu'elles reviennent à leur code d'origine
version	Afficher la version du CLI
history	Afficher l'historique des commandes
quit/exit	Terminer la session courante
stop/shutdown	Terminer le serveur Arthas : toutes les sessions sont terminées, tous les clients connectés au serveur sont déconnectés
keymap	Afficher les raccourcis clavier
options	Afficher les options globales d'Arthas et leurs valeurs

120.3.1.1. Quitter Arthas

Par défaut au bout de 30 minutes, la session entre le client et le serveur Arthas est fermée.

Résultat :

```
[arthas@17512]$ session (82a8208d-66fc-4b0a-95b9-1f949948bb1c) is closed because session is ina
ctive for 30 min(s).
```

Pour terminer la session et quitter Arthas, plusieurs commandes peuvent être utilisées.

Pour quitter la console Arthas, il faut utiliser la commande quit ou la commande exit. Ces deux commandes terminent la connexion avec le serveur sur la JVM en cours de monitoring et arrête la console. Attention dans ce cas, le code instrumenté n'est pas réinitialisé, le serveur sur la JVM est toujours en cours d'exécution et le port utilisé n'est pas libéré. Cela permet de connecter une autre console Arthas.

Pour arrêter complètement Arthas, le serveur et la console, il faut utiliser la commande shutdown ou la commande stop. Lors de l'invocation de l'une de ces deux méthodes, toutes les méthodes instrumentées pour capturer des informations sont réinitialisées pour revenir à leur code d'origine. Il est donc important de les utiliser notamment en production si la JVM doit poursuivre son exécution.

120.3.1.2. La commande cat

La commande cat a un rôle similaire à celle d'Unix : elle permet de concaténer et d'afficher le contenu d'un ou plusieurs fichiers.

La syntaxe de la commande cat est de la forme :

cat [--encoding <value>] [-h] files...

La commande cat possède plusieurs paramètres dont :

Paramètre	Rôle
--encoding <value>	Préciser le jeu d'encodage de caractères des fichiers

Résultat :

```
[arthas@14128]$ cat c:/temp/test.txt
ligne1
ligne2
ligne3[arthas@14128]$
```

120.3.1.3. La commande grep

Le commande grep a un rôle similaire à celle d'Unix : elle permet de filtrer les informations affichées par une commande à la console.

La syntaxe de la commande grep est de la forme :

grep [-A <value>] [-B <value>] [-C <value>] [-h] [-i] [-v] [-n] [-m <value>] [-e] [--trim-end <value>] pattern

La commande grep possède plusieurs paramètres dont :

Paramètre	Rôle
-i, --ignore-case	Filtrer sans tenir compte de la casse. Par défaut, la casse est prise en compte
-v, --invert-match	Filtrer les lignes qui ne respectent pas le motif
-n, --line-number	Afficher un numéro à chaque ligne
-m, --max-count <value>	Limiter le nombre de lignes affichées à celui précisé
-e, --regex	Activer l'utilisation d'une expression régulière
--trim-end <value>	Supprimer les espaces à la fin de chaque ligne. Valeur par défaut : true
<pattern>	Le motif à appliquer

Résultat :

```
[arthas@14128]$ thread 10 | grep fr.test
    at fr.jmdoudoux.dej.Service.traiter(Service.java:12)
    at fr.jmdoudoux.dej.ThreadService.run(ThreadService.java)
[arthas@14128]$
```

120.3.1.4. La commande pwd

La commande pwd renvoie le chemin du répertoire courant, celui dans lequel la CLI Arthas a été démarrée.

Résultat :

```
[arthas@14128]$ pwd
C:\java\Arthas-demo
[arthas@14128]$
```

120.3.1.5. La commande options

La commande options permet de voir et de modifier la valeur des options globales.

La syntaxe de la commande options est de la forme :

options [-h] [options-name] [options-value]

La commande options possède plusieurs paramètres dont :

Paramètre	Rôle
<options-name>	Le nom de l'option
<options-value>	La nouvelle valeur de l'option

Les options globales utilisables dans Arthas sont :

Nom	Valeur par défaut	Description
unsafe	false	Permettre d'utiliser des proxys sur les classes du JDK. A utiliser avec précaution car cela peut engendrer un crash de la JVM
dump	false	Effectuer un dump des classes enrichies dans des fichiers dont le chemin sera affiché dans la console
batch-re-transform	true	
json-format	false	Demander à ce que le résultat de l'exécution de commandes soit en JSON
disable-sub-class	false	Demander d'inclure les sous-classes lors de la recherche de correspondance sur le nom de classes
debug-for-asm	false	Demander le niveau de log debug pour ASM
save-result	false	Demander l'enregistrement des exécutions de commandes dans le fichier logs/arthas-cache/result.log dans le sous-répertoire home de l'utilisateur
job-timeout	1d	Durée du timeout par défaut des tâches exécutées en arrière-plan. Exemple : 1d, 2h, 5m, 10s

Sans paramètres, la commande options affiche la valeur de toutes les options globales.

Résultat :

```
[arthas@14128]$ options
 LEVEL TYPE     NAME           VALUE   SUMMARY                 DESCRIPTION
------------------------------------------------------------------------------------------------
 0     boolean  unsafe         false   Option to support syst  This option enables to proxy fun
                                       em-level class          ctionality of JVM classes. Due t
                                                               o serious security risk a JVM cr
                                                               ash is possibly be introduced. D
                                                               o not ctivate it unless you are
                                                               able to manage.
 1     boolean  dump           false   Option to dump the enh  This option enables the enhanced
                                       anced classes            classes to be dumped to externa
                                                               l file for further de-compilatio
                                                               n and analysis.
 1     boolean  batch-re-tran  true    Option to support batc  This options enables to reTransf
                sform                  h reTransform Class     orm classes with batch mode.
 2     boolean  json-format    false   Option to support JSON  This option enables to format ob
                                       format of object outpu  ject output with JSON when -x op
                                       t                       tion selected.
 1     boolean  disable-sub-c  false   Option to control incl  This option disable to include s
                lass                   ude sub class when cla  ub class when matching class.
                                       ss matching
 1     boolean  debug-for-asm  false   Option to print DEBUG   This option enables to print DEB
                                       message if ASM is invo  UG message of ASM for each metho
                                       lved                    d invocation.
 1     boolean  save-result    false   Option to print comman  This option enables to save each
                                       d's result to log file   command's result to log file, w
                                                               hich path is ${user.home}/logs/a
                                                               rthas-cache/result.log.
 2     String   job-timeout    1d      Option to job timeout   This option setting job timeout,
                                                               The unit can be d, h, m, s for d
                                                               ay, hour, minute, second. 1d is
                                                               one day in default
 1     boolean  print-parent-  true    Option to print all fi  This option enables print files
                fields                 leds in parent class    in parent class, default value t
                                                               rue.
[arthas@14128]$
```

Pour obtenir uniquement une seule option, il suffit de préciser son nom en paramètre.

Résultat :

```
[arthas@14128]$ options save-result
 LEVEL   TYPE     NAME           VALUE   SUMMARY                 DESCRIPTION
------------------------------------------------------------------------------------------------
 1       boolean  save-result    false   Option to print comman  This option enables to save ea
                                         d's result to log file  ch command's result to log fil
                                                                 e, which path is ${user.home}/
                                                                 logs/arthas-cache/result.log.
[arthas@14128]$
```

Pour modifier la valeur d'une option, il suffit de préciser son nom et sa nouvelle valeur en paramètre.

Résultat :

```
[arthas@14128]$ options save-result true
 NAME         BEFORE-VALUE  AFTER-VALUE
----------------------------------------
 save-result  false         true
[arthas@12512]$ options save-result
 LEVEL   TYPE     NAME           VALUE   SUMMARY                 DESCRIPTION
------------------------------------------------------------------------------------------------
 1       boolean  save-result    true    Option to print comman  This option enables to save ea
                                         d's result to log file  ch command's result to log fil
                                                                 e, which path is ${user.home}/
                                                                 logs/arthas-cache/result.log.
[arthas@14128]$
```

120.3.1.6. La commande help

La commande help affiche l'aide.

La syntaxe de la commande help est de la forme :

help [-h] [cmd]

La commande help possède plusieurs paramètres dont :

Paramètre	Rôle
<cmd>	Nom de la commande dont on souhaite obtenir l'aide

Sans option, la commande help affiche la liste des commandes utilisables.

Résultat :

```
[arthas@14128]$ help
 NAME         DESCRIPTION
 help         Display Arthas Help
 keymap       Display all the available keymap for the specified connection.
 sc           Search all the classes loaded by JVM
 sm           Search the method of classes loaded by JVM
 classloader  Show classloader info
 jad          Decompile class
...
```

Pour obtenir l'aide pour une commande particulière, il faut préciser le nom de la commande en paramètre de la commande help.

Résultat :

```
[arthas@14128]$ help version
 USAGE:
   version [-h]

 SUMMARY:
   Display Arthas version

 OPTIONS:
 -h, --help                           this help
[arthas@14128]$
```

De manière équivalente, il est possible d'utiliser la commande concernée en lui passant en paramètre -h.

Résultat :

```
[arthas@14128]$ version -h
 USAGE:
   version [-h]

 SUMMARY:
   Display Arthas version

 OPTIONS:
 -h, --help                           this help

[arthas@14128]$
```

120.3.1.7. La commande cls

La commande cls efface l'écran et repositionne le prompt en haut de l'écran.

La syntaxe de la commande cls est de la forme :

cls [-h]

120.3.1.8. La commande session

La commande session affiche l'identifiant de la session.

La syntaxe de la commande session est de la forme :

session [-h]

Résultat :

```
[arthas@14128]$ session
 Name        Value
--------------------------------------------------
 JAVA_PID    14128
 SESSION_ID  f69eeca0-0d2b-4285-904e-0b9e8572adec
[arthas@14128]$
```

120.3.1.9. La commande reset

La commande reset réinitialise les classes qui ont été enrichies pour permettre leur instrumentation afin qu'elles reviennent à leur code d'origine.

La syntaxe de la commande reset est de la forme :

reset [-h] [-E] [class-pattern]

La commande reset possède plusieurs paramètres dont :

Paramètre	Rôle
-E, --regex	Activer l'utilisation d'une expression régulière. Par défaut, seul le caractère * est utilisable
<class-pattern>	Motif pour le nom de classe

Sans paramètre, la commande réinitialise toutes les classes instrumentées

Résultat :

```
[arthas@14128]$ reset
Affect(class-cnt:6 , method-cnt:0) cost in 84 ms.
[arthas@14128]$
```

Il est possible de préciser en paramètre la classe concernée

Résultat :

```
[arthas@14128]$ reset fr.jmdoudoux.dej.Compteur
Affect(class-cnt:1 , method-cnt:0) cost in 50 ms.
[arthas@14128]$
```

120.3.1.10. La commande version

La commande affiche la version de la CLI.

Résultat :

```
[arthas@14128]$ version
3.1.7
[arthas@14128]$
```

120.3.1.11. La commande history

La commande history affiche l'historique des commandes.

La syntaxe de la commande history est de la forme :

history [-c <value>] [-h] [n]

La commande history possède plusieurs paramètres dont :

Paramètre	Rôle
-c, --clear <value>	Effacer l'historique
<n>	Nombre de commandes de l'historique à afficher

Sans option, la commande history affiche l'intégralité des commandes saisies.

Résultat :

```
[arthas@14128]$ history
    1  dashboard
    2  dashboard
    3  stop
    4  dashboard
    5  getstatic fr.jmdoudoux.dej.Main valeur
    6  vmoption
[arthas@14128]$
```

120.3.1.12. La commande keymap

La commande affiche les raccourcis clavier utilisables.

La syntaxe de la commande keymap est de la forme :

keymap [-h]

Résultat :

```
[arthas@14128]$ keymap

 Shortcut                    Description               Name
----------------------------------------------------------------------------------------------
 "\C-a"            Ctrl + a                    beginning-of-line
 "\C-e"            Ctrl + e                    end-of-line
 "\C-f"            Ctrl + f                    forward-word
 "\C-b"            Ctrl + b                    backward-word
 "\e[D"            Left arrow                  backward-char
 "\e[C"            Right arrow                 forward-char
 "\e[A"            Up arrow                    history-search-backward
 "\e[B"            Down arrow                  history-search-forward
 "\C-h"            Ctrl + h                    backward-delete-char
 "\C-?"            Ctrl + ?                    backward-delete-char
 "\C-u"            Ctrl + u                    undo
 "\C-d"            Ctrl + d                    delete-char
 "\C-k"            Ctrl + k                    kill-line
 "\C-i"            Ctrl + i                    complete
 "\C-j"            Ctrl + j                    accept-line
 "\C-m"            Ctrl + m                    accept-line
 "\C-w"            Ctrl + w                    backward-delete-word
 "\C-x\e[3~"       "\C-x\e[3~"                 backward-kill-line
 "\e\C-?"          "\e\C-?"                    backward-kill-word
 "\e[1~"           "\e[1~"                     beginning-of-line
 "\e[4~"           "\e[4~"                     end-of-line
 "\e[5~"           "\e[5~"                     beginning-of-history
 "\e[6~"           "\e[6~"                     end-of-history
 "\e[3~"           "\e[3~"                     delete-char
 "\e[2~"           "\e[2~"                     quoted-insert
 "\e[7~"           "\e[7~"                     beginning-of-line
 "\e[8~"           "\e[8~"                     end-of-line
 "\eOH"            "\eOH"                      beginning-of-line
 "\eOF"            "\eOF"                      end-of-line
 "\e[H"            "\e[H"                      beginning-of-line
 "\e[F"            "\e[F"                      end-of-line
[arthas@14128]$
```

120.3.2. Les pipes

Arthas propose un support des pipes de manière similaire aux shells sous Unix. Cela permet d'envoyer le résultat de la commande qui précède le pipe en entrée de la commande qui le suit. Les commandes utilisables après le pipe sont grep, plaintext et wc -l.

Résultat :

```
[arthas@14128]$ sm java.lang.String * | grep 'index'
java.lang.String indexOf(Ljava/lang/String;I)I
java.lang.String indexOf(Ljava/lang/String;)I
java.lang.String indexOf(II)I
java.lang.String indexOf(I)I
java.lang.String indexOf([CII[CIII)I
java.lang.String indexOf([CIILjava/lang/String;I)I
java.lang.String indexOfSupplementary(II)I
[arthas@14128]$ sm java.lang.String * | wc -l
94
[arthas@14128]$
```

120.3.3. Les fonctionnalités concernant la JVM

Arthas propose plusieurs commandes relatives à la JVM :

Commande	Rôle
dashboard	Afficher un tableau de bord sur l'activité de la JVM
thread	Afficher des informations sur les threads
jvm	Afficher des informations sur la JVM
sysprop	Afficher/modifier les propriétés systèmes de la JVM
sysenv	Afficher les variables d'environnement système
vmoption	Afficher/modifier les options de diagnostic de la JVM
logger	Afficher des informations sur les loggers, modifier le niveau de gravité d'un logger
mbean	Afficher les informations d'un MBean
heapdump	Créer un dump du heap dans un fichier au format hprof

120.3.3.1. La commande dashboard

La commande dashboard affiche un tableau de bord contenant des statistiques en temps réel sur certaines activités de la JVM : threads, mémoire, GC et des informations sur le système.

Les informations sont réaffichées toutes les 5 secondes par défaut. Il faut utiliser la combinaison de touches Ctrl+C pour arrêter le rafraîchissement.

Résultat :

```
ID    NAME                         GROUP        PRIORITY STATE    %CPU     TIME    INTERRUP DAEMON
10    Service_1                    main         5        TIMED_WA 100      2:4     false    false
14    AsyncAppender-Worker-arthas  system       5        WAITING  0        0:0     false    true
5     Attach Listener              system       5        RUNNABLE 0        0:0     false    true
3     Finalizer                    system       8        WAITING  0        0:0     false    true
2     Reference Handler            system       10       WAITING  0        0:0     false    true
11    Service_2                    main         5        TIMED_WA 0        2:2     false    false
4     Signal Dispatcher            system       9        RUNNABLE 0        0:0     false    true
31    Timer-for-arthas-dashboard-  system       10       RUNNABLE 0        0:0     false    true
29    as-command-execute-daemon    system       10       TIMED_WA 0        0:0     false    true
16    job-timeout                  system       5        TIMED_WA 0        0:0     false    true
1     main                         main         5        WAITING  0        0:0     false    false
17    nioEventLoopGroup-2-1        system       10       RUNNABLE 0        0:0     false    false
21    nioEventLoopGroup-2-2        system       10       RUNNABLE 0        0:0     false    false
26    nioEventLoopGroup-2-3        system       10       RUNNABLE 0        0:0     false    false
28    nioEventLoopGroup-2-4        system       10       RUNNABLE 0        0:0     false    false
18    nioEventLoopGroup-3-1        system       10       RUNNABLE 0        0:0     false    false
Memory                  used   total   max     usage   GC
heap                    34M    109M    1799M   1,91%   gc.ps_scavenge.count        2224
ps_eden_space           5M     23M     673M    0,87%   gc.ps_scavenge.time(ms)     4679
ps_survivor_space       160K   512K    512K    31,25%  gc.ps_marksweep.count       0
ps_old_gen              28M    85M     1349M   2,10%   gc.ps_marksweep.time(ms)    0
nonheap                 25M    28M     -1      90,26%
code_cache              3M     5M      240M    1,61%
metaspace               19M    19M     -1      96,51%
compressed_class_space  2M     2M      1024M   0,23%
Runtime
os.name                                                Windows 10
os.version                                             10.0
java.version                                           1.8.0_202
java.home                                              C:\Program Files\Java\jdk1.8.0_202\jre
systemload.average                                     -1,00
processors                                             4
uptime                                                 31006s
```

Les informations sont regroupées par thème

- Thread
- Memory
- GC
- Runtime

En haut du tableau de bord, une liste des threads en cours d'exécution, classés en fonction de l'utilisation du CPU, avec quelques informations sont affichées en colonnes :

Elles sont affichées en colonnes :

- ID : Id du thread
- NAME : nom du thread
- GROUP : nom du groupe de thread
- PRIORITY : priorité du thread (1 à 10, 10 étant la plus haute priorité).
- STATE : état du thread
- CPU% : pourcentage d'utilisation de la CPU par le thread, échantillonnée toutes les 100ms
- TIME : temps total d'exécution du thread au format minute:seconde
- INTERRUPTED : le thread est interrompu
- DAEMON : le thread est un démon

La section suivante concerné la mémoire : est découpée en deux parties, l'une concernant la mémoire de la JVM (heap (eden/survivor/old) et hors heap) et l'autre concernant le GC (statistiques sur le nombre et le time des collections dans la young et la old).

La dernière section affiche des informations de base sur l'environnement.

120.3.3.2. La commande heapdump

La commande heapdump permet de créer un fichier binaire au hprof du heap de la JVM.

Sans option, la commande heapdump créé un fichier dans le répertoire temporaire

Résultat :

```
[arthas@17512]$ heapdump
Dumping heap to C:\Users\jm\AppData\Local\Temp\heapdump2020-01-17-18-062141170027159928142.hpro
f...
Heap dump file created
[arthas@17512]$
```

L'option --live, la commande heapdump ne dump que les objets en cours d'utilisation.

Résultat :

```
[arthas@17512]$ heapdump
Dumping heap to C:\Users\jm\AppData\Local\Temp\heapdump2020-01-17-18-062141170027159928142.hpro
f...
Heap dump file created
[arthas@17512]$
```

Il est aussi possible de préciser le nom du fichier qui va contenir le dump.

Résultat :

```
[arthas@17512]$ heapdump c:/temp/dump.hprof
Dumping heap to c:/temp/dump.hprof...
Heap dump file created
[arthas@17512]$
```

120.3.3.3. La commande thread

La commande thread permet d'obtenir des informations sur les threads et leur stacktrace.

La syntaxe de la commande thread est de la forme :

thread [-h] [-b] [-i <value>] [--state <value>] [-n <value>] [id]

La commande thread possède plusieurs paramètres :

Paramètre	Rôle
Id	Id du thread dans la JVM
-n <value>	Afficher les nb threads les plus occupés avec leur stacktrace
-b	Identifier les threads qui bloquent les autres
-i <value>	Préciser un intervalle de temps en ms pour collecter les données utiles au calcul du ratio CPU
--state <value>	N'afficher que les threads dont le statut est précisé

Sans paramètre, la commande thread affiche la liste des threads.

Résultat :

```
[arthas@11516]$ thread
Threads Total: 15, NEW: 0, RUNNABLE: 6, BLOCKED: 0, WAITING: 5, TIMED_WAITING: 4, TERMINATED: 0
ID   NAME                         GROUP       PRIORITY STATE    %CPU    TIME     INTERRUP DAEMON
14   AsyncAppender-Worker-arthas system      5        WAITING  0       0:0      false    true
5    Attach Listener              system      5        RUNNABLE 0       0:0      false    true
3    Finalizer                    system      8        WAITING  0       0:0      false    true
2    Reference Handler            system      10       WAITING  0       0:0      false    true
10   Service_1                    main        5        TIMED_WA 0       0:2      false    false
11   Service_2                    main        5        TIMED_WA 0       0:4      false    false
4    Signal Dispatcher            system      9        RUNNABLE 0       0:0      false    true
22   as-command-execute-daemon    system      10       RUNNABLE 0       0:0      false    true
16   job-timeout                  system      5        TIMED_WA 0       0:0      false    true
1    main                         main        5        WAITING  0       0:0      false    false
17   nioEventLoopGroup-2-1        system      10       RUNNABLE 0       0:0      false    false
21   nioEventLoopGroup-2-2        system      10       RUNNABLE 0       0:0      false    false
18   nioEventLoopGroup-3-1        system      10       RUNNABLE 0       0:0      false    false
19   pool-1-thread-1              system      5        TIMED_WA 0       0:0      false    false
20   pool-2-thread-1              system      5        WAITING  0       0:0      false    false
Affect(row-cnt:0) cost in 111 ms.
[arthas@11516]$
```

Le calcul du ration CPU utilise lui-même du CPU : pour le réduire il est possible d'augmenter l'intervalle de temps de mesure en ms avec option -i.

Résultat :

```
[arthas@10552]$ thread -i 2000
Threads Total: 15, NEW: 0, RUNNABLE: 6, BLOCKED: 0, WAITING: 5, TIMED_WAITING: 4, TERMINATED: 0
ID   NAME                         GROUP       PRIORITY STATE    %CPU    TIME     INTERRUP DAEMON
11   Service_2                    main        5        TIMED_WA 54      0:3      false    false
10   Service_1                    main        5        TIMED_WA 45      0:2      false    false
14   AsyncAppender-Worker-arthas system      5        WAITING  0       0:0      false    true
5    Attach Listener              system      5        RUNNABLE 0       0:0      false    true
3    Finalizer                    system      8        WAITING  0       0:0      false    true
2    Reference Handler            system      10       WAITING  0       0:0      false    true
4    Signal Dispatcher            system      9        RUNNABLE 0       0:0      false    true
22   as-command-execute-daemon    system      10       RUNNABLE 0       0:0      false    true
16   job-timeout                  system      5        TIMED_WA 0       0:0      false    true
1    main                         main        5        WAITING  0       0:0      false    false
17   nioEventLoopGroup-2-1        system      10       RUNNABLE 0       0:0      false    false
21   nioEventLoopGroup-2-2        system      10       RUNNABLE 0       0:0      false    false
18   nioEventLoopGroup-3-1        system      10       RUNNABLE 0       0:0      false    false
19   pool-1-thread-1              system      5        TIMED_WA 0       0:0      false    false
20   pool-2-thread-1              system      5        WAITING  0       0:0      false    false
Affect(row-cnt:0) cost in 2006 ms.
```

L'option -n permet d'afficher la stacktrace des n (fournis comme valeur de l'option -n) threads les plus consommateur en CPU.

Résultat :

```
 [arthas@10552]$ thread -i 3000 -n 2
"Service_1" Id=10 cpuUsage=66% TIMED_WAITING
    at java.lang.Thread.sleep(Native Method)
    at java.lang.Thread.sleep(Thread.java:340)
    at java.util.concurrent.TimeUnit.sleep(TimeUnit.java:386)
    at fr.jmdoudoux.dej.Service.traiter(Service.java:12)
    at fr.jmdoudoux.dej.ThreadService.run(ThreadService.java:24)


"Service_2" Id=11 cpuUsage=33% TIMED_WAITING
    at java.lang.Thread.sleep(Native Method)
    at java.lang.Thread.sleep(Thread.java:340)
    at java.util.concurrent.TimeUnit.sleep(TimeUnit.java:386)
    at fr.jmdoudoux.dej.Service.traiter(Service.java:12)
    at fr.jmdoudoux.dej.ThreadService.run(ThreadService.java:24)


Affect(row-cnt:0) cost in 3012 ms.
```

Il est possible d'obtenir la stacktrace d'un thread simple passant son id en paramètre de la commande thread.

Résultat :

```
[arthas@10552]$ thread 10
"Service_1" Id=10 TIMED_WAITING
    at java.lang.Thread.sleep(Native Method)
    at java.lang.Thread.sleep(Thread.java:340)
    at java.util.concurrent.TimeUnit.sleep(TimeUnit.java:386)
    at fr.jmdoudoux.dej.Service.traiter(Service.java:12)
    at fr.jmdoudoux.dej.ThreadService.run(ThreadService.java:24)

Affect(row-cnt:0) cost in 7 ms.
```

L'option -b permet de trouver le ou les threads qui bloquent les autres.

L'option --state permet de n'afficher que les threads qui sont dans le statut précisé.

Résultat :

```
[arthas@10552]$ thread -state WAITING
Threads Total: 16, NEW: 0, RUNNABLE: 7, BLOCKED: 0, WAITING: 5, TIMED_WAITING: 4, TERMINATED: 0
ID   NAME                         GROUP       PRIORITY STATE    %CPU    TIME     INTERRUP DAEMON
14   AsyncAppender-Worker-arthas  system      5        WAITING  0       0:0      false    true
3    Finalizer                    system      8        WAITING  0       0:0      false    true
2    Reference Handler            system      10       WAITING  0       0:0      false    true
1    main                         main        5        WAITING  0       0:0      false    false
20   pool-2-thread-1              system      5        WAITING  0       0:0      false    false
Affect(row-cnt:0) cost in 102 ms.
```

120.3.3.4. La commande jvm

La commande jvm affiche des informations sur la JVM.

Résultat :

```
[arthas@17512]$ jvm
 RUNTIME
--------------------------------------------------------------------------------
 MACHINE-NAME                   17512@L-X1
 JVM-START-TIME                 2020-01-19 00:30:55
 MANAGEMENT-SPEC-VERSION        1.2
 SPEC-NAME                      Java Virtual Machine Specification
 SPEC-VENDOR                    Oracle Corporation
 SPEC-VERSION                   1.8
 VM-NAME                        Java HotSpot(TM) 64-Bit Server VM
 VM-VENDOR                      Oracle Corporation
 VM-VERSION                     25.202-b08
 INPUT-ARGUMENTS                []
 CLASS-PATH                     arthas-demo.jar
 BOOT-CLASS-PATH                C:\Program Files\Java\jdk1.8.0_202\jre\lib\resources.
                                jar;C:\Program Files\Java\jdk1.8.0_202\jre\lib\rt.jar
                                ;C:\Program Files\Java\jdk1.8.0_202\jre\lib\sunrsasig
                                n.jar;C:\Program Files\Java\jdk1.8.0_202\jre\lib\jsse
                                .jar;C:\Program Files\Java\jdk1.8.0_202\jre\lib\jce.j
                                ar;C:\Program Files\Java\jdk1.8.0_202\jre\lib\charset
                                s.jar;C:\Program Files\Java\jdk1.8.0_202\jre\lib\jfr.
                                jar;C:\Program Files\Java\jdk1.8.0_202\jre\classes
 LIBRARY-PATH                   C:\Program Files\Java\jdk1.8.0_202\bin;C:\WINDOWS\Sun
                                \Java\bin;C:\WINDOWS\system32;C:\WINDOWS;C:\outils\cm
                                der;.
```

```
-------------------------------------------------------------------------------------
 CLASS-LOADING
-------------------------------------------------------------------------------------
 LOADED-CLASS-COUNT             2429
 TOTAL-LOADED-CLASS-COUNT       2439
 UNLOADED-CLASS-COUNT           10
 IS-VERBOSE                     false

-------------------------------------------------------------------------------------
 COMPILATION
-------------------------------------------------------------------------------------
 NAME                           HotSpot 64-Bit Tiered Compilers
 TOTAL-COMPILE-TIME             3200(ms)

-------------------------------------------------------------------------------------
 GARBAGE-COLLECTORS
-------------------------------------------------------------------------------------
 PS Scavenge                    2866/3765(ms)
 [count/time]
 PS MarkSweep                   1/118(ms)
 [count/time]

-------------------------------------------------------------------------------------
 MEMORY-MANAGERS
-------------------------------------------------------------------------------------
 CodeCacheManager               Code Cache

 Metaspace Manager              Metaspace
                                Compressed Class Space

 PS Scavenge                    PS Eden Space
                                PS Survivor Space

 PS MarkSweep                   PS Eden Space
                                PS Survivor Space
                                PS Old Gen

-------------------------------------------------------------------------------------
 MEMORY
-------------------------------------------------------------------------------------
 HEAP-MEMORY-USAGE              76021760(72,50 MiB)/134217728(128,00 MiB)/1886912512(
 [committed/init/max/used]      1,76 GiB)/32195992(30,70 MiB)
 NO-HEAP-MEMORY-USAGE           21626880(20,63 MiB)/2555904(2,44 MiB)/-1(-1 B)/206968
 [committed/init/max/used]      00(19,74 MiB)
 PENDING-FINALIZE-COUNT         0

-------------------------------------------------------------------------------------
 OPERATING-SYSTEM
-------------------------------------------------------------------------------------
 OS                             Windows 10
 ARCH                           amd64
 PROCESSORS-COUNT               4
 LOAD-AVERAGE                   -1.0
 VERSION                        10.0

-------------------------------------------------------------------------------------
 THREAD
-------------------------------------------------------------------------------------
 COUNT                          16
 DAEMON-COUNT                   7
 PEAK-COUNT                     16
 STARTED-COUNT                  20
 DEADLOCK-COUNT                 0

-------------------------------------------------------------------------------------
 FILE-DESCRIPTOR
-------------------------------------------------------------------------------------
 MAX-FILE-DESCRIPTOR-COUNT      -1
 OPEN-FILE-DESCRIPTOR-COUNT     -1
Affect(row-cnt:0) cost in 39 ms.
[arthas@17512]$
```

Les informations relatives aux threads sont :

- COUNT : nombre de threads actifs
- DAEMON-COUNT : nombre de threads de type démon actifs
- PEAK-COUNT : nombre maximum de threads actifs depuis de démarrage de la JVM
- STARTED-COUNT : nombre total de threads créés depuis de démarrage de la JVM
- DEADLOCK-COUNT : nombre de threads en deadlock

Les informations relatives aux descripteurs de fichiers sont :

- MAX-FILE-DESCRIPTOR-COUNT : nombre maximum de descripteurs de fichiers que la JVM peut ouvrir
- OPEN-FILE-DESCRIPTOR-COUNT : nombre de descripteurs de fichiers ouvert par la JVM

120.3.3.5. La commande sysprop

La commande sysprop permet d'afficher et de modifier des propriétés systèmes de la JMV.

La syntaxe est de la forme :

sysprop [-h] [property-name] [property-value]

Sans option, la commande sysprop affiche toutes les propriétés systèmes avec leur valeur.

Résultat :

```
[arthas@17512]$ sysprop
 KEY                    VALUE
------------------------------------------------------------------------------------
 java.runtime.name      Java(TM) SE Runtime Environment
 sun.boot.library.pat   C:\Program Files\Java\jdk1.8.0_202\jre\bin
 h
 java.vm.version        25.202-b08
 java.vm.vendor         Oracle Corporation
 java.vendor.url        http://java.oracle.com/
 path.separator         ;
 java.vm.name           Java HotSpot(TM) 64-Bit Server VM
 file.encoding.pkg      sun.io
 user.script
 user.country           FR
 sun.java.launcher      SUN_STANDARD
 sun.os.patch.level
 java.vm.specificatio   Java Virtual Machine Specification
 n.name
 user.dir               C:\java\workspace-2018-12-Test\Arthas-demo
 java.runtime.version   1.8.0_202-b08
 JM.LOG.PATH            C:\Users\jm\logs
 java.awt.graphicsenv   sun.awt.Win32GraphicsEnvironment
 java.endorsed.dirs     C:\Program Files\Java\jdk1.8.0_202\jre\lib\endorsed
 os.arch                amd64
 java.io.tmpdir         C:\Users\jm\AppData\Local\Temp\
 line.separator

 java.vm.specificatio   Oracle Corporation
 n.vendor
 user.variant
 os.name                Windows 10
 sun.jnu.encoding       Cp1252
 java.library.path      C:\Program Files\Java\jdk1.8.0_202\bin;C:\WINDOWS\Sun\Java\bin
                        ;C:\WINDOWS\system32;C:\WINDOWS;C:\Program Files\Java\jdk1.8.0
                        _202\bin;C:\outils\cmder\bin;C:\Program Files\Git\bin;C:\Progr
                        am Files\Git\usr\bin;C:\Program Files\Git\share\vim\vim74;C:\o
                        utils\cmder\vendor\conemu-maximus5\ConEmu\Scripts;C:\outils\cm
                        der\vendor\conemu-maximus5;C:\outils\cmder\vendor\conemu-maxim
                        us5\ConEmu;C:\Program Files\Java\jdk1.8.0_202\bin;c:\java;C:\P
                        rogram Files (x86)\Common Files\Oracle\Java\javapath;C:\Progra
                        m Files (x86)\EasyPHP-DevServer-14.1VC11\binaries\mysql/bin;c:
                        \outils;C:\java\apache-maven-3.6.0\bin;C:\Program Files (x86)\
                        EasyPHP-DevServer-14.1VC11\binaries\php\php_runningversion;C:\
                        ProgramData\Oracle\Java\javapath;C:\Program Files\Intel\iCLS C
                        lient\;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem
                        ;C:\WINDOWS\System32\WindowsPowerShell\v1.0\;C:\java\apache-an
                        t-1.9.6\bin;;C:\Program Files\Git\cmd;C:\Program Files\PuTTY\;
                        C:\WINDOWS\System32\OpenSSH\;C:\outils\OpenSSL-Win64\bin;C:\Pr
                        ogram Files (x86)\Microsoft VS Code\bin;C:\Program Files\nodej
                        s\;C:\Users\jm\AppData\Roaming\npm;C:\Program Files\Docker Too
                        lbox;C:\Program Files\Intel\WiFi\bin\;C:\outils\cmder;.
 sun.nio.ch.bugLevel
 java.specification.n  Java Platform API Specification
 ame
 java.class.version    52.0
 sun.management.compi  HotSpot 64-Bit Tiered Compilers
 ler
 os.version            10.0
 user.home             C:\Users\jm
 user.timezone         Europe/Paris
 java.awt.printerjob   sun.awt.windows.WPrinterJob
 java.specification.v  1.8
 ersion
 file.encoding         Cp1252
 user.name             jm
 java.class.path       arthas-demo.jar
 java.vm.specificatio  1.8
 n.version
 sun.arch.data.model   64
 java.home             C:\Program Files\Java\jdk1.8.0_202\jre
 sun.java.command      arthas-demo.jar
 java.specification.v  Oracle Corporation
 endor
 user.language         fr
 awt.toolkit           sun.awt.windows.WToolkit
 java.vm.info          mixed mode
 java.version          1.8.0_202
 java.ext.dirs         C:\Program Files\Java\jdk1.8.0_202\jre\lib\ext;C:\WINDOWS\Sun\
                        Java\lib\ext
 sun.boot.class.path   C:\Program Files\Java\jdk1.8.0_202\jre\lib\resources.jar;C:\Pr
                        ogram Files\Java\jdk1.8.0_202\jre\lib\rt.jar;C:\Program Files\
                        Java\jdk1.8.0_202\jre\lib\sunrsasign.jar;C:\Program Files\Java
                        \jdk1.8.0_202\jre\lib\jsse.jar;C:\Program Files\Java\jdk1.8.0_
                        202\jre\lib\jce.jar;C:\Program Files\Java\jdk1.8.0_202\jre\lib
                        \charsets.jar;C:\Program Files\Jav a\jdk1.8.0_202\jre\lib\jfr.
                        jar;C:\Program Files\Java\jdk1.8.0_202\jre\classes
 sun.stderr.encoding   cp850
 java.vendor           Oracle Corporation
 file.separator        \
 java.vendor.url.bug   http://bugreport.sun.com/bugreport/
 sun.cpu.endian        little
 sun.io.unicode.encod  UnicodeLittle
 ing
 sun.stdout.encoding   cp850
 sun.desktop           windows
 sun.cpu.isalist       amd64
[arthas@17512]$
```

Il est possible de préciser la propriété à afficher. La touche tab peut être utilisée pour compléter le nom de la propriété par auto-complétion.

Résultat :

```
[arthas@17512]$ sysprop java.version
java.version=1.8.0_202
[arthas@17512]$
```

Il est possible de modifier ou de définir une propriété en précisant en option son nom et sa valeur.

Résultat :

```
[arthas@17512]$ sysprop | grep ma_propriete
[arthas@17512]$ sysprop ma_propriete false
Successfully changed the system property.
ma_propriete=false
[arthas@17512]$ sysprop | grep ma_propriete
 ma_propriete          false
```

120.3.3.6. La commande sysenv

La commande sysenv affiche les variables d'environnement système.

La syntaxe est de la forme :

sysenv [-h] [env-name]

Sans option, la commande sysenv affiche toutes les variables d'environnement système et leur valeur.

Résultat :

```
[arthas@17512]$ sysenv

 KEY                   VALUE
-------------------------------------------------------------------------------------
 configsetroot         C:\WINDOWS\ConfigSetRoot
 USERDOMAIN_ROAMINGPR  L-X1
 OFILE
 NO_PROXY              192.168.99.100
 PROCESSOR_LEVEL       6
 ConEmuWorkDrive       C:
 FP_NO_HOST_CHECK      NO
 SESSIONNAME           Console
 ALLUSERSPROFILE       C:\ProgramData
 PROCESSOR_ARCHITECTU  AMD64
 RE
 ConEmuANSI            ON
 GIT_INSTALL_ROOT      C:\Program Files\Git
 PSModulePath          C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules\
 SystemDrive           C:
 =ExitCode             00000001
 MAVEN_HOME            C:\java\apache-maven-3.6.0

...
```

Pour n'afficher qu'une seule variable, il suffit de passer son nom en paramètre de la commande.

Résultat :

```
[arthas@9840]$ sysenv username
username=jm
[arthas@9840]$
```

Il est possible d'utiliser la touche tab pour invoquer l'auto-complétion sur le nom de la variable

Résultat :

```
[arthas@9840]$ sysenv USER[tab]

USERDOMAIN_ROAMINGPROFILE USERNAME               USERDOMAIN               USERPROFILE

[arthas@9840]$ sysenv USER
```

120.3.3.7. La commande vmoption

La commande vmoption permet d'afficher ou de modifier les options de la JVM relatives au diagnostic.

Sans option, la commande vmoption affiche les valeurs des options de diagnostic.

Résultat :

```
[arthas@6672]$ vmoption
 KEY                         VALUE                 ORIGIN                   WRITEABLE
-------------------------------------------------------------------------------------
 HeapDumpBeforeFullGC        false                 DEFAULT                  true
 HeapDumpAfterFullGC         false                 DEFAULT                  true
 HeapDumpOnOutOfMemoryError  false                 DEFAULT                  true
 HeapDumpPath                                      DEFAULT                  true
 CMSAbortablePrecleanWaitMi  100                   DEFAULT                  true
 llis
 CMSWaitDuration             2000                  DEFAULT                  true
 CMSTriggerInterval          -1                    DEFAULT                  true
 PrintGC                     false                 DEFAULT                  true
 PrintGCDetails              false                 DEFAULT                  true
 PrintGCDateStamps           false                 DEFAULT                  true
 PrintGCTimeStamps           false                 DEFAULT                  true
 PrintGCID                   false                 DEFAULT                  true
 PrintClassHistogramBeforeF  false                 DEFAULT                  true
 ullGC
 PrintClassHistogramAfterFu  false                 DEFAULT                  true
 llGC
 PrintClassHistogram         false                 DEFAULT                  true
 MinHeapFreeRatio            0                     DEFAULT                  true
 MaxHeapFreeRatio            100                   DEFAULT                  true
 PrintConcurrentLocks        false                 DEFAULT                  true
 UnlockCommercialFeatures    false                 DEFAULT                  true
[arthas@6672]$
```

Il est possible de n'afficher que l'option passée en paramètre

Résultat :

```
[arthas@6672]$ vmoption PrintGC
 KEY                         VALUE                ORIGIN                    WRITEABLE
-------------------------------------------------------------------------------------
 PrintGC                     false                MANAGEMENT                true
[arthas@6672]$
```

Il est possible de modifier la valeur d'une option en précisant cette valeur à la suite du nom de l'option concernée

Résultat :

```
[arthas@6672]$ vmoption
PrintGC true
Successfully updated the vm
option.
PrintGC=true
[arthas@6672]$
```

120.3.3.8. La commande mbean

La commande mbean permet d'afficher des informations sur des MBeans.

La syntaxe est de la forme :

mbean [-h] [-i <value>] [-m <value>] [-n <value>] [-E] [name-pattern] [attribute-pattern]

La commande mbean possède plusieurs paramètres optionnels :

Paramètre	Rôle
name-pattern	Motif pour le nom de MBean
attribute-pattern	Motif pour le nom d'attribut
[m]	Afficher les méta informations
[i:]	Définir l'intervalle de temps entre les rafraîchissements de la valeur des attributs (en ms)
[n:]	Nombre de rafraîchissements
[E]	Activer l'utilisation d'une expression régulière pour le nom de l'attribut

Sans paramètre, la commande mbean affiche la liste des MBeans disponibles.

Résultat :

```
[arthas@12844]$ mbean
java.lang:type=MemoryPool,name=Metaspace
java.lang:type=MemoryPool,name=PS Old Gen
java.lang:type=GarbageCollector,name=PS Scavenge
java.lang:type=MemoryPool,name=PS Eden Space
JMImplementation:type=MBeanServerDelegate
java.lang:type=Runtime
java.lang:type=Threading
java.lang:type=OperatingSystem
java.lang:type=MemoryPool,name=Code Cache
java.nio:type=BufferPool,name=direct
java.lang:type=Compilation
java.lang:type=MemoryManager,name=CodeCacheManager
java.lang:type=MemoryPool,name=Compressed Class Space
java.lang:type=Memory
java.nio:type=BufferPool,name=mapped
java.util.logging:type=Logging
java.lang:type=MemoryPool,name=PS Survivor Space
java.lang:type=ClassLoading
java.lang:type=MemoryManager,name=Metaspace Manager
com.sun.management:type=DiagnosticCommand
java.lang:type=GarbageCollector,name=PS MarkSweep
com.sun.management:type=HotSpotDiagnostic
[arthas@12844]$
```

Par défaut, en passant en paramètre de la commande mbean le nom d'un MBean, toutes les propriétés du MBean sont affichées avec leur valeur.

Résultat :

```
 [arthas@12844]$ mbean java.lang:type=ClassLoading
 NAME                   VALUE
----------------------------------------------------
 Verbose                false
 LoadedClassCount       3969
 UnloadedClassCount     10
 TotalLoadedClassCount  3979
 ObjectName             java.lang:type=ClassLoading
```

Il est possible d'utiliser le caractère * comme joker dans le nom du MBean.

Résultat :

```
[arthas@12844]$ mbean java.lang:type=Class*
 NAME                   VALUE
----------------------------------------------------
 Verbose                false
 LoadedClassCount       3994
 UnloadedClassCount     10
 TotalLoadedClassCount  4004
 ObjectName             java.lang:type=ClassLoading

[arthas@12844]$
```

Il est aussi possible d'utiliser le caractère * comme joker dans le nom de l'attribut.

Résultat :

```
[arthas@12844]$ mbean java.lang:type=ClassLoading *Count
 NAME                   VALUE
------------------------------
 LoadedClassCount       3970
 UnloadedClassCount     10
 TotalLoadedClassCount  3980

```

L'option -E permet d'utiliser une expression régulière.

Résultat :

```
[arthas@12844]$ mbean -E java.lang:type=ClassLoading LoadedClassCount|TotalLoadedClas
sCount
 NAME                   VALUE
------------------------------
 LoadedClassCount       3979
 TotalLoadedClassCount  3989

[arthas@12844]$
```

Il est possible de surveiller un MBean en précisant un intervalle de rafraichissement.

Résultat :

```
[arthas@12844]$ mbean -i 500 java.lang:type=ClassLoading *Count
 NAME                   VALUE
------------------------------
 LoadedClassCount       3970
 UnloadedClassCount     10
 TotalLoadedClassCount  3980

NAME                   VALUE
------------------------------
 LoadedClassCount       3970
 UnloadedClassCount     10
 TotalLoadedClassCount  3980

```

Il faut utiliser la combinaison de touche Ctrl+C pour arrêter la commande.

120.3.3.9. La commande logger

La commande logger permet d'afficher des informations sur les loggers et de modifier leur niveau de gravité.

La syntaxe est de la forme :

logger [-c <value>] [-h] [--include-arthas-logger] [--include-no-appender] [-l <value>] [-n <value>]

La commande logger possède plusieurs paramètres optionnels :

Paramètre	Rôle
-c, --classloader <value>	Préciser le hashcode du ClassLoader concerné. La valeur par défaut est celle du SystemClassLoader
--include-arthas-logger	Tenir compte des loggers d'Arthas. La valeur par défaut est false
--include-no-appender	Tenir compte des loggers qui n'ont pas d'appender. La valeur par défaut est false
-l, --level <value>	Modifier le niveau de gravité du logger
-n, --name <value>	Préciser le nom du logger concerné

Sans option, la commande logger affiche tous les loggers avec leur appenders associés.

Résultat :

```
[arthas@4736]$ logger
 name              ROOT
 class             ch.qos.logback.classic.Logger
 classLoader       sun.misc.Launcher$AppClassLoader@73d16e93
 classLoaderHash   73d16e93
 level             WARN
 effectiveLevel    WARN
 additivity        true
 codeSource        file:/C:/Users/jm/.m2/repository/ch/qos/logback/logback-classic/1.2
                   .3/logback-classic-1.2.
                   3.jar
 appenders         name            STDOUT
                   class           ch.qos.logback.core.ConsoleAppender
                   classLoader     sun.misc.Launcher$AppClassLoader@73d16e93
                   classLoaderHash 73d16e93
                   target          System.out
                   name            APPLICATION
                   class           ch.qos.logback.core.rolling.RollingFileAppender
                   classLoader     sun.misc.Launcher$AppClassLoader@73d16e93
                   classLoaderHash 73d16e93
                   file            app.log
                   name            ASYNC
                   class           ch.qos.logback.classic.AsyncAppender
                   classLoader     sun.misc.Launcher$AppClassLoader@73d16e93
                   classLoaderHash 73d16e93
                   blocking        true
                   appenderRef     [APPLICATION]

 name              fr.jmdoudoux.dej
 class             ch.qos.logback.classic.Logger
 classLoader       sun.misc.Launcher$AppClassLoader@73d16e93
 classLoaderHash   73d16e93
 level             INFO
 effectiveLevel    INFO
 additivity        false
 codeSource        file:/C:/Users/jm/.m2/repository/ch/qos/logback/logback-classic/1.2
                   .3/logback-classic-1.2.3.jar
 appenders         name            STDOUT
                   class           ch.qos.logback.core.ConsoleAppender
                   classLoader     sun.misc.Launcher$AppClassLoader@73d16e93
                   classLoaderHash 73d16e93
                   target          System.out

[arthas@4736]$
```

L'option -n permet de préciser le nom d'un logger particulier.

Résultat :

```
[arthas@4736]$ logger -n fr.jmdoudoux
 name              fr.jmdoudoux
 class             ch.qos.logback.classic.Logger
 classLoader       sun.misc.Launcher$AppClassLoader@73d16e93
 classLoaderHash   73d16e93
 level             null
 effectiveLevel    WARN
 additivity        true
 codeSource        file:/C:/Users/jm/.m2/repository/ch/qos/logback/logback-classic/1.
                   2.3/logback-classic-1.2.3.jar

[arthas@4736]$ logger -n fr.jmdoudoux.dej
 name              fr.jmdoudoux.dej
 class             ch.qos.logback.classic.Logger
 classLoader       sun.misc.Launcher$AppClassLoader@73d16e93
 classLoaderHash   73d16e93
 level             INFO
 effectiveLevel    INFO
 additivity        false
 codeSource        file:/C:/Users/jm/.m2/repository/ch/qos/logback/logback-classic/1.
                   2.3/logback-classic-1.2.3.jar
 appenders         name            STDOUT
                   class           ch.qos.logback.core.ConsoleAppender
                   classLoader     sun.misc.Launcher$AppClassLoader@73d16e93
                   classLoaderHash 73d16e93
                   target          System.out

[arthas@4736]$
```

L'option -l permet de modifier le niveau de gravité du logger en précisant la nouvelle valeur en paramètre.

Résultat :

```
[arthas@4736]$ logger -n fr.jmdoudoux.dej -l DEBUG
update logger level success.
[arthas@4736]$ logger -n fr.jmdoudoux.dej | grep level
 level             DEBUG
[arthas@4736]$
```

120.3.4. Les fonctionnalités concernant les classes/classloaders

Arthas propose plusieurs fonctionnalités concernant les classes ou les classloaders :

Commande	Rôle
sc	Chercher des classes parmi celles chargées dans la JVM
sm	Chercher des méthodes dans les classes chargées
jad	Décompiler le bytecode d'une class chargée
mc	Compiler en mémoire un fichier .java pour générer le bytecode dans la JVM
redefine	Charger un fichier .class et remplacer celui existant dans la JVM
dump	Extraire le bytecode chargé dans un fichier
classloader	Afficher des informations relatives aux ClassLoaders et interagir avec eux
getstatic	Afficher la valeur d'une propriété statique
ognl	Exécuter une expression au format ognl

120.3.4.1. La commande getstatic

La commande getstatic permet d'afficher la valeur d'un champ static.

La syntaxe est :

getstatic [-c <value>] [-x <value>] [-h] [-E] class-pattern field-pattern [express]

La commande getstatic possède plusieurs paramètres :

Paramètre	Rôle
-c, --classloader <value>	Préciser le hashcode du classloader concerné
-E, --regex	Activer l'utilisation d'une expression régulière. Par défaut, seul le caractère * est utilisable
<class-pattern>	Motif pour le nom de la classe. Le séparateur peut être '.' ou '/'
<field-pattern>	Motif pour le nom du champ
<express>	Préciser l'élément à afficher en utilisant la syntaxe OGNL

La façon la plus simple d'utiliser la commande getstatic est de lui passer en paramètre le nom pleinement qualifié de la classe et le nom du champ static concerné.

Résultat :

```
[arthas@6672]$ getstatic fr.jmdoudoux.dej.Main valeur
field: valeur
@Integer[1234]
Affect(row-cnt:1) cost in 12 ms.
[arthas@6672]$
```

120.3.4.2. La commande dump

La commande dump permet d'extraire le bytecode d'une classe chargée dans la JVM dans un fichier .class.

La syntaxe de la commande dump est de la forme :

dump [-c <value>] [-d <value>] [-h] [-l <value>] [-E <value>] class-pattern

La command dump possède plusieurs options :

Paramètre	Rôle
class-pattern	Motif pour le nom de la classe
[c:]	Préciser le hashcode du classloader utilisé pour charger la classe
[E]	Activer l'utilisation d'expression régulière

Résultat :

```
[arthas@4192]$ dump fr.jmdoudoux.dej.Compteur
 HASHCODE  CLASSLOADER                                    LOCATION
 55f96302  +-sun.misc.Launcher$AppClassLoader@55f96302    C:\Users\jm\logs\arthas\cla
                                                          ssdump\sun.misc.Launcher
             +-sun.misc.Launcher$ExtClassLoader@54e948e9  $AppClassLoader-55f96302\fr
                                                          \test\Compteur.class
Affect(row-cnt:1) cost in 16 ms.
[arthas@4192]$
```

120.3.4.3. La commande sc

La commande sc (search class) permet de rechercher des classes chargées dans la JVM.

La syntaxe est de la forme :

sc [-c <value>] [-d] [-x <value>] [-f] [-h] [-E] class-pattern

La commande sc possède plusieurs paramètres :

Paramètre	Rôle
class-pattern	Motif du nom pleinement qualifié de classe. Il est possible d'utiliser le séparateur '.' ou '/' ce qui permet de faire un copier/coller du nom d'une classe depuis la stacktrace d'une exception
-c, --classloader <value>	Préciser le hashcode du ClassLoader qui a chargé la classe
-d, --details	Afficher des informations sur la classe
-f, --field	Afficher les champs de la classe. Doit être utilisée avec l'option -d
-h, --help	Afficher l'aide de la commande
-E, --regex	Activer l'utilisation d'une expression régulière dans le motif du nom de classe (par défaut uniquement le caractère *)

Il est possible d'utiliser le caractère * comme joker.

Résultat :

```
[arthas@9840]$ sc fr.jmdoudoux.dej.*
fr.jmdoudoux.dej.Compteur
fr.jmdoudoux.dej.Main
fr.jmdoudoux.dej.Main$1
fr.jmdoudoux.dej.Service
fr.jmdoudoux.dej.ThreadService
Affect(row-cnt:5) cost in 3 ms.
[arthas@9840]$
```

Il est donc aussi possible de demander la liste de toutes les classes chargées dans la JVM : attention la liste peut être importante.

Résultat :

```
[arthas@9840]$ sc * | grep fr.test
fr.jmdoudoux.dej.Compteur
fr.jmdoudoux.dej.Main
fr.jmdoudoux.dej.Main$1
fr.jmdoudoux.dej.Service
fr.jmdoudoux.dej.ThreadService
[arthas@9840]$
```

L'option -d permet d'afficher des informations sur une classe

Résultat :

```
[arthas@9840]$ sc -d fr.jmdoudoux.dej.Compteur
 class-info        fr.jmdoudoux.dej.Compteur
 code-source       /C:/java/arthas-demo.jar
 name              fr.jmdoudoux.dej.Compteur
 isInterface       false
 isAnnotation      false
 isEnum            false
 isAnonymousClass  false
 isArray           false
 isLocalClass      false
 isMemberClass     false
 isPrimitive       false
 isSynthetic       false
 simple-name       Compteur
 modifier          public
 annotation
 interfaces
 super-class       +-java.lang.Object
 class-loader      +-sun.misc.Launcher$AppClassLoader@55f96302
                     +-sun.misc.Launcher$ExtClassLoader@49487aa8
 classLoaderHash   55f96302

Affect(row-cnt:1) cost in 8 ms.
```

L'option -f combinée avec l'option -d permet d'afficher dans le détail des informations sur la classe et ses champs.

Résultat :

```
[arthas@9840]$ sc -df fr.jmdoudoux.dej.Compteur
 class-info        fr.jmdoudoux.dej.Compteur
 code-source       /C:/java/workspace-2018-12-Test/Arthas-demo/arthas-demo.jar
 name              fr.jmdoudoux.dej.Compteur
 isInterface       false
 isAnnotation      false
 isEnum            false
 isAnonymousClass  false
 isArray           false
 isLocalClass      false
 isMemberClass     false
 isPrimitive       false
 isSynthetic       false
 simple-name       Compteur
 modifier          public
 annotation
 interfaces
 super-class       +-java.lang.Object
 class-loader      +-sun.misc.Launcher$AppClassLoader@55f96302
                     +-sun.misc.Launcher$ExtClassLoader@49487aa8
 classLoaderHash   55f96302
 fields            name     valeur
                   type     java.util.concurrent.atomic.AtomicInteger
                   modifier private,static
                   value    601


Affect(row-cnt:1) cost in 6 ms.
```

120.3.4.4. La commande sm

La commande sm (search method) permet de rechercher les méthodes d'une classe chargée dans la JVM. Seules les méthodes déclarées dans la classe sont trouvées : les méthodes héritées ne sont pas affichées.

La syntaxe est de la forme :

sm [-c <value>] [-d] [-h] [-E] class-pattern [method-pattern]

La commande sm possède plusieurs paramètres :

Paramètre	Rôle
class-pattern	Motif du nom pleinement qualifié de classe. Il est possible d'utiliser le séparateur '.' ou '/' ce qui permet de faire un copier/coller du nom d'une classe depuis la stacktrace d'une exception
method-pattern	Motif du nom de la méthode
-c, --classloader <value>	Préciser le hashcode du ClassLoader qui a chargé la classe
-d, --details	Affiche le détail de la méthode
-E, --regex	Activer l'utilisation d'une expression régulière dans le motif du nom de classe (par défaut uniquement le caractère *)

En passant uniquement le motif du nom de classe, la commande sm affiche les méthodes de la classe.

Résultat :

```
[arthas@4192]$ sm fr.jmdoudoux.dej.Compteur
fr.jmdoudoux.dej.Compteur <init>()V
fr.jmdoudoux.dej.Compteur incrementer()I
Affect(row-cnt:2) cost in 2 ms.
[arthas@4192]$
```

L'option -d permet d'afficher des détails sur la méthode fournie en paramètre.

Résultat :

```
[arthas@4192]$ sm -d fr.jmdoudoux.dej.Compteur incrementer
 declaring-class  fr.jmdoudoux.dej.Compteur
 method-name      incrementer
 modifier         public
 annotation
 parameters
 return           int
 exceptions
 classLoaderHash  55f96302

Affect(row-cnt:1) cost in 3 ms.
[arthas@4192]$
```

120.3.4.5. La commande classloader

La commande classloader permet d'interagir avec les classloaders utilisés dans la JVM

La syntaxe est de la forme :

classloader [-a] [-c <value>] [-h] [-i] [-l] [--load <value>] [-r <value>] [-t]

La commande classloader possède plusieurs paramètres :

Paramètre	Rôle
-a, --all	Afficher les classes chargées
-c, --classloader <value>	Préciser le hashcode du ClassLoader concerné
-i, --include-reflection-classloader	Inclure aussi sun.reflect.DelegatingClassLoader
-l, --list-classloader	Afficher pour chaque ClassLoader des informations (nombre de classes chargées, valeur de hachage, ClassLoader parent)
--load <value>	Charger une classe par le ClassLoader précisé avec l'option -c
-r, --resource <value>	Rechercher une ressource en utilisant le ClassLoader précisé avec l'option -c
-t, --tree	Afficher la hiérarchie des ClassLoaders

Sans option, la commande classloader affiche pour chaque classloader le nombre d'instances et le nombre de classes chargées.

Résultat :

```
[arthas@9840]$ classloader
 name                                     numberOfInstances  loadedCountTotal
 BootstrapClassLoader                     1                  1591
 com.taobao.arthas.agent.ArthasClassloader  1                  1107
 sun.misc.Launcher$AppClassLoader         1                  8
 sun.reflect.DelegatingClassLoader        6                  6
 sun.misc.Launcher$ExtClassLoader         1                  5
```

```
Affect(row-cnt:5) cost in 2 ms.
[arthas@9840]$
```

L'option -l affiche des informations sur chaque ClassLoader.

Résultat :

```
[arthas@9840]$ classloader -l

 name                            loadedCount  hash      parent
 BootstrapClassLoader            1591         null      null
 com.taobao.arthas.agent.Arthas  1108         2822a77f  sun.misc.Launcher$ExtClassLoa
 Classloader@2822a77f                                   der@49487aa8
 sun.misc.Launcher$AppClassLoad  8            55f96302  sun.misc.Launcher$ExtClassLoa
 er@55f96302                                            der@49487aa8
 sun.misc.Launcher$ExtClassLoad  5            49487aa8  null
 er@49487aa8
Affect(row-cnt:4) cost in 5 ms.
[arthas@9840]$
```

L'option -t affiche la hiérarchie des classloaders.

Résultat :

```
[arthas@9840]$ classloader -t
+-BootstrapClassLoader
+-sun.misc.Launcher$ExtClassLoader@49487aa8
  +-com.taobao.arthas.agent.ArthasClassloader@2822a77f
  +-sun.misc.Launcher$AppClassLoader@55f96302
Affect(row-cnt:4) cost in 2 ms.
[arthas@9840]$
```

L'option -c affiche les URL de l'URLClassLoader

Résultat :

```
[arthas@9840]$ classloader -c 55f96302
file:/C:/java/workspace-2018-12-Test/Arthas-demo/arthas-demo.jar
file:/C:/Users/jm/.arthas/lib/3.1.7/arthas/arthas-agent.jar

Affect(row-cnt:8) cost in 4 ms.
[arthas@9840]$
```

L'option -r permet de rechercher une ressource par le ClassLoader.

Résultat :

```
[arthas@9840]$ classloader -c 55f96302 -r maconfig.properties
 jar:file:/C:/java/workspace/Arthas-demo/arthas-demo.jar!/maconfig.properties

Affect(row-cnt:1) cost in 2 ms.
[arthas@9840]$
```

L'option --load permet de charger une classe par le Classloader

Résultat :

```
[arthas@9840]$ classloader -c 55f96302 --load fr.jmdoudoux.dej.AutreClasse
load class success.
 class-info        fr.jmdoudoux.dej.AutreClasse
 code-source       /C:/java/workspace/Arthas-demo/arthas-demo.jar
 name              fr.jmdoudoux.dej.AutreClasse
 isInterface       false
 isAnnotation      false
 isEnum            false
 isAnonymousClass  false
 isArray           false
 isLocalClass      false
 isMemberClass     false
 isPrimitive       false
 isSynthetic       false
 simple-name       AutreClasse
 modifier          public
 annotation
 interfaces
 super-class       +-java.lang.Object
 class-loader      +-sun.misc.Launcher$AppClassLoader@55f96302
                     +-sun.misc.Launcher$ExtClassLoader@1ac18008
 classLoaderHash   55f96302
[arthas@9840]$
```

L'option -a permet d'afficher les classes chargées. Attention, le nombre de classes chargées peut être important. Il est possible de ne demander que les classes chargées par un Classloader particulier en utilisant en complément l'option -c.

Résultat :

```
[arthas@9840]$ classloader -c 55f96302 -a
 hash:1442407170, sun.misc.Launcher$AppClassLoader@55f96302
 com.taobao.arthas.agent.AgentBootstrap
 com.taobao.arthas.agent.AgentBootstrap$1
 com.taobao.arthas.agent.ArthasClassloader
 fr.jmdoudoux.dej.AbstractCompteur
 fr.jmdoudoux.dej.AutreClasse
 fr.jmdoudoux.dej.Compteur
 fr.jmdoudoux.dej.Main
 fr.jmdoudoux.dej.Main$1
 fr.jmdoudoux.dej.Service
 fr.jmdoudoux.dej.ThreadService

Affect(row-cnt:0) cost in 12 ms.

[arthas@9840]$
```

120.3.4.6. La commande jad

La commande jad permet de décompiler une classe. Le code décompilé est mis en évidence par une coloration syntaxique pour une meilleure lisibilité dans la console.

La décompilation peut être intéressante et utile pour vérifier que c'est la bonne version d'une classe qui est utilisée ou pour pouvoir la modifier avant de la recompiler et de la recharger à chaud.

La syntaxe de la commande est de la forme :

jad [-c <value>] [-h] [-E] [--source-only] class-pattern [method-name]

La commande jad possède plusieurs paramètres :

Paramètre	Rôle
-c <value>	Préciser le hashcode du ClassLoader qui a chargé la classe
-E, --regex	Activer l'utilisation d'une expression régulière dans le motif du nom de classe (par défaut uniquement le caractère *)
--source-only	Obtenir uniquement le code source décompilé
<class-pattern>	Motif du nom pleinement qualifié de classe. Il est possible d'utiliser le séparateur '.' ou '/'
<method-name>	Motif de la méthode de la classe à uniquement décompilée

Pour décompiler une classe, il suffit de préciser son nom pleinement qualifié comme paramètre.

Résultat :

```
[arthas@9840]$ jad fr.jmdoudoux.dej.Compteur

ClassLoader:
+-sun.misc.Launcher$AppClassLoader@55f96302
  +-sun.misc.Launcher$ExtClassLoader@49487aa8

Location:
/C:/java/workspace/Arthas-demo/arthas-demo.jar

/*
 * Decompiled with CFR.
 */
package fr.test;

import java.util.concurrent.atomic.AtomicInteger;

public class Compteur {
    private static AtomicInteger valeur = new AtomicInteger(0);

    public int incrementer() {
        return valeur.incrementAndGet();
    }
}

Affect(row-cnt:1) cost in 753 ms.
[arthas@9840]$
```

Par défaut, le commande affiche des informations sur le classloader et l'artefact qui contient la classe. Pour n'afficher que le code source issue de la décompilation, il faut utiliser l'option --source-only.

Résultat :

```
[arthas@9840]$ jad --source-only fr.jmdoudoux.dej.Compteur
/*
 * Decompiled with CFR.
 */
package fr.test;

import java.util.concurrent.atomic.AtomicInteger;

public class Compteur {
    private static AtomicInteger valeur = new AtomicInteger(0);

    public int incrementer() {
        return valeur.incrementAndGet();
    }
}

[arthas@9840]$
```

Il est possible de rediriger la sortie de la commande dans un fichier.

Résultat :

```
[arthas@9840]$ jad --source-only fr.jmdoudoux.dej.Compteur > c:/temp/Compteur.java
[arthas@9840]$
```

Il est possible de ne demander que de décompiler une méthode d'une classe en les précisant en paramètres.

Résultat :

```
[arthas@9840]$ jad fr.jmdoudoux.dej.Compteur incrementer

ClassLoader:
+-sun.misc.Launcher$AppClassLoader@55f96302
  +-sun.misc.Launcher$ExtClassLoader@1ac18008

Location:
/C:/java/workspace-2018-12-Test/Arthas-demo/arthas-demo.jar

public int incrementer() {
    return valeur.incrementAndGet();
}

Affect(row-cnt:1) cost in 107 ms.
[arthas@9840]$
```

Si la classe à décompiler est chargée plusieurs fois, il faut préciser le ClassLoader concerné avec l'option -c.

120.3.4.7. La commande mc

La commande mc utilise un compilateur en mémoire pour compiler des fichiers code source Java en fichiers .class contenant du bytecode.

La syntaxe de la commande est de la forme :

mc [-c <value>] [-d <value>] [--encoding <value>] [-h] sourcefiles...

La commande mc possède plusieurs paramètres :

Paramètre	Rôle
-c, --classloader <value>	Préciser le hashcode du ClassLoader a utilisé
-d, --directory <value>	Préciser le répertoire dans lequel le résultat de la compilation va être stocké
--encoding <value>	Le jeu de caractères utilisé pour l'encodage du code source
<sourcefiles>	Le ou les fichiers source

L'option -d permet de préciser le répertoire qui va contenir le résultat de la compilation.

Résultat :

```
[arthas@9840]$ cat c:/temp/Compteur.java
/*
 * Decompiled with CFR.
 */
package fr.test;

import java.util.concurrent.atomic.AtomicInteger;

public class Compteur {
    private static AtomicInteger valeur = new AtomicInteger(0);

    public int incrementer() {
        System.out.println("incrementer");
        return valeur.incrementAndGet();
    }
}

[arthas@9840]$ mc -d c:/temp c:/temp/Compteur.java
Memory compiler output:
c:\temp\fr\test\Compteur.class
Affect(row-cnt:1) cost in 62 ms.
[arthas@9840]$
```

120.3.4.8. La commande redefine

La commande redefine permet de recharger à chaud une classe.

La syntaxe de la commande est de la forme :

redefine [-c <value>] [-h] classfilePaths...

La commande redefine possède plusieurs paramètres dont :

Paramètre	Rôle
-c, --classloader <value>	Préciser le hashcode du ClassLoader à utiliser

Avant de pouvoir recharger une classe, il faut trouver le hashcode de son ClassLoader pour le réutiliser pour le rechargement.

Résultat :

```
[arthas@9840]$ sc -d fr.jmdoudoux.dej.Compteur
 class-info        fr.jmdoudoux.dej.Compteur
 code-source       /C:/java/workspace-2018-12-Test/Arthas-demo/arthas-demo.jar
 name              fr.jmdoudoux.dej.Compteur
 isInterface       false
 isAnnotation      false
 isEnum            false
 isAnonymousClass  false
 isArray           false
 isLocalClass      false
 isMemberClass     false
 isPrimitive       false
 isSynthetic       false
 simple-name       Compteur
 modifier          public
 annotation
 interfaces
 super-class       +-java.lang.Object
 class-loader      +-sun.misc.Launcher$AppClassLoader@55f96302
                   +-sun.misc.Launcher$ExtClassLoader@49487aa8
 classLoaderHash   55f96302

Affect(row-cnt:1) cost in 11 ms.

[arthas@9840]$ redefine -c 55f96302 c:/temp/fr/test/Compteur.class
redefine success, size: 1
[arthas@9840]$
```

La commande redefine utilise la méthode redefineClasses() de la classe Instrumentation : elle suit donc les restrictions induites par l'utilisation de cette méthode. Il n'est par exemple pas possible d'ajouter, supprimer ou renommer un membre de la classe, de changer la signature d'une méthode ni de changer l'héritage de la classe.

Une fois la classe rechargée, elle ne peut pas être restaurée même en après l'invocation de la commande reset : la seule solution est de recharger la version initiale préalablement sauvegardée.

La commande redefine entre en conflit avec les commandes jad, monitor, trace, tt et watch. Après l'exécution de la commande redefine, si vous exécutez une des commandes précédentes, le bytecode de la classe sera réinitialisé.

120.3.4.9. La commande ognl

Le commande ognl permet d'évaluer une expression OGNL (Object Graph Navigation Language) et d'afficher le résultat de cette évaluation.

La syntaxe de la commande est de la forme :

ognl [-c <value>] [-x <value>] [-h] express

La commande ognl possède plusieurs paramètres :

Paramètre	Rôle
-c, --classloader <value>	Préciser le hashcode du ClassLoader à utiliser
-x, --expand <value>	Niveau de profondeur lors du parcours des objets (la valeur par défaut est 1)
Express	Expression ognl à évaluer

Le guide de la syntaxe des expressions est consultable à l'url : https://commons.apache.org/proper/commons-ognl/language-guide.html

Par exemple, l'expression peut demander la valeur d'une propriété statique :

Résultat :

```
[arthas@11140]$  ognl '@fr.jmdoudoux.dej.Compteur@valeur'
@AtomicInteger[
    serialVersionUID=@Long[6214790243416807050],
    unsafe=@Unsafe[sun.misc.Unsafe@1b417cbd],
    valueOffset=@Long[12],
    value=@Integer[18047],
    serialVersionUID=@Long[-8742448824652078965],
]
[arthas@11140]$
```

L'expression peut invoquer une méthode statique

Résultat :

```
 [arthas@11140]$ ognl '@fr.jmdoudoux.dej.Service@fabriquer("test",5)'
@AutreClasse[
    dateCrea=@LocalDateTime[2020-03-08T22:02:19.518],
    nom=@String[test],
    taille=@Integer[5],
]
[arthas@11140]$
```

L'expression peut contenir plusieurs valeurs.

Résultat :

```
[arthas@11140]$ ognl '#valeur1=@System@getProperty("user.home"), #valeur2=
@System@getProperty("java.version"),  {#valeur1, #valeur2}'
@ArrayList[
    @String[C:\Users\jm],
    @String[1.8.0_202],
]
[arthas@11140]$
```

120.3.5. Les fonctionnalités de profiling

Arthas propose plusieurs commandes relatives à la JVM :

Commande	Rôle
monitor	Surveiller les invocations de méthodes pour afficher des statistiques sur leurs exécutions
watch	Afficher des informations sur l'invocation de méthodes : les paramètres, la valeur de retour et les exceptions levées
trace	Tracer le temps d'exécution des méthodes invoquées en premier niveau lors de l'invocation des méthodes suivies
tt	Enregistrer les invocations de méthodes pour permettre leur analyse après la fin de l'enregistrement
stack	Afficher périodiquement la stacktrace d'invocation de méthodes
profiler	Utiliser async-profiler pour obtenir des informations de profiling

Plusieurs commandes (monitor, watch, trace, tt, ...) instrumentent le code en ajoutant du bytecode. Lorsque cette instrumentation n'est plus utile, il est important d'invoquer la commande stop ou reset pour retirer les aspects surtout si la JVM n'est pas arrêtée.

120.3.5.1. La commande monitor

La commande monitor permet de surveiller les invocations de méthodes pour afficher des statistiques sur leurs exécutions.

La syntaxe de la commande est de la forme

monitor [-c <value>] [-h] [-n <value>] [-E <value>] class-pattern method-pattern

La commande monitor possède plusieurs paramètres dont :

Options	Rôle
<class-pattern>	Motif pour désigner la ou les classes concernées
<method-pattern>	Motif pour désigner la ou les méthodes concernées
-c ou -cycle	Intervalle d'affichage des informations. La valeur par défaut est 60 secondes
-n ou -limits	Limite d'exécutions
-E ou -regex	Expression régulière

Il faut indiquer le motif de classe et de méthode : par exemple pour une méthode en particulier, il faut préciser le nom pleinement qualifié de la classe et le nom de la méthode.

Résultat :

```
[arthas@9840]$ monitor fr.jmdoudoux.dej.Service traiter
Press Q or Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 24 ms.
 timestamp            class           method  total success fail avg-rt(ms) fail-rate
-------------------------------------------------------------------------------------
 2020-02-15 23:16:22  fr.jmdoudoux.dej.Service traiter 56    56    0    2075,60    0,00%
[arthas@9840]$
```

Les informations affichées sont :

- total : le nombre d'invocations
- success : le nombre d'invocations exécutées normalement (sans lever d'exception)
- fail : le nombre d'invocations qui ont levé une exception
- avg-rt : le temps d'exécution moyen
- fail-rate : le pourcentage d'échecs durant la période

Si les motifs fournis ne permettent pas de correspondre à au moins une méthode, alors Arthas affiche un message

Résultat :

```
[arthas@9840]$ monitor fr.jmdoudoux.dej.Service afficher
No class or method is affected, try:
1. sm CLASS_NAME METHOD_NAME to make sure the method you are tracing actually exists
(it might be in your parent class).
2. reset CLASS_NAME and try again, your method body might be too large.
3. check arthas log: C:\Users\jm\logs\arthas\arthas.log
4. visit https://github.com/alibaba/arthas/issues/47 for more details.
[arthas@9840]$
```

Il est aussi possible d'utiliser le caractère joker * dans le motif pour désigner plusieurs classes et/ou méthodes.

Résultat :

```
[arthas@9840]$ monitor fr.jmdoudoux.dej.* *
Press Q or Ctrl+C to abort.
Affect(class-cnt:7 , method-cnt:13) cost in 94 ms.
 timestamp           class            method      total success fail avg-rt(ms) fail-rate
-----------------------------------------------------------------------------------------
 2020-02-15 23:10:27 fr.jmdoudoux.dej.Compteur incrementer 60    60    0    0,02       0,00%
 2020-02-15 23:10:27 fr.jmdoudoux.dej.Service  traiter     58    58    0    1886,19    0,00%

 timestamp           class            method      total success fail avg-rt(ms) fail-rate
-----------------------------------------------------------------------------------------
 2020-02-15 23:11:27 fr.jmdoudoux.dej.Compteur incrementer 67    67    0    0,02       0,00%
 2020-02-15 23:11:27 fr.jmdoudoux.dej.Service  traiter     67    67    0    1774,72    0,00%
[arthas@9840]$
```

L'option -c permet de préciser le nombre de secondes à attendre avant que la commande n'affiche des statistiques.

Résultat :

```
[arthas@9840]$ monitor fr.jmdoudoux.dej.Service traiter -c 5
Press Q or Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 48 ms.
 timestamp            class            method   total  success  fail  avg-rt(ms)  fail-rate
--------------------------------------------------------------------------------------------
 2020-01-19 21:32:01  fr.jmdoudoux.dej.Service  traiter  2      2     0     2041,27     0,00%

 timestamp            class            method   total  success  fail  avg-rt(ms)  fail-rate
--------------------------------------------------------------------------------------------
 2020-01-19 21:32:06  fr.jmdoudoux.dej.Service  traiter  0      0     0     0,00        0,00%

 timestamp            class            method   total  success  fail  avg-rt(ms)  fail-rate
--------------------------------------------------------------------------------------------
 2020-01-19 21:32:11  fr.jmdoudoux.dej.Service  traiter  2      2     0     1539,91     0,00%

 timestamp            class            method   total  success  fail  avg-rt(ms)  fail-rate
--------------------------------------------------------------------------------------------
 2020-01-19 21:32:16  fr.jmdoudoux.dej.Service  traiter  1      1     0     2042,88     0,00%

 timestamp            class            method   total  success  fail  avg-rt(ms)  fail-rate
--------------------------------------------------------------------------------------------
 2020-01-19 21:32:21  fr.jmdoudoux.dej.Service  traiter  1      1     0     4037,81     0,00%

 timestamp            class            method   total  success  fail  avg-rt(ms)  fail-rate
--------------------------------------------------------------------------------------------
 2020-01-19 21:32:26  fr.jmdoudoux.dej.Service  traiter  3      3     0     3044,34     0,00%
```

```
[arthas@9840]$
```

La commande monitor ne rend pas la main immédiatement. Pour arrêter l'exécution de la méthode, il faut utiliser la combinaison de touche Ctrl+C.

L'option -n permet de préciser le nombre d'itérations à atteindre avant d'arrêter l'exécution de la commande.

Résultat :

```
[arthas@9840]$ monitor fr.jmdoudoux.dej.Service * -c 5 -n 3
Press Q or Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:2) cost in 27 ms.
 timestamp            class             method   total  success  fail  avg-rt(ms)  fail-rate
-------------------------------------------------------------------------------------------
 2020-02-15 23:25:37  fr.jmdoudoux.dej.Service  traiter  1      1     0     2054,73     0,00%

 timestamp            class             method   total  success  fail  avg-rt(ms)  fail-rate
-------------------------------------------------------------------------------------------
 2020-02-15 23:25:42  fr.jmdoudoux.dej.Service  traiter  4      4     0     2542,40     0,00%

 timestamp            class             method   total  success  fail  avg-rt(ms)  fail-rate
-------------------------------------------------------------------------------------------
 2020-02-15 23:25:47  fr.jmdoudoux.dej.Service  traiter  6      6     0     1545,81     0,00%

Command execution times exceed limit: 3, so command will exit. You can set it with -n option.
[arthas@9840]$
```

Du côté du serveur d'Arthas, la commande s'exécute en arrière-plan, mais le code tissé n'a plus d'effet une fois la commande terminée.

120.3.5.2. La commande stack

La commande stack permet d'afficher périodiquement la stacktrace d'invocation de méthodes. Cela permet de savoir qui appelle la ou les méthodes sélectionnées.

La syntaxe de la commande stack est de la forme :

stack [-h] [-n <value>] [-E] class-pattern [method-pattern] [condition-express]

La commande stack possède plusieurs paramètres dont :

Options	Rôle
<class-pattern>	Motif pour désigner la ou les classes concernées
<method-pattern>	Motif pour désigner la ou les méthodes concernées
<condition-express>	Expression conditionnelle au format ognl
-n, --limits <value>	Limite d'exécution
-E, --regex	Expression régulière

Il faut au moins préciser la ou les classes concernées : dans ce cas toutes leurs méthodes sont prises en comptes.

Résultat :

```
[arthas@9840]$ stack fr.jmdoudoux.dej.Compteur
Press Q or Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:2) cost in 80 ms.
ts=2020-01-19 23:50:13;thread_name=Service_1;id=a;is_daemon=false;priority=5;TCCL=sun
    .misc.Launcher$AppClassLoader@55f96302
    @fr.jmdoudoux.dej.Service.traiter()
        at fr.jmdoudoux.dej.ThreadService.run(ThreadService.java:18)

ts=2020-01-19 23:50:18;thread_name=Service_1;id=a;is_daemon=false;priority=5;TCCL=sun
    .misc.Launcher$AppClassLoader@55f96302
    @fr.jmdoudoux.dej.Service.traiter()
        at fr.jmdoudoux.dej.ThreadService.run(ThreadService.java:18)

 [arthas@9840]$
```

Il est possible de préciser uniquement une seule méthode en paramètre pour restreindre les informations à cette dernière.

Résultat :

```
[arthas@9840]$ stack fr.jmdoudoux.dej.Compteur incrementer
Press Q or Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 20 ms.
ts=2020-01-19 23:52:15;thread_name=Service_1;id=a;is_daemon=false;priority=5;TCCL=sun
    .misc.Launcher$AppClassLoader@55f96302
    @fr.jmdoudoux.dej.Service.traiter()
        at fr.jmdoudoux.dej.ThreadService.run(ThreadService.java:18)

ts=2020-01-19 23:52:20;thread_name=Service_1;id=a;is_daemon=false;priority=5;TCCL=sun
    .misc.Launcher$AppClassLoader@55f96302
    @fr.jmdoudoux.dej.Service.traiter()
        at fr.jmdoudoux.dej.ThreadService.run(ThreadService.java:18)

ts=2020-01-19 23:52:20;thread_name=Service_2;id=b;is_daemon=false;priority=5;TCCL=sun
    .misc.Launcher$AppClassLoader@55f96302
    @fr.jmdoudoux.dej.Service.traiter()
        at fr.jmdoudoux.dej.ThreadService.run(ThreadService.java:18)

[arthas@9840]$
```

La commande s'exécute tant qu'elle n'est pas interrompue avec la combinaison de touches Ctrl + C.

L'option -n permet de préciser un nombre maximum d'invocations avant que la commande s'interrompe.

Résultat :

```
[arthas@9840]$ stack fr.jmdoudoux.dej.Compteur incrementer -n 3
Press Q or Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 12 ms.
ts=2020-02-20 20:12:53;thread_name=Service_2;id=b;is_daemon=false;priority=5;TCCL=sun
    .misc.Launcher$AppClassLoader@55f96302
    @fr.jmdoudoux.dej.Service.traiter()
        at fr.jmdoudoux.dej.ThreadService.run(ThreadService.java:18)

ts=2020-02-20 20:12:55;thread_name=Service_1;id=a;is_daemon=false;priority=5;TCCL=sun
    .misc.Launcher$AppClassLoader@55f96302
    @fr.jmdoudoux.dej.Service.traiter()
        at fr.jmdoudoux.dej.ThreadService.run(ThreadService.java:18)

ts=2020-02-20 20:12:56;thread_name=Service_2;id=b;is_daemon=false;priority=5;TCCL=sun
    .misc.Launcher$AppClassLoader@55f96302
    @fr.jmdoudoux.dej.Service.traiter()
        at fr.jmdoudoux.dej.ThreadService.run(ThreadService.java:18)

Command execution times exceed limit: 3, so command will exit. You can set it with -n
 option.
[arthas@9840]$
```

Le nombre d'invocations d'une méthode peut être important : il est possible d'utiliser une expression ognl pour conditionner les informations affichées.

Résultat :

```
[arthas@9840]$ stack fr.jmdoudoux.dej.* * '#cost>100' -n 2
Press Q or Ctrl+C to abort.
Affect(class-cnt:7 , method-cnt:13) cost in 98 ms.
ts=2020-02-16 22:43:25;thread_name=Service_2;id=b;is_daemon=false;priority=5;TCCL=sun
    .misc.Launcher$AppClassLoader@55f96302
    @fr.jmdoudoux.dej.Service.traiter()
        at fr.jmdoudoux.dej.ThreadService.run(ThreadService.java:-1)

ts=2020-02-16 22:43:26;thread_name=Service_2;id=b;is_daemon=false;priority=5;TCCL=sun
    .misc.Launcher$AppClassLoader@55f96302
    @fr.jmdoudoux.dej.Service.traiter()
        at fr.jmdoudoux.dej.ThreadService.run(ThreadService.java:-1)

Command execution times exceed limit: 2, so command will exit. You can set it with -n
 option.
[arthas@9840]$
```

120.3.5.3. La commande trace

La commande trace permet de tracer le temps d'exécution des méthode invoquées en premier niveau lors de l'invocation des méthodes suivies. Cela permet de trouver les goulets d'étranglements lors de l'invocation d'une méthode.

La syntaxe de la commande est de la forme :

trace [-h] [-n <value>] [-p <value>] [-E] [--skipJDKMethod <value>] class-pattern method-pattern [condition-express]

La commande trace possède plusieurs paramètres dont :

Options	Rôle
<class-pattern>	Motif pour désigner la ou les classes concernées
<method-pattern>	Motif pour désigner la ou les méthodes concernées
<condition-express>	Expression conditionnelle au format ognl
-n, --limits <value>	Limite d'exécution de la commande
-E, --regex	Expression régulière
-p, --path <value>	
--skipJDKMethod <value>	Ignorer les méthodes de classes du JDK. Valeur par défaut : true

Le plus simple est de préciser le motif de la classe et de la méthode concernée

Résultat :

```
[arthas@9840]$ trace fr.jmdoudoux.dej.Service traiter
Press Q or Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 22 ms.
`---ts=2020-01-19 23:56:54;thread_name=Service_2;id=b;is_daemon=false;priority=5;TCCL=sun
    .misc.Launcher$AppClassLoader@55f96302
    `---[4042.1098ms] fr.jmdoudoux.dej.Service:traiter()
        `---[0.7561ms] fr.jmdoudoux.dej.Compteur:incrementer() #11

`---ts=2020-01-19 23:56:57;thread_name=Service_1;id=a;is_daemon=false;priority=5;TCCL=sun
    .misc.Launcher$AppClassLoader@55f96302
    `---[2041.5307ms] fr.jmdoudoux.dej.Service:traiter()
        `---[0.5413ms] fr.jmdoudoux.dej.Compteur:incrementer() #11

`---ts=2020-01-19 23:57:00;thread_name=Service_1;id=a;is_daemon=false;priority=5;TCCL=sun
    .misc.Launcher$AppClassLoader@55f96302
    `---[41.7881ms] fr.jmdoudoux.dej.Service:traiter()
        `---[0.6608ms] fr.jmdoudoux.dej.Compteur:incrementer() #11
```

La première de chaque invocation affiche l'heure, des informations sur le thread et le ClassLoader.

Pour chaque méthode invoquée en premier niveau durant une exécution plusieurs informations sont affichées : le temps d'exécution entre crochets et des informations statistiques si le nombre d'invocations est supérieur à 1.

La commande s'exécute tant qu'elle n'est pas interrompue avec la combinaison de touches Ctrl + C.

L'option -n permet de préciser le nombre d'invocations à afficher avant d'arrêter l'exécution de la commande.

Résultat :

```
[arthas@9840]$ trace fr.jmdoudoux.dej.Service traiter -n 1
Press Q or Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 17 ms.
`---ts=2020-02-20 20:59:28;thread_name=Service_2;id=b;is_daemon=false;priority=5;TCCL
    =sun.misc.Launcher$AppClassLoader@55f96302
    `---[103.0994ms] fr.jmdoudoux.dej.Service:traiter()
        `---[0.0633ms] fr.jmdoudoux.dej.Compteur:incrementer() #11

Command execution times exceed limit: 1, so command will exit. You can set it with -n option.
[arthas@9840]$
```

Il est possible de préciser une expression ognl pour limiter les invocations restituées. Par exemple, dans l'exemple ci-dessous, seules les invocations qui durent plus de 2 secondes sont affichées.

Résultat :

```
[arthas@9840]$ trace fr.jmdoudoux.dej.Service traiter '#cost > 2000'
Press Q or Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 31 ms.
`---ts=2020-01-20 00:05:14;thread_name=Service_1;id=a;is_daemon=false;priority=5;TCCL
     =sun.misc.Launcher$AppClassLoader@55f96302
    `---[4048.5473ms] fr.jmdoudoux.dej.Service:traiter()
        `---[0.723ms] fr.jmdoudoux.dej.Compteur:incrementer() #11

`---ts=2020-01-20
00:05:30;thread_name=Service_2;id=b;is_daemon=false;priority=5;TCCL=sun.misc.Launcher
    $AppClassLoader@55f96302
    `---[3037.0302ms] fr.jmdoudoux.dej.Service:traiter()
        `---[0.9308ms] fr.jmdoudoux.dej.Compteur:incrementer() #11

`---ts=2020-01-20 00:05:33;thread_name=Service_1;id=a;is_daemon=false;priority=5;TCCL
    =sun.misc.Launcher$AppClassLoader@55f96302
    `---[2035.4533ms] fr.jmdoudoux.dej.Service:traiter()
        `---[0.9851ms] fr.jmdoudoux.dej.Compteur:incrementer() #11

[arthas@9840]$
```

L'option --skipJDKMethod permet de prendre en compte ou non les méthodes de classes du JDK.

Résultat :

```
[arthas@9840]$ trace --skipJDKMethod false fr.jmdoudoux.dej.Service traiter
Press Q or Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 13 ms.
`---ts=2020-02-20 20:55:17;thread_name=Service_2;id=b;is_daemon=false;priority=5;TCCL
    =sun.misc.Launcher$AppClassLoader@55f96302
    `---[293.1969ms] fr.jmdoudoux.dej.Service:traiter()
        +---[0.1066ms] fr.jmdoudoux.dej.Compteur:incrementer() #11
        +---[0.01ms] java.lang.Math:random() #12
        +---[0.0093ms] java.util.concurrent.TimeUnit:sleep() #12
        `---[min=1.0E-4ms,max=4.0424ms,total=139.8867ms,count=100000] java.util.UUID:
        randomUUID() #15

[arthas@9840]$
```

Il est possible de tracer des méthodes de classes différentes en utilisant une expression régulière en valeur de l'option -E. C'est notamment pratique pour tracer des invocations de niveaux inférieurs.

Résultat :

```
[arthas@9840]$ trace -E fr.jmdoudoux.dej.Service|fr.jmdoudoux.dej.Compteur traiter|incrementer
Press Q or Ctrl+C to abort.
Affect(class-cnt:2 , method-cnt:2) cost in 50 ms.
`---ts=2020-02-20 21:07:57;thread_name=Service_1;id=a;is_daemon=false;priority=5;TCCL
    =sun.misc.Launcher$AppClassLoader@55f96302
    `---[1104.748ms] fr.jmdoudoux.dej.Service:traiter()
        `---[0.1391ms] fr.jmdoudoux.dej.Compteur:incrementer() #11
            `---[0.0934ms] fr.jmdoudoux.dej.Compteur:incrementer()
[arthas@9840]$
```

120.3.5.4. La commande watch

La commande watch permet d'afficher des informations sur l'exécution de méthodes : les paramètres, la valeur de retour et les exceptions levées lors d'invocations de méthodes.

La syntaxe de la commande est de la forme :

watch [-b] [-e] [-x <value>] [-f] [-h] [-n <value>] [-E] [-M <value>] [-s] class-pattern method-pattern express [condition-express]

La commande watch possède plusieurs paramètres dont :

Options	Rôle
<class-pattern>	Motif pour désigner la ou les classes concernées. Il est possible d'utiliser le caractère joker *
<method-pattern>	Motif pour désigner la ou les méthodes concernées. Il est possible d'utiliser le caractère joker *
<express>	Les informations à surveiller, en utilisant le format ognl
<condition-express>	Expression conditionnelle au format ognl
-n, --limits <value>	Limite du nombre d'observations restitué par la commande
-E, --regex	Activation de l'utilisation d'expressions régulières
-b, --before	Regarder avant l'invocation
-e, --exception	Regarder après la levée d'une exception
-f, --finish	Regarder après l'invocation, activé par défaut
-M, --sizeLimit <value>	Taille maximale en octets du résultat affiché. Valeur par défaut : 10 * 1024 * 1024
-s, --success	Regarder après une invocation réussie
-x <value>	Niveau de profondeur lors du parcours des objets (la valeur par défaut est 1)

Il faut obligatoirement fournir au moins trois paramètres : le motif de la classe, le motif de la méthode et la ou les informations à surveiller. Dans l'exemple ci-dessous, c'est la valeur de retour de la méthode incrementer() de la classe fr.jmdoudoux.dej.Compteur qui est affichée à chaque invocation.

Résultat :

```
[arthas@9840]$ watch fr.jmdoudoux.dej.Compteur incrementer returnObj
Press Q or Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 13 ms.
ts=2020-02-20 21:39:07; [cost=0.125ms] result=@Integer[14950]
ts=2020-02-20 21:39:08; [cost=0.0683ms] result=@Integer[14951]
ts=2020-02-20 21:39:10; [cost=0.0515ms] result=@Integer[14952]
```

```
[arthas@9840]$
```

Les éléments utilisables dans l'expression des éléments à observer sont notamment :

Variable	Rôle
params	Les paramètres
params[n]	Le paramètre n
returnObj	La valeur de retour
throwExp	L'exception levée
target	L'objet courant (this) sur laquelle la méthode est invoquée
"{params, returnObj}"	Les paramètres et la valeur de retour
"{params[0], returnObj}"	Le premier paramètre et la valeur de retour
clazz	Le nom pleinement qualifié de la classe
method	La signature de la méthode

Les valeurs qui ne peuvent pas être obtenue (par exemple, target sur une méthode statique ou returnObj alors que l'observation est faite avant l'invocation) sont null.

La commande s'exécute tant qu'elle n'est pas interrompue avec la combinaison de touches Ctrl + C.

L'option -n permet le nombre d'invocations à afficher avant d'arrêter l'exécution de la commande.

Résultat :

```
[arthas@9840]$ watch fr.jmdoudoux.dej.Compteur incrementer returnObj -n 1
Press Q or Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 16 ms.
ts=2020-02-20 21:41:31; [cost=0.0633ms] result=@Integer[14997]
Command execution times exceed limit: 1, so command will exit. You can set it with -n option.
[arthas@9840]$
```

4 points de vue peuvent être utilisés lors de l'observation d'une méthode, chacune exprimée avec une option dédiée.

L'option -b demande l'observation avant l'invocation. Dans ce point de vue, la valeur de retour et l'exception levée sont toujours null

Résultat :

```
[arthas@9840]$ watch fr.jmdoudoux.dej.Service fabriquer '{params, returnObj}' -b -n 1
Press Q or Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 19 ms.
ts=2020-02-27 23:47:27; [cost=0.0379ms] result=@ArrayList[
    @Object[][isEmpty=false;size=2],
    null,
]
Command execution times exceed limit: 1, so command will exit. You can set it with -n option.
[arthas@9840]$
```

L'option -f demande l'observation après l'invocation : elle est activée par défaut.

L'option -s demande l'observation après une invocation réussie.

L'option -e demande l'observation après la levée d'une exception : seules les exécutions qui lèvent une exception sont observées :

Résultat :

```
[arthas@9840]$ watch fr.jmdoudoux.dej.Service fabriquer '{params, throwexp}' -e -n 1
 -x 3
Press Q or Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 38 ms.
ts=2020-02-25 01:04:42; [cost=0.1891ms] result=@ArrayList[
    @Object[][
        @String[nom4],
        @Integer[40],
    ],
    java.lang.IllegalArgumentException: nom invalide
        at fr.jmdoudoux.dej.Service.fabriquer(Service.java:29)
        at fr.jmdoudoux.dej.Service.traiter(Service.java:20)
        at fr.jmdoudoux.dej.ThreadService.run(ThreadService.java:24)
,
]
Command execution times exceed limit: 1, so command will exit. You can set it with -n option.
[arthas@9840]$
```

Il est possible de demander plusieurs points de vue pour une même invocation observée. Dans l'exemple ci-dessous, l'observation est faite avant et après l'exécution de la méthode.

Résultat :

```
[arthas@9840]$ watch fr.jmdoudoux.dej.Service fabriquer '{params, returnObj}' -b -s -n 2
Press Q or Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 19 ms.
ts=2020-02-27 23:51:48; [cost=0.0313ms] result=@ArrayList[
    @Object[][isEmpty=false;size=2],
    null,
]
ts=2020-02-27 23:51:48; [cost=5.645432150981E8ms] result=@ArrayList[
    @Object[][isEmpty=false;size=2],
    @AutreClasse[fr.jmdoudoux.dej.AutreClasse@11d6648a],
]
Command execution times exceed limit: 2, so command will exit. You can set it with -n option.
[arthas@9840]$
```

Il est possible d'observer l'instance sur laquelle la méthode est invoquée

Résultat :

```
[arthas@9840]$ watch fr.jmdoudoux.dej.AutreClasse afficher 'target' -n 2
Press Q or Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 51 ms.
ts=2020-02-26 22:22:24; [cost=4.6029ms] result=@AutreClasse[
    dateCrea=@LocalDateTime[2020-02-26T22:22:24.304],
    nom=@String[nom1],
    taille=@Integer[10],
]
ts=2020-02-26 22:22:25; [cost=0.2278ms] result=@AutreClasse[
    dateCrea=@LocalDateTime[2020-02-26T22:22:25.794],
    nom=@String[nom3],
    taille=@Integer[30],
]
Command execution times exceed limit: 2, so command will exit. You can set it with -n option.
[arthas@9840]$
```

Il est possible d'observer un champ de l'instance sur laquelle la méthode est invoquée

Résultat :

```
[arthas@9840]$ watch fr.jmdoudoux.dej.AutreClasse afficher 'target.nom' -n 2
Press Q or Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 16 ms.
ts=2020-02-26 22:24:22; [cost=0.3027ms] result=@String[nom3]
```

```
ts=2020-02-26 22:24:24; [cost=0.2338ms] result=@String[nom2]
Command execution times exceed limit: 2, so command will exit. You can set it with -n option.
[arthas@9840]$
```

L'option -x permet de préciser le niveau de profondeur de parcours et de restitution des objets observés.

Résultat :

```
[arthas@9840]$ watch fr.jmdoudoux.dej.Service fabriquer '{params, returnObj}' -x 1 -n 1
Press Q or Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 22 ms.
ts=2020-02-25 00:47:34; [cost=0.2728ms] result=@ArrayList[
    @Object[][isEmpty=false;size=2],
    @AutreClasse[fr.jmdoudoux.dej.AutreClasse@1d4537e5],
]
Command execution times exceed limit: 1, so command will exit. You can set it with -n option.
[arthas@21344]$ watch fr.jmdoudoux.dej.Service fabriquer '{params, returnObj}' -x 3 -n 1
Press Q or Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 29 ms.
ts=2020-02-25 00:47:44; [cost=0.2408ms] result=@ArrayList[
    @Object[][
        @String[nom2],
        @Integer[20],
    ],
    @AutreClasse[
        dateCrea=@LocalDateTime[
            MIN=@LocalDateTime[-999999999-01-01T00:00],
            MAX=@LocalDateTime[+999999999-12-31T23:59:59.999999999],
            serialVersionUID=@Long[6207766400415563566],
            date=@LocalDate[2020-02-25],
            time=@LocalTime[00:47:44.604],
        ],
        nom=@String[nom2],
        taille=@Integer[20],
    ],
]
Command execution times exceed limit: 1, so command will exit. You can set it with -n option.
[arthas@9840]$
```

Il est possible de filtrer les invocations observées en passant la condition en quatrième paramètre.

Résultat :

```
[arthas@9840]$ watch fr.jmdoudoux.dej.Service fabriquer '{params, returnObj}' 'params[0]=="no
m3"' -n 1 -x 3
Press Q or Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 25 ms.
ts=2020-02-25 00:54:05; [cost=0.2447ms] result=@ArrayList[
    @Object[][
        @String[nom3],
        @Integer[30],
    ],
    @AutreClasse[
        dateCrea=@LocalDateTime[
            MIN=@LocalDateTime[-999999999-01-01T00:00],
            MAX=@LocalDateTime[+999999999-12-31T23:59:59.999999999],
            serialVersionUID=@Long[6207766400415563566],
            date=@LocalDate[2020-02-25],
            time=@LocalTime[00:54:05.288],
        ],
        nom=@String[nom3],
        taille=@Integer[30],
    ],
]
Command execution times exceed limit: 1, so command will exit. You can set it with -n option.
[arthas@9840]$
```

Dans l'exemple ci-dessous, le temps d'exécution des méthodes affichées doit être supérieur à 500ms.

Résultat :

```
[arthas@9840]$ watch fr.jmdoudoux.dej.Service fabriquer '{params, returnObj}' '#cost>500' -n 2
Press Q or Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 55 ms.
[arthas@20904]$ watch fr.jmdoudoux.dej.Service traiter '{params, returnObj}' '#cost>500' -n 2
Press Q or Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 16 ms.
ts=2020-02-27 23:00:30; [cost=1047.055101ms] result=@ArrayList[
    @Object[][isEmpty=true;size=0],
    @Integer[1374],
]
ts=2020-02-27 23:00:31; [cost=1049.916499ms] result=@ArrayList[
    @Object[][isEmpty=true;size=0],
    @Integer[1375],
]
Command execution times exceed limit: 2, so command will exit. You can set it with -n option.
[arthas@9840]$
```

120.3.5.5. La commande tt

La commande tt (Time Tunnel) permet d'enregistrer les invocations de méthodes pour permettre leur analyse après la fin de l'enregistrement. Elle est particulièrement utile pour enregistrer de nombreuses invocations pour permettre leur analyse ultérieure.

Chaque invocation enregistrée est appelée un time fragment qui possède un identifiant.

La syntaxe de la commande est de la forme :

tt [-d] [--delete-all] [-x <value>] [-h] [-i <value>] [-n <value>] [-l] [-p] [-E] [--replay-interval <value>] [--replay-times <value>] [-s <value>] [-M <value>] [-t] [-w <value>] [class-pattern] [method-pattern] [condition-express]

Plusieurs options sont utilisables dont :

Options	Rôle
<class-pattern>	Motif pour désigner la ou les classes concernées
<method-pattern>	Motif pour désigner la ou les méthodes concernées
<condition-express>	Expression conditionnelle au format ognl
-n, --limits <value>	Limite d'exécution de la commande
-E, --regex	Expression régulière
-d, --delete	Supprimer le time fragment dont l'identifiant est fourni
--delete-all	Supprimer tous les time fragments
-x, --expand <value>	Niveau de profondeur lors du parcours des objets (la valeur par défaut est 1)
-i, --index <value>	Afficher des informations sur le time fragment dont l'identifiant est fourni
-n, --limits <value>	Limiter le nombre d'exécutions enregistrées à celui fourni
-l, --list	Lister tous les time fragments
-p, --play	Réexécuter le time fragment dont l'identifiant est fourni
-s, --search-express <value>	Rechercher des time fragments en utilisant le format ognl
-M, --sizeLimit <value>	Taille maximale en octets du résultat affiché. Valeur par défaut : 10 * 1024 * 1024
-t, --time-tunnel	Enregistrer les invocations de méthodes dans des time fragments
-w, --watch-express <value>	Définir les informations affichées d'un time fragment en utilisant le format ognl

L'option -t permet de demander un enregistrement des invocations de méthodes dans un laps de temps.

Résultat :

```
[arthas@9840]$ tt -t fr.jmdoudoux.dej.Service traiter
Press Q or Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 59 ms.
 INDEX  TIMESTAMP        COST(m  IS-RE  IS-EX  OBJECT     CLASS            METHOD
                         s)      T      P
-------------------------------------------------------------------------------------
 1000   2020-02-21 21:5  2063.1  true   false  NULL       Service          traiter
        9:06             742
 1001   2020-02-21 21:5  4039.0  true   false  NULL       Service          traiter
        9:08             635
 1002   2020-02-21 21:5  2038.6  true   false  NULL       Service          traiter
        9:16             877
 1003   2020-02-21 21:5  2037.9  true   false  NULL       Service          traiter
        9:16             356
 1004   2020-02-21 21:5  3040.0  true   false  NULL       Service          traiter
        9:20             753
[arthas@9840]$
```

Les informations affichées sont :

Colonne	Rôle
INDEX	Identifiant du fragment. Chaque invocation a son propre identifiant.
TIMESTAMP	Date/heure de la capture du fragment
COST(ms)	Temps d'exécution de la méthode
IS-RET	Est-ce que la méthode s'est correctement exécutée
IS-EXP	Est-ce que la méthode a levé une exception
OBJECT	Valeur retournée par la méthode hashCode() de l'objet sur lequel la méthode est invoquée. NULL si la méthode est statique
CLASS	La classe de la méthode exécutée
METHOD	La méthode exécutée

Il est possible d'interrompre la commande en utilisant la combinaison de touches Ctrl + C.

L'option -n de limiter le nombre de time fragments enregistrés durant l'exécution de la commande.

Résultat :

```
[arthas@9840]$ tt -t -n 3 fr.jmdoudoux.dej.Service traiter
Press Q or Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 19 ms.
 INDEX  TIMESTAMP        COST(m  IS-RE  IS-EX  OBJECT     CLASS           METHOD
                         s)      T      P
-------------------------------------------------------------------------------------
 1000   2020-02-28 00:1  1048.5  true   false  NULL       Service         traiter
        9:18             577
 1001   2020-02-28 00:1  1053.8  true   false  NULL       Service         traiter
        9:19             442
 1002   2020-02-28 00:1  2039.6  true   false  NULL       Service         traiter
        9:21             496
Command execution times exceed limit: 3, so command will exit. You can set it with -n
 option.
[arthas@9840]$
```

Le nombre d'invocations capturées peut être important : il est possible de préciser en paramètre une condition de capture des invocations.

Résultat :

```
[arthas@9840]$ tt -t -n 2 fr.jmdoudoux.dej.Service traiter '#cost>2000'
Press Q or Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 16 ms.
 INDEX  TIMESTAMP        COST(m  IS-RE  IS-EX  OBJECT     CLASS           METHOD
                         s)      T      P
-------------------------------------------------------------------------------------
 1004   2020-02-28 00:2  2038.7  true   false  NULL       Service         traiter
        4:08             198
 1005   2020-02-28 00:2  4034.7  true   false  NULL       Service         traiter
        4:11             803
Command execution times exceed limit: 2, so command will exit. You can set it with -n
 option.
[arthas@9840]$
```

Un fois l'enregistrement terminé, il est possible d'obtenir la liste des time fragments en utilisant l'option -l.

Résultat :

```
[arthas@9840]$ tt -l
 INDEX  TIMESTAMP        COST(m  IS-RE  IS-EX  OBJECT     CLASS           METHOD
                         s)      T      P
-------------------------------------------------------------------------------------
 1000   2020-02-21 21:5  2063.1  true   false  NULL       Service         traiter
        9:06             742
 1001   2020-02-21 21:5  4039.0  true   false  NULL       Service         traiter
        9:08             635
 1002   2020-02-21 21:5  2038.6  true   false  NULL       Service         traiter
        9:16             877
 1003   2020-02-21 21:5  2037.9  true   false  NULL       Service         traiter
        9:16             356
 1004   2020-02-21 21:5  3040.0  true   false  NULL       Service         traiter
        9:20             753
Affect(row-cnt:5) cost in 5 ms.
[arthas@9840]$
```

Il est possible d'obtenir des informations sur le contexte d'invocation d'un fragment particulier en utilisant l'option -i pour fournir l'identifiant du fragment

Résultat :

```
[arthas@9840]$ tt -i 1002
 INDEX         1002
 GMT-CREATE    2020-02-21 13:59:16
 COST(ms)      2038.6877
 OBJECT        NULL
 CLASS         fr.jmdoudoux.dej.Service
 METHOD        traiter
 IS-RETURN     true
 IS-EXCEPTION  false
 RETURN-OBJ    @Integer[125]
Affect(row-cnt:1) cost in 2 ms.
[arthas@9840]$
```

Il est possible d'obtenir des informations choisies sur un fragment particulier en utilisant l'option -i pour fournir l'identifiant du fragment et l'option -w pour préciser les informations à obtenir (de manière similaire à celle de la commande watch)

Résultat :

```
[arthas@9840]$ tt -i 1002 -w '{params,returnObj}'
@ArrayList[
    @Object[][isEmpty=true;size=0],
    @Integer[125],
]
Affect(row-cnt:1) cost in 2 ms.
```

```
[arthas@9840]$
```

L'option -s permet de chercher les invocations enregistrées d'une méthode particulière

Résultat :

```
[arthas@9840]$ tt -s 'method.name=="traiter"'
 INDEX  TIMESTAMP        COST(m  IS-RE  IS-EX  OBJECT     CLASS           METHOD
                         s)      T      P
-------------------------------------------------------------------------------------
 1000   2020-02-21 21:5  2063.1  true   false  NULL       Service         traiter
        9:06             742
 1001   2020-02-21 21:5  4039.0  true   false  NULL       Service         traiter
        9:08             635
 1002   2020-02-21 21:5  2038.6  true   false  NULL       Service         traiter
        9:16             877
 1003   2020-02-21 21:5  2037.9  true   false  NULL       Service         traiter
        9:16             356
 1004   2020-02-21 21:5  3040.0  true   false  NULL       Service         traiter
        9:20             753
Affect(row-cnt:5) cost in 32 ms.
[arthas@9840]$
```

Arthas conserve le contexte de chaque invocation : il est possible de réexécuter une exécution en utilisant les options -i pour préciser le fragment concerné via son identifiant et l'option -p.

Résultat :

```
[arthas@9840]$ tt -i 1002 -p
 RE-INDEX      1002
 GMT-REPLAY    2020-02-21 15:20:40
 OBJECT        NULL
 CLASS         fr.jmdoudoux.dej.Service
 METHOD        traiter
 IS-RETURN     true
 IS-EXCEPTION  false
 COST(ms)      4037.3472
 RETURN-OBJ    @Integer[1609]
Time fragment[1002] successfully replayed 1 times.
[arthas@9840]$
```

Attention : la réexécution d'une méthode se fait dans un autre thread, ce qui implique un accès à un autre ThreadLocal si la méthode en a besoin.

120.3.5.6. La commande profiler

La commande profiler utilise async-profiler pour obtenir des informations de profiling et les restituer dans différents formats dont le frame graph.

La syntaxe de la commande est de la forme :

profiler [--allkernel] [--alluser] [-e <value>] [-f <value>] [--format <value>] [-h] [-i <value>] [--threads] action [actionArg]

Plusieurs options sont utilisables dont :

Options	Rôle
--allkernel	Prendre en compte uniquement les événements de type kernel-mode
--alluser	Prendre en compte uniquement les événements de type user-mode

-e, --event <value>	Préciser le type d'événements à profiler : (cpu, alloc, lock, cache-misses etc.), la valeur par défaut est cpu
-f, --file <value>	Préciser le chemin du fichier résultat
--format <value>	Préciser le format du fichier résultat (svg, html, jfr), la valeur par est svg
-i, --interval <value>	Préciser l'intervalle d'échantillonnage en ns (par défaut : 10000000 soit 10 ms)
--threads	Profiler les différents threads séparément
<action>	Préciser l'action à exécuter. Les valeurs possibles sont : resume, dumpCollapsed, getSamples, start, list, execute, version, stop, load, dumpFlat, actions, dumpTraces, status

L'action actions permet d'obtenir la liste des actions utilisables dans la commande.

Résultat :

```
[arthas@2884]$ profiler actions
Supported Actions: [resume, dumpCollapsed, getSamples, start, list, execute, version,
 stop, load, dumpFlat, actions, dumpTraces, status]
[arthas@2884]$ profiler version
1.7[arthas@2884]$
```

La commande profiler n'est pas utilisable sur toutes les plateformes : c'est notamment le cas sous Windows.

L'action list permet d'afficher tous les événements qui peuvent être mesurés.

Résultat :

```
[arthas@14128]$ profiler list
Current OS do not support AsyncProfiler, Only support Linux/Mac.
```

Sur les systèmes supportés l'option list permet d'afficher la liste de profilers utilisables. Par exemple, sous Linux :

Résultat :

```
[arthas@829]$ profiler list
Basic events:
  cpu
  alloc
  lock
  wall
  itimer
Perf events:
  page-faults
  context-switches
  cycles
  instructions
  cache-references
  cache-misses
  branches
  branch-misses
  bus-cycles
  L1-dcache-load-misses
  LLC-load-misses
  dTLB-load-misses
  mem:breakpoint
  trace:tracepoint
[arthas@829]$
```

Le plus simple pour démarrer le profiling est d'utiliser l'action start.

Résultat :

```
[arthas@2884]$ profiler start --event cpu
Started [cpu] profiling
```

Selon la configuration du système, la commande peut échouer. Par exemple, sous Linux :

Résultat :

```
[arthas@2884]$ profiler start
Perf events unavailable. See stderr of the target process.
```

Il faut consulter la console de l'application connectée.

Résultat :

```
WARNING: Kernel symbols are
unavailable due to restrictions. Try

echo 0 > /proc/sys/kernel/kptr_restrict

echo 1 > /proc/sys/kernel/perf_event_paranoid
perf_event_open failed: Permission
non accordée
```

Il faut alors suivre les instructions proposées dans la console de l'application connectée.

Résultat :

```
jm@VMUbuntu:~$ sudo sh
# echo 0 >
/proc/sys/kernel/kptr_restrict
# echo 1 >
/proc/sys/kernel/perf_event_paranoid
# exit
jm@VMUbuntu:~$
```

Une fois la configuration effectuée, il est possible de lancer le profiling.

Par défaut, le profiling se fait sur les événements de type cpu.

Résultat :

```
[arthas@2884]$ profiler start
Started [cpu] profiling
```

L'action status permet de connaître le type d'événements capturés et le temps d'exécution du profiling.

Résultat :

```
[arthas@2884]$ profiler status
[perf] profiling is running for 29
seconds
```

L'action getSamples permet de connaître le nombre de mesures effectuées.

Résultat :

```
[arthas@2884]$ profiler getSamples
243
```

L'action stop permet d'arrêter le profiling et de générer un fichier de résultat. Par défaut c'est un flame graph, il est au format SVG, stocké dans le sous-répertoire arthas-output et le chemin du fichier est affiché.

Résultat :

```
[arthas@2884]$ profiler stop
profiler output file:
/home/jm/arthas-output/20200423-131511.svg
OK
[arthas@2884]$
```

Il est possible de fournir le nom du fichier avec l'action stop en utilisant l'option -file

Résultat :

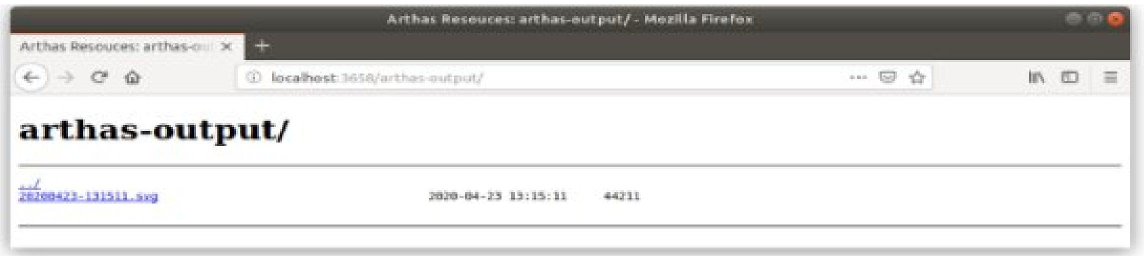
```
[arthas@2884]$ profiler stop --file
/tmp/mon_app_profilling.svg
```

Le format du fichier généré est par défaut SVG, mais il est possible d'utiliser d'autres formats comme HTML en utilisant l'option --format.

Résultat :

```
[arthas@2884]$ profiler stop
--format html
```

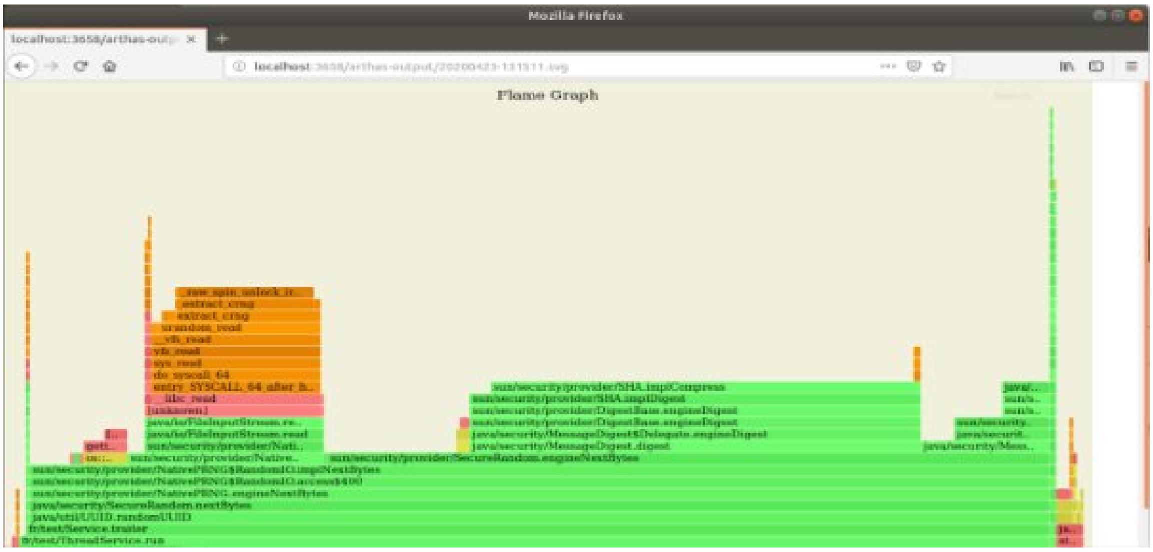
Il est possible de visualiser le résultat dans un navigateur. Par défaut, Arthas utilise le port 3658 et il suffit d'ouvrir l'URL http://localhost:3658/arthas-output/


Arthas Resouces: arthas-output/ - Mozilla Firefox
Arthas Resouces: arthas-ou
localhost:3658/arthas-output/

arthas-output/

```
../
20200423-131511.svg                    2020-04-23 13:15:11      44211
```

En cliquant sur le résultat désiré, les informations sont affichées.

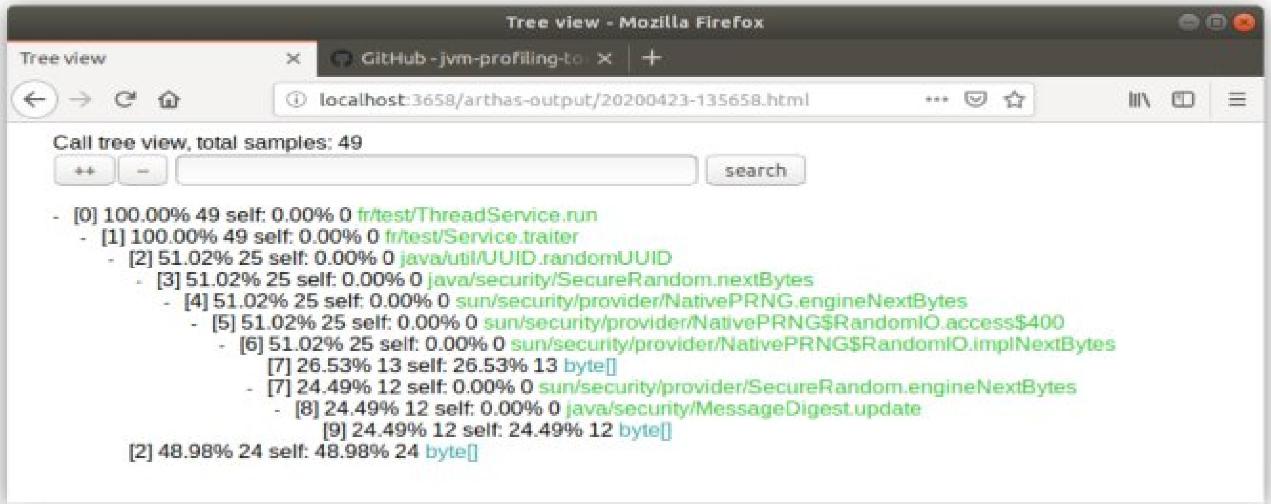


Il est possible de profiler d'autres événements et obtenir le résultat sous un autre format.

Résultat :

```
[arthas@2884]$ profiler start
--event alloc
Started [alloc] profiling
[arthas@2884]$ profiler stop
--format html
profiler output file:
/home/jm/arthas-output/20200423-135658.html
OK
```

L'action resume permet de démarrer un profiling en concervant les données capturées lors de la précédente commande stop.


Tree view - Mozilla Firefox
Tree view
GitHub - jvm-profiling-to
localhost:3658/arthas-output/20200423-135658.html
++
−
search

Call tree view, total samples: 49

```
- [0] 100.00% 49 self: 0.00% 0 fr/test/ThreadService.run
  - [1] 100.00% 49 self: 0.00% 0 fr/test/Service.traiter
    - [2] 51.02% 25 self: 0.00% 0 java/util/UUID.randomUUID
      - [3] 51.02% 25 self: 0.00% 0 java/security/SecureRandom.nextBytes
        - [4] 51.02% 25 self: 0.00% 0 sun/security/provider/NativePRNG.engineNextBytes
          - [5] 51.02% 25 self: 0.00% 0 sun/security/provider/NativePRNG$RandomIO.access$400
            - [6] 51.02% 25 self: 0.00% 0 sun/security/provider/NativePRNG$RandomIO.implNextBytes
                [7] 26.53% 13 self: 26.53% 13 byte[]
              - [7] 24.49% 12 self: 0.00% 0 sun/security/provider/SecureRandom.engineNextBytes
                - [8] 24.49% 12 self: 0.00% 0 java/security/MessageDigest.update
                    [9] 24.49% 12 self: 24.49% 12 byte[]
      [2] 48.98% 24 self: 48.98% 24 byte[]
```

L'action version affiche la version d'async-profiler utilisée.

120.4. L'exécution de commandes en asynchrone

Par défaut, toutes les commandes exécutées dans le CLI Arthas sont exécutées de manière synchrone : durant leur exécution, il n'est pas possible d'interagir avec le CLI pour par exemple exécuter une autre commande.

L'exécution de commandes en asynchrone peut être utile et pratique lorsqu'un problème est difficilement reproductible dans l'environnement.

Arthas permet l'exécution de commandes en arrière-plan (background) pour une exécution asynchrone. Le mode de fonctionnement de ce type d'exécution est inspiré de ce que propose Linux :

Fonctionnalité	Rôle
&	Exécution de la commande en arrière-plan
>	Redirection de la sortie d'une commande avec écrasement
>>	Redirection de la sortie d'une commande
jobs	Afficher la liste des commandes exécutées en arrière-plan
kill	Forcer l'arrêt d'une tâche en arrière-plan
fg	Basculer une tâche en arrière-plan pour la remettre en premier plan
bg	Demander l'exécution d'une commande en arrière-plan

Les tâches en arrière-plan poursuivent leur exécution même si la session est déconnectée.

Attention : l'exécution de nombreuses tâches en arrière-plan peut avoir un impact sur les performances de la JVM sur laquelle Arthas est connecté.

120.4.1. L'exécution de commandes en arrière-plan

Pour exécuter une commande en arrière-plan, il suffit de terminer la commande par un caractère &.

Résultat :

```
[arthas@20904]$ tt -t -n 100 fr.jmdoudoux.dej.Service traiter '#cost>2000' &
[arthas@20904]$ Affect(class-cnt:1 , method-cnt:1) cost in 15 ms.
 INDEX  TIMESTAMP        COST(m  IS-RE  IS-EX  OBJECT    CLASS          METHOD
                         s)      T      P
-------------------------------------------------------------------------------------
 1007   2020-02-28 00:3  2040.5  true   false  NULL      Service        traiter
        0:27             601
 1008   2020-02-28 00:3  3080.2  true   false  NULL      Service        traiter
        0:30             453
```

Une fois une commande exécutée en arrière-plan, il est possible d'exécuter d'autres commandes.

Il est préférable de rediriger la sortie d'une commande exécutée en arrière-plan pour éviter que celle-ci ne viennent perturber l'affichage dans la console.

120.4.2. La redirection de la sortie

Arthas permet de rediriger la sortie d'une commande en utilisant les opérateurs > ou >>. Ils peuvent être combinés avec l'opérateur & pour éviter de polluer la console avec les sorties d'une commande exécutée en arrière-plan.

Résultat :

```
[arthas@7656]$ tt -t fr.jmdoudoux.dej.Service traiter >> &
job id  : 11
cache location  : C:\Users\jm\logs\arthas-cache\7656\11
[arthas@7656]$
```

Si aucun chemin n'est précisé, Arthas redirige la sortie dans le cache local précisé avec la propriété « cache location ». Pour garantir l'unicité, le chemin contient le PID de la JVM et l'identifiant du job.

Il est possible de préciser le chemin d'un fichier qui va contenir la redirection : ce fichier est écrit sur la machine locale où est lancé la console Arthas.

Résultat :

```
[arthas@20904]$ tt -t fr.jmdoudoux.dej.Service traiter > log_tt.txt &
[arthas@20904]$
```

Pour enrichir le contenu du fichier existant, il faut un double caractère >.

Résultat :

```
[arthas@20904]$ tt -t fr.jmdoudoux.dej.Service traiter >> log_tt.txt &
[arthas@20904]$
```

Les commandes exécutées en arrière-plan poursuivent leur exécution dans la JVM même si la session est arrêtée.

Les commandes en arrière-plan ont un timeout défini par défaut à une journée grâce à l'option globale job-timeout. Cette valeur peut être modifiée avec la commande options.

120.4.3. Afficher la liste des tâches en arrière-plan

La commande jobs permet de lister toutes les commandes en cours d'exécution en arrière-plan.

Résultat :

```
[arthas@7656]$ jobs
[11]*
       Running           tt -t fr.jmdoudoux.dej.Service traiter >> &
       execution count : 26
       start time      : Sat Feb 22 01:26:55 CET 2020
       cache location  : C:\Users\jm\logs\arthas-cache\7656\11
       timeout date    : Sun Feb 23 01:26:55 CET 2020
       session         : 313bd832-8236-42cc-a81b-ac0641e2d5f6 (current)
[arthas@7656]$
```

La commande affiche plusieurs informations sur les tâches exécutées en arrière-plan :

- le job id entre crochets
- le statut et la commande exécutée
- le nombre d'exécutions (execution count)
- la date/heure de démarrage (start time)
- Le fichier qui contient la sortie de la commande
- la date/heure du timeout de fin d'exécution : dès qu'elle est atteinte, la commande est arrêtée
- la session

120.4.4. L'arrêt d'une tâche en arrière-plan

Pour arrêter une tâche exécutée en arrière-plan, il faut utiliser la commande kill en lui passant en paramètre le job-id de la tâche concernée.

Résultat :

```
[arthas@7656]$ kill 11
kill job 11 success
[arthas@7656]$ jobs
[arthas@7656]$
```

120.5. Les scripts batch

Les scripts batch permettent d'exécuter plusieurs commandes contenues dans un fichier texte.

Le nom du fichier est libre.

Chaque commande doit être sur une ligne dédiée.

Il faut s'assurer que les commandes s'exécutent en mode batch et qu'elles sont configurées pour s'arrêter, par exemple :

- -b -n de la commande dashboard
- -n pour les commandes monitor, stack, trace, tt et watch

Exemple dans un fichier script.as

Résultat :

```
session
thread -i 30000 -n 1
monitor fr.jmdoudoux.dej.Service traiter -c 5 -n 3
```

Pour exécuter le script, il faut lancer Arthas en utilisant en paramètre :

- l'option -f <script> pour préciser le fichier contenant le script à exécuter
- le pid de la JVM sur laquelle se connecter

Résultat :

```
C:\java> java -jar arthas-boot.jar -f script.as 4192 > script.log

C:\java
```

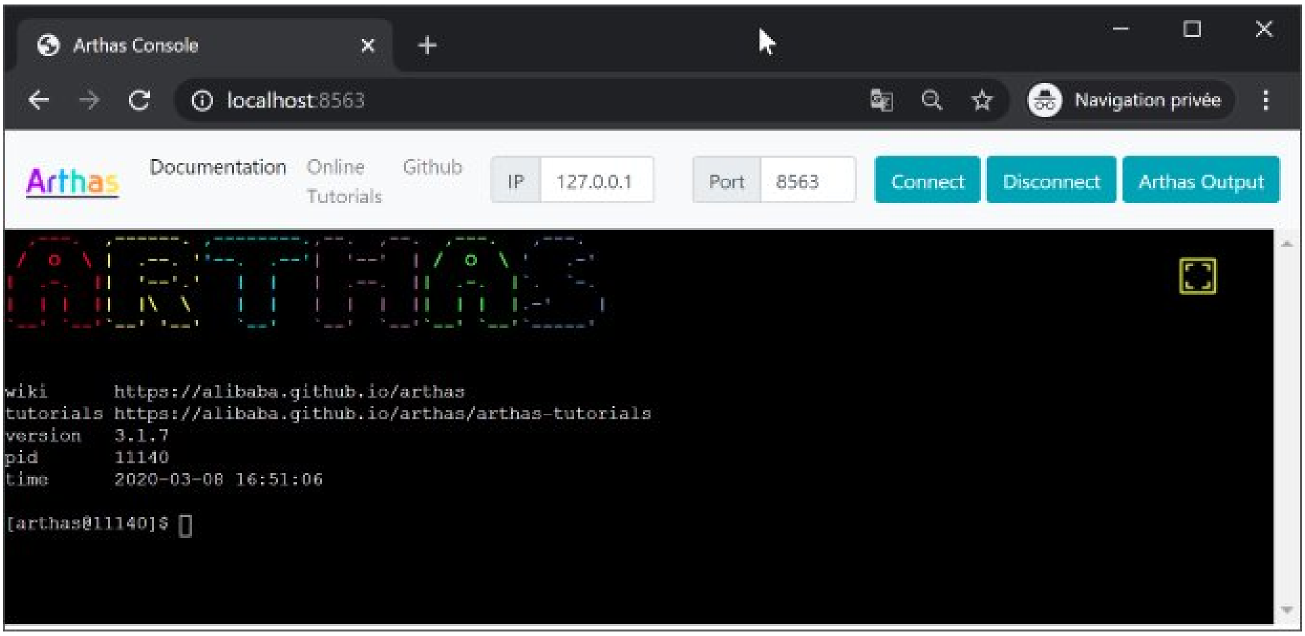
Il est alors possible de consulter le résultat de l'exécution des commandes en consultant le fichier vers lequel la sortie standard a été redirigée.

120.6. La console web

Arthas propose une console web qui s'exécute dans un navigateur : les échanges se font en utilisant des websockets.

Une fois le client Arthas connecté à une JVM, il est possible d'ouvrir un navigateur en utilisant :

- l'IP de la machine sur lequel le client s'exécute. Localement, il est possible d'utiliser localhost ou 127.0.0.1
- le port utilisé. Par défaut, le port est le 8563. Il est indiqué au démarrage du client Arthas.



```
wiki      https://alibaba.github.io/arthas
tutorials https://alibaba.github.io/arthas/arthas-tutorials
version   3.1.7
pid       11140
time      2020-03-08 16:51:06

[arthas@11140]$
```

Le menu en haut permet de se connecter à un autre client en saisissant l'IP et le port et en cliquant sur le bouton « Connect ».

Une fois connectée, il est possible de saisir les commandes à exécuter.

121. VisualVM

Chapitre 121

Niveau : 🟩🟩⬜⬜⬜⬜ Elémentaire

VisualVM est un outil intégré au JDK depuis la version 6 update 7: il permet de superviser l'activité d'une JVM et propose quelques fonctionnalités de base pour profiler des applications Java. Ces fonctionnalités sont similaires à celles de JConsole avec des possibilités supplémentaires.

VisualVM offre de nombreuses fonctionnalités proposées par différents autres outils du JDK pour obtenir des informations de la JVM. C'est un outil graphique qui permet de superviser, d'analyser et de profiler une application exécutée dans une JVM.

VisualVM permet de se connecter sur des JVM locales ou distantes. Chaque JVM utilise un onglet : il est donc possible d'ouvrir plusieurs JVM. Chaque onglet d'une JVM possède plusieurs onglets pour afficher les informations selon leur type.

VisualVM est extensible en permettant l'utilisation de plugins.

La page officielle de VisualVM est à l'url https://visualvm.github.io/

A partir de Java 9, VisualVM n'est plus fourni avec le JDK. Il est développer dans un projet open source.

Depuis sa mise en open source, plusieurs versions ont été publiées :

- version 1.4 en décembre 2017 : support de Java 9
- version 2.0 en février 2020 : les plugins 1.x ne sont pas compatibles 2.x, visualisation des fichiers JFR, recherche dans les threads dumps

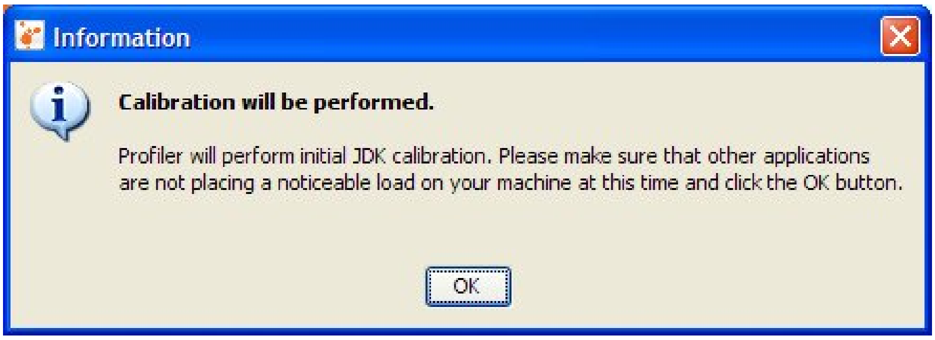
Ce chapitre contient plusieurs sections :

- L'utilisation de VisualVM
- Les plugins pour VisualVM
- L'utilisation de VisualVM
- La connexion à une JVM
- L'obtention d'informations
- Le profilage d'une JVM
- La création d'un snapshot
- Le plugin VisualGC

121.1. L'utilisation de VisualVM

Pour exécuter VisualVM, il faut exécuter l'application jvisualvm.exe contenue dans le sous-répertoire bin du répertoire d'installation du JDK.

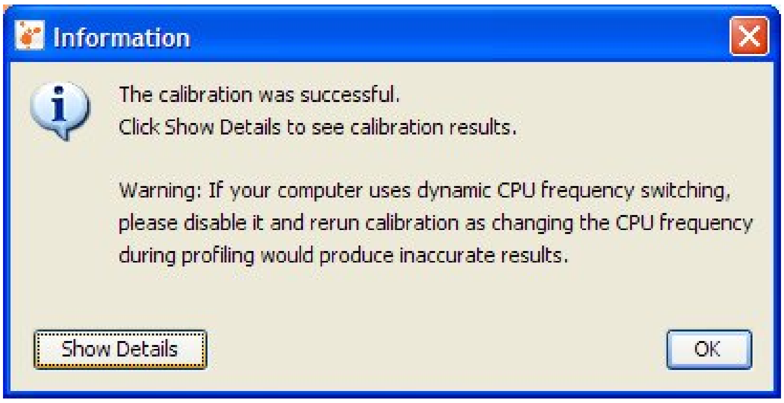
La splashscreen de Visual VM s'affiche. Lors du premier lancement, une boîte de dialogue informe que l'application va se calibrer.



Calibration will be performed.

Profiler will perform initial JDK calibration. Please make sure that other applications are not placing a noticeable load on your machine at this time and click the OK button.

Cliquez sur le bouton « OK ».



The calibration was successful.
Click Show Details to see calibration results.

Warning: If your computer uses dynamic CPU frequency switching, please disable it and rerun calibration as changing the CPU frequency during profiling would produce inaccurate results.

Cliquez sur le bouton « OK ».

Pour utiliser toutes les fonctionnalités de Visual VM, il est nécessaire d'exécuter l'application dans une JVM de Java 6.0 minimum. Le code peut cependant avoir été compilé en Java 5. Il est aussi possible d'utiliser VisualVM sur une JVM 5.0 mais dans ce cas toutes les fonctionnalités ne seront pas utilisables.

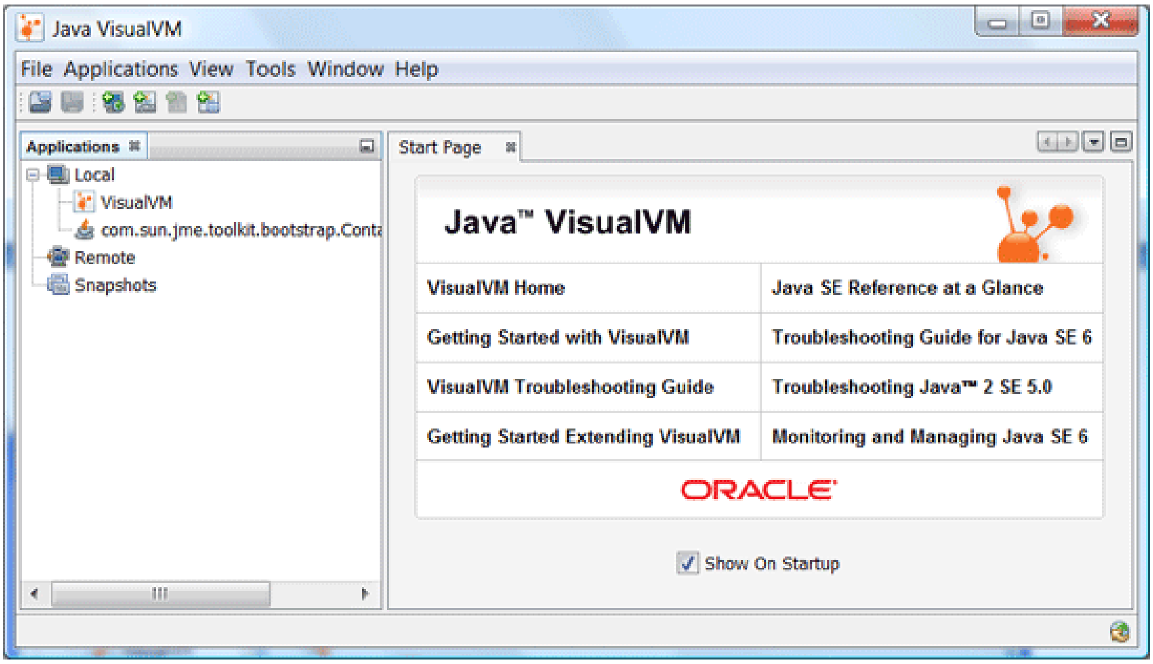
En utilisant une JVM de Java version 6 ou supérieure, VisualVM va pouvoir obtenir des informations plus détaillées notamment concernant la mémoire, la CPU, les threads, ...

VisualVM est une application graphique permettant d'obtenir des informations détaillées sur des applications s'exécutant dans une JVM. Ceci permet d'analyser et de contrôler le fonctionnement et les dysfonctionnements d'une application.

Le JDK proposait déjà plusieurs outils pour obtenir certaines de ces informations : jconsole, jmap, jstack, jstat ou jinfo. VisualVM permet d'obtenir des informations équivalentes et supplémentaires tout en les présentant sous une forme graphique.

Les informations fournies par Visual VM peuvent permettre notamment :

- D'obtenir des informations sur la configuration de la JVM
- D'analyser la consommation de mémoire et de CPU
- De visualiser et de gérer les activités du ramasse-miettes
- De capturer et analyser le heap
- D'identifier des fuites de mémoires
- De mesurer les performances pour les améliorer
- De gérer les MBeans


Java VisualVM
File Applications View Tools Window Help
Applications
Local
VisualVM
com.sun.jme.toolkit.bootstrap.Conta
Remote
Snapshots
Start Page
Java™ VisualVM
VisualVM Home
Java SE Reference at a Glance
Getting Started with VisualVM
Troubleshooting Guide for Java SE 6
VisualVM Troubleshooting Guide
Troubleshooting Java™ 2 SE 5.0
Getting Started Extending VisualVM
Monitoring and Managing Java SE 6
ORACLE
☑ Show On Startup

A son lancement, la fenêtre de VisualVM est composée de deux parties :

- Applications : affiche la liste des applications exécutées dans une JVM locale ou distante dont il est possible d'obtenir des informations et de gérer les snapshots
- La partie principale : l'onglet start propose des liens vers des ressources

Un snapshot est une capture d'informations d'une JVM à un moment donné.

Le menu contextuel d'une application permet plusieurs actions :

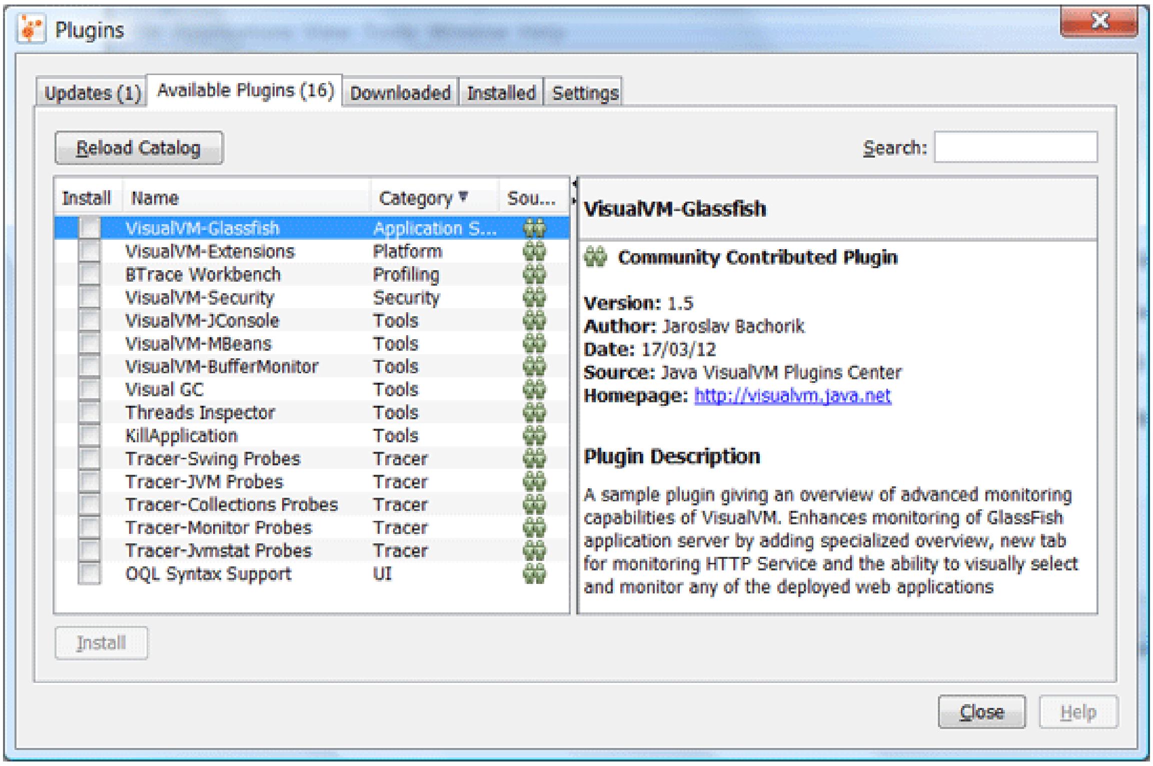
- Open : ouvrir un onglet pour obtenir des informations sur l'application
- Thread dump : obtenir une image des threads
- Heap dump : obtenir une image du tas (instances de classes)
- Application snapshot : créer une image sauvegardée des informations

Les informations concernant une application sont affichées dans la partie centrale. Chaque application possède son propre onglet.

121.2. Les plugins pour VisualVM

VisualVM peut être enrichi de fonctionnalités grâce à des plugins qui peuvent être téléchargés sur le VisualVM plugins Center et gérés dans le plugins manager.

Pour installer un plugin, il faut utiliser le menu Tools/Plugins.



Install	Name	Category	Sou...
☐	VisualVM-Glassfish	Application S...	
☐	VisualVM-Extensions	Platform	
☐	BTrace Workbench	Profiling	
☐	VisualVM-Security	Security	
☐	VisualVM-JConsole	Tools	
☐	VisualVM-MBeans	Tools	
☐	VisualVM-BufferMonitor	Tools	
☐	Visual GC	Tools	
☐	Threads Inspector	Tools	
☐	KillApplication	Tools	
☐	Tracer-Swing Probes	Tracer	
☐	Tracer-JVM Probes	Tracer	
☐	Tracer-Collections Probes	Tracer	
☐	Tracer-Monitor Probes	Tracer	
☐	Tracer-Jvmstat Probes	Tracer	
☐	OQL Syntax Support	UI	

VisualVM-Glassfish

Community Contributed Plugin

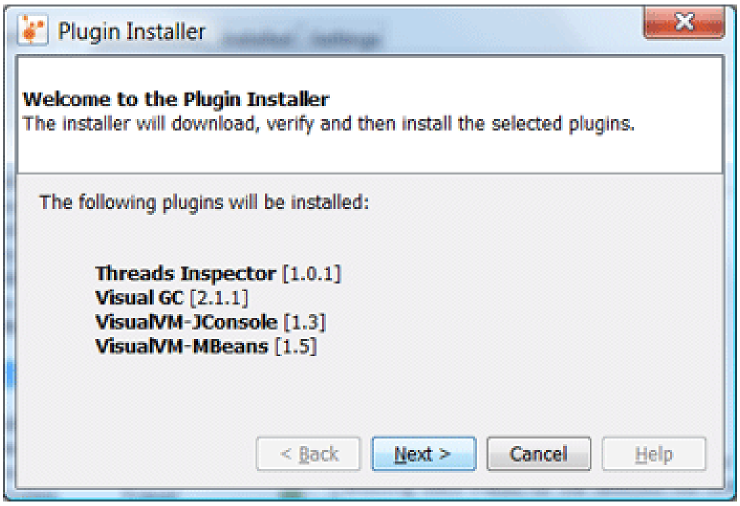
Version: 1.5
Author: Jaroslav Bachorik
Date: 17/03/12
Source: Java VisualVM Plugins Center
Homepage: http://visualvm.java.net

Plugin Description

A sample plugin giving an overview of advanced monitoring capabilities of VisualVM. Enhances monitoring of GlassFish application server by adding specialized overview, new tab for monitoring HTTP Service and the ability to visually select and monitor any of the deployed web applications

L'onglet Updates permet de mettre à jour les plugins déjà installés.

L'onglet Available Plugins permet d'installer de nouveaux plugins. Il suffit de sélectionner les cases à cocher des plugins souhaités et de cliquer sur le bouton « Install ».



Welcome to the Plugin Installer
The installer will download, verify and then install the selected plugins.

The following plugins will be installed:

Threads Inspector [1.0.1]
Visual GC [2.1.1]
VisualVM-JConsole [1.3]
VisualVM-MBeans [1.5]

Cliquer sur le bouton « Next ».

Lisez la licence et si vous l'acceptez, sélectionnez la case à cocher et cliquez sur le bouton « Install ». Les plugins sont téléchargés et installés.

Cliquez sur le bouton « Finish » pour prendre en compte les plugins en redémarrant VisualVM.

121.3. L'utilisation de VisualVM

La fenêtre principale est composée de deux parties :

- la partie de gauche affiche une arborescence des JVM locales et distantes, les threaddumps et les snapshots
- la partie de droite permet d'obtenir les informations sur la ou les JVM connectées (un onglet par JVM)

L'arborescence de l'onglet Applications comporte plusieurs noeuds principaux :

- local : affiche les JVM en cours d'exécution sur la même machine que VisualVM
- remote : affiche les machines distantes et les JVM à surveiller
- snapshots : affiche les snapshots qui ont été pris

L'arborescence affiche aussi pour chaque JVM, les threaddumps, les heapdumps et les snapshots qui ont été sauvegardés. Sur Solaris ou Linux, les coredumps sont aussi affichés.

Chaque machine distante doit être ajoutée explicitement : elle sera conservée par VisualVM tant qu'il arrivera à s'y connecter au démarrage.

Pour afficher automatiquement les JVM en cours d'exécution sur la machine distante, celle-ci doit exécuter l'utilitaire jstatd fourni avec le JDK de Sun/Oracle.

Le menu contextuel associé à une JVM locale permet plusieurs actions :

- Open : se connecter à la JVM
- Thread Dump : demander la génération d'un thread dump
- Heap Dump : demander la génération d'un heap dump
- Profile : se connecter à la JVM et la profiler
- Application Snapshot : capturer les informations sur la JVM
- Enable Heap Dump on OOME : demander la génération d'un heap dump si une exception de type OutOfMemeoryException est levée dans la JVM

Pour chaque JVM connectée, plusieurs onglets permettent d'obtenir des informations :

- Overview : afficher des informations générales sur la JVM
- Monitor : afficher des informations graphiques sur l'activité de la JVM
- Threads : donner un aperçu de l'activité des threads
- Profiler : analyser le comportement de l'application concernant la CPU et la mémoire

D'autres onglets peuvent aussi être disponibles en fonction des plugins installés.

121.4. La connexion à une JVM

VisualVM est capable de détecter et de se connecter automatiquement aux JVM version 6 et supérieures. Pour des JVM version 5 ou distante, il est nécessaire de configurer JMX pour l'activer (en utilisant les propriétés com.sun.management.jmxremote.* lors du lancement de la JVM) et de s'y connecter avec VisualVM.

JMX peut être utilisé pour surveiller et gérer une JVM locale ou distante. Pour activer et configurer la connexion à la JVM, plusieurs propriétés peuvent être fournies au moment de son lancement :

- com.sun.management.jmxremote.port : permet de préciser le port sur lequel il sera possible de se connecter au serveur de MBeans
- com.sun.management.jmxremote.ssl : booléen qui permet de préciser si les échanges sont encryptés avec SSL
- com.sun.management.jmxremote.authenticate : booléen qui permet de préciser si la connection requiert un mot de passe

Exemple :

```
java -Dcom.sun.management.jmxremote.port=3456 -Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.authenticate=false MonApp
```

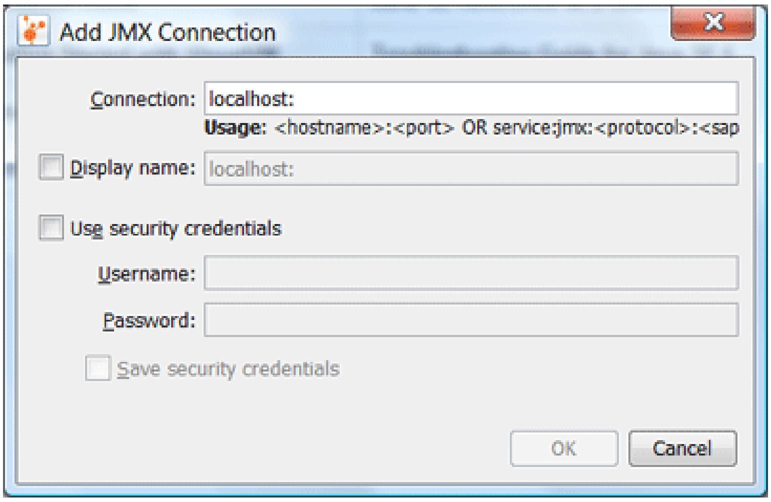
VisualVM n'est pas capable de détecter et se connecter à une JVM :

- locale exécutée avec un utilisateur différent de celui utilisé pour lancer Visual VM
- distante sur laquelle le démon jstatd n'est pas lancé ou si les utilisateurs utilisés pour lancer jstad et la JVM sont différents

L'outil jps peut uniquement détecter les JVM locales exécutées avec l'utilisateur qui a lancé jps.

L'outil jstatd propose une interface qui permet de se connecter aux JVM exécutées sur la machine depuis une machine distante.

Pour se connecter à une JVM locale avec JMX, il suffit d'utiliser l'option "Add JMX connection" du menu contextuel de l'élément Local de l'arborescence Applications. Une boîte de dialogue permet de saisir les informations de connexions à JMX.



Add JMX Connection

Connection: localhost:

Usage: <hostname>:<port> OR service:jmx:<protocol>:<sap

□ Display name: localhost:

□ Use security credentials

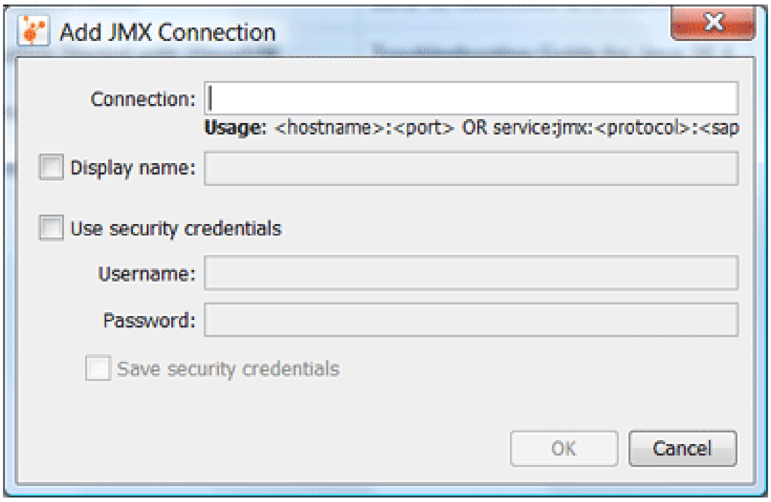
Username:

Password:

□ Save security credentials

Le plus simple est de saisir le numéro du port du serveur de MBean après les deux points dans la zone de saisie Connection et de cliquer sur le bouton "OK". Un nouvel élément apparaît dans la sous-arborescence du noeud Local avec une icône spéciale.

Pour se connecter à une JVM distante en utilisant JMX , il suffit d'utiliser l'option "Add JMX connection" du menu contextuel hors de tout élément de l'arborescence.



Add JMX Connection

Connection:

Usage: <hostname>:<port> OR service:jmx:<protocol>:<sap

□ Display name:

□ Use security credentials

Username:

Password:

□ Save security credentials

Il suffit alors de saisir le nom de la machine hôte, suivi du caractère deux points et du port JMX à utiliser.

VisualVM permet de se connecter à une JVM distante : les JVM distantes sont affichées comme éléments fils de l'élément Remote dans l'arborescence.

Pour se connecter à une JVM distante sans utiliser JMX, il faut lancer l'outil jstatd fourni avec le JDK sur la machine distante. Ensuite, il faut ajouter cette machine en utilisant l'option "Add Remote Host" du menu contextuel de l'élément "Remote" dans l'arborescence.

Une petite boîte de dialogue permet de saisir le nom de l'hôte ou son adresse I.P. et son nom d'affichage.

Si l'outil jstatd est exécuté sur la machine hôte, les JVM qui s'exécutent sur la machine sont affichées comme éléments fils.

121.5. L'obtention d'informations

VisualVM permet d'obtenir des informations générales et sur l'activité de la mémoire et des threads.

121.5.1. La génération d'un thread dump

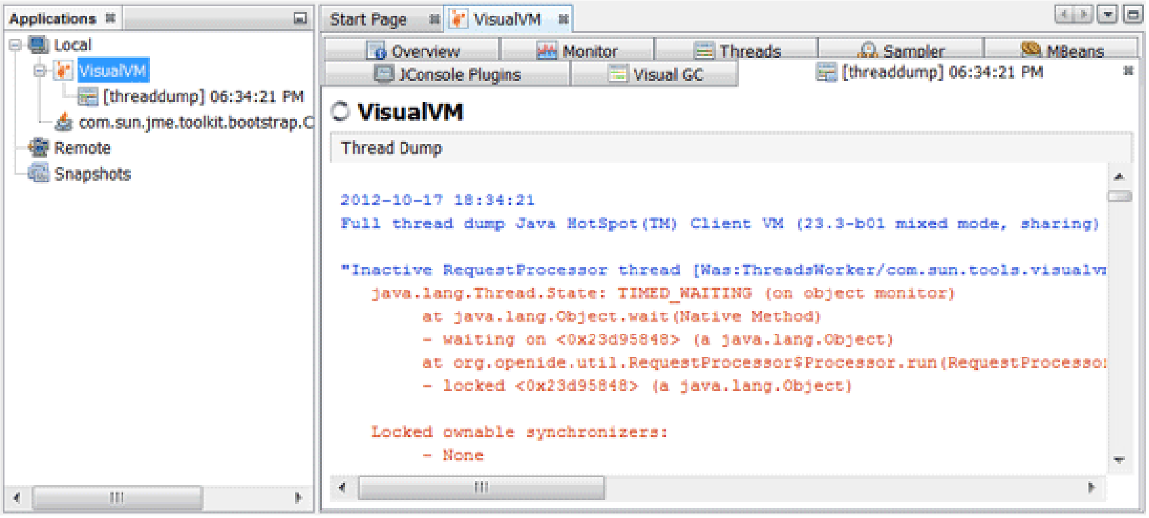
Il est possible d'utiliser VisualVM pour capturer des informations sur les threads dans un thread dump au moment où il est demandé. Un thread dump permet d'obtenir la stacktrace de tous les threads sans arrêter définitivement l'application.

Pour demander un thread dump, il y a deux actions possibles :

- utiliser l'option Thread Dump du menu contextuel de la JVM concernée
- cliquer sur le bouton Thread Dump de l'onglet Threads

Une fois le thread dump généré, un onglet est ajouté pour afficher son contenu ainsi qu'un élément fils de la JVM concernée dans l'arborescence.

Un thread dump est particulièrement utile pour savoir quels traitements sont exécutés par chaque thread.



```
2012-10-17 18:34:21
Full thread dump Java HotSpot(TM) Client VM (23.3-b01 mixed mode, sharing)

"Inactive RequestProcessor thread [Was:ThreadsWorker/com.sun.tools.visualvm
   java.lang.Thread.State: TIMED_WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x23d95848> (a java.lang.Object)
        at org.openide.util.RequestProcessor$Processor.run(RequestProcesso
        - locked <0x23d95848> (a java.lang.Object)

   Locked ownable synchronizers:
        - None
```

121.5.2. La génération d'un heap dump

Il est possible d'utiliser VisualVM pour capturer des informations sur les objets contenus dans le tas (heap) de la JVM au moment de la demande.

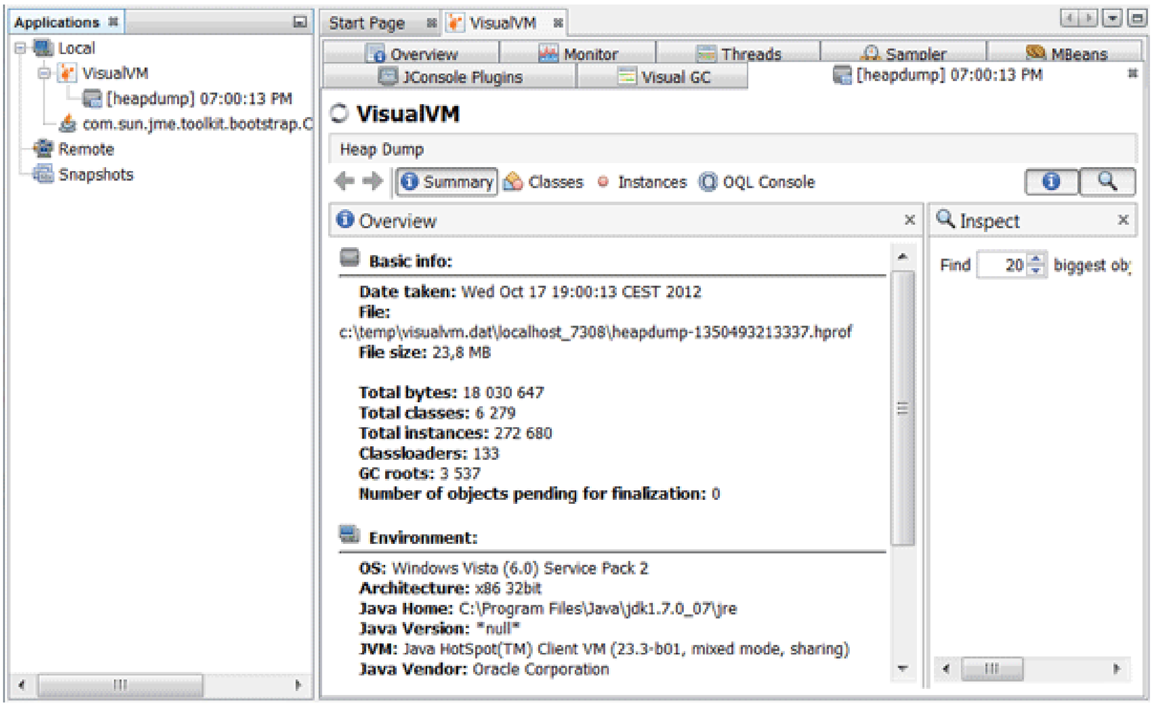
Un heap dump est une image à un instant donné de tous les objets contenus dans le heap d'une JVM. Le parcours d'un heap dump permet de connaître les objets qui ont été créés dans le heap de la JVM.

VisualVM peut être utilisé pour créer un heap dump d'une JVM locale version 6 ou supérieures. Un heap dump créé avec VisualVM est temporaire et doit être explicitement sauvegardé en utilisant l'option Save as du menu contextuel. Si un heap dump n'est pas sauvegardé explicitement, il sera perdu lors de la fermeture de VisualVM.

Pour demander la génération d'un heap dump à VisualVM, il a deux possibilités :

- utiliser l'option "Heap Dump" du menu contextuel de la JVM concernée
- cliquer sur le bouton "Heap Dump" de l'onglet Monitor de la JVM concernée

La génération d'un heap dump crée une entrée dans l'arborescence fille de la JVM et ouvre un nouvel onglet qui va permettre de consulter les informations qu'il contient.



Basic info:

Date taken: Wed Oct 17 19:00:13 CEST 2012
File:
c:\temp\visualvm.dat\localhost_7308\heapdump-1350493213337.hprof
File size: 23,8 MB

Total bytes: 18 030 647
Total classes: 6 279
Total instances: 272 680
Classloaders: 133
GC roots: 3 537
Number of objects pending for finalization: 0

Environment:

OS: Windows Vista (6.0) Service Pack 2
Architecture: x86 32bit
Java Home: C:\Program Files\Java\jdk1.7.0_07\jre
Java Version: "null"
JVM: Java HotSpot(TM) Client VM (23.3-b01, mixed mode, sharing)
Java Vendor: Oracle Corporation

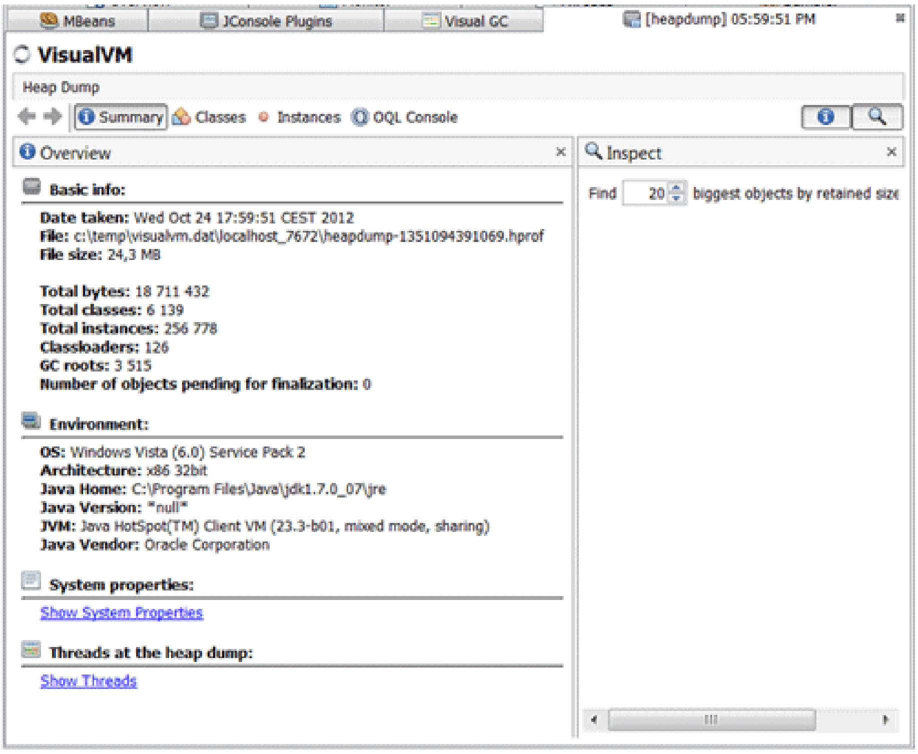
121.5.3. Le parcours d'un heap dump

VisualVM permet de parcourir le contenu d'un fichier heap dump et ainsi de voir les objets du tas. Ces fichiers heap dump peuvent être des fichiers .hprof ou des fichiers heap dump créés grâce à VisualVM.

Pour ouvrir un heap dump, il est possible :

- de double cliquer sur un heap dump affiché dans l'arborescence pour les heap dumps créés avec VisualVM
- d'utiliser le menu File/Load et de sélectionner un fichier .hprof

Chaque heap dump s'affiche dans son propre onglet pour la JVM concernée.



Basic info:

Date taken: Wed Oct 24 17:59:51 CEST 2012
File: c:\temp\visualvm.dat\localhost_7672\heapdump-1351094391069.hprof
File size: 24,3 MB

Total bytes: 18 711 432
Total classes: 6 139
Total instances: 256 778
Classloaders: 126
GC roots: 3 515
Number of objects pending for finalization: 0

Environment:

OS: Windows Vista (6.0) Service Pack 2
Architecture: x86 32bit
Java Home: C:\Program Files\Java\jdk1.7.0_07\jre
Java Version: "null"
JVM: Java HotSpot(TM) Client VM (23.3-b01, mixed mode, sharing)
Java Vendor: Oracle Corporation

System properties:

Show System Properties

Threads at the heap dump:

Show Threads

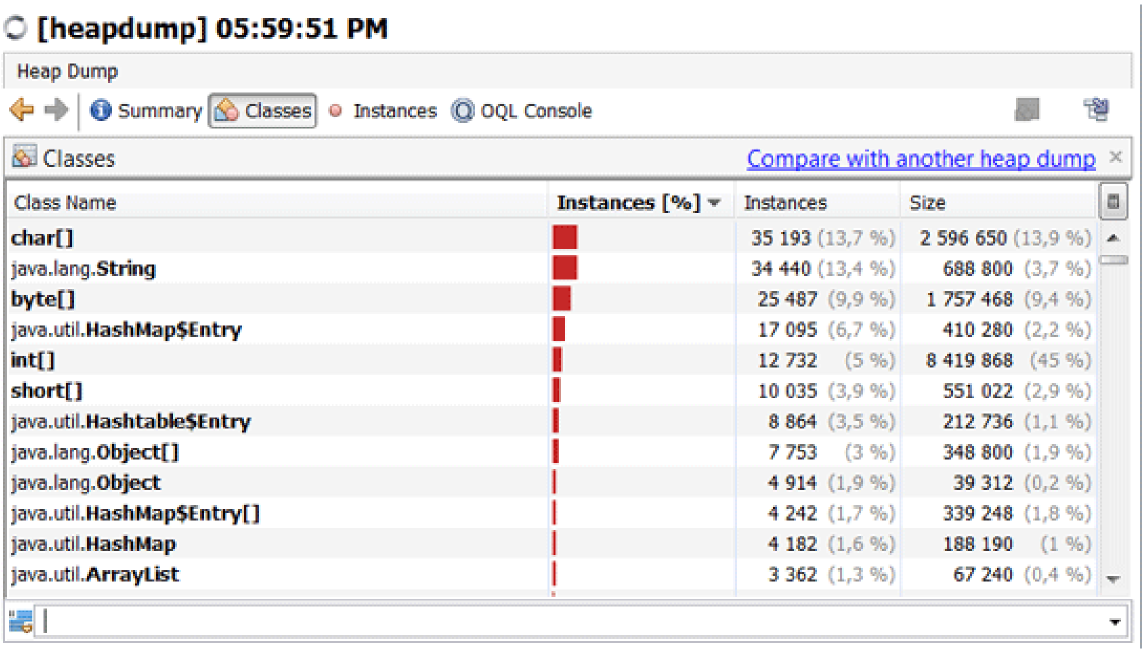
VisualVM permet de consulter le contenu d'un heap dump grâce à plusieurs vues :

- Summary : c'est la vue par défaut.
- Classes : permet d'obtenir le nombre d'instances et leur taille pour chaque classe
- Instances : permet d'afficher les instances d'une classe

La vue Summary affiche des informations sur le contenu du heap dump :

- un résumé du contenu
- l'environnement d'exécution
- les propriétés de la JVM
- les threads

La vue Classes permet d'obtenir pour chaque type le nombre d'instances dans le heap et la taille qu'elles y occupent.

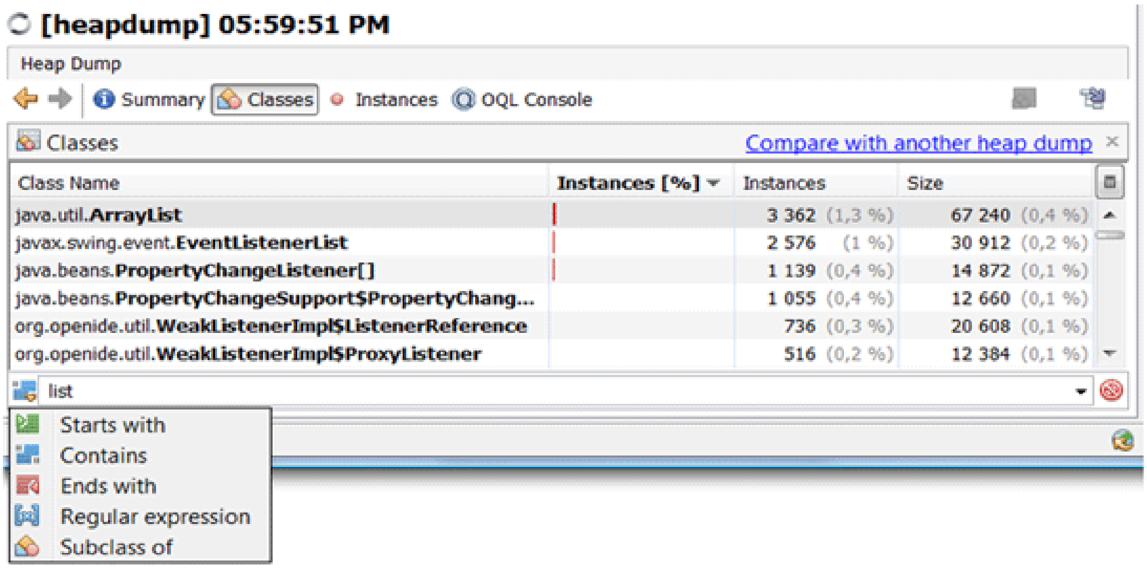


Class Name	Instances [%]	Instances	Size
char[]		35 193 (13,7 %)	2 596 650 (13,9 %)
java.lang.String		34 440 (13,4 %)	688 800 (3,7 %)
byte[]		25 487 (9,9 %)	1 757 468 (9,4 %)
java.util.HashMap$Entry		17 095 (6,7 %)	410 280 (2,2 %)
int[]		12 732 (5 %)	8 419 868 (45 %)
short[]		10 035 (3,9 %)	551 022 (2,9 %)
java.util.Hashtable$Entry		8 864 (3,5 %)	212 736 (1,1 %)
java.lang.Object[]		7 753 (3 %)	348 800 (1,9 %)
java.lang.Object		4 914 (1,9 %)	39 312 (0,2 %)
java.util.HashMap$Entry[]		4 242 (1,7 %)	339 248 (1,8 %)
java.util.HashMap		4 182 (1,6 %)	188 190 (1 %)
java.util.ArrayList		3 362 (1,3 %)	67 240 (0,4 %)

Pour visualiser les instances d'une classe, il faut utiliser le menu contextuel « Show in Instances View »

Il est possible :

- de cliquer sur l'en-tête de chaque colonne pour modifier l'ordre de tri des informations.
- de limiter les classes affichées aux sous-classes de celle dont le menu contextuel « Show Only Subclasses » est utilisé.
- de filtrer les classes affichées. Pour cela, il faut saisir tout ou partie du nom de la classe, sélectionner au besoin le type de filtre à appliquer et cliquer sur le bouton vert à droite de la zone de saisie de texte.

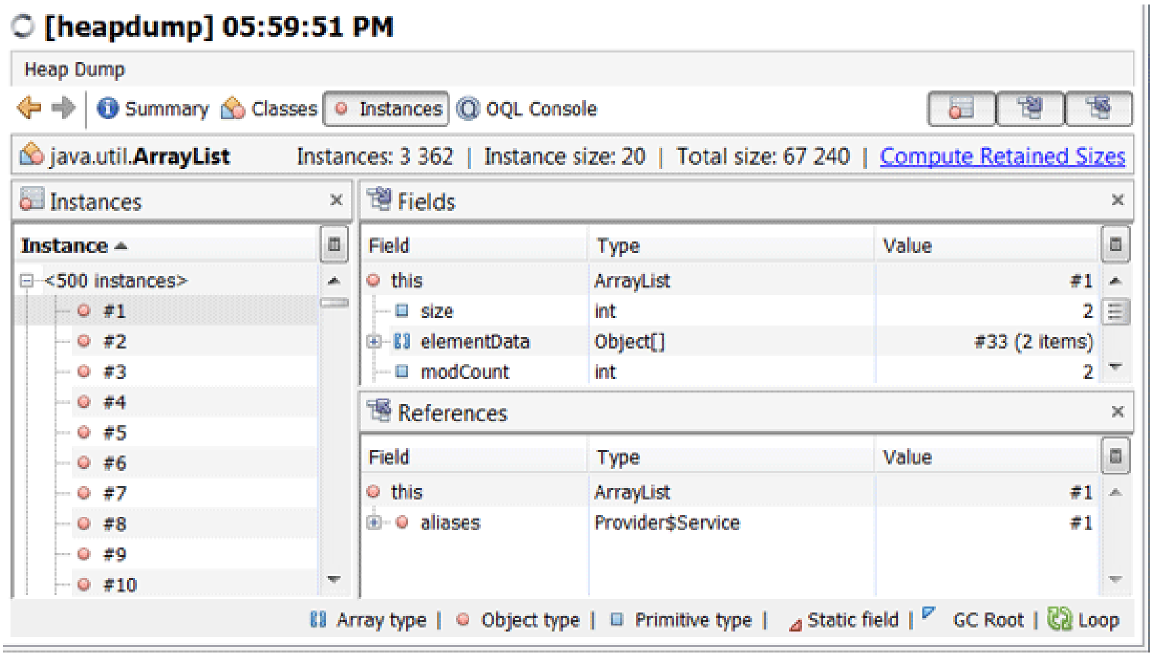


Class Name	Instances [%]	Instances	Size
java.util.ArrayList		3 362 (1,3 %)	67 240 (0,4 %)
javax.swing.event.EventListenerList		2 576 (1 %)	30 912 (0,2 %)
java.beans.PropertyChangeListener[]		1 139 (0,4 %)	14 872 (0,1 %)
java.beans.PropertyChangeSupport$PropertyChang...		1 055 (0,4 %)	12 660 (0,1 %)
org.openide.util.WeakListenerImpl$ListenerReference		736 (0,3 %)	20 608 (0,1 %)
org.openide.util.WeakListenerImpl$ProxyListener		516 (0,2 %)	12 384 (0,1 %)

list

Pour annuler le filtre, il faut cliquer sur le bouton rouge à droite de la zone de saisie de texte.

La vue Instances permet d'afficher la liste des instances d'une classe.



java.util.**ArrayList** Instances: 3 362 | Instance size: 20 | Total size: 67 240 | Compute Retained Sizes

```
Instance ▲
⊟ <500 instances>
    #1
    #2
    #3
    #4
    #5
    #6
    #7
    #8
    #9
    #10
```

Fields

Field	Type	Value
this	ArrayList	#1
size	int	2
⊞ elementData	Object[]	#33 (2 items)
modCount	int	2

References

Field	Type	Value
this	ArrayList	#1
⊞ aliases	Provider$Service	#1

La sélection d'une instance permet d'afficher ses champs et ses références.

121.5.4. L'onglet Overview

L'onglet Overview affiche des informations générales sur la JVM et son environnement d'exécution :

- PID : l'identifiant du processus de la JVM dans le système
- Host : la machine sur laquelle la JVM s'exécute
- Main class : le nom pleinement qualifié de la classe principale
- Arguments : les arguments passés à la classe principale
- JVM : la version de la JVM
- Java Home : le chemin de la JVM
- JVM flags : les arguments passés à la JVM
- Heap dump on OOME : affiche si l'option Heap dump on OOME de la JVM est activée

C'est l'onglet qui est affiché par défaut lors de la connexion à une JVM.

L'onglet Saved Data affiche le nombre de fichiers créés avec VisualVM pour la JVM.

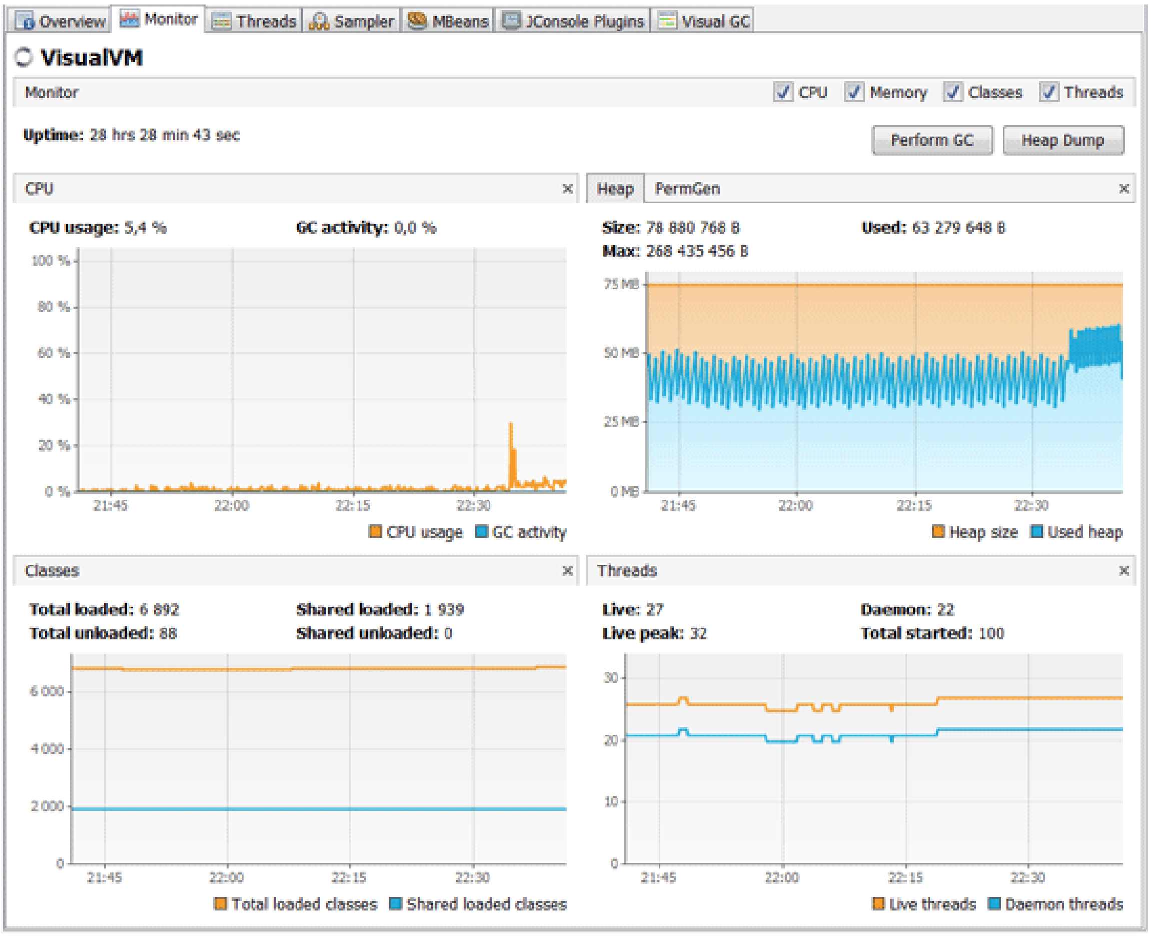
L'onglet JVM Arguments affiche les arguments fournis à la JVM.

L'onglet System properties affiche les propriétés de la JVM.

121.5.5. L'onglet Monitor

L'onglet Monitor affiche des informations en temps réel sur l'évolution de la charge CPU de la JVM, de la taille du Heap et de la PermGen, du nombre de classes chargées et de Threads dans la JVM.

L'utilisation de cet onglet implique uns légère surcharge sur l'utilisation de la JVM.



Uptime: 28 hrs 28 min 43 sec

CPU usage: 5,4 % **GC activity:** 0,0 %

Size: 78 880 768 B **Used:** 63 279 648 B
Max: 268 435 456 B

Total loaded: 6 892 **Shared loaded:** 1 939
Total unloaded: 88 **Shared unloaded:** 0

Live: 27 **Daemon:** 22
Live peak: 32 **Total started:** 100

Les informations sont affichées sous la forme de graphiques en temps réel avec le temps en abscisse :

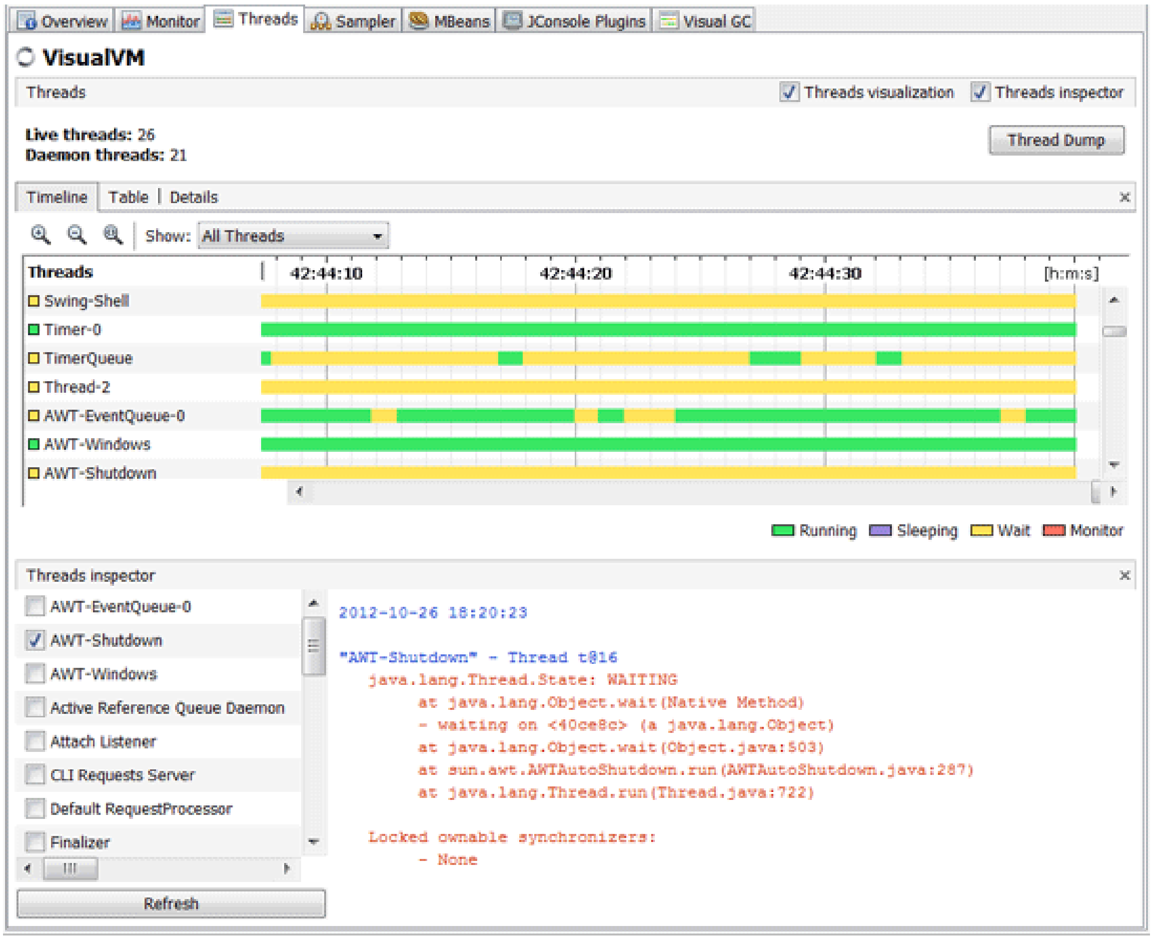
- CPU : affiche l'utilisation CPU de la JVM et l'utilisation de la CPU par le garbage collector
- Heap : affiche la taille du heap, celle utilisée et sa taille maximale. Le graphique affiche l'évolution de ces deux premières propriétés au cours du temps
- PermGen : affiche la taille de la permanent generation, la taille de la permgen et la taille maximale de la permgen. Le graphique affiche l'évolution de ces deux premières propriétés au cours du temps
- Classes : affiche le nombre total de classes chargées et déchargées ainsi que les classes partagées
- Threads : affiche le nombre de threads actifs et de démons

L'onglet Monitor permet de réaliser deux actions grâce à deux boutons :

- « Perform GC » : faire une demande d'exécution du ramasse-miettes à la JVM
- « Heap Dump » : demander la génération d'un heap dump

121.5.6. L'onglet Threads

L'onglet Threads permet de suivre l'activité des threads de la JVM.



Live threads: 26
Daemon threads: 21

☑ Threads visualization ☑ Threads inspector

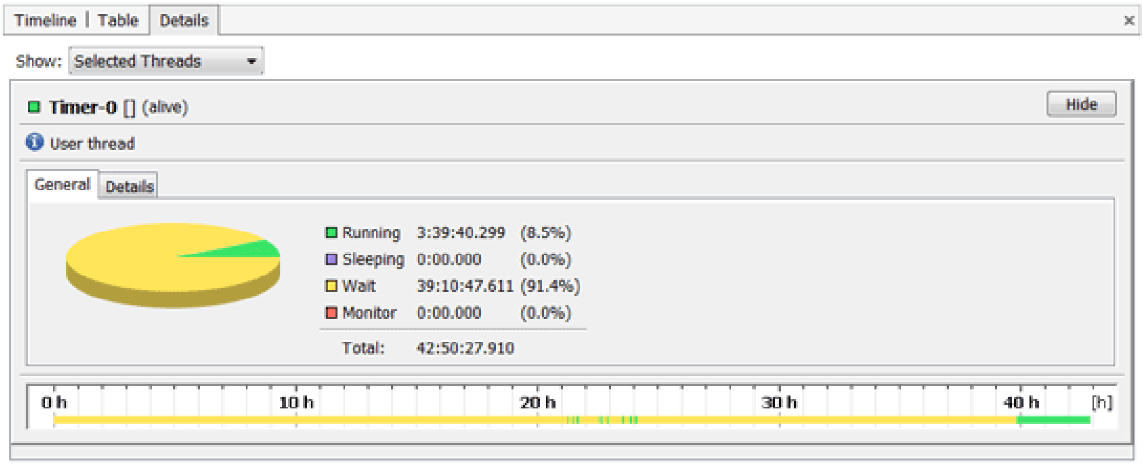
□ AWT-EventQueue-0
☑ AWT-Shutdown
□ AWT-Windows
□ Active Reference Queue Daemon
□ Attach Listener
□ CLI Requests Server
□ Default RequestProcessor
□ Finalizer

```
2012-10-26 18:20:23

"AWT-Shutdown" - Thread t@16
   java.lang.Thread.State: WAITING
        at java.lang.Object.wait(Native Method)
        - waiting on <40ce8c> (a java.lang.Object)
        at java.lang.Object.wait(Object.java:503)
        at sun.awt.AWTAutoShutdown.run(AWTAutoShutdown.java:287)
        at java.lang.Thread.run(Thread.java:722)

   Locked ownable synchronizers:
        - None
```

L'onglet Timeline affiche en temps réel l'activité des threads en affichant leur état. Cette activité est obtenue grâce à JMX ou une JVM locale si sa version est supérieure ou égale à 6.

Il est possible de cliquer sur les boutons "Zoom in" et "Zoom out" pour modifier l'échelle de la ligne de temps. La liste déroulante permet de sélectionner les threads qui sont affichés : tous les threads, les threads actifs ou les threads terminés.

Il est possible de double-cliquer sur un thread pour basculer sur l'onglet Details.



Timer-0 [] (alive)

User thread

🟩 Running	3:39:40.299	(8.5%)
🟪 Sleeping	0:00.000	(0.0%)
🟨 Wait	39:10:47.611	(91.4%)
🟥 Monitor	0:00.000	(0.0%)
Total:	42:50:27.910	

La liste déroulante permet de sélectionner les threads qui sont affichés : les threads sélectionnés, tous les threads, les threads actifs ou les threads terminés.

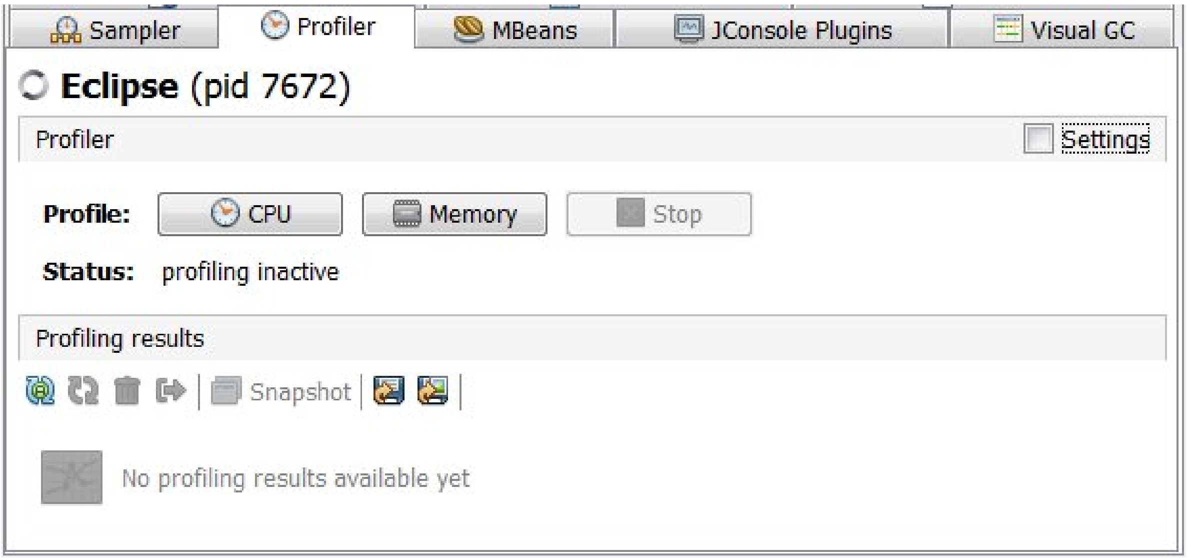
Le sous-onglet Details permet d'obtenir l'heure de chaque changement de statut du thread.

L'onglet General affiche des statistiques sur les états du thread.

121.6. Le profilage d'une JVM

VisualVM intègre un profiler qui permet de fournir des informations pour analyser la performance et l'utilisation mémoire d'une application. VisualVM ne peut pas être utilisé pour se profiler lui-même ou pour profiler une JVM distante.

Pour profiler une application, il faut utiliser l'option Profile du menu contextuel de la JVM concernée qui affiche l'onglet Profiler.



Eclipse (pid 7672)

Profiler

Profile:

Status: profiling inactive

Profiling results

No profiling results available yet

L'onglet Profiler permet de démarrer et de stopper une session de profiling sur l'utilisation de la CPU ou de la mémoire d'une JVM locale.

Le profiling CPU permet de mesurer les performances des classes exécutées dans la JVM.

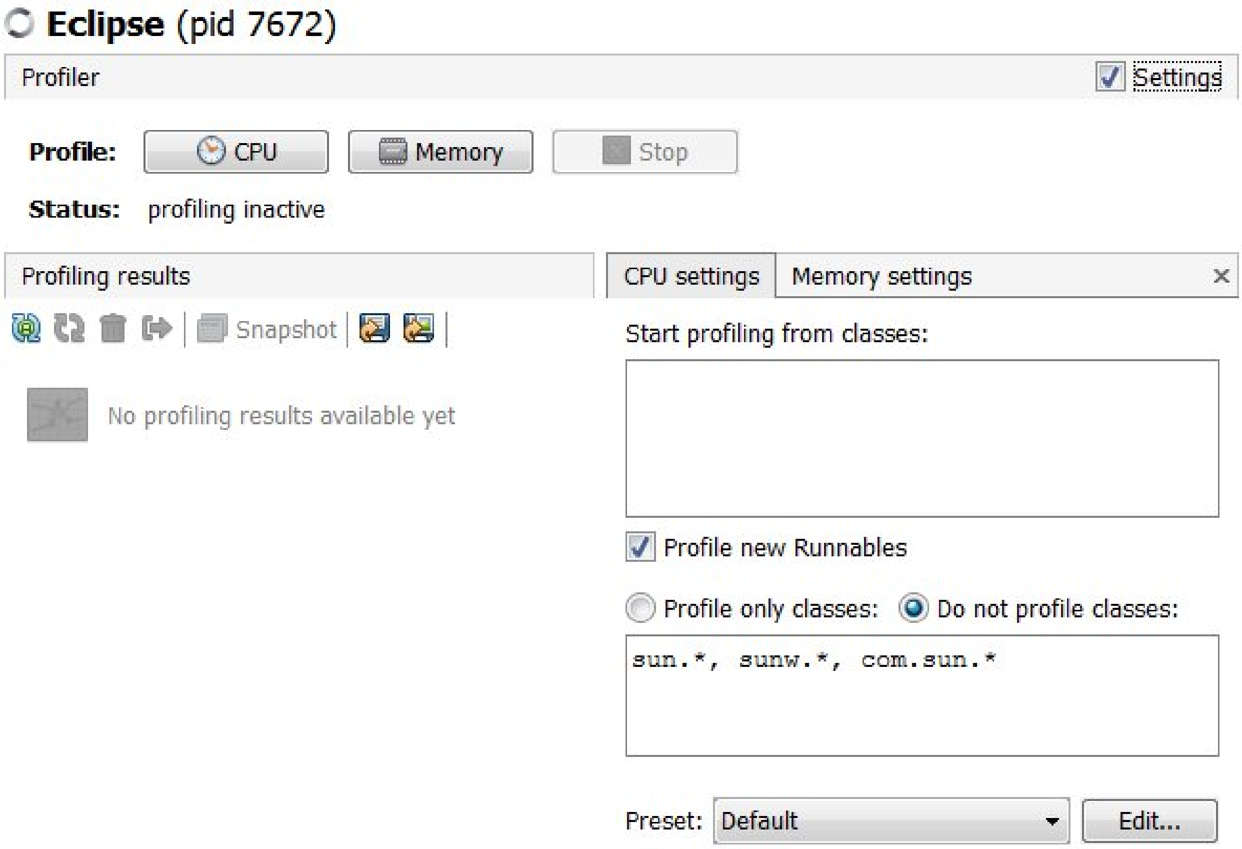
Le profiling mémoire permet d'analyser l'utilisation du heap.

Lors du lancement d'une session de profiling, VisualVM se connecte à la JVM et collecte les informations qui sont affichées dans la partie « Profiling results ».

Elle possède plusieurs boutons :

- Update Result Automatically : une fois activé, les informations sont rafraîchies automatiquement toutes les 2 secondes
- Update Result Now : permet de demander le rafraîchissement des informations
- Run Garbage Collection : demande à la JVM d'exécuter le ramasse-miettes
- Reset Collected Results : permet de réinitialiser les informations déjà collectées
- Take Snapshot : permet de prendre un snapshot des informations collectées. Le snapshot est affiché dans un onglet dédié
- Save Current View : permet de sauvegarder l'affichage courant sous la forme d'une image de type png.

Par défaut, aucune classe n'est instrumentée pour permettre la capture d'informations la concernant. Pour configurer la session de profiling, il faut cocher la case "Settings". La modification de la configuration ne peut se faire que si aucune session de profiling n'est en cours d'exécution.



Status: profiling inactive

Start profiling from classes:

☑ Profile new Runnables

○ Profile only classes: ● Do not profile classes:

```
sun.*, sunw.*, com.sun.*
```

Preset: Default

L'onglet CPU settings permet de configurer les classes à instrumenter. La zone de texte "Start profiling from classes" permet de préciser les classes qui serviront de points d'entrées pour le profiling.

En utilisant le caractère * et selon le bouton radio sélectionné, il est possible de préciser le ou les packages des classes à profiler ("Profile only classes") ou au contraire, de celles qui ne le sont pas ("Dot not profile classes").

L'onglet Memory settings permet de configurer le profiling de la mémoire.

Un bouton radio permet de profiler uniquement les objets créés ou les objets créés et l'activité du ramasse-miettes.

Un tableau affiche pour chaque classe : le nombre d'instances créées, la taille occupée par les objets depuis le début du profiling ainsi que le pourcentage correspondant.

La zone de texte en dessous du tableau permet d'appliquer un filtre sur le nom des classes affichées.

121.7. La création d'un snapshot

VisualVM permet de créer un snapshot qui va contenir les informations collectées de la JVM. Il est possible de sauvegarder un snapshot pour le réouvrir ultérieurement.

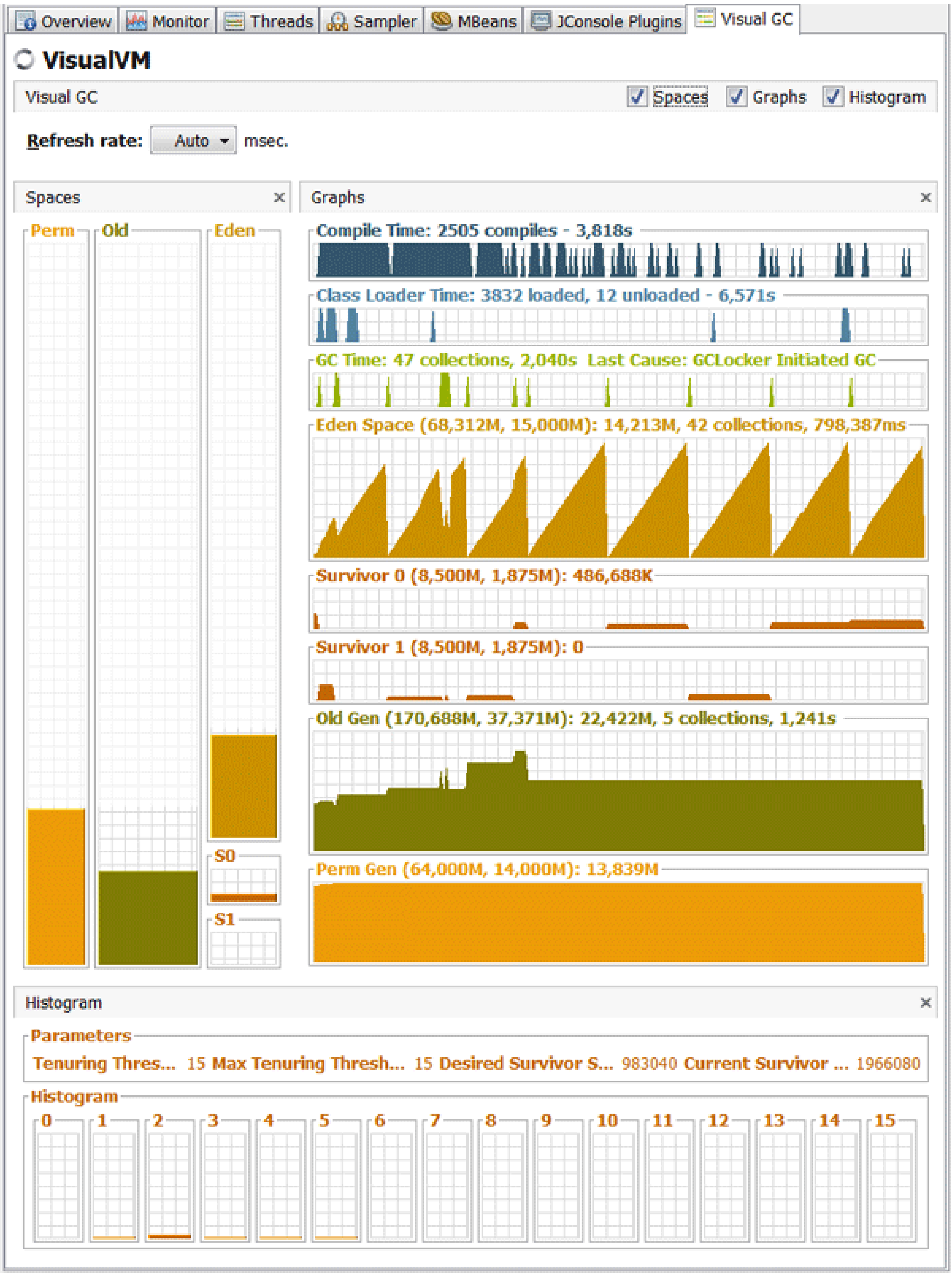
VisualVM peut créer deux types de snapshots :

- Profiler snapshot : le snapshot contient les informations capturées pendant une session de profiling sur l'utilisation de la CPU ou de la mémoire. Ce type de snapshot ne peut être créé que lorsqu'une session de profiling est en cours.
- Application snaphot : le snapshot contient les heap dumps et les thread dumps qui ont été créés

121.8. Le plugin VisualGC

VisualGC est un plugin pour VisualVM qui permet de représenter graphiquement l'activité du ramasse-miettes et de certaines activités de la JVM (PermGen, chargement de classes, compilation JIT).

Une représentation graphique affiche l'occupation des espaces mémoires des différentes générations : young generation (Eden, survivor 1 et 2 (S0 et S1)), old generation et PermGen.



Refresh rate: Auto msec.

Compile Time: 2505 compiles - 3,818s

Class Loader Time: 3832 loaded, 12 unloaded - 6,571s

GC Time: 47 collections, 2,040s Last Cause: GCLocker Initiated GC

Eden Space (68,312M, 15,000M): 14,213M, 42 collections, 798,387ms

Survivor 0 (8,500M, 1,875M): 486,688K

Survivor 1 (8,500M, 1,875M): 0

Old Gen (170,688M, 37,371M): 22,422M, 5 collections, 1,241s

Perm Gen (64,000M, 14,000M): 13,839M

Tenuring Thres... 15 Max Tenuring Thresh... 15 Desired Survivor S... 983040 Current Survivor ... 1966080

Le panneau Compile Time indique le nombre de tâches de compilation exécutées et le temps consacré à cette compilation du bytecode en code natif depuis le lancement de la JVM. La hauteur du graphe n'a pas d'échelle : elle indique simplement une activité de compilation courte ou longue.

Le panneau Class Loader Time indique le nombre de classes chargées, déchargées et le temps passé à ces tâches depuis le démarrage de la JVM. La hauteur du graphe n'a pas d'échelle : il représente l'activité de chargement et de déchargement de classes par la JVM.

Le panneau GC Time indique le nombre d'opérations de récupérations de mémoire réalisées, le temps consacré à ces tâches et la raison de l'exécution de la dernière récupération. La hauteur du graphe n'a pas d'échelle : il représente l'activité du ramasse-miettes.

Le panneau Eden Space indique : la taille maximale de l'espace et l'occupation actuelle entre parenthèses, la taille des objets qu'il contient, le nombre de récupérations de mémoire effectuées dans cet espace et le temps que cela a nécessité depuis le lancement de la JVM. La hauteur du graphe correspond à la taille maximale de l'espace : il représente l'occupation de l'espace Eden.

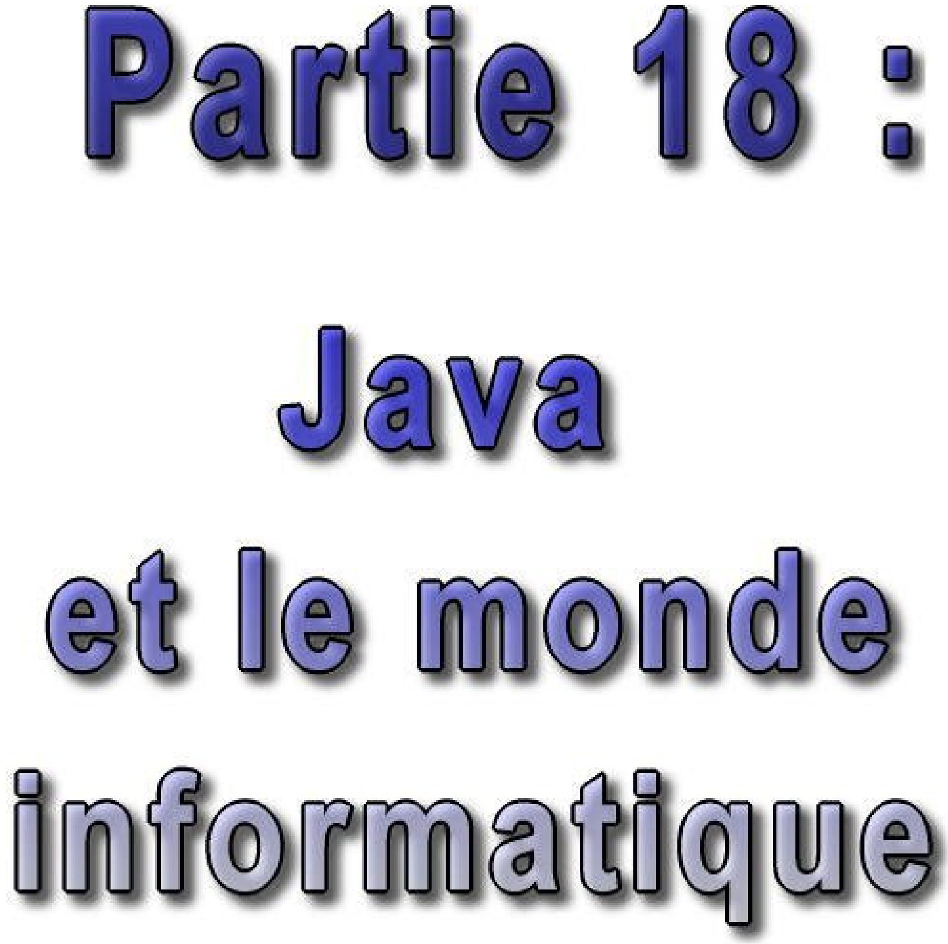
Les panneaux Survivor 0 et Survivor 1 indiquent la taille maximale de l'espace et l'occupation actuelle entre parenthèses, et la taille des objets qu'il contient. La hauteur du graphe correspond à la taille maximale de l'espace : il représente l'occupation de l'espace Survivor concerné.

Le panneau Old Gen indique la taille maximale de l'espace et l'occupation actuelle entre parenthèses, la taille des objets qu'il contient, le nombre de récupérations de mémoire effectuées dans cet espace (full garbage) et le temps que cela à nécessité depuis le lancement de la JVM. La hauteur du graphe correspond à la taille maximale de l'espace : il représente l'occupation de l'espace Old.

Le panneau PermGen indique la taille maximale de l'espace et l'occupation actuelle entre parenthèses ainsi que la taille des objets qu'il contient. La hauteur du graphe correspond à la taille courante de l'espace : il représente l'occupation de l'espace PermGen.

Le panneau Histogram affiche une répartition de l'âge des objets actifs dans l'espace Survivor après la dernière exécution du ramasse-miettes dans cet espace et les paramètres utilisés pour réaliser la promotion des objets.

Partie 18 : Java et le monde informatique



Cette partie contient les chapitres suivants :

- La communauté Java : ce chapitre présente quelques-unes des composantes de l'imposante communauté Java
- Les plates-formes Java et .Net : ce chapitre présente rapidement les deux plates-formes
- Java et C# : ce chapitre détaille les principales fonctionnalités des langages Java et C#

122. La communauté Java

Chapitre 122

Niveau : 🟩□□□□□ Fondamental

En 2020, la plate-forme Java fête son 25^{eme} anniversaire. Une telle durée de vie lui permet d'avoir une large communauté très productive, voire peut être même trop, à tel point que les débutants en Java sont souvent noyés devant une telle masse d'informations et de produits.

La communauté Java est donc très riche de part le monde. Sun puis Oracle contribuent à la vie de cette importante communauté au travers de programme comme le JCP, SDN/OTN, java.net, ...

Divers organismes open source (Apache, Eclipse, Netbeans, CodeHaus, SpringSource, JBoss, ...) enrichissent la communauté d'APIs et d'outils particulièrement utiles et sont même moteurs d'inspirations sur certaines évolutions de Java.

Ce chapitre contient plusieurs sections :

- Le JCP
- Les ressources proposées par Oracle
- Oracle Technology Network
- La communauté Java.net
- Les JUG
- Les Cast Codeurs Podcast
- Parleys.com
- Les conférences Devoxx et Voxxed Days
- Les conférences
- Les unconférences
- Webographie
- Les communautés open source

122.1. Le JCP

Créé en 1998, le JCP (Java Community Process) est le processus chargé de définir les évolutions de Java : cela concerne aussi bien les plates-formes que les API. Le site du JCP est à l'url www.jcp.org

Chaque évolution est traitée sous la forme de propositions nommées JSR (Java Specification Request). Le contenu d'une JSR peut être très varié, allant d'une API, d'une spécification à la définition d'une plate-forme ou encore les évolutions du JCP lui-même. Par exemple, voici quelques JSR :

- JSR 3 : JMX
- JSR 59 : Java 1.4
- JSR 153 : EJB 2.1
- JSR 215 : la version 2.6 du JCP lui-même
- JSR 221 : JDBC 4.0
- ....

Chaque JSR possède un numéro qui est un identifiant unique. Une JSR est prise en charge par plusieurs personnes :

- le leader de la spécification (specification leader)
- un groupe de travail (experts group)

Le groupe de travail est composé de collaborateurs de sociétés (de toutes tailles), de membres de communautés open source (par exemple Apache, Object Web, ...) et même de personnes isolées. La participation au JCP est payante sauf pour les particuliers.

Une spécification évolue selon plusieurs états :

- initialisation
- brouillon
- early draft review
- final
- maintenance

Chaque JSR doit fournir plusieurs éléments pour être validée :

- un document de spécifications
- une implémentation de référence (RI : reference implementation) dont le code source est diffusé
- un kit de tests de compatibilité (TCK : technology compatibility kit) : permet de valider une implémentation des spécifications

Les spécifications et l'implémentation de référence sont publiques par contre la licence du TCK est définie par le groupe de travail.

Certaines JSR ont été purement et simplement abandonnées.

122.2. Les ressources proposées par Oracle

Oracle propose plusieurs sites relatifs à la technologie Java :

- https://www.oracle.com/java/technologies/ : ce site est une véritable mine d'or pour les développeurs Java
- https://dev.java/ : portail qui propose de nombreuses informations sur Java proposées par Oracle et la communauté
- https://inside.java/ : nouvelles et points de vue des membres de l'équipe Java chez Oracle
- Java Magazine : publication numérique d'Oracle proposant des articles techniques sur la plateforme Java rédigés par la communauté
- https://www.java.com/fr/ : ce site est destiné aux utilisateurs de la plateforme Java
- une base de bugs et d'évolutions (Request for enhancements) qui permet d'obtenir la liste, connaître leur état et voter pour déterminer les plus importants à l'url https://bugs.java.com/bugdatabase/

122.3. Oracle Technology Network

Le programme Oracle Technology Network (OTN) fournit de nombreuses ressources sur les technologies Oracle et notamment pour les développeurs Java comme des articles, des vidéos, des outils, ... Pour bénéficier de tout le programme, il faut préalablement s'inscrire gratuitement.

Le site à l'url : https://www.oracle.com/technical-resources/.

122.4. La communauté Java.net

Ce site, historiquement proposé par Sun, permettait à la communauté de trouver un espace pour des projets relatifs à la technologie Java.

Java.net était un site communautaire qui hébergeait de nombreux projets open source, documentations, blogs et autres ressources.

Le site de cette communauté était à l'url https://www.java.net/.

Il a été fermé par Oracle en 2016 après avoir annoncé sa fermeture en 2015.

122.5. Les JUG

Les JUG (Java User Group) sont des regroupements périodiques et généralement géographiques, de passionnés de Java dans le but de partager des expériences et des sujets techniques et de promouvoir la technologie Java.

Depuis 2008, plusieurs JUG se sont créés en France et dans les pays limitrophes.

JUG de Lorraine

https://www.lorrainejug.fr/

JUG de Paris

https://www.parisjug.org

JUG du Luxembourg

https://yajug.lu/

JUG de Belgique

http://www.bejug.org

https://java-developpez-com.dev.java.net

JUG de Bretagne

http://www.breizhjug.org/

JUG de Tours

https://www.toursjug.org

JUG de Bordeaux

http://www.bordeauxjug.org

JUG de Nantes

https://www.nantesjug.org

JUG de Nice et de Sophia Antipolis

http://www.rivierajug.org

LavaJUG : le JUG de Clermont-Ferrand

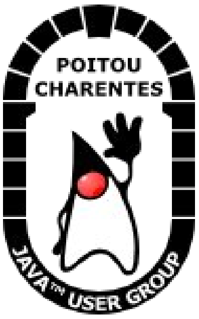
https://www.lavajug.org/

Lyon JUG

https://www.lyonjug.org/

Ch'ti JUG : JUG de Lille

http://chtijug.org



Poitou-Charentes JUG

http://www.poitoucharentesjug.org

Toulouse JUG

https://www.toulousejug.org/

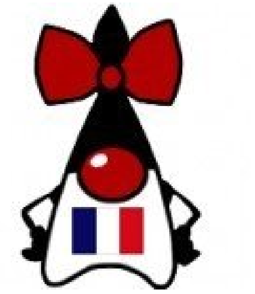
MarsJUG : JUG de Marseille

http://marsjug.org

AlpesJUG : JUG de Grenoble

https://www.alpesjug.fr/

JUGL : JUG Lausanne



JDuchess France

https://www.duchess-france.org/

GenevaJUG : JUG de Genève

https://genevajug.ch/

ElsassJUG : JUG d'Alsace

https://www.meetup.com/fr-FR/ElsassJUG/

MontpellierJUG

https://www.jug-montpellier.org/

JUG de Normandie

Finist JUG : le JUG de Brest

https://finistjug.fr

WAJUG

https://www.wajug.be

Orleans JUG

http://www.jugorleans.fr/

Il existe aussi de nombreux autres JUG à travers le monde, notamment :

Mauritius JUG

Une liste complète des Jug mondiaux est consultable à l'url https://dev.java/community/jugs/.

122.6. Les Cast Codeurs Podcast

Les castcodeurs est un podcast en français relatif à l'industrie Java animé par Emmanuel Bernard, Arnaud Héritier, Guillaume Laforge, Antonio Goncalves, Vincent Massol et Audrey Neveu.

122.7. Parleys.com

Initialement Parleys était une plate-forme de diffusion de vidéos : c'est toujours son rôle principal mais l'outil tend à devenir une plate-forme d'elearning. Parleys diffuse les vidéos de nombreuses conférences notamment Devoxx et Java

One mais aussi les sessions de nombreux JUG.

Certains canaux sont payants mais la plupart sont gratuits. C'est une source d'informations sans équivalent.

De nombreuses vidéos issues de plusieurs conférences (Devoxx, JFocus, Scala Days, ...) sont disponibles sur la chaîne ParleysDotCom de YouTube : https://www.youtube.com/user/Parleysdotcom.

122.8. Les conférences Devoxx et Voxxed Days

122.8.1. Devoxx Belgium (ex : JavaPolis)

Devoxx Belgium (anciennement Javapolis) est le plus important événement indépendant européen relatif aux technologies Java : en 2008, il y avait 3200 participants venant de 35 pays différents, 160 speakers, ... Créé en 2002, il a lieu chaque année au dernier trimestre au Metropolis/Kinepolis d'Anvers en Belgique.

- du 8 au 12 décembre 2008
- du 16 au 20 novembre 2009
- du 15 au 19 novembre 2010
- du 14 au 18 novembre 2011
- du 12 au 16 novembre 2012
- du 11 au 15 novembre 2013
- du 10 au 14 novembre 2014
- du 9 au 13 novembre 2015
- du 7 au 11 novembre 2016
- du 6 au 10 novembre 2017
- du 12 au 16 novembre 2018
- du 4 au 8 novembre 2019
- du 10 au 14 octobre 2022
- du 2 au 6 octobre 2023 (20eme anniversaire)
- du 7 au 11 octobre 2024

Devoxx est organisé par Stephan Janssen et le Bejug. Il se déroule sur 4,5 jours et est composé de deux parties :

- les deux premiers jours : les universités sont des sessions longues et les tools in action
- les trois derniers jours : commencent par des keynotes puis se poursuivent par les conférences qui sont des sessions courtes

Il y a aussi un hall d'exposition, des BOFs, des quickies, ...

Le site de l'événement est à l'url : https://www.devoxx.com

122.8.2. Devoxx France

Devoxx France, franchise de Devoxx, est la plus grande conférence Java en France. Plusieurs éditions ont eu lieu à Paris :

- du 18 au 20 avril 2012
- du 27 au 29 mars 2013
- du 16 au 18 avril 2014
- du 8 au 10 avril 2015, au Palais des Congrès, porte Maillot à Paris
- du 20 au 22 avril 2016
- du 5 au 7 avril 2017
- du 18 au 20 avril 2018
- du 17 au 19 avril 2019
- L'édition 2020 prévue initialement du 15 au 17 avril puis reporté du 1er au 3 juillet est finalement annulée en raison de la crise mondiale du Covid-19
- du 29 septembre au 1er avril 2021
- du 20 au 22 avril 2022 (10^{eme} anniversaire)
- du 12 au 14 avril 2023
- du 17 au 18 avril 2024

Le première journée est constitués des universités (sessions de 3 heures sur un même sujet) et des tools in action (session de 30 minutes). Les deux autres journées sont composées de keynotes pour débuter suivies de conférences (session d'une heure par sujet). Des quickies (session de 15 minutes) et des labs (sessions pratiques de 3 heures) ont été organisés sur les trois journées et des BOF (session informelle d'une heure) sur les deux premières journées.

Cette conférence, organisée par des indépendants issus du Paris JUG, est la déclinaison francophone de Devoxx : 75% des sujets sont en français.

Le site de l'événement est à l'url : https://www.devoxx.fr. Toutes les vidéos sont disponible sur Youtube.

122.8.3. VoxxedDays Luxembourg

Voxxed Days Luxembourg est un évènement IT pour les développeurs organisé à Mondorf Les Bains au Luxembourg par le YaJUG (Java User Group de Luxembourg) :

- 22 juin 2016
- 22 juin 2017
- 22 juin 2018
- 20 juin 2019
- du 21 au 22 juin 2022
- du 21 au 22 juin 2023

Le site de l'événement est à l'url : https://luxembourg.voxxeddays.com/.

122.8.4. VoxxedDays Microservices

VoxxedDays Microservices est un événement axé uniquement sur les microservices organisé à Paris. La conférence est organisée en deux jours de conférences et un jour d'atelier (facultatif) uniquement sur les microservices :

- du 29 au 31 octobre 2018
- du 21 au 23 octobre 2019

Le site de l'événement est à l'url : https://voxxeddays.com/microservices/.

122.9. Les conférences

Plusieurs conférences relatives à Java ont lieu dans le monde dont quelques-unes en Europe. Ces conférences sont l'occasion de rencontrer des membres de la communauté Java et d'obtenir de nombreuses informations sur les API et technologies présentes et futures relatives aux plates-formes Java.

Les conférences jouent un rôle important dans la progression d'un développeur Java non seulement pour assister à des sessions thématiques techniques mais aussi rencontrer les autres membres connus ou non de la communauté Java. Lorsque l'on assiste à ses premières conférences, on y va pour assister aux sessions puis on y participe pour rencontrer d'autres amateurs de technologies Java.

122.9.1. JavaOne

JavaOne est la grande conférence annuelle organisée par Sun Microsystems au centre Moscone de San Fransisco. Cette conférence permet de découvrir de nombreuses applications et technologies relatives à Java. C'est aussi le moment pour Sun de diffuser des annonces et faire connaître des utilisations anodines de Java.

Le site de l'événement est à l'url : https://www.oracle.com/javaone/index.html.

122.9.2. JCertif

JCertif est la plus grande conférence Java en Afrique Centrale. Elle a lieu à Brazaville au Congo :

- du 26 au 29 août 2010
- du 31 août au 4 septembre 2011
- du 3 au 9 septebre 2012
- du 9 au 16 septembre 2013

- du 8 au 14 septembre 2014
- du 23 au 26 septembre 2015
- du 29 septembre au 1 octobre 2016
- du 27 au 29 septembre 2018

Le site de l'événement est à l'url : https://www.jcertif.com

122.9.3. Mix-IT

Mix-IT la conférence autour des technologies Java, l'agilité et l'innovation organisée par le Lyon JUG et le Club Agile Rhône-Alpes :

- 5 avril 2011
- 26 avril 2012
- 25 et 26 avril 2013
- 29 et 30 avril 2014
- 16 et 17 avril 2015
- 21 et 22 avril 2016
- 20 et 21 avril 2017
- 19 et 20 avril 2018
- 23 au 24 mai 2019
- L'édition 2020 prévue les 29 et 30 avril est annulée en raison de la crise mondiale du Covid-19
- 24 au 35 mai 2022
- 13 au 14 avril 2023
- 25 et 26 avril 2024

La conférence est organisée autour de 5 thèmes : techy, agility, mixy, trendy et gamy

Elle permet de rassembler des acteurs de différents horizons et différents domaines en leur permettant d'échanger sur des sujets qui peuvent être variés.

Le site de l'événement est à l'url : https://mixitconf.org/

122.9.4. JUG Summer Camp

Le Poitou-Charentes JUG organise une journée complète de conférences dédiées à Java à La Rochelle au mois de septembre :

- 10 septembre 2010
- 16 septembre 2011
- 14 septembre 2012
- 20 septembre 2013
- 19 septembre 2014
- 18 septembre 2015
- 16 septembre 2016
- 15 septembre 2017
- 14 septembre 2018
- 13 septembre 2019
- 9 septembre 2022
- 8 septembre 2023

Le site de l'événement est à l'url : https://www.jugsummercamp.org/

122.9.5. Codeurs en Seine

Codeurs en Seine est une conférence gratuite qui se déroule à l'Université de Rouen, sur plusieurs thèmes en simultané : le Java, l'Agile, le Web, la technologie.

- 17 octobre 2013
- 27 novembre 2014
- 26 novembre 2015
- 24 novemvre 2016
- 23 novembre 2017
- 22 novembre 2018
- 21 novembre 2019
- 18 novembre 2021
- 26 octobre 2023

Le site de l'événement est à l'url : https://www.codeursenseine.com

122.9.6. Breizhcamp

Breizhcamp est une conférence orienté vers les développeurs organisé à Rennes à l'nitiative du BreizhJUG.

- 17 juin 2011
- 14 et 15 juin 2012
- 13 et 14 juin 2013
- 21 au 23 mai 2014
- 10 au 12 juin 2015
- 23 au 25 mars 2016
- 19 au 21avril 2017
- 28 au 30 mars 2018
- 20 au 22 mars 2019
- L'édition 2020 prévue du 25 au 27 mars est annulée en raison de la crise mondiale du Covid-19
- du 29 juin au 1er juillet 2022
- du 28 juin au 30 juin 2023

Le site de l'événement est à l'url : https://www.breizhcamp.org/

122.9.7. RivieraDev

RivieraDev est une conférence à destination des développeurs proposant des conférences et des ateliers sur les technologies numériques qui se déroule à Sophia-Antipolis.

- du 20 au 21 octobre 2011
- du 11 au 12 juin 2015
- du 16 au 17 juin 2016
- du 11 au 12 mai 2017
- du 16 au 18 mai 2018
- du 15 au 17 mai 2019
- L'édition 2020 prévue du 13 au 15 mai est annulée en raison de la crise mondiale du Covid-19
- 2 juillet 2021
- 10 au 12 juillet 2023

Le site de l'événement est à l'url : https://rivieradev.fr/

122.9.8. Sunny Tech

Sunny Tech est une conférence sur les technologies numériques qui se déroule à Montpellier.

- 28 et 29 juin 2018
- 27 et 28 juin 2019
- L'édition 2020 prévue les 1er et 2 juillet est annulée en raison de la crise mondiale du Covid-19
- 30 juin au 1er juillet 2022
- 28 au 20 juin 2023
- 4 et 5 juillet 2024

Le site de l'événement est à l'url : https://sunny-tech.io/

122.9.9. SnowCamp

SnowCamp est une conférence pour les devs, les ops et les archis composée d'universités, de conférences et d'unconférence au centre des Congrès de Grenoble.

- du 21 au 22 janvier 2016
- du 8 au 10 février 2017
- du 24 au 26 janvier 2018
- du 23 au 25 janvier 2019
- du 22 au 25 janvier 2020
- du 2 au 5 février 2022
- du 25 au 28 janvier 2023
- du 31 janvier au 2 février 2024

Le site de l'événement est à l'url : https://snowcamp.io/fr/

122.9.10. Touraine Tech

Touraine Tech est une conférence sur les technologies numériques qui se déroule à Tours.

- 23 février 2018
- 1er février 2019
- 31 janvier 2020
- 21 janvier 2022
- 19 et 20 janvier 2023
- 8 et 9 fevrier 2024

Le site de l'événement est à l'url : https://touraine.tech/

122.9.11. Bdx.io

Bdx.io est une conférence Bordelaise sur le thème de la progammarion et des métiers annexes qui se déroule au palais des congrès de Bordeaux.

- 17 octobre 2014
- 16 octobre 2015
- 10 octobre 2016
- 10 novembre 2017
- 9 novembre 2018
- 15 novembre 2019
- 30 octobre 2020
- 2 décembre 2022
- 10 novembre 2023

Le site de l'événement est à l'url : https://www.bdxio.fr/

122.9.12. SophiaConf

SophiaConf est un ensemble de conférences dont certaines autour des technologies Java qui ont lieu à Sophia Antipolis :

- 30 juin au 9 juillet 2010
- 4 au 7 juillet 2011
- 2 au 4 juillet 2012
- 1er au 3 juillet 2013
- 30 juin au 1er juillet 2014
- 6 au 9 juillet 2015
- 4 au 7 juillet 2016
- 3 au 6 juillet 2017
- 2 au 5 juillet 2018
- 1er au 3 juillet 2019
- 28 au 30 juin 2021
- 27 au 29 juin 2022

122.9.13. EclipseCon France

Eclipse Con France est une conférence en anglais organisé par la fondation Eclipse qui se déroule à Toulouse.

- 18 et 19 juin 2014
- 24 et 25 juin 2015
- 7 au 9 juin 2016
- 21 et 22 juin 2017
- 13 et 14 juin 2018

Le site de l'événement est à l'url : https://www.eclipsecon.org/

122.9.14. Jazoon

Jazoon est une conférence sur les technologies Java, et .Net et méthodologies depuis 2011, qui a lieu au mois de Juin.

Un des avantages de l'événement est d'être situé au milieu de l'Europe puisqu'il a lieu à Zurich en Suisse. La première session a eu lieu en 2007 et elle est reconduite chaque année :

- 24 au 28 juin 2007 : https://jazoon.com/history/portals/0/Content/ArchivWebsite/jazoon.com/jazoon07/en.html
- 23 au 26 juin 2008 : https://jazoon.com/history/portals/0/Content/ArchivWebsite/jazoon.com/jazoon08/en.html
- 22 au 25 juin 2009 : https://jazoon.com/history/portals/0/Content/ArchivWebsite/jazoon.com/jazoon09/en.html
- 1 au 3 juin 2010 : https://jazoon.com/history/2010/
- 21 au 23 juin 2011 : https://jazoon.com/history/2011/
- 26 au 28 juin 2012 : https://jazoon.com/history/2012/
- 22 au 23 octobre 2013
- 21 au 22 octobre 2014

Le site de l'événement est à l'url : https://www.jazoon.com

122.10. Les unconférences

Plusieurs unconférences relatives à Java ont lieu dans le monde dont quelques-unes en Europe et une en France.

122.10.1. JChateau

JChateau est une unconference qui a lieu en France dans la vallée des chateaux de la Loire à Ambroise.

- 11 au 15 mars 2020
- 15 au 18 mars 2023
- 6 au 9 mars 2024

Le site de l'événement est à l'url : https://www.jchateau.org/

122.11. Webographie

https://java.developpez.com

https://www.theserverside.com/

TheServerSide est un site communautaire qui aborde les sujets relatifs aux développements d'entreprises avec Java au travers d'articles et de débats souvent animés et engagés surtout ceux relatifs aux technologies de demain.

https://dzone.com/java-jdk-development-tutorials-tools-news

https://www.javaspecialists.eu/

http://javaposse.com/

http://javatoolbox.com/ propose un recensement très complet des outils, frameworks, APIs pour Java

https://www.infoq.com/ est un site qui propose de nombreuses ressources sur Java mais aussi sur d'autres technologies. Les ressources Java sont directement accessibles à l'url https://www.infoq.com/java/

https://www.ibm.com/developerworks/java/ proposé par IBM, contient de nombreux articles, ressources et téléchargements.

https://www.javaworld.com/ propose depuis très longtemps des articles techniques relatifs aux technologies Java

https://www.thoughtco.com/java-programming-4133478

http://www.javaperformancetuning.com/

122.12. Les communautés open source

La communauté open source Java est très vaste et très productive.

122.12.1. Apache - Jakarta

Le projet Jakarta de la fondation Apache regroupe un ensemble de sous-projets très connus composés :

- de bibliothèques : commons, POI, Cactus, ORO, TagLibs, JCS, ...
- de frameworks : Struts, Tapestry, HiveMind, ...
- et d'outils : Tomcat, Ant, Jmeter, Maven, ...

Le site est à l'url https://jakarta.apache.org/

122.12.2. La fondation Eclipse

Créé en 2004, la foundation Eclipse regroupe plus de 350 projets :

- L'IDE Eclipse et de nombreux plugins
- BIRT
- Jakarta EE
- Open J9
- Eclipse Collections
- Vert.x
- ...

Le site est à l'url http://www.eclipse.org/

122.12.3. Codehaus

La fondation Codehaus proposait une infrastructure pour permettre à la communauté de développer des projets open source. Parmi ces projets, il y avait Xfire, izpack, mojo, Sonar, m2eclipse, Castor, XStream, ...

Le site www.codehaus.org a été fermé en 2015.

122.12.4. OW2

OW2 est un consortium qui regroupe des organismes de recherche et des entreprises dans le but de développer des projets et même une plate-forme open source notamment Jonas, Joram, Enhydra, Petals, Easybeans, ...

Le site est à l'url https://www.ow2.org/

122.12.5. JBoss

JBoss propose une plate-forme complète incluant un serveur d'applications (jBoss AS, JBoss transaction, JBoss web services, ...), un portail (JBoss Portal), un ESB (JBoss ESB), de nombreuses bibliothèques (Hibernate, Seam, RichFaces, JGroups, RestEasy...) et des outils (Jboss Tools, ...)

Le site est à l'url https://www.jboss.org

122.12.6. Source Forge

Même s'il n'est pas dédié exclusivement à Java, SourceForge héberge de nombreux projets relatifs à Java comme le framework ZK, Dozer, FreeMarker, DBUnit, JfreeChart, Granite DS, ...

Il propose aussi d'excellents outils tels que PMD, Findbugs, SoapUI, WinMerge, MinGW, ...

Le site est à l'url https://sourceforge.net/

123. Les plates-formes Java et .Net

Chapitre 123

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Java et .Net sont les deux principales plates-formes de développement d'applications de différents types (standalone, client/serveur, applications web et applications mobiles).

Java est découpée en plusieurs plates-formes :

- Java SE (J2SE) : la plate-forme standard
- Java EE (J2EE) : la plate-forme pour le développement d'applications d'entreprise
- Java ME (J2ME) : la plate-forme pour le développement d'applications mobiles
- Java Card

.Net est découpée en deux plates-formes :

- .Net Framework : pour le développement d'applications standalone et web
- compact .Net Framework : pour le développement d'applications web

.Net semble plus homogène, essentiellement au niveau des API de ses plates-formes car son développement est plus récent et le nombre de systèmes cibles est beaucoup plus restreint (Windows et Windows CE/Mobile).

Le tableau ci-dessous recense les principaux points communs et différences des deux principales plates-formes.

	Java (SE et EE)	.Net
Environnement d'exécution	Java Virtual Machine (JVM)	Common Language Runtime (CLR)
Format de la compilation	Bytecode	Microsoft Intermediate language (MSIL)
Langage	Mono langage : Java uniquement Quelques solutions open source permettent de générer du bytecode (exemple : JPython, Groovy, ...)	Multi-langage : de nombreux langages dont C#, C++, Visual Basic .Net, C++, JScript, Delphi, Cobol .Net, ...
Mode d'exécution	Mode interprété ou compilateur JIT	Compilateur JIT
Système cible	Support pour tous les systèmes proposant un JVM	Plateforme Windows uniquement Quelques solutions open source permettent l'exécution sur d'autres plates-formes (exemple Mono sous Linux)
Mode de diffusion	Fournie sous la forme de spécifications avec une implémentation de référence Chaque spécification peut être implémentée par un tiers	Fournie sous la forme de produit
IDE	Nombreux outils commerciaux et open source (Eclipse, NetBean)	Microsoft Visual Studio

	Java (SE et EE)	.Net
Evolutions	Gérées par le JCP (Java Community Process composé de membres de nombreuses sociétés ou d'individuels) Chaque évolution est traitée dans une JSR	Uniquement à l'initiative de Microsoft notamment pour les bibliothèques La version 1.0 de C# et CLI (CLR et bibliothèques de bases) sont normalisées par les organismes ECMA et ISO
gestion de la mémoire	Garbage collector	Garbage collector
Packaging	Archive jar, war ou ear selon le type de projet	Assembly (.dll) ou executable (.exe)
Serveur d'applications	Nombreux serveurs d'applications J2EE (Websphere, Weblogic, OAS, Jboss, Jonas, ...)	IIS uniquement

Ce chapitre contient plusieurs sections :

- La présentation des plates-formes Java et .Net
- La compilation
- Les environnements d'exécution
- Le déploiement des modules
- Les version des modules
- L'interopérabilité inter-language
- La décompilation
- Les API des deux plates-formes

123.1. La présentation des plates-formes Java et .Net

En juin 2000, Microsoft lance la plate-forme .Net et un nouveau langage qui lui est dédié : C# (prononcer C Sharp). Cette plate-forme tranche profondément avec les précédentes plates-formes de Microsoft tant au niveau développement qu'exécution.

L'environnement d'exécution des applications .Net est le CLR (Common Language Runtime) qui est l'équivalent de la JVM pour les applications Java.

Le développement d'applications .Net et Java s'appuie sur un ensemble d'API de base fourni respectivement par les deux plates-formes.

Les applications .Net peuvent être écrites dans les différents langages qui proposent un compilateur pour produire du MSIL (MicroSoft Intermediate Language) qui est l'équivalent du bytecode de Java.

L'un des auteurs du langage C# est Anders Heljsberg qui est aussi l'auteur de Delphi.

La documentation de la plate-forme .Net est consultable aux url suivantes :

https://msdn.microsoft.com/fr-fr/netframework/default.aspx

https://msdn.microsoft.com/en-us/netframework/default.aspx

123.1.1. Les plates-formes supportées

Pour .Net, Microsoft ne supporte officiellement que les systèmes d'exploitation Windows et Windows CE/Mobile.

Certaines parties de la bibliothèque de base de .Net sont dépendantes du système d'exploitation Windows notamment la bibliothèque WinForm.

Plusieurs projets open source concernent le portage de .Net sur d'autres systèmes que Windows notamment Mono sous Linux et Rotor sous Free BSD.

La portabilité multiplateformes est un point fort qui a favorisé le succès de Java (Write Once, Run Anywhere). Celle-ci est assurée grâce à la compilation du code source en un langage intermédiaire exécutable par toutes les JVM.

Les plates-formes Windows, Solaris et Linux sont officiellement supportées : d'autres éditeurs proposent le support d'autres systèmes notamment AIX (IBM) et Mac OS (Apple).

123.1.2. Standardisation

Le langage C# est normalisé par l'ECMA (ECMA-334) fin 2001 et par l'ISO/CEI(ISO/CEI 23270) en 2003.

Les évolutions des plates-formes Java sont réalisées par le JCP.

123.2. La compilation

La compilation de code Java et d'un langage supporté par .Net crée une entité qui contient du pseudo code : Bytecode pour Java et MSIL (Microsoft Intermediate Langage) pour .Net.

Le code MSIL est commun à tous les langages de .Net : il est ainsi possible de créer une classe dans un langage, de la compiler et de dériver cette classe dans un autre langage de .Net.

Le SDK de .Net fourni pour chaque langage un compilateur, par exemple csc.exe pour C#

Exemple

```
csc.exe /out :MonApp.exe fichierSource.cs
```

123.3. Les environnements d'exécution

Le code Java et .Net est compilé dans un langage intermédiaire respectivement Bytecode et MSIL (Microsoft Intermediate Language) et est exécuté dans un environnement d'exécution respectivement JVM (Java Virtual Machine) et CLR (Common Langage Runtime). La compilation dans un langage indépendant de tous systèmes n'est pas une nouveauté : par exemple, elle était déjà mise en oeuvre dans Smalltalk.

.Net et Java compilent leur code source dans un langage indépendant de tout système et de tout hardware : ce langage intermédiaire (.Net) ou bytecode (Java) est exécuté par interprétation (Java) ou compilation à la volée (compilation JIT : Just In Time) par l'environnement d'exécution de la plate-forme (.Net et Java).

En Java, chaque classe peut implémenter une méthode main avec une signature spécifique qui peut être autant de points d'entrée pour l'application. Il faut préciser à la commande java le nom de la classe qui contient la méthode main à exécuter.

En C#, le point d'entrée de l'application peut être précisé avec l'option /main:maClasse du compilateur.

123.3.1. Les machines virtuelles

Le pseudo code produit à la compilation est exécuté par une machine virtuelle : JVM en Java et CLR en .Net

CLR et JVM mettent en oeuvre un compilateur Just In Time et une gestion de la mémoire grâce à un ramasse-miettes (garbage collector) pour les objets créés dans le tas (heap).

Il est possible en Java de gérer la taille de la mémoire allouée à la JVM. Ceci n'est pas possible avec le CLR.

123.3.2. Le ramasse-miettes

Le CLR et la JVM propose tous les deux un mécanisme de libération automatique de la mémoire des objets inutilisés : le ramasse-miettes (garbage collector).

Les environnements Java et .Net assurent donc leur propre gestion de la mémoire en utilisant un ramasse-miettes. Le ramasse-miettes a pour rôle de rechercher les objets inutilisés et de libérer leur espace mémoire, évitant ainsi aux développeurs d'avoir à le faire explicitement.

Sur les deux plates-formes le ramasse-miettes facilite le travail du développeur. Cependant les fuites de mémoire existent et peuvent être particulièrement difficiles à corriger du fait même que les développeurs se désengagent de la gestion de mémoire ou méconnaissent les mécanismes sous-jacents. Généralement, ces fuites de mémoire sont difficiles à corriger sans l'utilisation d'un profiler qui permet d'inspecter le contenu de la mémoire.

123.4. Le déploiement des modules

Avec .Net, le code est compilé et packagé dans une assembly.

Le SDK de .Net propose l'outil al.exe (Assembly Linker) pour créer des assemblies.

Une assembly peut être physiquement un .exe ou .dll : elle contient le code compilé, des ressources (icônes, images, texte, ...) et des métadonnées.

Ces métadonnées correspondent, entre autres, à un numéro de version qui permet au CLR de charger dynamiquement la bonne version sans avoir à enregistrer l'assembly dans la base de registres. Il est ainsi possible d'avoir deux versions d'une assembly dans un même répertoire.

Le déploiement d'une application .Net peut alors se résumer à une simple copie de fichiers.

Java propose plusieurs formats de packaging des modules selon le type de composants qu'il contient : jar (pour les applications ou les bibliothèques), ear (pour les applications d'entreprises), war (pour les applications web), ...

123.5. Les version des modules

Fort de l'expérience du DLL Hell, Microsoft a réussi à proposer une gestion des modules qui soit un point fort de .Net car elle offre une simplicité d'utilisation tout en étant puissante en permettant, par exemple, d'utiliser plusieurs versions d'une assembly dans un même domaine d'application.

Cette gestion repose sur le stockage dans l'assembly de metadonnées qui contiennent le nom mais aussi le numéro de version de l'assembly.

En Java, la gestion des modules peut rapidement devenir problématique essentiellement à cause des classloaders qui par défaut parcourent de façon linéaire le classpath pour rechercher une classe dans le premier module trouvé.

Pourtant, le mécanisme des classloaders est puissant : il est utilisé par la JVM mais il l'est aussi, par exemple, pour isoler les bibliothèques des applications web exécutées dans un conteneur web ou pour permettre le chargement dynamique de modules par OsGI.

123.6. L'interopérabilité inter-languague

Java propose l'API JNI (Java Native Interface) pour permettre l'exécution de code natif dans la JVM.

Le mot clé native est utilisé pour marquer une méthode contenant du code natif.

L'outil javah permet de générer des fichiers header qui devront être exploités pour produire le code qui sera compilé en bibliothèque native.

Historiquement, la plate-forme Java propose aussi un support pour Corba.

En .Net, l'invocation d'objets COM est intégrée dans la plate-forme, ce qui facilite la réexploitation de code existant.

L'invocation de code natif sous la forme d'objets COM est simplifiée grâce à des outils du SDK (tlbimp et tlbexp) qui génèrent des proxys.

En C#, il est possible d'utiliser directement une DLL grâce au mot clé extern et à l'attribut DllImport, ce qui offre un moyen très simple de réutiliser des fonctionnalités existantes tant que celles-ci sont des DLL Windows.

Dans les deux cas, l'exécution de code natif ne permet pas aux machines virtuelles de gérer la mémoire des objets natifs.

L'intéropérabilité des deux plates-formes est possible facilement en utilisant les services web. Un projet commun vise à favoriser l'interopérabilité de ces services entre le framework WCF de .Net et Metro qui est l'implémentation de référence de JAX-WS.

123.7. La décompilation

Comme Java et .Net compilent leur code source dans un langage dédié, il est possible de décompiler ce langage intermédiaire pour obtenir le code source.

Il existe des solutions d'obfuscation proposées par des tiers pour les deux plates-formes.

Le SDK de .Net propose un outil graphique pour réaliser cette décompilation : ildasm (intermediate Langage Desasambler).

Java propose l'outil javap en ligne de commandes pour visualiser le bytecode.

123.8. Les API des deux plates-formes

Un langage est moins utile sans un ensemble de bibliothèques qui fournissent les fonctionnalités de base pour créer des applications. Ces fonctionnalités peuvent couvrir de nombreux sujets : threads, entrées/sorties, réseaux, collections, utilitaires, manipulations XML, accès aux bases de données, composants graphiques, ...

Les plates-formes Java et .Net fournissent un ensemble très riche d'API

	Java	**.NET**
Développement Web	API de bas niveau : Servlet, JSP JSTL, Java Server Faces Nombreux frameworks open source (exemple : Struts, Tapestry, Wicket, Spring MVC, ...)	ASP.Net
GUI	Swing (ou AWT) SWT est une alternative possible	Win Forms WPF (Windows Presentation Framework) depuis .Net 3.0
Accès aux données	JDBC	ADO.Net

	Java	**.NET**
Appels d'objets distants	RMI	Remoting
Ajax	Partiellement intégré dans JSF 2.0 Nombreux frameworks open source (exemple : DWR, ...)	Atlas proposé indépendamment par Microsoft puis intégré dans ASP .Net
Services Web	JAX-RPX, JAX-WS, JAX-RS Nombreuses solutions tiers (Axis, ...)	ASP.Net
XML	JAXP (manipulation de document XML) JAXB (binding document XML/ objets Java) SAAJ (mise en oeuvre de SOAP) JAXR (utilisation des registres UDDI et ebXML)	Intégré dans la plate-forme WCF (Windows Communication Framework) depuis .Net 3.0
Sécurité	JAAS JCE	Intégré dans la plate-forme
Déploiement d'applications à travers le réseau	Java Web start	One click
Interaction avec du code natif	JNI	la plate-forme est capable de prendre en compte du code managé (MSIL) et non managé (natif) avec des interactions utilisant COM
Objets métiers	EJB	
Mapping O/R	JDO, JPA Nombreux frameworks open source (exemple : hibernate, ...)	Entity Framework Quelques frameworks pour la plupart portés de Java (NHibernate, ...)
Manipulation dynamique des objets	java.lang.reflect	System.Reflection
Gestion des événements	Listener	Delegate, Event
Documentation technique	Javadoc : commentaires particuliers avec des tags dédiés @xyz	Support uniquement en C# avec des tags XML dans des commentaires particuliers

En terme d'API, .Net propose une unique API fournie en standard dans la plate-forme ce qui n'oblige à aucun choix majeur. Quelques API open source se développent : ce sont essentiellement des portages d'API Java populaires (Log4net, Spring.Net, Nhibernate, ...)

Java propose généralement une ou plusieurs API qui sont des spécifications. Une implémentation de référence est proposée et ces API sont généralement implémentées par des tiers commerciaux ou open source. Ceci permet d'avoir le choix et ne pas être tributaire d'un seul fournisseur. Mais ce choix est d'autant plus complexe qu'en plus des API standards, de nombreux frameworks, notamment open source, proposent des alternatives particulièrement intéressantes (Struts, Spring, Hibernate, Log4J, ...)

123.8.1. La correspondance des principales classes

Attention : la correspondance entre les classes fournies est rarement parfaite et généralement les classes sont seulement proches ou similaires dans leur fonctionnement. Une consultation des documentations respectives des deux plates-formes est nécessaire lors du portage du code de l'une à l'autre.

123.8.1.1. La correspondance des classes de bases

Chaque type de base possède une représentation sous la forme d'un wrapper objet.

Java	.Net
java.lang.Boolean	System.Boolean
java.lang.Byte	System.Byte
java.lang.Character	System.Char
java.lang.Class	System.Type
java.lang.Double	System.Double
java.lang.Float	System.Single
java.lang.Integer	System.Int32
java.lang.Long	System.Int64
java.lang.Math	System.Math
java.lang.Object	System.Object
java.lang.Process	System.Diagnostics.Process
java.lang.Runtime	System.Diagnostics.Process
java.lang.Short	System.Int16
java.lang.StrictMath	System.Math
java.lang.String	System.String
java.lang.StringBuffer	System.Text.StringBuilder
java.lang.Thread	System.Threading.Thread
java.lang.Throwable	System.Exception

123.8.1.2. La correspondance des classes utilitaires

Java et .Net proposent un ensemble de classes utilitaires de base.

Java	.Net
java.util.BitSet	System.Collections.BitArray
java.util.Calendar	System.Globalization.Calendar
java.util.Currency	System.Globalization.RegionInfo
java.util.Date	System.DateTime
java.util.EventObject	System.EventArgs
java.util.GregorianCalendar	System.Globalization.GregorianCalendar
java.util.ListResourceBundle	System.Resources.ResourceManager
java.util.Locale	System.Globalization.CultureInfo
java.util.Random	System.Random
java.util.ResourceBundle	System.Resources.ResourceSet
java.util.SimpleTimeZone	System.DateTime
java.util.Timer	System.Threading.Timer
java.util.TimerTask	System.Threading.TimerCallback

java.util.TimeZone	System.DateTime

123.8.2. Les collections

Les deux plates-formes offres des fonctionnalités similaires en proposant des interfaces, des classes abstraites et des implémentations concrètes de classes permettant de manipuler des collections.

.Net et Java proposent une bibliothèque facilitant la mise en oeuvre de collections de données sous différentes formes (tableaux, listes chaînées, ensembles, ...)

.Net regroupe les API de gestions de collections dans le namespace System.Collections qui contient notamment :

- Des interfaces et classes abstraites : ICollection, IEnumerable, IDictionary, IList, CollectionBase
- Des implémentations concrètes : ArrayList, SortedList, Stack, Queue

Les éléments des collections .Net peuvent être accédés par des indexeurs dont la syntaxe est similaire à celle utilisée pour accéder à un élément d'un tableau.

.NET 2.0 propose le namespace System.Collections.Generic qui propose des collections mettant en oeuvre les generics.

L'API Collection de Java est incluse dans le package java.util.

Java offre un nombre plus important d'implémentations de collections notamment avec les ensembles (Set) et les listes chaînées (LinkedList). De plus, les fonctionnalités offertes sur ces collections sont plus nombreuses en Java.

Les fonctionnalités de la classe statique Java Collections (Reverse(), Sort(), BinarySearch(), ...) sont utilisables directement dans les classes d'implémentations de .Net.

Attention en .Net, par défaut les collections ne sont pas synchronisées. En Java, certaines collections notamment les plus anciennes (Vector, Hashtable, ...) sont synchronisées par défaut.

L'interface System.Collections.ICollection est similaire à l'interface java.util.Collection. Les interfaces Ilist et IDictionary de .Net sont similaires aux interfaces java.util.List et java.util.Map.

Java	.Net
java.util.AbstractCollection	System.Collections.CollectionsBase
java.util.ArrayList	System.Collections.ArrayList
java.util.Arrays	System.Array
java.util.Dictionary	System.Collections.DictionaryBase
java.util.HashMap	
java.util.Hashtable	System.Collections.Hashtable
java.util.Set	
java.util.Stack	System.Collections.Stack
java.util.TreeSet	System.Collections.SortedList
java.util.Vector	System.Collections.ArrayList

123.8.3. Les entrées/sorties

Les plates-formes Java et C# proposent un ensemble complet de classes pour manipuler des flux de données textuelles ou binaires, entrants ou sortants.

L'exemple ci-dessous copie le contenu d'un fichier texte.

Exemple C# :

```
using System;
using System.IO;

public class CopieFichierTexte
{
    public static void Main(string[] args)
    {
        FileStream inputFile  = new FileStream("source.txt", FileMode.Open);
        FileStream outputFile = new FileStream("copie.txt", FileMode.Open);
        StreamReader sr     = new StreamReader(inputFile);
        StreamWriter sw     = new StreamWriter(outputFile);
        String str;
        while((str = sr.ReadLine())!= null)
            sw.Write(str);
        sr.Close();
        sw.Close();
   }
}
```

Exemple Java :

```
import java.io.*;

public class CopieFichierTexte{

  public static void main(String[] args) throws IOException {
    File inputFile  = new File("source.txt");
    File outputFile = new File("copie.txt");
    FileReader in     = new FileReader(inputFile);
    BufferedReader br = new BufferedReader(in);
    FileWriter out    = new FileWriter(outputFile);
    BufferedWriter bw = new BufferedWriter(out);
    String str;
    while((str = br.readLine())!= null)
      bw.write(str);
    br.close();
    bw.close();
  }
}
```

Ces exemples de code sont similaires hormis le fait que Java ne travaille pas avec des tampons par défaut.

123.8.3.1. La correspondance des classes pour gérer les entrées/sorties

Le tableau ci-dessous propose une correspondances entre les classes des deux plate-formes relatives aux entrées/sorties.

Java	.Net
java.io.BufferedInputStream	System.IO.BufferedStream
java.io.BufferedOutputStream	System.IO.BufferedStream
java.io.BufferedReader	System.IO.StreamReader
java.io.BufferedWriter	System.IO.StreamWriter
java.io.ByteArrayInputStream	System.IO.MemoryStream
java.io.ByteArrayOutputStream	System.IO.MemoryStream
java.io.CharArrayReader	System.IO.StreamReader
java.io.CharArrayWriter	System.IO.StreamWriter
java.io.DataInputStream	System.IO.BinaryReader

java.io.DataOutputStream	System.IO.BinaryWriter
java.io.File	System.IO.File
java.io.FileInputStream	System.IO.FileStream
java.io.FileOutputStream	System.IO.FileStream
java.io.FileReader	System.IO.StreamReader
java.io.FileWriter	System.IO.StreamWriter
java.io.InputStream	System.IO.Stream
java.io.OutputStream	System.IO.Stream
java.io.PrintStream	System.IO.StreamWriter
java.io.PrintWriter	System.IO.StreamWriter
java.io.PushbackInputStream	System.IO.StreamReader
java.io.PushbackOutputStream	System.IO.StreamReader
java.io.RandomAccessFile	System.IO.FileStream
java.io.StringBufferInputStream	System.IO.StringReader
java.io.StringReader	System.IO.StringReader
java.io.StringWriter	System.IO.StringWriter

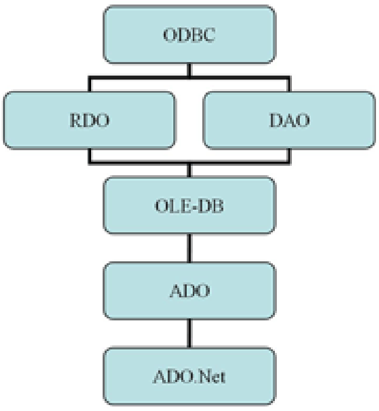
123.8.4. L'accès aux bases de données

Java et .Net propose en standard des API pour l'accès aux bases de données de bas niveau ou de type ORM.

123.8.4.1. Les API de bas niveau

Java propose l'API JDBC qui est une abstraction de l'accès aux bases de données grâce aux spécifications de l'API et à leur implémentation au travers de pilotes (Driver) fournis par des tiers. Ces pilotes peuvent être de 4 types dont un fourni en standard qui permet une utilisation d'ODBC.

.Net propose l'accès aux bases de données aux travers de l'API ADO.Net qui est le résultat des évolutions des API d'accès aux données proposées par Microsoft :



ADO.Net est une spécification qui nécessite une implémentation sous la forme de Providers. La plate-forme propose en standard deux Providers : un pour SQL Server (System.Data.SqlClient) ou un pour OLE-DB (System.Data.OleDb). Il est possible d'utiliser d'autres Providers fournis par des tiers.

Ce mode de fonctionnement impose d'avoir un code dépendant du Provider utilisé : pour une utilisation de plusieurs bases de données, il est nécessaire de développer sa propre abstraction sous la forme d'un DAL (Data Acces Layer) en utilisant notamment le design pattern factory.

Les informations de connexions et leurs paramètres de configuration en ADO.Net sont fournis par une chaîne de caractères contenant ces informations sous la forme clé=valeur. JDBC utilise une url dépendante du pilote pour préciser les informations de connexion.

Les deux API proposent les modes de fonctionnement connecté (ResultSet pour JDBC et DataReader pour ADO.Net) et déconnecté (depuis JDBC 3, RowSet avec une implémentation offrant cette fonctionnalité comme CachedRowSet et DataSet pour ADO.Net).

Depuis la version 2 de JDBC, il est possible de parcourir un ResultSet dans les deux sens (si le pilote le permet) ; ceci est aussi possible avec un RowSet.

Les Providers ADO.NET fournis en standard mettent automatiquement en oeuvre un pool de connexions. En Java, l'utilisation d'un pool de connexions doit être explicite et mise en oeuvre avec la classe DataSource, généralement stockée dans un annuaire et accédée avec JNDI.

Depuis sa version 2, JDBC permet de regrouper plusieurs opérations de mises à jour (batchUpdate).

Java	.Net
java.sql.Blob	System.Data.SqlClient.SqlDataReader System.Data.OleDb.OleDbDataReader
java.sql.CallableStatement	System.Data.SqlClient.SqlCommand System.Data.OleDb.OleDbCommand
java.sql.Clob	System.Data.SqlClient.SqlDataReader System.Data.OleDb.OleDbDataReader
java.sql.Connection	System.Data.SqlClient.Sql System.Data.OleDb.OleDb
java.sql.Date	System.Data.SqlTypes.SqlDateTime
java.sql.ParameterMetaData	System.Data.SqlClient.SqlParameter System.Data.OleDb.OleDbParameter
java.sql.PreparedStatement	System.Data.SqlClient.SqlCommand System.Data.OleDb.OleDbCommand
java.sql.ResultSet	System.Data.SqlClient.SqlDataReader System.Data.OleDb.OleDbDataReader
java.sql.ResultSetMetaData	System.Data.SqlClient.SqlDataReader System.Data.OleDb.OleDbDataReader
java.sql.Savepoint	System.Data.SqlClient.SqlTransaction
java.sql.Statement	System.Data.SqlClient.SqlCommand System.Data.OleDb.OleDbCommand
java.sql.Time	System.Data.SqlTypes.SqlDateTime
javax.sql.RowSet	System.Data.DataSet

123.8.4.2. Les frameworks de type ORM

Java propose deux solutions standards pour l'accès aux données au travers d'une API de type ORM :

- JDO (Java Data Object)
- JPA (Java Persistence API)

.Net propose le framework Entity.

123.8.5. Les interfaces graphiques

Tous les composants graphiques de base ont une correspondance dans les deux plate-formes.

.Net ne propose pas d'équivalence au look and feel de Swing puisque seule la plate-forme Windows est supportée nativement.

Java propose deux frameworks pour le développement d'interfaces graphiques : AWT et Swing. Historiquement, AWT est le plus ancien et repose sur des composants natifs de l'environnement d'exécution ce qui limite le nombre de composants disponibles. Swing utilise des composants légers écrits en Java qui possèdent une toute petite partie native. Swing propose donc des composants plus nombreux et plus riches.

Java	.Net
javax.swing.AbstractButton	System.Windows.Forms.ButtonBase
javax.swing.AbstractListModel	System.Windows.Forms.ListControl
javax.swing.AbstractSpinnerModel	System.Windows.Forms.UpDownBase
javax.swing.ImageIcon	System.Windows.Forms.Image
javax.swing.JButton	System.Windows.Forms.Button
javax.swing.JCheckBox	System.Windows.Forms.CheckBox
javax.swing.JColorChooser	System.Windows.Forms.ColorDialog
javax.swing.JComboBox	System.Windows.Forms.ComboBox
javax.swing.JComponent	System.Windows.Forms.UserControl
javax.swing.JDialog	System.Windows.Forms.CommonDialog
javax.swing.JEditorPane	System.Windows.Forms.TextBoxBase
javax.swing.JFileChooser	System.Windows.Forms.OpenFileDialog
javax.swing.JFormattedTextField	System.Windows.Forms.RichTextBox
javax.swing.JFrame	System.Windows.Forms.Form
javax.swing.JLabel	System.Windows.Forms.Label
javax.swing.JList	System.Windows.Forms.ListBox
javax.swing.JMenuBar	System.Windows.Forms.MainMenu
javax.swing.JMenuItem	System.Windows.Forms.MenuItem
javax.swing.JPanel	System.Windows.Forms.Panel
javax.swing.JPasswordField	System.Windows.Forms.TextBox
javax.swing.JPopupMenu	System.Windows.Forms.ContextMenu
javax.swing.JProgressBar	System.Windows.Forms.StatusBar
javax.swing.JRadioButton	System.Windows.Forms.RadioButton
javax.swing.JScrollBar	System.Windows.Forms.HScrollBar System.Windows.Forms.VScrollBar
javax.swing.JScrollPane	System.Windows.Forms.Panel
javax.swing.JSlider	System.Windows.Forms.TrackBar
javax.swing.JSpinner	System.Windows.Forms.DomainUpDown
javax.swing.JSplitPane	System.Windows.Forms.Splitter
javax.swing.JTabbedPane	System.Windows.Forms.TabControl
javax.swing.JTable	System.Windows.Forms.ListView

javax.swing.JTextArea	System.Windows.Forms.TextBox
javax.swing.JTextField	System.Windows.Forms.TextBox
javax.swing.JTextPane	System.Windows.Forms.RichTextBox
javax.swing.JToggleButton	System.Windows.Forms.ButtonBase
javax.swing.JToolBar	System.Windows.Forms.ToolBar
javax.swing.JToolTip	System.Windows.Forms.ToolTip
javax.swing.JTree	System.Windows.Forms.ListView

La gestion des événements en Java se fait avec des listeners. En .Net, elle se fait avec des events et des delegates.

123.8.6. Le développement d'applications web

123.8.6.1. Les APIs de bas niveau

En Java, les développements web reposent sur les servlets. Les JSP sont une abstraction de plus haut niveau permettant la réalisation de la partie graphique d'une application web. Les JSP sont compilées de façon transparente en servlets.

123.8.6.2. Les frameworks

En .Net, les développements d'applications web se font avec le framework ASP.Net.

En standard, les applications web en Java utilisent les JSF (Java Server Faces).

Il existe de nombreux frameworks open source (Struts, Spring MVC, Tapestry, Wicket, ...)

La suite de cette section sera développée dans une version future de ce document

123.8.7. Le développement d'applications de type RIA

.Net propose Silverlight

Historiquement, Java a proposé les applets qui sont des applications Java s'exécutant dans une page web du navigateur. Par défaut, l'utilisation d'une applet présente de nombreuses restrictions notamment pour garantir la sécurité (exécution dans une sandbox, pas d'accès au système, communication uniquement avec le serveur depuis lequel la servlet est téléchargée, ...)

Java propose aussi JavaFX.

124. Java et C#

Chapitre 124

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

En l'an 2000, Microsoft dévoile sa nouvelle plate-forme de développement nommée .Net et un nouveau langage de développement dédié : C#. C# est un des langages utilisables pour développer des applications de tous types (standalone, web, mobile, ...) pour la plate-forme .Net de Microsoft.

Java et C# partagent un ensemble de fonctionnalités communes :

- Compilation dans un langage intermédiaire indépendant de la machine et exécution dans un environnement dédié (une machine virtuelle)
- Gestion automatique de la mémoire grâce à un ramasse-miettes
- Introspection pour manipuler dynamiquement les objets
- Toutes les classes héritent d'une même classe (Object) et sont allouées sur le tas
- Pas de support de l'héritage multiple mais utilisation d'interfaces
- Tout doit être encapsulé dans une classe : il n'existe pas de fonctions ou constantes globales
- Gestion des erreurs grâce aux exceptions
- ...

Java et C# possèdent cependant des différences :

- Java possède des exceptions vérifiées (traitement ou déclaration de la propagation de ces exceptions)
- C# propose le mode unsafe qui permet de manipuler la mémoire (hors du contrôle de l'environnement managé)
- C# propose les propriétés, les indexeurs, les délégués et les événements
- C# propose la surcharge des opérateurs
- C# possède les structures qui sont des types valeurs
- La génération de la documentation est différente : Javadoc génère une documentation au format HTML, C# génère des fichiers XML
- C# propose l'instruction goto
- ...

C# et Java possèdent chacun des points forts dont certains sont communs. Ces deux langages se distinguent cependant sur de nombreux points particuliers. Le but de ce chapitre n'est pas de les comparer pour déterminer quel serait le « meilleur » mais simplement de recenser une partie de leurs points communs et de leurs différences ceci afin de faciliter le passage de l'un à l'autre et vice versa.

Le contenu de ce chapitre n'est pas exhaustif et propose simplement d'aborder les points principaux. Il concerne essentiellement Java 6 et C# 2.0.

Chaque langage possède des fonctionnalités importantes qui lui sont propres notamment :

- En C# : passage par référence, delegates, événements, indexeurs, objets de type valeur, surcharge des opérateurs, pointeurs (dans des portions de code non managée), préprocesseur, l'instruction goto, ...
- En Java : la portabilité inter plate-forme, les exceptions vérifiées (checked exception), les classes internes anonymes, la documentation automatique du code (Javadoc), ...

Il est intéressant de remarquer que certaines fonctionnalités d'un des langages sont incorporées dans l'autre et vice versa au fur et à mesure de leurs nouvelles versions (exemple : les fonctionnalités boxing/unboxing, enumération, parcours de collections et les generics ajoutés dans Java 5.0, les classes anonymes ajoutées dans C# 2.0 ou les classes anonymes internes ajoutées dans C# 3.0).

Ce chapitre contient plusieurs sections :

- ♦ La syntaxe
- ♦ La programmation orientée objet
- ♦ Les chaînes de caractères
- ♦ Les tableaux
- ♦ Les indexeurs
- ♦ Les exceptions
- ♦ Le multitâche
- ♦ L'appel de code natif
- ♦ Les pointeurs
- ♦ La documentation automatique du code
- ♦ L'introspection/reflection
- ♦ La sérialisation

124.1. La syntaxe

La syntaxe de Java et C# sont relativement proches puisqu'elles dérivent pour les deux de celle du langage C :

- ils sont tous les deux sensibles à la casse
- un bloc de code est défini entre accolade
- les instructions de base pour les boucles et les conditions dans les traitements sont similaires
- ...

124.1.1. Les mots clés

Il y a énormément de similitudes entre les mots clés des deux langages, presque tous les mots-clés Java ont un équivalent en C# à part quelques exceptions telles que transient, throws ou strictfp. C# possède de nombreux mots clés sans équivalent en Java. Le tableau ci-dessous recense les mots clés des deux langages avec leur équivalence en Java (même si leur rôle n'est pas toujours exactement le même).

C#	Java	C#	Java	C#	Java	C#	Java
abstract	abstract	false	false	override		typeof	
as		finally	finally	params		uint	
base	super	fixed		partial (C# 2)		ulong	
bool	boolean	float	float	private	private	unchecked	
break	break	for	for	protected		unsafe	
byte		foreach	for (java 5)	public	public	ushort	
case	case	get		readonly		using	import
catch	catch	goto		ref		value	
char	char	if	if	return	return	virtual	
checked		implicit		sbyte	byte	volatile	volatile
class	class	in		sealed	final	void	void
const		int	int	set		where (C# 2)	
continue	continue	interface	interface	short	short	while	while
decimal		internal	protected	sizeof		yield (C# 3)	

default	default	is	instanceof	stackalloc		:	extends
delegate		lock	synchronized	static	static	:	implements
do	do	long	long	string			assert (java 4)
double	double	namespace	package	struct			strictfp
else	else	new	new	switch	switch		throws
enum	enum (java 5)	null	null	this	this		transient
event		object		throw	throw		
explicit		operator		true	true		
extern	native	out		try	try		

Remarque : les équivalences entre les mots clés Java et C# du tableau ci-dessus ne sont pas toujours parfaites mais les deux mots présentent des similitudes plus ou moins importantes dans leur principal rôle.

Java définit const et goto dans ses mots clés mais ne les utilise pas dans sa syntaxe pour le moment.

C# propose des raccourcis syntaxiques, par exemple :

Mot clé	Remplacé à la compilation par
String ou string	System.String
Object ou object	System.Object

Le mot clé string de C# est donc un alias sur le type correspondant String de la plate-forme .NET. Java utilise la classe java.lang.String mais Java ne définit aucun mot clé pour cette classe.

124.1.2. L'organisation des classes

Pour organiser les classes, Java utilise le concept de packages et C# utilise le concept de Namespaces. La grande différence est qu'avec Java le nom du packages impose une structure de répertoires correspondante. Avec C#, les namespaces sont purement indicatifs.

En Java, un fichier ne peut donc appartenir qu'à un seul package alors qu'en C#, un fichier peut déclarer plusieurs namespaces.

L'utilisation de cette liberté dans la mise en oeuvre des namespaces n'est cependant pas recommandée et il est préférable de s'imposer quelques règles simples pour se faciliter la tâche notamment dans de gros projets.

Le contenu d'un fichier source est soumis à quelques contraintes en Java : le nom du fichier doit correspondre à la casse près au nom de l'unique classe publique qu'il contient. Il est possible de définir d'autres classes dans le fichier mais elles ne peuvent pas être public. En C#, il n'y a aucune contrainte : le nom du fichier est libre et peut contenir plusieurs classes publiques.

Là encore, l'utilisation de cette fonctionnalité n'est pas recommandée. Il est préférable de s'imposer quelques règles simples pour se faciliter la tâche notamment dans de gros projets.

En C#, les éléments du code source (classes, structs, delegates, enums, ...) sont organisés en fichiers, namespaces et assemblies.

Un namespace permet de regrouper des éléments du code de façon similaire au package en Java. Cependant, un package définit une structure physique de répertoires qui correspond au nom du package. Un namespace définit uniquement une structure logique.

L'utilisation des éléments d'un namespace se fait avec l'instruction using.

Exemple C# :

```
namespace fr.jmdoudoux.dej
{
    public class MaClasse
    {
        public void MaClasse()
        {
        }
    }
}
```

Exemple Java :

```
package fr.jmdoudoux.dej;

public class MaClasse {

  public void maMethode() {
  }
}
```

En C#, il est possible d'imbriquer plusieurs namespaces.

Exemple C# :

```
namespace fr.jmdoudoux.dej
{
    public class MaClasse
    {
        public void MaMethode()
        {
        }
    }

    namespace fr.jmdoudoux.dej.donnees
    {
        public class MaDonnee
        {
            public void Afficher()
            {
            }
        }
    }
}
```

Une assembly est l'unité de packaging fondamentale de .Net : elle regroupe des classes compilées en MSIL, des métadonnées et des ressources. Une assembly peut contenir plusieurs namespaces et un namespace peut être réparti sur plusieurs assemblies. Un numéro de version est géré au niveau de l'assembly.

Un fichier JAR est au format zip alors qu'une assembly peut être au format exe ou dll.

Java propose aussi des packagings dédiés pour certains types d'applications : ear (pour les applications d'entreprises), war pour les applications web, ...

124.1.3. Les conventions de nommage

C# et Java sont sensibles à la casse. Ils proposent tous les deux leurs propres conventions de nommage pour les entités mises en oeuvre dans le code source. Leur utilisation n'est pas obligatoire mais elles sont communément appliquées.

La suite de cette section sera développée dans une version future de ce document

124.1.4. Les types de données

Java et C# sont tous les deux des langages orientés objets : ils proposent donc des types primitifs et des types objets pour les données. C# propose en plus le type valeur.

124.1.4.1. Les types primitifs

Chaque type primitif Java possède un type équivalent de même nom en C# sauf byte qui est signé en Java et correspond donc au type sbyte en C#.

C# possède en plus des types numériques non signés (byte, ushort, uint, ulong).

C# propose aussi le type decimal qui permet de stocker un nombre décimal avec une moins grande capacité mais une meilleure précision et sans erreur d'arrondi.

Les types primitifs de .Net sont définis dans le CTS (Common Type System)

C#			Java		
Type	taille	valeurs	type	taille	valeurs
Byte System.Byte	8	0 à 255			
Sbyte System.Sbyte	8	-128 à 127	byte	8	
Short System.Int16	16	-32768 à 32767	short	16	
Ushort System.UInt16	16	0 à 65635			
Int System.Int32	32		int	32	-2147483648 à 2147483647
Uint System.UInt32	32				
Long System.Int64	64		long		-9223372036854775808 à 9223372036854775807
Ulong System.Uint64	64				
Float System.Single	32		float	32	1.401e-045 à 3.40282e+038
Double System.Double	64		double	64	2.22507e-308 à 1.79769e+308
Decimal System.Decimal	96				

C# utilise le suffixe f pour les valeurs de type float, d pour les valeurs de type double (suffixe par défaut) et m pour les valeurs de type décimal.

124.1.4.2. Les types objets

Java et C# proposent tous les deux la définition et la manipulation d'objets. Leur mise en oeuvre est détaillée dans une section dédiée.

124.1.4.3. Les types valeur (ValueTypes et Structs)

C# permet au travers des structures (structs) de stocker des objets dans la pile plutôt que sur le tas. Ces objets sont nommés type valeur (ValueType) par opposition au type référence. Les types valeurs sont donc stockés dans la pile : ils sont toujours passés par valeur et ne sont pas gérés par le ramasse-miettes.

L'utilisation de la pile par rapport au tas possède plusieurs avantages : la gestion mémoire est plus rapide et sans fragmentation.

En C#, tous les types primitifs sont encapsulés dans des structures qui héritent de ValueType.

Tous les objets de type struct héritent implicitement de la classe object et ne peuvent pas hériter d'une autre classe mais peuvent implémenter des interfaces. L'héritage n'est pas supporté pour les types struct. Ainsi les types struct sont implicitement marqués avec le modificateur sealed et ne peuvent donc pas être abstraits.

Exemple C# :

```
using System;

namespace TestCS
{
   struct Position
   {
      private int coordX;

      public int CoordX
      {
         get { return coordX; }
         set { coordX = value; }
      }
      private int coordY;

      public int CoordY
      {
         get { return coordY; }
         set { coordY = value; }
      }

      public Position(int x, int y)
      {
         this.coordX = x;
         this.coordY = y;
      }

      public override string ToString()
      {
         return String.Format("(Position = {0}, {1})", coordX, coordY);
      }
   }
}
```

124.1.5. La déclaration de constantes

Java définit une constante grâce au mot clé final : elle ne peut être initialisée qu'une seule fois de façon statique (à la compilation) ou dynamique (à l'exécution uniquement dans un constructeur). Une fois la valeur affectée, elle ne peut plus être modifiée.

Exemple Java :

```
public class TestFinal {

  public final int valeurA;
  public final int valeurB = 20;

  public TestFinal() {
    valeurA = 10;
  }

  public static void main(String[] args) {
    TestFinal f = new TestFinal();
    System.out.println("valeurA="+f.valeurA);
    System.out.println("valeurB="+f.valeurB);
  }
}
```

C# utilise le mot clé const pour définir une constante statique (valorisation à la compilation) et le mot clé readonly pour une constante dynamique (à l'exécution uniquement dans un constructeur ou à l'initialisation de la variable).

Exemple Java :

```
using System;

namespace TestCS
{
   class TestConstantes
   {
      const int i = 10;
      readonly int j;

      public TestConstantes(int valeur)
      {
         j = valeur;
         Console.Out.WriteLine("i="+i);
         Console.Out.WriteLine("j="+j);
      }
   }
}
```

124.1.6. Les instructions

Java et C# proposent un jeu d'instructions similaire. Quelques différences existent notamment avec l'instruction switch et goto.

124.1.6.1. L'instruction switch

La syntaxe de l'instruction switch est similaire en Java et C#. Cependant C# possède deux différences :

- il permet l'utilisation de type à valeurs finies (int, char, enum, ...) mais aussi de chaînes de caractères
- il ne permet pas d'omettre l'instruction break en fin du bloc défini par l'instruction case (fall through) si celui-ci contient au moins une instruction. Java offre cette possibilité qui est source d'erreur mais permet l'exécution d'un même bloc de code pour plusieurs valeurs différentes.

Exemple C# :

```
      public static string formatterValeur(string code)
      {
         string resultat = "";
         switch(code)
         {
            case "A":
            case "B":
               resultat = "Bien";
               break;
            case "C":
               resultat = "Mauvais";
               break;
            default :
               resultat = "Inconnu";
               break;
         }
         return resultat;
      }
```

124.1.6.2. L'instruction goto

Goto est un mot clé Java, mais il ne correspond pas à une instruction du langage : c'est uniquement un mot réservé pour le moment.

C# propose l'instruction goto qui permet de débrancher l'exécution du code vers une étiquette (label) définie.

L'utilisation de cette instruction est cependant fortement déconseillée depuis de nombreuses années.

Exemple C# :

```
        private void button1_Click(object sender, EventArgs e)
        {
            int compteur = 0;
        incrementation:
            compteur++;
            listBox1.Items.Add("element "+compteur);
            if (compteur < 5)
            {
                goto incrementation;
            }
        }
```

124.1.6.3. Le parcours des collections de données

C# propose l'instruction foreach pour faciliter le parcours dans son intégralité d'une collection qui implémente l'interface System.Collections.IEnumerable.

Exemple C# :

```
         string[] donnees = { "element1", "element2", "element3", "element4" };

         foreach (string element in donnees)
            Console.WriteLine(element);
```

A partir de Java 1.5 la même fonctionnalité est proposée. Auparavant, pour parcourir un tableau java, il fallait utiliser une boucle pour itérer sur chaque élément du tableau.

Exemple Java :

```
package fr.jmdoudoux.dej;

public class TestParcoursTableau {

  public static void main(String[] args) {

    String[] donnees = { "element1", "element2", "element3", "element4" };

    for (int i = 0; i < donnees.length; i++) {
      System.out.println(donnees[i]);
    }
  }
}
```

Pour les collections, il fallait utiliser un objet de type Iterator.

Exemple Java:

```
package fr.jmdoudoux.dej;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class TestParcoursListe {

  public static void main(String[] args) {

    List donnees = new ArrayList();
    donnees.add("element1");
    donnees.add("element2");
    donnees.add("element3");
    donnees.add("element4");

    for (Iterator iter = donnees.iterator(); iter.hasNext();) {
      String element = (String) iter.next();
      System.out.println(element);
    }
  }
}
```

Java 1.5 propose une version différente de l'instruction for permettant de réaliser un parcours sur un ensemble de données.

Exemple Java :

```
package fr.jmdoudoux.dej;

public class TestParcoursTableauFor {

  public static void main(String[] args) {

    String[] donnees = { "element1", "element2", "element3", "element4" };

    for (String element : donnees) {
      System.out.println(element);
    }
  }
}
```

Dans les collections, il faut utiliser les generics pour typer leurs éléments.

Exemple Java :

```
package fr.jmdoudoux.dej;

import java.util.ArrayList;
import java.util.List;

public class TestParcoursListeFor {

  public static void main(String[] args) {

    List<String> donnees = new ArrayList<String>();
    donnees.add("element1");
    donnees.add("element2");
    donnees.add("element3");
    donnees.add("element4");

    for(String element : donnees) {
      System.out.println(element);
    }
  }
}
```

124.1.7. Les metadatas

C# propose un support des metadatas dans le langage avec les attributs.

Avant Java 5.0 les seules metadatas utilisables en Java étaient celles de l'outil Javadoc insérées dans des commentaires dédiés. Le seul tag utilisable par le compilateur est le tag @deprecated qui signale que la méthode ne devrait plus être utilisée et permet au compilateur d'afficher un avertissement. Tous les autres tags ne sont utilisés que pour la génération de la documentation.

Depuis la version 5.0, Java intègre les annotations dans le langage.

La suite de cette section sera développée dans une version future de ce document

124.1.8. Les énumérations

C# propose la définition d'énumérations en utilisant le mot clé enum. Par défaut, chaque membre de l'énumération a pour valeur un entier incrémenté pour chaque valeur, la première étant 0.

Exemple C# :

```
        public enum StatutOperation {
            ouverte,
            traitee,
            enAttente,
            fermee
        }
```

Il est possible de forcer le type de l'énumération et la valeur d'un de ses membres.

Exemple C# :

```
        public enum StatutOperation : byte
        {
            ouverte = 10,
            traitee = 20 ,
            enAttente = 30,
            fermee = 40
        }
```

Exemple C# :

```
        static void Main(string[] args)
        {

            StatutOperation statut = StatutOperation.ouverte |
            StatutOperation.enAttente;
            if ((statut & StatutOperation.enAttente) != 0)
            {
                Console.WriteLine("En attente");
            }
            Console.ReadLine();
        }
```

Java propose depuis la version 1.5 un support des énumérations en utilisant le mot clé enum. Les énumérations en Java sont converties par le compilateur en une classe.

Exemple Java :

```
public enum StatutOperation {
  ouverte,
  traitee,
  enAttente,
  fermee
}
```

Exemple Java :

```
public class Operation {
  private StatutOperation operation;
  private String libelle;

  public String getLibelle() {
    return libelle;
  }

  public StatutOperation getOperation() {
    return operation;
  }

  public Operation(String libelle) {
    super();
    this.libelle = libelle;
    this.operation = StatutOperation.ouverte;
  }

  public void Fermer()
  {
    this.operation = StatutOperation.fermee;
  }
}
```

124.1.9. Les délégués

La suite de cette section sera développée dans une version future de ce document

124.1.10. Les événements

La suite de cette section sera développée dans une version future de ce document

124.1.11. Le contrôle sur le débordement d'un downcast

C# permet de gérer ou ignorer le débordement de capacité lors d'un downcast (conversion vers un type plus petit). Les mots clés checked et unchecked permettent respectivement d'activer ou de désactiver ce contrôle dans le bloc de code qu'ils définissent.

Exemple C# :

```
class MaClasseChecked
{
   public MaClasseChecked()
   {
      int valeur = 1000;

      try
      {
         checked
         {
            byte b1 = (byte)valeur;
         }
      }
      catch (OverflowException ofe)
      {
         Console.Out.WriteLine(ofe.StackTrace);
      }

      try
      {
         unchecked
         {
            byte b2 = (byte)valeur;
         }
      }
      catch (OverflowException ofe)
      {
         Console.Out.WriteLine(ofe.StackTrace);
      }
   }
}
```

124.1.12. Les directives de précompilation

La suite de cette section sera développée dans une version future de ce document

124.1.13. La méthode main()

Les points d'entrée d'une application Java et C# sont la méthode main() en Java et Main() en C# d'une classe de cette application.

Exemple C# :

```
using System;

class MaClasse{

   public static void Main(String[] args){
     Console.WriteLine("Hello World");
   }
}
```

Exemple Java :

```
 class MaClasse{

   public static void main(String[] args){
     System.out.println("Hello World");
   }
}
```

Java propose d'avoir une méthode main() dans plusieurs classes de l'application. Lors de l'exécution de l'application, il suffit de fournir en paramètre de la JVM le nom de la classe à exécuter. Java permet donc d'avoir plusieurs points d'entrée dans une application.

La version compilée d'une application en C# ne peut avoir qu'un seul point d'entrée. Si plusieurs classes possèdent une méthode main, il est nécessaire de préciser la classe servant de point d'entrée grâce à l'option /main du compilateur.

Résultat :

```
C:\>csc /main:MaClass1 /out:MonApp.exe MaClasse1.cs MaClasse2.cs
```

Sous Visual Studio, dans l'onglet Application, il faut utiliser l'option "open" du menu contextuel sur les properties du projet et sélectionner la classe qui fait office de point d'entrée dans la liste déroulante « Startup Objet ».

124.2. La programmation orientée objet

La classe mère de tous les objets est java.lang.Object en Java et System.Object en .Net : elles possèdent des méthodes ayant des rôles similaires (toString()/ToString(), ...). La classe java.lang.Object propose en plus des méthodes utilisées lors de la synchronisation de threads (notify(), notifyAll() et wait()).

C# propose un alias pour la classe Object : le mot clé object est remplacé à la compilation par System.Object.

En Java et en C#, toutes les méthodes doivent être membre d'une classe.

Java et C# ne proposent pas l'héritage multiple au niveau des classes mais le permettent au niveau des interfaces.

Java et C# supportent l'héritage avec une syntaxe différente : Java utilise le mot clé extends pour l'héritage de classe et le mot clé implements pour l'implémentation d'interface. C# utilise la syntaxe du C++ pour définir l'héritage ou l'implémentation d'une interface.

Exemple Java :

```
public class MaClasseFille extends MaClasseMere implements Comparable {
...
}
```

Exemple C# :

```
public class MaClasseFille : MaClasseMere, IComparable
{
...
}
```

Pour empêcher le sous-classement d'une classe, il faut utiliser le mot clé final en Java et sealed en C#.

Java et C# permettent la définition de constructeurs static (permettant d'initialiser les variables statiques au chargement de la classe)

Exemple Java :

```
static {
    System.out.println(« initialisation »);
}
```

Exemple C# :

```
static MaClasse()
{
     Console.WriteLine(« initialisation »);
}
```

L'opérateur instanceof en Java et is en C# permettent de déterminer si une instance est du type précisé.

124.2.1. Les interfaces

Java et C# proposent la définition d'interfaces.

C# propose en plus l'implémentation explicite d'une interface. Ceci permet d'éviter les conflits de nom dans l'implémentation des méthodes de chaque interface.

Exemple C# :

```
   public interface IMonIterface1
   {
      void MaMethode();
   }

   public interface IMonInterface2
   {
      void MaMethode();
   }

   public class MonImplementation : IMonIterface1, IMonInterface2
   {
      void IMonIterface1.MaMethode()
      {
      }
      void IMonInterface2.MaMethode()
      {
      }
   }
```

Dans ce cas, pour appeler l'une ou l'autre des méthodes, il faut obligatoirement downcaster l'objet vers l'interface de la méthode concernée.

Exemple C# :

```
         MonImplementation mi = new MonImplementation();
         ((IMonIterface1)mi).MaMethode();
```

En Java, il est possible de définir des constantes dans une interface : ces constantes seront ajoutées dans les classes qui implémentent l'interface.

Exemple Java :

```
public interface MonInterface {
  public final static int maValeur = 100;
}
```

124.2.2. Les modificateurs d'accès

Pour assurer la mise en oeuvre de l'encapsulation, Java et C# propose un ensemble de modificateurs d'accès :

C#	Java	
private	private	Visible uniquement dans le type
public	public	Visible par tout le mode
	aucun : "package-private"	Visible uniquement pour les classes du même package
internal		Visible uniquement depuis une classe de la même assembly
protected		Visible uniquement dans la classe ou ses classes filles indépendamment de l'assembly
internal protected	protected	Visible uniquement dans la classe ou ses classes filles ou depuis une classe de la même assembly en .Net ou du même package en Java

Par défaut une méthode est package-private en Java et private en C#.

En C#, internal et internal protected ne sont pas utilisables sur les membres des structures.

124.2.3. Les champs

En Java, il est possible d'initialiser une variable d'instance avec la valeur d'une autre variable d'instance.

Exemple Java :

```
package fr.jmdoudoux.dej;

public class MaClasse {

  int x = 0;
  int y = x + 10;

}
```

En C#, il n'est pas possible d'initialiser une variable d'instance avec la valeur d'une autre variable d'instance : une erreur CS0236 (A field initializer cannot reference the nonstatic field, method, or property 'field')

Exemple C# :

```
namespace fr.jmdoudoux.dej
{
    public class MaClasse
    {
        int x = 0;
        int y = x + 10;
    }
}
```

La seule solution est de réaliser cette initialisation dans le constructeur

Exemple C# :

```
namespace fr.jmdoudoux.dej
{
    public class MaClasse
    {
        int x = 0;
        int y = 0;

        public MaClasse()
        {
            y = x + 10;
        }
    }
}
```

124.2.4. Les propriétés

Les propriétés permettent la mise en oeuvre de l'encapsulation en offrant un contrôle sur l'accès des champs d'un objet.

Java ne propose aucune fonctionnalité spécifique dans sa syntaxe mais recommande au travers des spécifications des Javabeans de définir des méthodes de type getter et setter nommées plus généralement accesseurs.

Exemple Java :

```
private int taille;

public int getTaille() {
  return taille;
}

public void setTaille (int value) {
  this.taille = value;
}
```

Les propriétés en C# sont similaires à celles proposées par Delphi.

Exemple C# :

```
   private int _taille;


      public int Taille
      {
         get { return _taille; }
         set { _taille = value; }
      }
```

Le mot clé value fait référence à la valeur fournie.

Syntaxiquement, l'utilisation d'une propriété se fait avec la même syntaxe que pour un champ mais en réalité ce sont les méthodes get et set qui sont invoquées de manière transparente.

Il est possible de définir des propriétés en lecture seule, en écriture seule ou en lecture/écriture selon que les méthodes get et set soient définies ou non.

La déclaration est donc plus concise grâce aux propriétés et leur utilisation l'est aussi.

Exemple Java :

```
setTaille(getTaille()+1) ;
```

Exemple C# :

```
Taille++;
```

Cependant, la syntaxe de l'utilisation d'une propriété ne permet pas de savoir si c'est un champ ou une propriété qui est utilisée mais cela facilite le remplacement d'un champ par une propriété.

Il est possible de lever une exception dans le code de mise à jour de la valeur d'une propriété. Ceci peut être cependant déroutant d'avoir la levée d'une exception lors de l'affectation d'une valeur.

124.2.5. Les indexeurs

La suite de cette section sera développée dans une version future de ce document

124.2.6. Les constructeurs

C# et Java permettent la surcharge des constructeurs et l'appel dans un constructeur d'une autre surcharge du constructeur. C# utilise le mot clé this précédé du caractère : et des éventuels paramètres entre parenthèses dans la signature du constructeur

Java utilise le mot clé this suivi des éventuels paramètres entre parenthèses dans le corps du constructeur.

Java et C# appellent automatiquement le constructeur hérité : ceci garantit qu'une instance de la classe fille ne soit pas dans un état inconsistant.

Exemple Java :

```java
public class ClasseA {

  public ClasseA() {
    System.out.println("invocation constructeur ClasseA");
  }

  public void afficher()
  {
      System.out.println("ClasseA");
  }
}
```

Exemple Java :

```java
public class ClasseB extends ClasseA {

  public ClasseB() {
    System.out.println("invocation constructeur ClasseB");
  }

  public void afficher()
  {
      System.out.println("ClasseB");
  }
}
```

Exemple Java :

```
public class Test {

  public static void main(String[] args) {
    ClasseA classeA = new ClasseA();
    classeA.afficher();

    ClasseB classeB = new ClasseB();
    classeB.afficher();
  }
}
```

Résultat :

```
Résultat :
invocation constructeur ClasseA
ClasseA
invocation constructeur ClasseA
invocation constructeur ClasseB
ClasseB
```

Exemple C# :

```
    class ClasseA
    {
        public ClasseA()
        {
            Console.WriteLine("Invocation constructeur ClasseA");
        }

        public virtual void Afficher()
        {
            Console.WriteLine("ClasseA");
        }
    }
```

Exemple C# :

```
    class ClasseB : ClasseA
    {

        public ClasseB()
        {
            Console.WriteLine("Invocation constructeur ClasseB");
        }

        public override sealed void Afficher()
        {
            Console.WriteLine("ClasseB");
        }
    }
```

Exemple C# :

```
   class Program
    {
        public static void Main(String[] args)
        {
            ClasseA classeA = new ClasseA();
            classeA.Afficher();

            ClasseB classeB = new ClasseB();
            classeB.Afficher();
        }

    }
```

Résultat :

```
Invocation constructeur ClasseA
ClasseA
```

```
Invocation constructeur ClasseA
Invocation constructeur ClasseB
ClasseB
```

C# et Java permettent aussi un appel explicite à un constructeur père. Cet appel est obligatoire dans les deux langages sous peine d'avoir une erreur de compilation lorsqu'un constructeur est défini dans une classe fille sans que la classe mère ne possède un constructeur avec les mêmes paramètres.

C# utilise le mot clé base précédé du caractère ":" et des éventuels paramètres entre parenthèses dans la signature du constructeur.

Java utilise le mot clé super suivi des éventuels paramètres entre parenthèses dans le corps du constructeur.

Exemple C# :

```
using System;
using System.Collections.Generic;
using System.Text;

namespace TestCS
{
   class MaClasseMere
   {
      private int id;

      public MaClasseMere(int id)
      {
         this.id = id;
         Console.Out.WriteLine("MaClasseMere id="+id);
      }
   }

   class MaClasse : MaClasseMere
   {
      private string libelle;

      public MaClasse(int id, string libelle)
         : base(id)
      {
         this.libelle = "";
         Console.Out.WriteLine("MaCLasse id="+id);
         Console.Out.WriteLine("MaClasse libelle="+libelle);
      }

      public MaClasse(int id)
         : this(id,"")
      {
      }

   }
}
```

Exemple Java :

```
public class MaClasseMere {
  private int id;

  public MaClasseMere(int id) {
    this.id = id;
    System.out.println("MaClasseMere id=" + id);
  }
}

public class MaClasse extends MaClasseMere {
  private String libelle;

  public MaClasse(int id, String libelle) {
    super(id);
    this.libelle = "";
    System.out.println("MaCLasse id=" + id);
    System.out.println("MaClasse libelle=" + libelle);
  }

  public MaClasse(int id) {
    this(id, "");
  }
}
```

En Java et en C#, il est possible d'invoquer un autre constructeur dans un constructeur : ceci permet de réduire la duplication de code dans les différents constructeurs.

En Java et en C#, les constructeurs ne sont pas hérités : chaque classe ne possède que les constructeurs qu'elle définit.

Exemple C# :

```
    class ClasseA
    {
        public ClasseA()
        {
            Console.WriteLine("Invocation constructeur ClasseA");
        }

        public virtual void Afficher()
        {
            Console.WriteLine("ClasseA");
        }
    }

    class ClasseB : ClasseA
    {

        public ClasseB(int valeur)
        {
            Console.WriteLine("Invocation constructeur ClasseB");
        }

        public void Afficher()
        {
            Console.WriteLine("ClasseB");
        }
    }
```

Exemple C# :

```
'ApplicationTest.ClasseB' does not contain a constructor that takes '0' arguments
```

124.2.7. Les constructeurs statics

La suite de cette section sera développée dans une version future de ce document

124.2.8. Les destructeurs

La suite de cette section sera développée dans une version future de ce document

124.2.9. Le passage de paramètres

En Java, les arguments de type primitifs sont toujours passés en paramètres par valeur (les arguments sont copiés dans la pile). Pour pouvoir modifier la valeur d'un type primitif fourni en paramètre d'une méthode, il faut l'encapsuler dans son wrapper et ainsi le passer sous forme d'objet en paramètre.

En C#, il est possible de préciser si le passage se fait par valeur ou par référence selon l'utilisation des mots clés ref et out.

La suite de cette section sera développée dans une version future de ce document

124.2.10. Liste de paramètres multiples

En C#, le mot clé params permet de préciser qu'un paramètre pourra accepter plusieurs occurrences. La syntaxe dans la signature de la méthode est le mot clé params suivi d'un tableau du type de données puis du nom du paramètre.

En Java, cette fonctionnalité est disponible à partir de la version 5 en utilisant la notation ... précédée du type et du nom de la variable dans la signature de la méthode. Le compilateur va transformer ce paramètre en un tableau du type précisé et c'est ce tableau qui sera manipulé dans le corps de la méthode.

En Java et en C# :

- elle ne peut être utilisée que sur le dernier paramètre d'une méthode.
- il est possible de passer en paramètre un nombre indéterminé de paramètres ou un tableau du type précisé

Exemple C# :

```
class TestParams
{
    public static void Main(String[] args)
    {
        Console.WriteLine("valeur a="+ajouter(1, 2, 3, 4));
        Console.WriteLine("valeur b="+ajouter(new int[] { 1, 2, 3, 4 }));
    }

    public static long ajouter(params int[] array)
    {
        long retour = 0;
        foreach (int i in array)
        {
            retour += i;
        }
        return retour;
    }

}
```

Exemple Java :

```
class TestParams  {

  public static void main(String Args[]) {
      System.out.println("valeur a="+ajouter(1,2,3,4));
      System.out.println("valeur b="+ajouter(new int[] { 1, 2, 3, 4 }));
  }

  public static long ajouter(int ... valeurs) {
    long retour = 0l;

    for (int i : valeurs ) {
      retour += i;
    }

    return retour;
  }
}
```

124.2.11. L'héritage

Java et C# ne proposent pas de support pour l'héritage multiple mais ils proposent tous les deux les interfaces.

Java propose une syntaxe différente pour l'héritage (extends) et l'implémentation (implements).

Exemple Java :

```
package fr.jmdoudoux.dej.heritage;

public class MaClasse extends MaClasseMere implements MonInterfaceA, MonInterfaceB {

}

class MaClasseMere{}

interface MonInterfaceA {}

interface MonInterfaceB {}
```

En C#, il n'y a pas de distinction syntaxique entre un héritage et l'implémentation d'une interface : ils se font tous les deux avec le nom de la classe suivi du caractère deux-points puis, éventuellement, de la classe mère et/ou d'une ou plusieurs interfaces séparées par un caractère virgule.

Exemple C# :

```
namespace fr.jmdoudoux.dej.heritage
{

   public class MaClasse : MaClasseMere, IMonInterfaceA, IMonInterfaceB
   {
   }

   class MaClasseMere { }

   interface IMonInterfaceA { }

   interface IMonInterfaceB { }

}
```

Remarque : par convention, les interfaces commencent par un I majuscule en C#, ce qui permet facilement de les identifier dans la définition d'une classe sous réserve que cette convention soit appliquée.

En Java et en C#, les constructeurs ne sont pas hérités : seuls les constructeurs définis dans la classe sont utilisables.

En Java, pour appeler un constructeur de la classe mère, il faut utiliser le mot clé super() dans le corps du constructeur en lui passant les éventuels paramètres.

Exemple Java :

```
package fr.jmdoudoux.dej;

public class MonException extends Exception {

  public MonException() {
    super();
  }

  public MonException(String message, Throwable cause) {
    super(message, cause);
  }

  public MonException(String message) {
    super(message);
  }

  public MonException(Throwable cause) {
    super(cause);
  }
}
```

C# propose le mot clé base à utiliser dans la signature du constructeur avec les éventuels paramètres.

Exemple C# :

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Runtime.Serialization;

namespace fr.jmdoudoux.dej
{
   [Serializable]
   public class MonException : ApplicationException
   {
      public MonException() : base() { }

      public MonException(string message) : base(message) { }

      public MonException(string message, Exception inner) : base(message, inner) { }

      protected MonException(SerializationInfo info, StreamingContext ctx) : base(info, ctx)
      { }
   }
}
```

En Java, pour empêcher une classe d'être dérivée (ou sous-classée), il faut utiliser le modificateur final dans la déclaration de la classe. C# utiliser le mot clé sealed.

Exemple Java :

```
   public final class MaClasseMere { }
```

Exemple C# :

```
   public sealed class MaClasseMere { }
```

124.2.12. Le polymorphisme

En Java, toutes les méthodes sont virtuelles. Il suffit simplement de redéfinir la méthode dans une classe fille pour mettre en oeuvre le polymorphisme.

Exemple Java :

```
public class ClasseA {

  public void afficher() {
      System.out.println("ClasseA");
  }
}

public class ClasseB extends ClasseA {

  public void afficher() {
      System.out.println("ClasseB");
  }
}

public class TestPolymorph {

  public static void main(String[] args) {
    ClasseA classeA = new ClasseA();
    classeA.afficher();

    ClasseB classeB = new ClasseB();
    classeB.afficher();

    ClasseA classe = new ClasseB();
    classe.afficher();
  }
}
```

Résultat :

```
ClasseA
ClasseB
ClasseB
```

Ceci facilite la vie des développeurs puisque ce mécanisme est mis en oeuvre implicitement mais cela peut poser des soucis de performance puisqu'à l'exécution d'une méthode, il faut parcourir la hiérarchie des classes filles pour trouver une éventuelle redéfinition à utiliser. En plus de nuire aux performances, cela compromet les optimisations réalisables par la machine virtuelle.

Lorsqu'une méthode ne sera pas redéfinie dans une classe fille, les développeurs Java peuvent la déclarer avec le modificateur final pour éviter cette recherche.

Le fait que toutes les méthodes soient virtuelles peut aussi être source d'erreurs difficiles à détecter.

Exemple Java :

```
public class ClasseA {

  public void afficher() {
      System.out.println("ClasseA");
  }
}

public class ClasseB extends ClasseA {

  public void aficher() {
      System.out.println("ClasseB");
  }
}

public class TestPolymorph {

  public static void main(String[] args) {
    ClasseA classeA = new ClasseA();
    classeA.afficher();
    ClasseB classeB = new ClasseB();
    classeB.afficher();

    ClasseA classe = new ClasseB();
    classe.afficher();
  }
}
```

Résultat :

```
ClasseA
ClasseA
ClasseA
```

Pour pallier cette difficulté, Java 5 a introduit l'annotation standard @Override qui permet au compilateur de vérifier que la classe redéfinit bien une méthode définie dans la classe mère.

Exemple Java :

```
public class ClasseB extends ClasseA {

  @Override
  public void aficher() {
      System.out.println("ClasseB");
  }
}
```

Résultat :

```
C:\java\Test\src\com\jmd\test>javac ClasseB.java
ClasseB.java:3: cannot find symbol
symbol: class ClasseA
public class ClasseB extends ClasseA {
                             ^
ClasseB.java:5: method does not override or implement a method from a supertype
  @Override
  ^
2 errors
```

Enfin, le fait que toutes les méthodes soient virtuelles peut permettre dans une classe fille de redéfinir involontairement une méthode.

En C#, aucune méthode n'est virtuelle. Chaque méthode qui pourra être redéfinie doit être déclarée avec le mot clé virtual. Chaque méthode redéfinie doit être déclarée avec le mot clé override. Si ce n'est pas le cas, aucune erreur n'est signalée par le compilateur et le comportement ne sera pas celui attendu.

Pour déclarer une méthode virtuelle, il faut utiliser le mot clé virtual dans la déclaration de la méthode. Cependant cela ne suffit pas, car dans ce cas, aucune méthode redéfinie n'est trouvée dans la hiérarchie d'objets.

Exemple C# :

```
    class ClasseA
    {
        public virtual void Afficher()
        {
            Console.WriteLine("ClasseA");
        }
    }

    class ClasseB : ClasseA
    {
        public void Afficher()
        {
            Console.WriteLine("ClasseB");
        }
    }

    class Program
    {
        public static void Main(String[] args)
        {
            ClasseA classeA = new ClasseA();
            classeA.Afficher();

            ClasseB classeB = new ClasseB();
            classeB.Afficher();

            ClasseA classe = new ClasseB();
            classe.Afficher();
        }
    }
```

Résultat :

```
ClasseA
ClasseB
ClasseA
```

Voici l'exemple ou le polymorphisme est correctement mis en oeuvre en déclarant la méthode de la classe virtuelle avec le mot clé virtual et en redéfinissant la méthode de la classe fille avec le mot clé override.

Exemple C# :

```
    class ClasseA
    {
        public virtual void Afficher()
        {
            Console.WriteLine("ClasseA");
        }
    }

    class ClasseB : ClasseA
    {
        public override void Afficher()
        {
            Console.WriteLine("ClasseB");
        }

    }

    class Program
    {
        public static void Main(String[] args)
        {
            ClasseA classeA = new ClasseA();
            classeA.Afficher();

            ClasseB classeB = new ClasseB();
            classeB.Afficher();

            ClasseA classe = new ClasseB();
            classe.Afficher();
        }
    }
```

Résultat :

```
ClasseA
ClasseB
ClasseB
```

Si le mot clé override est utilisé sur une méthode redéfinie de la classe mère qui ne possède pas le mot clé virtual alors il y a une erreur de compilation.

Exemple C# :

```
    class ClasseA
    {
        public void Afficher()
        {
            Console.WriteLine("ClasseA");
        }
    }

        class ClasseB : ClasseA
    {
        public override void Afficher()
        {
            Console.WriteLine("ClasseB");
        }
    }
```

Résultat :

```
'ApplicationTest.ClasseB.Afficher()': cannot override inherited member
'ApplicationTest.ClasseA.Afficher()'  because it is not marked virtual, abstract, or override
C:\Documents and Settings\jmd\My Documents\Visual Studio 2008\
Projects\ApplicationTest\ApplicationTest\ClasseB.cs
      10        30      ApplicationTest
```

Il est aussi possible d'utiliser le mot clé new dans la déclaration de la méthode redéfinie pour préciser explicitement que la méthode masque la méthode héritée.

Exemple C# :

```
    class ClasseA
    {
        public virtual void Afficher()
        {
            Console.WriteLine("ClasseA");
        }
    }

        class ClasseB : ClasseA
    {
        public new void Afficher()
        {
            Console.WriteLine("ClasseB");
        }
    }
```

Pour les développeurs Java, il faut être particulièrement vigilant avec les méthodes virtuelles et leur redéfinition en C# car leur déclaration est à la charge du développeur.

En C#, le mot clé sealed empêche toute redéfinition d'une méthode même marquée avec override.

Exemple C# :

```
    class ClasseA
    {
        public virtual void Afficher()
        {
            Console.WriteLine("ClasseA");
        }
    }

    class ClasseB : ClasseA
    {
        public override sealed void Afficher()
        {
            Console.WriteLine("ClasseB");
        }

    }

    class ClasseC : ClasseB
    {
        public override void Afficher()
        {
            Console.WriteLine("ClasseC");
        }
    }
```

Résultat :

```
'ApplicationTest.ClasseC.Afficher()': cannot override inherited member
 'ApplicationTest.ClasseB.Afficher()' because it is sealed
 C:\Documents and Settings\jmd\My Documents\Visual Studio 2008\
 Projects\ApplicationTest\ApplicationTest\ClasseC.cs
      10      30        ApplicationTest
```

En Java, le mot clé final empêche toute redéfinition d'une méthode.

Exemple Java :

```
public class ClasseA {

  public final void afficher() {
      System.out.println("ClasseA");
  }

}

public class ClasseB extends ClasseA {

  public void afficher() {
      System.out.println("ClasseB");
  }
}
```

Résultat :

```
C:\java\Test\src\com\jmd\test>javac -cp C:\java\Test\src ClasseB.java
ClasseB.java:5: afficher() in fr.jmdoudoux.dej.ClasseB cannot override afficher() in
 fr.jmdoudoux.dej.ClasseA; overridden method is final
   public void afficher()
              ^
              1 error
```

124.2.13. Les generics

Java propose le support des generics depuis sa version 5 et C# depuis sa version 2.0.

La suite de cette section sera développée dans une version future de ce document

124.2.14. Le boxing/unboxing

La suite de cette section sera développée dans une version future de ce document

124.2.15. La surcharge des opérateurs

La suite de cette section sera développée dans une version future de ce document

124.2.16. Les classes imbriquées

La suite de cette section sera développée dans une version future de ce document

124.2.17. Les classes anonymes internes (Anonymous Inner classes)

La suite de cette section sera développée dans une version future de ce document

124.2.18. L'import de classes

La suite de cette section sera développée dans une version future de ce document

124.2.19. Déterminer et tester le type d'un objet

Java propose l'opérateur instanceof pour tester le type d'un objet.

Exemple Java :

```
  public void tester(Object monObjet)
  {
     if (monObjet instanceof MaClasse)
     {
        MaClasse maClasse = (MaClasse) monObjet;
        // suite des traitements
     }
  }
```

C# propose l'opérateur is qui est équivalent.

Exemple C# :

```
      public void Tester(object monObjet)
      {
         if (monObjet is MaClasse)
         {
            MaClasse maClasse = (MaClasse) monObjet;
            // suite des traitements
         }
      }
```

124.2.20. L'opérateur as

L'opérateur as en C# permet de demander une conversion vers un autre type. Si la conversion n'est pas possible aucune exception n'est levée et la valeur retournée par l'opérateur est null.

Exemple C# :

```
      public static void tester(object monObjet)
      {
         MaClasse maClasse = monObjet as MaClasse;
         if (maClasse != null)
         {
            // suite des traitements
         }
      }
```

Il n'existe pas d'équivalent en Java.

124.3. Les chaînes de caractères

Java et .Net encapsulent les chaînes de caractères dans des objets immuables respectivement java.lang.String et System.String qu'il n'est pas possible de sous-classer. Chaque opération sur ces objets ne modifie pas l'instance courante mais crée une nouvelle instance. Chaque méthode qui modifie le contenu de la chaîne retourne une instance qui contient le résultat des modifications.

Exemple Java :

```
    String chaine = "test";
    chaine.toUpperCase();
    System.out.println(chaine);
```

Exemple C# :

```
    string chaine = "test";
    chaine.ToUpper();
    Console.WriteLine(chaine);
```

Dans les deux exemples, la chaîne affichée est en minuscule.

C# propose le mot clé string qui sera remplacé à la compilation par System.String.

Pour gérer des concaténations répétées, Java et .Net proposent respectivement java.lang.StringBuffer (ou java.lang.StringBuilder depuis Java 5) et System.Text.StringBuilder

La taille d'une chaîne est obtenue en utilisant la méthode length() de la classe String en Java et en utilisant la propriété Length de la classe String en C#.

Pour gérer des chaînes de caractères contenant des caractères spéciaux, C# propose de les échapper comme en Java en les faisant précéder d'un caractère antislash ou sans les échapper en faisant précéder la chaîne de caractères par un caractère @ :

Exemple C# :

```
  string chemin1 = "C:\\temp\\monfichier.txt";
  string chemin2 = @"C:\temp\monfichier.txt";
```

En Java, pour tester l'égalité de deux chaines en tenant compte de la casse, il faut utiliser la méthode equals() de la classe String. L'opérateur == appliqué sur deux instances d'un objet de type String teste l'égalité de la référence des objets.

En C#, pour tester l'égalité de deux chaines en tenant compte de la casse, il faut utiliser la méthode equals() de la classe String ou l'opérateur ==.

124.4. Les tableaux

La suite de cette section sera développée dans une version future de ce document

124.5. Les indexeurs

La suite de cette section sera développée dans une version future de ce document

124.6. Les exceptions

Java et .Net supportent le mécanisme des exceptions pour la gestion des faits inattendus lors de l'exécution des traitements.

La gestion des erreurs est assurée en Java et C# par les mots clés try/catch/finally.

Il est possible en Java et en C# de capturer une exception et de la repropager ou de lever une autre exception.

Les deux plates-formes proposent une hiérarchie de classes d'exceptions standard dérivant de la classe java.lang.Exception pour Java et System.Exception pour C#. Chaque plates-forme permet la définition de ses propres exceptions et proposent le support du chaînage des exceptions (depuis la version 1.4 de Java).

La grande différence dans l'utilisation des exceptions est l'obligation en Java de déclarer, dans la signature des méthodes et grâce au mot clé throws, la propagation d'une exception de type checked non gérée. Les exceptions de type Runtime n'ont pas l'obligation d'être déclarée dans une clause throws.

En C#, il n'y a pas de déclaration des exceptions pouvant être levées dans la signature des méthodes : il n'y a donc pas d'équivalent au mot clé throws de Java. Ainsi l'exception remonte la pile d'appels et si elle n'est pas traitée avant le début de la pile, le CLR s'arrête. Pour compenser ce manque de gestion imposée, il faut documenter les API pour informer les développeurs des exceptions qui peuvent être levées.

En C#, sans le code source ou la documentation associée, il n'est donc pas possible de connaître les exceptions peuvant être levées par une méthode.

La suite de cette section sera développée dans une version future de ce document

124.7. Le multitâche

Java et C# proposent tous les deux un support du multitâche au travers de threads

124.7.1. Les threads

La suite de cette section sera développée dans une version future de ce document

124.7.2. La synchronisation de portions de code

Java et C# proposent un mécanisme de verrous sur une portion de code pour éviter son exécution simultanée par plusieurs threads. C# propose le mot clé lock et Java le mot clé synchronised : dans les deux langages le mode d'utilisation est le même et repose sur un moniteur d'objets.

Exemple C# :

```
lock(this)
{
        compteur++ ;
}
```

Exemple Java :

```
synchronised(this)
{
        compteur++ ;
}
```

Remarque : le mot clé lock de C# est un raccourci syntaxique à l'utilisation des méthodes Enter() et Exit() de la classe System.Threading.Monitor.

C# propose aussi la classe System.Threading.Interlocked pour synchroniser quelques opérations basiques (incrémentation, décrémentation, échange de valeurs, ajout d'une valeur, ...).

Exemple C# :

```
public static int compteur = 0;

 public static void incrementer() {
   Interlocked.Increment( ref compteur );
 }
```

Java et C# proposent d'appliquer le mécanisme à une méthode dans son intégralité. Java utilise le mot clé synchronised dans la déclaration de la méthode. C# propose le mot clé interlocked ou la métadonnée MethodImpl avec l'option MethodImplOptions.Synchronized

Exemple C# :

```
public static int compteur = 0;

[MethodImpl(MethodImplOptions.Synchronized)]
 public static void incrementer() {
   compteur++;
 }
```

124.7.3. Le mot clé volatile

La suite de cette section sera développée dans une version future de ce document

124.8. L'appel de code natif

La suite de cette section sera développée dans une version future de ce document

124.9. Les pointeurs

La suite de cette section sera développée dans une version future de ce document

124.10. La documentation automatique du code

La suite de cette section sera développée dans une version future de ce document

124.11. L'introspection/reflection

La suite de cette section sera développée dans une version future de ce document

124.12. La sérialisation

C# et Java proposent des mécanismes pour permettre la sérialisation d'objets. La sérialisation est mise en oeuvre notamment dans RMI en Java et .Net Remoting en C#.

La suite de cette section sera développée dans une version future de ce document

Partie 19 : Le développement d'applications mobiles

Partie 19 :

Développement d'applications mobiles

Le marché des machines portables est en pleine expansion : téléphones mobiles, PDA, ... De plus en plus d'applications s'exécutent aussi sur des machines embarquées.

Une édition particulière de Java est proposée pour ce type de développement : J2ME (Java 2 Micro Edition) / Java ME (Java Micro Edition).

Cette partie contient les chapitres suivants :

- J2ME / Java ME : présente la plate-forme Java pour le développement d'applications sur des appareils mobiles tels que des PDA ou des téléphones cellulaires
- CLDC : présente les packages et les classes de la configuration CLDC
- MIDP : propose une présentation et une mise en oeuvre du profil MIDP pour le développement d'applications mobiles
- CDC : présente les packages et les classes de la configuration CDC
- Les profils du CDC : propose une présentation et une mise en oeuvre des profils pouvant être utilisés avec la configuration CDC
- Les autres technologies pour les applications mobiles : propose une présentation des autres technologies basées sur Java pour développer des applications mobiles

125. J2ME / Java ME

Chapitre 125

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

J2ME est la plate-forme Java pour développer des applications sur des appareils mobiles tels que des PDA, des téléphones cellulaires, des terminaux de points de vente, des systèmes de navigation pour voiture, ...

C'est une sorte de retour aux sources puisque Java avait été initialement développé pour piloter des appareils électroniques avant de devenir une plate-forme pour le développement et l'exécution d'applications polyvalentes.

Un environnement J2ME est composé de plusieurs éléments :

- Une machine virtuelle dédiée tenant compte des ressources limitées du matériel cible
- Un ensemble d'API de base
- Des API spécifiques

Pour répondre à la plus large gamme d'appareils cibles, J2ME est modulaire grâce à trois types d'entités qui s'utilisent par composition :

- Les configurations : définissent des caractéristiques minimales d'un large sous type de matériel, d'une machine virtuelle et d'API de base
- Les profiles : définissent des API relatives à une fonctionnalité commune à un ensemble d'appareils (exemple : interface graphique, ...)
- Les packages optionnels : définissent des API relatives à une fonctionnalité spécifique dont le support est facultatif

Ce chapitre contient plusieurs sections :

- ♦ L'historique de la plate-forme : fournit un rapide historique de la plate-forme J2ME / Java ME
- ♦ La présentation de J2ME / Java ME : présentation rapide des éléments et concepts de la plate-forme J2ME
- ♦ Les configurations : présentation des deux configurations sur lesquelles la plate-forme J2ME repose
- ♦ Les profiles : présentation des profiles qui enrichissent les configurations pour un type de machines ou à une fonctionnalité spécifique
- ♦ J2ME Wireless Toolkit 1.0.4 : installation et mise en oeuvre de cet outil proposé par Sun pour le développement d'applications utilisant MIDP 1.0
- ♦ J2ME wireless toolkit 2.1 : installation et mise en oeuvre de cet outil proposé par Sun pour le développement d'applications utilisant MIDP 1.0 et 2.0

125.1. L'historique de la plate-forme

Le langage Java a été développé à son origine pour la programmation d'appareils électroniques de grande consommation (langage Oak). Cependant au fil des années, Java a évolué pour être principalement utilisé pour le développement d'applications standalone et serveurs. La plate-forme Java ME peut ainsi être vue comme un retour aux buts originaux de Java.

Historiquement, Sun a proposé plusieurs plates-formes pour le développement d'applications sur des machines possédant des ressources réduites, typiquement celles ne pouvant exécuter une JVM répondant aux spécifications complètes de la plate-forme Java SE.

- JavaCard (1996) : pour le développement sur des cartes à puces
- PersonnalJava (1997) : pour le développement sur des machines possédant au moins 2Mo de mémoire
- EmbeddedJava (1998) : pour des appareils avec de faibles ressources

En 1999, Sun propose de mieux structurer ces différentes plates-formes sous l'appellation J2ME (Java 2 Micro Edition). Courant 2000, la plate-forme J2ME est créée pour le développement d'applications sur appareils mobiles ou embarqués tels que des téléphones mobiles, des PDA, des terminaux, ... : elle est donc la descendante des différentes plates-formes antérieures relatives aux appareils mobiles. Seule la plate-forme JavaCard n'est pas incluse dans Java ME et reste à part.

Java ME cible de très nombreux appareils électroniques possédant différentes caractéristiques dans une même plate-forme.

En juin 2005, la plate-forme J2ME a été renommée, comme les autres plates-formes Java, en Java ME (Java Micro Edition).

125.2. La présentation de J2ME / Java ME

La plate-forme Java Mobile Edition (J2ME/Java ME) cible le marché des appareils électroniques et embarqués tels que les pagers, les téléphones cellulaires, les PDA, les set top boxes, etc ... Elle est composée de plusieurs éléments :

- Des spécifications
- Des machines virtuelles
- Des API dédiées
- Des outils pour le développement, le déploiement et la configuration

La plate-forme Java ME cible des appareils électroniques mobiles ou embarqués dont les caractéristiques peuvent être particulièrement différentes et qui représentent un nombre très important d'appareils différents. La grande difficulté est donc de définir une plate-forme qui propose des services pour le plus grand nombre d'appareils possible.

La seule solution pour répondre à cette problématique est de rendre la conception de la plate-forme modulaire. L'ensemble des appareils sur lequel peut s'exécuter une application écrite avec J2ME est tellement vaste et disparate que J2ME est composé de plusieurs parties : les configurations et les profiles qui sont spécifiés par le JCP. J2ME propose donc une architecture modulaire.

J2ME définit deux grandes familles d'appareils :

- Appareils à fonctionnalités dédiées ou limitées : ressources et interface graphique limitées, peuvent se connecter par intermittence au réseau (exemple : téléphone mobile, agenda électronique, PDA, pagers, ...)
- Appareils proposant une interface graphique riche, possédant une connexion continue au réseau (exemple : PDA haut de gamme, smartphone, set top boxes, système de navigation, ...)

La modularité de la plate-forme est assurée par trois concepts :

- Configuration : définit une spécification pour une plate-forme Java pour une des deux familles définies, une machine virtuelle et des API de base
- Profile : définit des API pour des fonctionnalités communes pour une catégorie d'appareils similaires. Un profile est définit pour une configuration sur laquelle il s'appuie et peut s'appuyer un autre profile
- Package optionnel : définit des API pour des fonctionnalités spécifiques

L'inconvénient de ce principe est qu'il déroge à la devise de Java "Write Once, Run Anywhere". Ceci reste cependant partiellement vrai pour des applications développées pour un profile particulier. Il ne faut cependant pas oublier que les types de machines cibles de J2ME sont tellement différents (du téléphone mobile au set top box), qu'il est surement impossible de trouver un dénominateur commun. Ceci associé à l'explosion du marché des machines mobiles explique les nombreuses évolutions en cours de la plate-forme.

Java ME ne définit pas un nouveau langage de programmation mais adapte la technologie Java aux appareils mobiles et embarqués.

Java ME tente de conserver autant que possible la compatibilité avec Java SE. Pour répondre aux besoins spécifiques des appareils mobiles, Java ME remplace certaines API ou en propose de nouvelles.

Une application Java ME est organisée en plusieurs couches logicielles :

Application Java ME
Packages optionnels, API tiers
Profiles
Configuration
Machine virtuelle (VM)
Appareil

Une application est développée en reposant sur une configuration qui cible une large famille d'appareils cibles, un ou plusieurs profiles qui fournissent des fonctionnalités de base et des packages optionnels ou des API tiers pour des fonctionnalités spécifiques.

Chaque configuration peut être utilisée avec un ensemble de packages optionnels qui permet d'utiliser des technologies particulières (Bluetooth, services web, lecteur de codes barre, etc ...). Ces packages sont le plus souvent dépendant du matériel.

Les API tiers ne font pas partie de Java ME mais elles s'appuient sur elle ou l'étendent pour définir des API spécifiques à un appareil ou une fonctionnalité.

Par rapport à Java SE, Java ME utilise des machines virtuelles différentes. Certaines classes de base de l'API sont communes avec cependant de nombreuses omissions dans l'API Java ME.

Java ME définit des environnements qui servent de socles pour développer des applications portables :

- Java Technologies for Wireless Industry
- Mobile Service Architecture

Java ME est la plate-forme Java la plus récente.

De plus amples informations peuvent être obtenues sur le site :
https://www.oracle.com/java/technologies/javameoverview.html

125.3. Les configurations

Les configurations définissent les caractéristiques de bases d'un environnement d'exécution pour un certain type de machine possédant un ensemble de caractéristiques et de ressources similaires. Elles se composent d'une machine virtuelle et d'un ensemble d'API de base.

Deux configurations sont actuellement définies :

- CLDC (Connected Limited Device Configuration)
- CDC (Connected Device Configuration)

La CLDC 1.0 est spécifiée dans la JSR 030 : elle concerne des appareils possédant des ressources faibles (moins de 512 Kb de RAM, faible vitesse du processeur, connexion réseau limitée et intermittente) et une interface utilisateur réduite (par exemple un téléphone mobile ou un PDA "entrée de gamme"). Elle s'utilise sur une machine virtuelle KVM. La version 1.1 est le résultat des spécifications de la JSR 139 : une des améliorations les plus importantes est le support des nombres flottants.

La CDC est spécifiée dans la JSR 036 : elle concerne des appareils possédant des ressources plus importantes (au moins 2Mb de RAM, un processeur 32 bits, une meilleure connexion au réseau), par exemple un set top box ou certains PDA "haut de gamme". Elle s'utilise sur une machine virtuelle CVM.

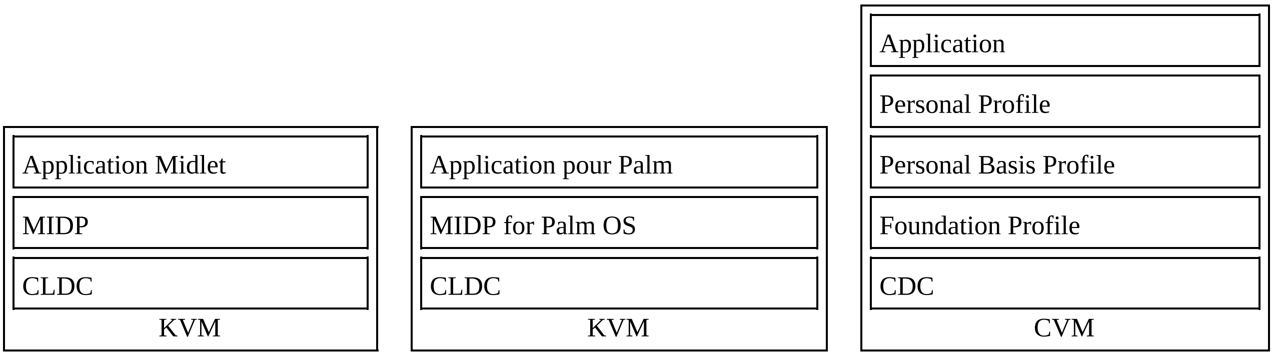
125.4. Les profiles

Les profiles se composent d'un ensemble d'API particulières à un type de machines ou à une fonctionnalité spécifique. Ils permettent l'utilisation de fonctionnalités précises et doivent être associés à une configuration. Ils permettent donc d'assurer une certaine modularité à la plate-forme J2ME.

Il existe plusieurs profiles :

Profil	Configuration	JSR	
MIDP 1.0	CLDC	37	Package javax.microedition.*
Foundation Profile	CDC	46	
Personal Profile	CDC	62	
MIDP 2.0	CLDC	118	
Personal Basis Profile	CDC	129	
RMI optional Profile	CDC	66	
Mobile Media API (MMAPI) 1.1	CLDC	135	Permet la lecture de clips audio et vidéo
PDA		75	
JDBC optional Profile	CDC	169	
Wireless Messaging API (WMA) 1.1	CLDC	120	Permet l'envoi et la réception de SMS

Les utilisations possibles des profils sont :



MIDP est un profile standard qui n'est pas défini pour une machine particulière mais pour un ensemble de machines embarquées possédant des ressources et une interface graphique limitée.

Sun a développé un profil particulier nommé KJava pour le développement spécifique sur Palm. Ce profile a été remplacé par un nouveau profil nommé MIDP for Palm OS.

Le Foundation Profile est un profil de base qui s'utilise avec CDC. Ce profil ne permet pas de développer des IHM. Il faut lui associer un des deux profils suivants :

- le Personal Basic Profile permet le développement d'applications connectée avec le réseau
- le Personal Profile est un profil qui permet le développement complet d'une IHM et d'applet grâce à AWT.

PersonalJava est remplacé par le Personal Profile.

Le choix du ou des profils utilisés pour les développements est important car il conditionne l'exécution de l'application sur un type de machine supporté par le profil.

Cette multitude de profils peut engendrer un certain nombre de problèmes lors de l'exécution d'une application sur différents périphériques car il n'y a pas la certitude d'avoir à disposition les profils nécessaires. Pour résoudre ce problème, une spécification particulière issue des travaux de la JSR 185 et nommée Java Technology for the Wireless Industry (JTWI) a été développée. Cette spécification impose aux périphériques qui la respectent de mettre en oeuvre au minimum : CLDC 1.0, MIDP 2.0, Wireless Messaging API 1.1 et Mobile Media API 1.1. Son but est donc d'assumer une meilleure compatibilité entre les applications et les différents téléphones mobiles sur lesquelles elles s'exécutent.

125.5. J2ME Wireless Toolkit 1.0.4

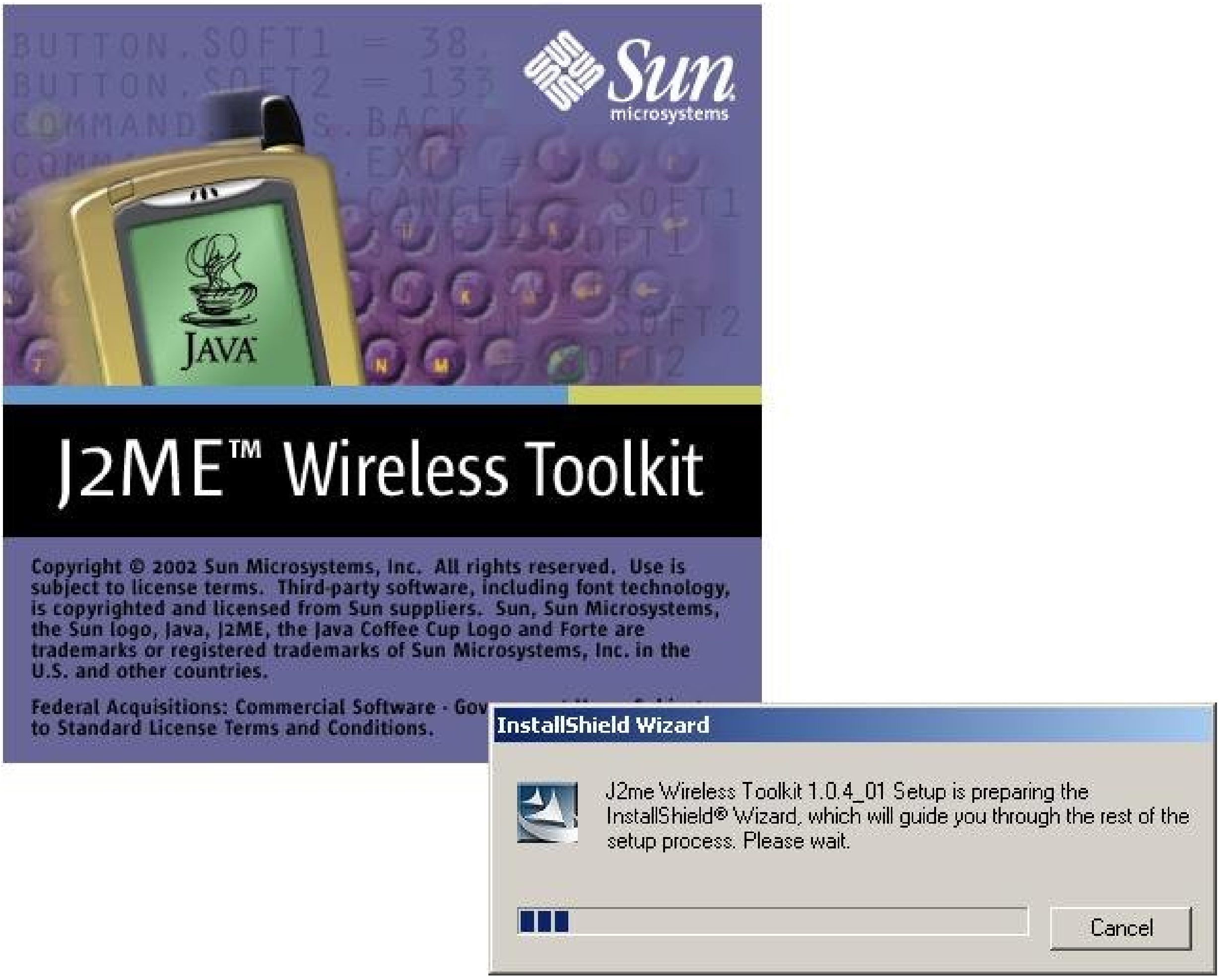
Sun propose un outil pour développer des applications J2ME utilisant CLDC/MIDP. Cet outil peut être téléchargé à l'url suivante : https://www.oracle.com/java/technologies/java-archive-downloads-javame-downloads.html

La version 1.0.4 de cet outil permet de développer des applications utilisant MIDP 1.0.

125.5.1. L'installation du J2ME Wireless Toolkit 1.0.4

L'installation ci-dessous concerne la version 1.0.4.

Il faut exécuter le fichier j2me_wireless_toolkit-1_0_4_01-bin-win.exe



Copyright © 2002 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms. Third-party software, including font technology, is copyrighted and licensed from Sun suppliers. Sun, Sun Microsystems, the Sun logo, Java, J2ME, the Java Coffee Cup Logo and Forte are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Federal Acquisitions: Commercial Software - Gov
to Standard License Terms and Conditions.

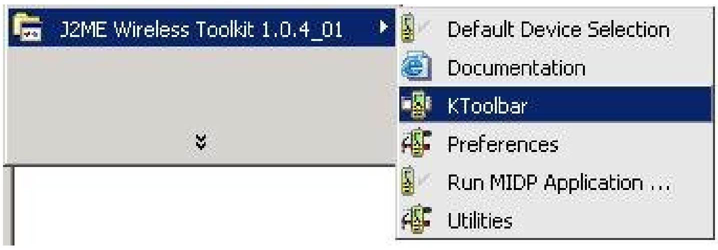
J2me Wireless Toolkit 1.0.4_01 Setup is preparing the InstallShield® Wizard, which will guide you through the rest of the setup process. Please wait.

Il faut suivre les instructions suivantes, guidées par l'assistant d'installation :

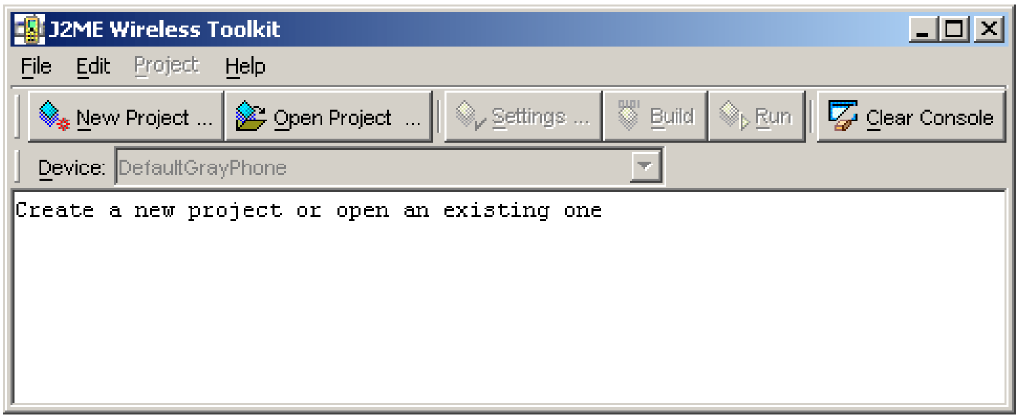
- sur la page d'accueil (welcome) , cliquez sur "Suivant"
- sur la page d'acceptation de la licence, lire la licence et l'approuver en cliquant sur "Yes"
- sur la page de sélection de localisation de la JVM, cliquez sur "Next" (sélectionner l'emplacement si aucune JVM n'a été détectée automatiquement)
- sélectionner l'emplacement de l'installation de l'outil et cliquez sur "Next"
- cliquez sur "Next" pour accepter le menu par défaut dans le menu "Démarrer/Programme"

- sur la page de résumé des opérations, cliquez sur "Next"
- sur le dernière page (Complete), cliquez sur "Finish"

125.5.2. Les premiers pas



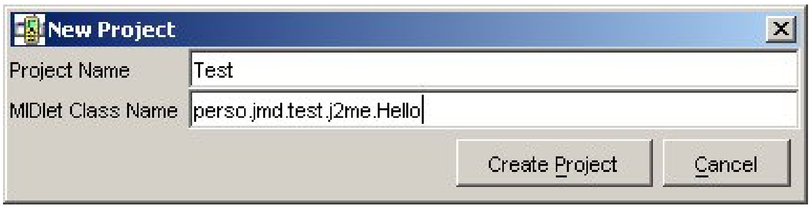
Il faut exécuter l'outil KToolBar.



Device: DefaultGrayPhone

```
Create a new project or open an existing one
```

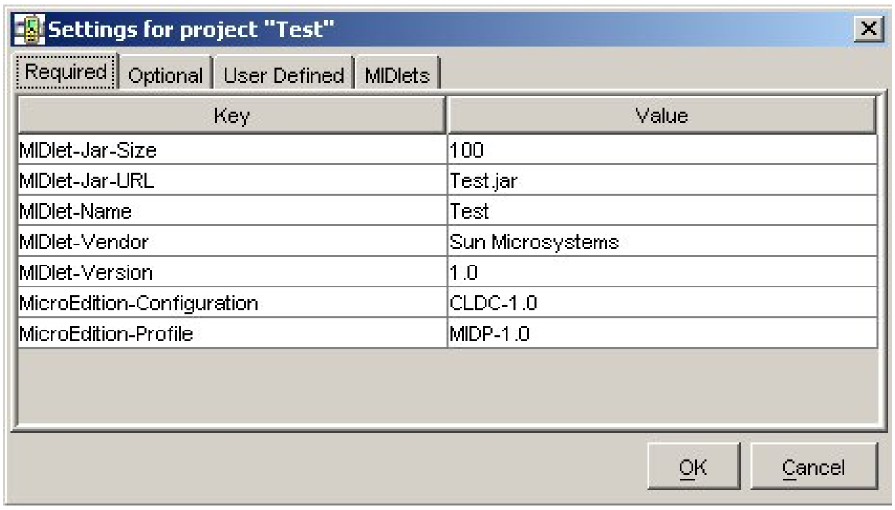
Pour créer un projet, il faut cliquer sur le bouton "New Project" ou sur l'option "New Project" du menu "File".



Project Name: Test

MIDlet Class Name: perso.jmd.test.j2me.Hello

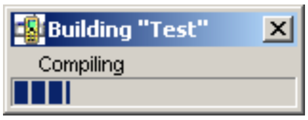
Il faut saisir le nom du projet et le nom qualifié de la midlet puis cliquer sur "Create Project".



Key	Value
MIDlet-Jar-Size	100
MIDlet-Jar-URL	Test.jar
MIDlet-Name	Test
MIDlet-Vendor	Sun Microsystems
MIDlet-Version	1.0
MicroEdition-Configuration	CLDC-1.0
MicroEdition-Profile	MIDP-1.0

Il faut ensuite créer la ou les classes dans le répertoire src de l'arborescence du projet.

Pour construire le projet, il faut cliquer sur le bouton "Build".



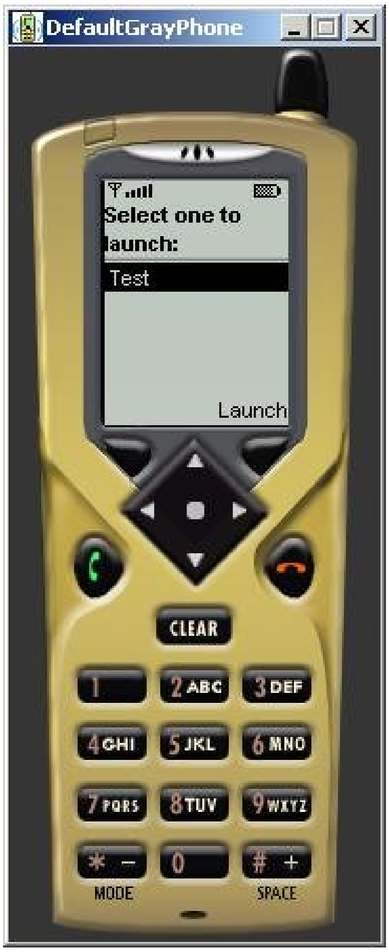
Building "Test"

Compiling

Pour exécuter le projet, il suffit de choisir le type d'émulateur à utiliser et cliquer sur le bouton "Run".

Exemple : avec l'émulateur de téléphone par défaut.

Cliquer sur l'application "Test"



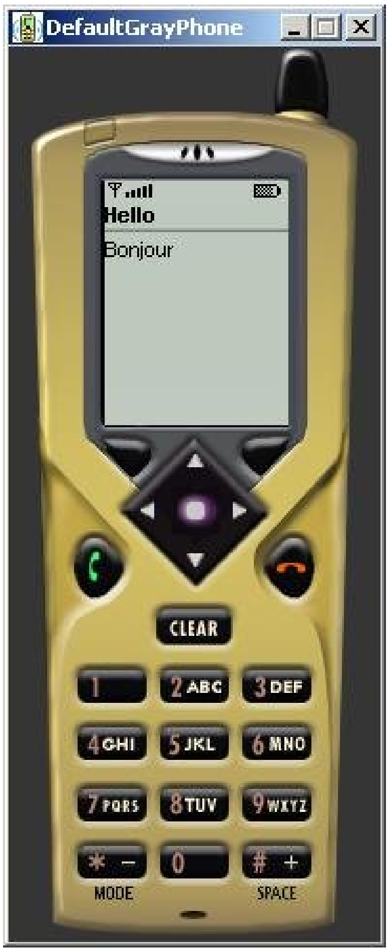
DefaultGrayPhone

Select one to launch:

Test

Launch

puis cliquer sur le bouton entre les flèches

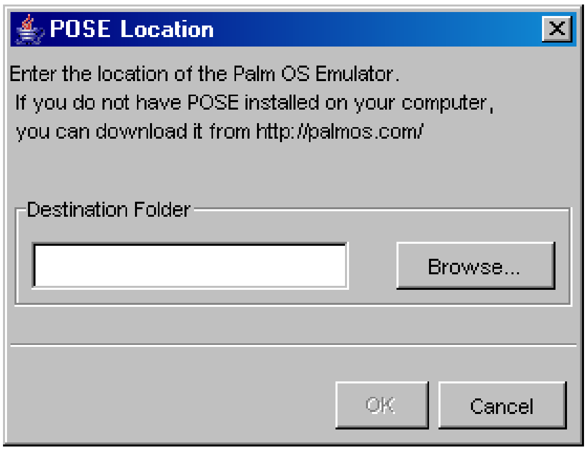


DefaultGrayPhone

Hello

Bonjour

Il est aussi possible d'utiliser l'émulateur Palm POSE (Palm O.S. Emulator). L'outil demande le chemin d'accès à POSE,

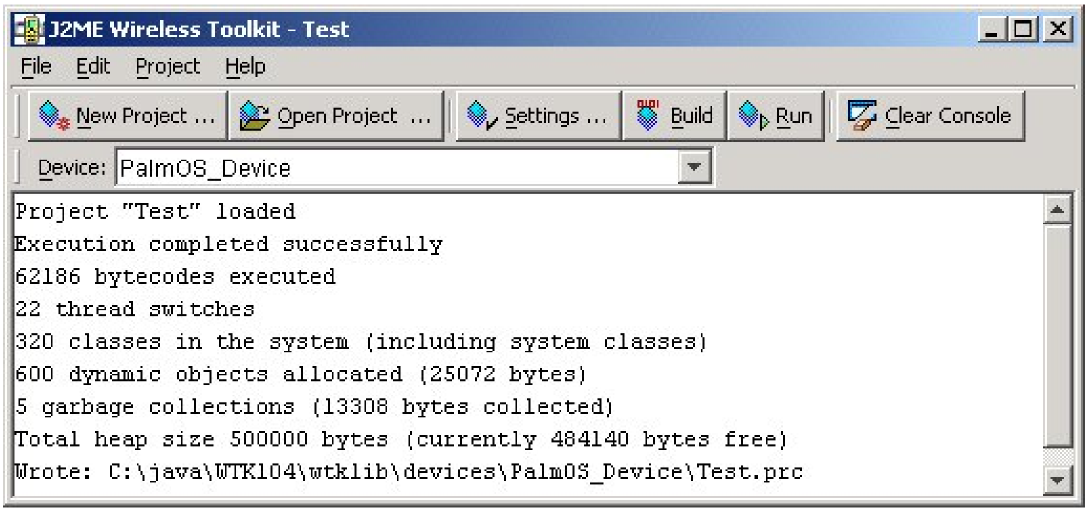


POSE Location

Enter the location of the Palm OS Emulator.
If you do not have POSE installed on your computer,
you can download it from http://palmos.com/

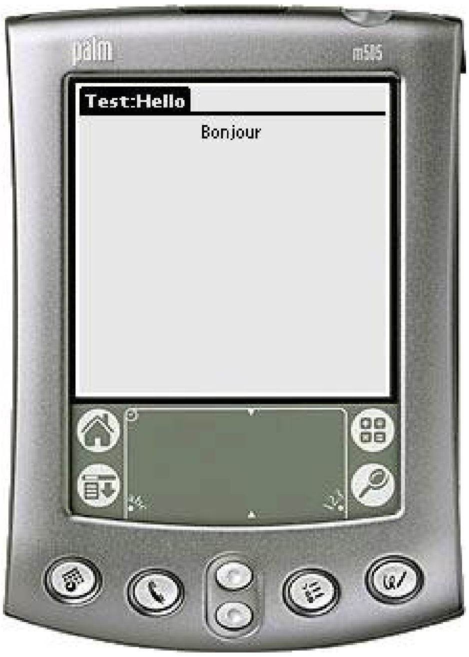
Destination Folder

Puis l'outil génère un fichier .prc.



```
Project "Test" loaded
Execution completed successfully
62186 bytecodes executed
22 thread switches
320 classes in the system (including system classes)
600 dynamic objects allocated (25072 bytes)
5 garbage collections (13308 bytes collected)
Total heap size 500000 bytes (currently 484140 bytes free)
Wrote: C:\java\WTK104\wtklib\devices\PalmOS_Device\Test.prc
```

Enfin, il lance l'émulateur et installe le fichier pour l'exécuter.



Pour plus de détails, voir la section sur MIDP for Palm OS.

125.6. J2ME wireless toolkit 2.1

La version 2.0 permet d'utiliser MIDP 1.0 ou 2.0 ainsi que les API optionnels Mobile Media et Wireless Messaging . Il peut être intégré dans d'autres IDE tels que Sun Studio Mobile Edition ou JBuilder.

La version 2.1 permet d'utiliser CLDC 1.1 et l'API J2ME Web service et de développer des applications pour des périphériques qui respectent les spécifications JTWI.

125.6.1. L'installation du J2ME Wireless Toolkit 2.1

La version 2.1 du J2ME Wireless Toolkit nécessite la présence sur le système d'un J2SE 1.4 minimum.

Elle permet le développement d'applications répondant aux spécifications de la JSR-185 (Java Technology for the Wireless Industry) qui inclue : CLDC 1.1, MIDP 2.0, WMA 1.1 MMAPI 1.1.

Elle permet aussi l'utilisation de la JSR-172 (J2ME Web Services Specification).

Lancer l'application j2me_wireless_toolkit-2_1-windows.exe. Un assistant guide l'utilisateur dans les différentes étapes de l'installation :

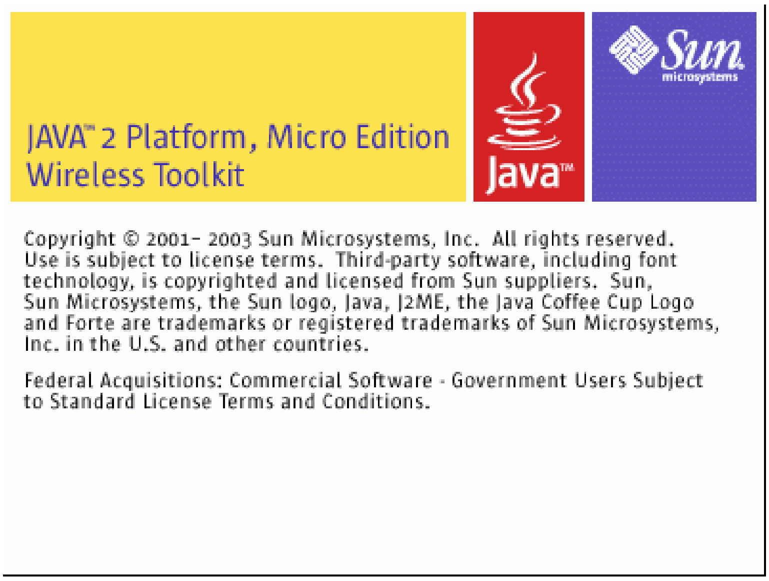
- sur la page d'accueil, cliquez sur le bouton « Next »
- sur la page « License Agreement » : lire la licence et si vous l'acceptez, cliquez sur le bouton « Yes »
- sur la page « Java Virtual Machine Location » : le programme détecte automatiquement la présence d'un JDK 1.4 ou supérieure, cliquez sur le bouton « Next »
- sur la page « Choose Destination Location » : sélectionnez le répertoire d'installation de l'application et cliquez sur le bouton « Next »
- sur la page « Select Program Folder » : saisissez ou sélectionner le dossier du menu démarrer qui va contenir les raccourcis vers l'application si celui par défaut ne convient pas. Cliquez sur le bouton « Next »
- sur la page « Start Copying files » : un résumé des options d'installation est affiché. Cliquez sur le bouton « Next »
- les fichiers de l'application sont copiés. Une fois celle-ci terminée, la page « Installshield Wizard Complete » s'affiche. Cliquez sur le bouton « Finish ».

L'installation créé les répertoires suivants :

appdb\	contient les bases de données de type RMS des applications
apps\	contient les applications développées comme applications de démo
bin\	contient les outils du WTK
docs\	contient la documentation du WTK et des API
lib\	contient les bibliothèques des API

125.6.2. Les premiers pas

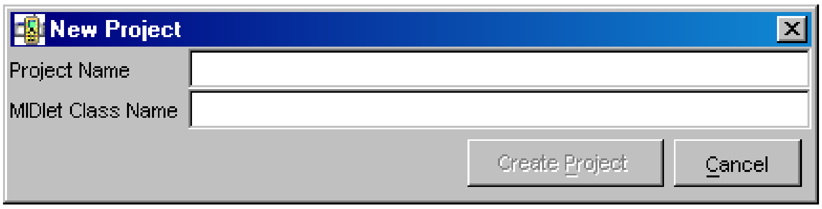
L'outil Ktoolbar est un petit IDE qui permet de compiler, pré-vérifier, packager et exécuter des applications utilisant le profile MIDP. Il ne permet pas l'édition du code des applications : il faut utiliser un éditeur externe pour réaliser cette tâche.



Copyright © 2001- 2003 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms. Third-party software, including font technology, is copyrighted and licensed from Sun suppliers. Sun, Sun Microsystems, the Sun logo, Java, J2ME, the Java Coffee Cup Logo and Forte are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Federal Acquisitions: Commercial Software - Government Users Subject to Standard License Terms and Conditions.

La première chose à faire pour créer une application est de créer un nouveau projet. Pour cela, il faut sélectionner l'option « File/New Project » du menu ou cliquer sur le bouton « New Project » dans la barre d'outils.

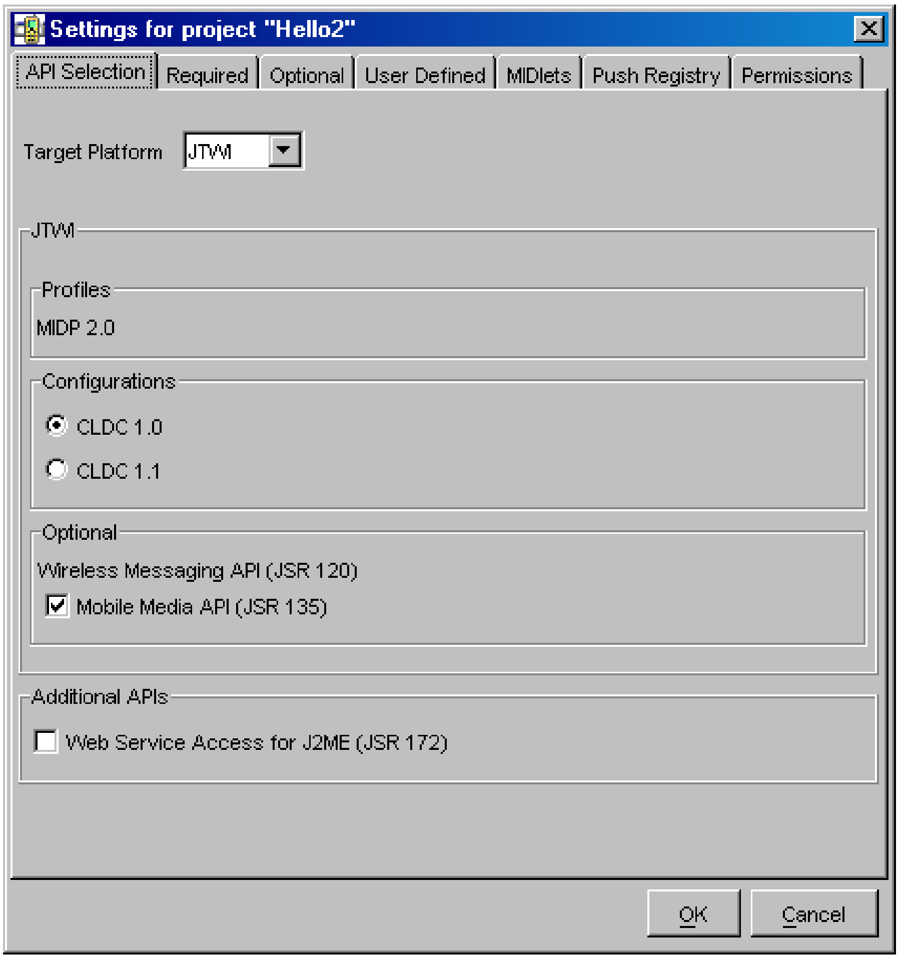


Il faut saisir le nom du projet et le nom de la classe de la Midlet.

La création du projet permet la création d'une structure de répertoires dans le sous-répertoire apps du répertoire du WTK. Dans ce répertoire apps, un répertoire est créé nommé du nom du projet. Ce répertoire contient lui-même plusieurs sous-répertoires :

%WTK%/apps/nom_projet/bin	contient le fichier jar, jad et le fichier manifest
%WTK%/apps/nom_projet/classes	contient les classes compilées
%WTK%/apps/nom_projet/lib	contient les bibliothèques utiles à l'application
%WTK%/apps/nom_projet/res	contient les ressources utiles à l'application
%WTK%/apps/nom_projet/src	contient les sources des classes

La page des propriétés du projet est différente de celle proposée dans la version 1.0. Pour l'utiliser, il faut utiliser l'option « Project/settings » ou cliquer sur le bouton « Settings » de la barre d'outils.



Target Platform JTWI

JTWI

Profiles

MIDP 2.0

Configurations

● CLDC 1.0

○ CLDC 1.1

Optional

Wireless Messaging API (JSR 120)

☑ Mobile Media API (JSR 135)

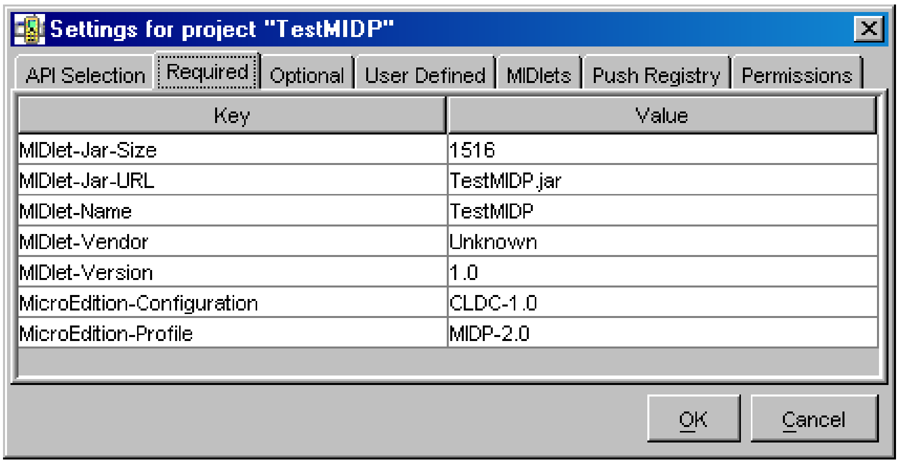
Additional APIs

□ Web Service Access for J2ME (JSR 172)

L'onglet « API sélection » permet de sélectionner la plate-forme cible ainsi que les API particulières qui vont être utilisées par l'application.

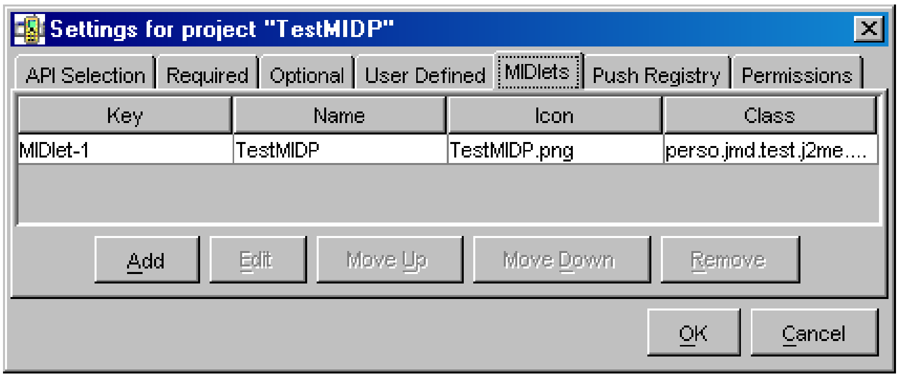
Le « target plateform » permet de sélectionner le type de plate-forme cible utilisée :

- JTWI : plate-forme répondant aux spécifications de la JSR-185
- MIDP 1.0 : plate-forme composée de CLDL 1.0 et MIDP 1.0
- Custom : plate-forme personnalisée pour laquelle il faut préciser toutes les API utilisées



Key	Value
MIDlet-Jar-Size	1516
MIDlet-Jar-URL	TestMIDP.jar
MIDlet-Name	TestMIDP
MIDlet-Vendor	Unknown
MIDlet-Version	1.0
MicroEdition-Configuration	CLDC-1.0
MicroEdition-Profile	MIDP-2.0

L'onglet « Required », « Optionnal » et « User defined » permet de préciser les attributs respectivement obligatoires, optionnels et particuliers à l'application dans le fichier manifest sous la forme de paires clé/valeur.

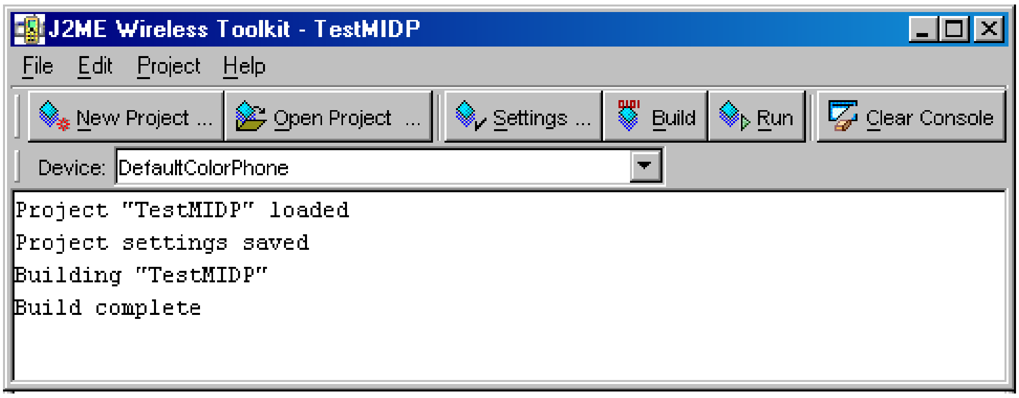


Key	Name	Icon	Class
MIDlet-1	TestMIDP	TestMIDP.png	perso.jmd.test.j2me....

L'onglet « Midlets » permet de saisir les Midlets qui composent la suite de Midlets de l'application.

Pour créer et éditer le code des classes qui composent l'application, il faut utiliser un outil externe dans le répertoire %WTK%/apps/nom_projet/src, en respectant la structure des répertoires correspondant aux packages des classes.

La compilation et la pré-vérification des sources se fait en utilisant l'option « Build » du menu « Project » ou en cliquant sur le bouton « Build ».



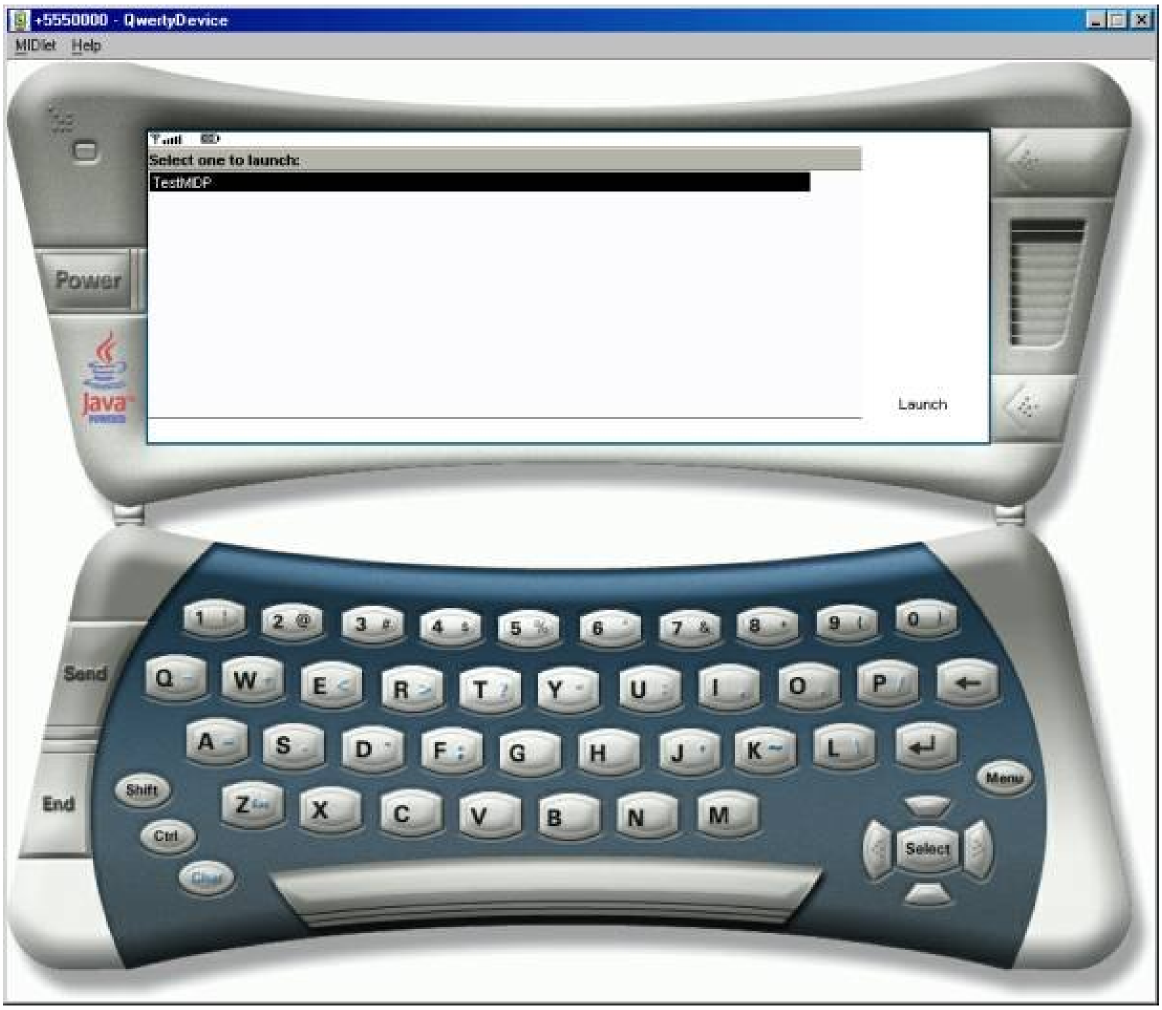
Device: DefaultColorPhone

```
Project "TestMIDP" loaded
Project settings saved
Building "TestMIDP"
Build complete
```

Si aucune erreur de compilation n'est détectée, il est possible d'exécuter le code en utilisant l'option « Run » du menu « Project » ou en cliquant sur le bouton « Run » de la barre d'outils.

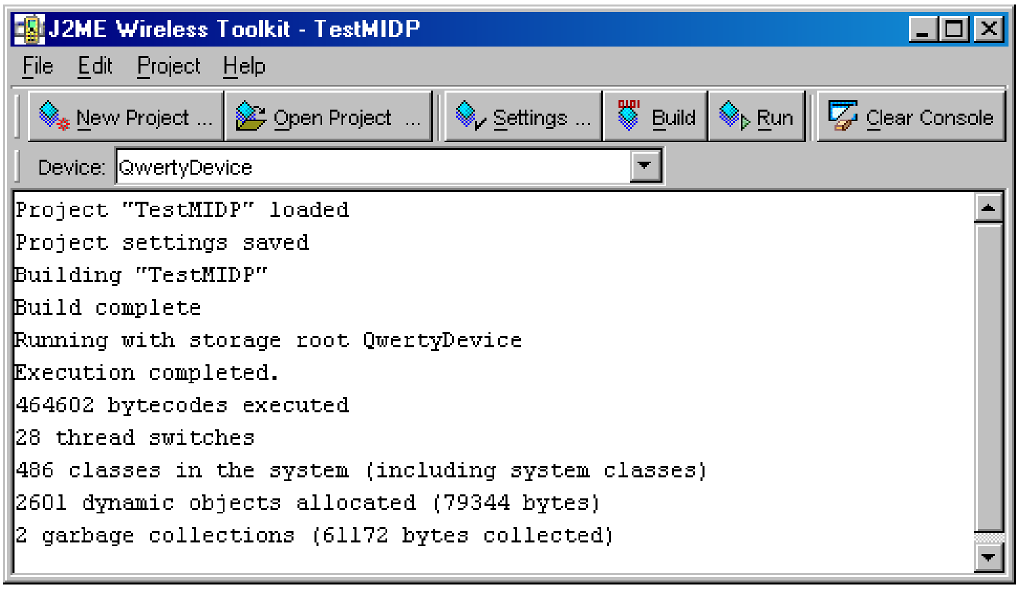
Avant de lancer l'exécution, il est possible de sélectionner l'émulateur de périphérique (device) utilisé pour exécuter le code. Le J2ME Wireless Toolkit 2.1 est fourni avec quatre émulateurs :

- DefaultColorPhone : un téléphone mobile avec un écran couleur. C'est l'émulateur par défaut.
- DefaultGrayPhone : un téléphone mobile avec un écran monochrome
- MediaControlSkin : un téléphone mobile avec des capacités multimédia accrues (video et audio)
- QwertyDevice : un périphérique avec un clavier Qwerty



Select one to launch:
TestMIDP

L'option « Clean » du menu « Project » permet de faire du ménage dans les fichiers temporaires générés lors des différents traitements.



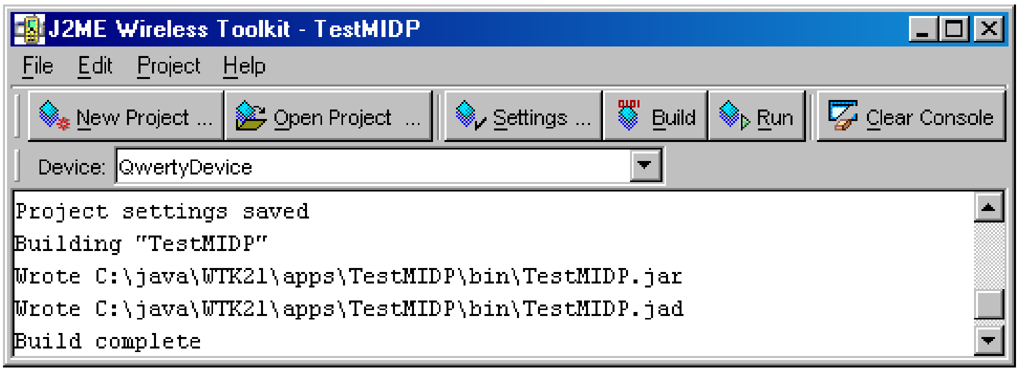
Device: QwertyDevice

```
Project "TestMIDP" loaded
Project settings saved
Building "TestMIDP"
Build complete
Running with storage root QwertyDevice
Execution completed.
464602 bytecodes executed
28 thread switches
486 classes in the system (including system classes)
2601 dynamic objects allocated (79344 bytes)
2 garbage collections (61172 bytes collected)
```

Le bouton « Clear console » permet d'effacer le contenu de la console.

L'option « Package » du menu « Project » propose deux options pour packager l'application une fois celle-ci mise au point :

- « Create package » : permet de créer un package sous la forme de fichiers .jar et .jad
- « Create obfuscated package » : permet de créer un package sous la forme d'un fichier .jad et d'un fichier .jar plus compact, et ceci grâce à un outil tiers non fourni réalisant l'opération d'obscurcissement



```
Project settings saved
Building "TestMIDP"
Wrote C:\java\WTK21\apps\TestMIDP\bin\TestMIDP.jar
Wrote C:\java\WTK21\apps\TestMIDP\bin\TestMIDP.jad
Build complete
```

126. CLDC

Chapitre 126

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

La suite de ce chapitre sera développée dans une version future de ce document

L'API du CLDC se compose de quatre packages :

- java.io : classes pour la gestion des entrées / sorties par flux
- java.lang : classes de base du langage java
- java.util : classes utilitaires notamment pour gérer les collections, la date et l'heure, ...
- javax.microedition.io : classes pour gérer des connections génériques

Ils ont des fonctionnalités semblables à ceux proposés par J2SE avec quelques restrictions, notamment il n'y a pas de gestion des nombres flottants dans CLDC 1.0.

De nombreuses classes sont définies dans J2SE et J2ME mais souvent elles possèdent moins de fonctionnalités dans l'édition mobile.

La version courant de CLDC est la 1.1 dont les spécifications sont les résultats des travaux de la JSR 139.

Ce chapitre contient plusieurs sections :

- Le package java.lang
- Le package java.io
- Le package java.util
- Le package javax.microedition.io

126.1. Le package java.lang

Il définit l'interface Runnable ainsi que les classes suivantes :

Nom	Rôle
Boolean	Classe qui encapsule une valeur du type booléen
Byte	Classe qui encapsule une valeur du type byte
Character	Classe qui encapsule une valeur du type char
Class	Classe qui encapsule une classe ou une interface

Integer	Classe qui encapsule une valeur du type int
Long	Classe qui encapsule une valeur du type long
Math	Classe qui contient des méthodes statiques pour les calculs mathématiques
Object	Classe mère de toutes les classes
Runtime	Classe qui permet des interactions avec le système d'exploitation
Short	Classe qui encapsule une valeur du type short
String	Classe qui encapsule une chaîne de caractères immuable
StringBuffer	Classe qui encapsule une chaîne de caractères
System	
Thread	Classe qui encapsule un traitement exécuté dans un thread
Throwable	Classe mère de toutes les exceptions et des erreurs

Il définit les exceptions suivantes : ArithmeticException, ArrayIndexOutOfBoundsException, ArrayStoreException, ClassCastException, ClassNotFoundException, Exception, IllegalAccessException, IllegalArgumentException, IllegalMonitorStateException, IllegalThreadStateException, IndexOutOfBoundsException, InstantiationException, InterruptedException, NegativeArraySizeException, NullPointerException, NumberFormatException, RuntimeException, SecurityException, StringIndexOutOfBoundsException

Il définit les erreurs suivantes : Error, OutOfMemoryError, VirtualMachineError

126.2. Le package java.io

Il définit les interfaces suivantes : DataInput, DataOutput ainsi que les classes suivantes :

Nom	Rôle
ByteArrayInputStream	Lecture d'un flux d'octets bufférisé
ByteArrayOutputStream	Ecriture d'un flux d'octets bufférisé
DataInputStream	Lecture de données stockées au format Java
DataOutputStream	Ecriture de données stockées au format Java
InputStream	Classe abstraite dont héritent toutes les classes gérant la lecture de flux par octets
InputStreamReader	Lecture d'octets sous la forme de caractères
OutputStream	Classe abstraite dont hérite toutes les classes gérant l'écriture de flux par octets
OutputStreamWriter	Ecriture de caractères sous la forme d'octets
PrintStream	
Reader	Classe abstraite dont héritent toutes les classes gérant la lecture de flux par caractères
Writer	Classe abstraite dont héritent toutes les classes gérant l'écriture de flux par caractères

Il définit les exceptions suivantes : EOFException, InterruptedIOException, IOException, UnsupportedEncodingException, UTFDataFormatException

126.3. Le package java.util

Il définit l'interface Enumeration ainsi que les classes suivantes :

Nom	Rôle
Calendar	Classe abstraite pour manipuler les éléments d'une date
Date	Classe qui encapsule une date
Hashtable	Classe qui encapsule une collection d'éléments composée de paire clé/valeur
Random	Classe qui permet de générer des nombres aléatoires
Stack	Classe qui encapsule une collection de type pile LIFO
TimeZone	Classe qui encapsule un fuseau horaire
Vector	Classe qui encapsule une collection de type tableau dynamique

Il définit les exceptions EmptyStackException et NoSuchElementException

126.4. Le package javax.microedition.io

Il définit les interfaces suivantes :

Nom	Rôle
Connection	Interface pour une connexion générique
ContentConnection	
Datagram	Interface pour un paquet de données
DatagramConnection	Interface pour une connexion utilisant des paquets de données
InputConnection	Interface pour une connexion entrante
OutputConnection	Interface pour une connexion sortante
StreamConnection	Interface pour une connexion utilisant un flux
StreamConnectionNotifier	

127. MIDP

Chapitre 127

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

C'est le premier profile qui a été développé dont l'objectif principal est le développement d'application sur des machines aux ressources et à l'interface limitées tel qu'un téléphone cellulaire. Ce profil peut aussi être utilisé pour développer des applications sur des PDA de type Palm.

L'API du MIDP se compose des API du CDLC et de trois packages :

- javax.microedition.midlet : cycle de vie de l'application
- javax.microedition.lcdui : interface homme machine
- javax.microedition.rms : persistance des données

Des informations complémentaires et le téléchargement de l'implémentation de référence de ce profil peuvent être trouvés sur le site : https://jcp.org/aboutJava/communityprocess/final/jsr118/index.html

Il existe deux versions du MIDP :

- 1.0 : la dernière révision est la 1.0.3 dont les spécifications sont issues de la JSR 37
- 2.0 : c'est la version la plus récente dont les spécifications sont issues de la JSR 118

Ce chapitre contient plusieurs sections :

- ♦ Les Midlets
- ♦ L'interface utilisateur
- ♦ La gestion des événements
- ♦ Le stockage et la gestion des données
- ♦ Les suites de midlets
- ♦ Packager une midlet
- ♦ MIDP for Palm O.S.

127.1. Les Midlets

Les applications créées avec MIDP sont des midlets : ce sont des classes qui héritent de la classe abstraite javax.microedition.midlet.Midlet. Cette classe permet le dialogue entre le système et l'application.

Elle possède trois méthodes qui permettent de gérer le cycle de vie de l'application en fonction des trois états possibles (active, suspendue ou détruite) :

- startApp() : cette méthode est appelée à chaque démarrage ou redémarrage de l'application
- pauseApp() : cette méthode est appelée lors de la mise en pause de l'application
- destroyApp() : cette méthode est appelée lors de la destruction de l'application

Ces trois méthodes doivent obligatoirement être redéfinies.

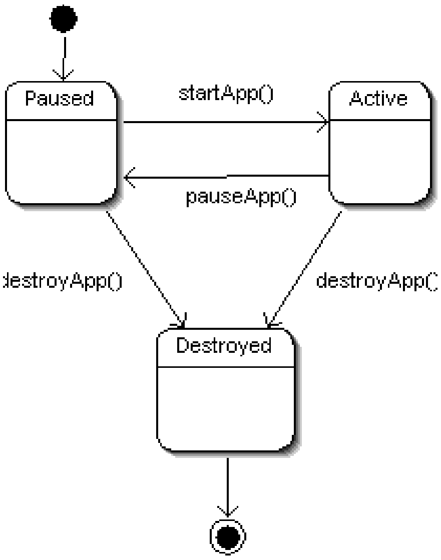
Exemple (MIDP 1.0) :

```
package fr.jmdoudoux.dej.j2me;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class Test extends MIDlet {

  public Test() {
  }

  public void startApp() {
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
  }
}
```

Le cycle de vie d'une midlet est semblable à celui d'une applet. Elle possède plusieurs états :

- paused :
- active :
- destroyed :

Le changement de l'état de la midlet peut être provoqué par l'environnement d'exécution ou la midlet.



La méthode startApp() est appelée lors du démarrage ou redémarrage de la midlet. Il est important de comprendre que cette méthode est aussi appelée lors du redémarrage de la midlet : elle peut donc être appelée plusieurs fois au cours de son exécution.

Les méthodes pauseApp() et destroyApp() sont appelées respectivement lors de mise en pause de la midlet et juste avant la destruction de la midlet.

127.2. L'interface utilisateur

Les possibilités concernant l'IHM de MIDP sont très réduites pour permettre une exécution sur un maximum de machines allant du téléphone portable au PDA. Ces machines présentent des contraintes fortes concernant l'interface qu'elles proposent à leurs utilisateurs.

Avec le J2SE, deux API permettent le développement d'IHM : AWT et Swing. Ces deux API proposent des composants pour développer des interfaces graphiques riches de fonctionnalités avec un modèle de gestion des événements complet. Ils prennent en compte un système de pointage par souris, avec un écran couleur possédant de nombreuses couleurs et une résolution importante.

Avec MIDP, le nombre de composants et le modèle de gestion des événements sont spartiates. Il ne prend en compte qu'un écran tactile souvent monochrome ayant une résolution très faible. Avec un clavier limité en nombres de touches et dépourvu de système de pointage, la saisie de données sur de tels appareils est particulièrement limitée.

L'API pour les interfaces utilisateurs du MIDP est regroupée dans le package javax.microedition.lcdui.

Elle se compose des éléments de haut niveaux et des éléments de bas niveaux.

127.2.1. La classe Display

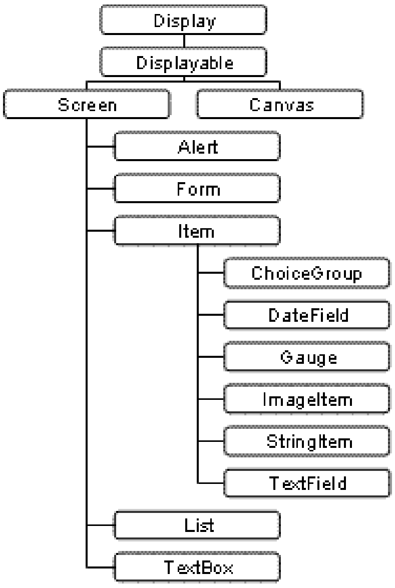
Pour pouvoir utiliser les éléments graphiques, il faut obligatoirement obtenir un objet qui encapsule l'écran. Un tel objet est du type de la classe Display. Cette classe possède des méthodes pour afficher les éléments graphiques.

La méthode statique getDisplay() renvoie une instance de la classe Display qui encapsule l'écran associé à la midlet fournie en paramètre de la méthode.

Exemple (MIDP 1.0) :

```
package fr.jmdoudoux.dej.j2me;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class Hello extends MIDlet {

  private Display display;

  public Hello() {
    display = Display.getDisplay(this);
  }

  public void startApp() {
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
 }
}
```

Les éléments de l'interface graphique appartiennent à une hiérarchie d'objets : tous les éléments affichables héritent de la classe abstraite Displayable.



La classe Screen est la classe mère des éléments graphiques de haut niveau. La classe Canvas est la classe mère des éléments graphiques de bas niveau.

Il n'est pas possible d'ajouter directement un élément graphique dans un Display sans qu'il soit inclus dans un objet héritant de Displayable.

Un seul objet de type Displayable peut être affiché à la fois. La classe Display possède la méthode getCurrent() pour connaître l'objet couramment affiché et la méthode setCurrent() pour afficher l'objet fourni en paramètre.

127.2.2. La classe TextBox

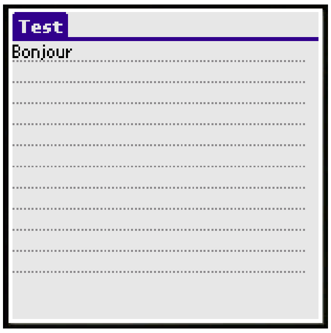
Ce composant permet de saisir du texte.

Exemple (MIDP 1.0) :

```
package fr.jmdoudoux.dej.j2me;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class Hello extends MIDlet {

  private Display display;
  private TextBox textbox;

  public Hello() {
    display = Display.getDisplay(this);
    textbox = new TextBox("", "Bonjour", 20, 0);
  }

  public void startApp() {
    display.setCurrent(textbox);
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
  }
}
```

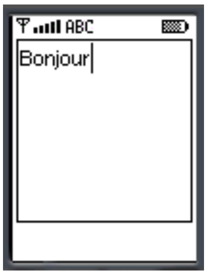
Résultat :

sur l'émulateur Palm



Bonjour

sur l'émulateur de téléphone mobile



Bonjour

127.2.3. La classe List

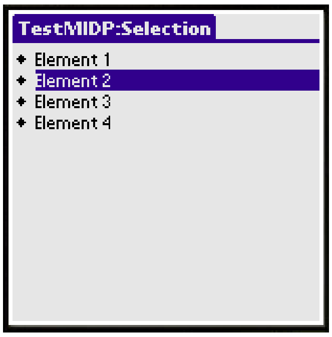
Ce composant permet la sélection d'un ou plusieurs éléments dans une liste d'éléments.

Exemple (MIDP 1.0) :

```
package fr.jmdoudoux.dej.j2me;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class Test extends MIDlet {
  private Display display;
  private List liste;

  protected static final String[] elements = {"Element 1",
                                          "Element 2",
                                          "Element 3",
                                          "Element 4"};

  public Test() {
    display = Display.getDisplay(this);
    liste = new List("Selection", List.EXCLUSIVE, elements, null);;
  }

  public void startApp() {
    display.setCurrent(liste);
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
  }
}
```

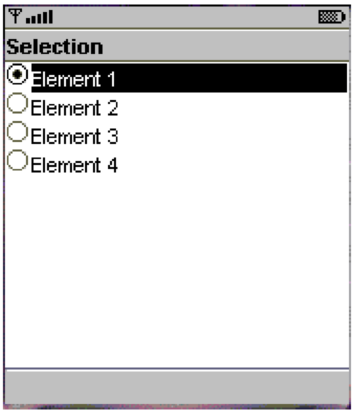
Résultat :

sur l'émulateur Palm



- Element 1
- Element 2
- Element 3
- Element 4

sur l'émulateur de téléphone mobile


Selection

○ Element 1

○ Element 2

○ Element 3

○ Element 4

La suite de cette section sera développée dans une version future de ce document

127.2.4. La classe Form

La classe From permet d'insérer dans l'élément graphique qu'elle représente d'autres éléments graphiques : cette classe sert de conteneurs. Les éléments insérés sont des objets qui héritent de la classe abstraite Item.

Exemple (MIDP 1.0) :

```
package fr.jmdoudoux.fr.j2me;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class Hello extends MIDlet {

  private Display display;
  private Form mainScreen;

  public Hello() {
    display = Display.getDisplay(this);
  }

  public void startApp() {
    mainScreen = new Form("Hello");
    mainScreen.append("Bonjour");
   display.setCurrent(mainScreen);
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
  }
}
```

127.2.5. La classe Item

La classe javax.microedition.lcdui.Item est la classe mère de tous les composants graphiques qui peuvent être insérés dans un objet de type Form.

Cette classe définit seulement deux méthodes, getLabel() et setLabel() qui sont le getter et le setter pour la propriété label.

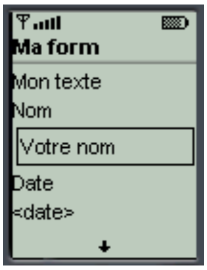
Il existe plusieurs composants qui héritent de la classe Item

Classe	Rôle
ChoiceGroup	sélection d'un ou plusieurs éléments
DateField	affichage et saisie d'une date
Gauge	affichage d'une barre de progression
ImageItem	affichage d'une image
StringItem	affichage d'un texte
TextField	saisie d'un texte

Exemple (MIDP 1.0) :

```
package fr.jmdoudoux.fr.j2me;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class Hello extends MIDlet {

  private Display display;
  private Form form;
  private ChoiceGroup choiceGroup;
  private DateField dateField;
  private DateField timeField;
  private Gauge gauge;
  private StringItem stringItem;
  private TextField textField;

  public Hello() {
    display = Display.getDisplay(this);
    form = new Form("Ma form");

    String choix[] = {"Choix 1", "Choix 2"};
    stringItem = new StringItem(null,"Mon texte");
    choiceGroup = new ChoiceGroup("Sélectionner",Choice.EXCLUSIVE,choix,null);
    dateField = new DateField("Heure",DateField.TIME);
    timeField = new DateField("Date",DateField.DATE);
    gauge = new Gauge("Avancement",true,10,1);
    textField = new TextField("Nom","Votre nom",20,0);

    form.append(stringItem);
    form.append(choiceGroup);
    form.append(timeField);
    form.append(dateField);
    form.append(gauge);
    form.append(textField);
  }

  public void startApp() {
    display.setCurrent(form);
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
  }
}
```

Résultat sur l'émulateur de téléphone mobile :



Ma form

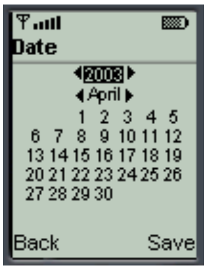
Mon texte

Nom

Votre nom

Date

<date>

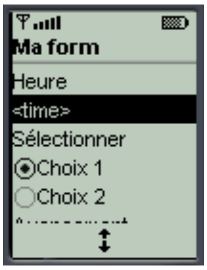


Date

2003

April

```
         1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30
```



Ma form

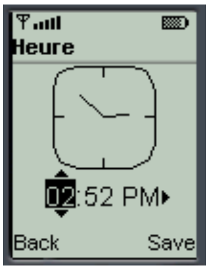
Heure

<time>

Sélectionner

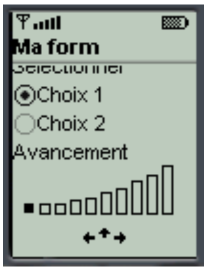
◉Choix 1

○Choix 2



Heure

02:52 PM



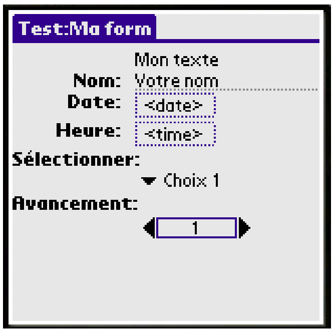
Ma form

◉Choix 1

○Choix 2

Avancement

Résultat sur l'émulateur Palm OS :



Test:Ma form

Mon texte

Nom: Votre nom

Date: <date>

Heure: <time>

Sélectionner:

Choix 1

Avancement:

1

127.2.6. La classe Alert

Cette classe permet d'afficher une boîte de dialogue pendant un temps déterminé.

Elle possède deux constructeurs :

- l'un demandant le titre de l'objet
- l'autre le titre, le texte, l'image et le type de l'image

Elle possède des getters et des setters sur chacun de ces éléments.

Pour préciser le type de la boîte de dialogue, il faut utiliser une des constantes définies dans la classe AlertType dans le constructeur ou dans la méthode setType() :

Constante	type de la boîte de dialogue
ALARM	informer l'utilisateur d'un événement programmé
CONFIRMATION	demander la confirmation à l'utilisateur
ERROR	informer l'utilisateur d'une erreur
INFO	informer l'utilisateur
WARNING	informer l'utilisateur d'un avertissement

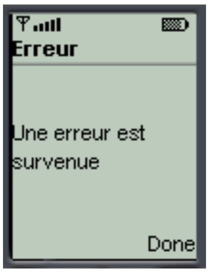
Pour afficher un objet de type Alert, il faut utiliser une version surchargée de la méthode setCurrent() de l'instance de la classe Display. Cette version nécessite deux paramètres : l'objet Alert à afficher et l'objet de type Displayable qui sera affiché lorsque l'objet Alert sera fermé.

La méthode setTimeout() qui attend un entier en paramètre permet de préciser la durée d'affichage en milliseconde de la boîte de dialogue. Pour la rendre modale, il faut lui passer le paramètre Alert.FOREVER.

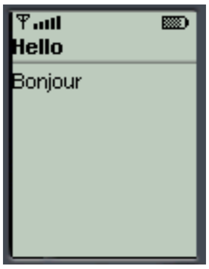
Exemple (MIDP 1.0) :

```
package fr.jmdoudoux.dej.j2me;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class Test extends MIDlet {

  private Display display;
private Alert alert;
  private Form form;

  public Test() {
    display = Display.getDisplay(this);
    form = new Form("Hello");
    form.append("Bonjour");

   alert = new Alert("Erreur", "Une erreur est survenue", null, AlertType.ERROR);
   alert.setTimeout(Alert.FOREVER);
  }

  public void startApp() {
    display.setCurrent(alert, form);
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
  }
}
```

Résultat sur l'émulateur de téléphone mobile:

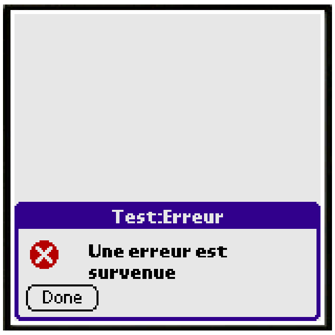


Une erreur est survenue

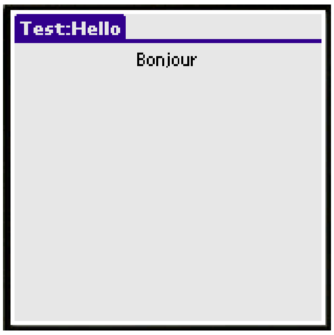


Bonjour

Résultat sur l'émulateur Palm OS:



Une erreur est survenue



Bonjour

127.3. La gestion des événements

Les interactions entre l'utilisateur et l'application se concrétisent par le traitement d'événements particuliers pour chaque action.

MIDP définit des interfaces de type Listener pour la gestion des événements :

Interface	Rôle

CommandListener	Listener pour une activation d'une commande
ItemStateListener	Listener pour un changement d'état d'un composant(modification du texte d'une zone de texte, ...)

Cette section sera développée dans une version future de ce document

127.4. Le stockage et la gestion des données

Avec MIDP, le mécanisme pour la persistance des données est appelé RMS (Record Management System). Il permet le stockage de données et leur accès ultérieur.

RMS propose un accès standardisé au système de stockage de la machine dans laquelle s'exécute le programme. Il n'impose pas aux constructeurs la façon dont les données doivent être stockées physiquement.

Du fait de la simplicité des mécanismes utilisés, RMS ne définit qu'une seule classe : RecordStore. Cette classe ainsi que les interfaces et les exceptions qui composent RMS sont regroupées dans le package javax.microedition.rms.

Les données sont stockées dans un ensemble d'enregistrements (records). Un enregistrement est un tableau d'octets. Chaque enregistrement possède un identifiant unique nommé recordId qui permet de retrouver un enregistrement particulier.

A chaque fois qu'un ensemble de données est modifié (ajout, modification ou suppression d'un enregistrement), son numéro de version est incrémenté.

Un ensemble de données est associé à un unique ensemble composé d'une ou plusieurs Midlets (Midlet Suite).

Un ensemble de données possède un nom composé de 32 caractères maximum.

127.4.1. La classe RecordStore

L'accès aux données se fait obligatoirement en utilisant un objet de type RecordStore.

Les principales méthodes sont :

Méthode	Rôle
int addRecord(byte[],int, int)	Ajouter un nouvel enregistrement
void addRecordListener(RecordListener)	
void closeRecordStore()	Fermer l'ensemble d'enregistrements
void deleteRecord(int)	Supprimer l'enregistrement dont l'identifiant est fourni en paramètre
static void deleteRecordStore(String)	Supprimer l'ensemble d'enregistrements dont le nom est fourni en paramètre
Enumeration enumerateRecords(RecordFilter , RecordComparator, boolean)	Renvoyer une énumération pour parcourir tout ou partie de l'ensemble
String getName()	Renvoyer le nom de l'ensemble d'enregistrements
int getNextRecordID()	Renvoyer l'identifiant du prochain enregistrement créé
int getNumRecords()	Renvoyer le nombre d'enregistrements contenu dans l'ensemble

byte[] getRecord(int)	Renvoyer l'enregistrement dont l'identifiant est fourni en paramètre
int getRecord(int, byte[], int)	Obtenir les données contenues dans un enregistrement dont l'identifiant est fourni en paramètre. Renvoie le nombre d'octets de l'enregistrement
int getRecordSize(int)	Renvoyer la taille en octets de l'enregistrement dont l'identifiant est fourni en paramètre
int getSize()	Renvoyer la taille en octets occupée par l'ensemble
static String[] listRecordStores()	Renvoyer un tableau de chaînes de caractères contenant les noms des ensembles de données associés au Midlet courant
static RecordStore openRecordStore(String, boolean)	Ouvrir un ensemble de données dont le nom est fourni en paramètre. Celui-ci est créé s'il n'existe pas et que le booléen est à true
void setRecord(int, byte[], int, int)	Mettre à jour l'enregistrement précisé avec les données fournies en paramètre

Pour pouvoir utiliser un ensemble d'enregistrements, il faut utiliser la méthode statique openRecordStore() en fournissant le nom de l'ensemble et un booléen qui précise si l'ensemble doit être créé au cas où celui-ci n'existerait pas. Elle renvoie un objet RecordStore qui encapsule l'ensemble d'enregistrements.

L'appel de cette méthode peut lever l'exception RecordStoreNotFoundException si l'ensemble n'est pas trouvé, RecordStoreFullException si l'ensemble de données est plein ou RecordStoreException dans les autres cas problématiques.

La méthode closeRecordStore() permet de fermer un ensemble précédemment ouvert. Elle peut lever les exceptions RecordStoreNotOpenException et RecordStoreException.

La suite de cette section sera développée dans une version future de ce document

127.5. Les suites de midlets

Cette section sera développée dans une version future de ce document

127.6. Packager une midlet

Une application constituée d'une suite de midlets est packagée sous la forme d'une archive .jar. Cette archive doit contenir un fichier manifest et tous les éléments nécessaires à l'exécution de l'application (fichiers .class et les ressources telles que les images, ...).

127.6.1. Le fichier manifest

Ce fichier contient des informations sur l'application.

Ce fichier contient une définition des propriétés utilisées par l'application. Ces propriétés sont sous la forme clé/valeur.

Plusieurs propriétés sont définies par les spécifications des midlets : celles-ci commencent par MIDlet-.

Propriétés	Rôle
MIDlet-Name	Nom de l'application
MIDlet-Version	Numéro de version de l'application
MIDlet-Vendor	Nom du fournisseur de l'application
MIDlet-Icon	Nom du fichier .png contenant l'icône de l'application
MIDlet-Description	Description de l'application
MIDlet-Info-URL	
MIDlet-Jar-URL	URL de téléchargement de fichier jar
MIDlet-Jar-Size	taille en octets du fichier .jar
MIDlet-Data-Profile	
MicroEdition-Configuration	

Il est possible de définir ses propres attributs

127.7. MIDP for Palm O.S.

MIDP for Palm O.S. est une implémentation particulière du profile MIDP pour le déploiement et l'exécution d'applications sur des machines de type Palm. Elle permet d'exécuter des applications écrites avec MIDP sur un PALM possédant une version 3.5 ou supérieure de cet O.S.

Cette implémentation remplace l'ancienne implémentation développée par Sun nommée KJava.

127.7.1. L'installation

MIDP for Palm O.S. n'est plus téléchargeable : il fallait télécharger le fichier midp4palm-1_0.zip et sa documentation dans le fichier midp4palm-1_0-doc.zip.

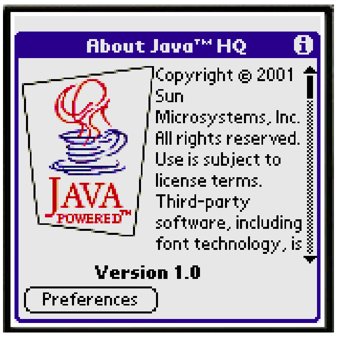
L'installation comprend une partie sur le poste de développement PC et une partie sur la machine Palm pour les tests d'exécution.

Pour pouvoir utiliser MIDP for Palm O.S., il faut déjà avoir installé CLDC et MIDP.

Il faut commencer l'installation sur le PC en décompressant les deux fichiers dans un répertoire.

Pour pouvoir exécuter les applications sur le Palm, il faut installer le fichier MIDP.prc contenu dans le répertoire PRCFiles sur le Palm en procédant comme pour toute application Palm.

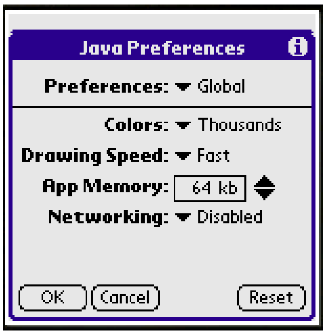
En cliquant sur l'icône, on peut régler différents paramètres.



Copyright © 2001 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms. Third-party software, including font technology, is

Version 1.0

Un clic sur le bouton "Preferences" permet de modifier ces paramètres.



Preferences: Global

Colors: Thousands

Drawing Speed: Fast

App Memory: 64 kb

Networking: Disabled

127.7.2. La création d'un fichier .prc

MIDP for Palm O.S. fournit un outil pour transformer les fichiers .jad et .jar qui composent une application J2ME en un fichier .prc directement installable sur un Palm.

Sous Windows, il suffit d'exécuter le programme converter.bat situé dans le sous-répertoire Converter du répertoire d'installation.

Il faut que la variable d'environnement JAVA_PATH pointe vers le répertoire d 'installation d'un JDK 1.3. minimum. Si ce n'est pas le cas, un message d'erreur est affiché.

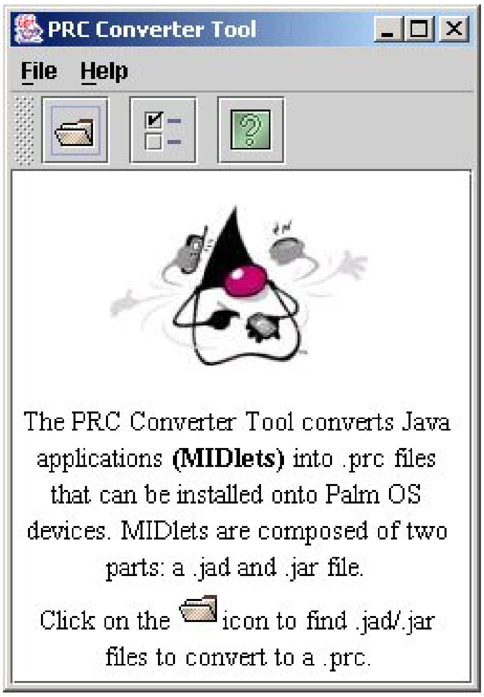
```
Error: Java path is missing in your environment
         Please set JAVA_PATH to point to your Java directory
         e.g. set JAVA_PATH=c:\bin\jdk1.3\
```

Si tout est correct, l'application se lance.

Sun microsystems
JAVA
FORM_HIDE_EVENT FORM_SETUP_EVENT
FORM_SCROLL_EVENT FORM_HEIGHT_EVENT FORM_PA

PRC CONVERTER TOOL

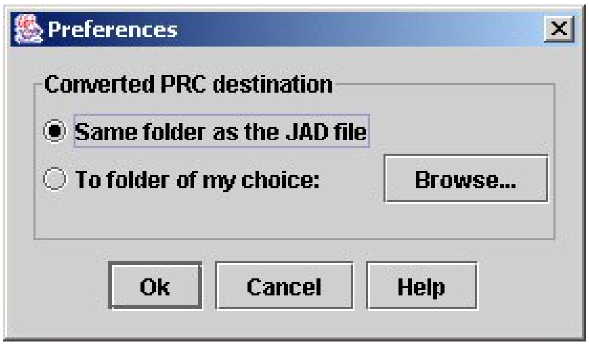
© Copyright 2001 Sun Microsystems, Inc. All rights reserved.
Use is subject to license terms.



The PRC Converter Tool converts Java applications **(MIDlets)** into .prc files that can be installed onto Palm OS devices. MIDlets are composed of two parts: a .jad and .jar file.

Click on the 📁 icon to find .jad/.jar files to convert to a .prc.

Il est possible de préciser le répertoire du ou des fichiers .prc générés en utilisant l'option "Preference" du menu "File" :



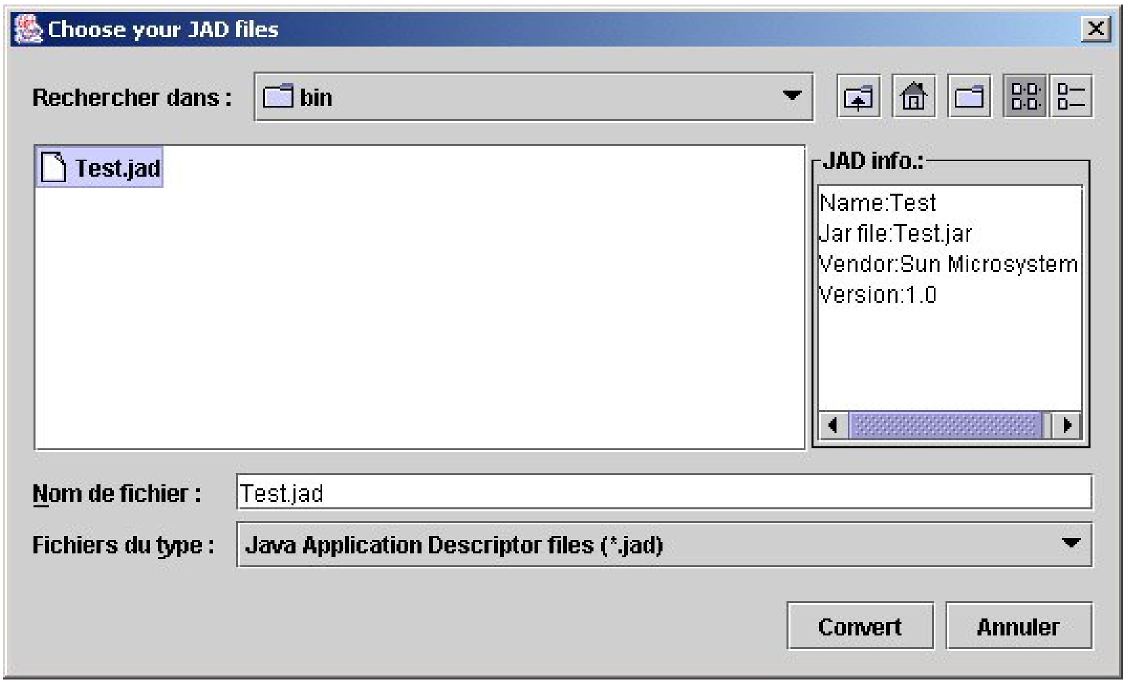
Converted PRC destination

● Same folder as the JAD file

○ To folder of my choice:

Une boîte de dialogue permet de choisir entre le même répertoire que celui qui contient le fichier .jad ou de sélectionner un répertoire quelconque.

Il suffit de cliquer sur l'icône en forme de répertoire dans la barre d'icônes pour sélectionner le fichier .jad. Les fichiers .jad et .jar de l'application doivent obligatoirement être dans le même répertoire.



Rechercher dans : bin

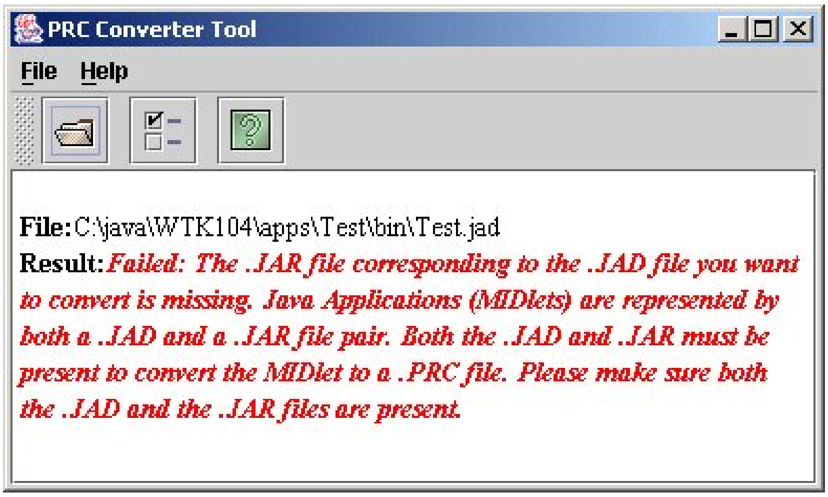
Name:Test
Jar file:Test.jar
Vendor:Sun Microsystem
Version:1.0

Nom de fichier : Test.jad

Fichiers du type : Java Application Descriptor files (*.jad)

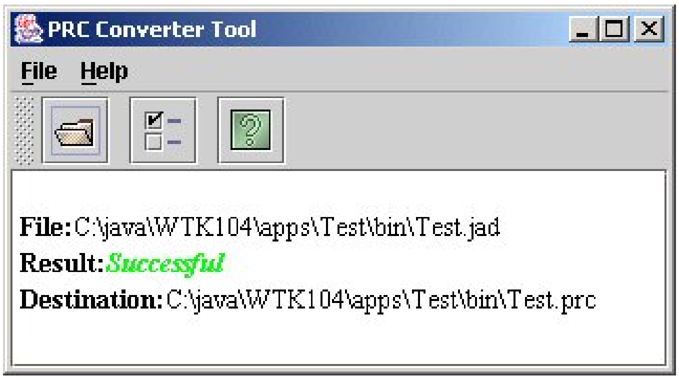
Un clic sur le bouton "Convert", lance la conversion.

Si la conversion échoue, un message d'erreur est affiché. Exemple, si le fichier .jar correspond au fichier .jad est absent, alors le message suivant est affiché :



File: C:\java\WTK104\apps\Test\bin\Test.jad
Result: *Failed: The .JAR file corresponding to the .JAD file you want to convert is missing. Java Applications (MIDlets) are represented by both a .JAD and a .JAR file pair. Both the .JAD and .JAR must be present to convert the MIDlet to a .PRC file. Please make sure both the .JAD and the .JAR files are present.*

Si toutes les opérations se sont correctement passées, alors un message récapitulatif est affiché :



File: C:\java\WTK104\apps\Test\bin\Test.jad
Result: *Successful*
Destination: C:\java\WTK104\apps\Test\bin\Test.prc

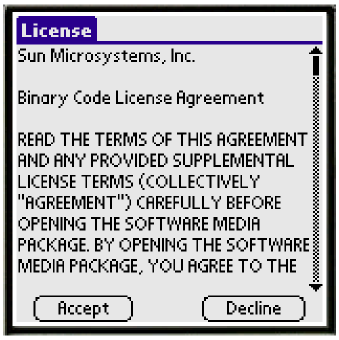
127.7.3. L'installation et l'exécution d'une application

Une fois le fichier .prc créé, il suffit d'utiliser la procédure standard d'installation d'un tel fichier sur le Palm (ajouter le fichier dans la liste avec "l'outil d'installation" du Palm et lancer une synchronisation).

Une fois l'application installée, l'icône de l'application apparaît.

Pour exécuter l'application, il suffit comme pour une application native, de cliquer sur l'icône.

Lors de la première exécution, il faut lire et valider la licence d'utilisation.

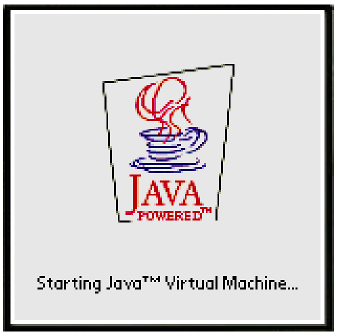


Sun Microsystems, Inc.

Binary Code License Agreement

READ THE TERMS OF THIS AGREEMENT AND ANY PROVIDED SUPPLEMENTAL LICENSE TERMS (COLLECTIVELY "AGREEMENT") CAREFULLY BEFORE OPENING THE SOFTWARE MEDIA PACKAGE. BY OPENING THE SOFTWARE MEDIA PACKAGE, YOU AGREE TO THE

Une splash screen s'affiche durant le lancement de la machine virtuelle.



Starting Java™ Virtual Machine...

Puis l'application s'exécute.

128. CDC

Chapitre 128

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Cette configuration se destine à l'utilisation de Java sur des machines mobiles possédant un processeur 32 bits, au moins 2Mo de RAM et une connexion au réseau.

CDC est une spécification définit par la JSR numéro 036.

La machine virtuelle utilise par le CDC est nommé CVM. Elle respecte intégralement les spécifications de la plate-forme Java 2 version 1.3.

Le CDC ne peut être utilisé seul : il faut lui adjoindre un ou plusieurs profiles qui lui sont spécifiques.

Le CDC définit aussi un ensemble d'API de base :

- java.lang
- java.util
- java.net
- java.io
- java.text
- java.security

Le contenu de ces packages est très proche de celui de la plate-forme J2SE excepté quelques exceptions et surtout la suppression de toutes les API déclarées deprecated.

La version 1.1 du CDC est en cours de spécification dans la JSR 218

La suite de ce chapitre sera développée dans une version future de ce document

129. Les profils du CDC

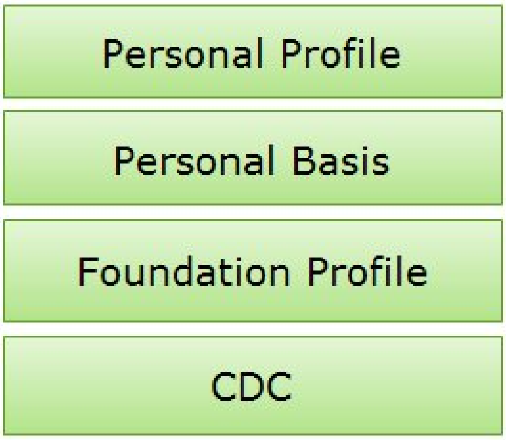
Chapitre 129

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Plusieurs profiles reposent sur la configuration CDC :

- Foundation Profile
- Personal Basis Profile
- Personal Profile

Ces profiles peuvent être utilisés et cumulées en fonction des besoins. Exemple : CDC / Foundation Profile / Personal Basis / Personnal Profile.



Le Foundation Profile ajoute à CDC des classes de Java SE notamment relatives à la sécurité, aux locales et des utilitaires.

Le Personal Basis Profile repose sur le Foundation Profile et ajoute des classes de bases pour les interfaces graphiques.

Le Personal Profile repose sur le Personal Basis Profile et ajoute des classes qui représentent un sous-ensemble de l'API AWT pour les interfaces graphiques.

Ce chapitre contient plusieurs sections :

♦ Foundation profile
♦ Le Personal Basis Profile (PBP)
♦ Le Personal Profile (PP)

129.1. Foundation profile

Ce profile sert de base pour le développement d'applications sur des outils mobiles utilisant la configuration CDC tels que des Pockets PC ou des Tablets PC.

Le but du Foundation Profile est de servir de support pour le développement d'autres profiles.

Ce profile ne propose aucune classe pour les interfaces graphiques. Une partie importante de ce profile concerne les différentes formes de connexions au réseau.

Package	Description
java.lang	Classes de base
java.lang.ref	Classes pour les différents types de références d'objets
java.lang.reflect	Classes et interfaces pour utiliser l'introspection
java.math	Classes pour les calculs entier (BigInteger)
java.text	Classes et interfaces pour formatter les textes, dates et nombres
java.util	Classes utilitaires (collections, dates, I18n, ...)
java.util.jar	Classes pour lire et écrire des archives jar
java.util.zip	Classes pour lire et écrire des archives zip
java.io	Classes et interfaces pour les entrées/sorties
java.net	Classes pour les interactions avec le réseaux : support des protocoles datagram, socket et http
java.security	Classes et interfaces pour le framework de sécurité
java.security.acl	Classes et interface pour mettre en oeuvre Access Control List
java.security.cert	Classes et interfaces pour gérer et utiliser les certificats
java.security.interfaces	Interfaces pour générer des clés RSA et DSA
java.security.spec	Classes et interfaces pour les clés de certains algorithmes
javax.microedition.io	Classes et interfaces reposant sur le Generic Connection Framework proposant un support des protocoles datagram, socket, file et http

La version 1.1 du Foundation Profile est essentiellement une adaptation de son API sur celle de J2SE 1.4

129.2. Le Personal Basis Profile (PBP)

Ce profile contient les éléments de bases pour développer une interface graphique avec le CDC et le Foundation Profile : son but principal est de proposer un support minimum pour les interfaces graphiques sous la forme d'un sous-ensemble de l'API AWT. Ce profile propose un support pour les applications de type Xlet.

Le Personal Basis Profile repose sur le Foundation Profile.

La version 1.0 du Personal Basis Profile est spécifiée par la JSR 129.

La version 1.1 du Personal Basis Profile est spécifiée par la JSR 217.

Package	Description
java.awt	Classes pour créer des interfaces graphiques simples
java.awt.color	Classes pour utiliser les couleurs
java.awt.event	Classes et interfaces pour la gestion des événements des composants graphiques
java.awt.image	Classes pour utiliser les images
java.beans	Classes pour utiliser les Javabeans
java.rmi	Classes pour utiliser RMI

java.rmi.registry	Classes pour utiliser RMI
javax.microedition.xlet	Classes pour créer des Xlets
javax.microedition.xlet.ixc	Classes pour communiquer entre Xlets

La version 1.1 du Personal Basis Profile est essentiellement une adaptation de son API sur celle de J2SE 1.4.

129.3. Le Personal Profile (PP)

Ce profile se destine au développement d'applications sur des PDA disposant de ressources importantes tels que les Pockets PC. Ce profile permet notamment le développement d'IHM évoluées. Son but est de proposer un support pour les interfaces graphiques sous la forme d'un sous-ensemble assez complet de l'API AWT et des applets.

Le Personal Profile repose sur le Personal Basis Profile.

Package	Description
java.applet	Classes permettant la création d'applets
java.awt	Composants graphiques
java.awt.datatransfer	Classes et interfaces pour l'échange de données

La version 1.0 du Personal Profile est spécifiée par la JSR 62.

La version 1.1 du Personal Profile est spécifiée par la JSR 216.

La version 1.1 du Personal Profile est essentiellement une adaptation de son API sur celle de J2SE 1.4

La suite de ce chapitre sera développée dans une version future de ce document

130. Les autres technologies pour les applications mobiles

Chapitre 130

Niveau : 🟩🟩🟨🟨⬜⬜ Supérieur

Technologies legacy
Ce chapitre est conservé pour des raisons historiques

La suite de ce chapitre sera développée dans une version future de ce document

Ce chapitre contient plusieurs sections :

- KJava
- PDAP (PDA Profile)
- PersonalJava
- Embedded Java

130.1. KJava

Ce n'était pas un profil officiel mais un projet proposé par Sun pour réaliser des développements sur des machines de type Palm.

KJava n'est plus supporté. Il faut utiliser MIDP for Palm O.S. à la place.

130.2. PDAP (PDA Profile)

Ce profile permet le développement d'applications sur PDA en tenant compte notamment de l'accès aux données. Il utilise la configuration CLDC.

Il propose deux packages optionnels :

- Personal Information Management (PIM) : pour standardiser l'accès aux données personnelles stockées dans la plupart des PDA tels que le carnet d'adresse, l'agenda, le bloc-notes, ...
- File Connection (FC) : pour permettre l'accès aux données stockées dans un système de fichiers externe tel que les cartes mémoires

Les spécifications de ce profile sont en cours de développement sous la JSR 075 : https://jcp.org/en/jsr/detail?id=075 (PDA Optional Packages for the J2ME Platform)

130.3. PersonalJava

La dernière version de ses spécifications est la 1.2.

PersonalJava était composé de packages obligatoires et facultatifs.

Sun proposait un outil pour émuler un environnement d'exécution pour des applications développer avec PersonalJava : PJEE (PersonalJava Emulation Environment).

Ce profile a été abandonné au profit d'un ensemble de profils qui respecte mieux le découpage des rôles de J2ME : CDC, Foundation profile et Personal Profile.

130.4. Embedded Java

Cette technologie n'est plus supportée par Sun qui propose en remplacement CLDC et MIDP de la plate-forme J2ME/Java ME.

Partie 20 : Annexes

Partie 20 :

Annexes

Annexe A : GNU Free Documentation License

Version 1.1, March 2000

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND définiTIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative définition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original

English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Annexe B : Glossaire

A B C D E F
G H I J K L
M N O P Q R
S T U V W X
Y Z

A

API (Application Programming Interface)	Une API est une bibliothèque qui regroupe des fonctions sous forme de classes pouvant être utilisées pour développer.
Applet	Une petite application Java compilée, incluse dans une page html, qui est chargée par un navigateur et qui est exécutée sous le contrôle de celui-ci. Pour des raisons de sécurité, par défaut, les applets ont des possibilités très resteintes.
AWT (Abstract Window Toolkit)	Une bibliothèque qui regroupe des classes pour développer des interfaces graphiques. Ces composants sont dits "lourds" car ils utilisent les composants du système sur lequel ils s'executent. Ainsi, le nombre des composants est volontairement resteint pour ne conserver que les composants présents sur tous les systèmes.

B

BDK (Beans Development Kit)	Un outil fourni par Sun qui permet d'assembler des beans de façon graphique pour générer des applications.
Bean	Un composant réutilisable. Il possède souvent une interface graphique mais pas obligatoirement.

BluePrints	Ce sont des documents proposés par Sun pour faciliter le développement avec Java (exemple de code, conseils, design patterns, FAQ, ...)
BMP (Bean Managed Persistence)	Type dEJB entité dont la persistance est à la charge du code qu'il contient
Bytecode	Un programme source Java est compilé en bytecode. C'est un langage machine indépendant du processeur. Le bytecode est ensuite traduit par la machine virtuelle en langage machine compréhensible par le système où il s'execute. Ceci permet de rendre java indépendant de tout système.

C

CLASSPATH	Variable d'environement qui recense les répertoires contenant des bibliothèques utilisables pour la compilation et l'exécution du code.
CLDC (Connected Limited Device Configuration)	Configuration J2ME pour des appareils possédant de faibles ressources et une interface utilisateur réduite tels que des téléphones mobiles, des pagers, des PDA, etc ...
CDC (Connected Device Configuration)	Configuration J2ME pour des appareils embarqués possédant certaines ressources et une connexion à internet tels que des set top box, certains PDA haut de gamme, des systèmes de navigations pour voiture, etc ...
CMP (Container Managed Persistence)	Type d'EJB entité dont la persistance est assurée par le conteneur
CORBA (Common Object Request Broker Architecture)	Modèle d'objets distribués indépendant du langage de développement des objets dont les spécifications sont fournies par l'OMG.
Core class	Classe standard qui est disponible sur tous les systèmes où tourne Java.
Core packages	Ensemble des packages qui composent les API de la plate-forme Java.

D

DAO (Data Acces Object)	
Deprecated	Terme anglais qui peut être attribué à une classe, une interface, un constructeur, une méthode ou un attribut lorque celle-ci ne doivent plus être utilisés car Sun ne garantit pas que cet élément sera encore présent dans les prochaines versions de l'API.
DOM (Document Object Model)	Spécification et API pour représenter et parcourir un document XML sous la forme d'un arbre en mémoire
DTD (Document Type Definition)	Décrit le modèle d'une document XML ppour permettre sa validation

E

EAR (Enterprise ARchive)	Archive qui contient une application J2EE
EJB (Entreprise Java Bean)	Les EJB sont des composants métiers qui répondent à des spécifications précises. Il existe deux types d'EJB : EJB Entity qui s'occupe de la persistence des données et EJB session qui gère les traitements. Les EJB doivent s'executer sur un serveur dans un conteneur d'EJB.

Exception	Mécanisme qui permet de gérer les anomalies et les erreurs détectées dans une application en facilitant leur détection et leur traitement. Les exceptions sont largement utilisées et intégrées dans le langage Java pour accroître la sécurité du code.

F

G

Garbage Collector (Ramasse miettes)	Mécanisme intégré à la machine virtuelle qui récupère automatiquement la mémoire inutilisée en restituant les zones de mémoire laissées libres suite à la destruction des objets.

H

HotJava	Navigateur web de Sun écrit en Java
HTML (HyperText Markup Language)	Langage à base de balises pour formatter une page web affichée dans un navigateur

I

IDL (Interface Définition Language)	Langage qui permet de définir des objets devant être utilisés avec CORBA
IIOP (Internet Iter Orb Protocole)	Protocole pour faire communiquer des objets CORBA
Interface	Définition de méthodes et de variables de classes que doivent respecter les classes qui l'implémentent. Une classe peut implémenter plusieurs interfaces. La classe doit définir toutes les méthodes des interfaces sinon elle est abstraite.
Introspection	Fonction qui permet d'obtenir dynamiquement les entités (champs et méthodes) qui composent un objet

J

J2EE (Java 2 Entreprise Edition)	Version du JDK qui contient la version standard plus un ensemble de plusieurs API permettant le développement d'applications destinées aux entreprises : EJB, Servlet, JSP, JNDI, JMS, JTA, JTS, ...
J2ME (Java 2 Micro Edition)	Version du JDK qui contient le nécessaire pour développer des applications capables de fonctionner dans des environnements limités tels que les assistants personnels (PDA), les téléphones portables ou les systèmes de navigation embarqués
J2SE (Java 2 Standard Edition)	Version du JDK qui contient le nécessaire pour développer des applications et des applets.
JAAS (Java Authentication and Authorization Service)	API qui permet d'authentifier un utilisateur et de lui accorder des droits d'accès

JAI (Java Advanced Imaging)	API dédiée à l'utilisation et à la transformation d'images
JAR (Java ARchive)	Technique qui permet d'archiver avec ou sans compression des classes Java et des ressources dans un fichier unique de façon indépendante de toute plate-forme. Ce format supporte aussi la signature électronique.
Java Media API	Regroupement d'API pour le multimédia
Java One	Conférence des développeurs Java périodiquement organisée par Sun
JavaHelp	Système d'aide pour les utilisateurs d'applications entièrement écrites en Java
JavaMail	API pour utiliser la messagerie électronique (e-mail)
Java Web Start	Outil qui permet d'utiliser une application cliente par téléchargement automatique via le réseau
Java XML Pack	Regroupe des API pour l'utilisation de XML avec Java
JAXB (Java API for XML Binding)	API pour faciliter la persistence entre objets Java et document XML
JAXM (Java API for XML Messaging)	API pour échanger des messages XML notamment avec les services web
JAXP (Java API for XML Processing)	API pour parcourir un document XML (DOM et SAX) et le transformer avec XSLT
JAXR (Java API for XML Registries)	API pour utiliser les services d'annuaires avec les services web (UDDI)
JAX-RPC (Java API for XML Remote Procedure Calls)	API pour utiliser l'appel de méthodes distantes via SOAP
JCA (Java Connector Architecture)	Spécification pour normaliser le développement de connecteurs vers des progiciels
JCP (Java Community Process)	Processus utilisé par Sun et de nombreux partenaires pour gèrer les évolutions de Java et de ses API
JDBC (Java Data Base Connectivity)	API qui permet un accès à des bases de données tout en restant indépendante de celles-ci. Un driver spécifique à la base utilisée permet d'assurer cette indépendance car le code Java reste le même.
JDC (Java Delevopper Connection)	Service en ligne proposé gratuitement par Sun. Après enregistrement, il propose de nombreuses ressources sur Java (tutorial, cours, information, mailing ...).
JDO (Java Data Objects)	API et spécification pour faciliter le mapping entre objet Java et une source de données
JDK (Java Development Kit)	Environnement de développement Java.Il existe plusieurs versions majeures : 1.0, 1.1, 1.2 (aussi appelée Java 2) et 1.3. Tous les outils fournis sont à utiliser sur une ligne de commandes.
JEP (JDK Enhancement Proposal)	
JFC (Java Foundation Class)	Ensemble de classes qui permet de développer des interfaces graphiques plus riches et plus complètes qu'avec AWT
JIT Compiler (Just In Time Compiler)	Compilateur qui, pour améliorer les performances, compile le bytecode à la volée lors de l'exécution des programmes.
JMS (Java Message Service)	API qui permet l'échange de messages asynchrones entre applications en utilisant un MOM (Middleware Oriented Message)
JMX (Java Management eXtension)	API et spécification qui permet de développer un système d'administration d'application à distance via le réseau
JNDI (Java Naming and Directory Interface)	Bibliothèque qui permet un accès aux annuaires de l'entreprise. Plusieurs protocoles sont supportés : LDAP, DNS, NIS et NDS.

JNI (Java Native Interface)	API qui normalise et permet les appels de code natif dans une application java.
JRE (Java Runtime Environment)	L'environnement d'exécution des programmes Java.
JSDK (Java Servlet Development Kit)	Ensemble de deux packages qui permettent le développement des servlets.
JSP (Java Server Page)	Technologie comparable aux ASP de Microsoft mais utilisant Java. C'est une page HTML enrichie de tags JSP et de code Java. Une JSP est traduite en servlet pour être exécutée. Ceci permet de séparer la logique de présentation et la logique de traitement contenue dans un composant serveur tel que des servlets, des EJB ou des beans.
JSR (Java Specification Request)	Démande d'évolution ou d'ajout des API Java traitée par le JCP
JSSE (Java Secure Socket Extension)	API permettant l'utilisation du protocole SSL pour des échanges HTTP sécurisés
JSTL (Java Standard Tag Library)	Bibliothèque de tags JSP standards
JTS (Java Transaction Service)	API pour utiliser les transactions
JTWI	Spécification issue de la JSR 185 visant à définir un environnement d'exécution utilisant CLDC, MIDP et plusieurs profiles de façon homogène
JUG (Java User Group)	Groupe d'utilisateurs Java
JVM (Java Virtual Machine)	C'est la machine virtuelle dans laquelle s'exécute le code Java. C'est une application native dépendante du système d'exploitation sur laquelle elle s'exécute. Elle répond à des normes dictées par Sun pour assurer la portabilité du langage. Il en existe plusieurs développées par différents éditeurs notamment Sun, IBM, Borland, Microsoft, ...
JVMDI (Java Virtual Machine Debugger Interface)	API native utilisée par les débogueurs et d'autres outils pour d'inspecter l'état et contrôler l'exécution des applications dans une JVM.

K

L

Layout Manager (gestionnaire de présentation)	Les layout manager sont des classes qui gèrent la disposition des composants d'une interface graphique sans utiliser des coordonnées.
LDAP	Protocole qui permet d'accéder à un annuaire d'entreprise

M

Message Driven Bean	Type d'EJB qui traite les messages reçus d'un MOM de façon asynchrone
Midlet	Application mobile développée avec CLDC et MIDP
MIDP (Mobile Information Device Profile)	Profil utilisé avec la configuration CLDC pour le développement d'applications mobiles sous la forme de Midlets
MOM (Middleware Oriented Message)	Outil qui permet l'échange de messages entre applications
MVC (Model View Controler)	

	Modèle de conception largement répandu qui permet de séparer l'interface graphique, les traitements et les données manipulées

N

O

ODBC (Open Database Connectivity)	API et spécifications de Microsoft pour l'accès aux bases de données sous Windows
ORB	Middleware orienté objet pour mettre en oeuvre CORBA

P

Package (Paquetage)	Ils permettent de regrouper des classes par critères. Ils impliquent une structuration des classes dans une arborescence correspondant au nom donné au package.
POJI (Plain Old Java Interface)	
POJO (Plain Old Java Object)	

Q

R

Ramasse miette	
RI (Reference Implementation)	Implementation de référence proposée pour une spécification particulière
RMI (Remote Method Invocation)	C'est une technologie développée par Sun qui permet de faire des appels d'objets distants. Cette technologie est plus facile à mettre en oeuvre que Corba mais elle ne peut appeler que des objets java.

S

Sandbox (bac à sable)	Il désigne un ensemble de fonctionnalités et d'objets qui assure la sécurité des applications. Son composant principal est le gestionnaire de sécurité. Par exemple, il empêche, par défaut, une applet d'accéder aux ressources du système.
SAX (Simple API for XML)	API pour traiter séquentiellement un document XML en utilisant des événements
Serialization	Fonction qui permet à un objet d'envoyer son état dans un flux pour permettre sa persistance ou son envoi à travers un reseau par exemple
Servlet	C'est un composant Java qui s'exécute côté serveur dans un environnement dédié pour répondre à des requêtes. L'usage le plus fréquent est la génération dynamique de page Web. On les compare

	souvent aux applets qui s'éxecutent côté client mais elles n'ont pas d'interface graphique.
SOAP (Simple Object Access Protocol)	Protocole des services web pour l'échange de messages et l'appel de méthodes distances grace à XML
SQL/J	spécification qui permet d'imbriquer du code SQL dans du code Java
Swing	Framework pour le développement d'interfaces graphiques composé de composants légers

T

Taglibs	Bibliothèques de tags personnalisés utilisés dans les JSP

U

V

W

WAR (Web ARchive)	Archive qui contient une application web
webapp (web application)	Application web reposant sur les servlets et les JSP

X

Y

Z